Lecture 19:
Data Storage and Indexes

Wednesday, November 15, 2006

Outline

- Representing data elements (12)
- Index structures (13.1, 13.2)
- B-trees (13.3)

Files and Tables

- A disk = a sequence of blocks
- A file = a subsequence of blocks, usually contiguous
- Need to store tables/records/indexes in files/block

Representing Data Elements

- Relational database elements:

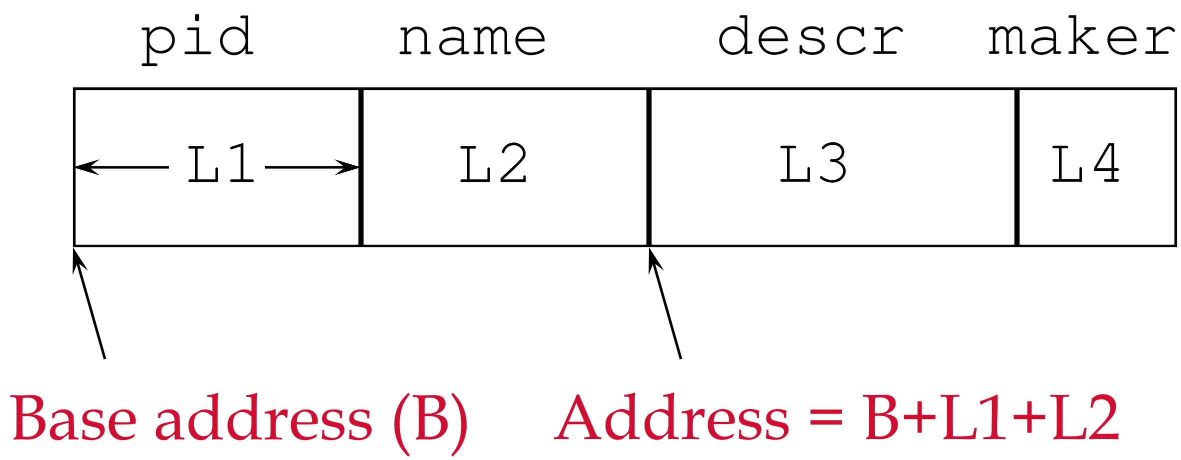
```
CREATE TABLE Product (

    pid INT PRIMARY KEY,
    name CHAR(20),
    description VARCHAR(200),
    maker CHAR(10) REFERENCES Company(name)
)
```

- A tuple is represented as a record
- The table is a sequence of records

Issues

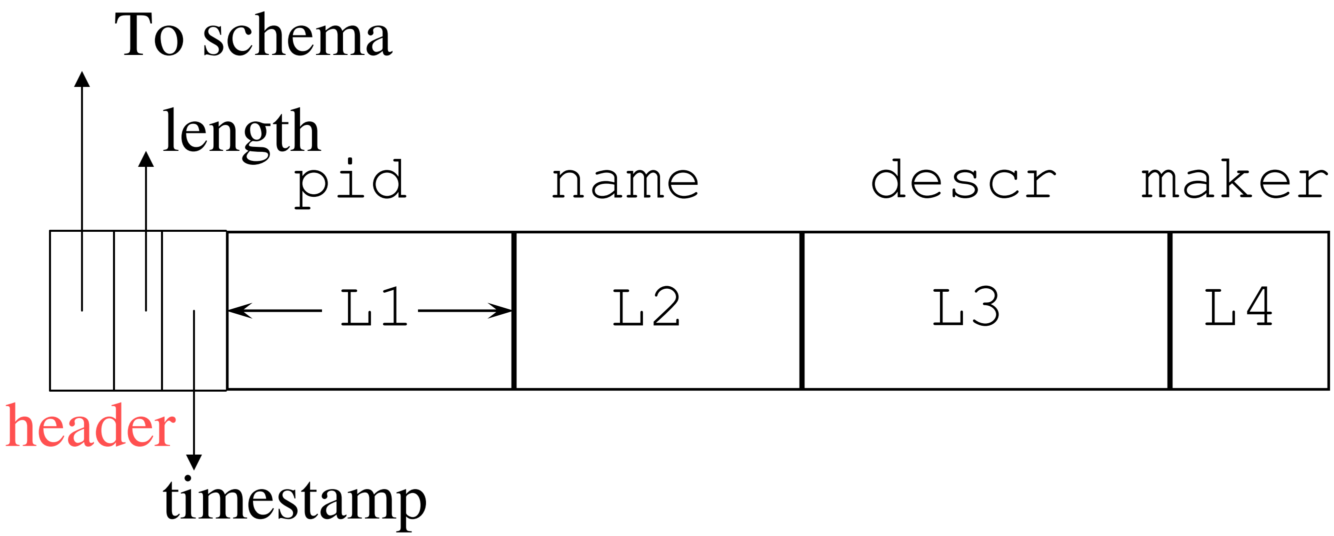
- Represent attributes inside the records
- Represent the records inside the blocs

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!**

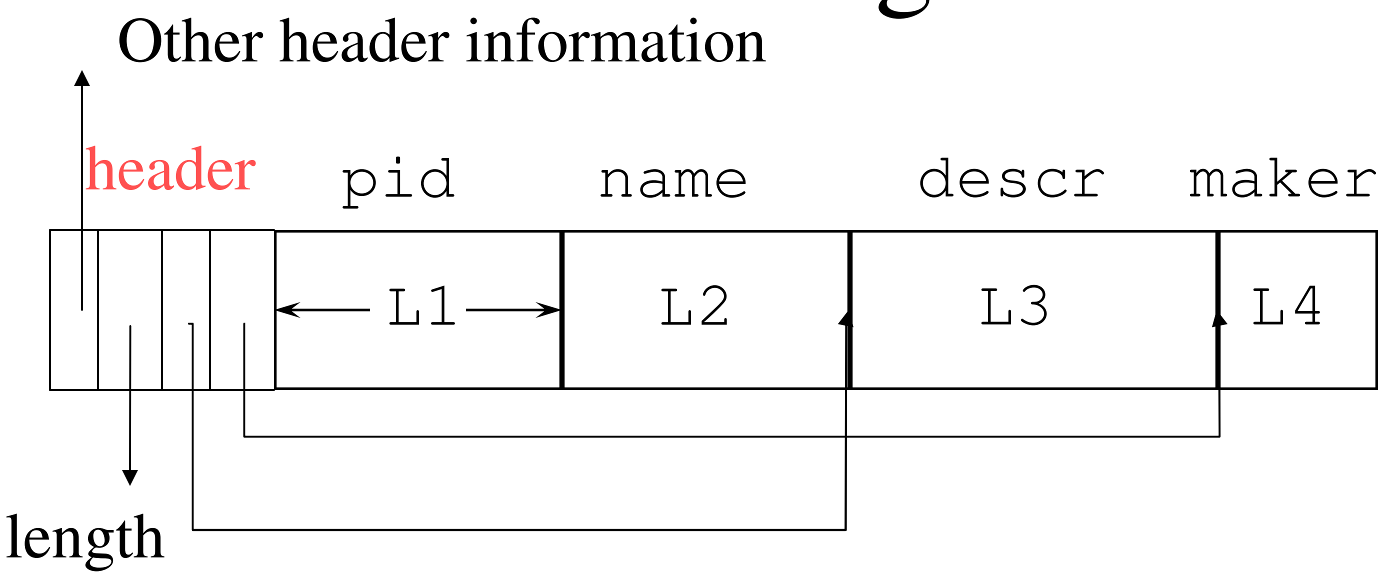
Record Header



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

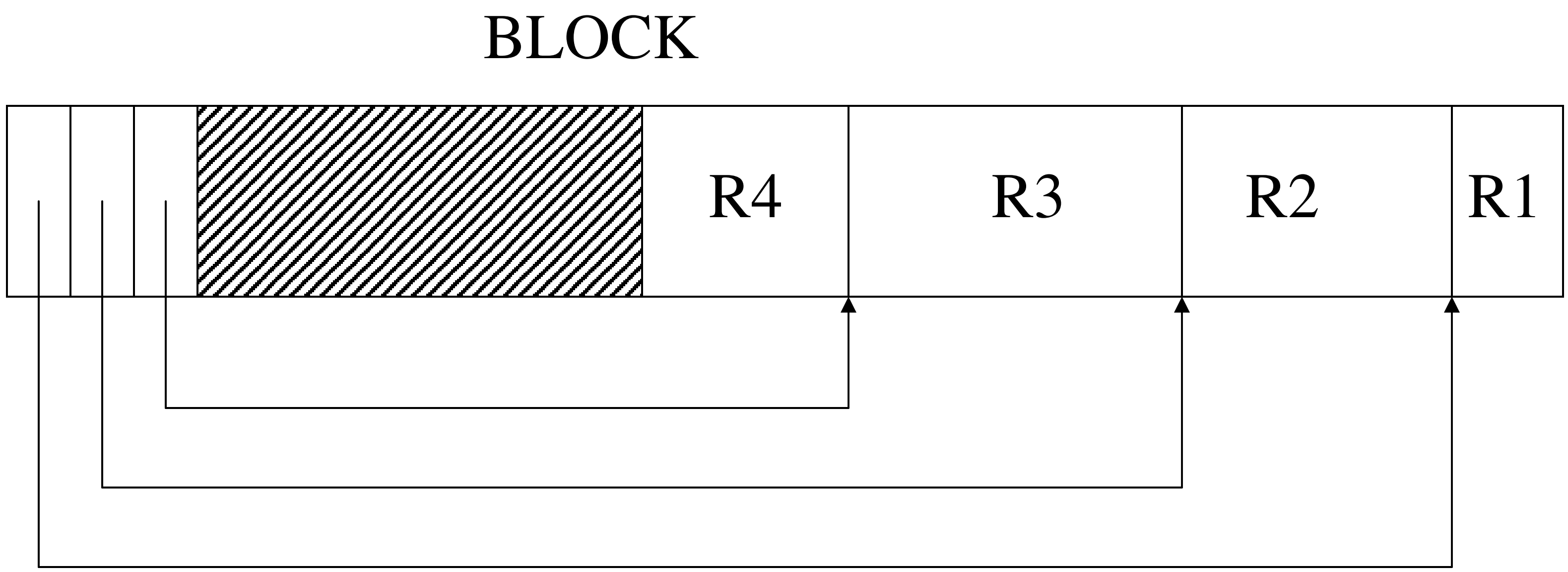
Then the variable length fields: F2, F3, F4

Null values take 2 bytes only

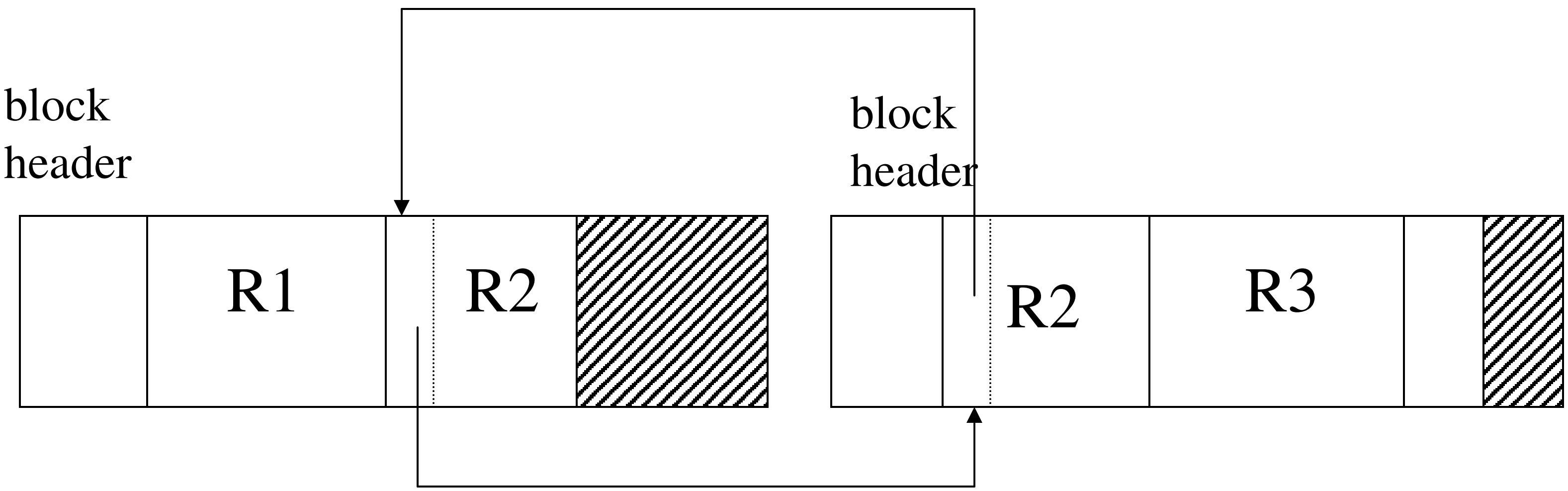
Sometimes they take 0 bytes (when at the end)

Storing Records in Blocks

- Blocks have fixed size (typically 4k – 8k)



Spanning Records Across Blocks



- When records are very large
- Or even medium size: saves space in blocks

BLOB

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

CLOB = character large objec

- Supports only restricted operations

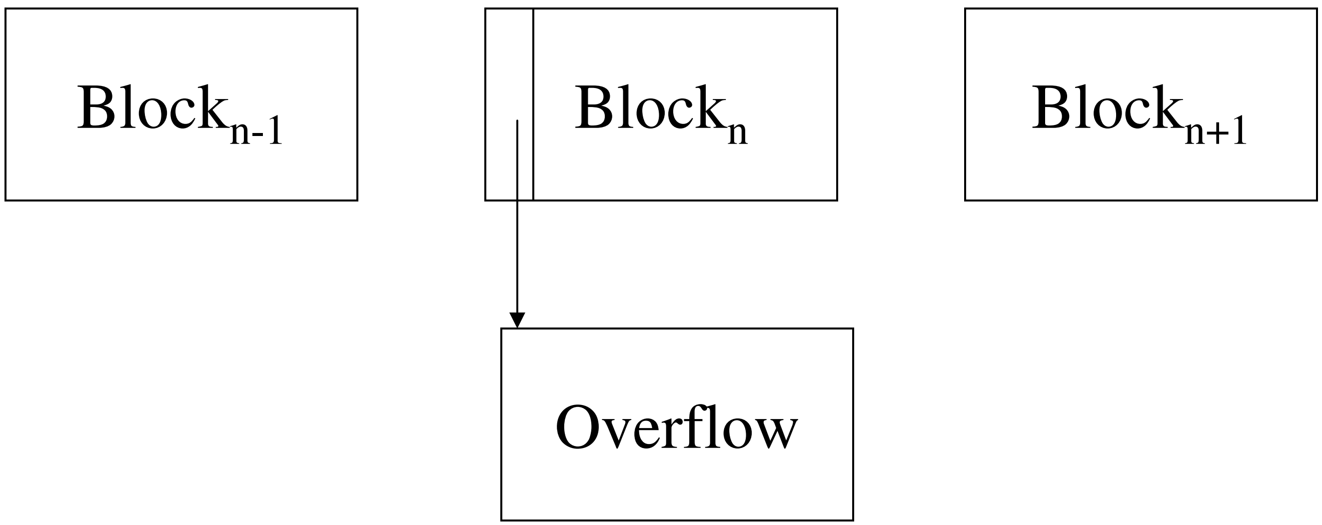
File Types

- Unsorted (heap)
- Sorted (e.g. by pid)

Modifications: Insertion

- File is unsorted: 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 <u>*overflow block*</u>

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)

How can we *point* to a record in an RDBMS ?

Modifications: Updates

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

Pointers

Where do we need them in RDBMS ?

Logical pointer to a record consists of:

- Logical block number
- An offset in the block's header

Note: review what a pointer in C is

Indexes

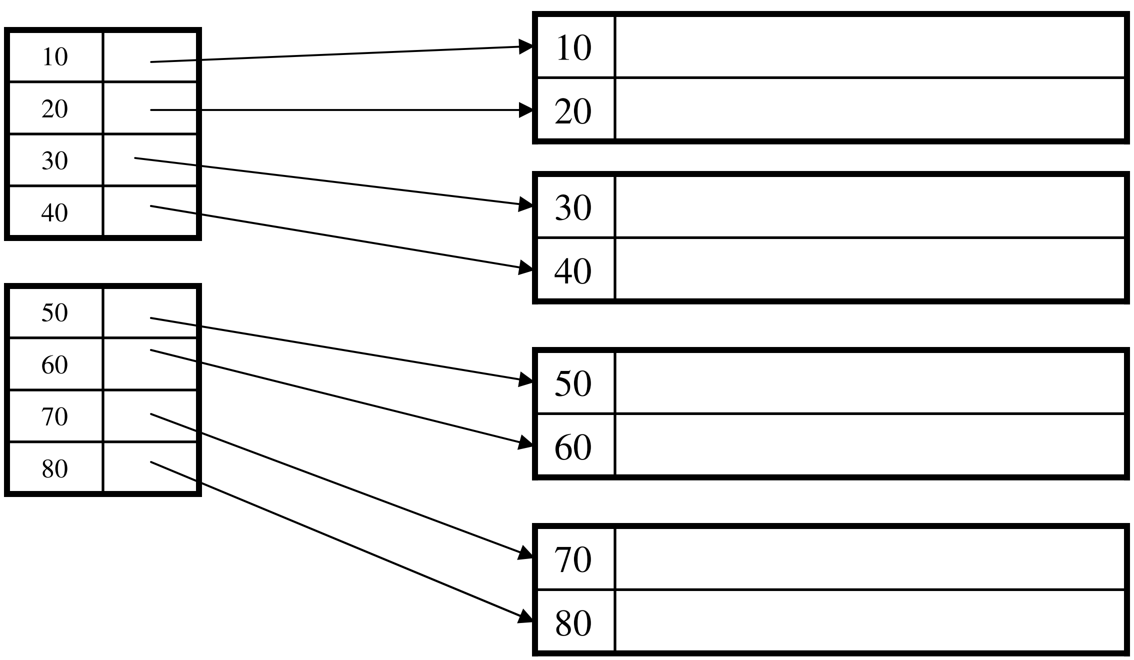
- An *index* on a file speeds up selections on the *search key fields* for the index.
  - Any subset of the fields of a relation can be the search key for an index on the relation.
  - *Search key* is not the same as *key* (minimal set of fields that uniquely identify a record in a relation).
- An index contains a collection of *data entries*, and supports efficient retrieval of all data entries with a given key value **k**.

Index Classification

- Primary/secondary
  - Primary = may reorder data according to index
  - Secondary = cannot reorder data
- Clustered/unclustered
  - Clustered = records close in the index are close in the data
  - Unclustered = records close in the index may be far in the data
- Dense/sparse
  - Dense = every key in the data appears in the index
  - Sparse = the index contains only some keys
- B+ tree / Hash table / …

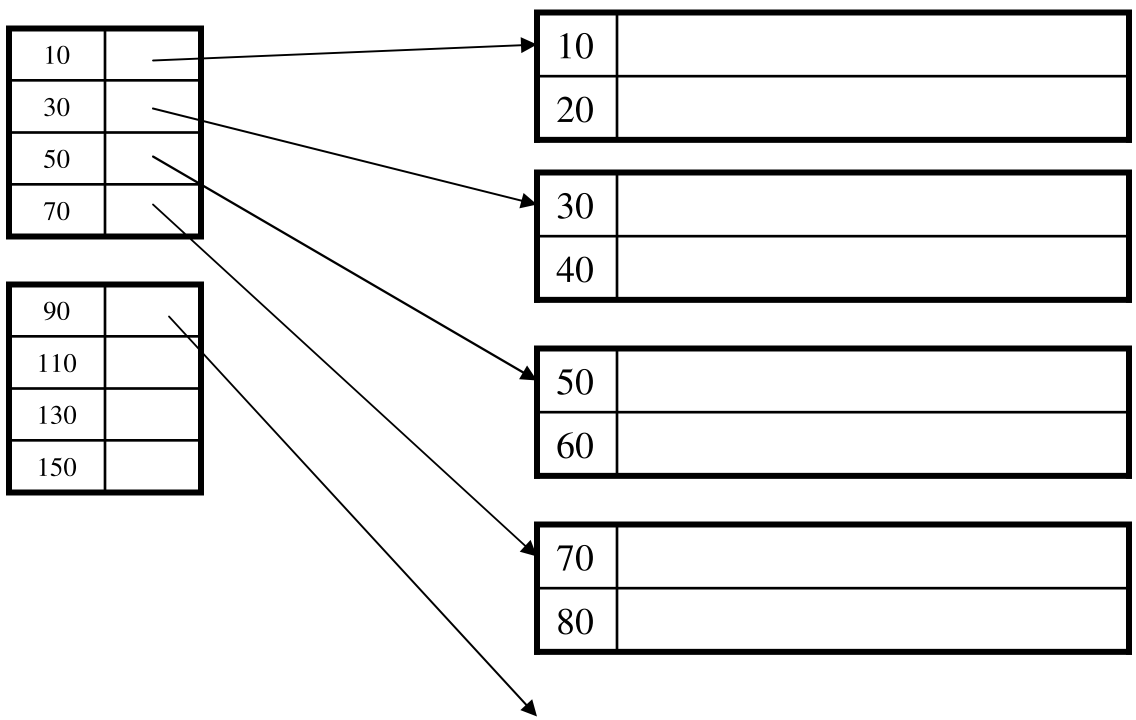
Primary Index

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



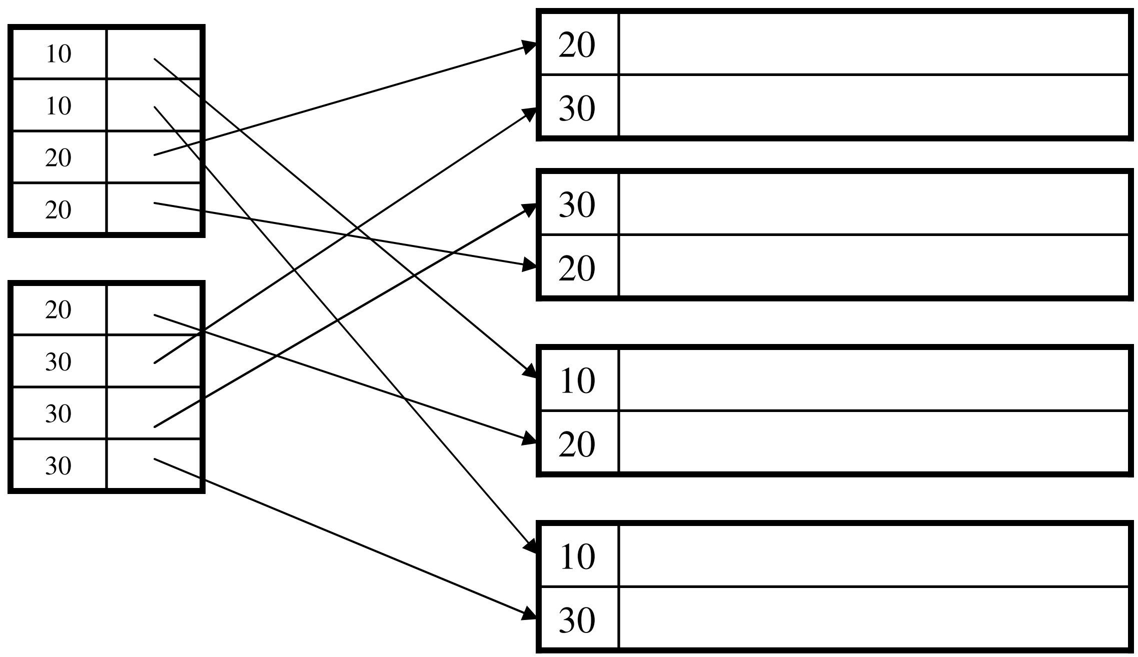
Primary Index

- *Sparse* index



Secondary Indexes

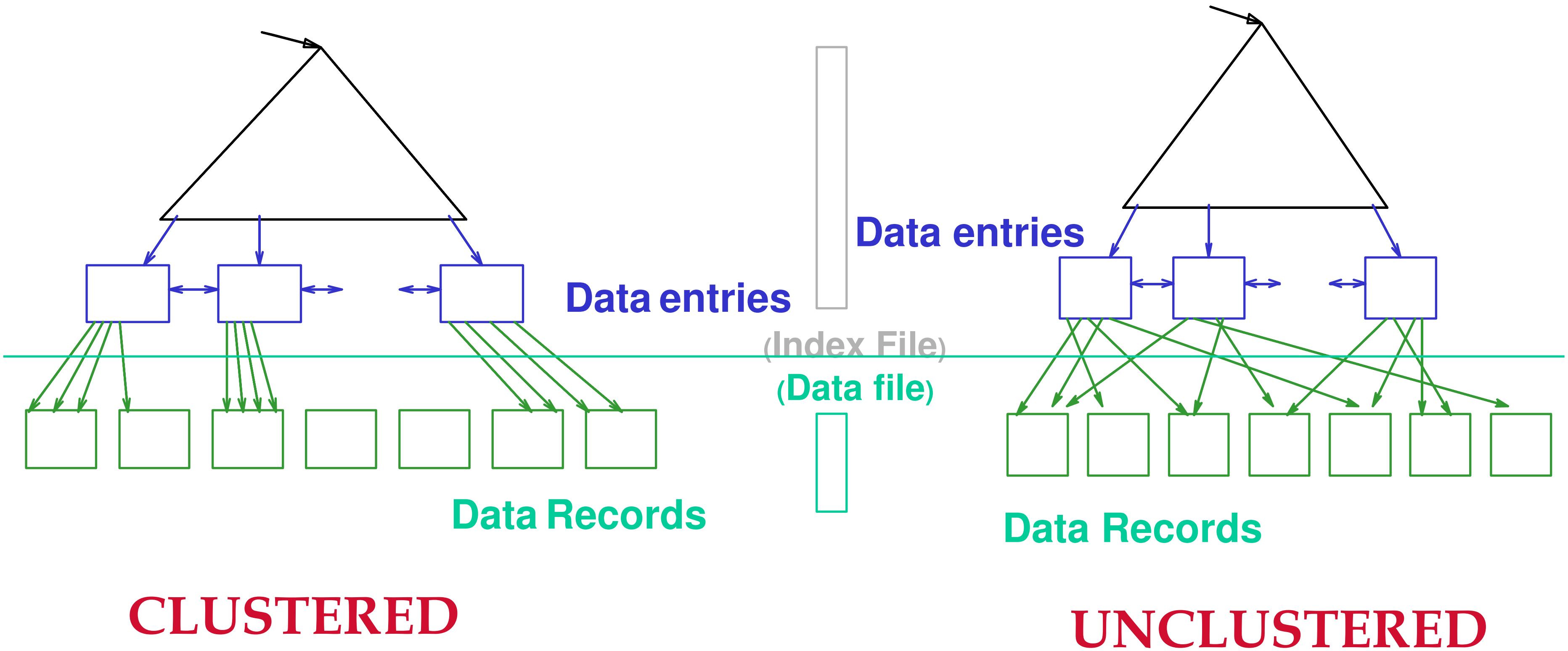
- To index other attributes than primary key
- Always dense (why ?)



Clustered/Unclustered

- Primary indexes = usually clustered
- Secondary indexes = usually unclustered

Clustered vs. Unclustered Index



Secondary Indexes

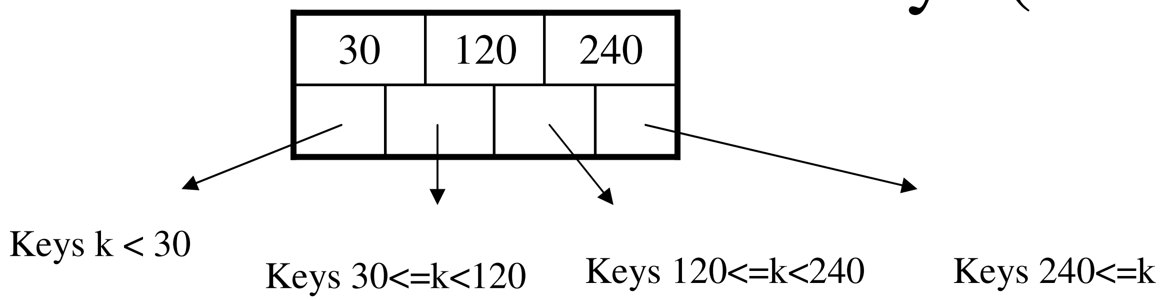
- Applications:
  - index other attributes than primary key
  - index unsorted files (heap files)
  - index clustered data

B+ Trees

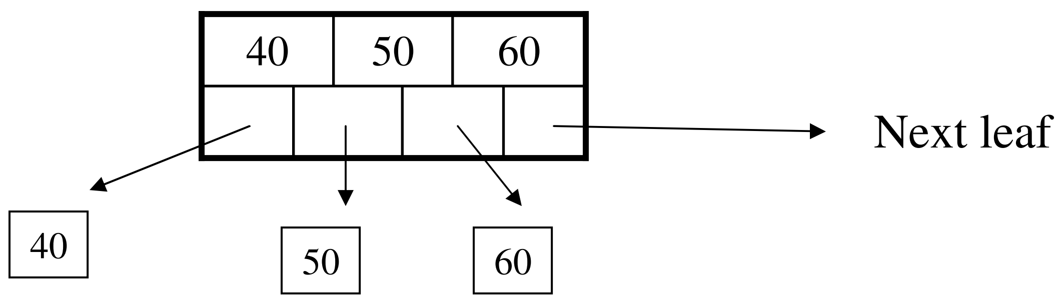
- Search trees
- Idea in B   Trees:
  - make 1 node = 1 block
- Idea in B+ Trees:
  - Make leaves into a linked list (range queries are easier)

B+ Trees Basics

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

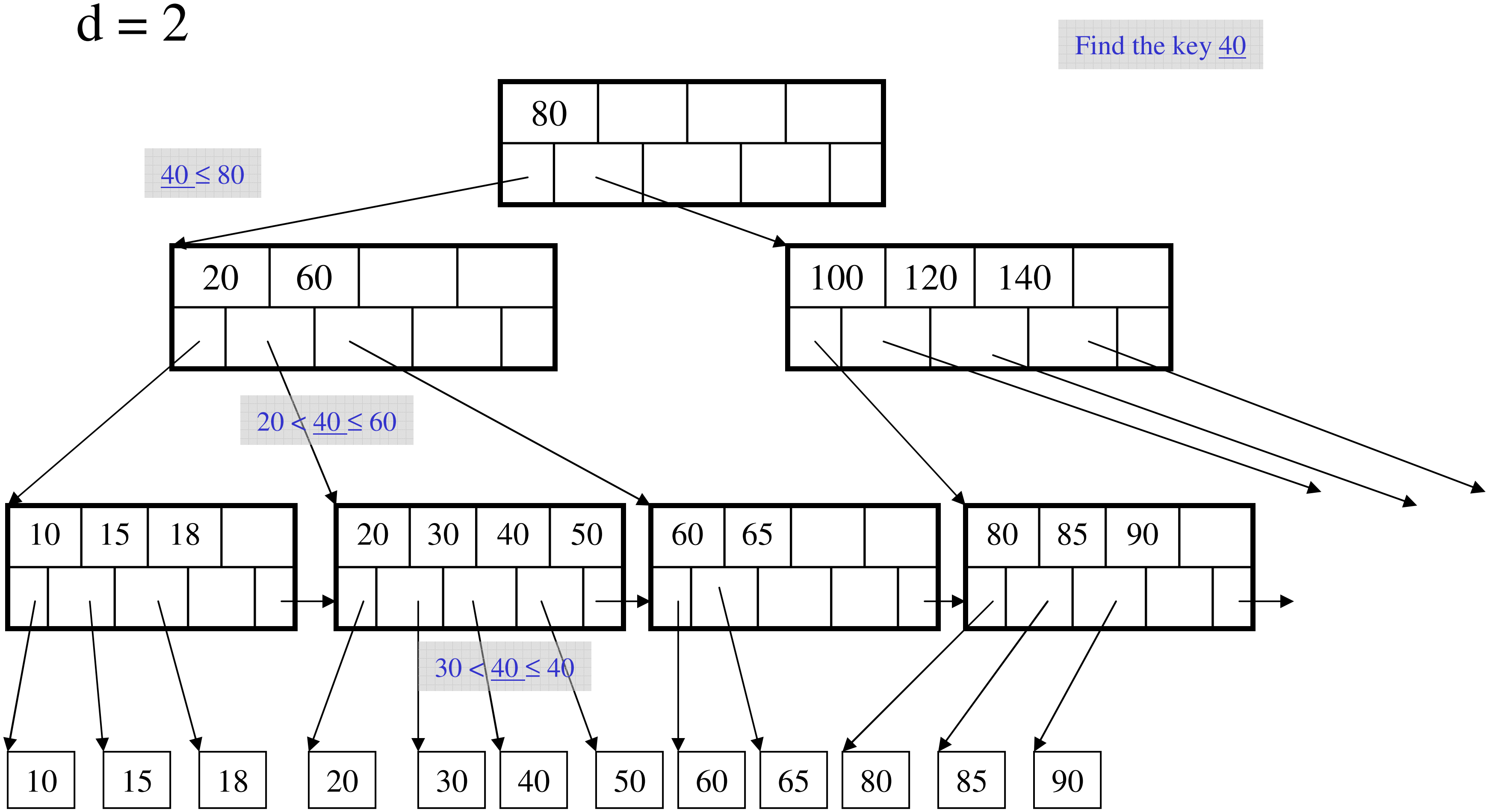


- Each leaf has >=d and <= 2d keys:



B+ Tree Example

d = 2



B+ Tree Design

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

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
```

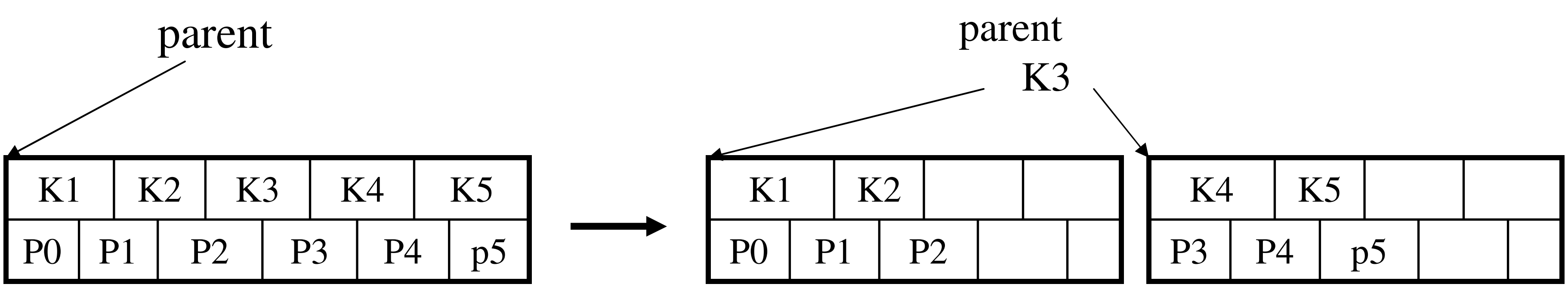
B+ Trees in Practice

- 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)

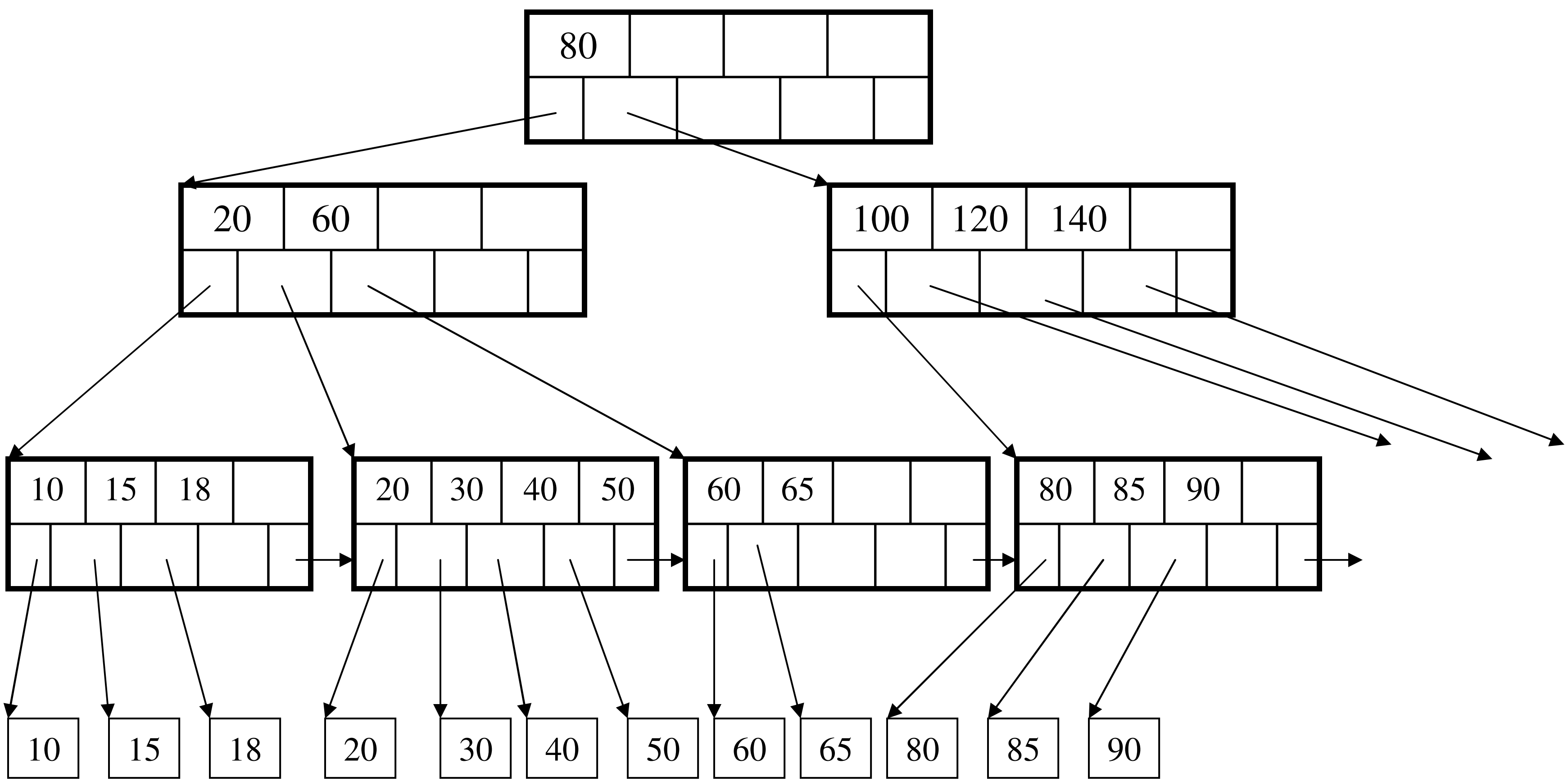
- Find leaf where K belongs, insert
- If no overflow (2d keys or less), halt
- If overflow (2d+1 keys), split node, insert in parent:



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

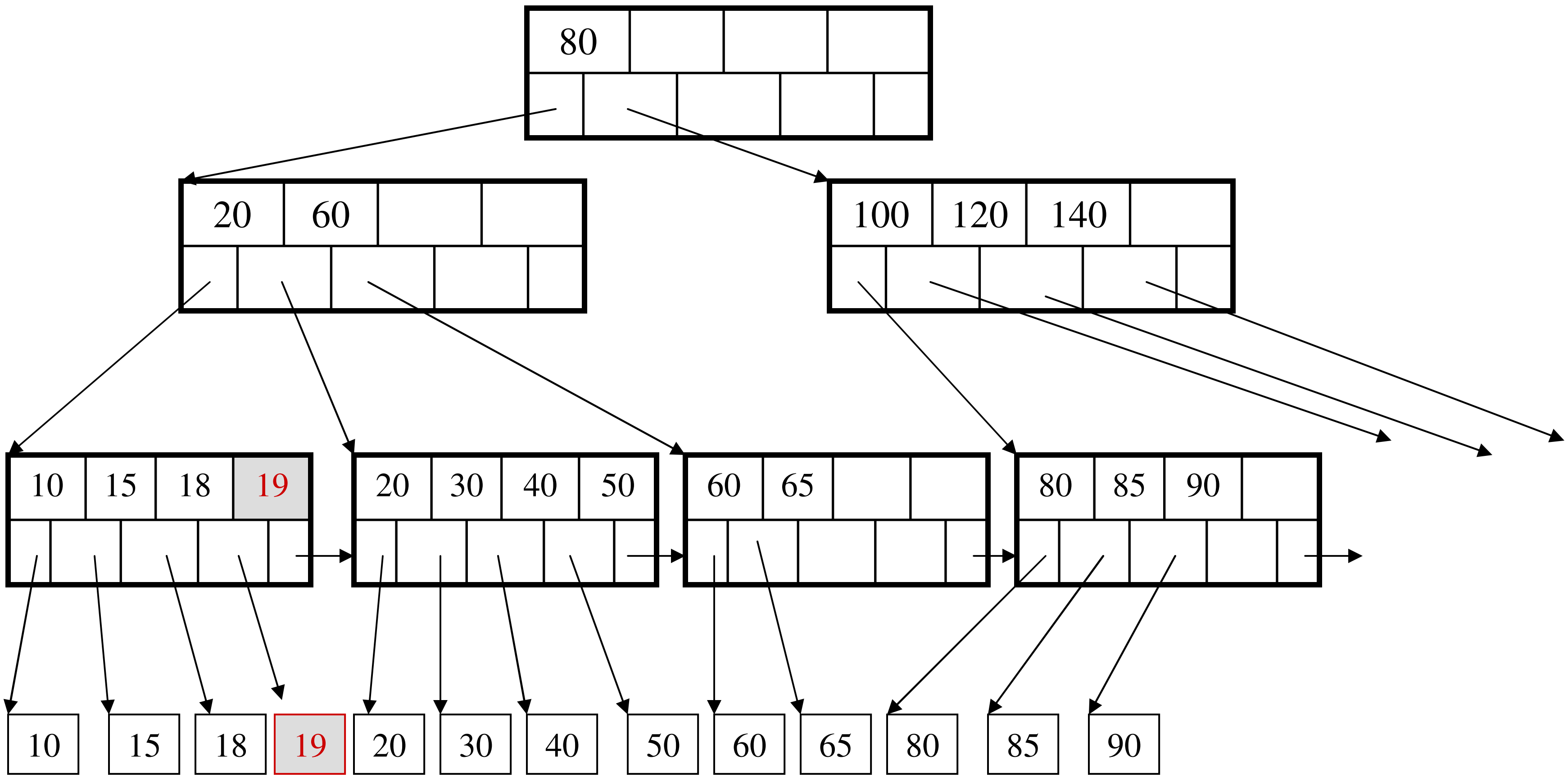
Insertion in a B+ Tree

Insert K=19



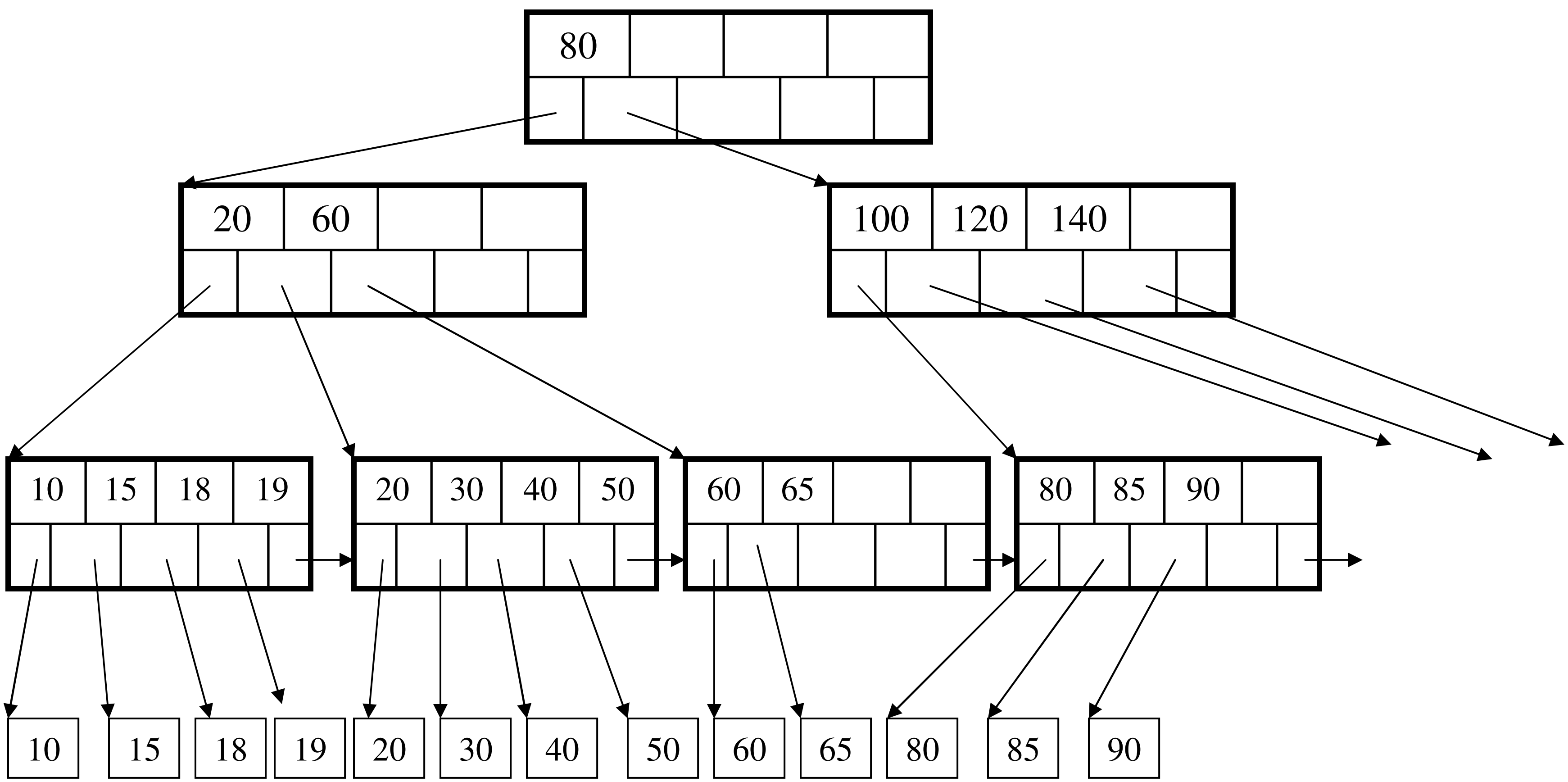
Insertion in a B+ Tree

After insertion



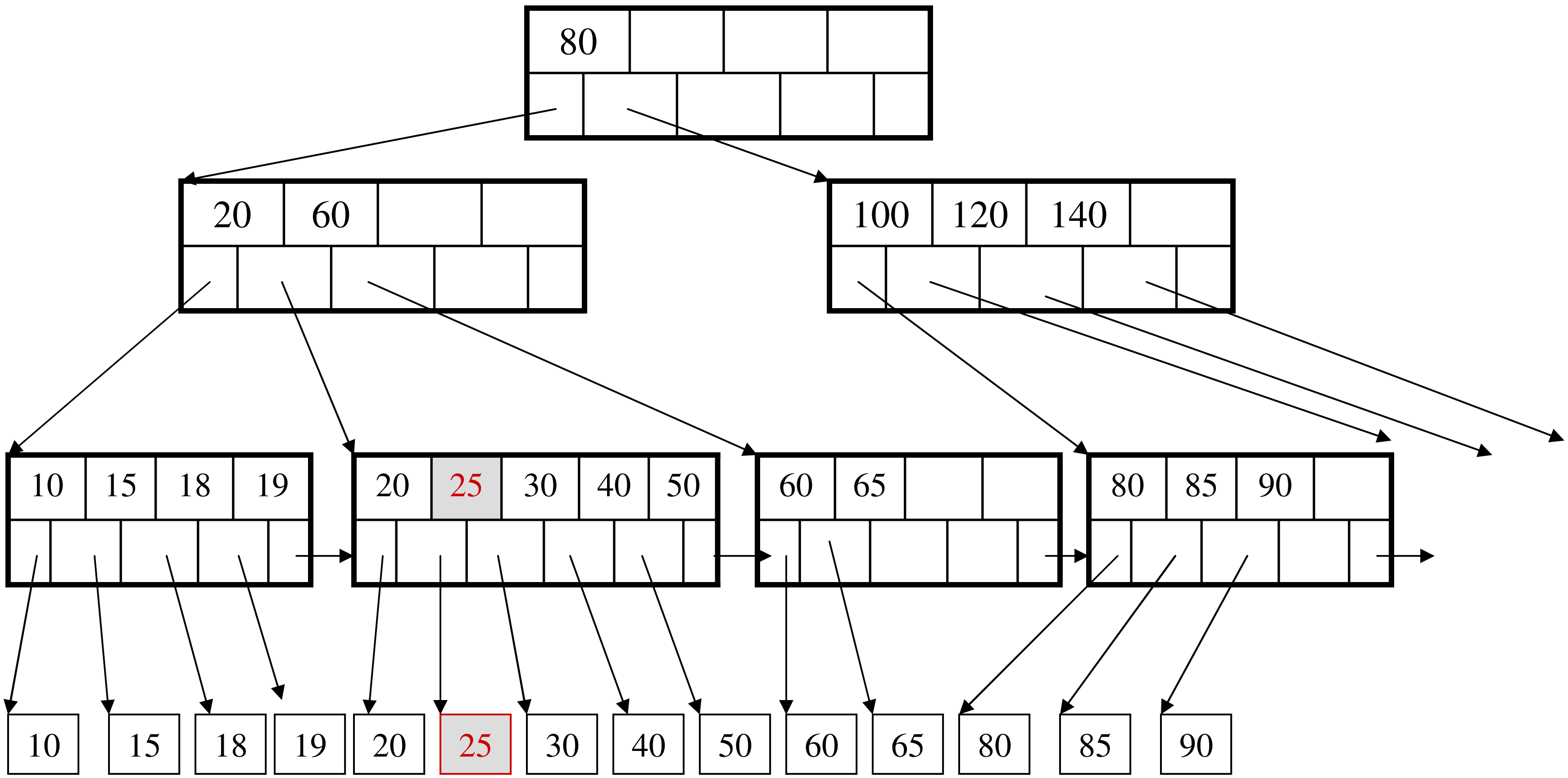
Insertion in a B+ Tree

Now insert 25



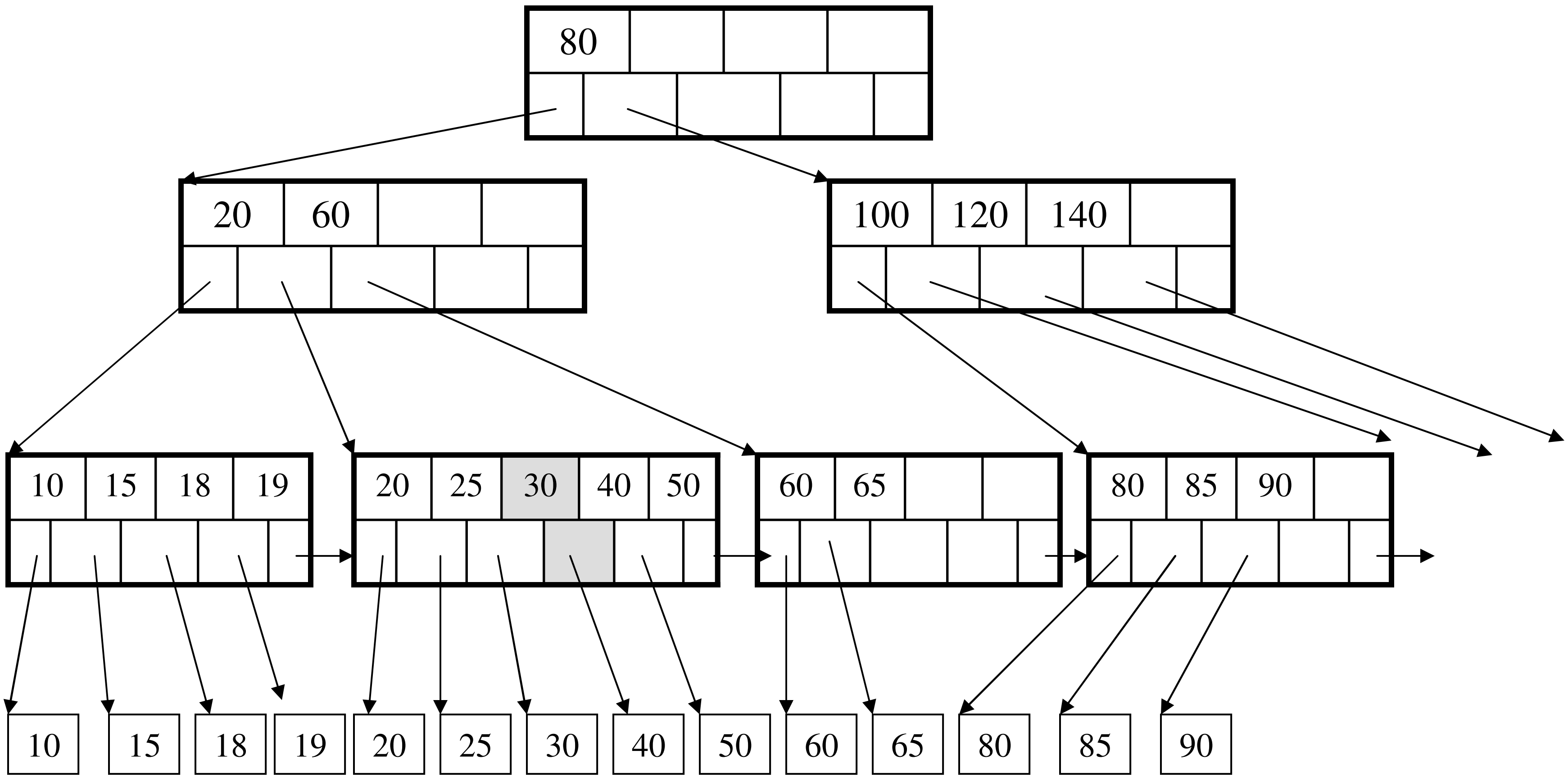
Insertion in a B+ Tree

After insertion



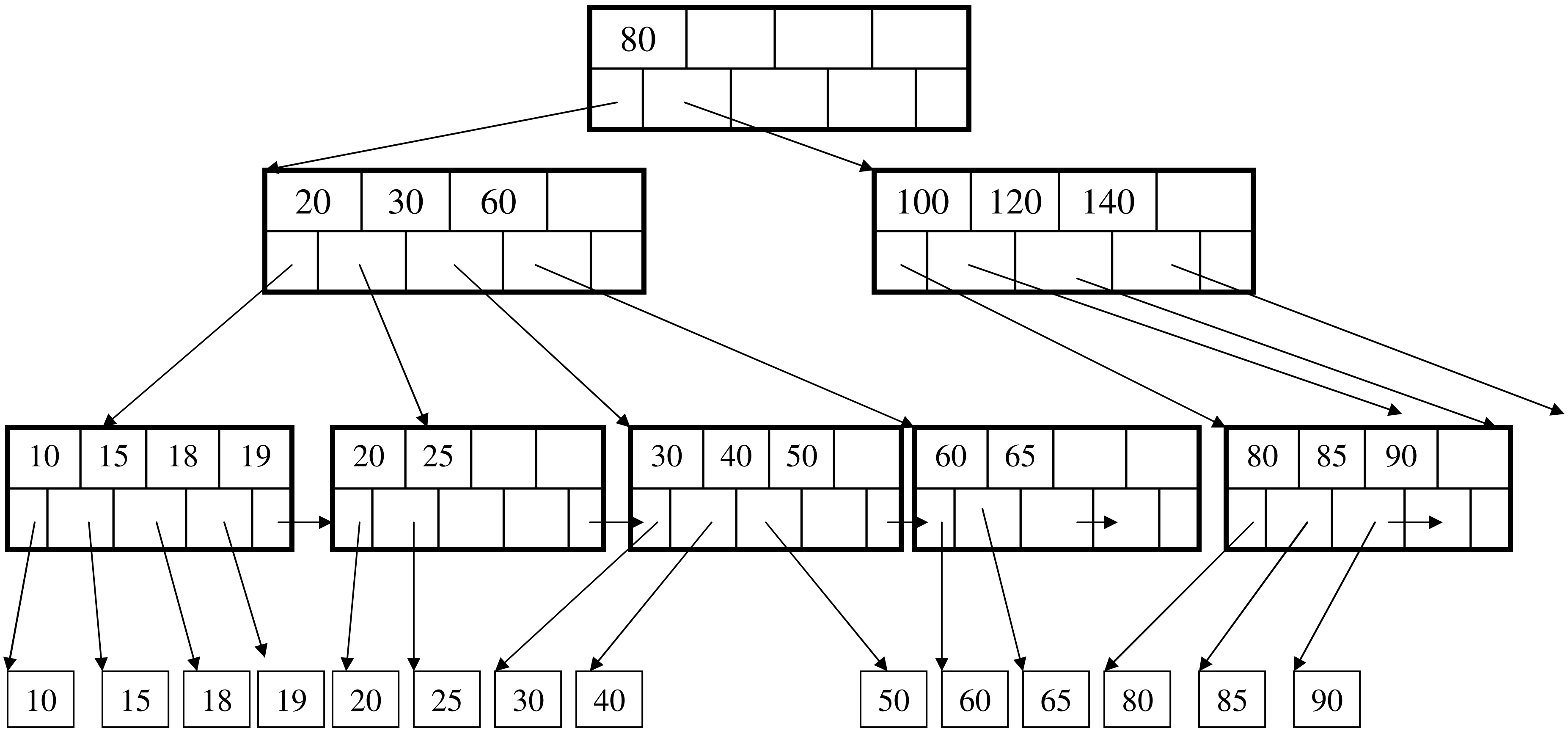
Insertion in a B+ Tree

But now have to split !



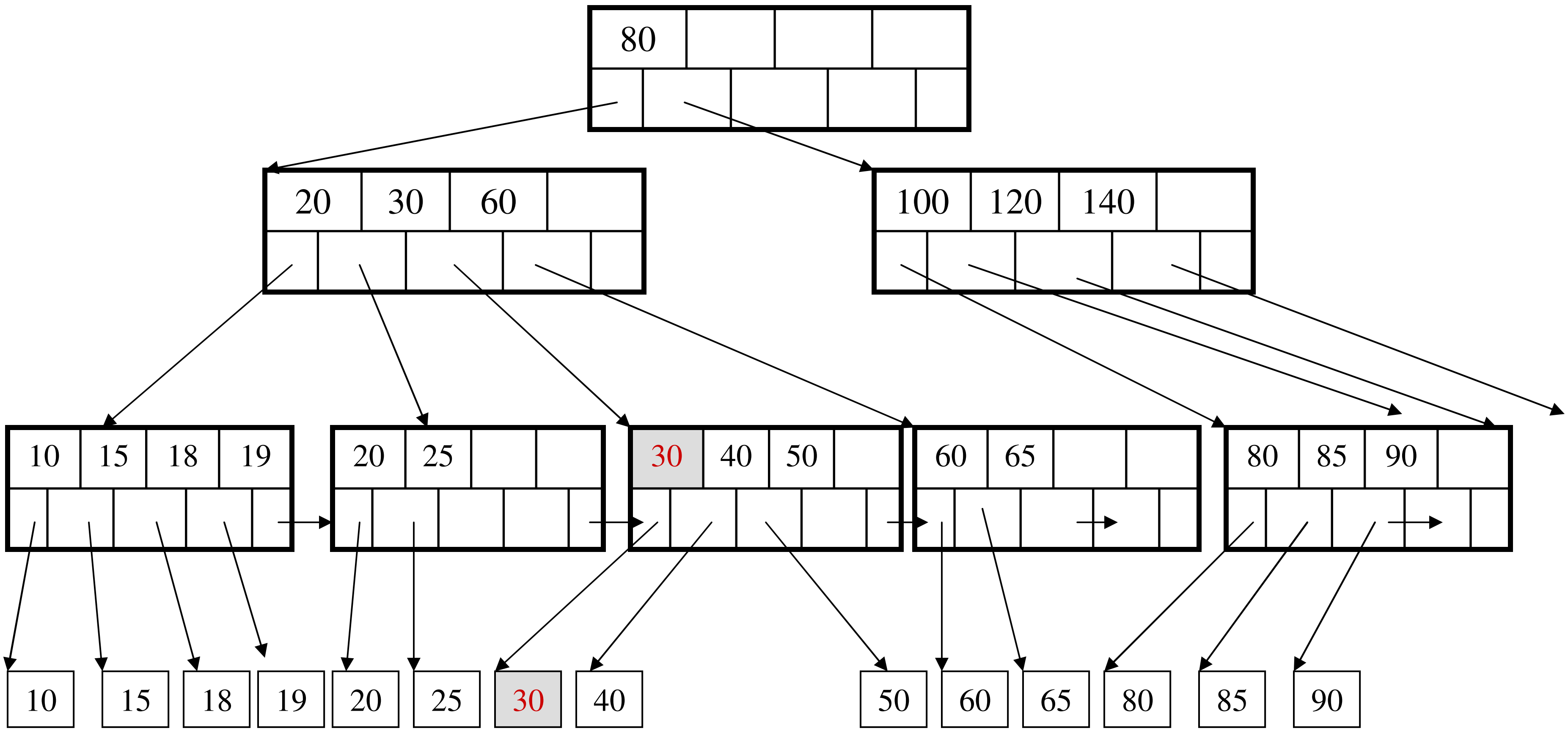
Insertion in a B+ Tree

After the split



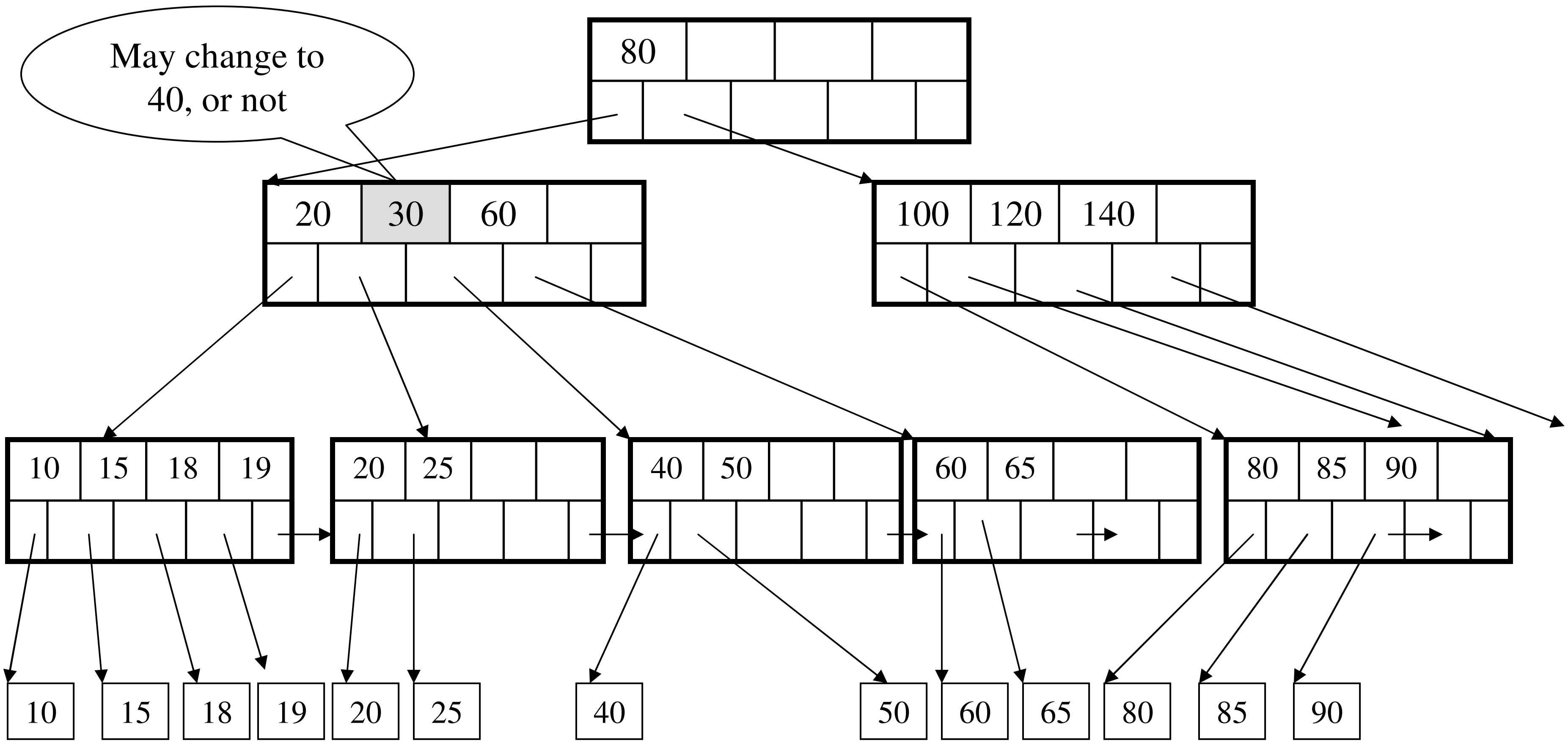
Deletion from a B+ Tree

Delete 30



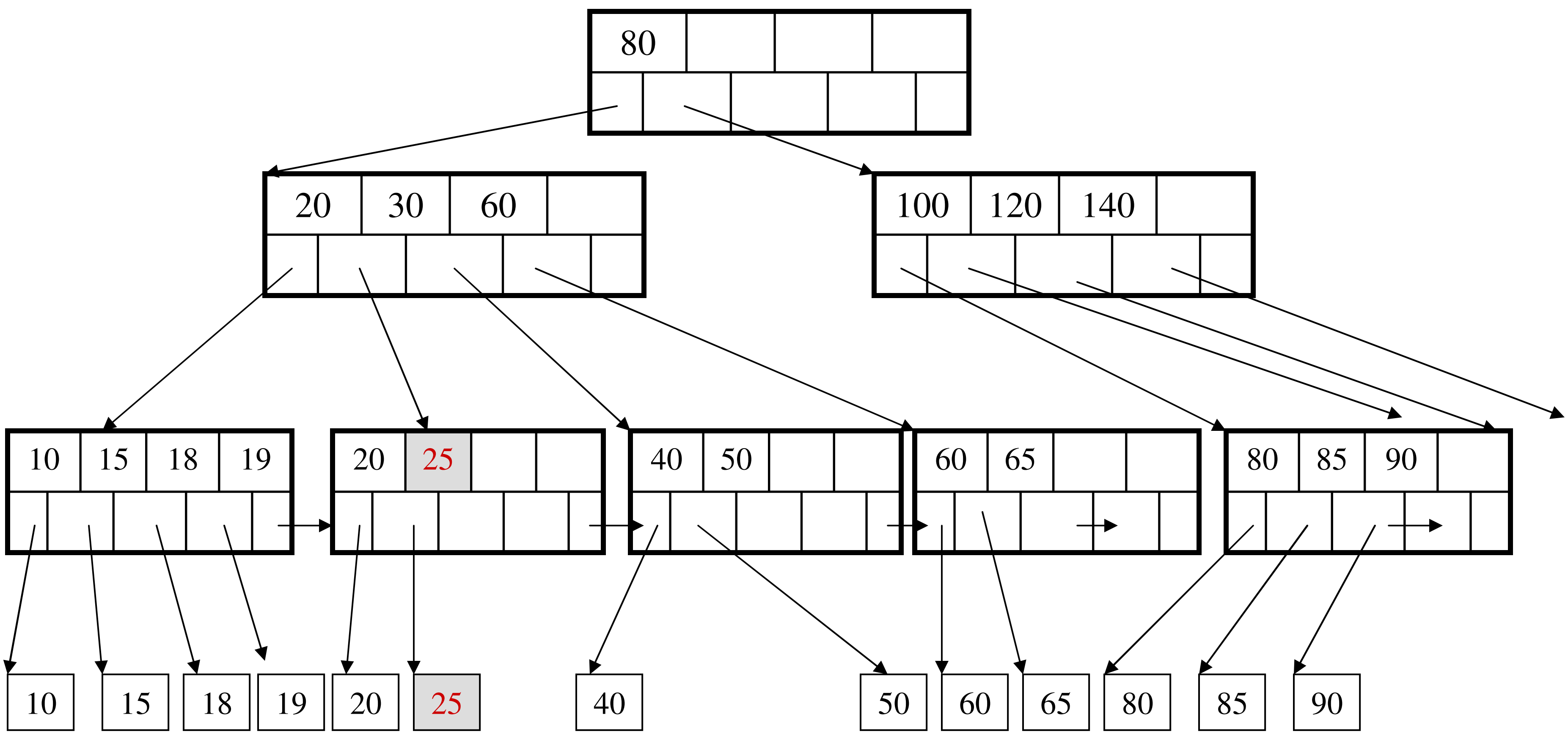
Deletion from a B+ Tree

After deleting 30



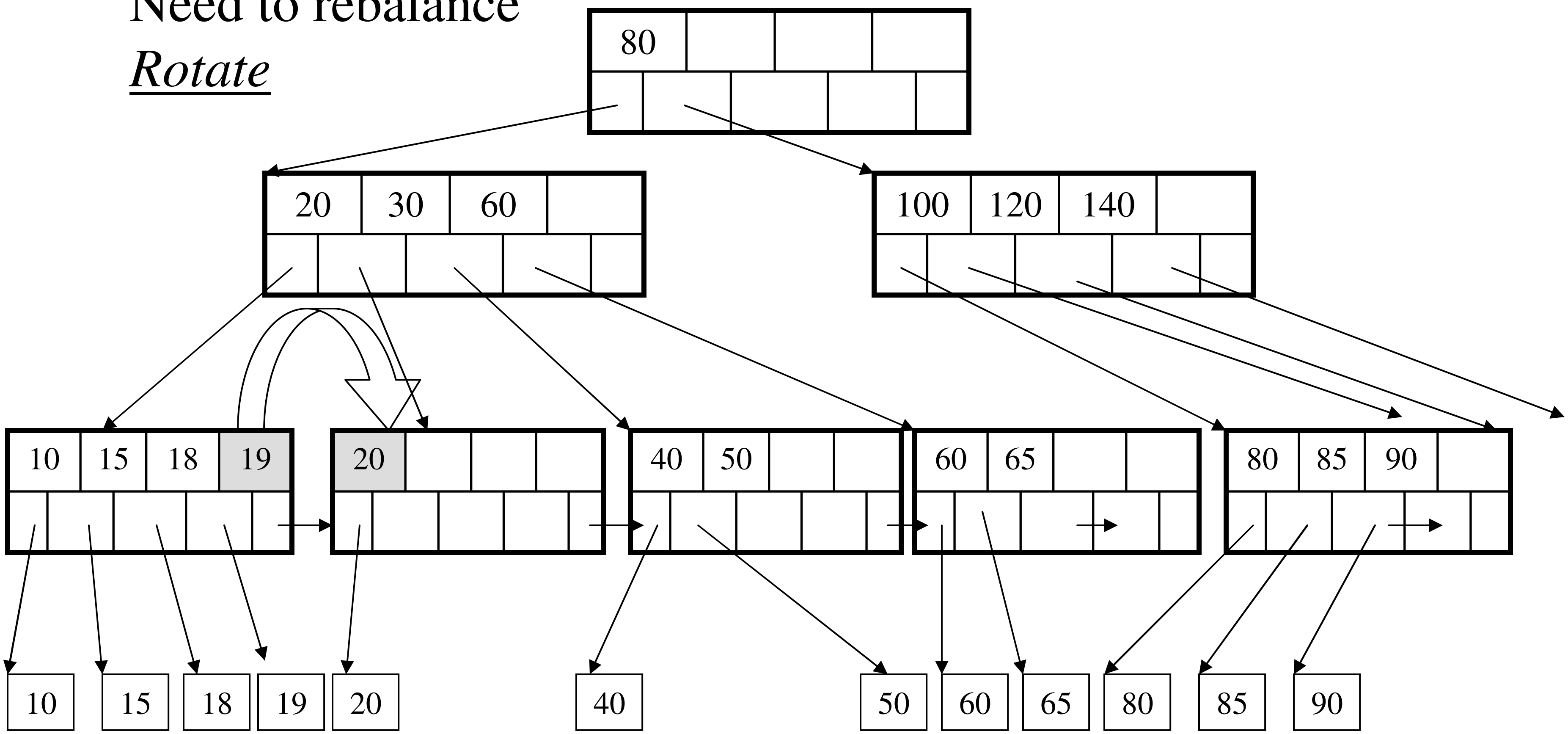
Deletion from a B+ Tree

Now delete 25



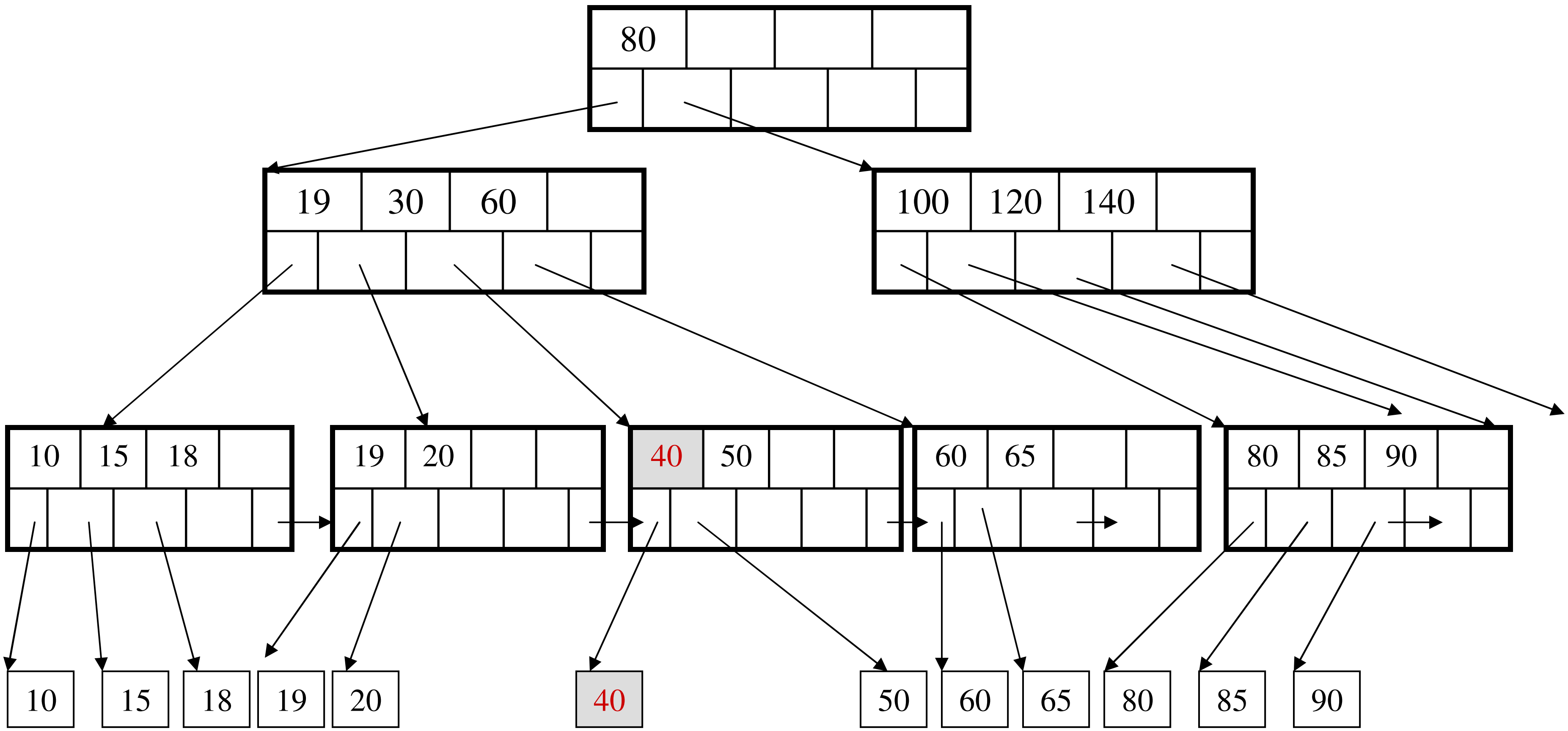
Deletion from a B+ Tree

After deleting 25
Need to rebalance
Rotate



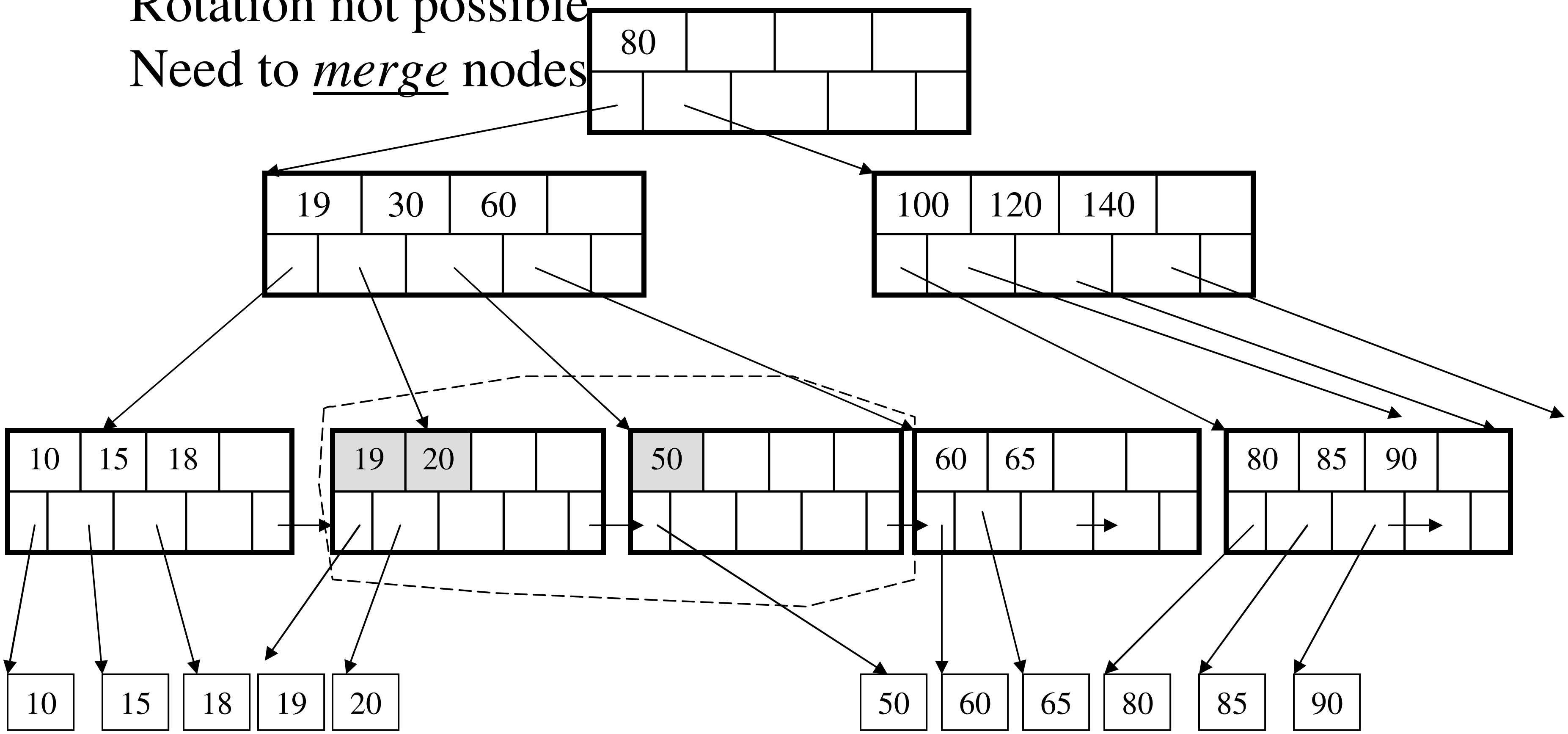
Deletion from a B+ Tree

Now delete 40



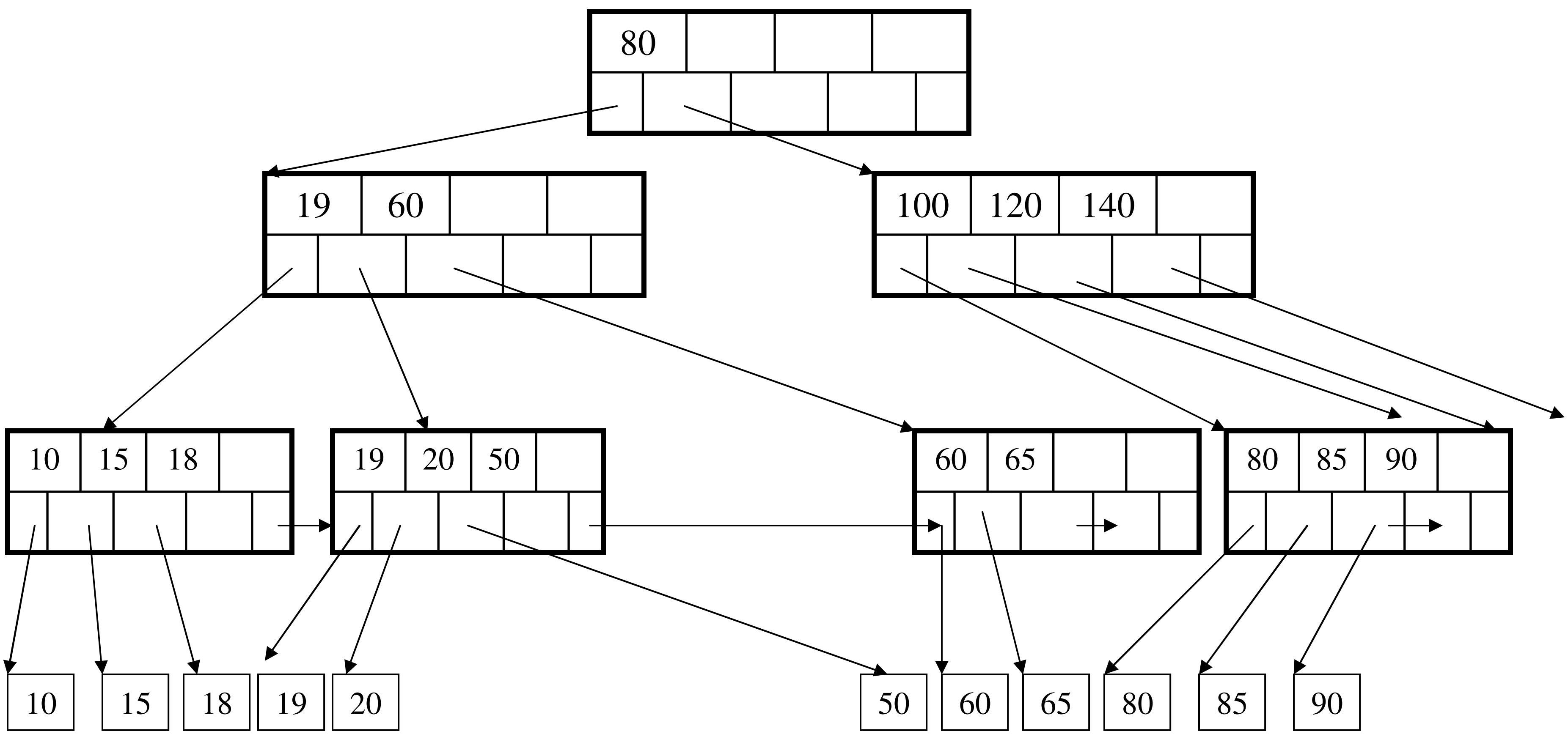
Deletion from a B+ Tree

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



Deletion from a B+ Tree

Final tree



Summary on B+ Trees

- Default index structure on most DBMS
- Very effective at answering ‘point’ queries:
  productName = ‘gizmo’
- Effective for range queries:
  50 < price AND price < 100
- Less effective for multirange:
  50 < price < 100 AND 2 < quant < 20