

Indexing

Bloom filter

DSCI 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 = key
 - r = record(s) OR record id(s)

Index Classification

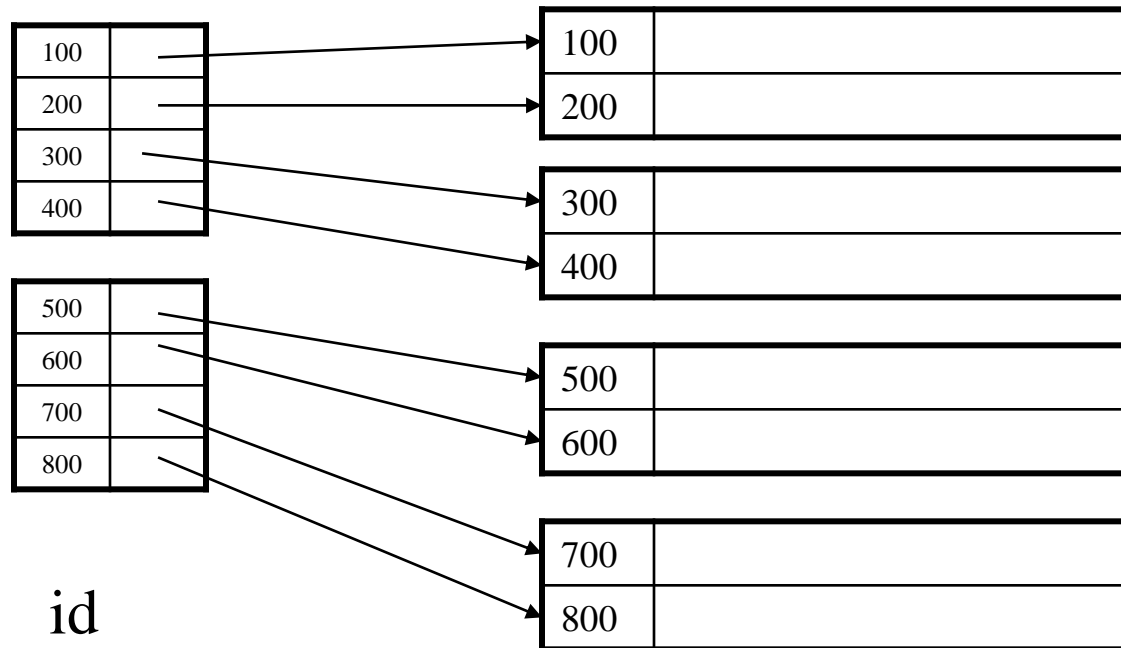
- Clustered/unclustered
 - Clustered = records sorted & stored in the order of search key
 - Unclustered = records are not sorted in key order
- 聚簇索引/非聚簇索引
- B+ tree / hash table / ...

MySQL

- Automatically creates a clustered index for:
 - Primary key if exists;
 - Otherwise 1st unique key;
 - If no unique keys, on row ID (a hidden attribute)
- Row data are stored with the clustered index
- More details [here](#)

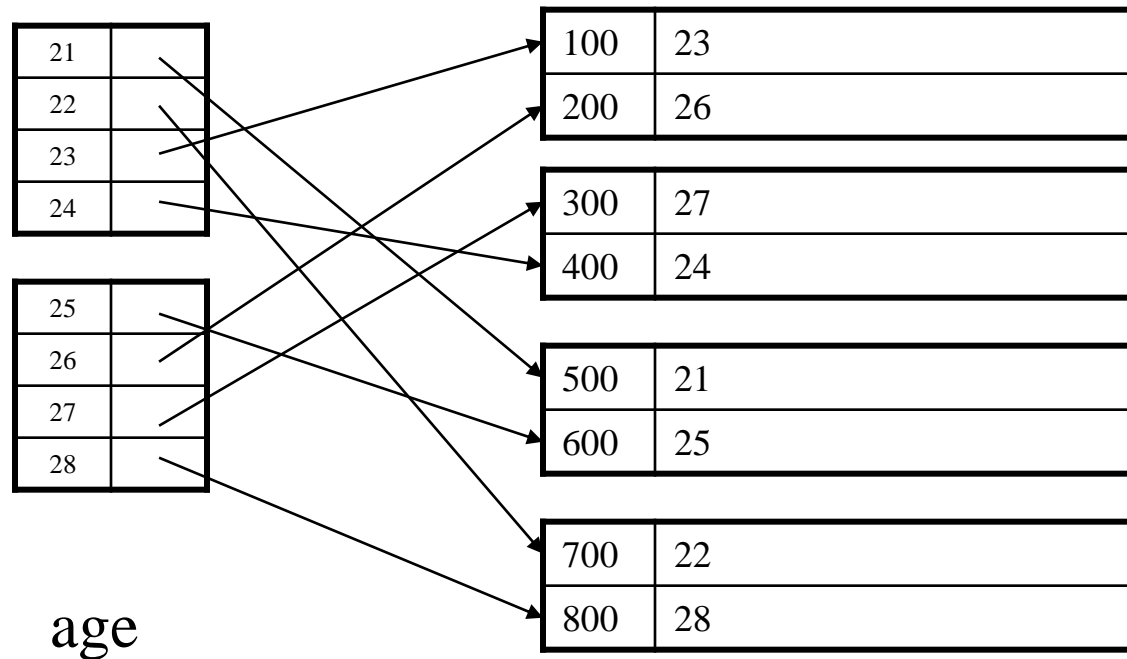
Clustered Index

- Records are sorted on the search key
 - E.g., employee(id, name, age, salary)



Unclustered Indexes

- Records NOT sorted by the search key



Query Types

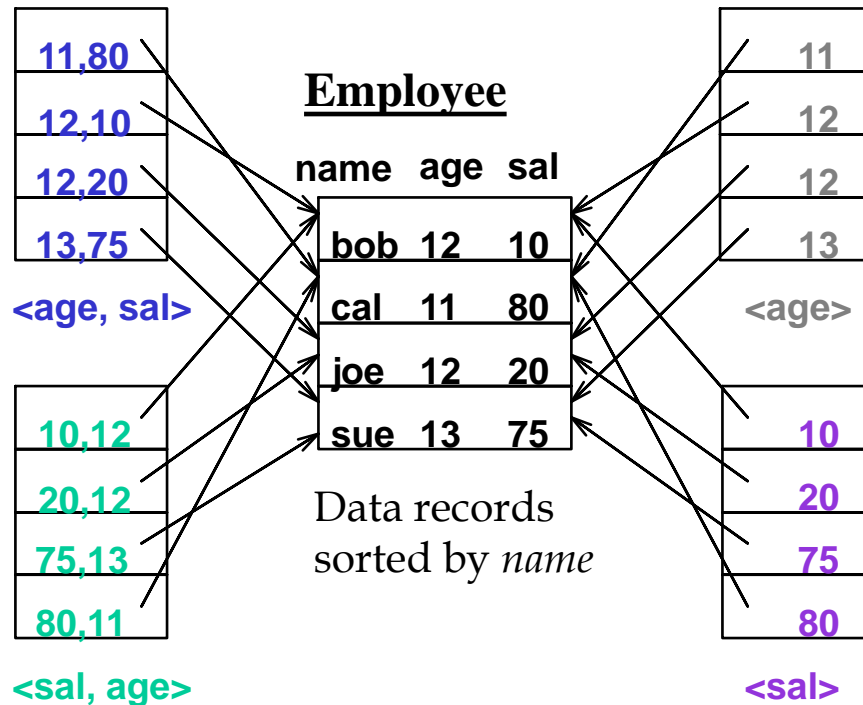
- Equality/point query: $\langle \text{attribute} \rangle = \langle \text{value} \rangle$
 - E.g., $\text{age} = 20$, $\text{sal} = 75$
- Range query: $\langle \text{attribute} \rangle \langle \text{inequality operator} \rangle \langle \text{value} \rangle$
 - Inequality operator: $<$, $>$, \leq , \geq
 - E.g., $\text{age} > 20$ or $\text{sal} \leq 75$

Composite Search Keys

联合索引

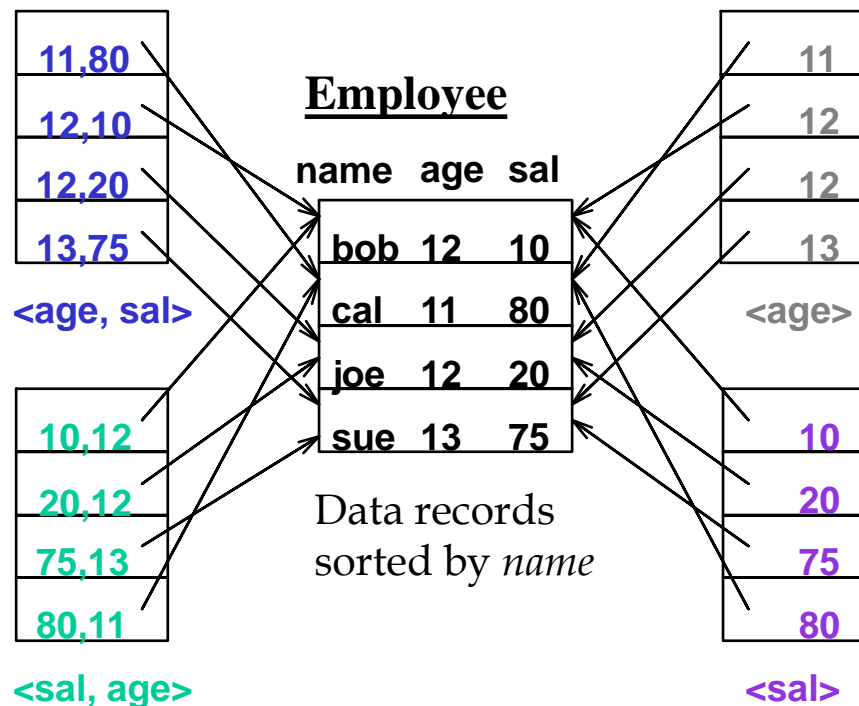
- Composite Search Keys*: Search key = a list of fields.

Keys in index
sorted by $\langle \text{age}, \text{sal} \rangle$:
i.e., first by age; if ties,
by sal



Questions

- Which index is useful for queries:
 - $Sal > 75$
 - $Age = 12$ and $sal > 60$
 - $Age > 12$



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

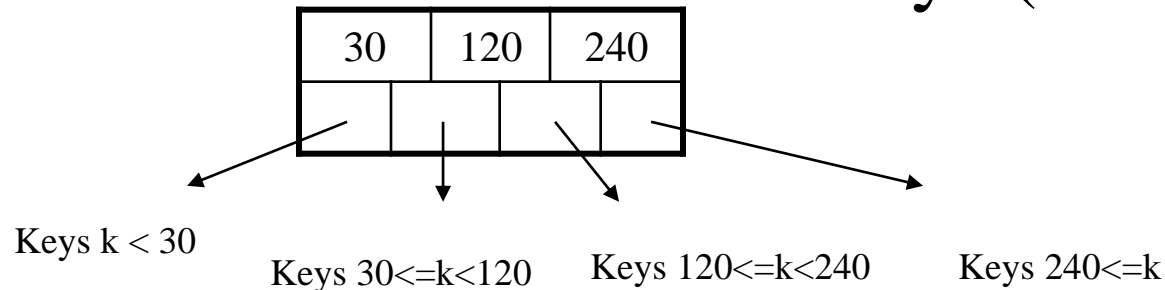
B+ Trees Basics

degree 度数

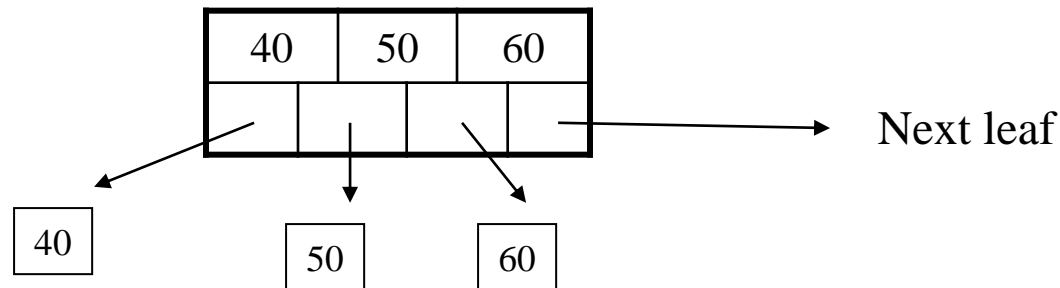
或

max-degree 最大度数

- Parameter d = the **degree** (also called order)
- Each node has $\geq d$ and $\leq 2d$ keys (except root)



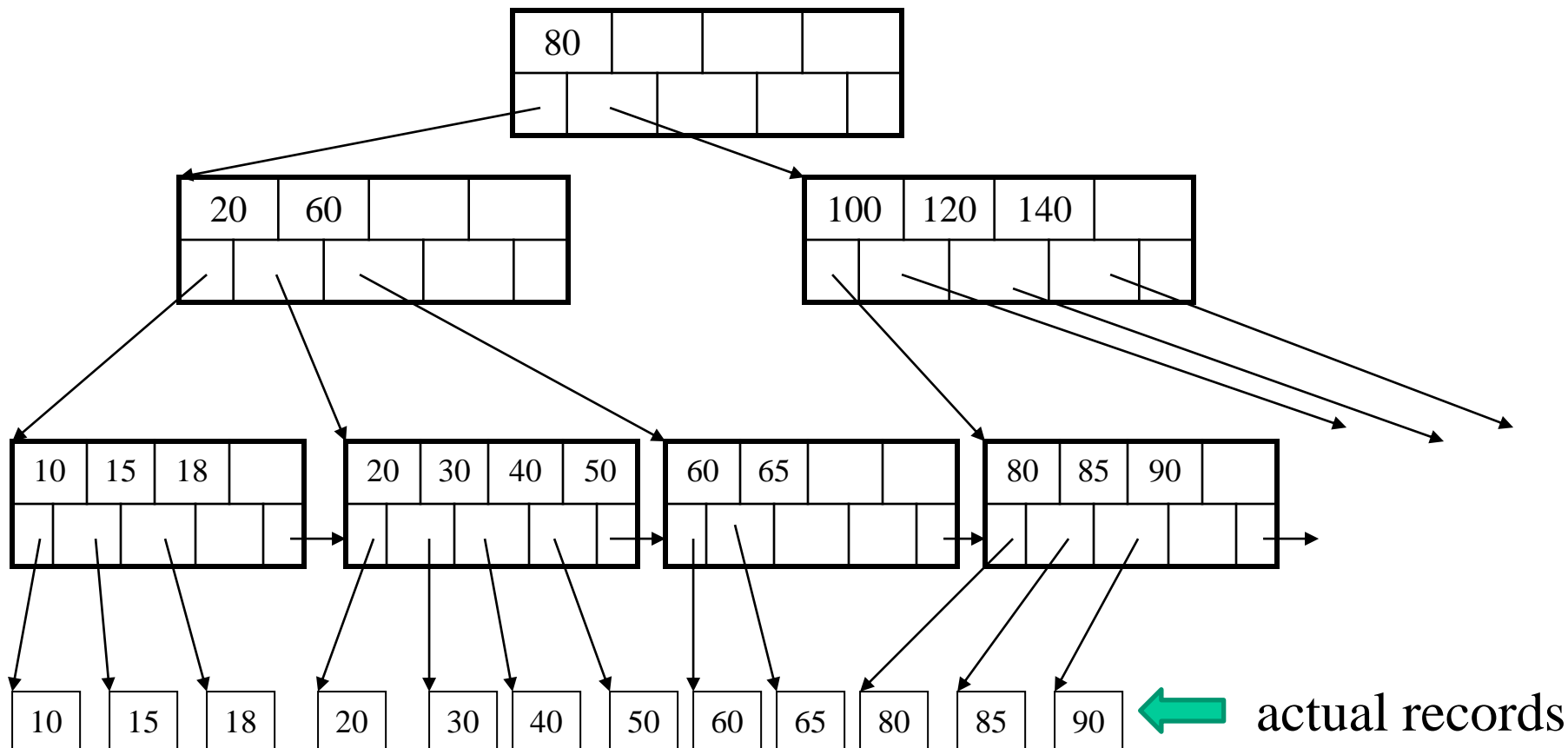
- Each leaf has $\geq d$ and $\leq 2d$ keys:



m filter

B+ Tree Example

$d = 2$



B+ Tree Design

- How large is d ?
- Example:
 - Key size = 4 bytes
 - Pointer size = 8 bytes
 - Block size = 4096 bytes
- $2d * 4 + (2d+1) * 8 \leq 4096$
- $d = 170 (\sim 170.33)$

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
- Range query [a, b]:
 - Finding the first leaf in the range
 - Then sequential traversal of leaves until ...

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

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

Searching a B+ Tree

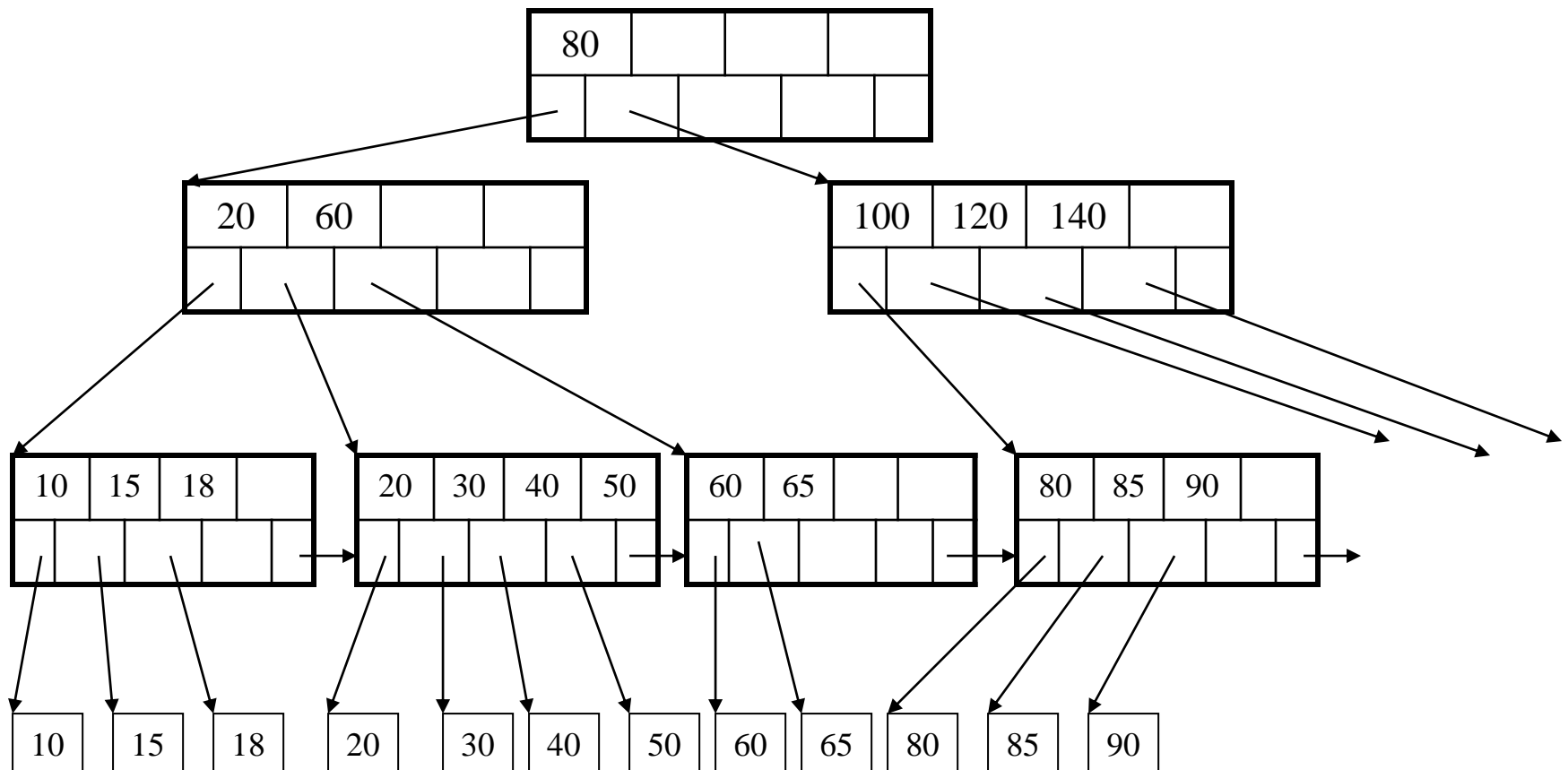
- Range query $[-, b]$:
 - Finding the left-most leaf
 - Then sequential traversal of leaves until ...
- Range query $[a, -]$:
 - Finding the leaf with a
 - Then sequential traversal until ...

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

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

Example

$20 \leq \text{age}$ and $\text{age} \leq 85$



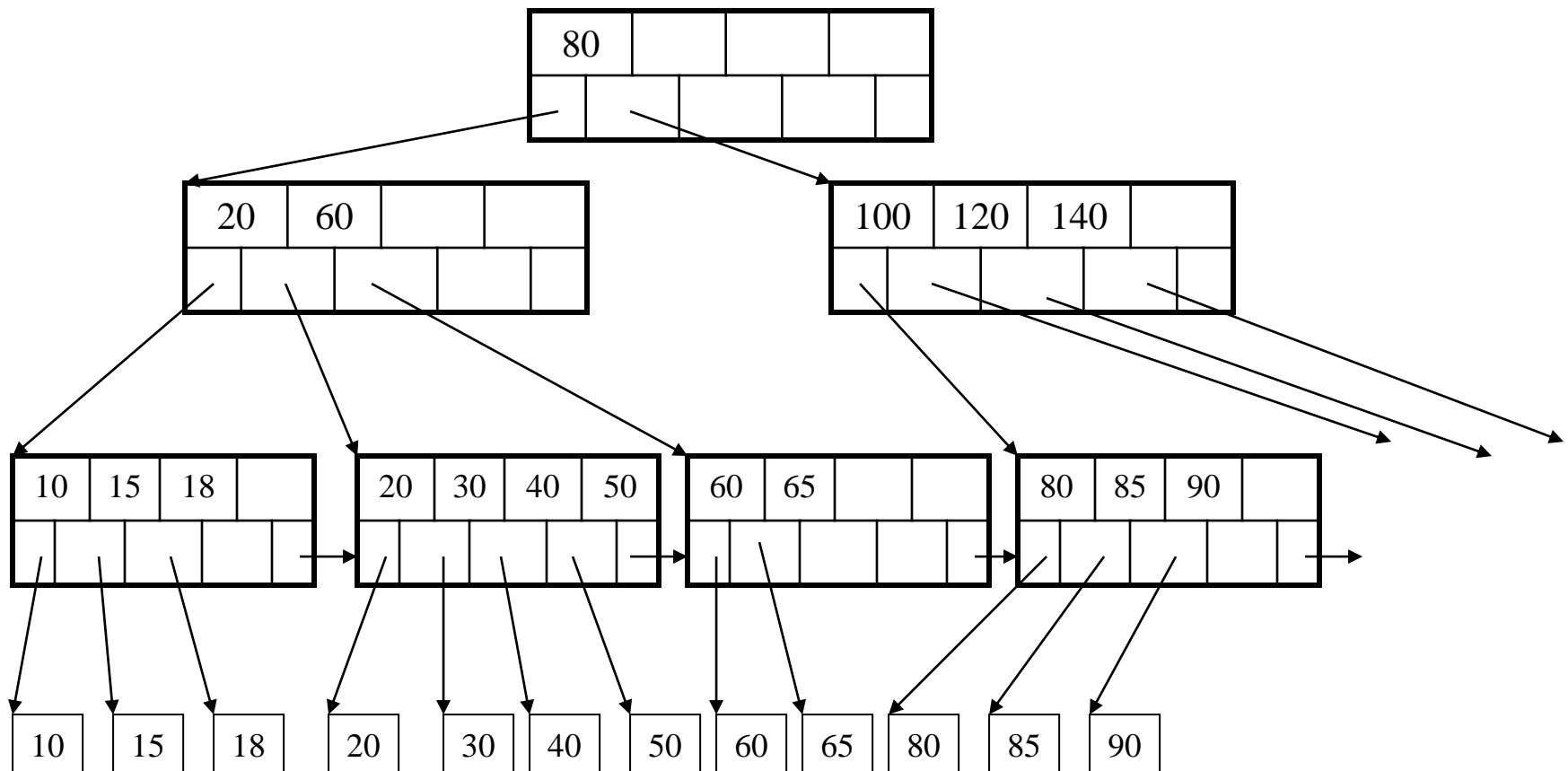
Insertion into a B+ Tree

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

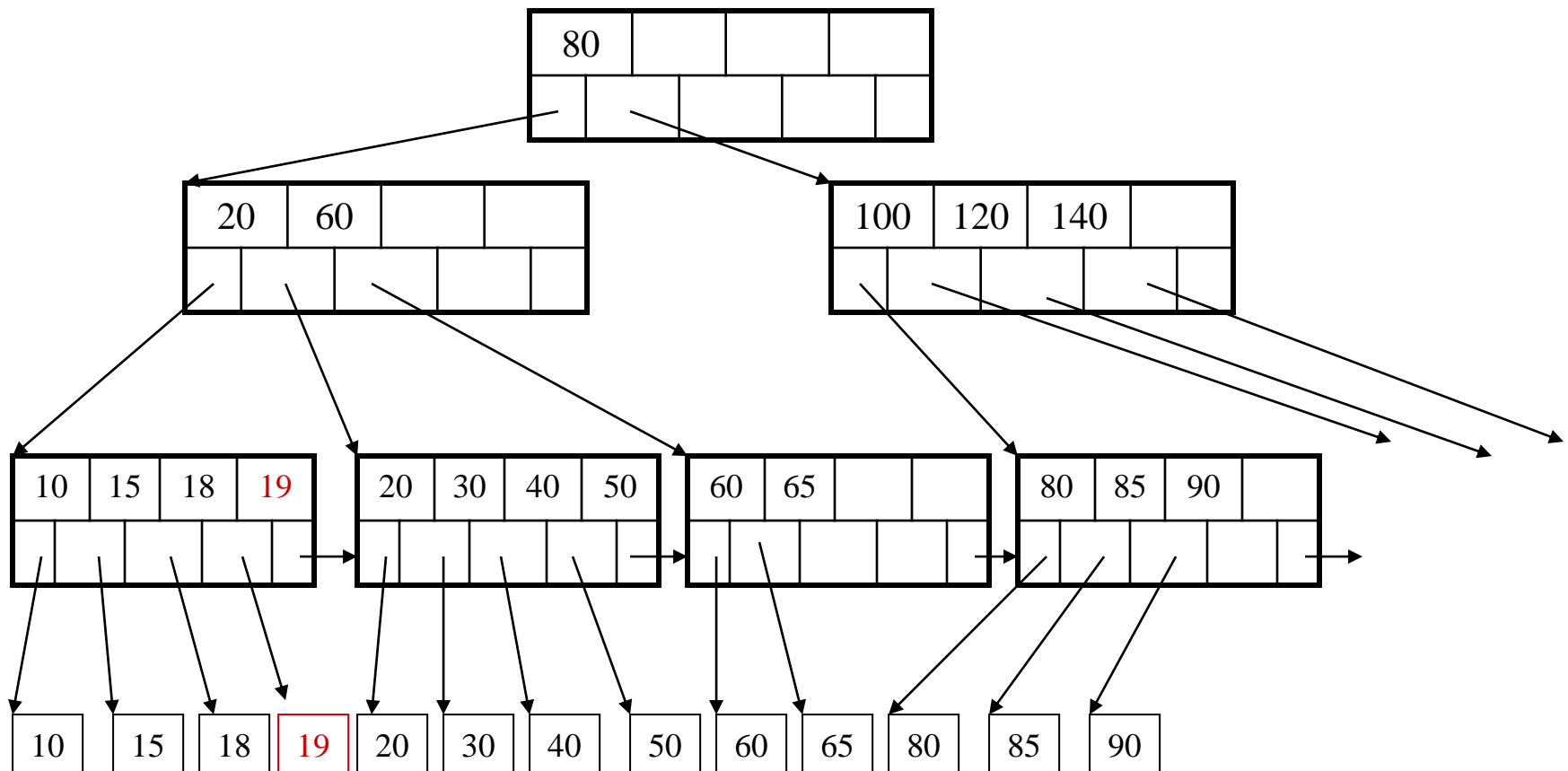
Insertion into a B+ Tree

Insert K=19



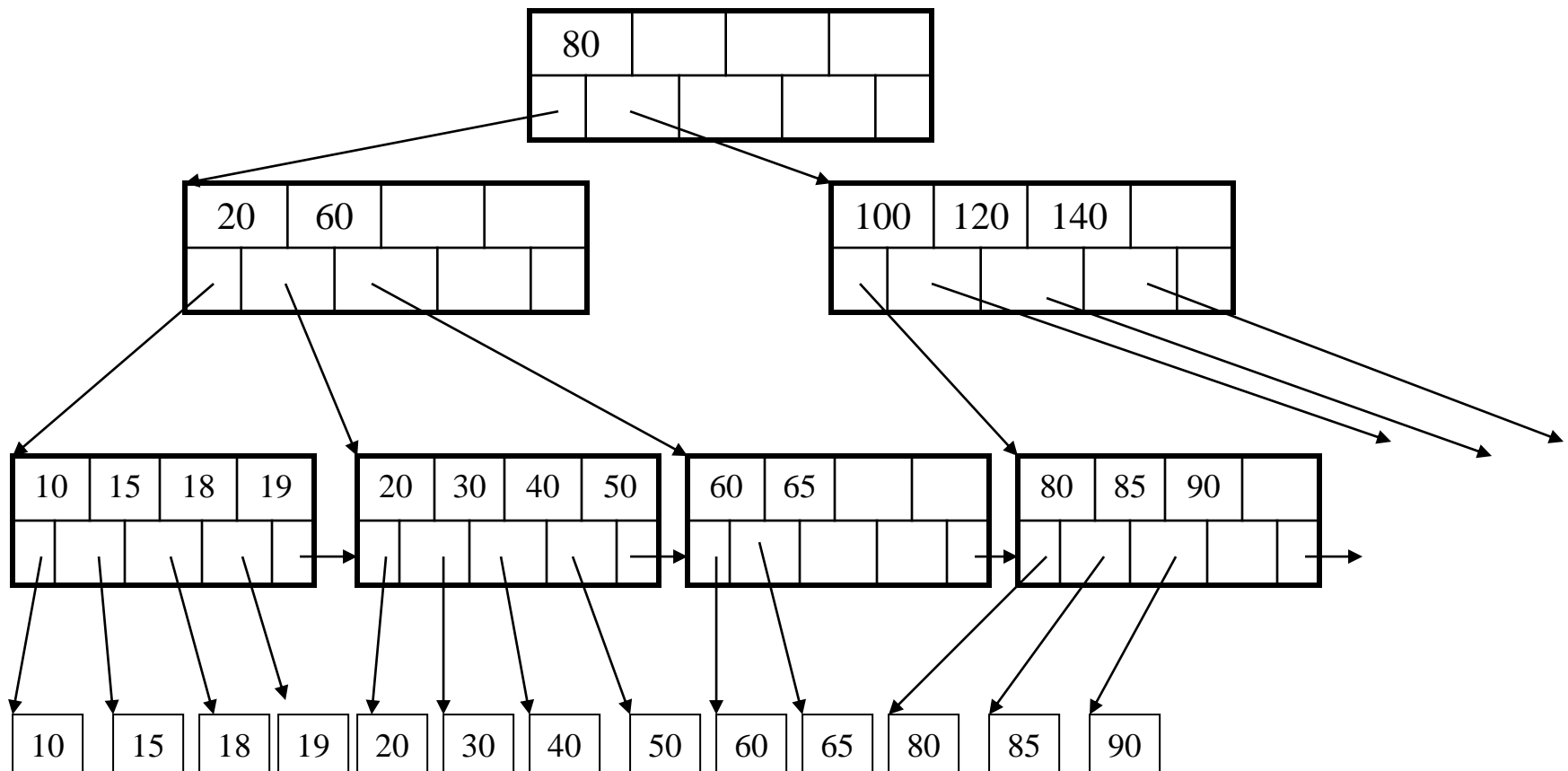
Insertion into a B+ Tree

After insertion



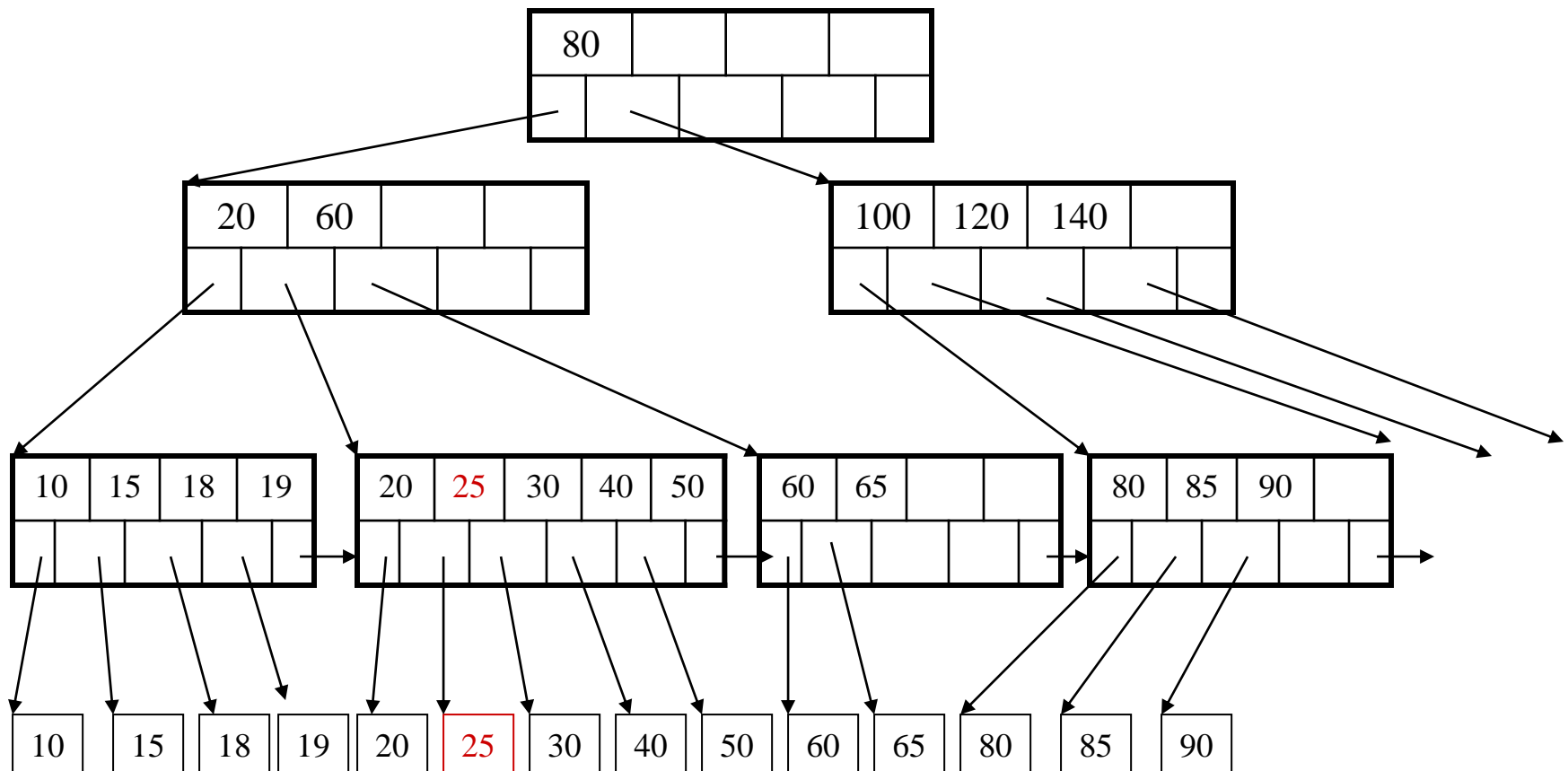
Insertion into a B+ Tree

Now insert 25



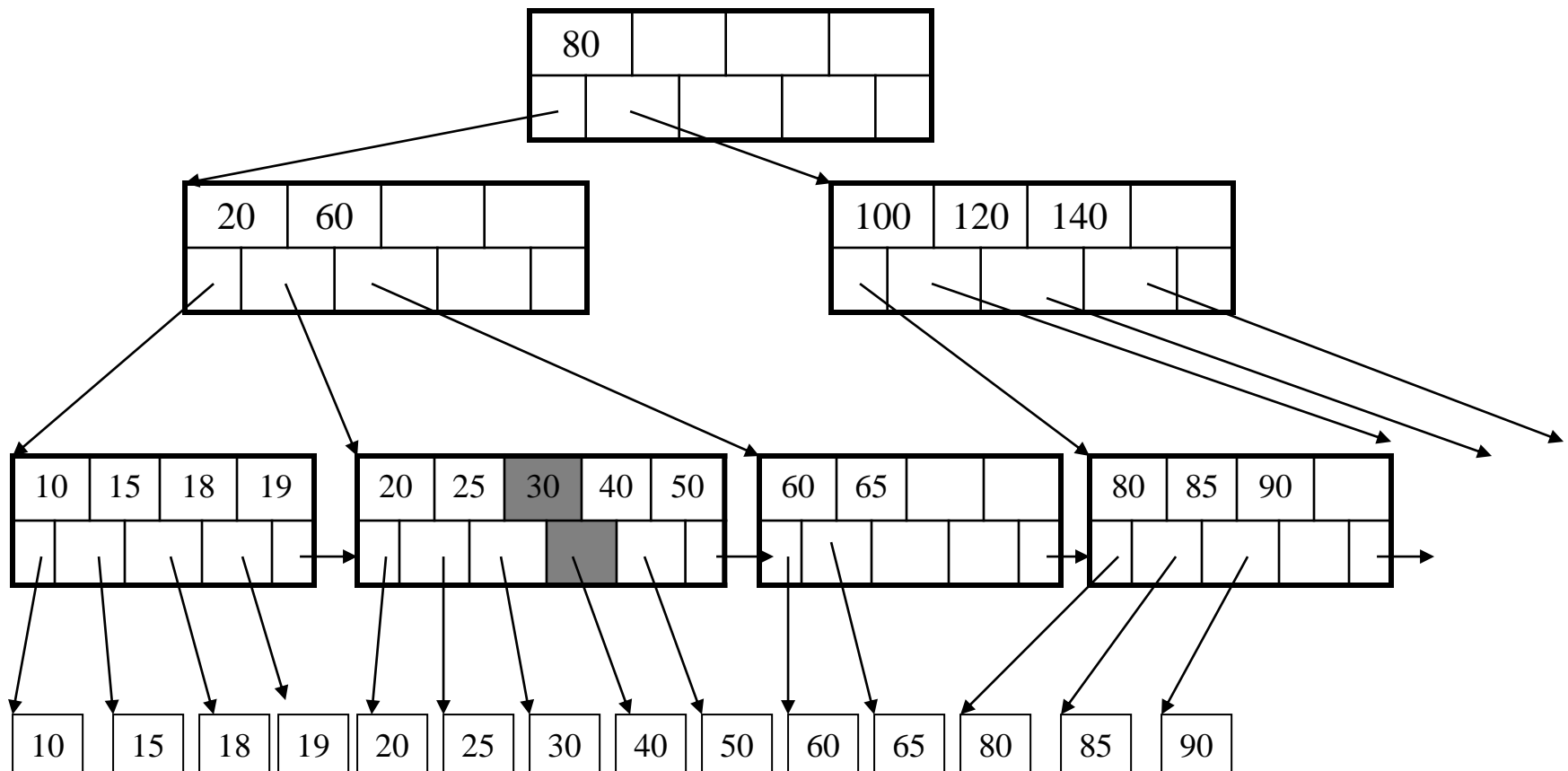
Insertion into a B+ Tree

After insertion



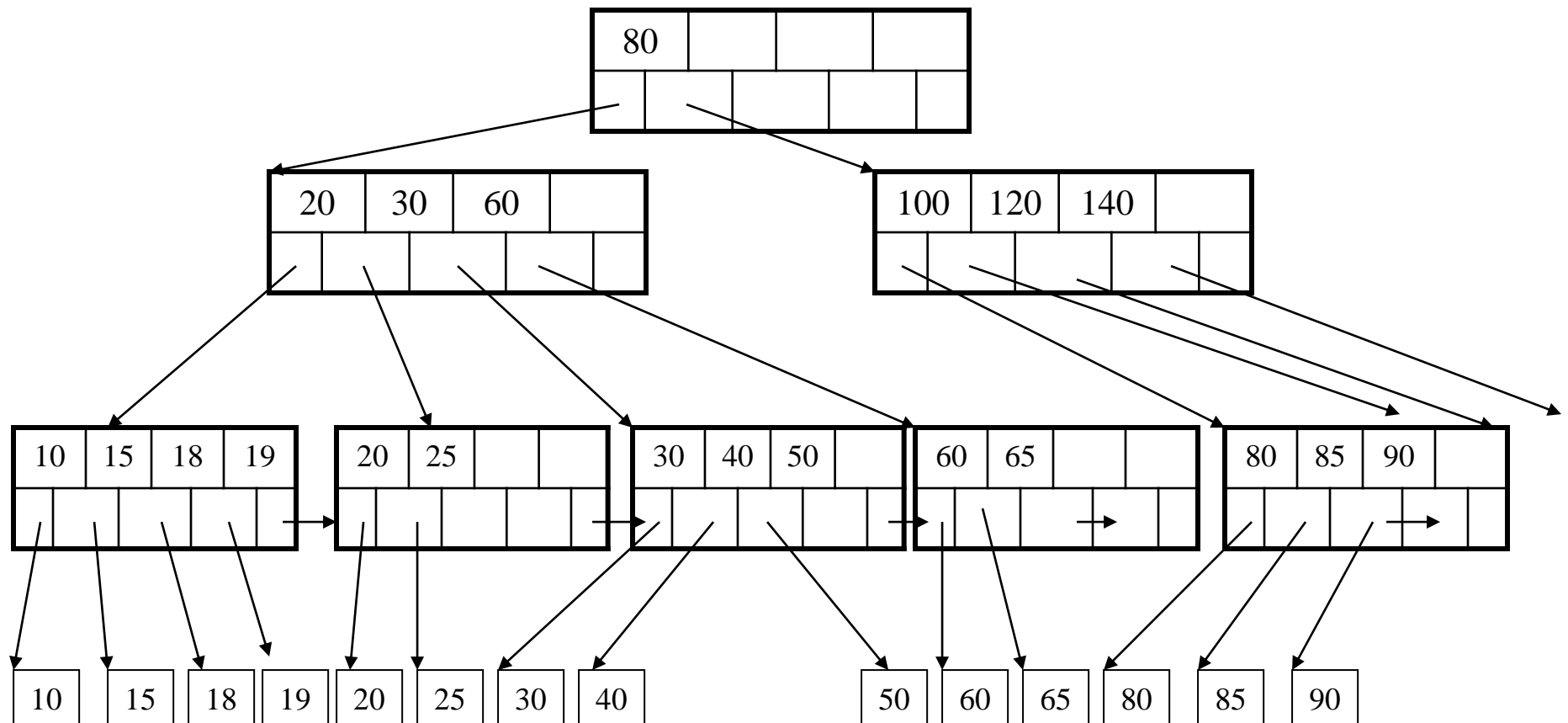
Insertion into a B+ Tree

But now have to split !



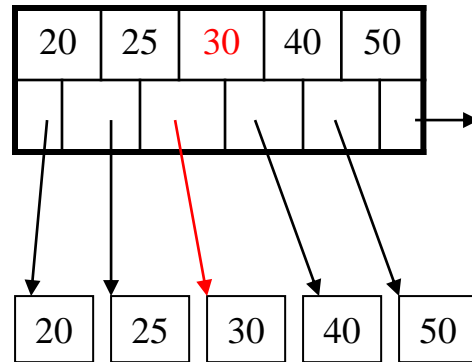
Insertion into a B+ Tree

After the split

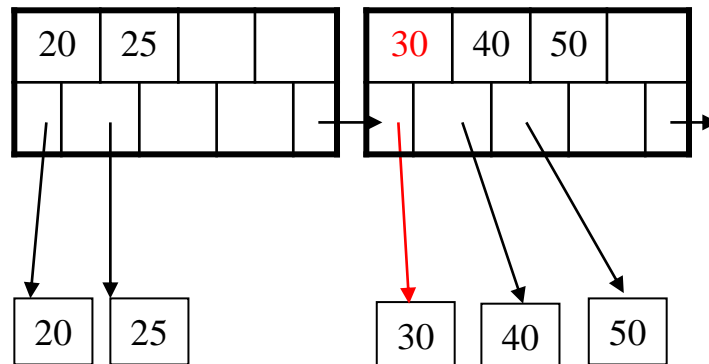


Splitting a leaf node

- Before split: 5 keys, 6 pointers

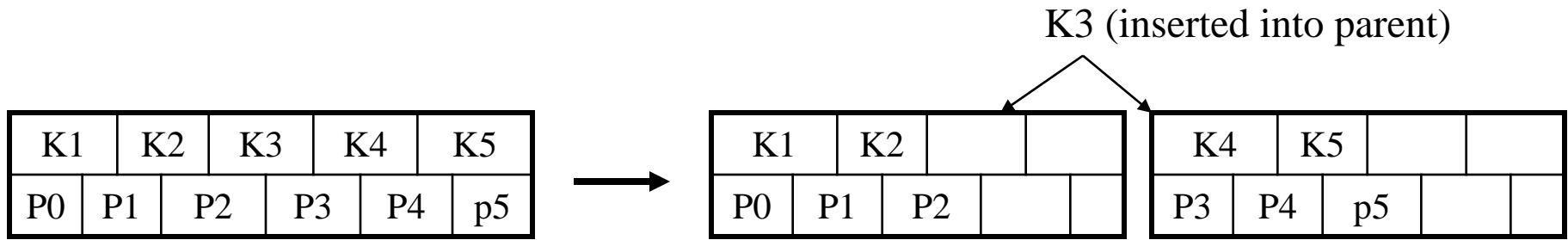


- After split: how many keys and pointers?



Splitting an internal node

- Should not keep the K3 in the right node
 - Unlike splitting a leaf
 - But **why**?



- Count keys & pointers before & after split...

Insertion cascaded...

- Splitting of leaf may lead to splitting of its parent and ancestors

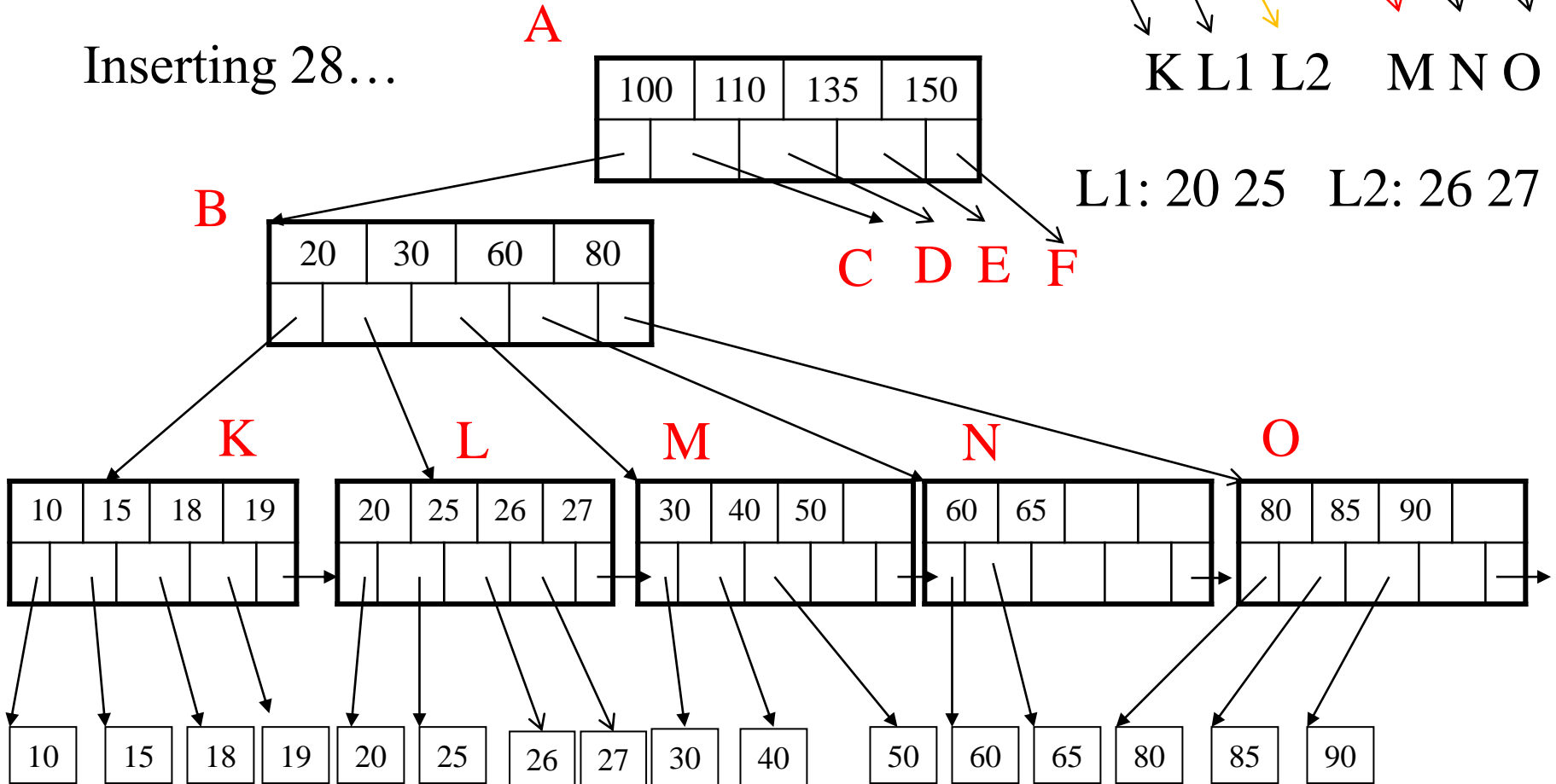
Example

B: 20 26 30 60 80
 ↓ ↓ ↓ ↓ ↓
 K L1 L2 M N O

Inserting 28...

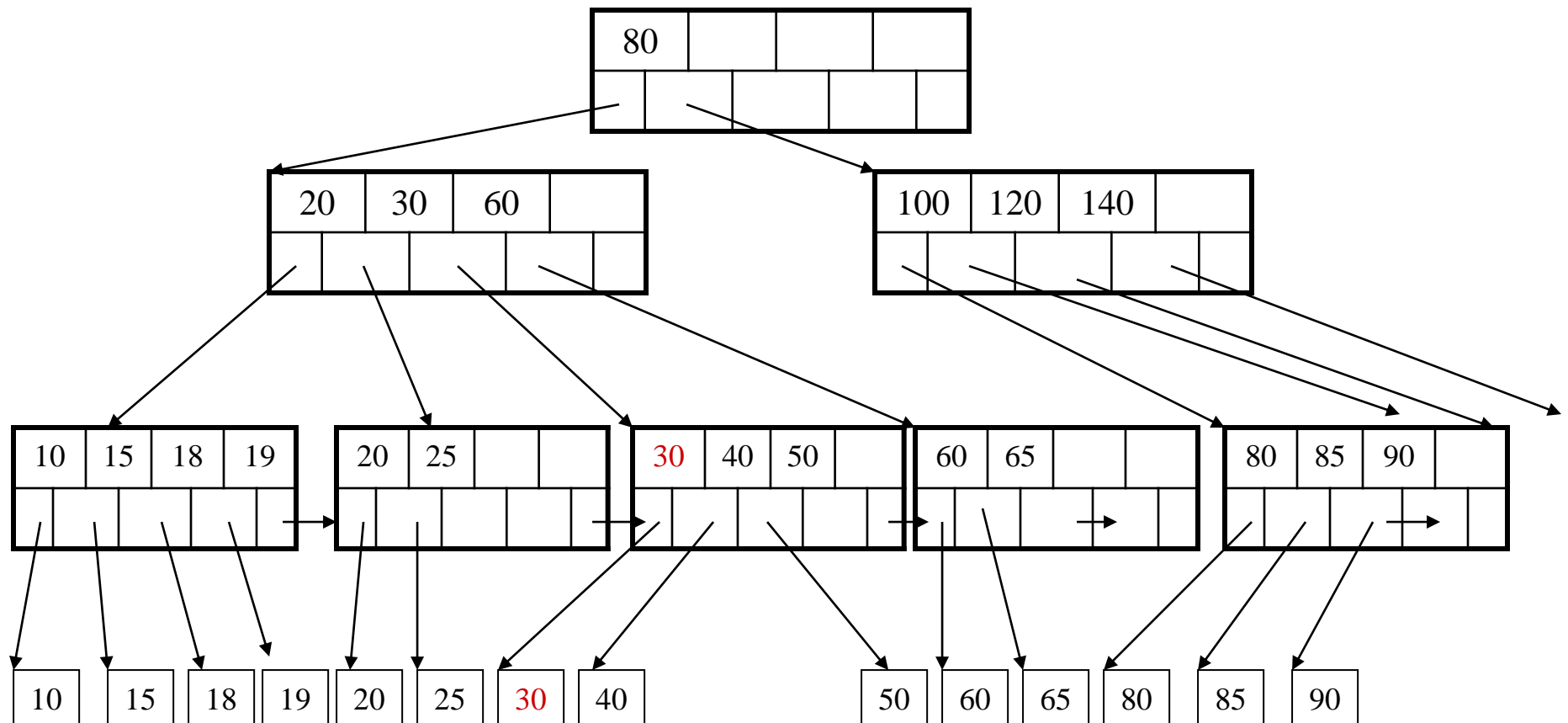
30
 ↙ ↘
 B1: 20 26 B2: 60 80
 ↓ ↓ ↓ ↓ ↓
 K L1 L2 M N O

L1: 20 25 L2: 26 27 28



Deletion from a B+ Tree

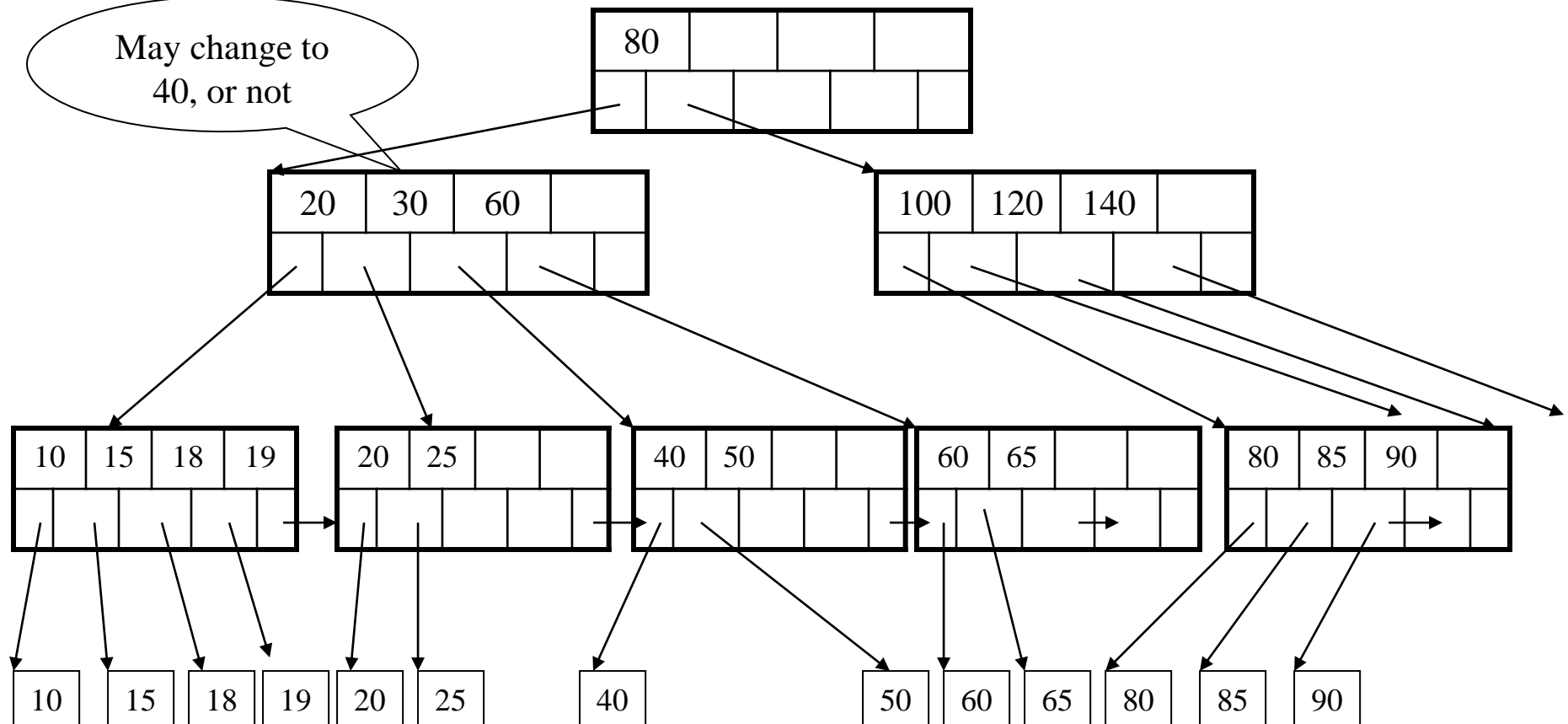
Delete 30



Deletion from a B+ Tree

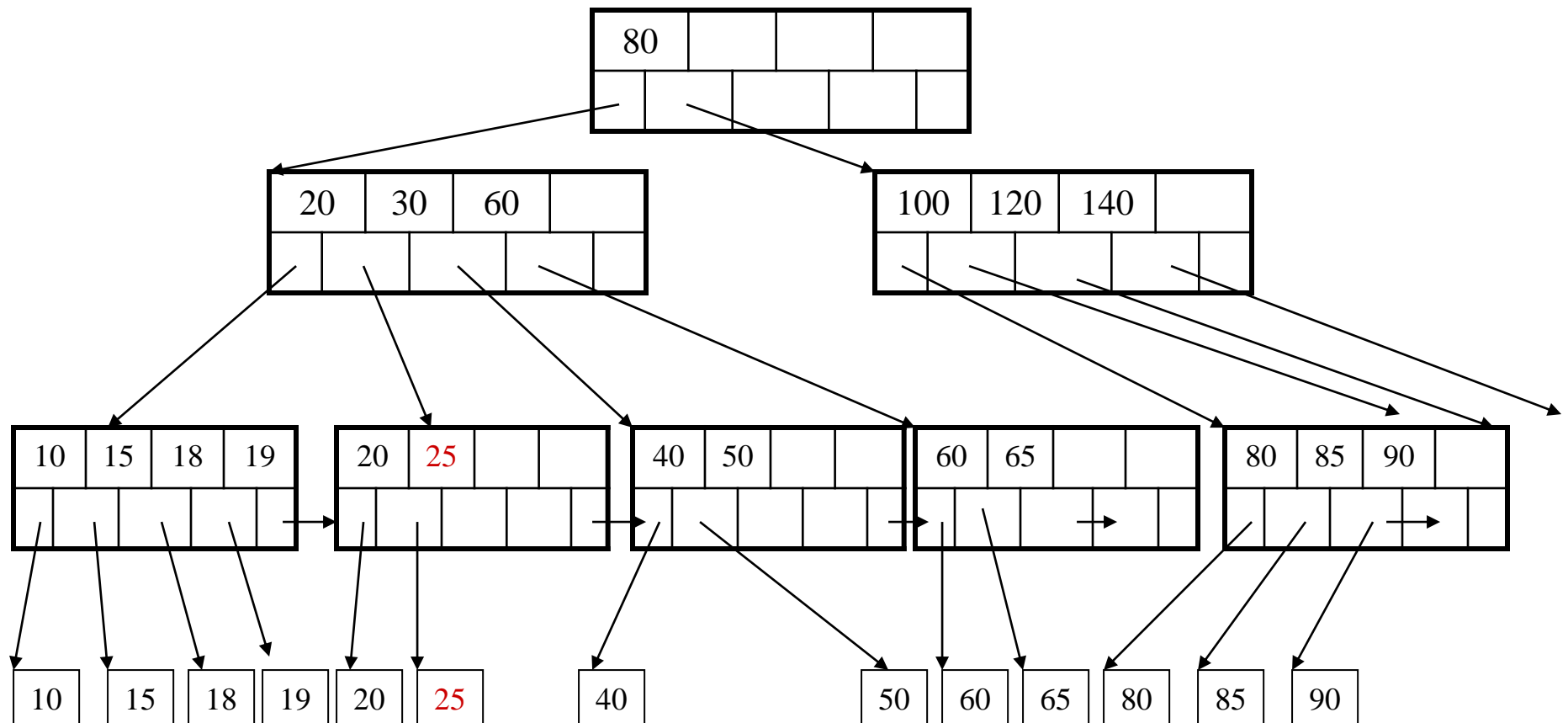
After deleting 30

May change to
40, or not



Deletion from a B+ Tree

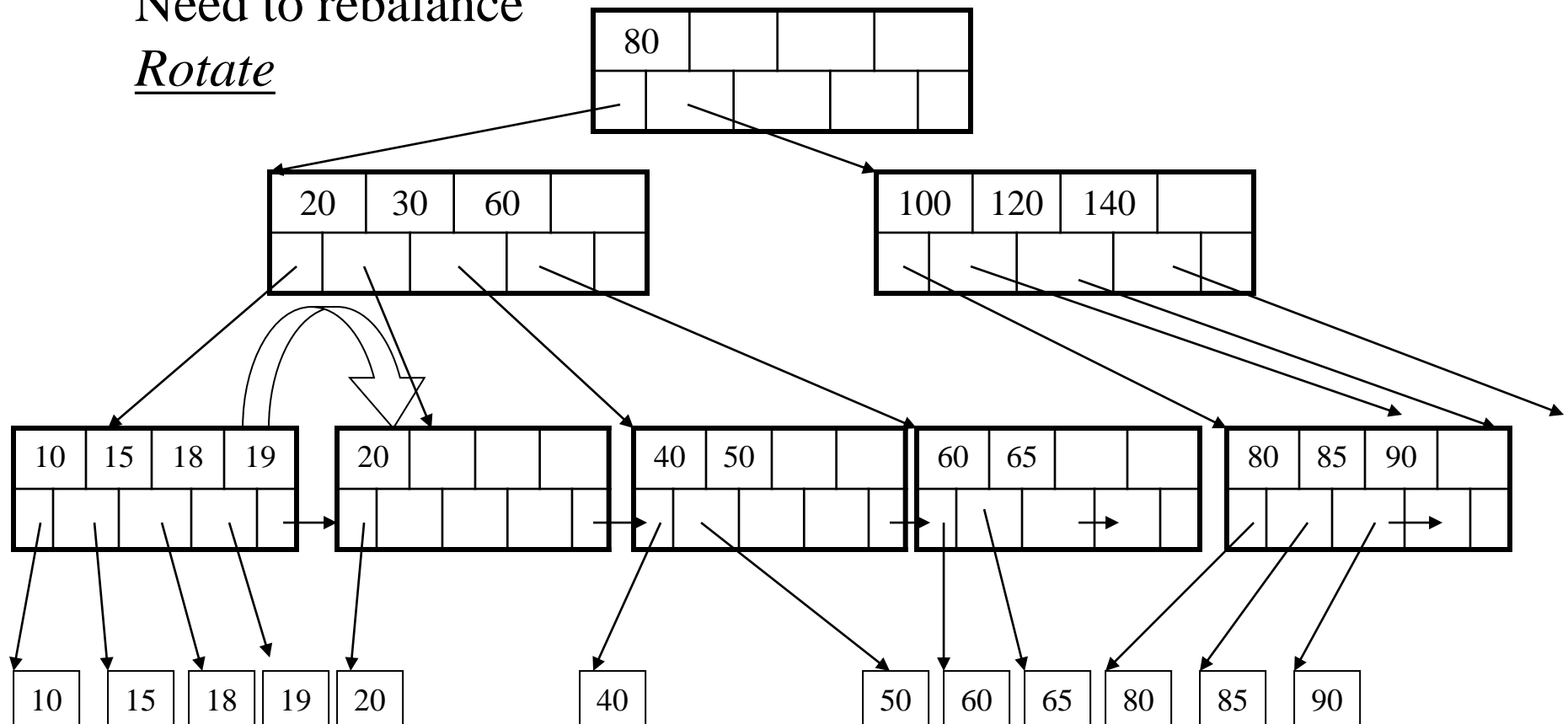
Now delete 25



Deletion from a B+ Tree

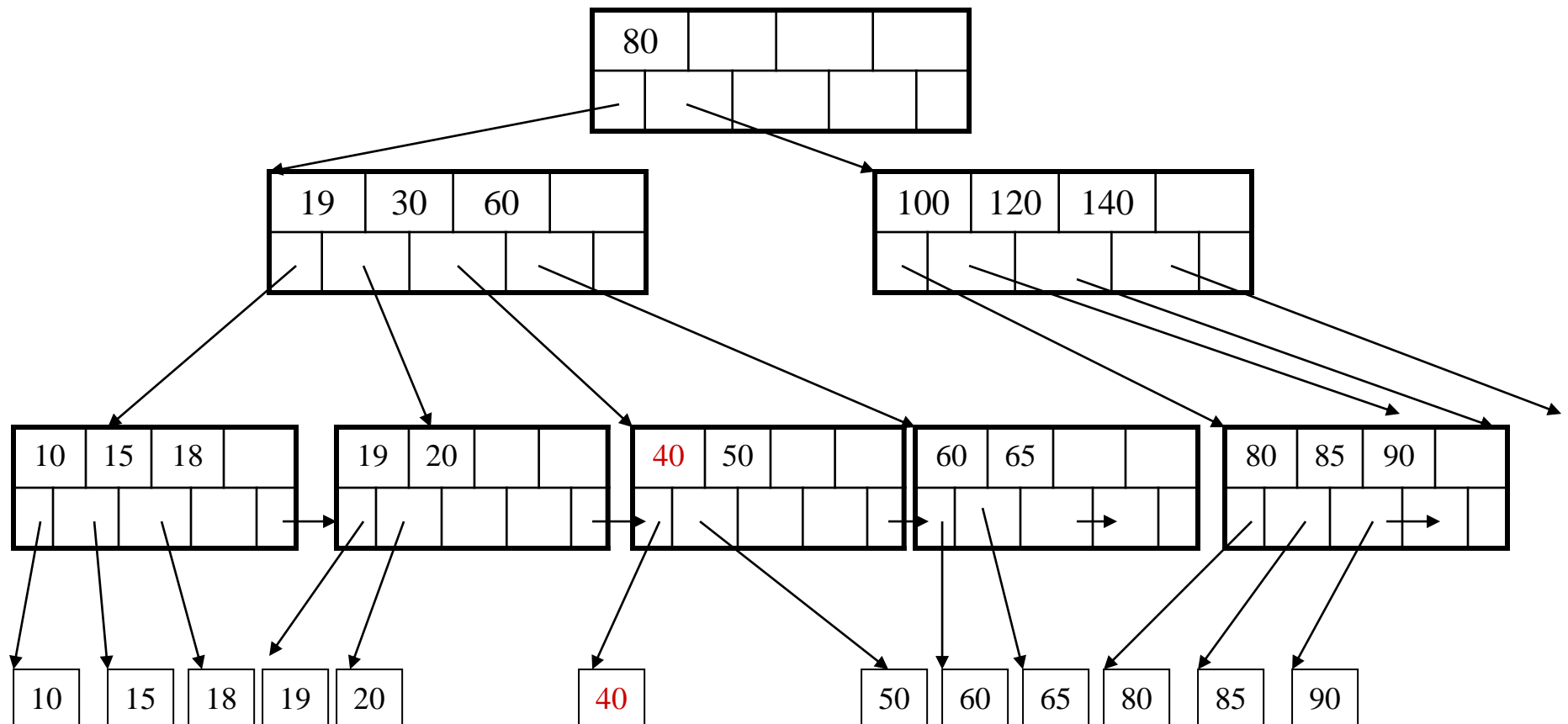
After deleting 25
Need to rebalance

Rotate



Deletion from a B+ Tree

Now delete 40

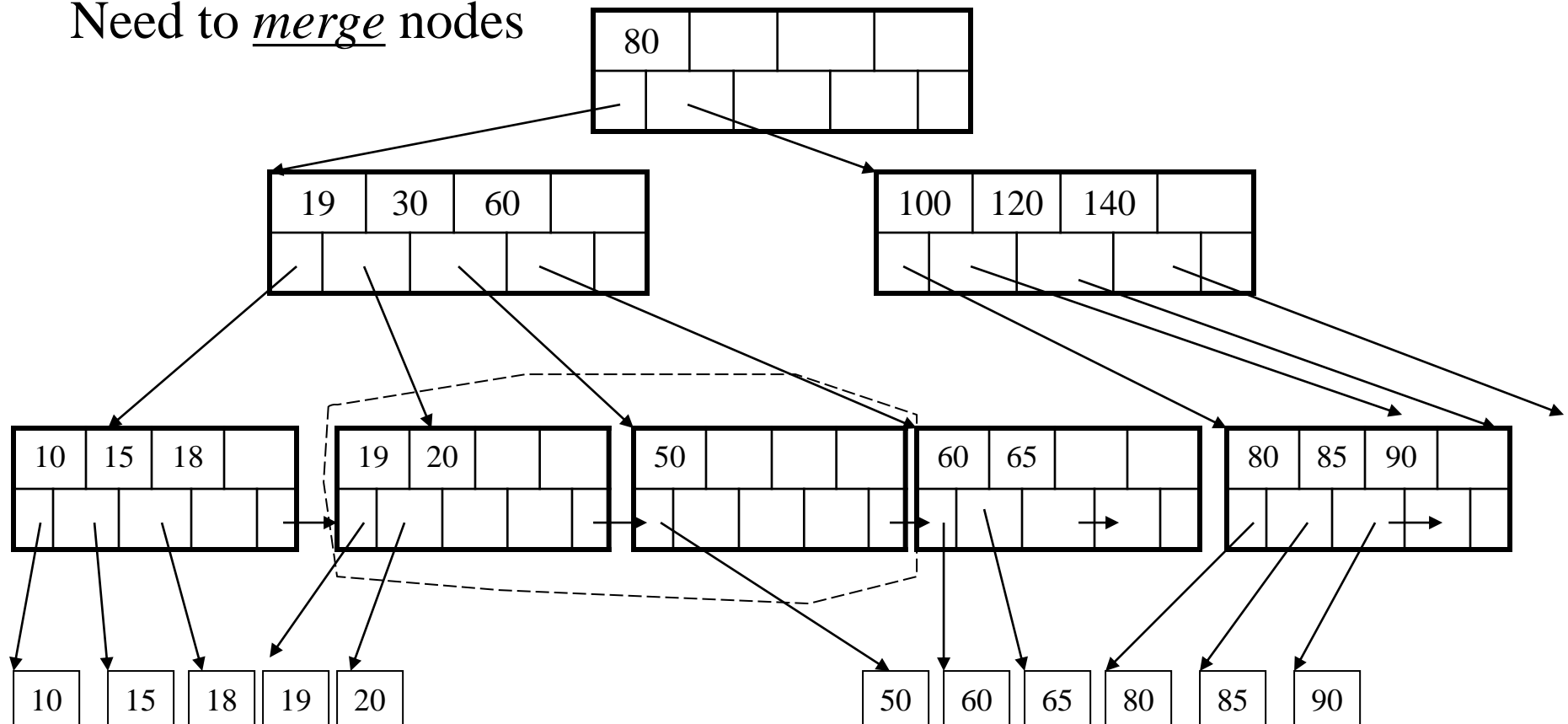


Deletion from a B+ Tree

After deleting 40

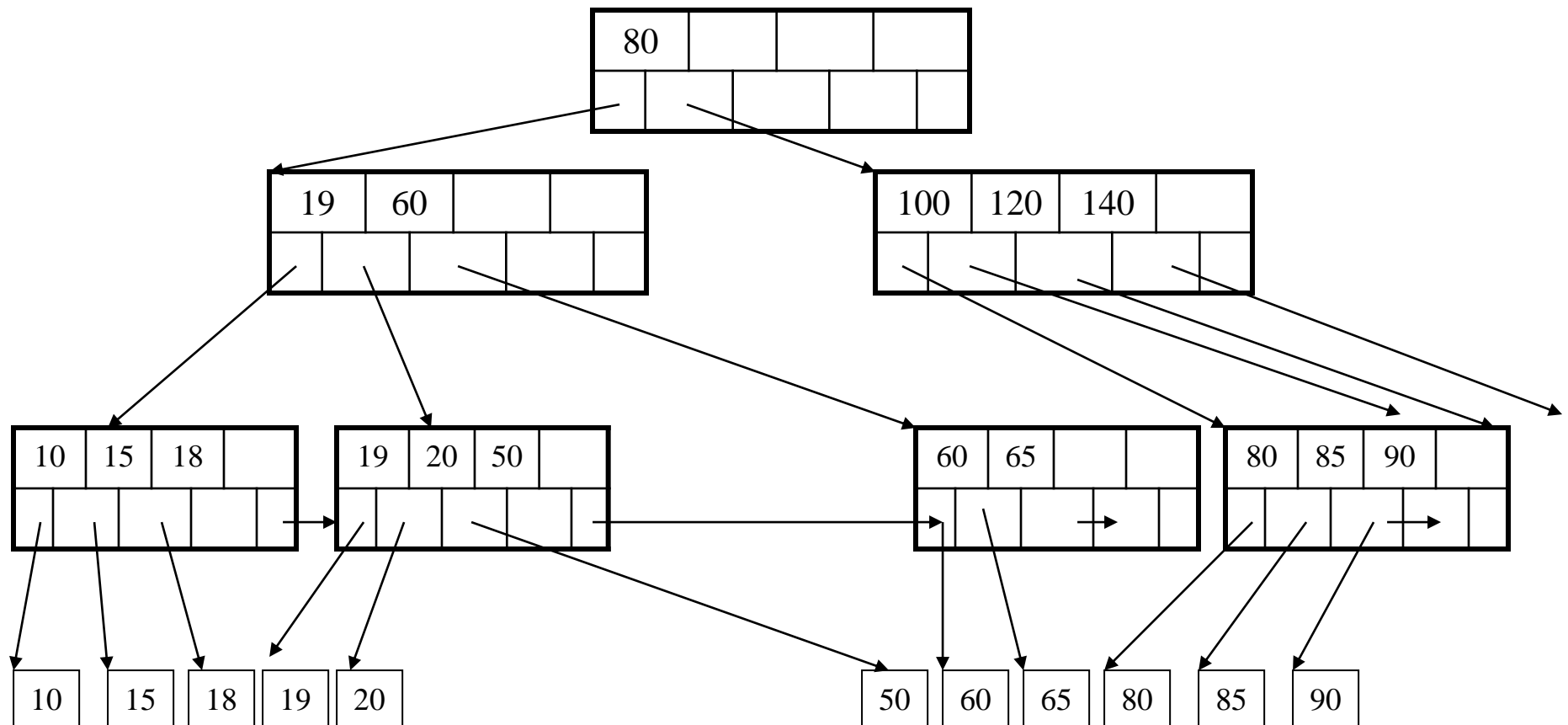
Rotation not possible

Need to merge nodes

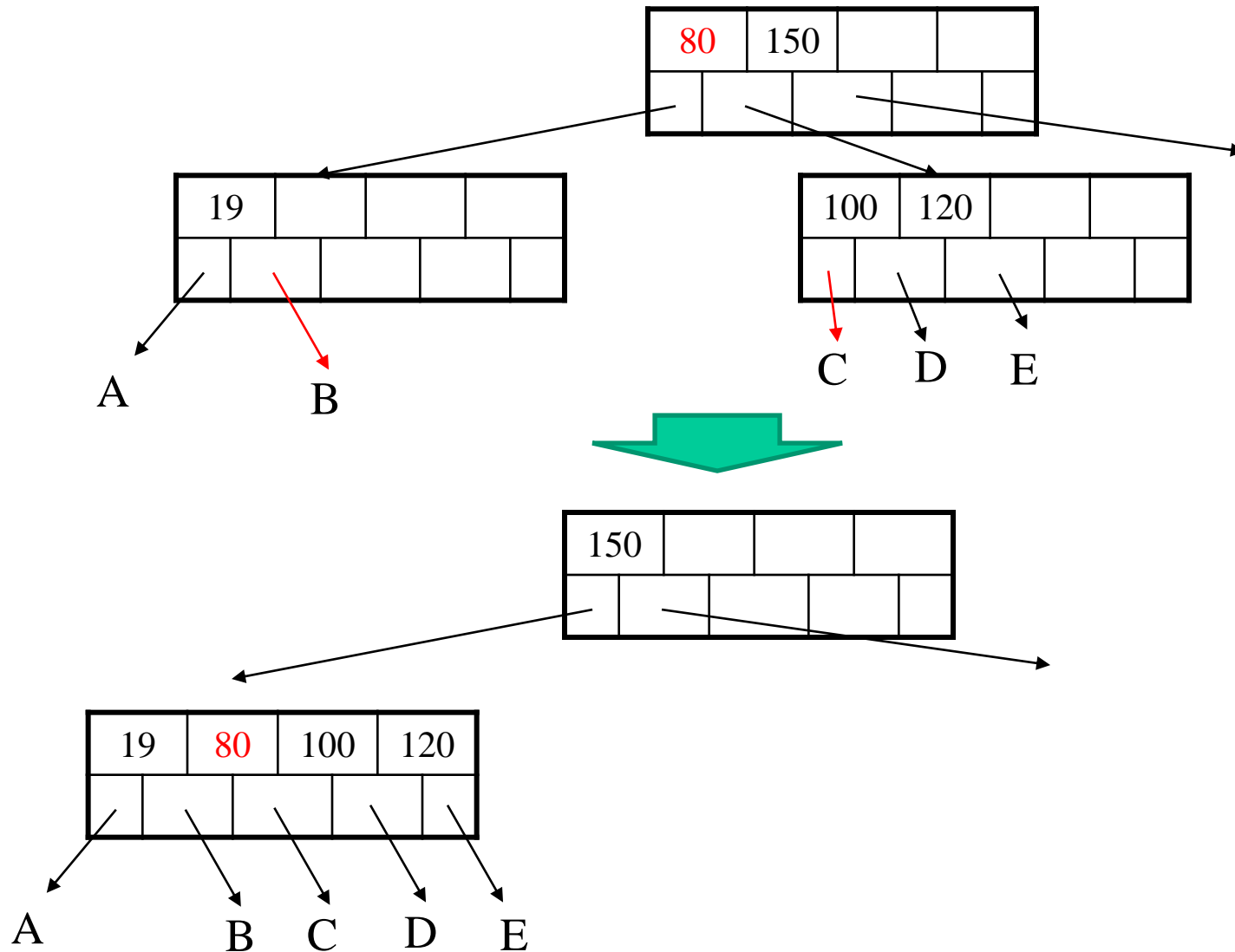


Deletion from a B+ Tree

Final tree



Merging internal nodes...



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

