Indexing

INF 551

Wensheng Wu

Outline

- Types of indexes
- B+ trees

Indexes

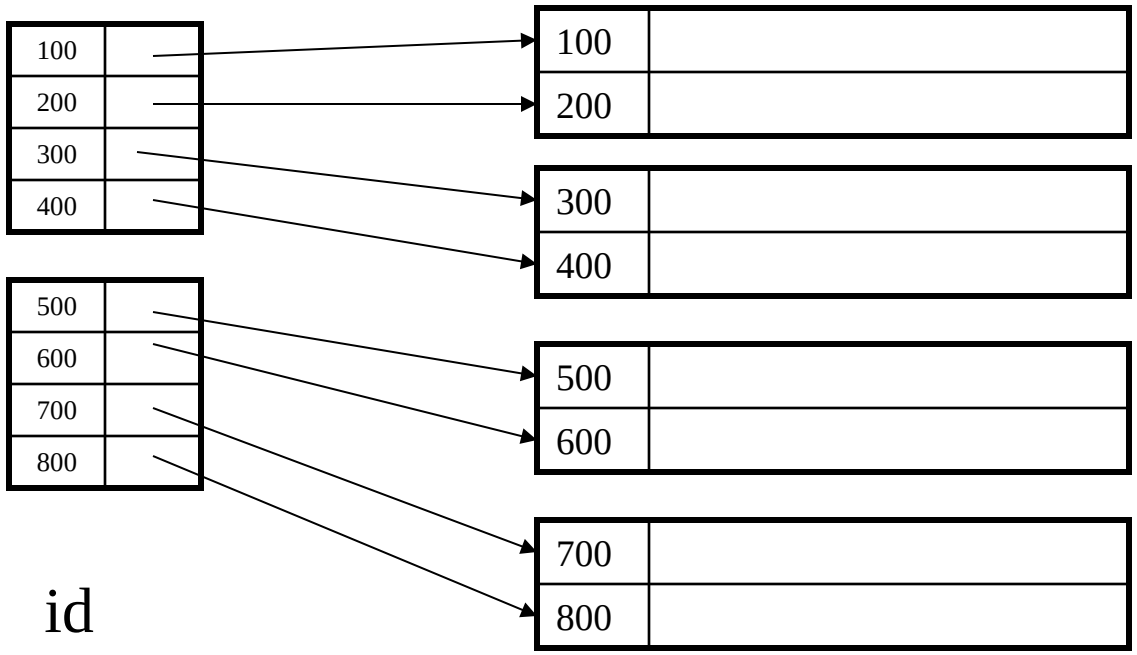
- An *index* is a data structure that speeds up selections on the *search key field(s)*
- *Fields = attributes*
- Search key = any subset of the fields of a relation
  - *Search key* is not the same as *key* (minimal set of fields that uniquely identify a record in a relation).
- Entries in an index: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

Index Classification

- Clustered/unclustered
  - Clustered = records sorted & stored in the order of search key
  - Unclustered = records are not sorted in key order
- Dense/sparse
  - Dense = each record has an entry in the index
  - Sparse = only some records have
- B+ tree / hash table / …

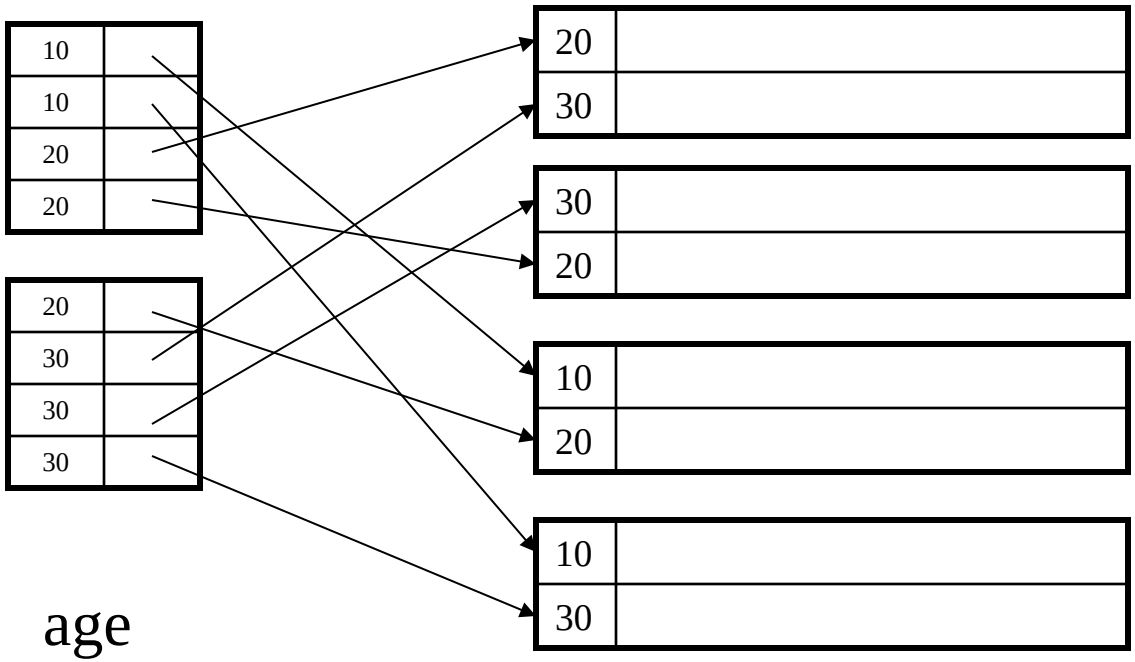
Clustered Index

- Records are sorted on the index attribute
  - Often for index on primary key
  - E.g., employee(id, name, age, salary)



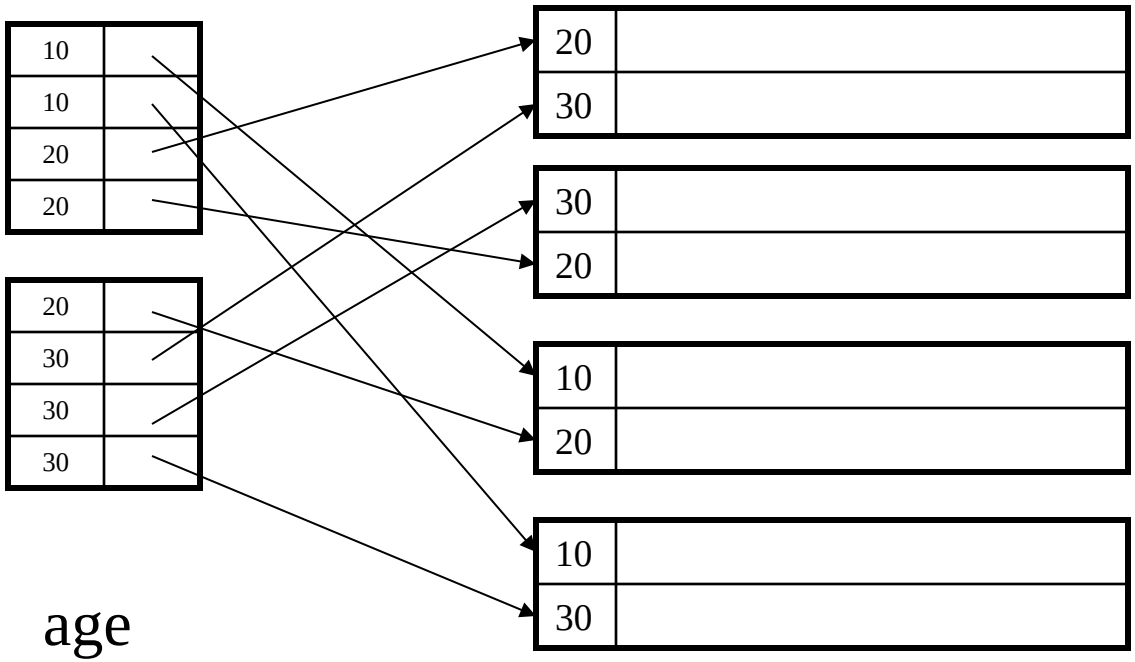
Unclustered Indexes

- Often for indexes on attributes other than primary key



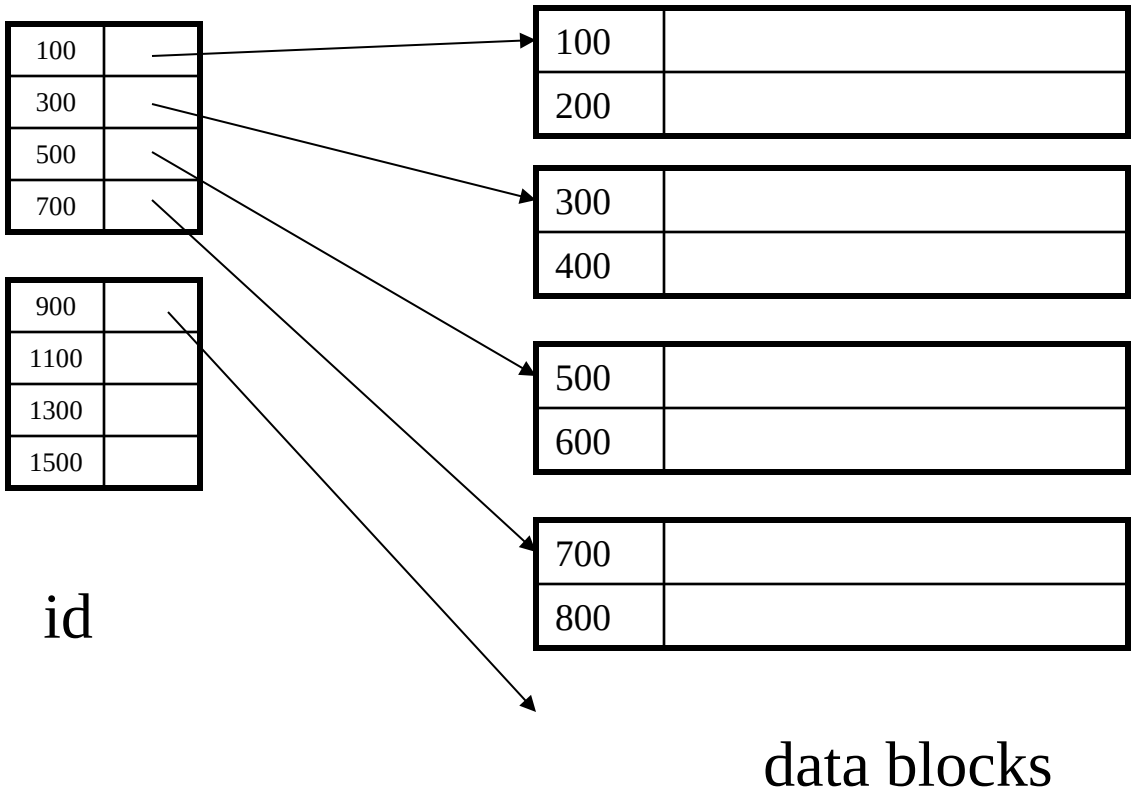
Dense Index

- *Dense* index: one key per data record
- See Sections 14.1.2 – 14.1.3 in book [GVW]



Sparse Index

- *Sparse* index: one key per data block



Query Types

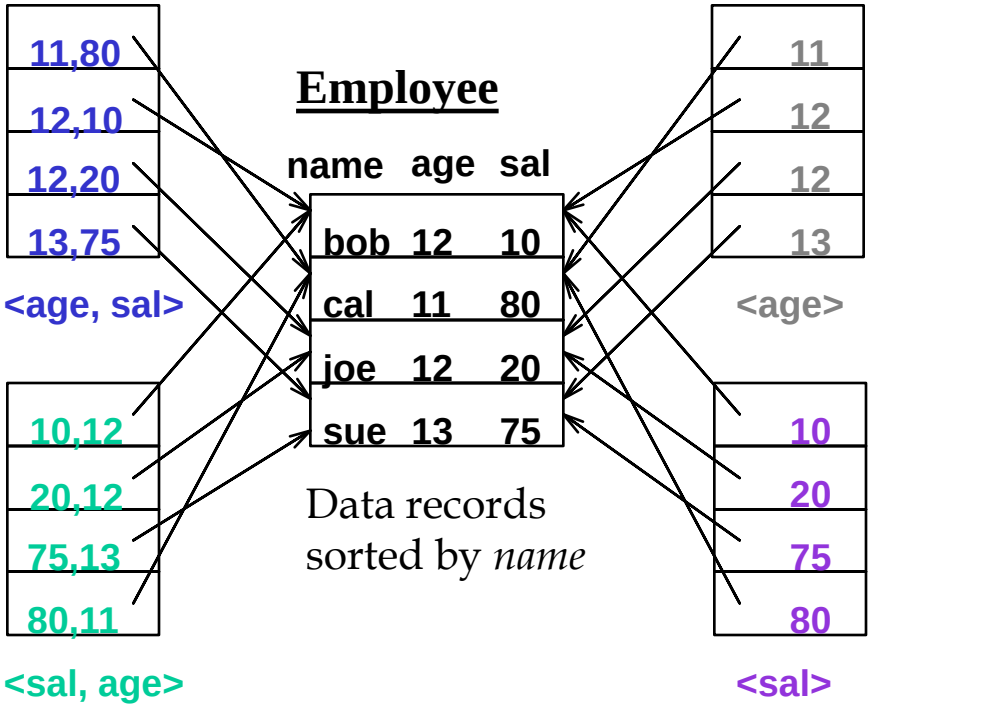
- Equality query: <attribute> = <value>
  - E.g., age = 20, sal = 75

- Range query: <attribute> <inequality operator> <value>
  - Inequality operator: <, >, <=, >=
  - E.g., age > 20 or sal <= 75

Composite Search Keys

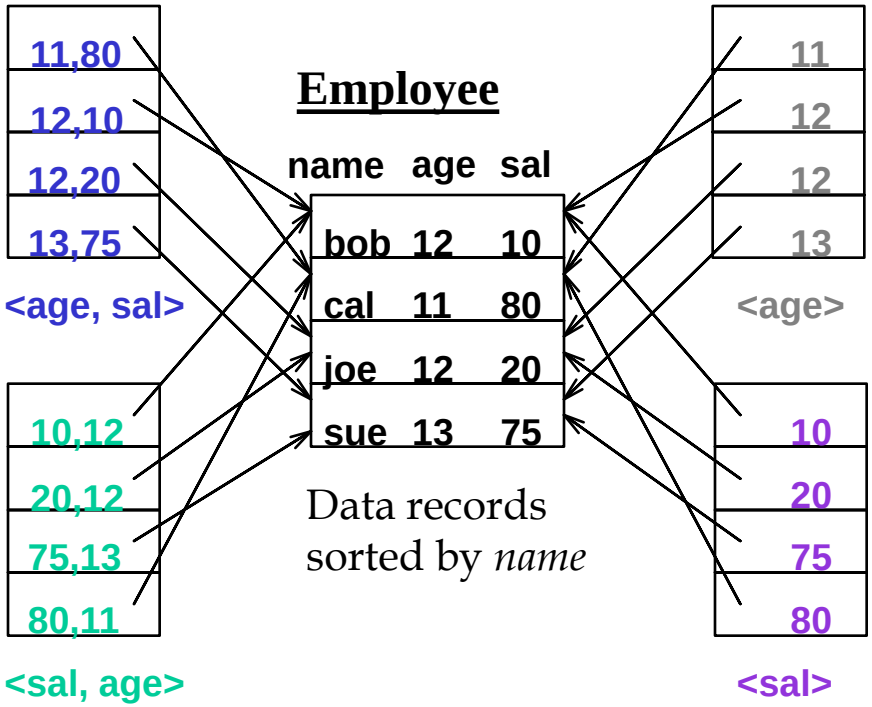
- *Composite Search Keys*: Search key = a list of fields.

Keys in index sorted by <*age,sal*>: *i.e., first by age; if ties, by sal*



Questions

- Which index is useful for queries:
  - Sal > 75
  - Age = 12 and sal > 10
  - Age > 12



Outline

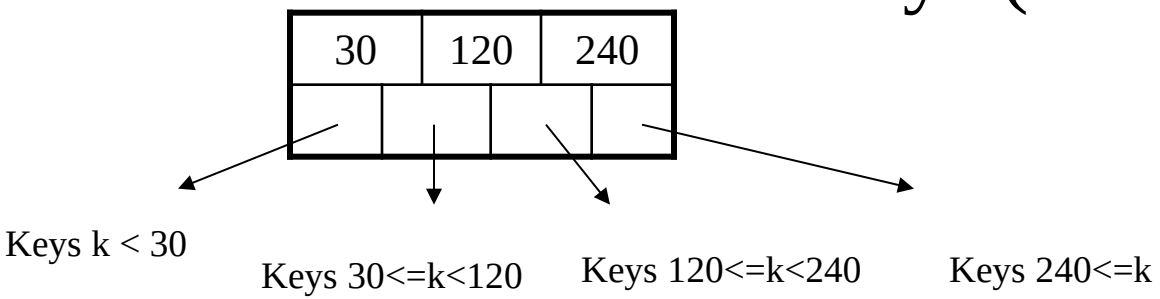
- Types of indexes
- B+ trees

B+ Trees

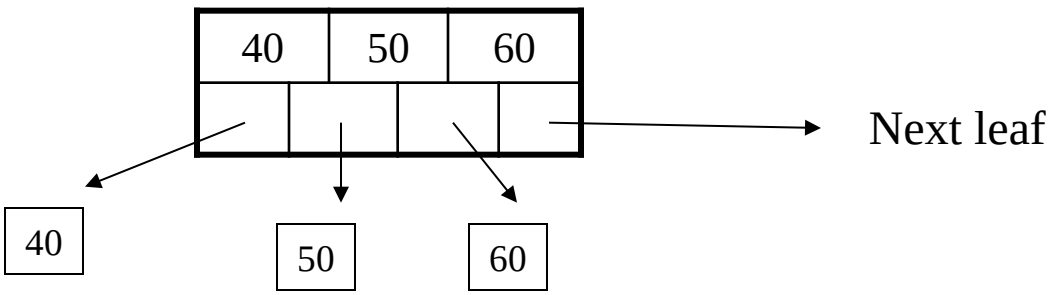
- Search trees
- Idea in B Trees:
  - make 1 node = 1 block
- Idea in B+ Trees:
  - Make leaves into a linked list
  - Efficiently support range queries

B+ Trees Basics

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

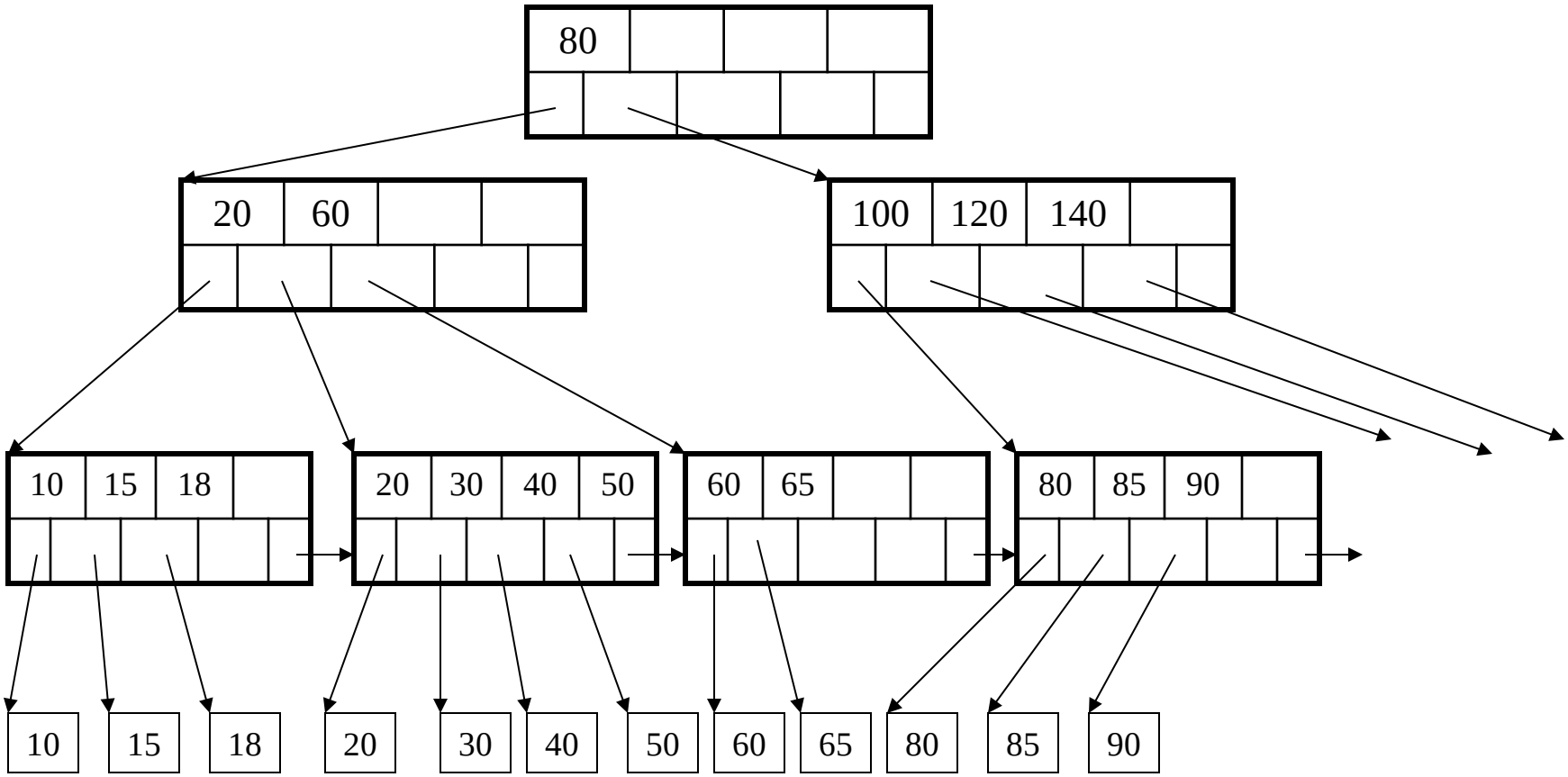


- Each leaf has >=d and <= 2d keys:



B+ Tree Example

d = 2



B+ Tree Design

- How large is d ?
- Example:
  - Key size = 4 bytes
  - Pointer size = 8 bytes
  - Block size = 4096 byes
- 2d * 4 + (2d+1) * 8 <= 4096
- d = 170 (~170.33)

B+ Trees in Practice

- Typical order d = 100.
- Typical fill-factor (minimum in practice): 66.7% (i.e., 2/3) (note minimum fill factor in design: 50%)
  - Minimum # of keys in a node = 133 (200 * 2/3)

- Capacities (# of records which the index supports):
  - Height 1 (tree with a single root): 133 records
  - Height 2: 133^2 = 17, 689 records (134*133 to be exact)
  - Height 3: 133^3 = 2, 352, 637 records (134^2 *133)
  - Height 4: 133^4 = 312,900,721 records (134^3 * 133)

B+-tree in Practice

- Can often hold top levels in buffer pool:
  - Level 1 = 1 page = 4KB
  - Level 2 = 133 pages = 532KB
  - Level 3 = 17,689 pages = 70, 756KB ~ 70MB

Searching a B+ Tree

- Equality search:
  - Start at the root
  - Proceed down, to the leaf

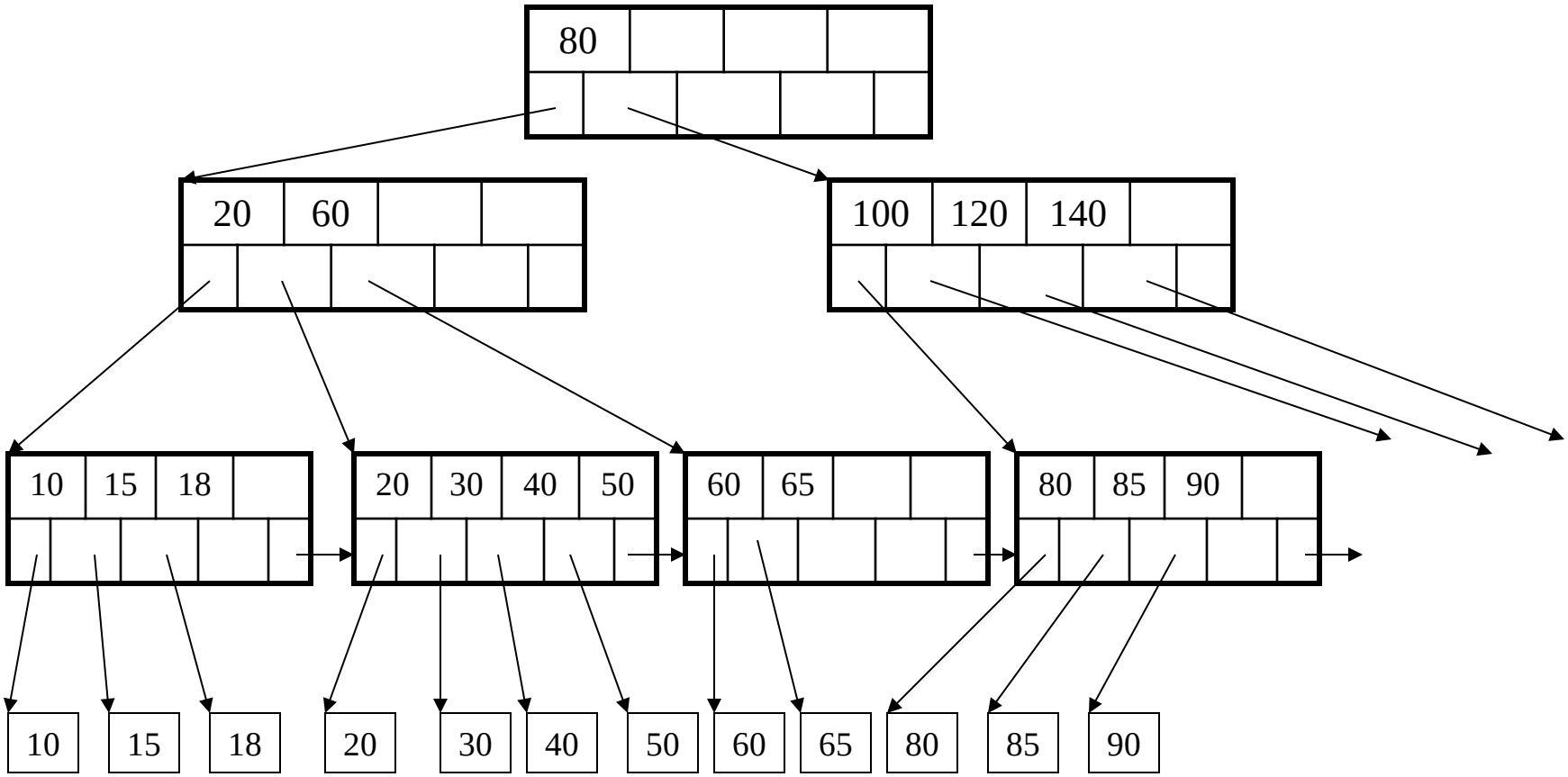
```
Select name
From people
Where age = 25
```

- Range query:
  - Finding the first leaf
  - Then sequential traversal of leaves

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

Example

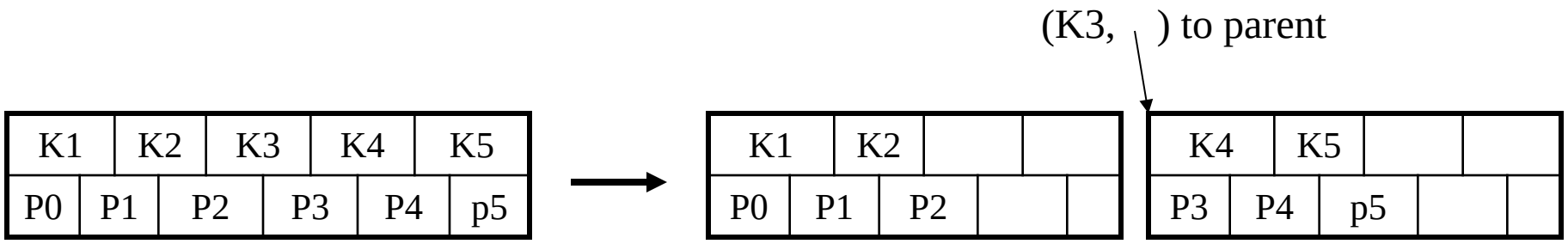
20 <= age and age <= 55



Insertion into a B+ Tree

Insert (K, P)

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

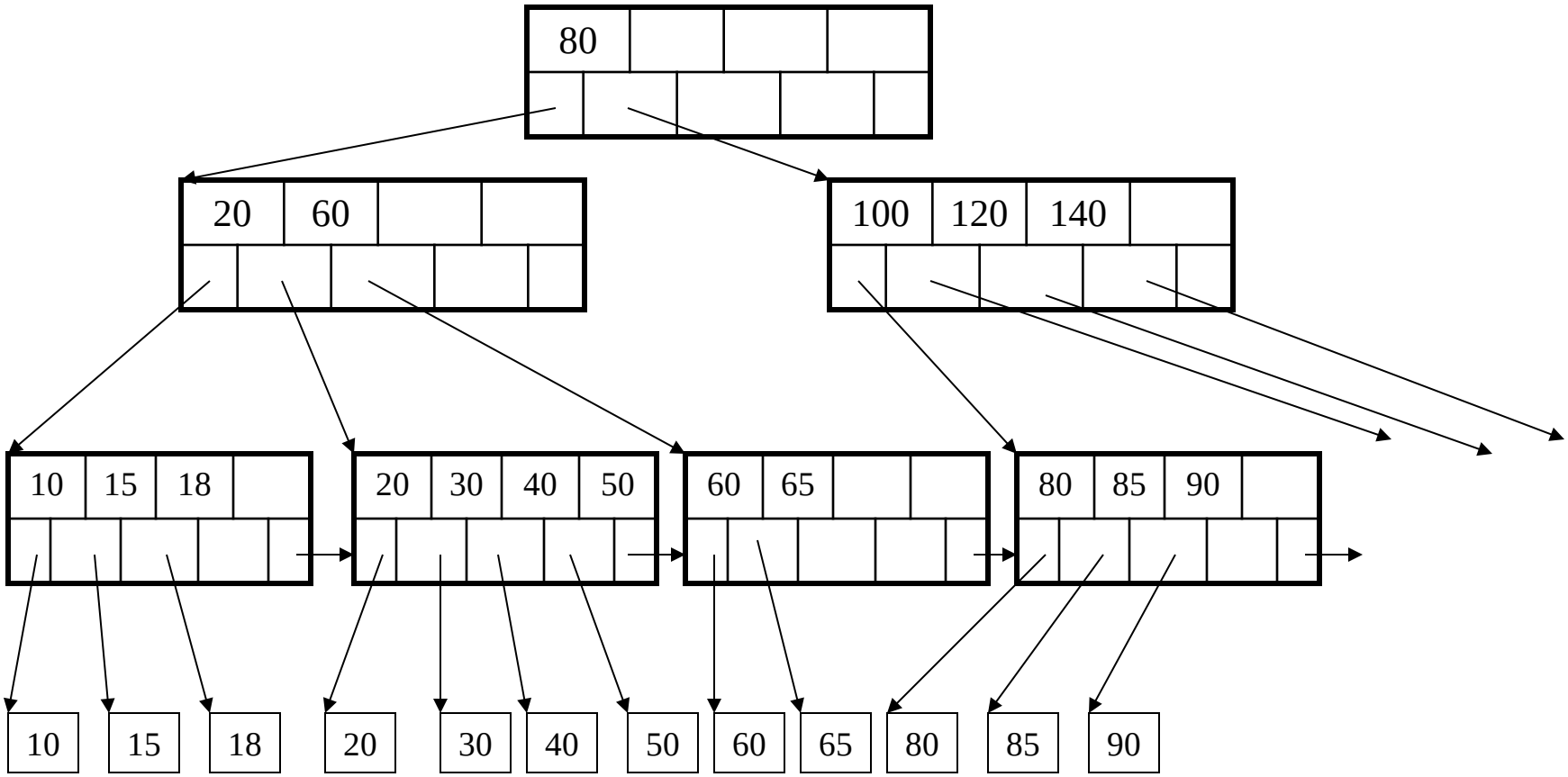


Notes

- Splitting of leaf may lead to splitting of its parent and ancestors
- When splitting a leaf, middle key (e.g., K3) is also kept in the new node on the right
- No need to retain middle key in the split node when splitting an internal node (but need to insert it into parent)
- When root is split, new root has only one key

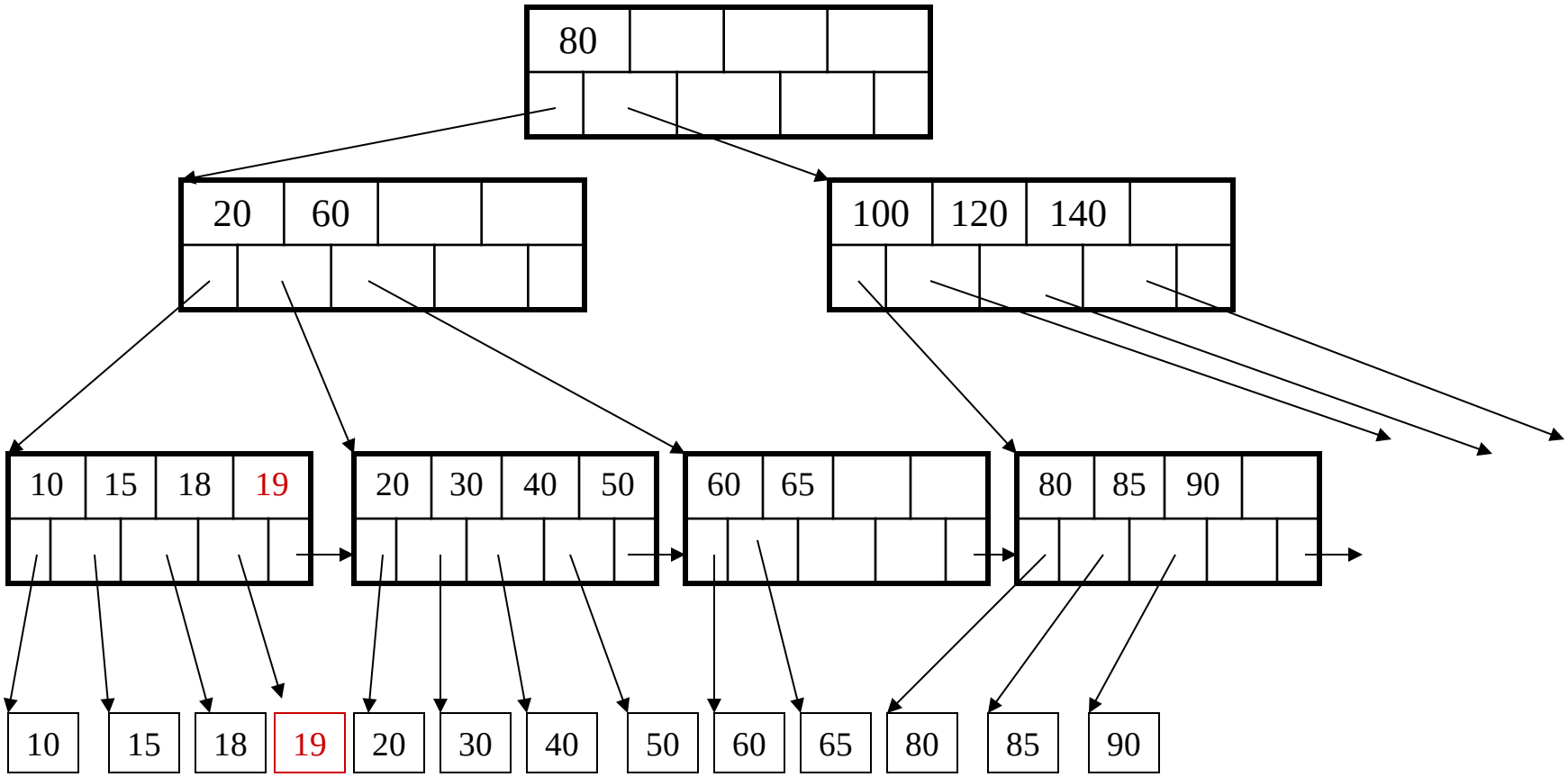
Insertion into a B+ Tree

Insert K=19



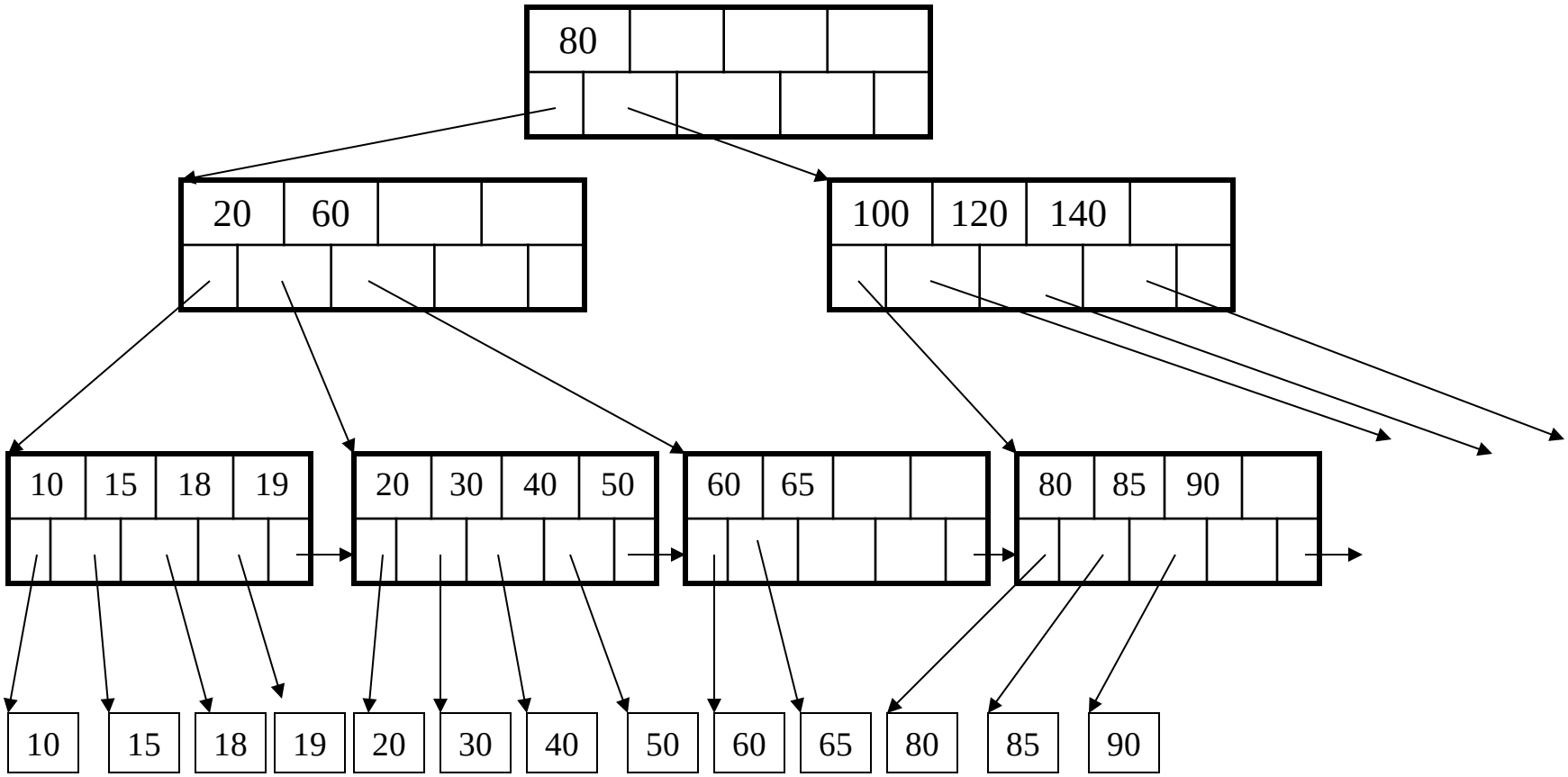
Insertion into a B+ Tree

After insertion



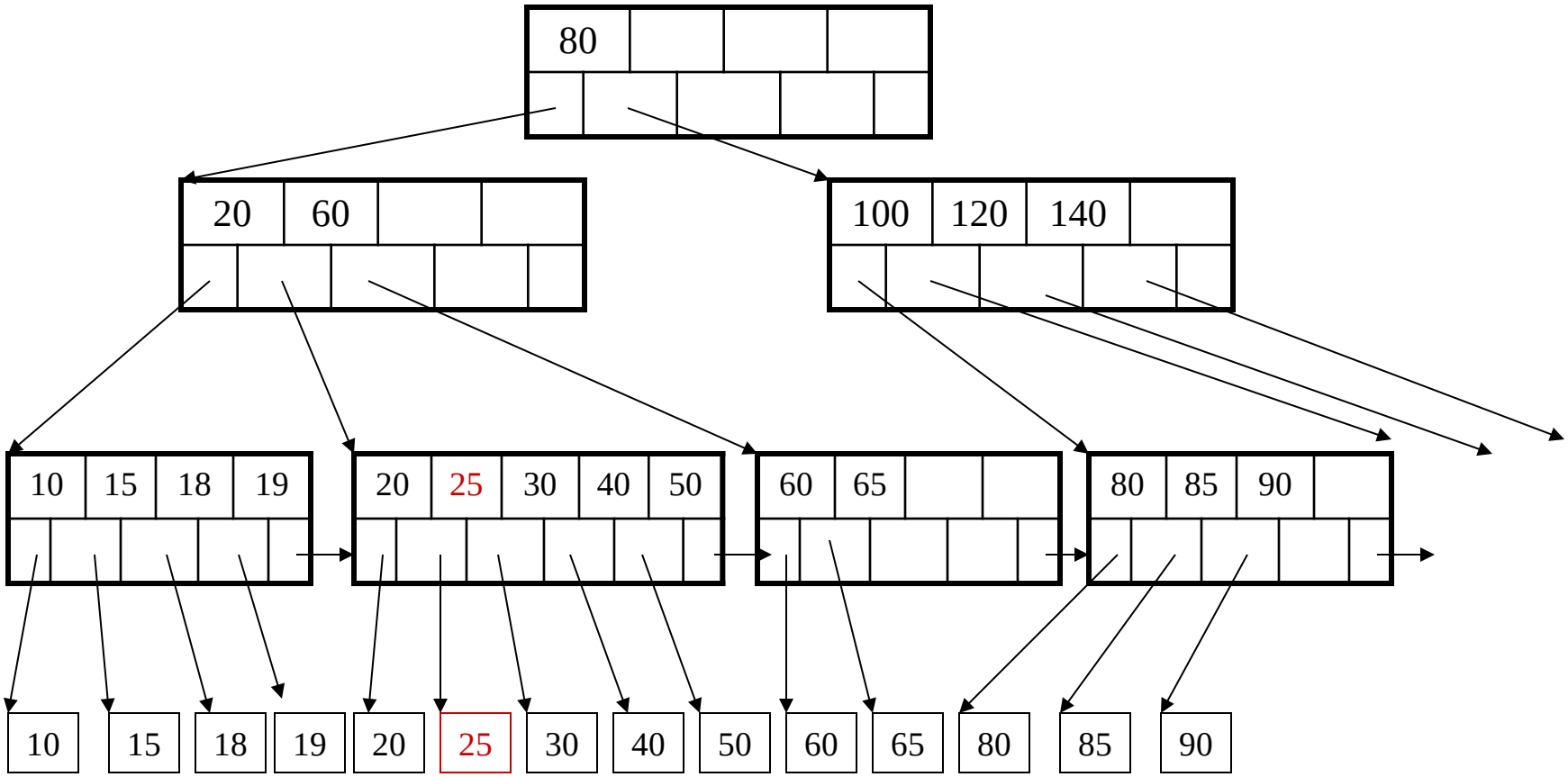
Insertion into a B+ Tree

Now insert 25



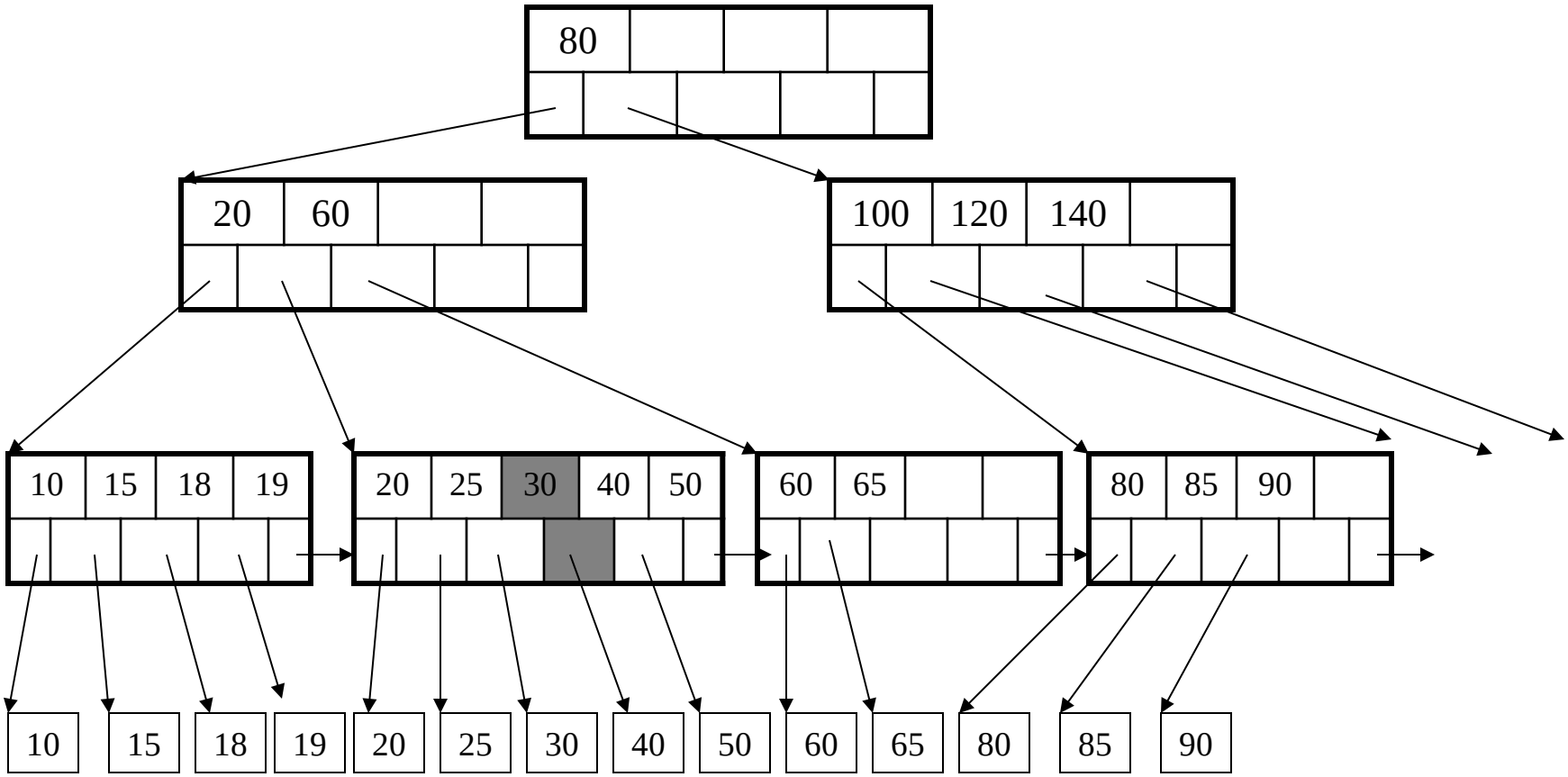
Insertion into a B+ Tree

After insertion



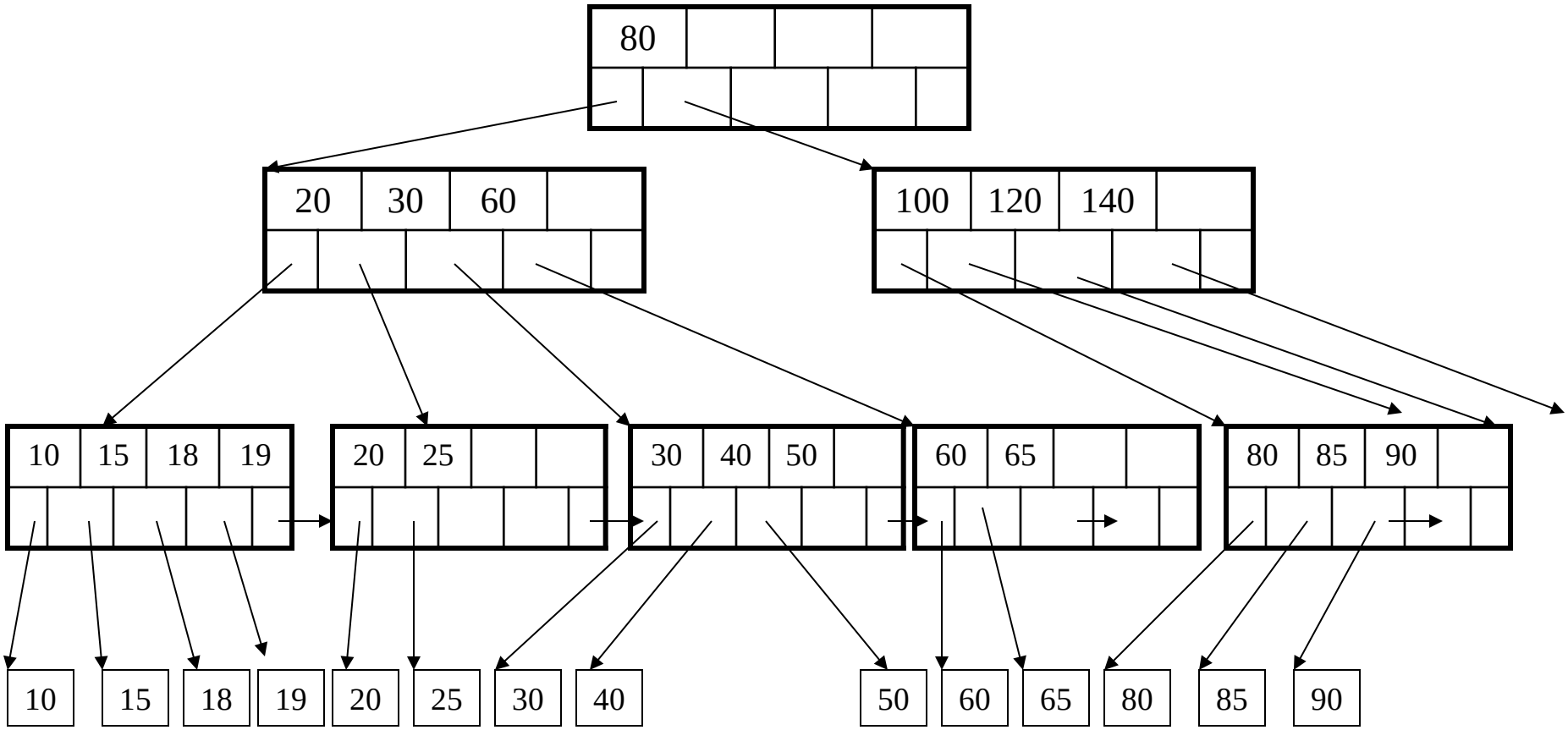
Insertion into a B+ Tree

But now have to split !



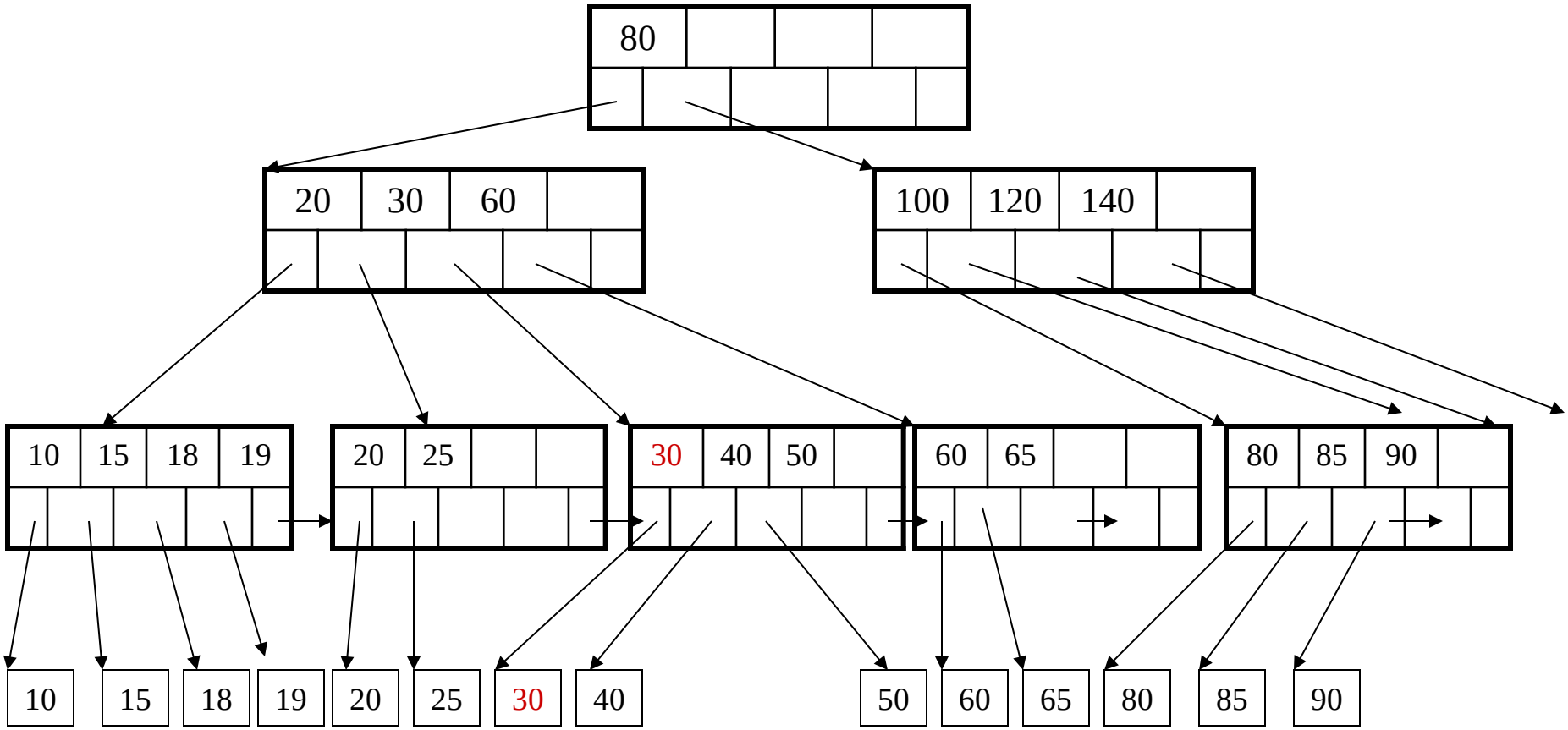
Insertion into a B+ Tree

After the split



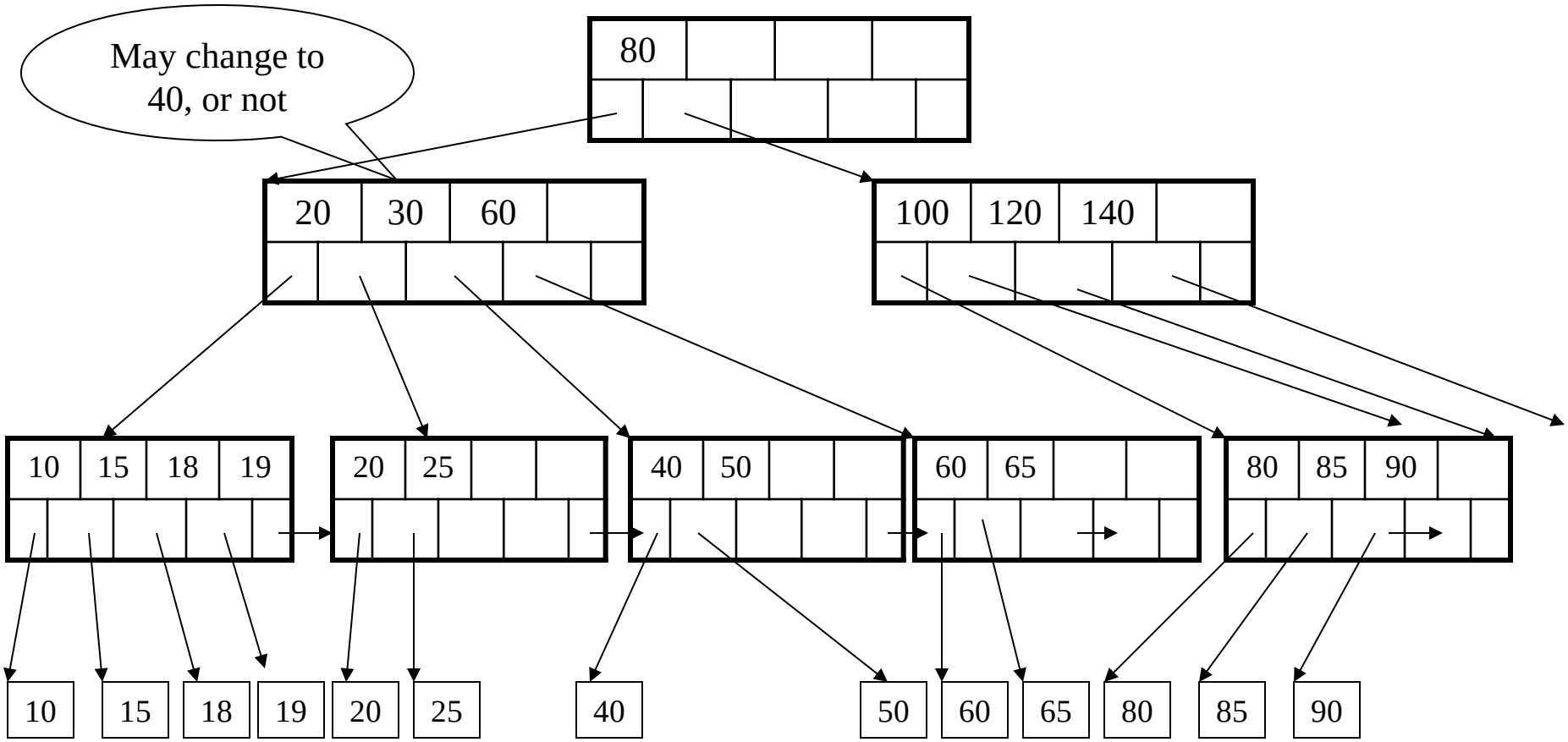
Deletion from a B+ Tree

Delete 30



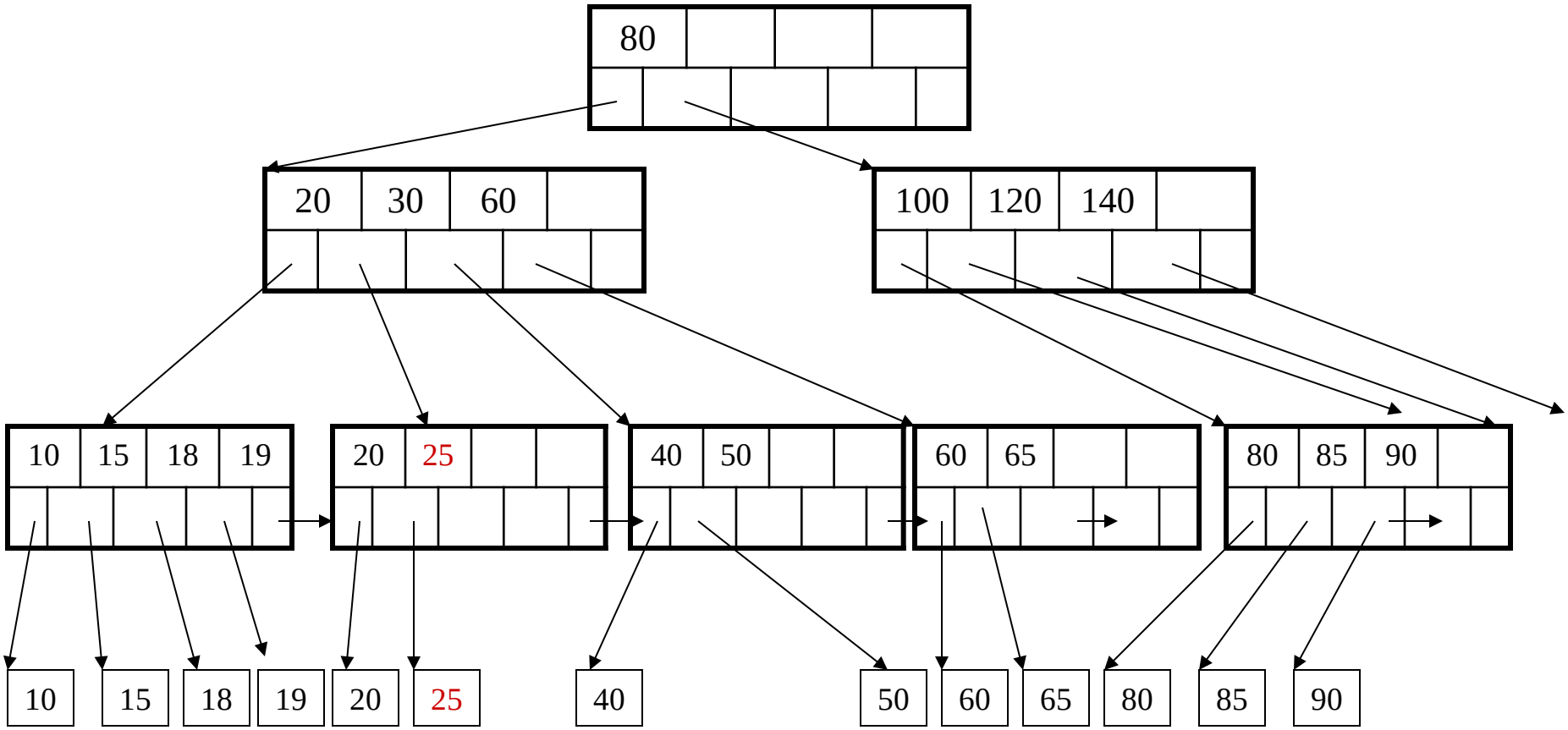
Deletion from a B+ Tree

After deleting 30



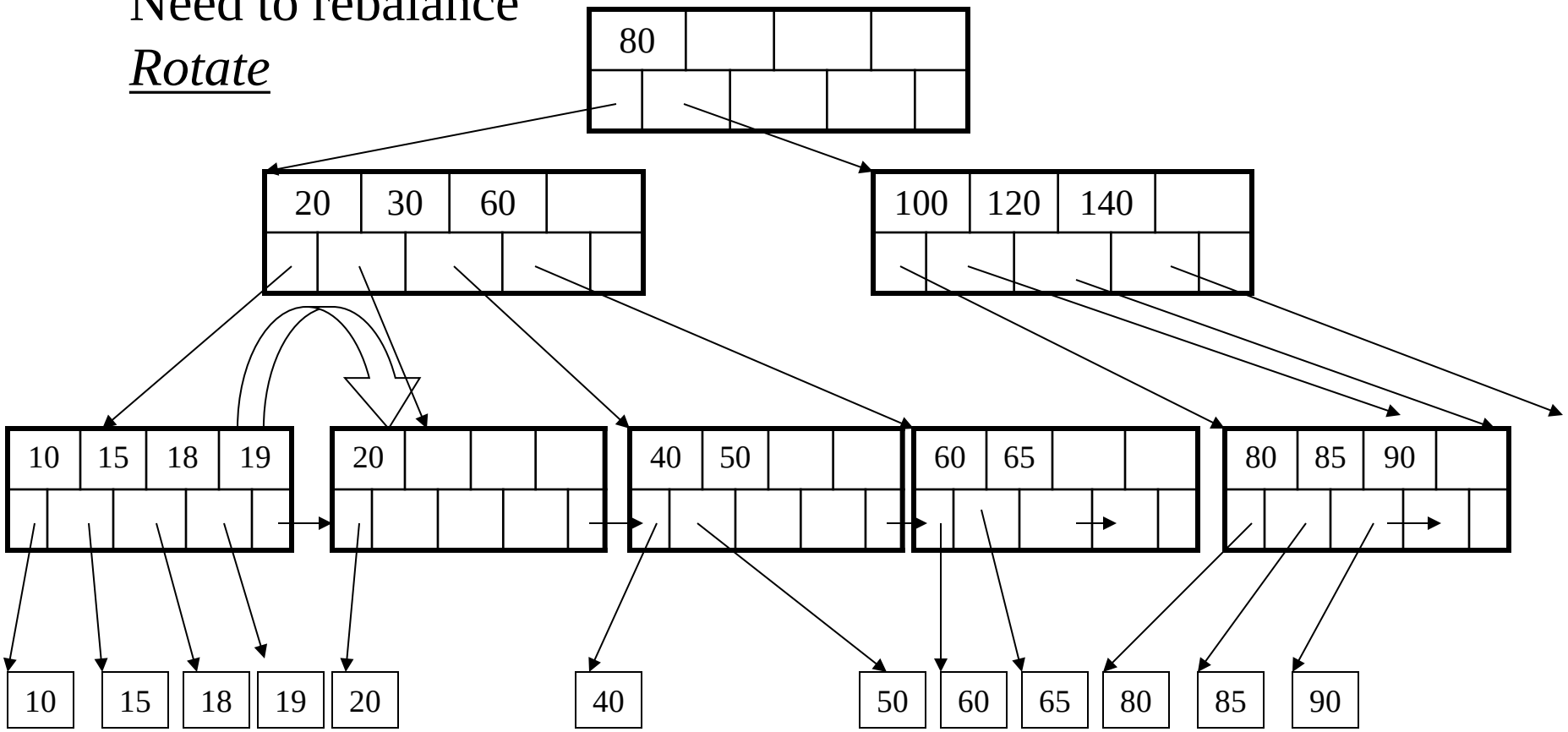
Deletion from a B+ Tree

Now delete 25



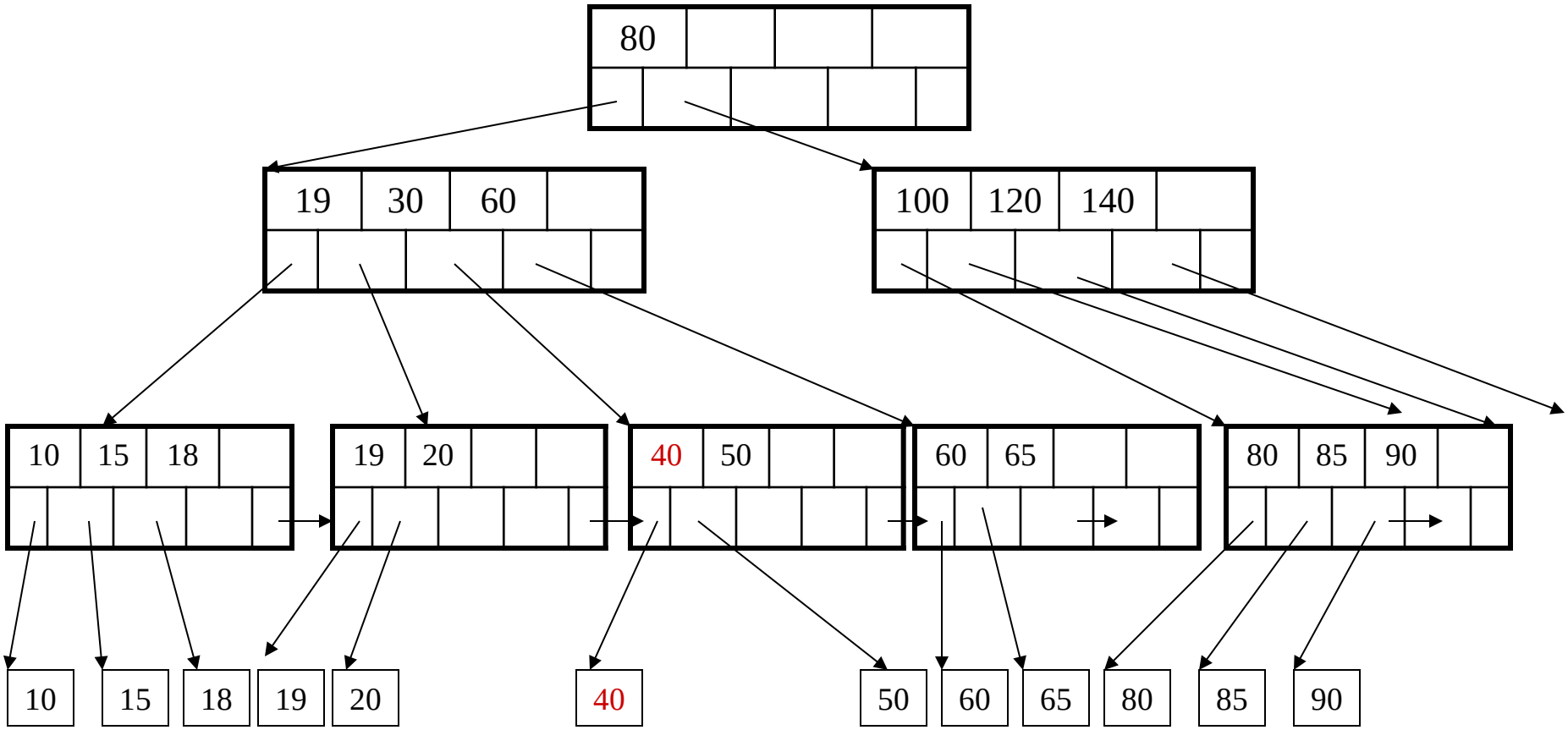
Deletion from a B+ Tree

After deleting 25
Need to rebalance
Rotate



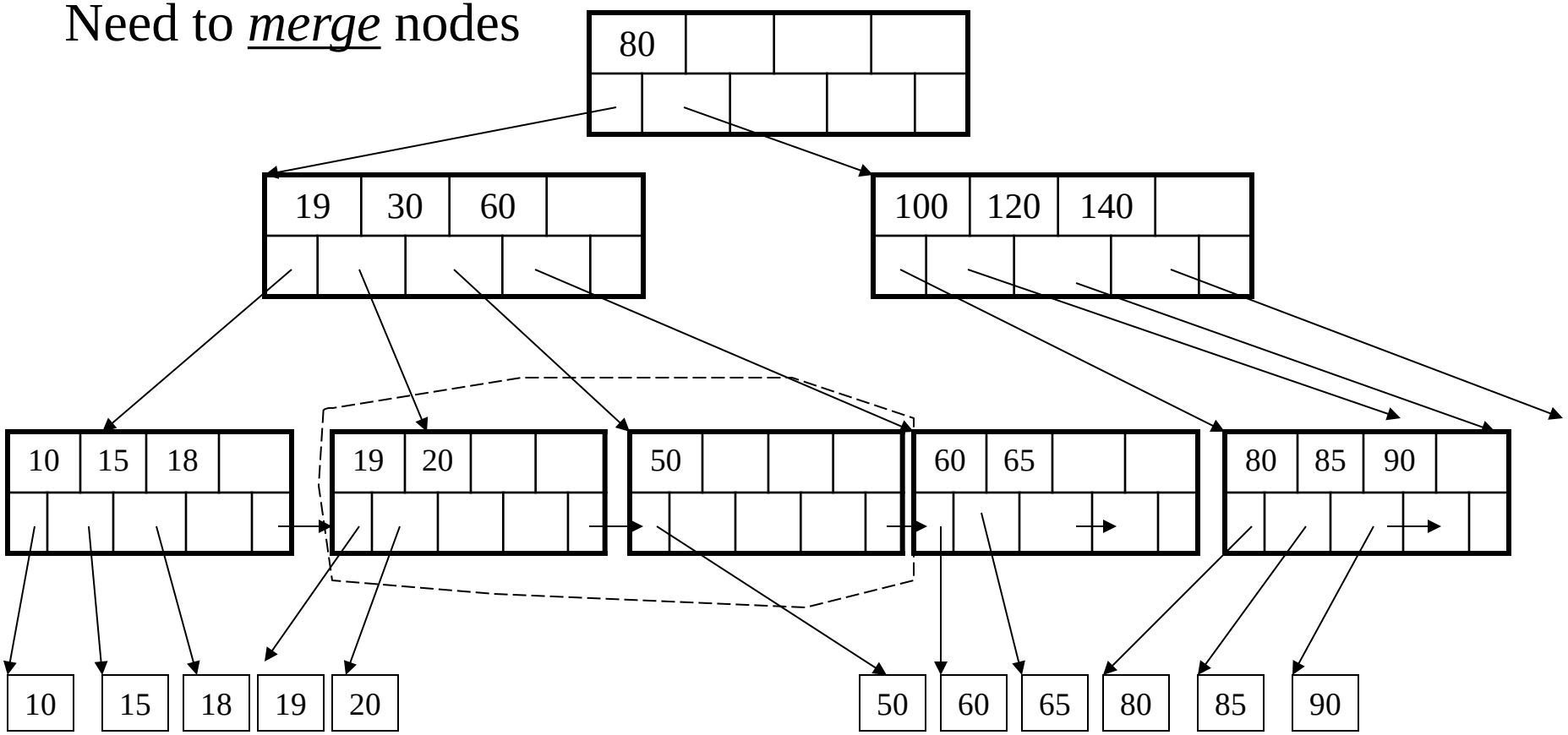
Deletion from a B+ Tree

Now delete 40



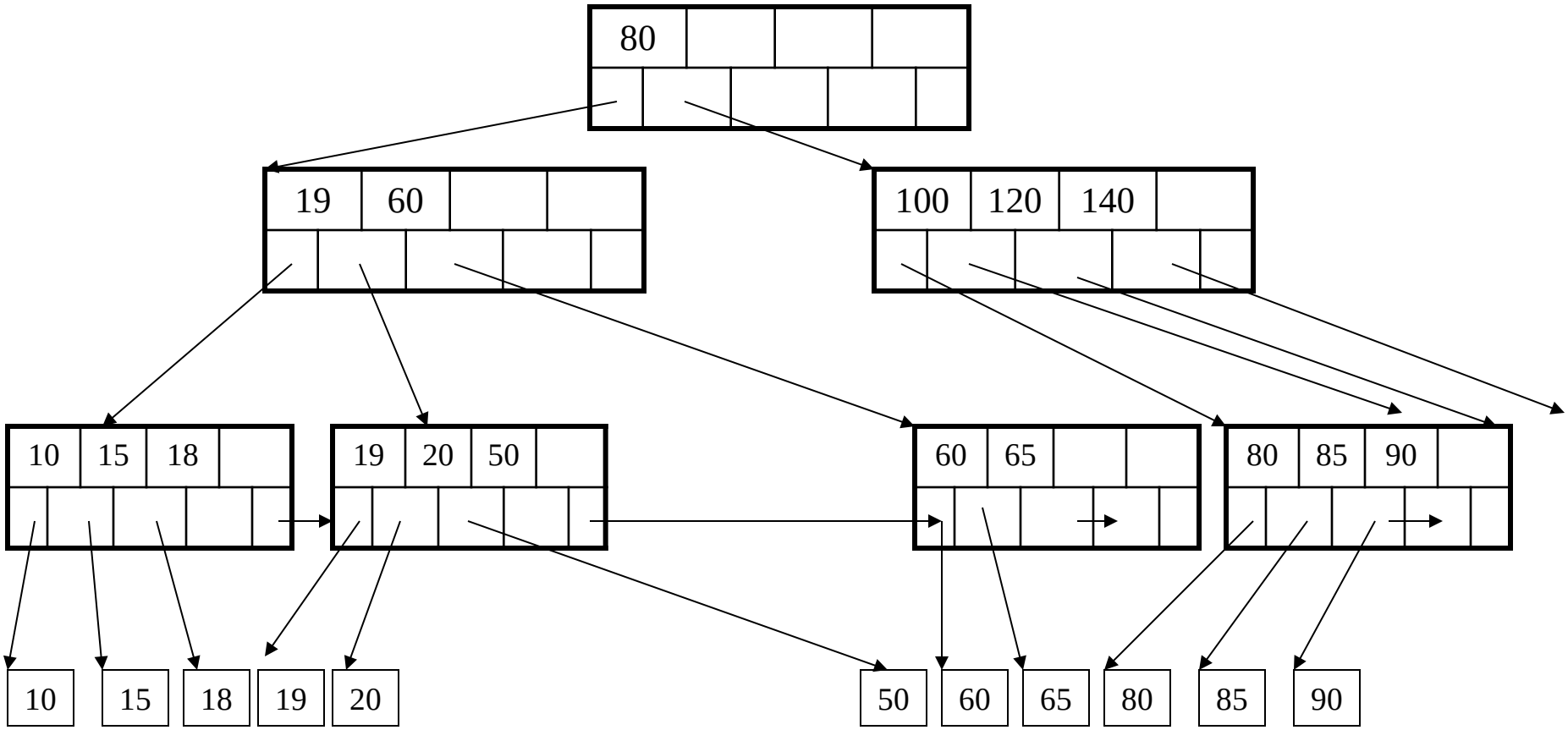
Deletion from a B+ Tree

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



Deletion from a B+ Tree

Final tree



Deletion Strategy

- If a node is below the min capacity after deletion…
- Try the following in the given order
  1. move a key from immediate left sibling;
  2. move a key from immediate right sibling;
  3. merge with immediate left sibling;
  4. merge with immediate right sibling

- Cases 3 and 4 may lead to further removal of key from parent, and more fixing

Another insertion example