

# CSE 444: Database Internals

## Lectures 5-6 Indexing

# Announcements

- HW1 due tonight by 11pm
  - Turn in an electronic copy (word/pdf) by 11pm, or
  - Turn in a hard copy after class or during office hour.
- Lab1 is due on Wednesday, 11pm
  - Do not fall behind on labs! Labs build on each other
- 544M first reading due tonight... but flexible
- HW2 has been released
- Monday is a holiday

# Basic Access Method: Heap File

## API

- **Create** or **destroy** a file
- **Insert** a record
- **Delete** a record with a given rid (rid)
  - rid: unique tuple identifier (more later)
- **Get** a record with a given rid
  - Not necessary for sequential scan operator
  - But used with indexes
- **Scan** all records in the file

## But Often Also Want....

- **Scan** all records in the file that match a **predicate** of the form **attribute op value**
  - Example: Find all students with  $\text{GPA} > 3.5$
- Critical to support such requests efficiently
  - Why read all data from disk when we only need a small fraction of that data?
- This lecture and next, we will learn how

# Searching in a Heap File

File is **not sorted** on any attribute

`Student(sid: int, age: int, ...)`

30	18 ...
70	21

— 1 record

20	20
40	19

} 1 page

80	19
60	18

10	21
50	22

# Heap File Search Example

- 10,000 students
- 10 student records per page
- Total number of pages: 1,000 pages
- Find student whose sid is 80
  - Must read on average 500 pages
- Find all students older than 20
  - Must read all 1,000 pages
- Can we do better?

# Sequential File

File **sorted on an attribute**, usually on primary key

`Student(sid: int, age: int, ...)`

10	21 ...
20	20

30	18
40	19

50	22
60	18

70	21
80	19

# Sequential File Example

- Total number of pages: 1,000 pages
- Find student whose sid is 80
  - Could do binary search, read  $\log_2(1,000) \approx 10$  pages
- Find all students older than 20
  - Must still read all 1,000 pages
- Can we do even better?
- Note: Sorted files are inefficient for inserts/deletes

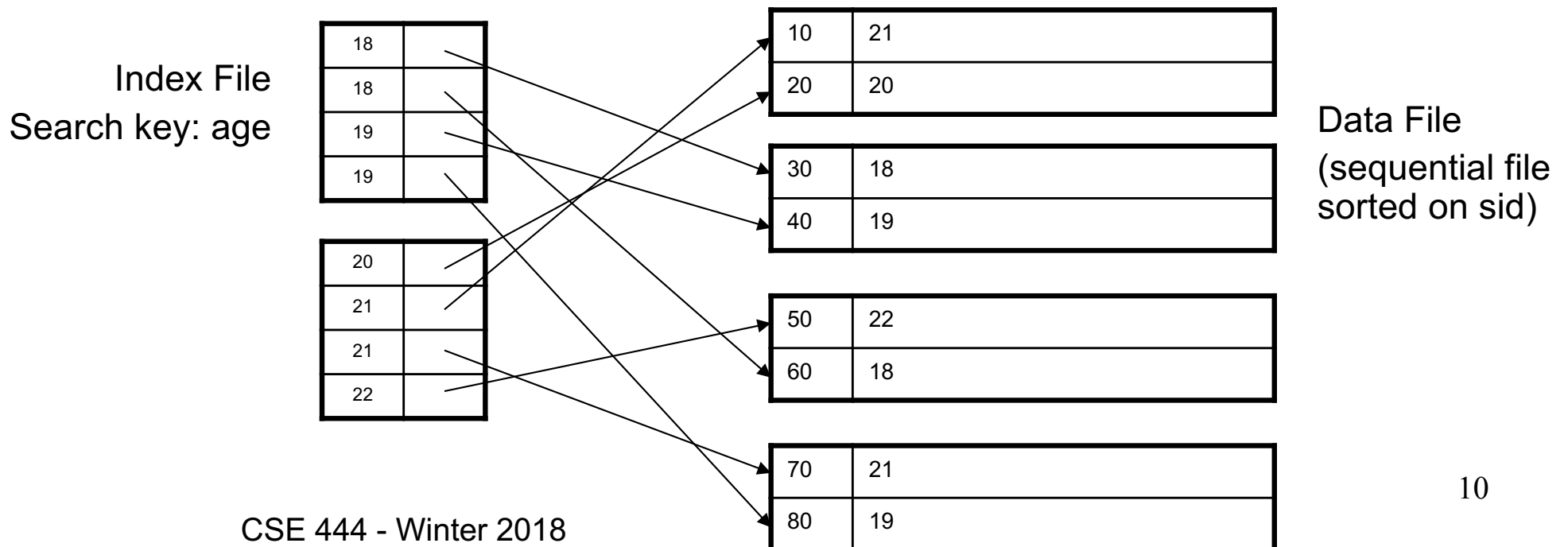


# Outline

- Index structures
  - Hash-based indexes
  - B+ trees
- } Today
- } Next time

# Indexes

- **Index**: data structure that organizes data records on disk to optimize selections on the **search key fields** for the index
- An index contains a collection of **data entries**, and supports efficient retrieval of all data entries with a given search key value **k**
- Indexes are also access methods!
  - So they provide the same API as we have seen for Heap Files
  - And efficiently support scans over tuples matching predicate on search key



# Indexes

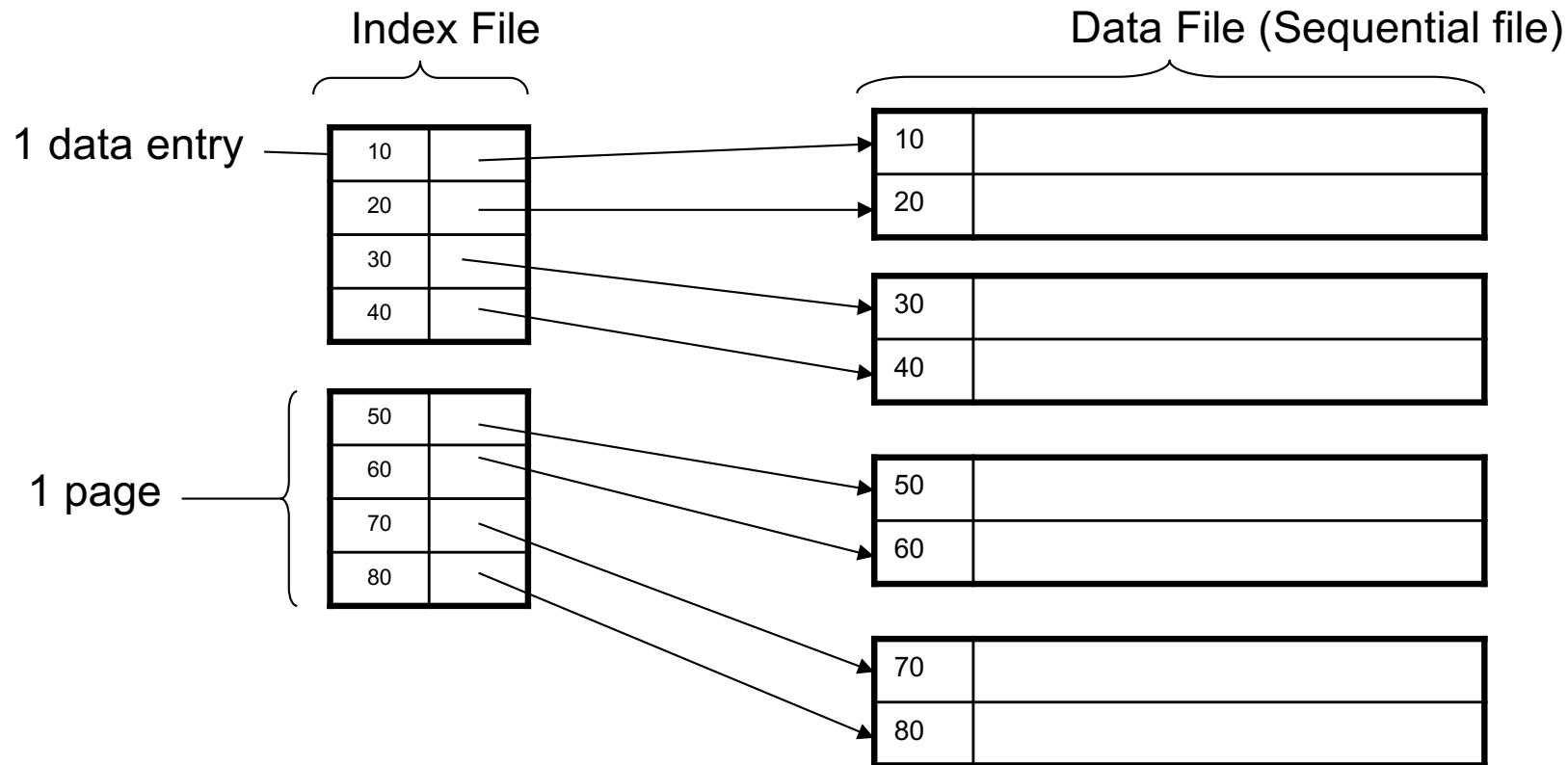
- **Search key** = can be any set of fields
  - not the same as the primary key, nor a key
- **Index** = collection of data entries
- **Data entry** for key  $k$  can be:
  - $(k, \text{RID})$
  - $(k, \text{list-of-RIDs})$
  - The actual record with key  $k$ 
    - In this case, **the index is also a special file organization**
    - Called: “indexed file organization”

# Different Types of Files

- For the data inside base relations:
  - **Heap file** (tuples stored without any order)
  - **Sequential file** (tuples sorted on some attribute(s))
  - **Indexed file** (tuples organized following an index)
- Then we can have additional **index files** that store (key,rid) pairs
- Index can also be a “**covering index**”
  - Index contains (search key + other attributes, rid)
  - Index suffices to answer some queries

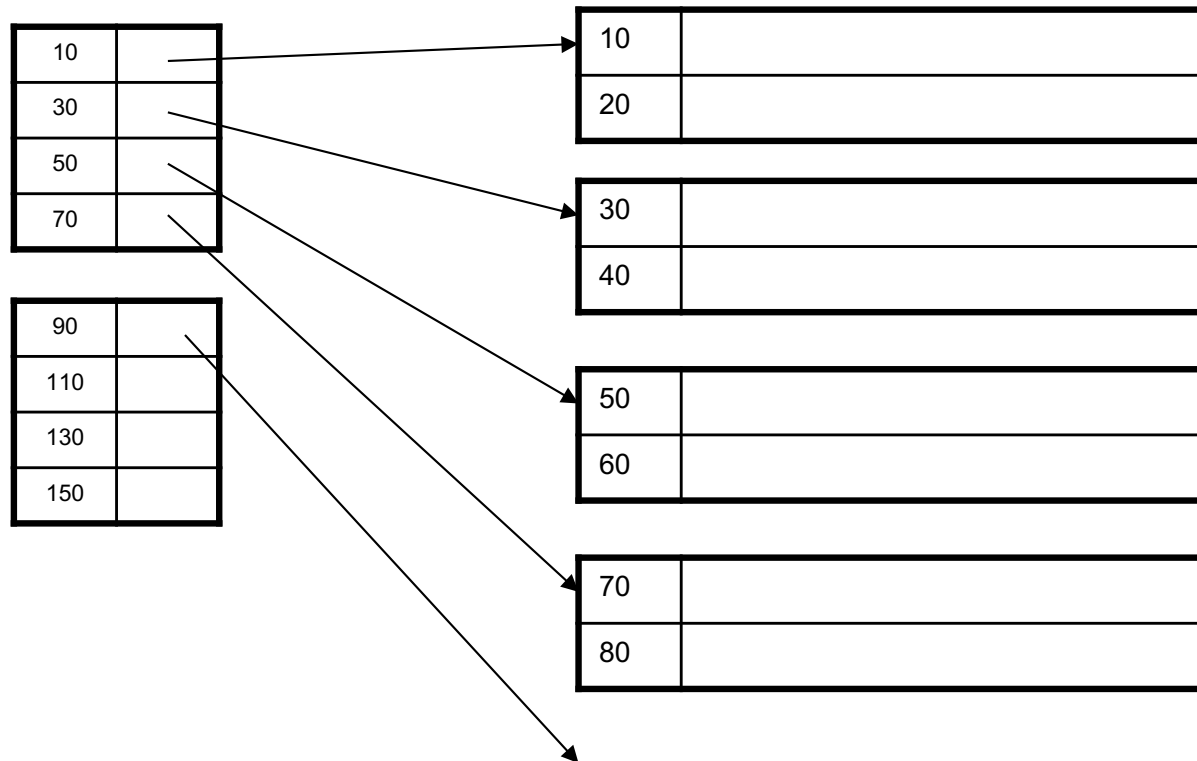
# Primary Index

- Primary index determines location of indexed records
- Dense index: sequence of (key,rid) pairs



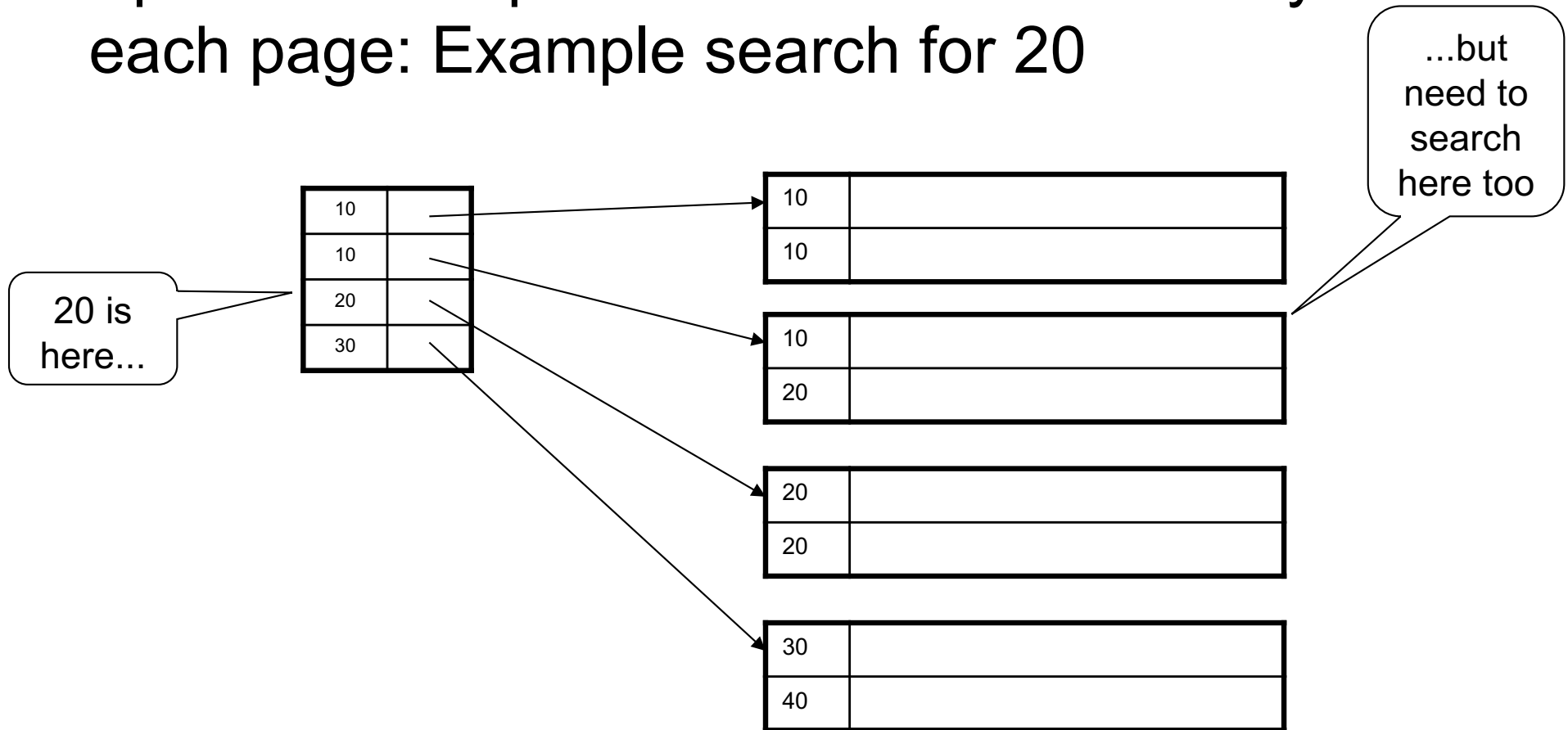
# Primary Index

- Sparse index



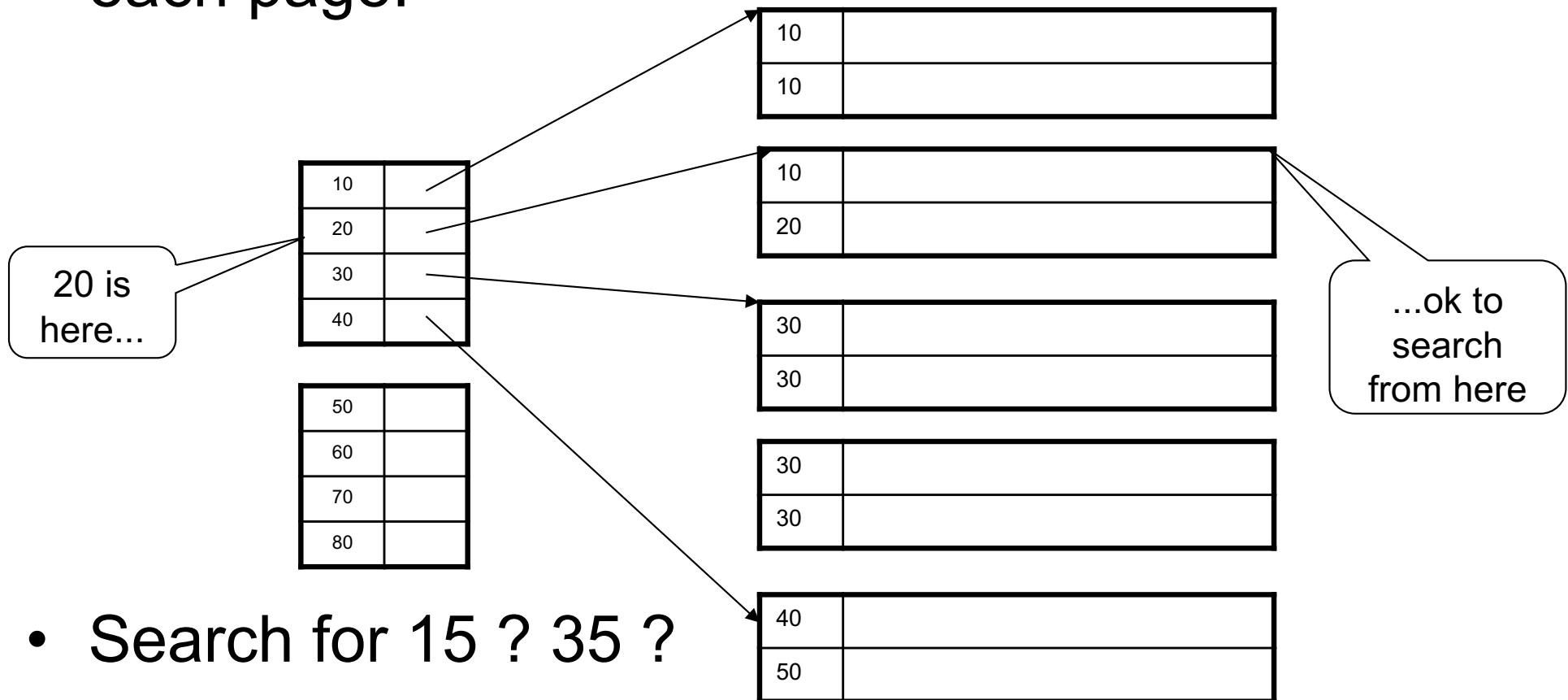
# Primary Index with Duplicate Keys

- Sparse index: pointer to lowest search key on each page: Example search for 20



# Primary Index with Duplicate Keys

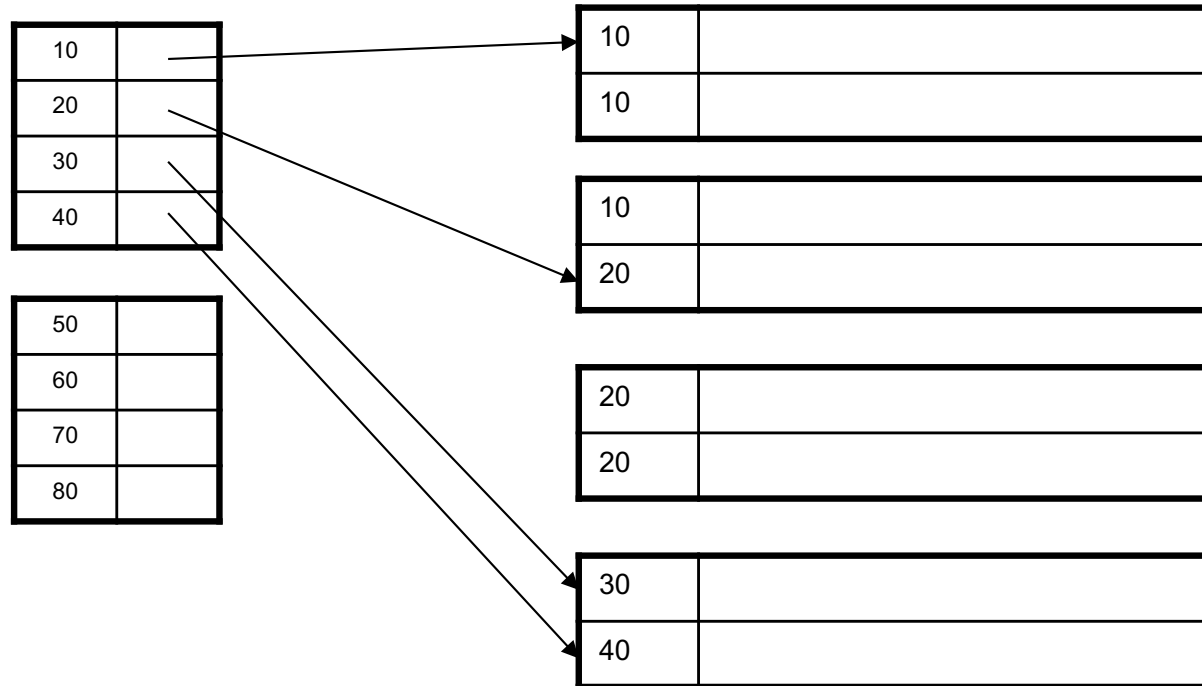
- Better: pointer to ***lowest new search key*** on each page:





# Primary Index with Duplicate Keys

- Dense index:

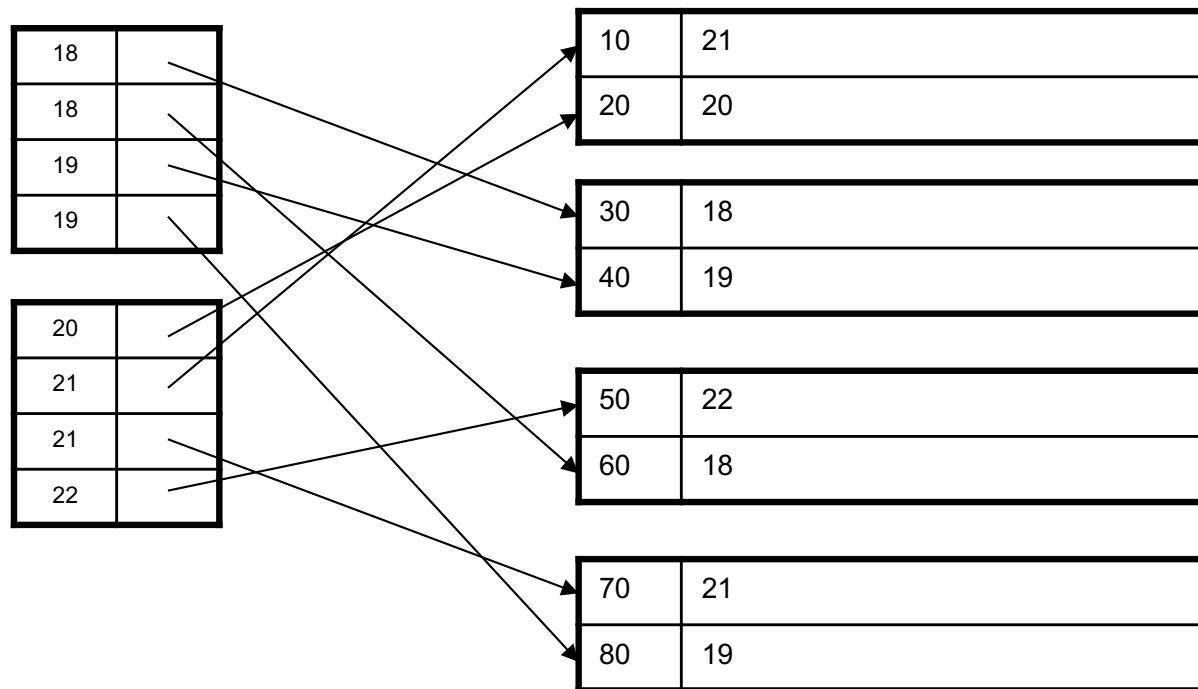


# Primary Index: Back to Example

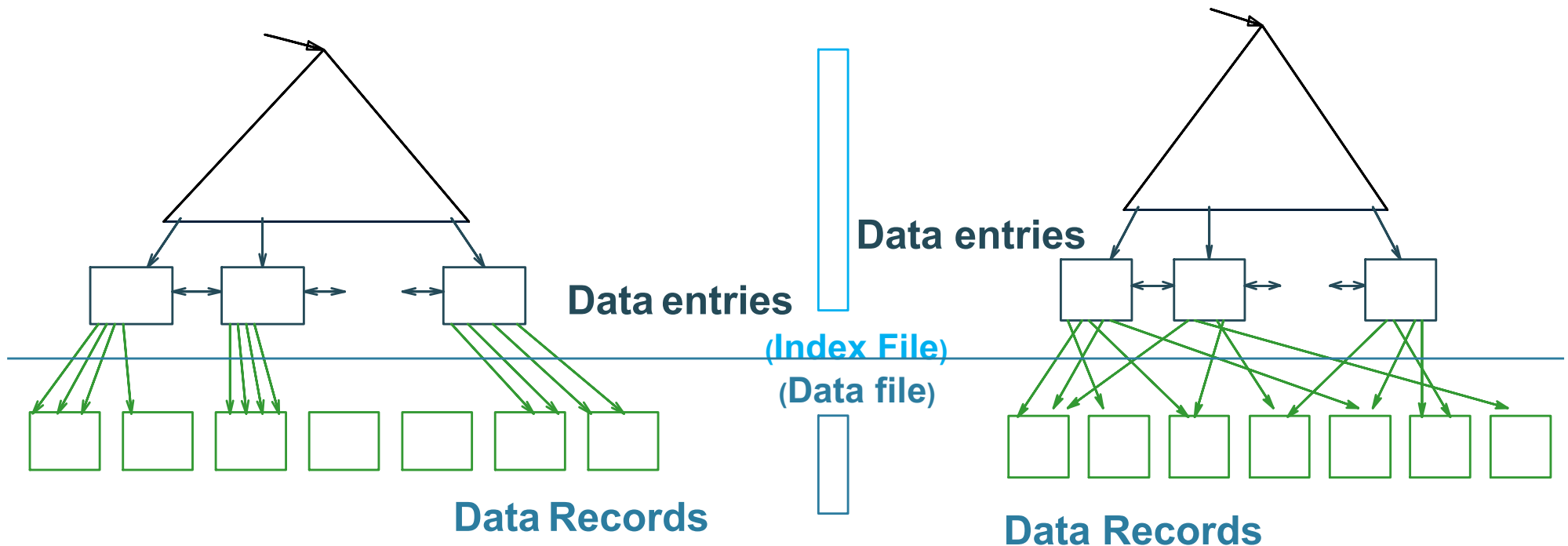
- Let's assume all pages of index fit in memory
- Find student whose sid is 80
  - Index (dense or sparse) points directly to the page
  - Only need to read 1 page from disk.
- Find all students older than 20
  - Must still read all 1,000 pages.
- How can we make *both* queries fast?

# Secondary Indexes

- Do not determine placement of records in data files
- Always dense (why ?)



# Clustered vs. Unclustered Index



**CLUSTERED**

**UNCLUSTERED**

Clustered = records close in index are close in data

# Clustered/Unclustered

- Primary index = clustered by definition
- Secondary indexes = usually unclustered

# Secondary Indexes

- Applications
  - Index unsorted files (heap files)
  - When necessary to have multiple indexes
  - Index files that hold data from two relations
    - Called “clustered file”
    - Notice the different use of the term “clustered”!

# Index Classification Summary

- Primary/secondary
  - Primary = determines the location of indexed records
  - Secondary = cannot reorder data, does not determine data location
- Dense/sparse
  - Dense = every key in the data appears in the index
  - Sparse = the index contains only some keys
- Clustered/unclustered
  - Clustered = records close in index are close in data
  - Unclustered = records close in index may be far in data
- B+ tree / Hash table / ...

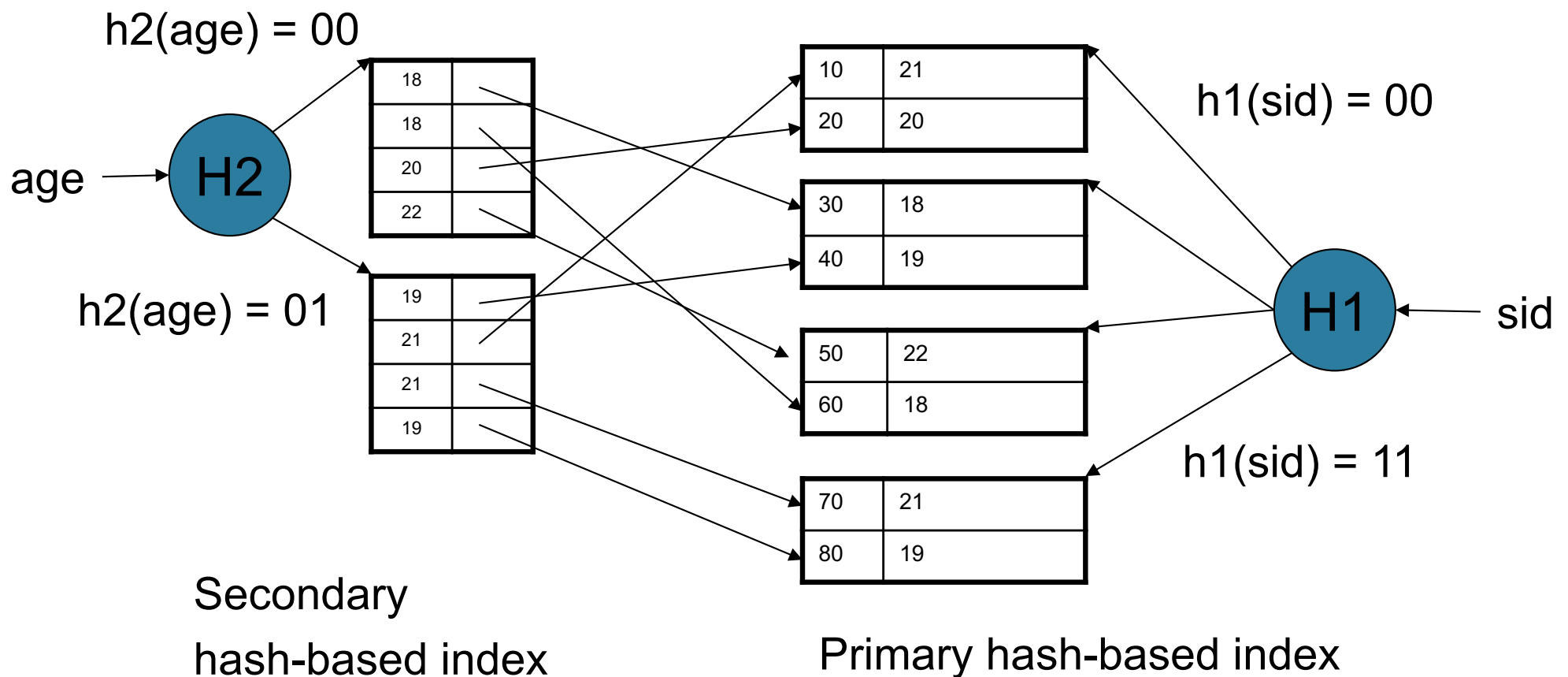
# Large Indexes

- What if index does not fit in memory?
- Would like to index the index itself
  - Hash-based index
  - Tree-based index



# Hash-Based Index

Good for point queries but not range queries



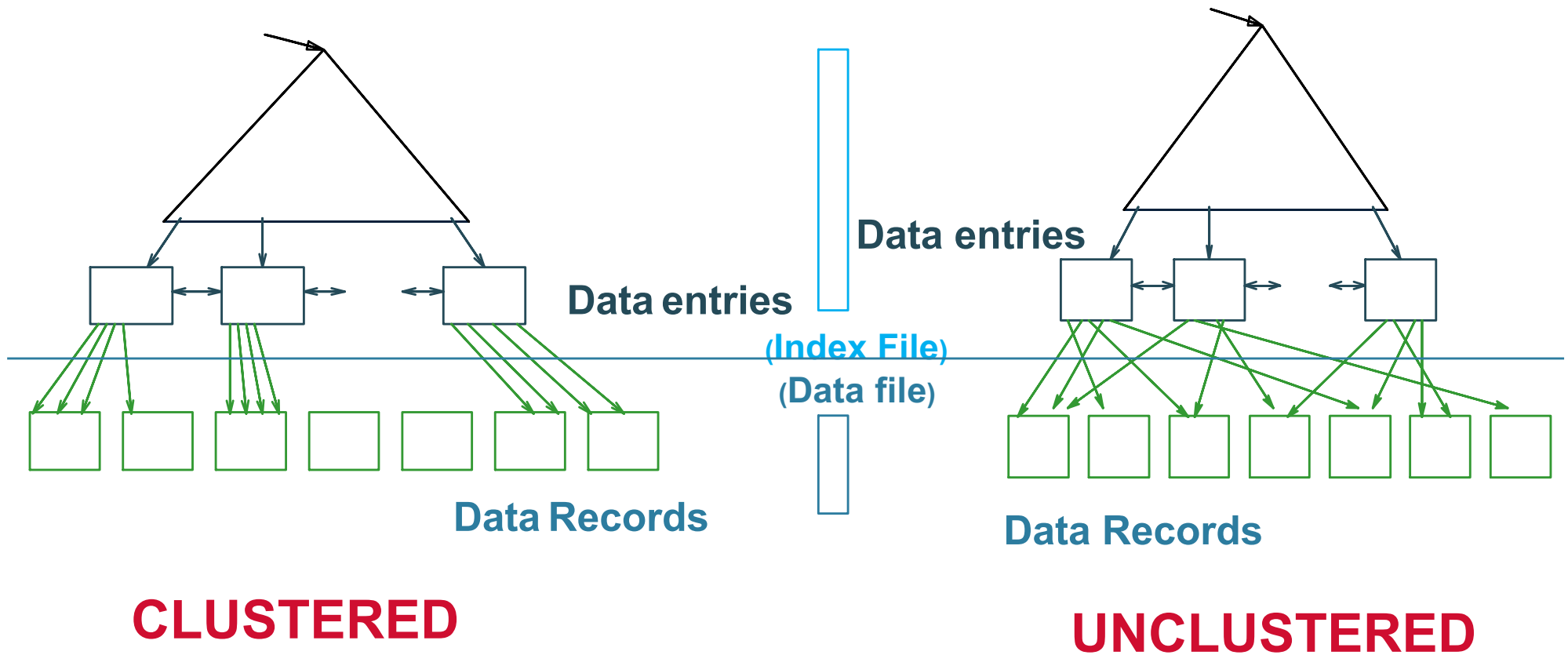
# Tree-Based Index

- How many index levels do we need?
- Can we create them automatically? **Yes!**
- **Can do something even more powerful!**

# B+ Trees

- Search trees
- Idea in B Trees
  - Make 1 node = 1 page (= 1 block)
  - Keep tree balanced in height
- Idea in B+ Trees
  - Make leaves into a linked list : facilitates range queries

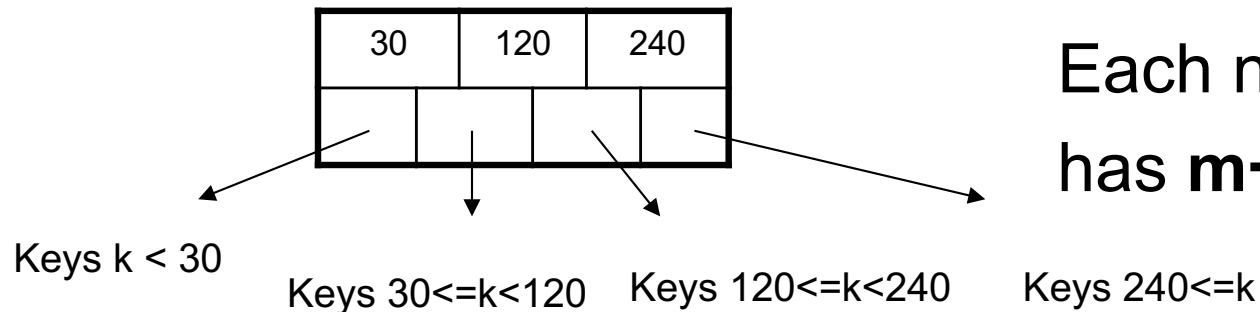
# B+ Trees



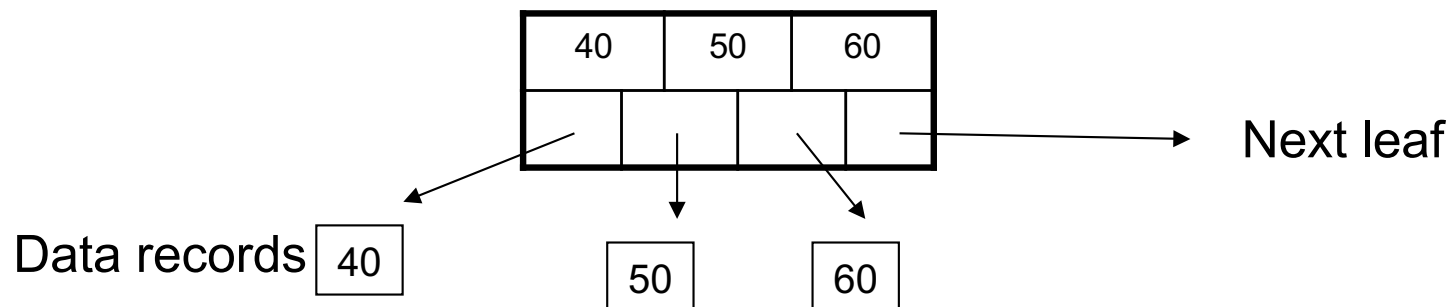
Note: can also store data records directly as data entries

# B+ Trees Basics

- Parameter **d** = the degree
- Each node has  **$d \leq m \leq 2d$  keys** (except root)



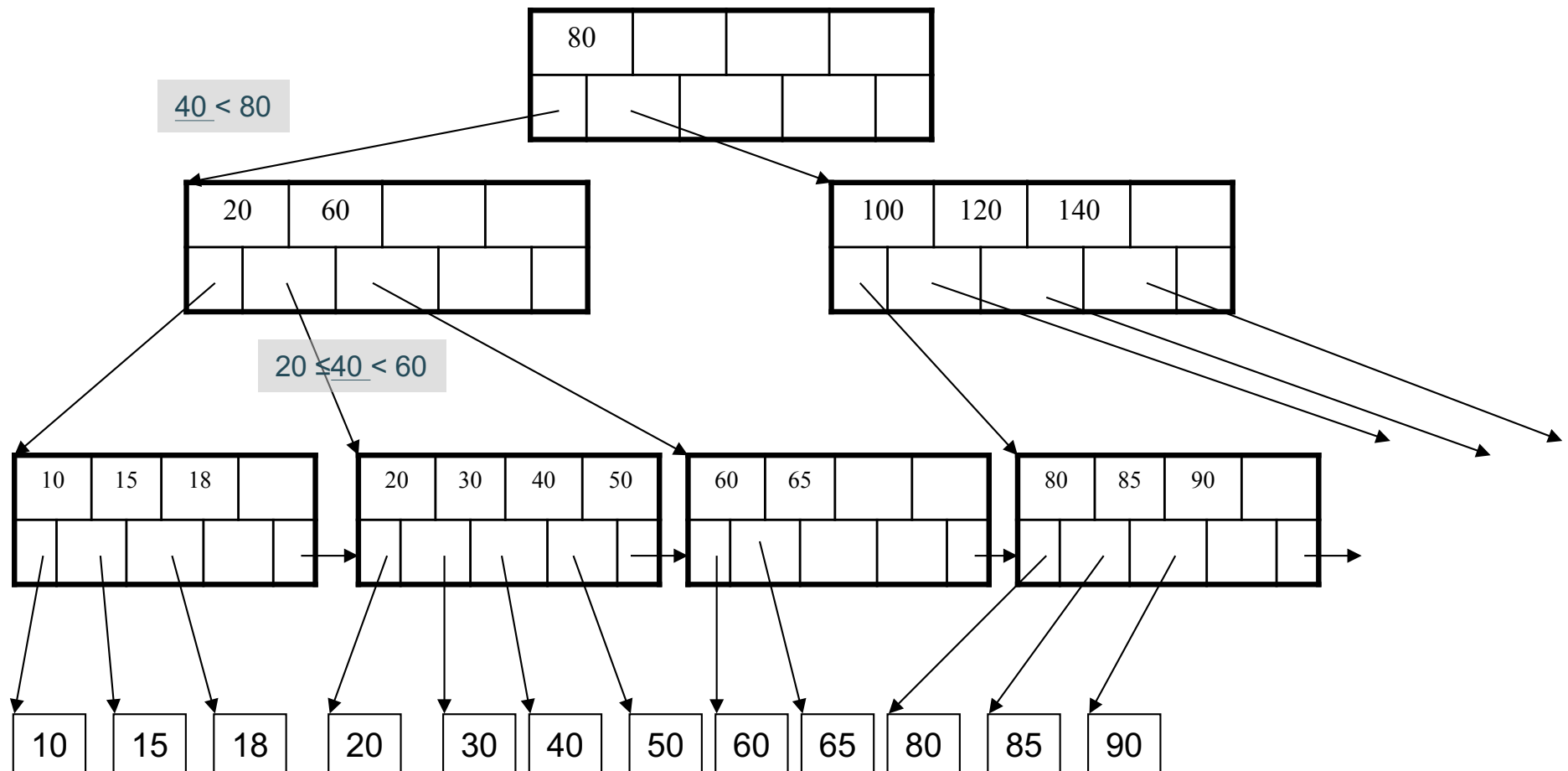
- Each leaf has  **$d \leq m \leq 2d$  keys**:



# B+ Tree Example

$d = 2$

Find the key 40



# Searching a B+ Tree

- Exact key values:
  - Start at the root
  - Proceed down, to the leaf
- Range queries:
  - Find lowest bound as above
  - Then sequential traversal

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

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

# B+ Tree Design

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



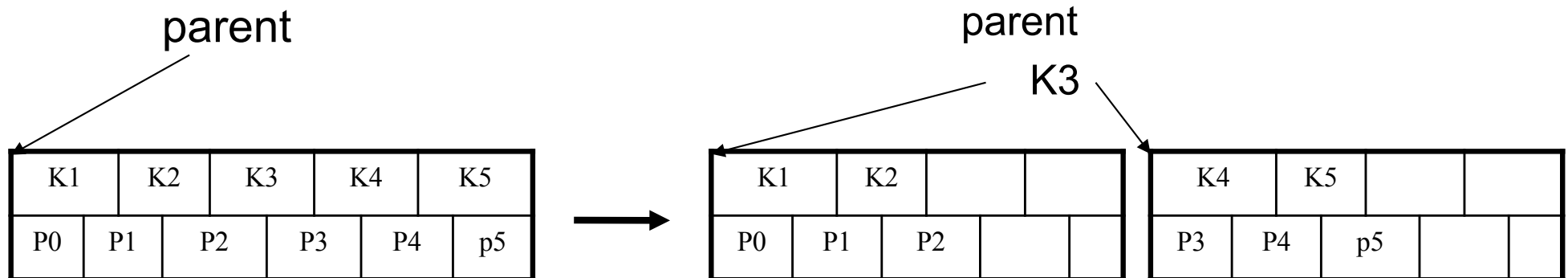
# B+ Trees in Practice

- 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)

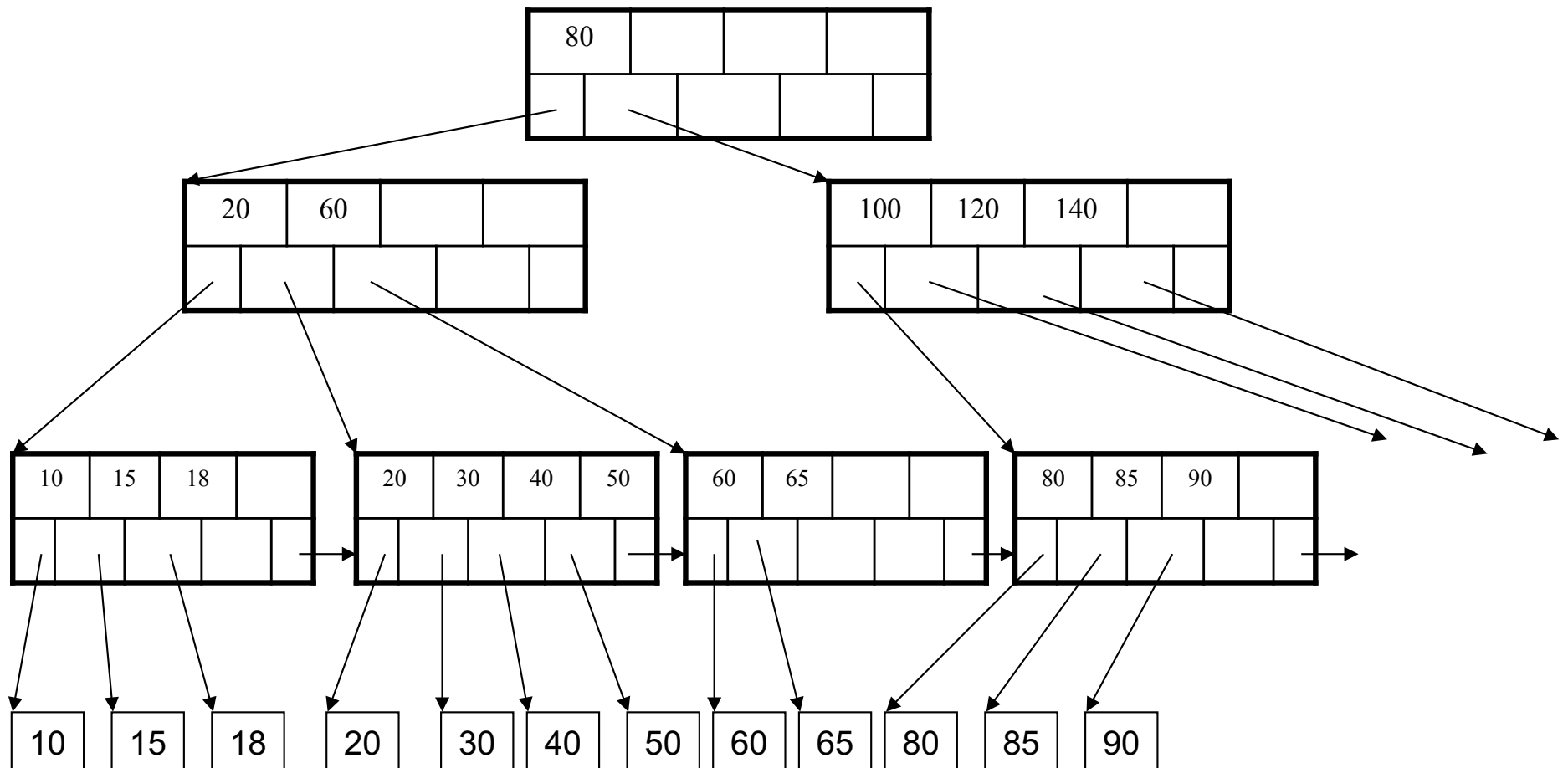
- Find leaf where K belongs, insert
- If no overflow ( $2d$  keys or less), halt
- If overflow ( $2d+1$  keys), split node, insert in parent:



- If leaf, also keep  $K_3$  in right node
- When root splits, new root has 1 key only

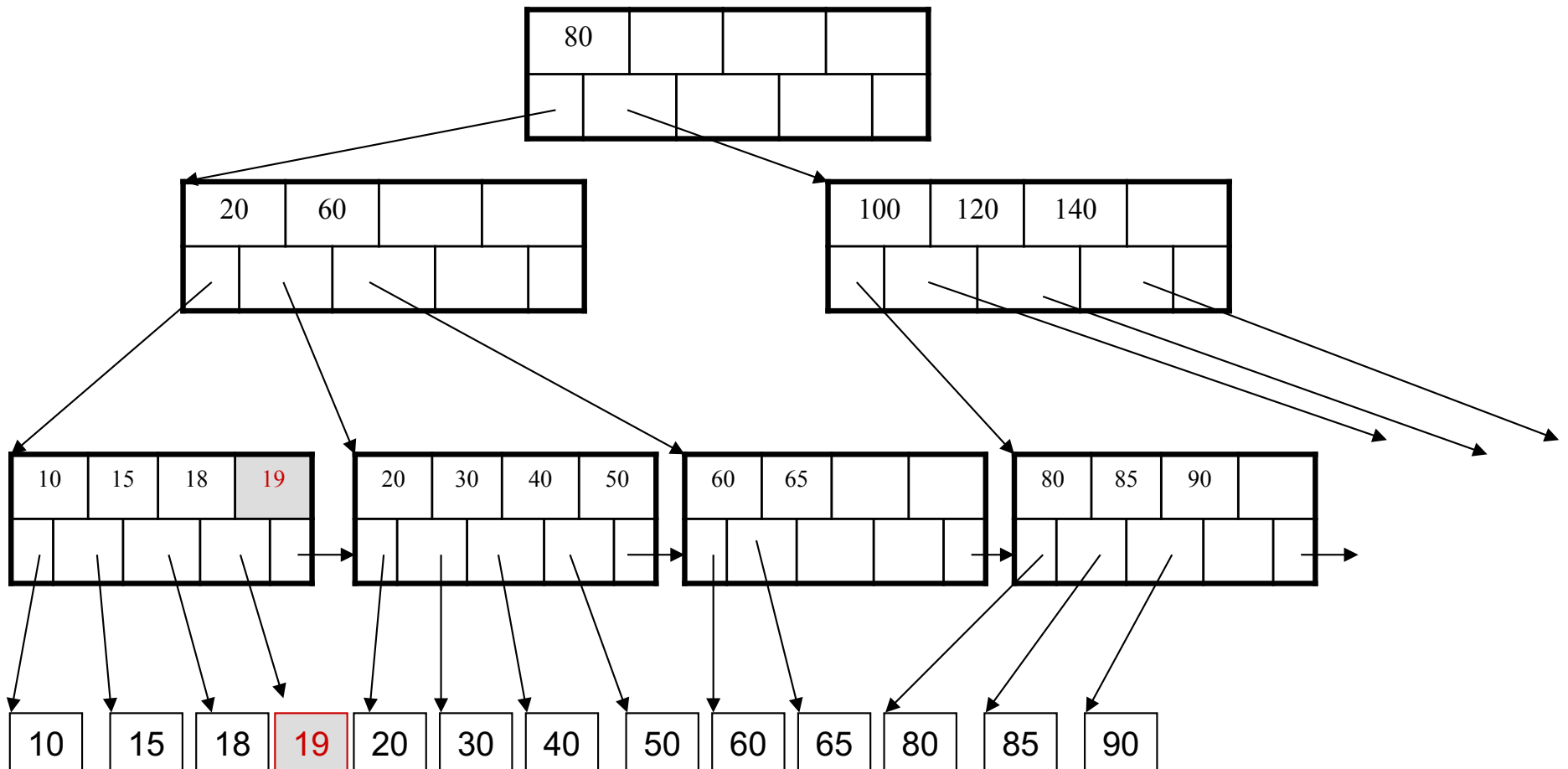
# Insertion in a B+ Tree

Insert K=19



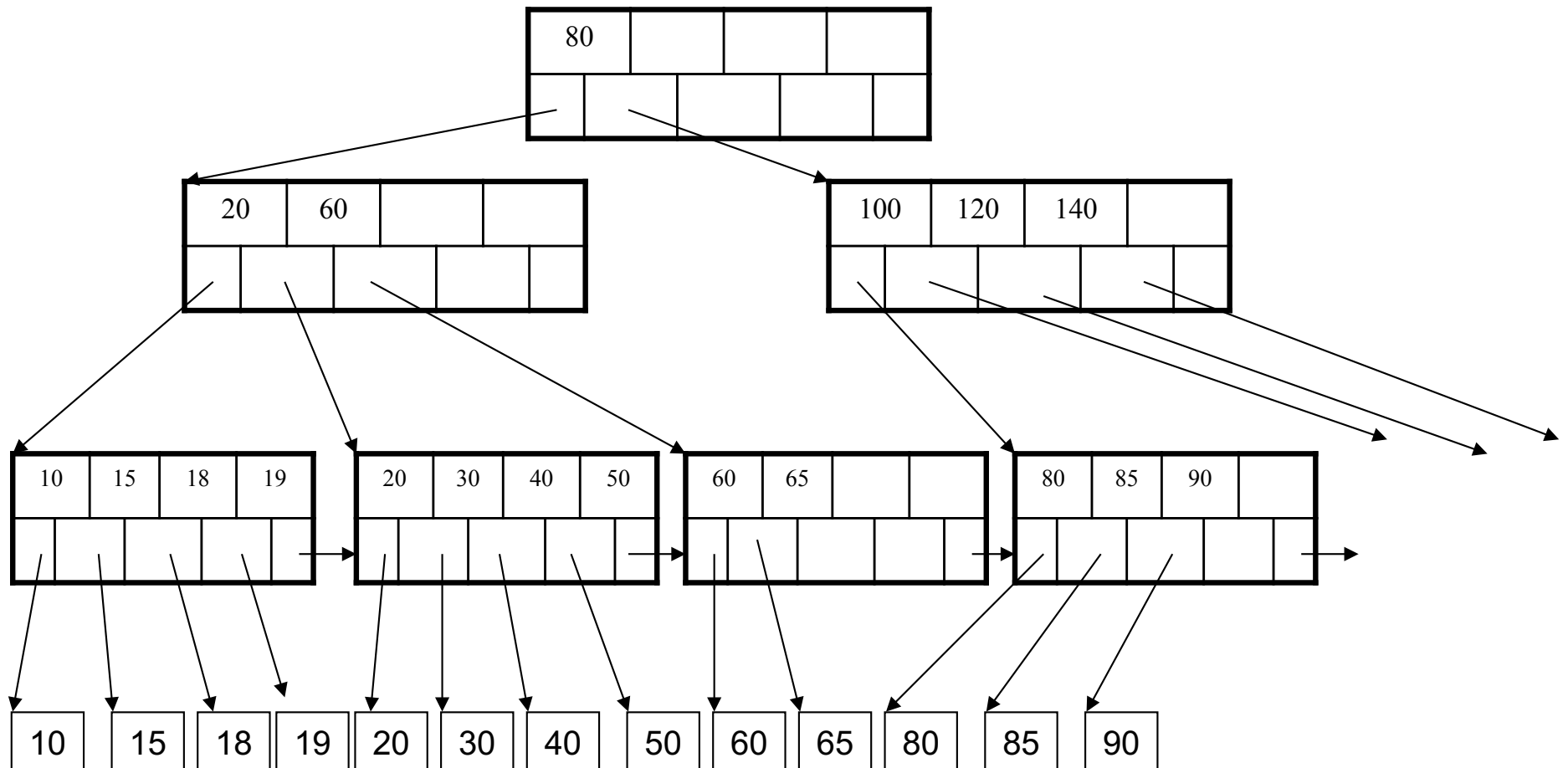
# Insertion in a B+ Tree

After insertion



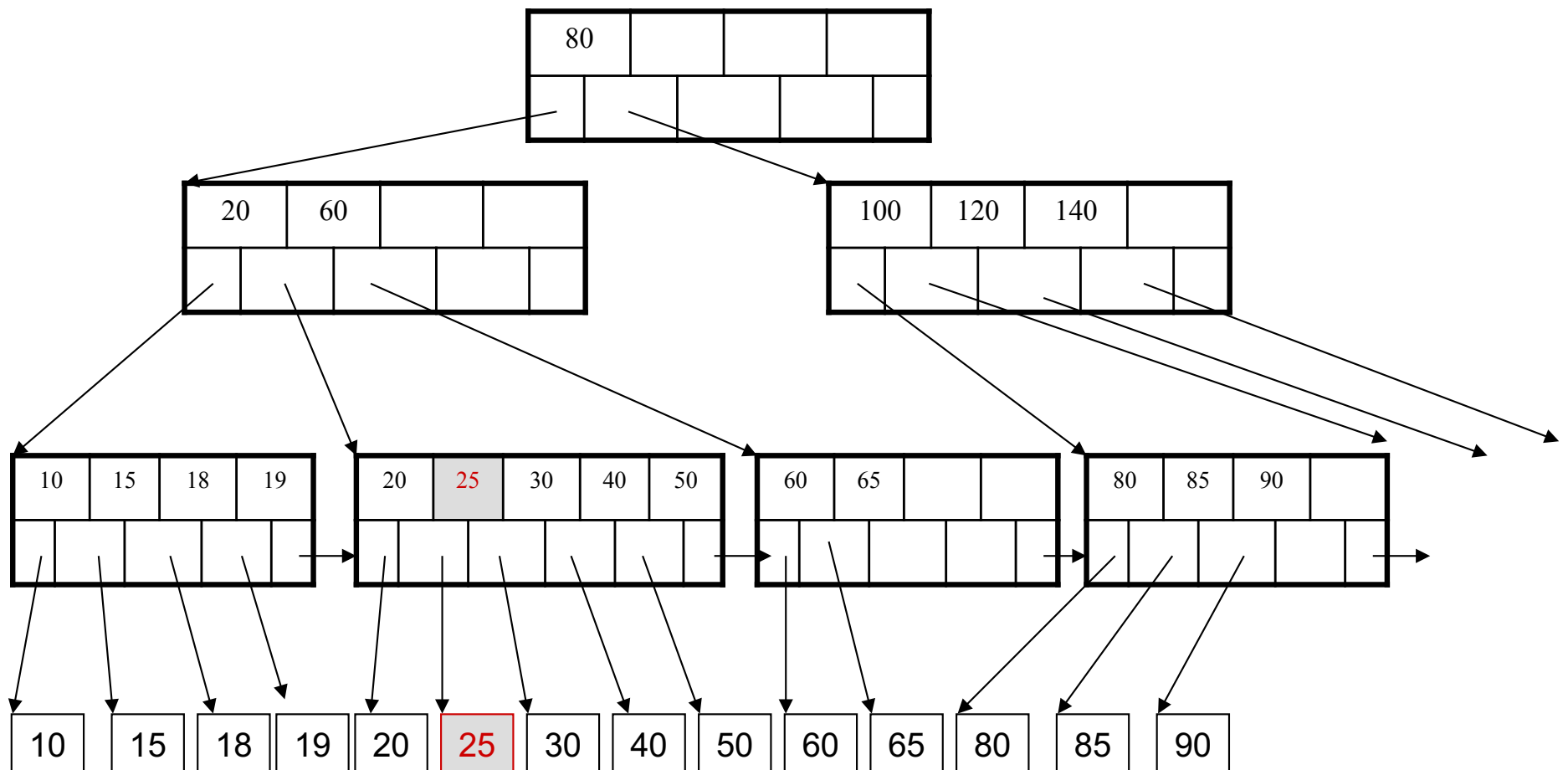
# Insertion in a B+ Tree

Now insert 25



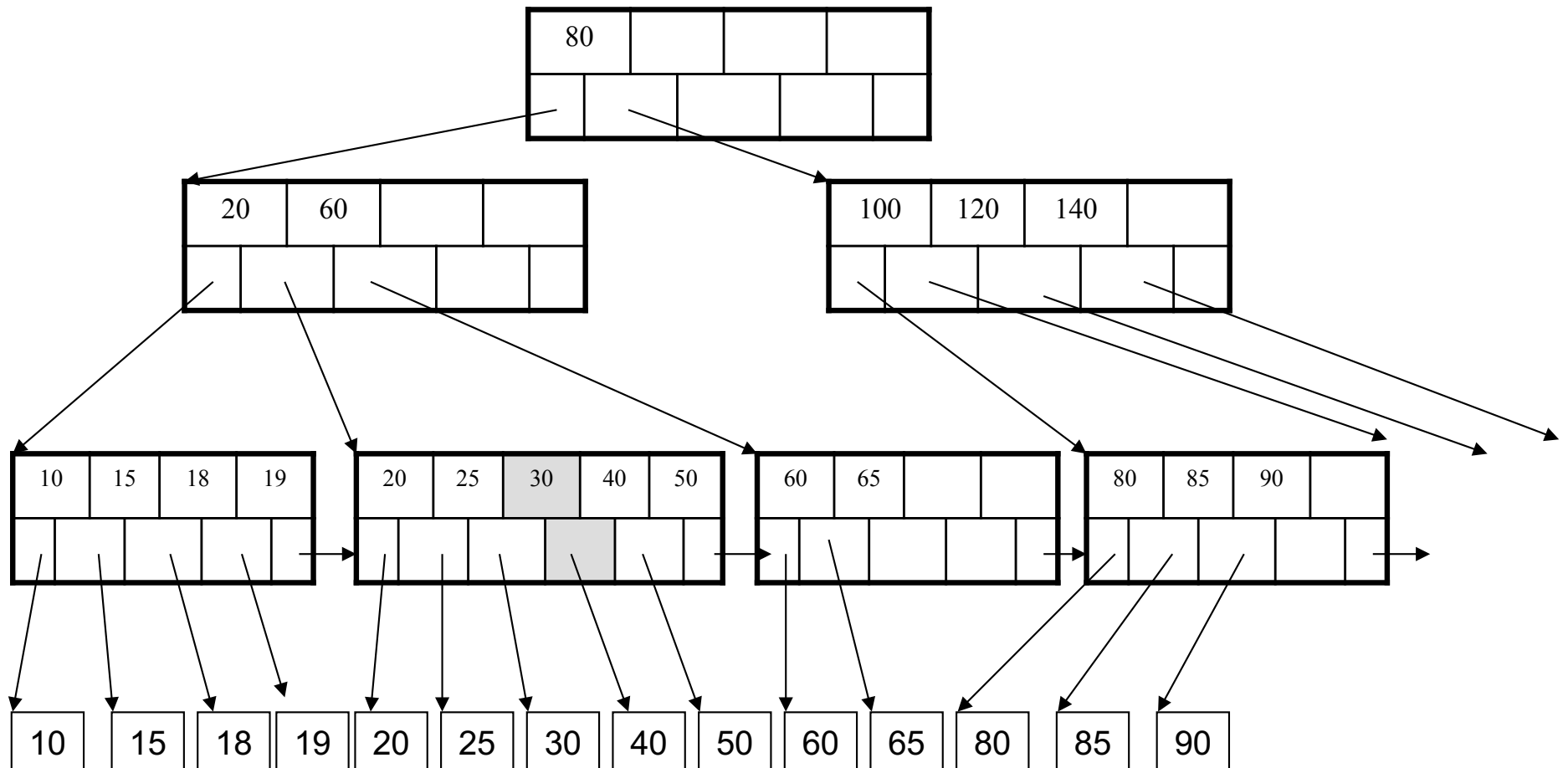
# Insertion in a B+ Tree

After insertion



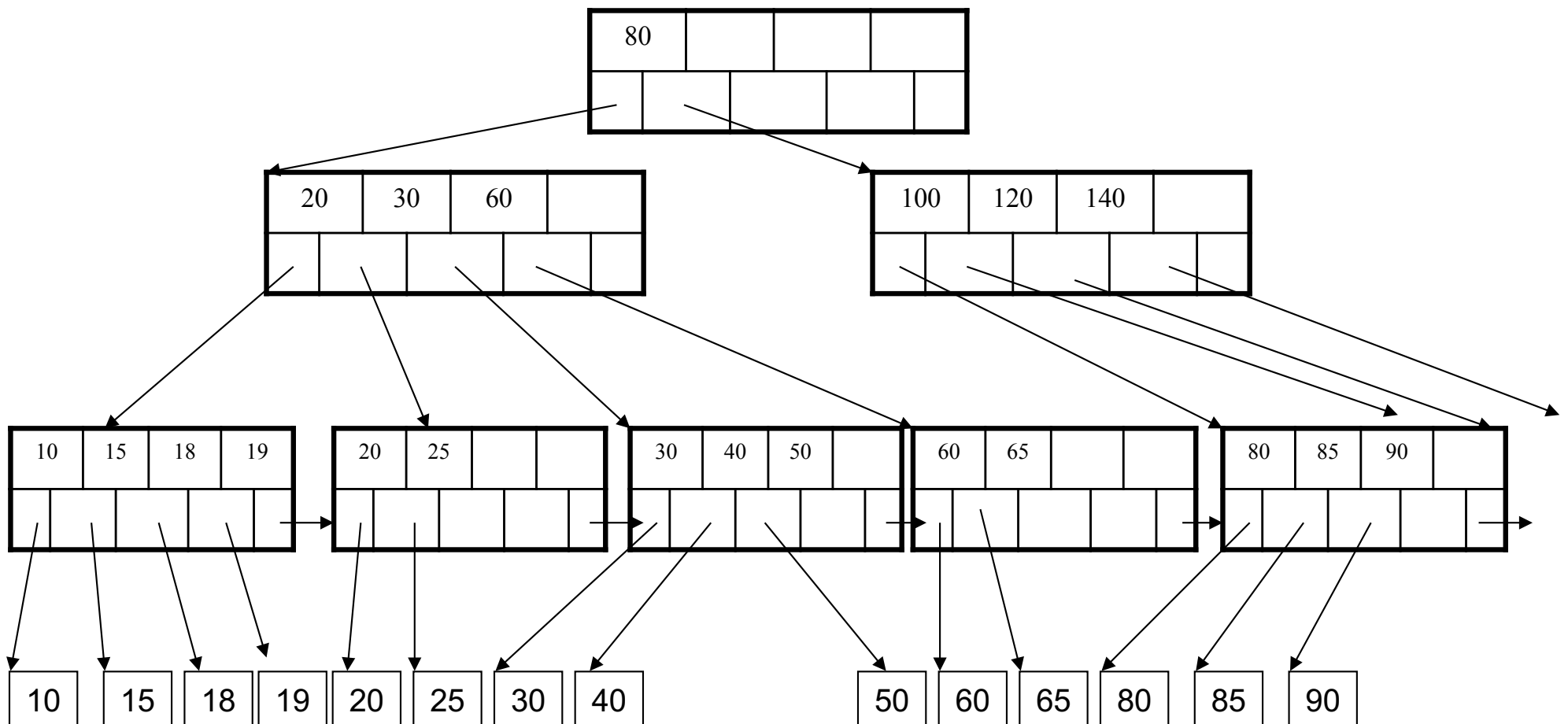
# Insertion in a B+ Tree

But now have to split !



# Insertion in a B+ Tree

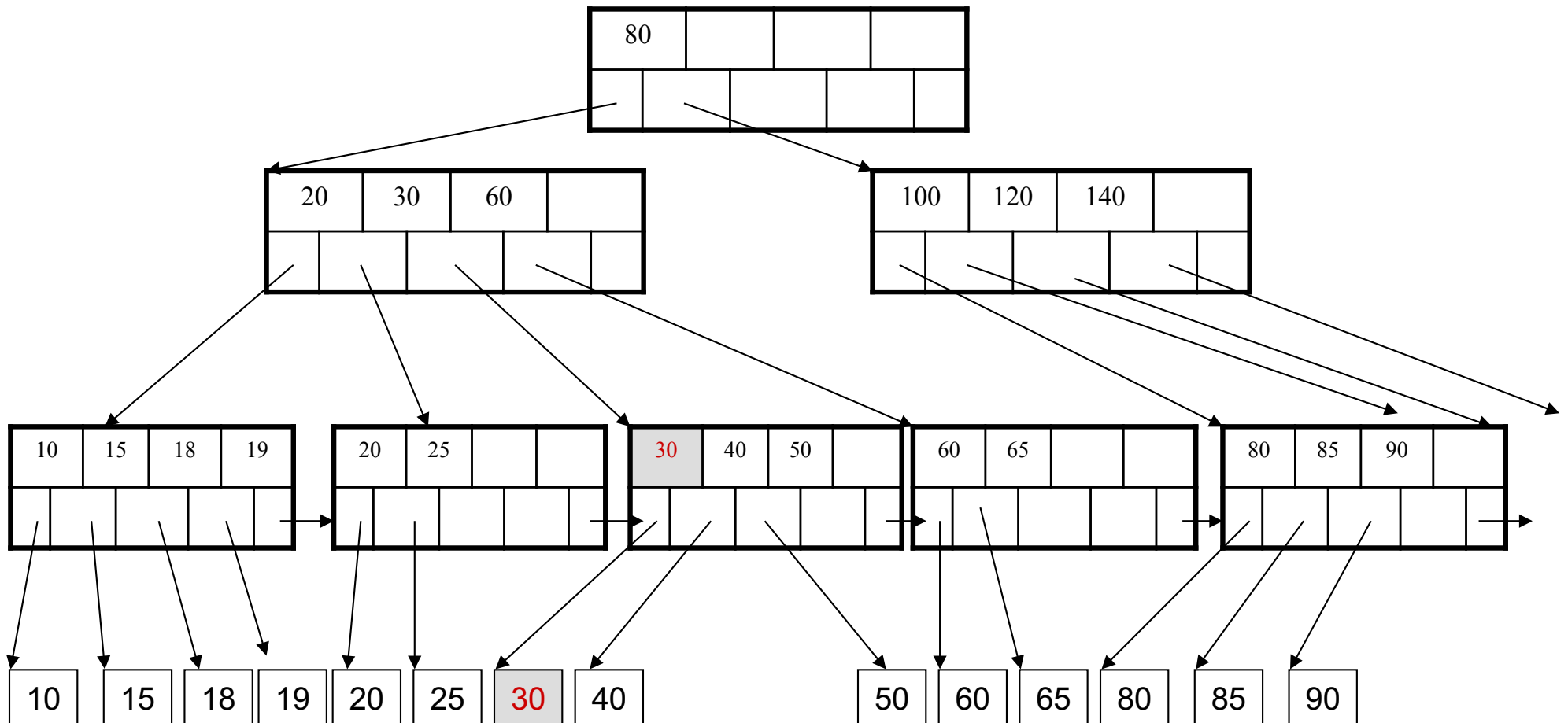
After the split





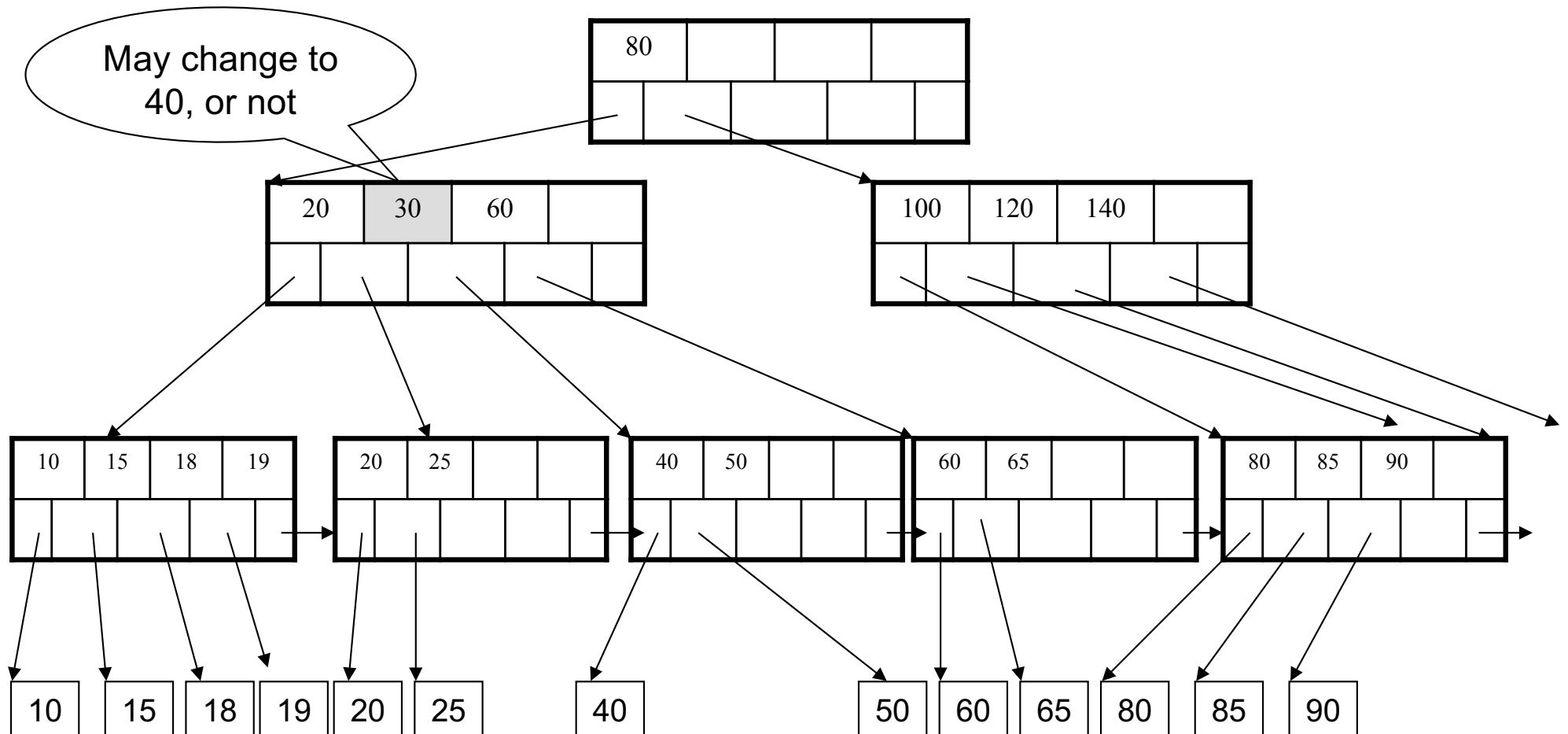
# Deletion from a B+ Tree

Delete 30



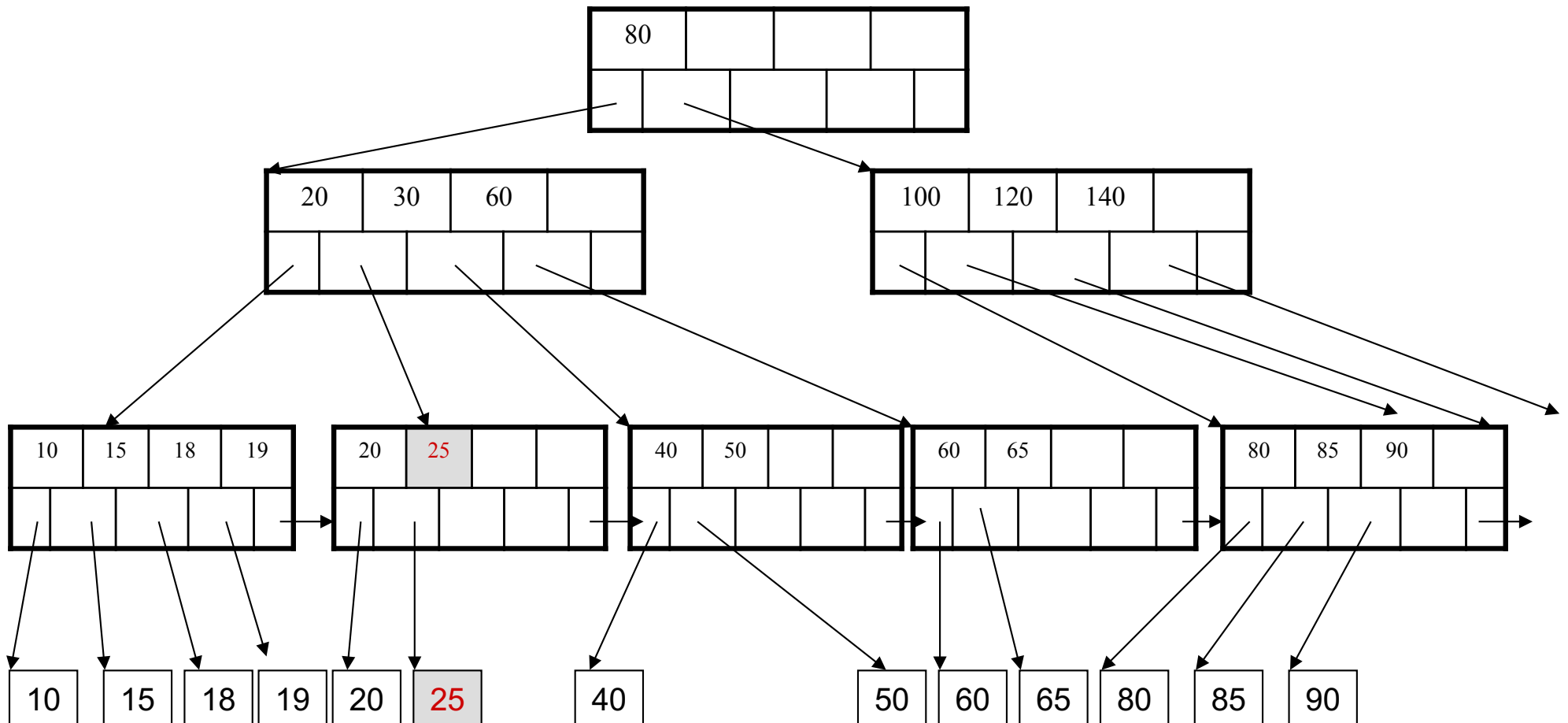
# Deletion from a B+ Tree

After deleting 30



# Deletion from a B+ Tree

Now delete 25

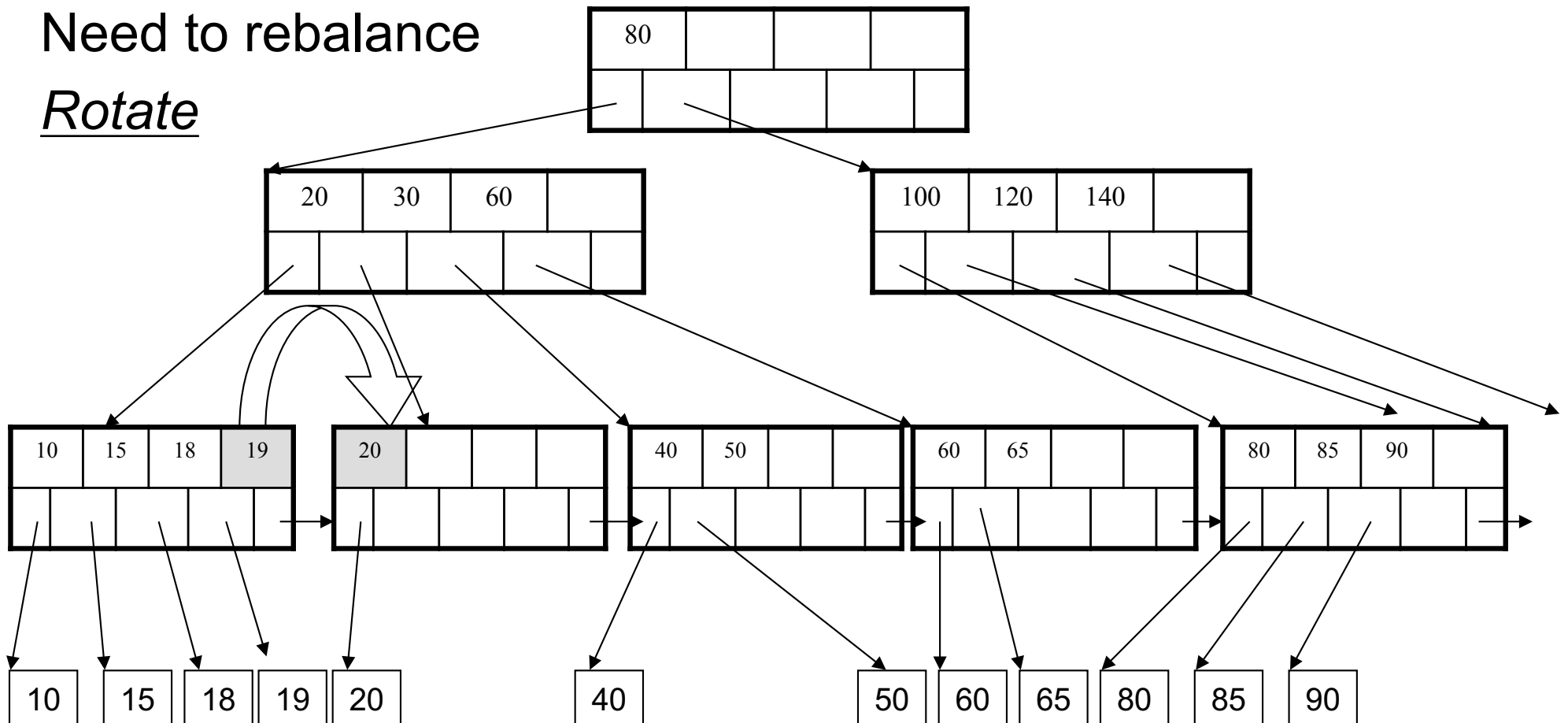


# Deletion from a B+ Tree

After deleting 25

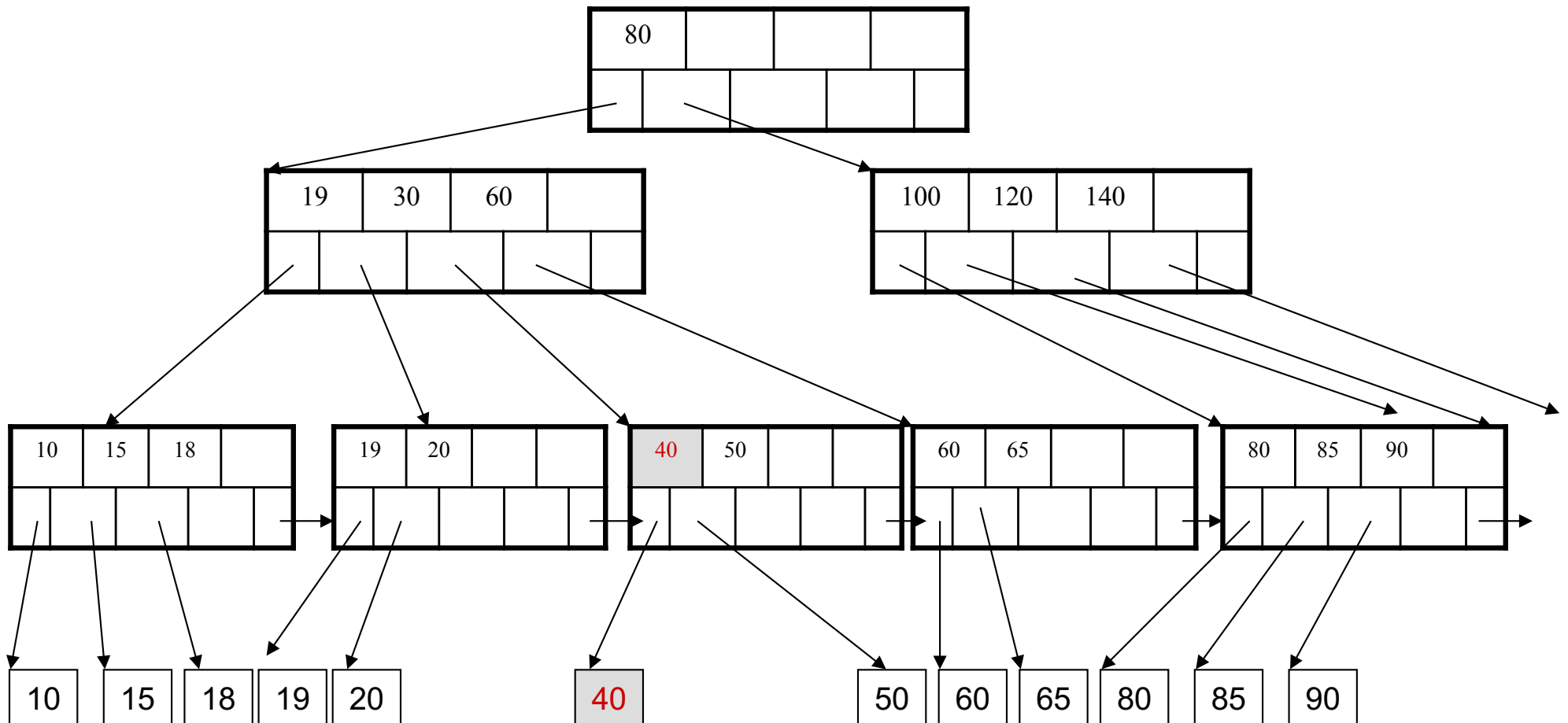
Need to rebalance

Rotate



# Deletion from a B+ Tree

Now delete 40

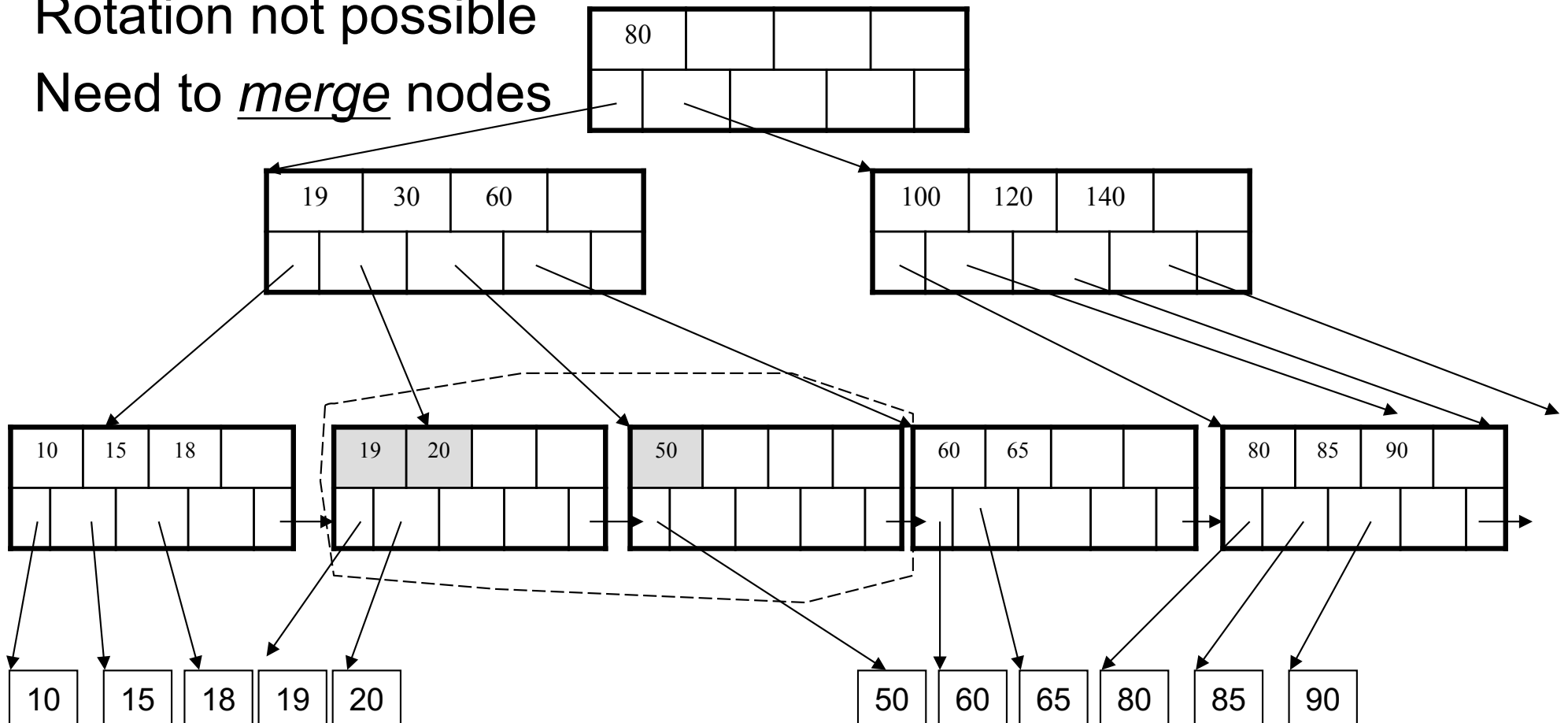


# Deletion from a B+ Tree

After deleting 40

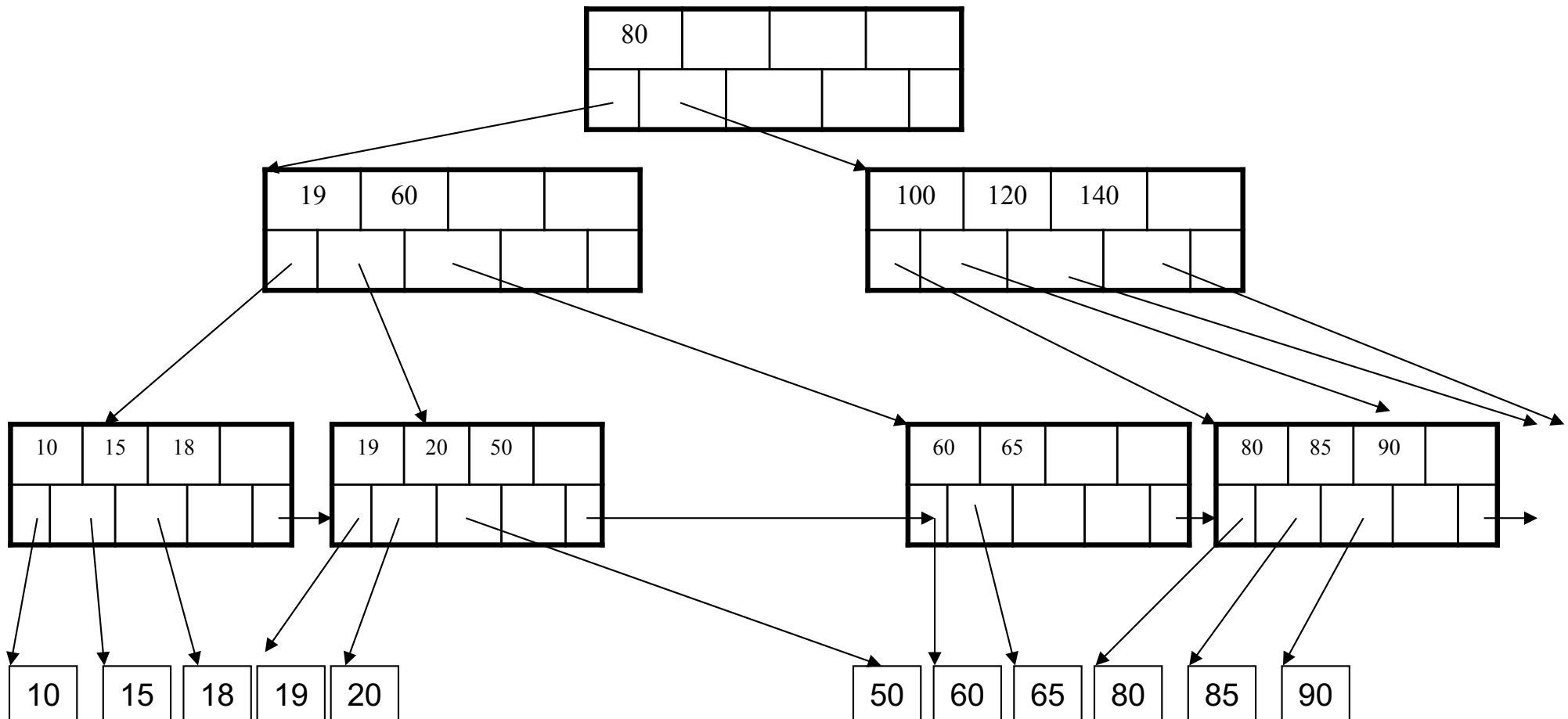
Rotation not possible

Need to merge nodes



# Deletion from a B+ Tree

Final tree



# Summary on B+ Trees

- Default index structure on most DBMSs
- 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



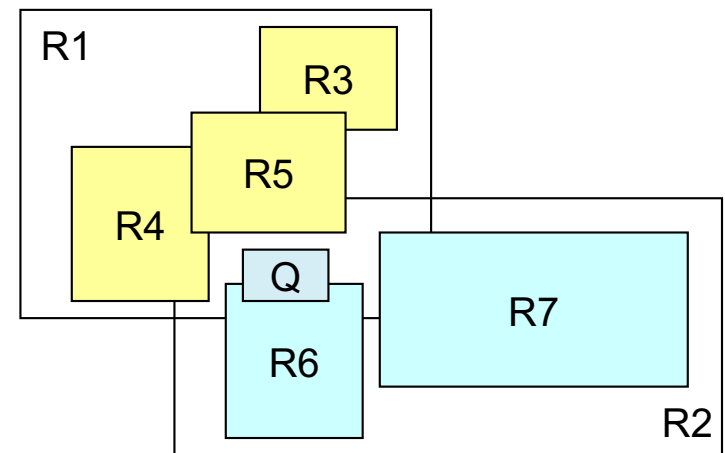
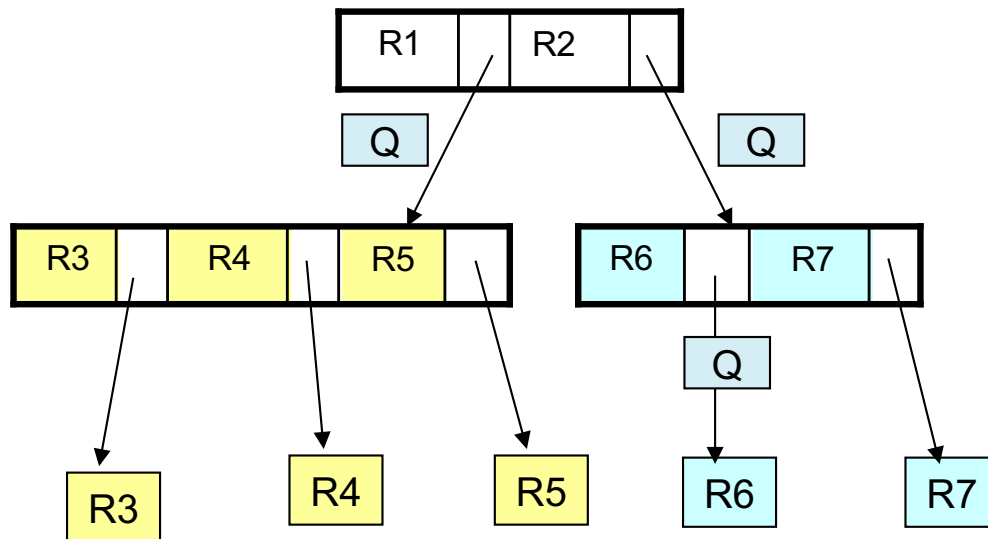
# Optional Material

- Let's take a look at another example of an index....

# R-Tree Example

Designed for spatial data

Search key values are bounding boxes



For insertion: at each level, choose child whose bounding box needs least enlargement (in terms of area)