

CSE 544

Principles of Database Management Systems

Magdalena Balazinska

Fall 2009

Lecture 5 - Storage and Indexing

References

- **Generalized Search Trees for Database Systems.**
J. M. Hellerstein, J. F. Naughton and A. Pfeffer. VLDB 1995.
- **Database management systems.** Third Edition. R. Ramakrishnan and J. Gehrke. Chapters 8 through 11

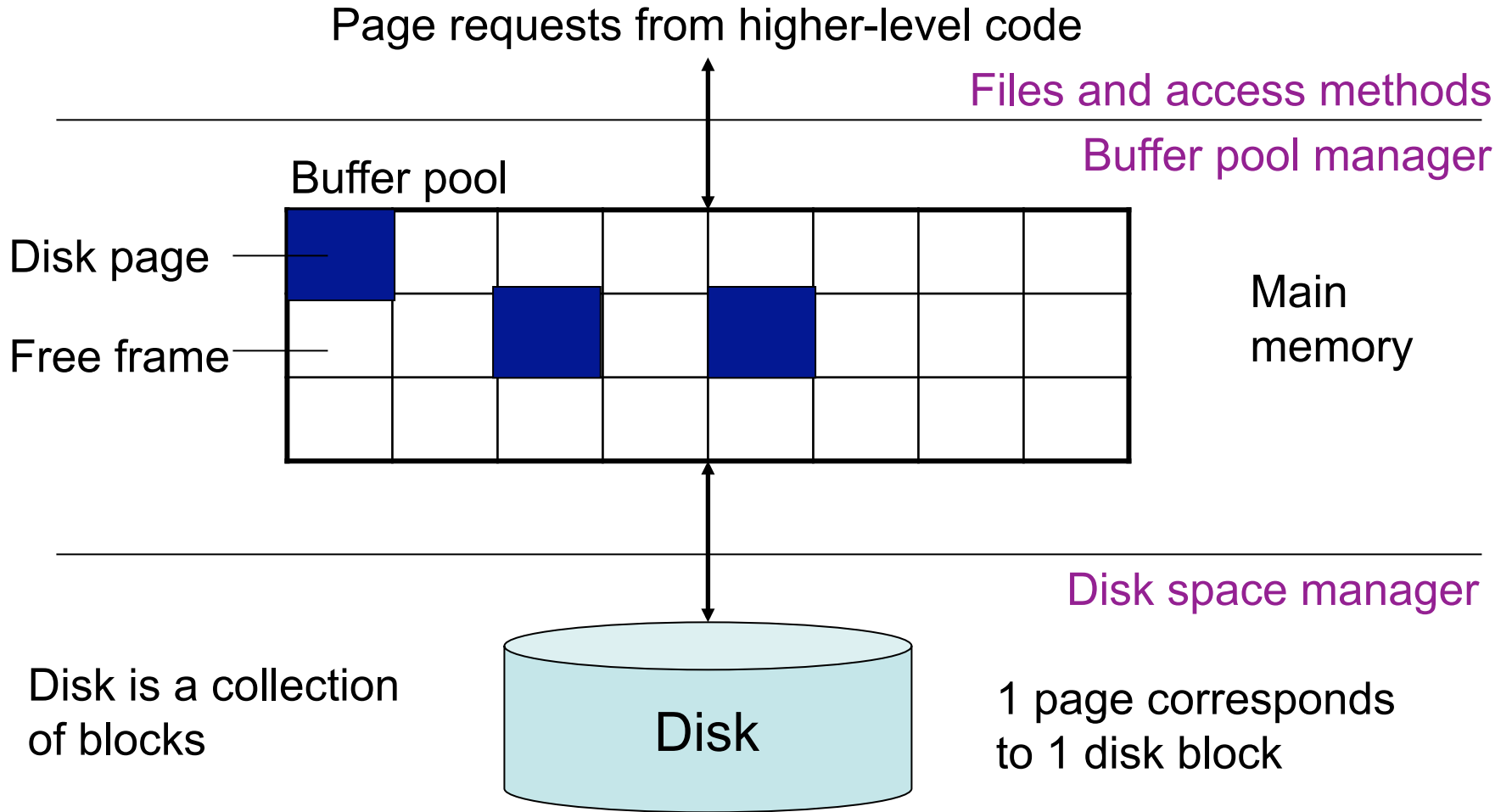
Storage Management

- Can be done by the OS or by the DBMS
- What are the trade-offs? See lecture 4
- How does the DBMS manage storage?

Outline

- **Data storage**
 - Disk and files: Sections 9.3 through 9.7
 - Operations on files
- **Indexes**
 - Index structures: Section 8.3
 - Hash-based indexes: Section 8.3.1 and Chapter 11
 - B+ trees: Section 8.3.2 and Chapter 10
 - GiST: Hellerstein et. al.'s VLDB'95

Buffer Manager



Data Storage

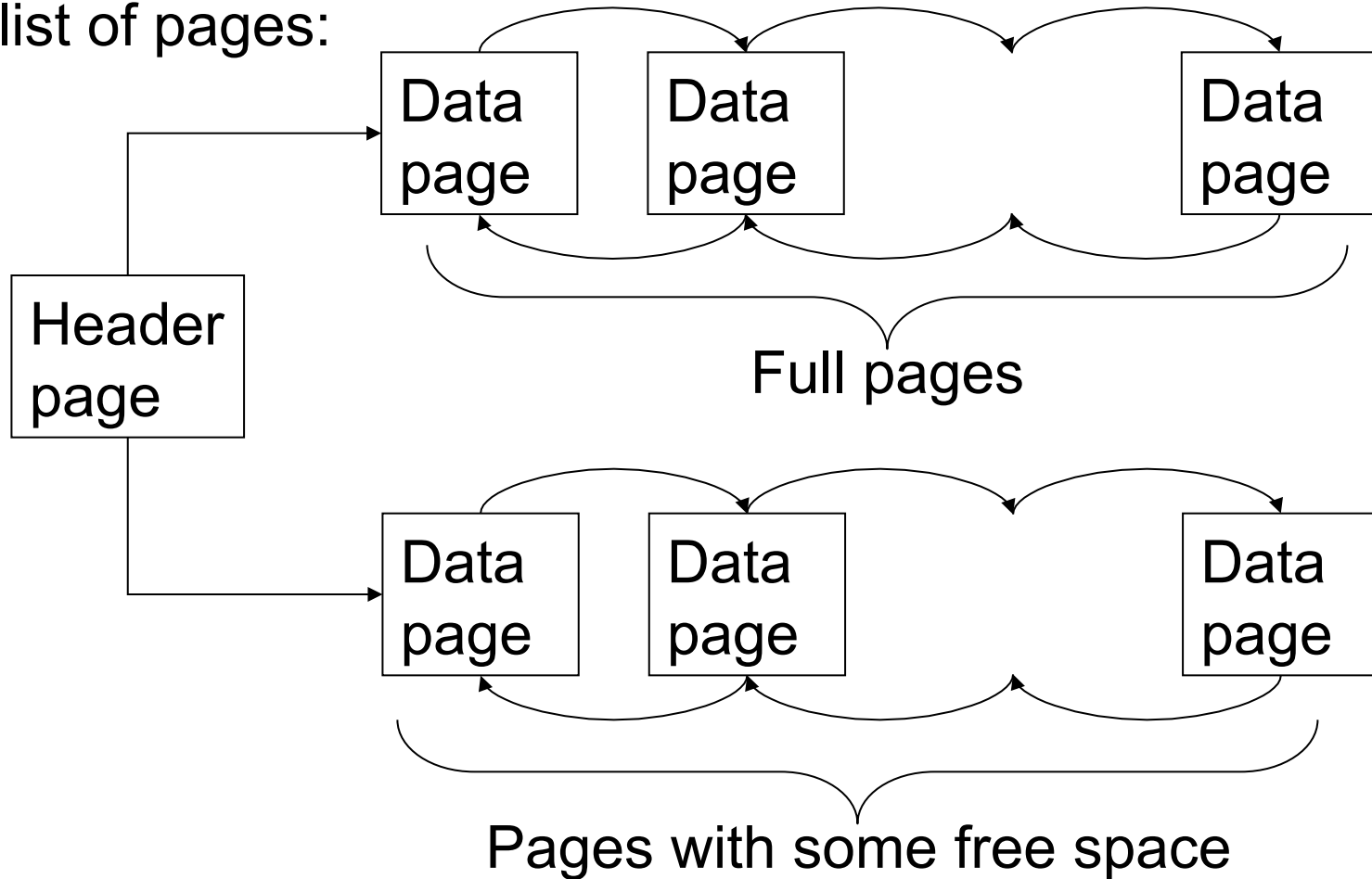
- **Basic abstraction**
 - *Collection of records or file*
 - Typically, 1 relation = 1 file
 - A file consists of *one or more pages*
- How to organize pages into files?
- How to organize records inside a file?
- Simplest approach: **heap file** (unordered)

Heap File Operations

- **Create** or **destroy** a file
- **Insert** a record
- **Delete** a record with a given rid (rid)
 - rid: unique tuple identifier such that
 - can identify disk address of page containing record by using rid
- **Get** a record with a given rid
- **Scan** all records in the file

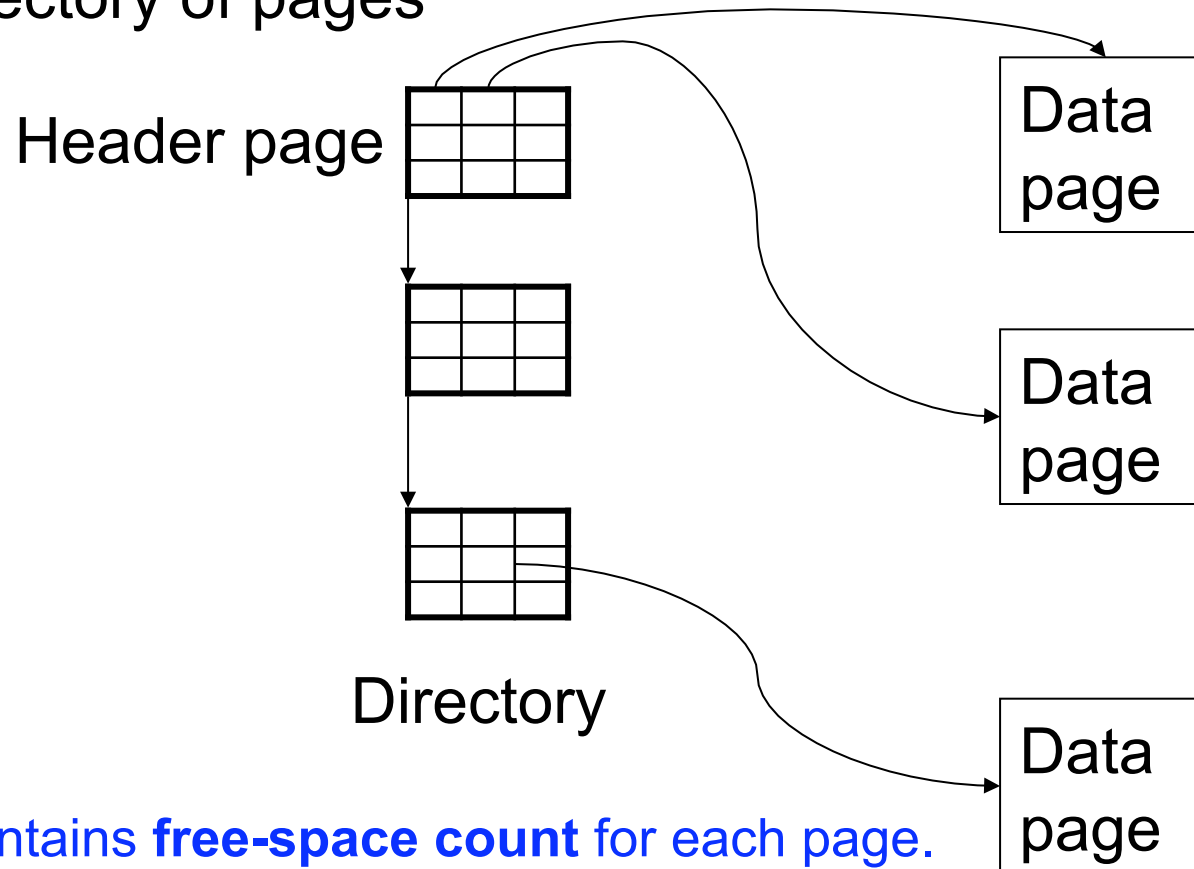
Heap File Implementation 1

Linked list of pages:



Heap File Implementation 2

Better: directory of pages



Directory contains **free-space count** for each page.
Faster inserts for variable-length records

Page Formats

Issues to consider

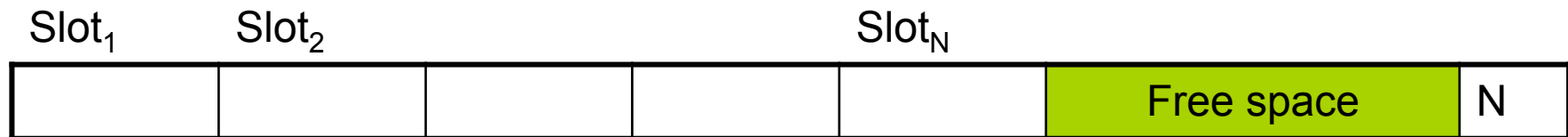
- 1 page = 1 disk block = fixed size (e.g. 8KB)
- Records:
 - Fixed length
 - Variable length
- Record id = RID
 - Typically RID = (PageID, SlotNumber)

Why do we need RID's in a relational DBMS ?

See discussion about indexes later in the lecture

Page Format Approach 1

Fixed-length records: packed representation



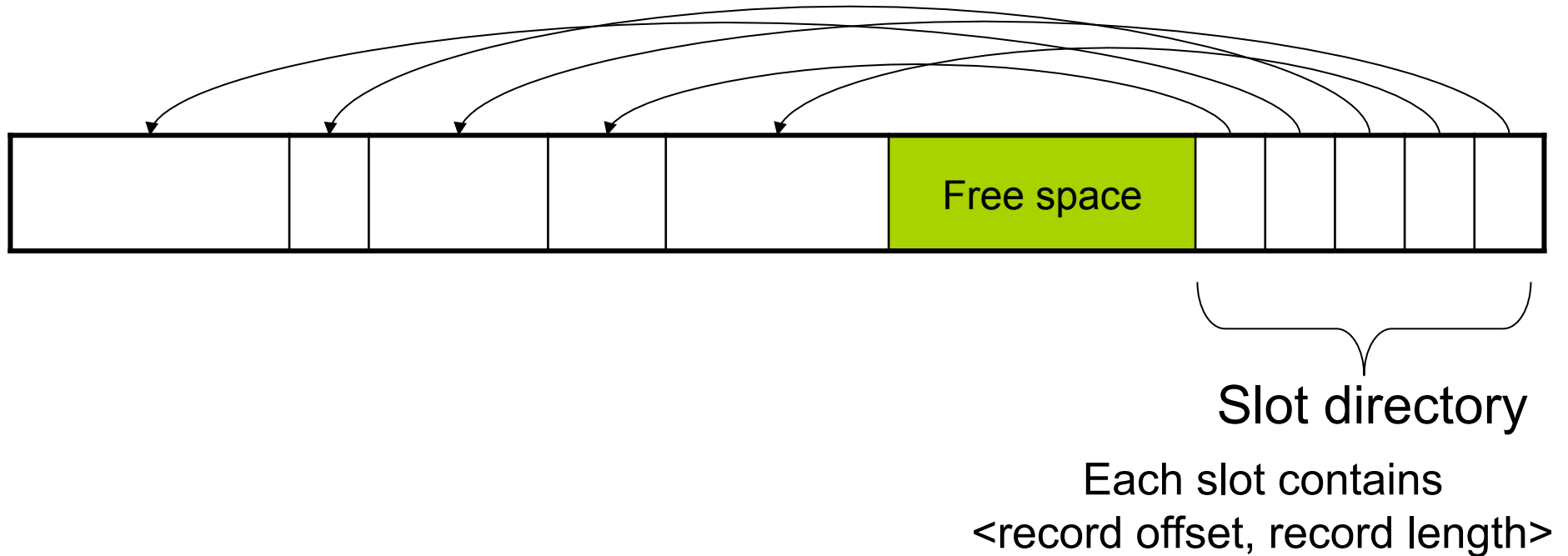
Problems ?

How to handle variable-length records?

Need to move records for each deletion, changing RIDs

Number of records

Page Format Approach 2

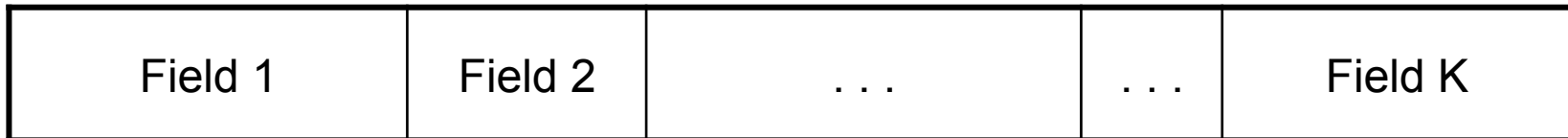


Can handle variable-length records

Can move tuples inside a page without changing RIDs

Record Formats

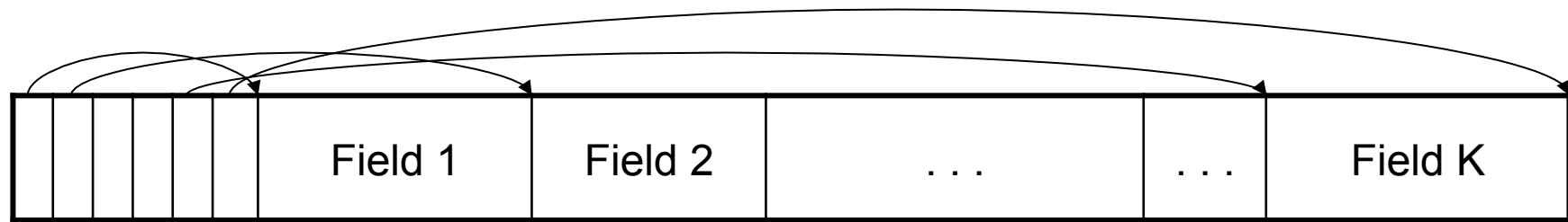
Fixed-length records → Each field has a fixed length (i.e., it has the same length in all the records)



Information about field lengths and types is in the catalog

Record Formats

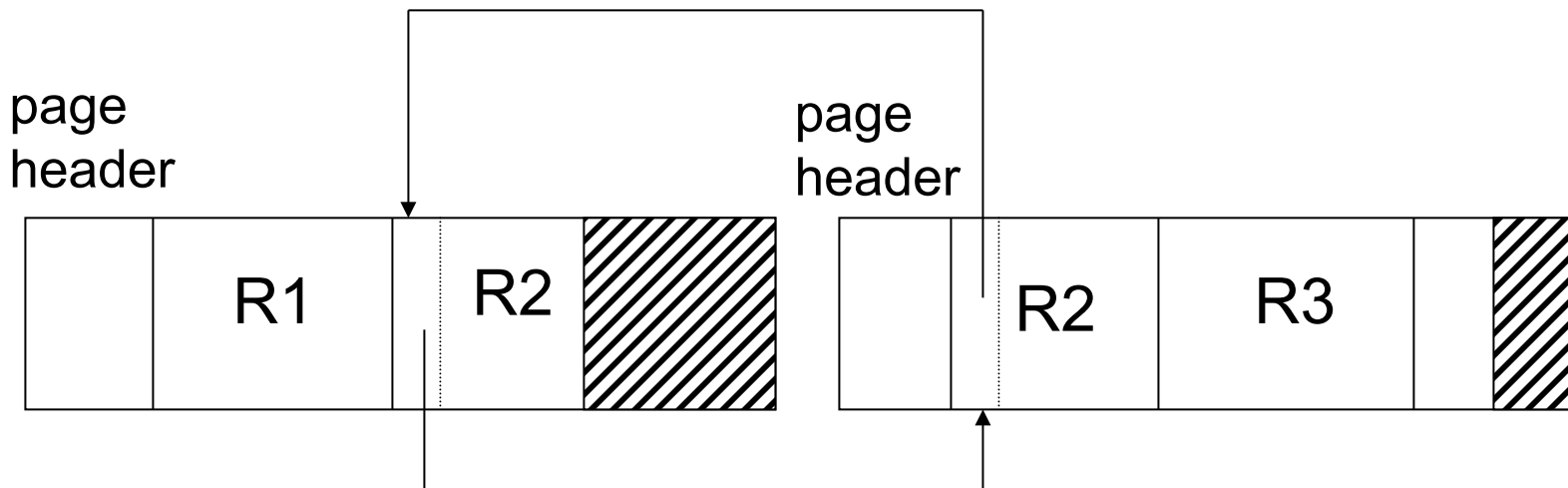
Variable length records



Record header

Remark: NULLS require no space at all (why ?)

Long Records Across Pages



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

LOB

- Large objects
 - Binary large object: BLOB
 - Character large object: CLOB
- Supported by modern database systems
- E.g. images, sounds, texts, etc.
- Storage: attempt to cluster blocks together

Types of Files

- **Heap file**
 - Unordered
- **Sorted file (also called sequential file)**
- **Clustered file**

We discussed heap files

The others are similar

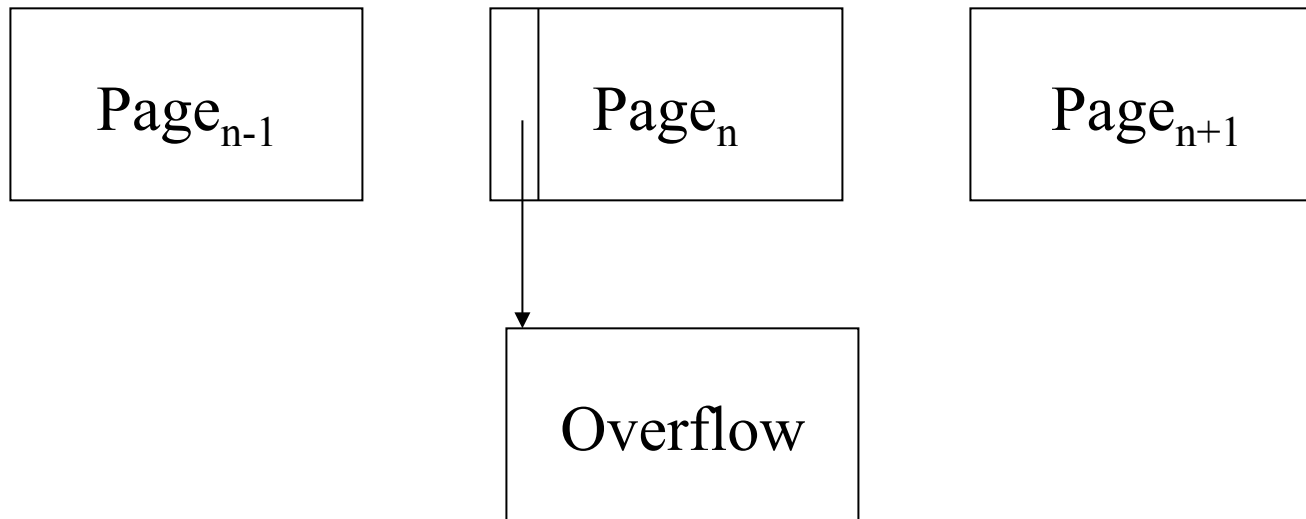
Outline

- **Data storage**
 - Disk and files: Sections 9.3 through 9.7
 - **Operations on files**
- **Indexes**
 - Index structures: Section 8.3
 - Hash-based indexes: Section 8.3.1
 - B+ trees: Chapter 10
 - GiST: Hellerstein et. al.'s VLDB'95

Modifications: Insertion

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

Overflow Pages



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

Modifications: Deletions

- Free space in page, shift records
 - Be careful with slots
 - RIDs for remaining tuples must NOT change
- May be able to eliminate an overflow page

Modifications: Updates

- If new record is shorter than previous, easy 😊
- If it is longer, need to shift records
 - May have to create overflow pages

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?

Outline

- **Data storage**
 - Disk and files: Sections 9.3 through 9.7
 - Operations on files
- **Indexes**
 - **Index structures: Section 8.3**
 - Hash-based indexes: Section 8.3.1
 - B+ trees: Chapter 10
 - GiST: Hellerstein et. al.'s VLDB'95

Indexes

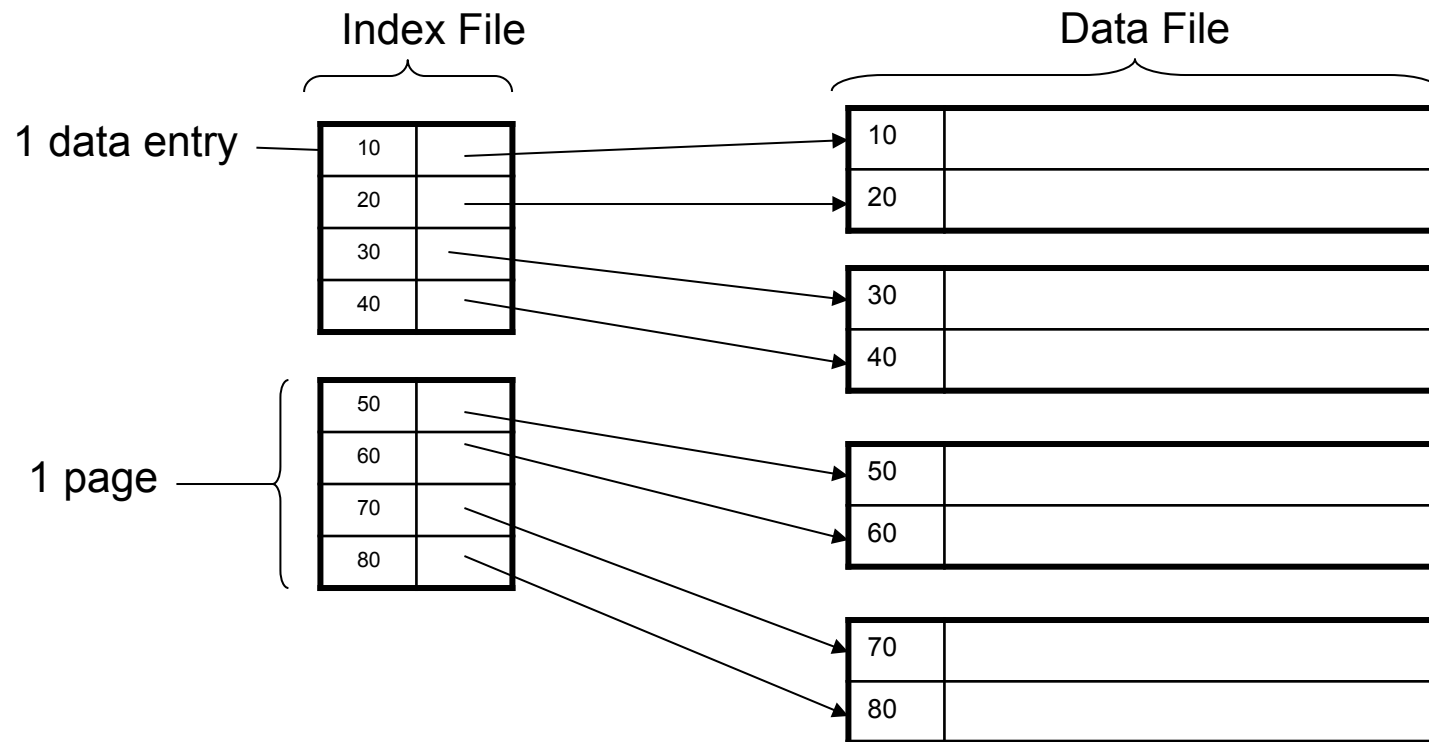
- **Index**: data structure that organizes data records on disk to optimize selections on the *search key fields* for the index
- An index contains a collection of *data entries*, and supports efficient retrieval of all data entries with a given search key value *k*

Indexes

- **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:
 - The actual record with key k
 - In this case, **the index is also a special file organization**
 - This type of index is also called the **primary index** of a file
 - (k, RID)
 - (k, list-of-RIDs)

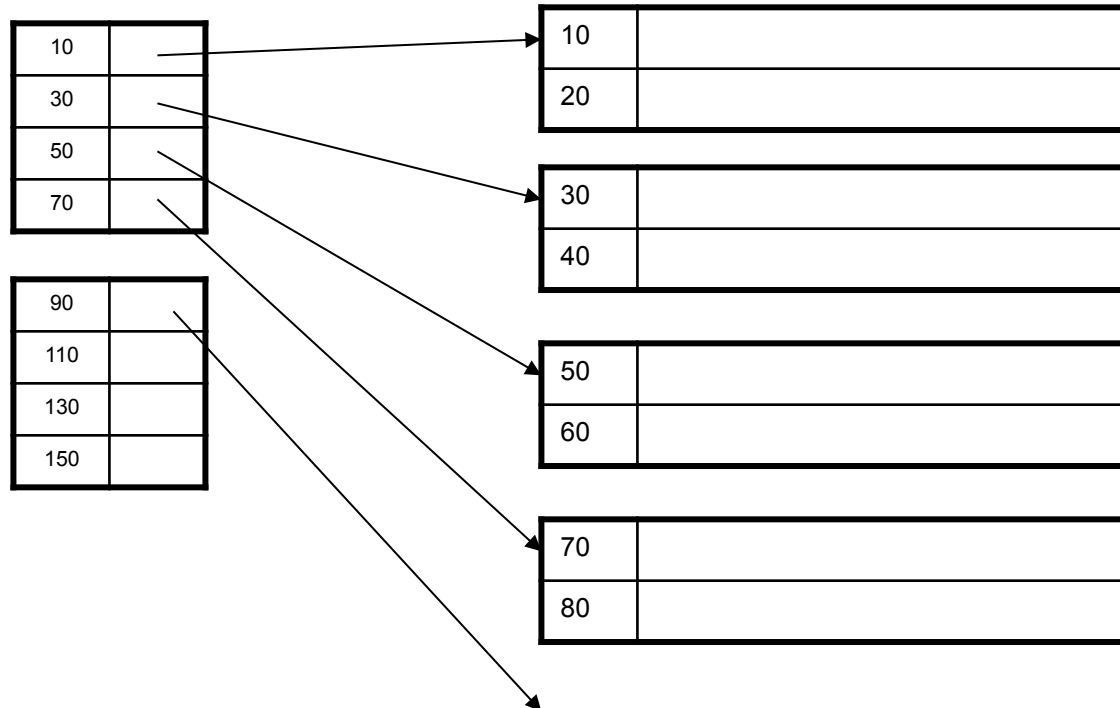
Primary Index

- Index determines the location of indexed records
- Dense index: sequence of (key,pointer) pairs



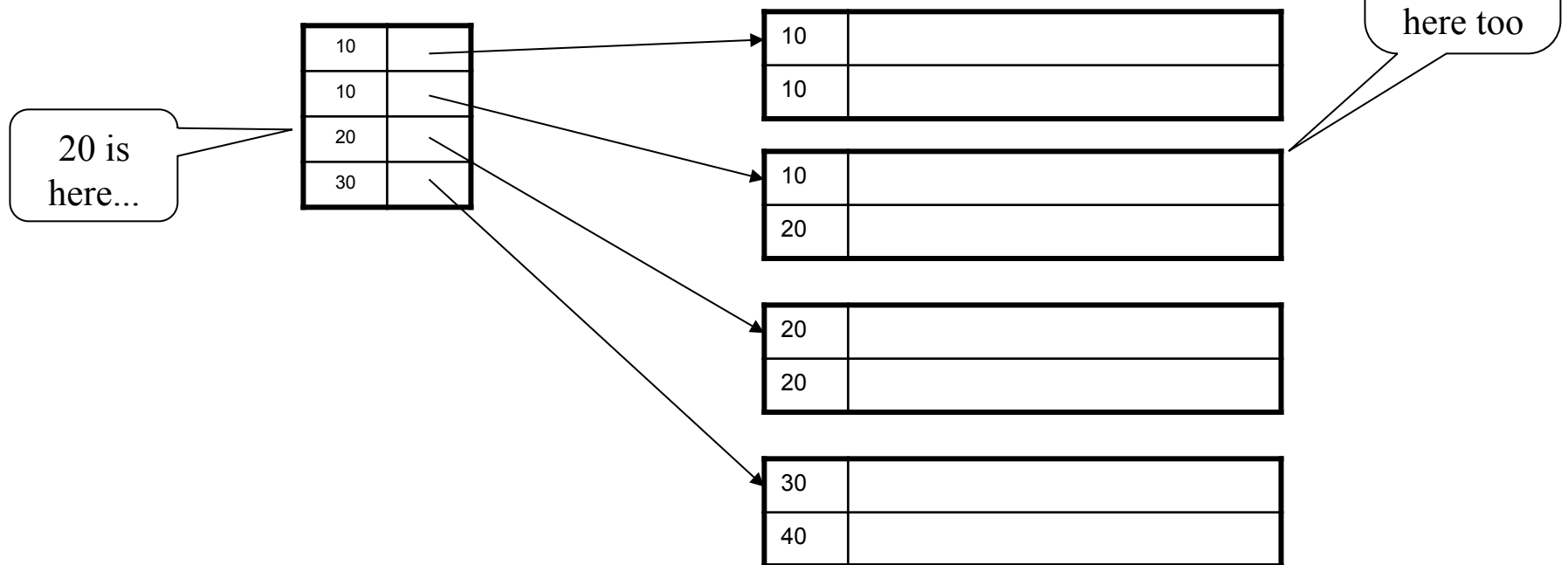
Primary Index

- Sparse index



Primary Index with Duplicate Keys

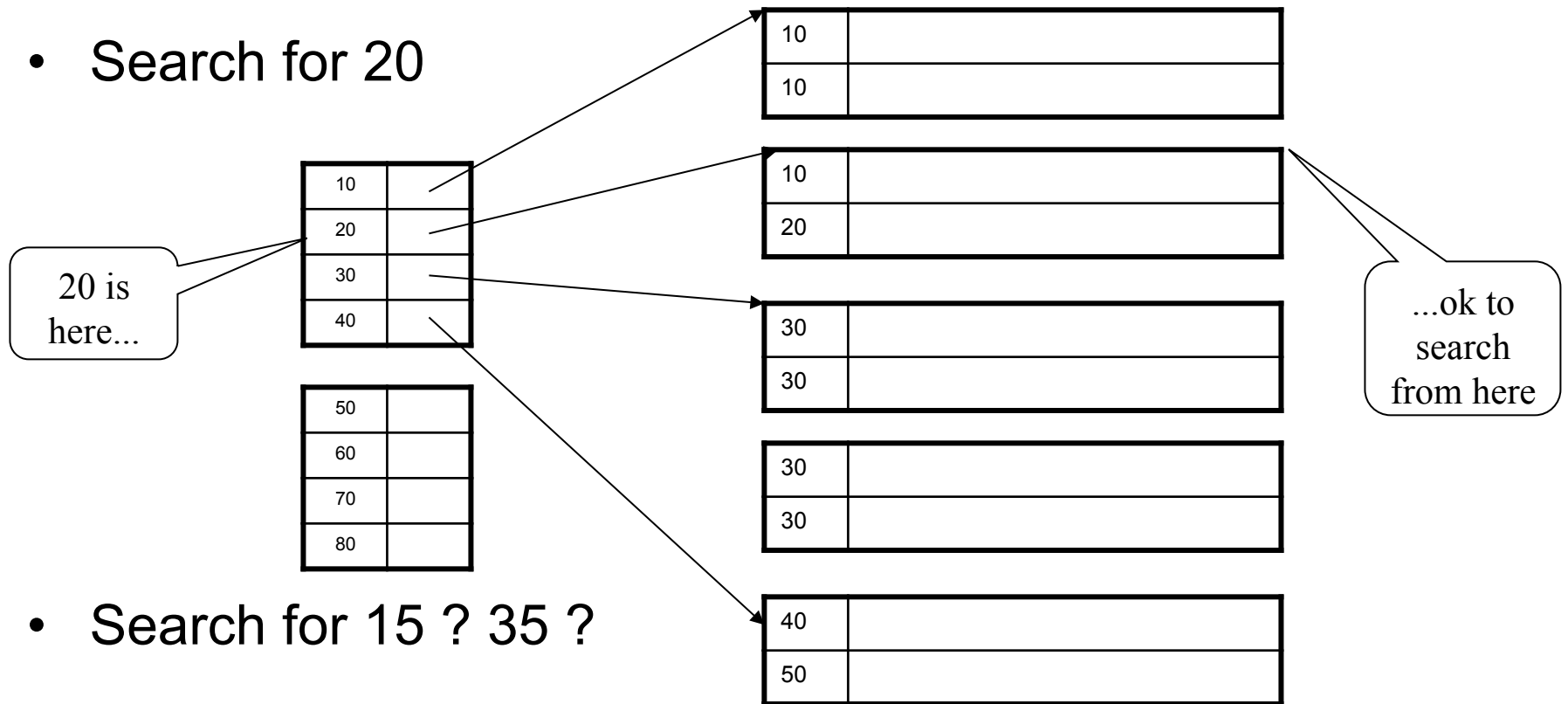
- Sparse index: pointer to lowest search key on each page:
- Search for 20



Primary Index with Duplicate Keys

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

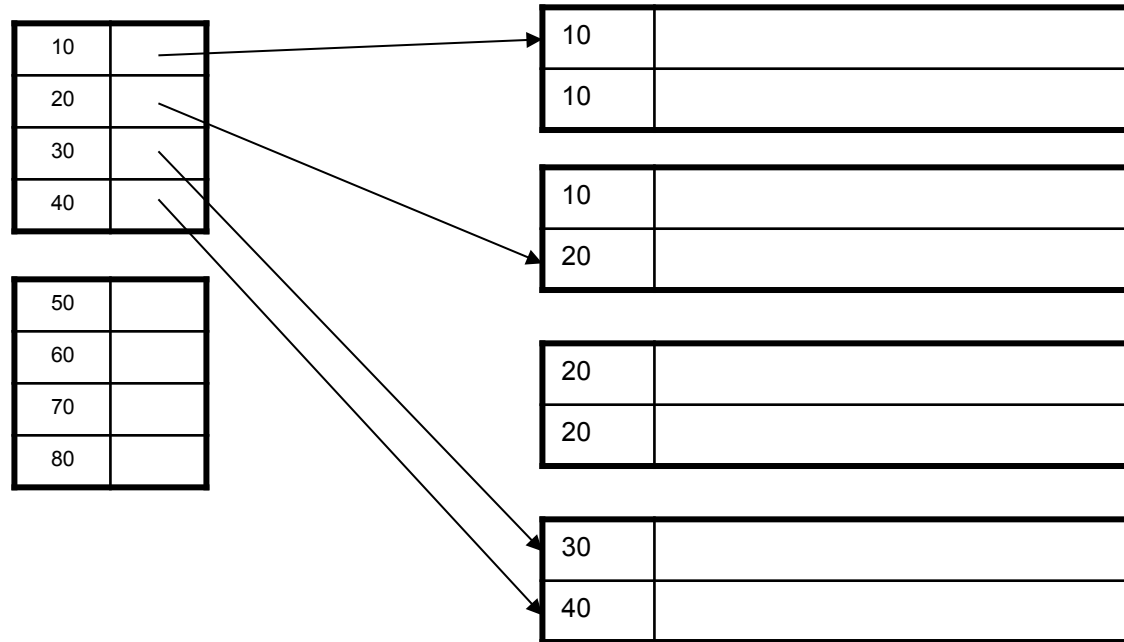
- Search for 20



- Search for 15 ? 35 ?

Primary Index with Duplicate Keys

- Dense index:

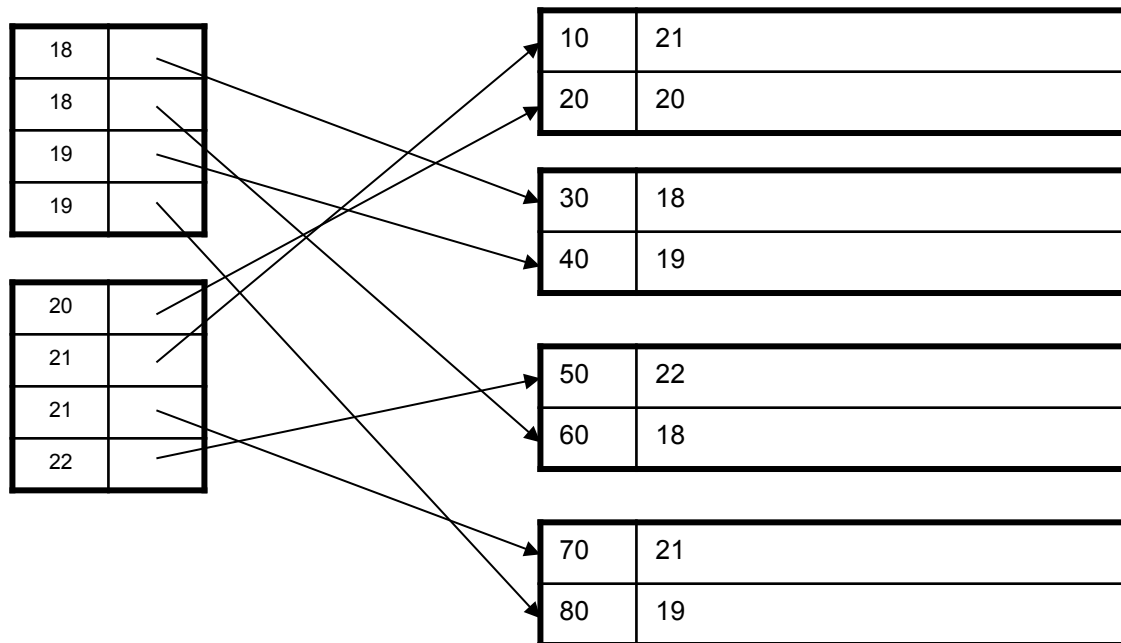


Primary Index Example

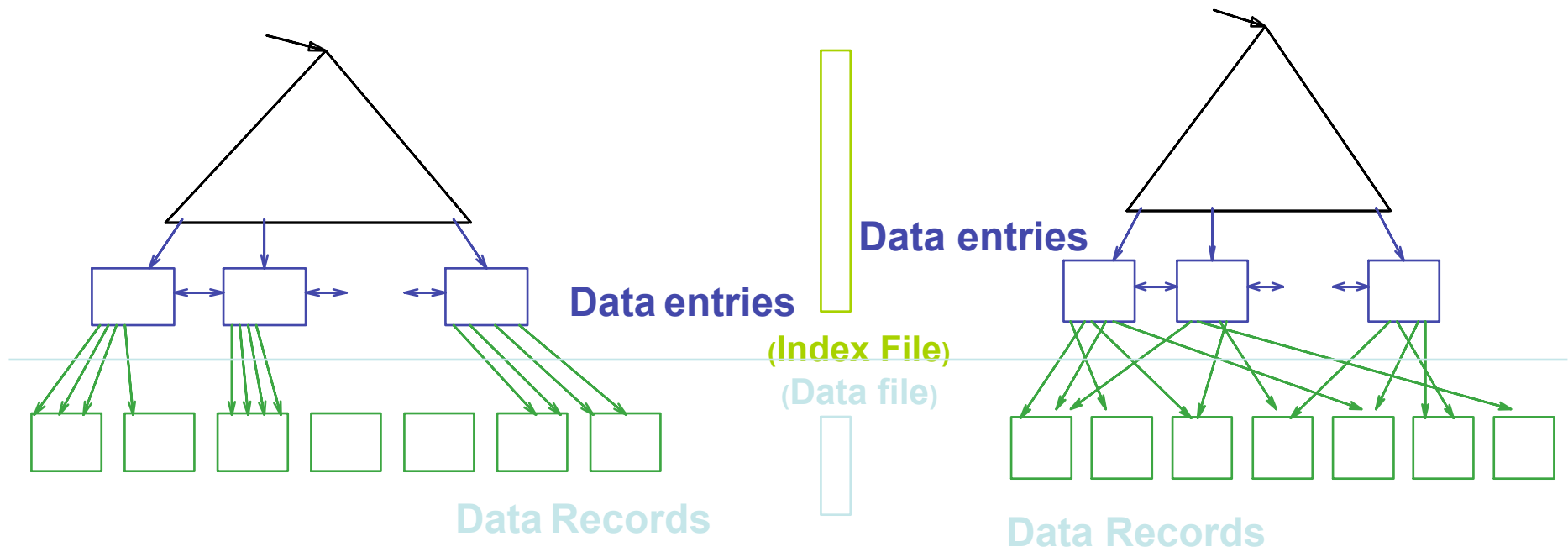
- Let's assume all pages of index fit in memory
- Find student whose sid is 80
 - Index (dense or sparse) points directly to the page
 - Only need to read 1 page from disk.
- Find all students older than 20
 - Must still read all 1,000 pages.
- How can we make *both* queries fast?

Secondary Indexes

- To index **other attributes than primary key**
- 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 other attributes than primary key
 - Index unsorted files (heap files)
 - Index clustered data

Index Classification Summary

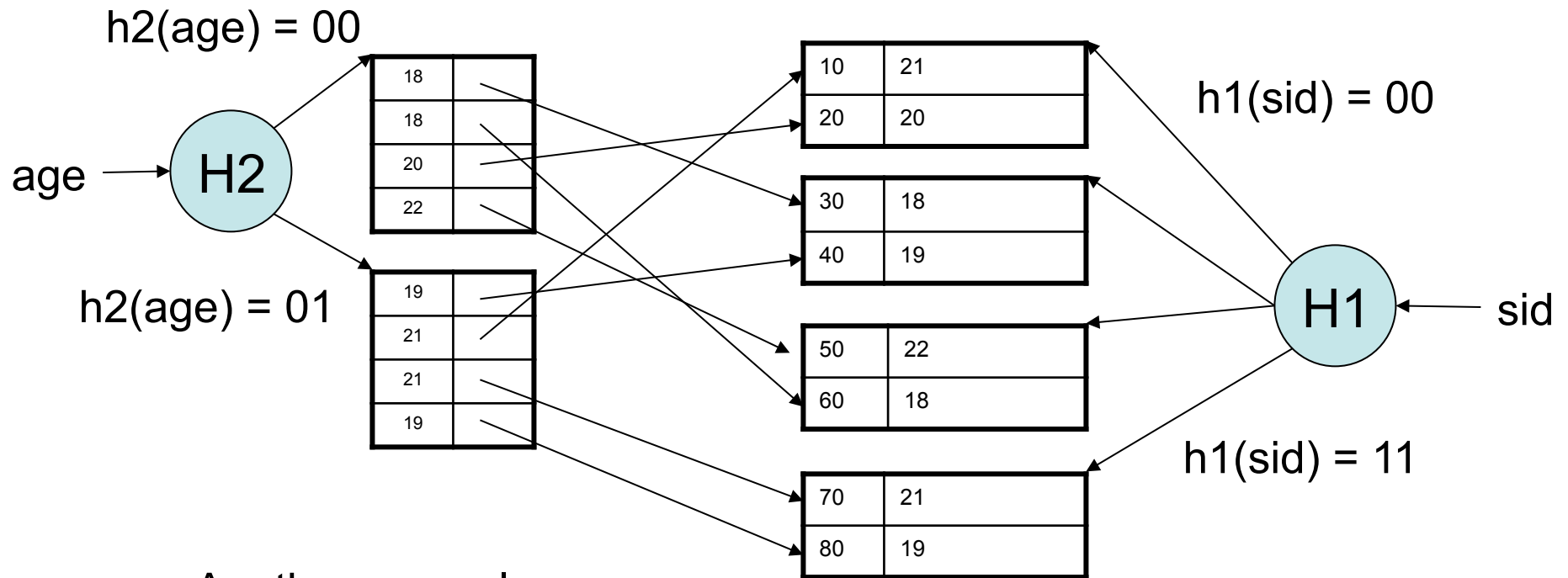
- **Primary/secondary**
 - Primary = determines the location of indexed records
 - Secondary = cannot reorder data, does not determine data location
- **Dense/sparse**
 - Dense = every key in the data appears in the index
 - Sparse = the index contains only some keys
- **Clustered/unclustered**
 - Clustered = 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



Another example
of secondary index

Another example of primary index

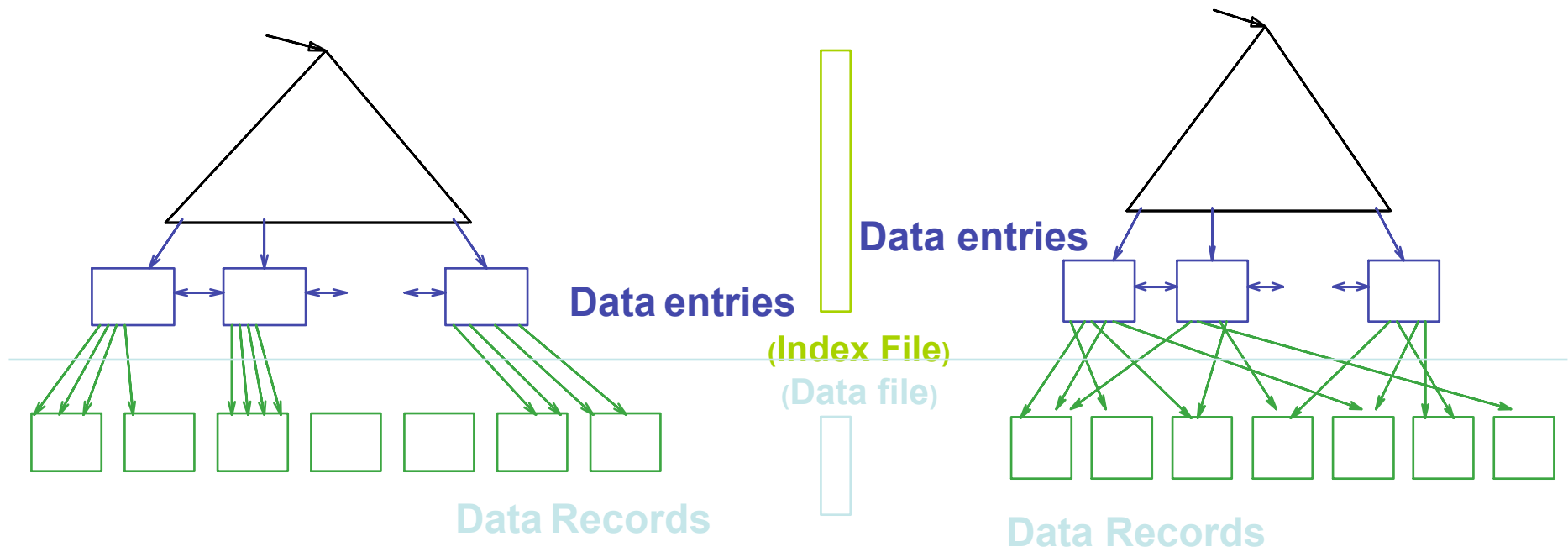
Tree-Based Index

- 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



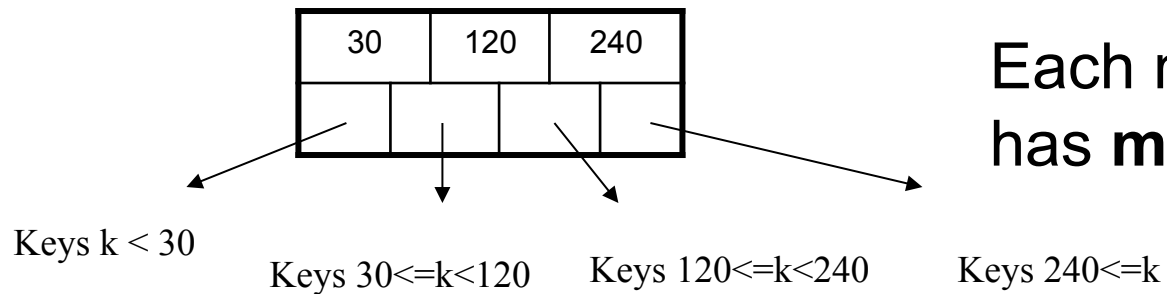
CLUSTERED

UNCLUSTERED

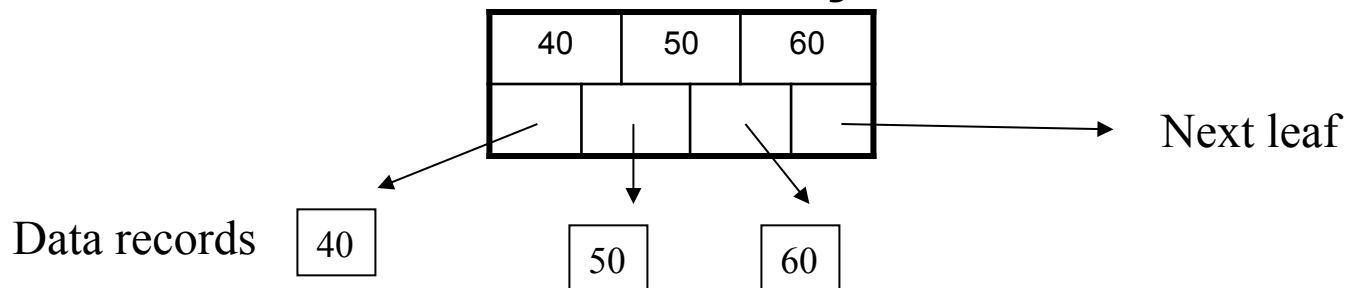
Note: can also store data records directly as data entries (primary index)

B+ Trees Basics

- Parameter $d = \text{the } \underline{\text{degree}}$
- Each node has $d \leq m \leq 2d$ keys (except root)



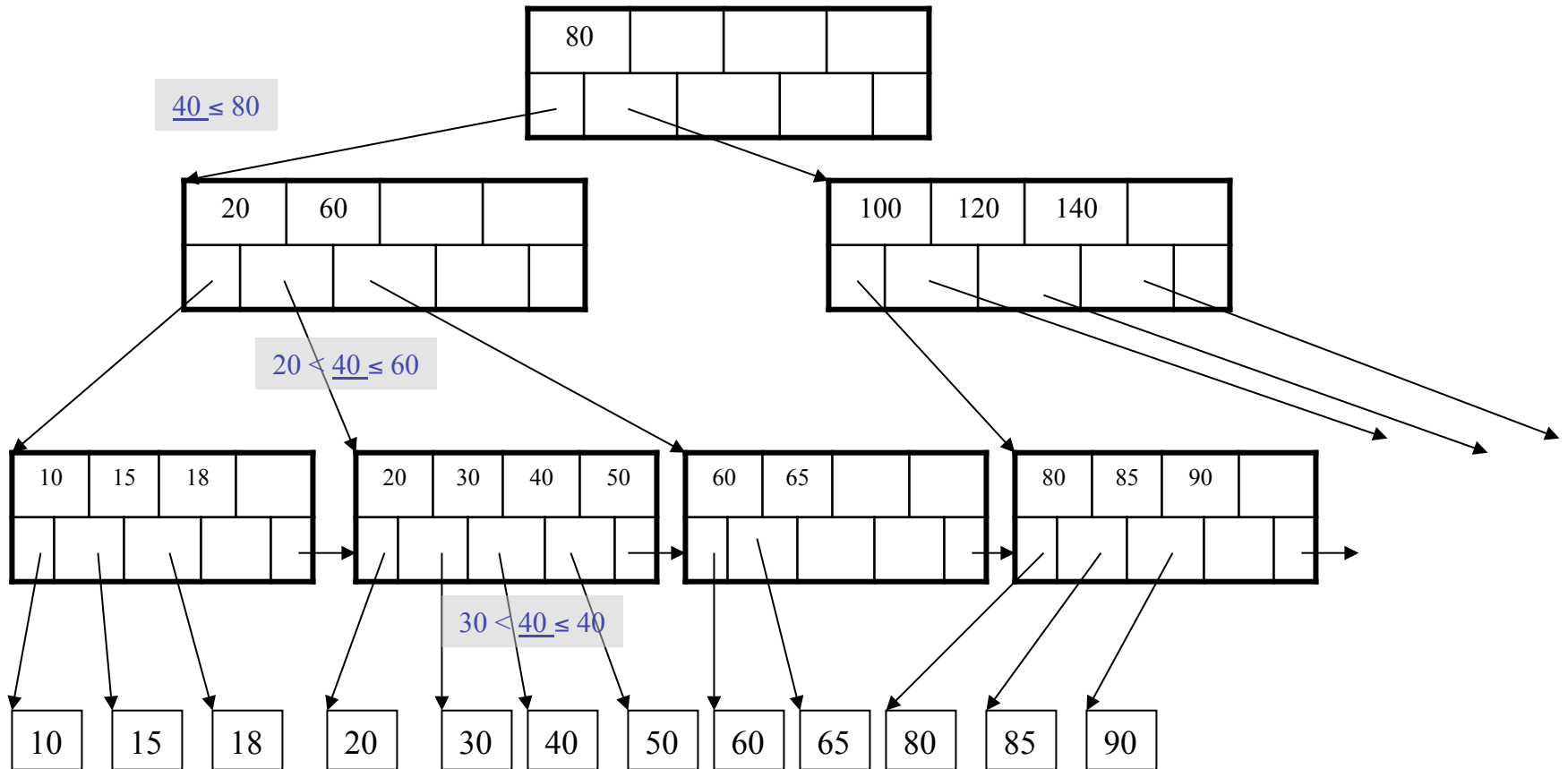
- Each leaf has $d \leq m \leq 2d$ keys:



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
- $2d \times 4 + (2d+1) \times 8 \leq 4096$
- $d = 170$

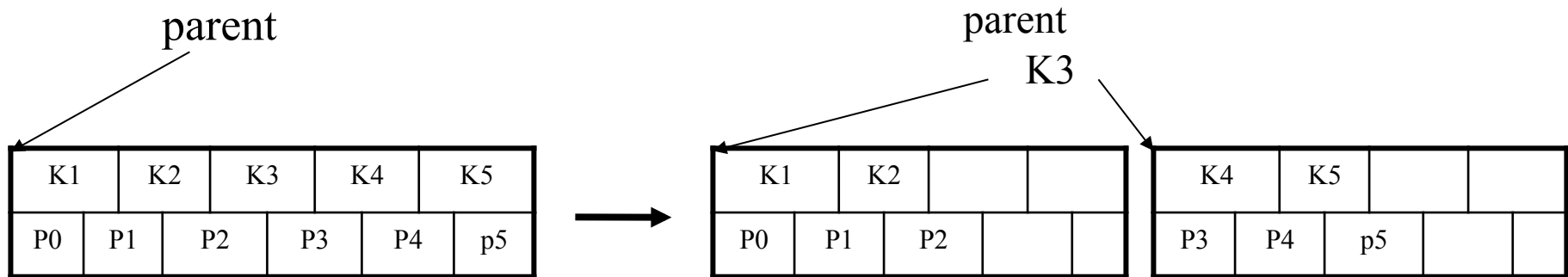
B+ Trees in Practice

- 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)

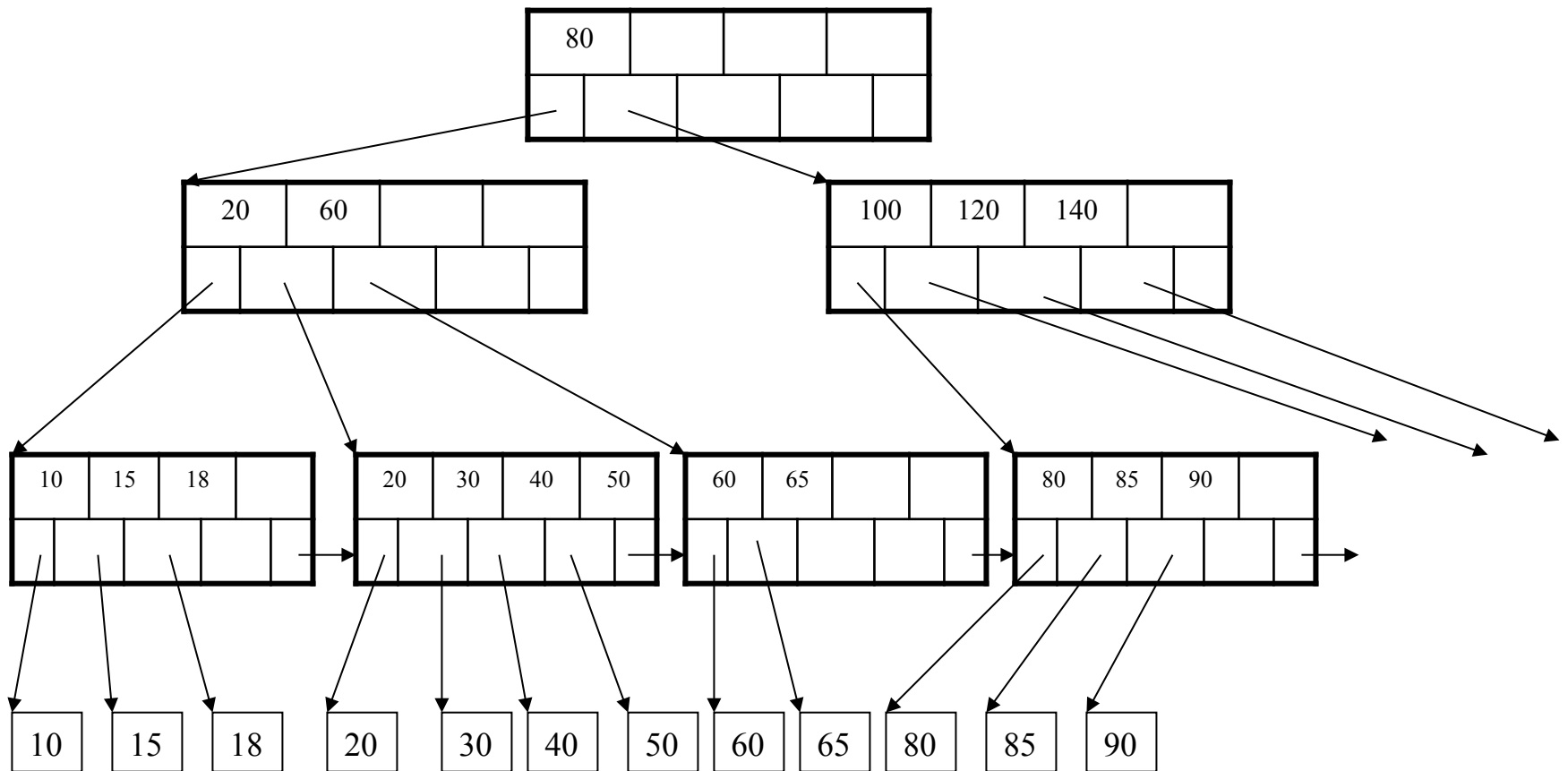
- Find leaf where K belongs, insert
- If no overflow ($2d$ keys or less), halt
- If overflow ($2d+1$ keys), split node, insert in parent:



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

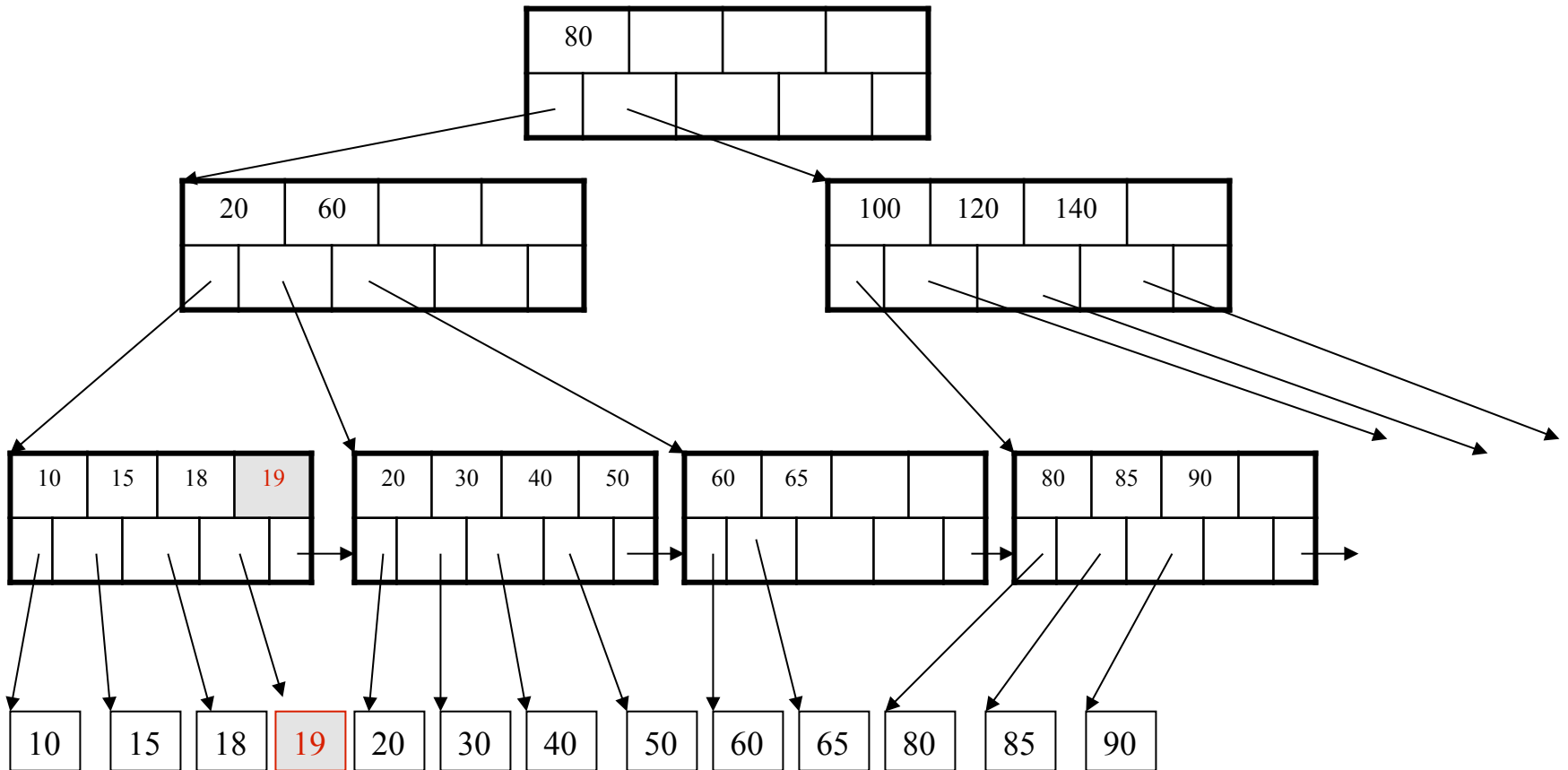
Insertion in a B+ Tree

Insert K=19



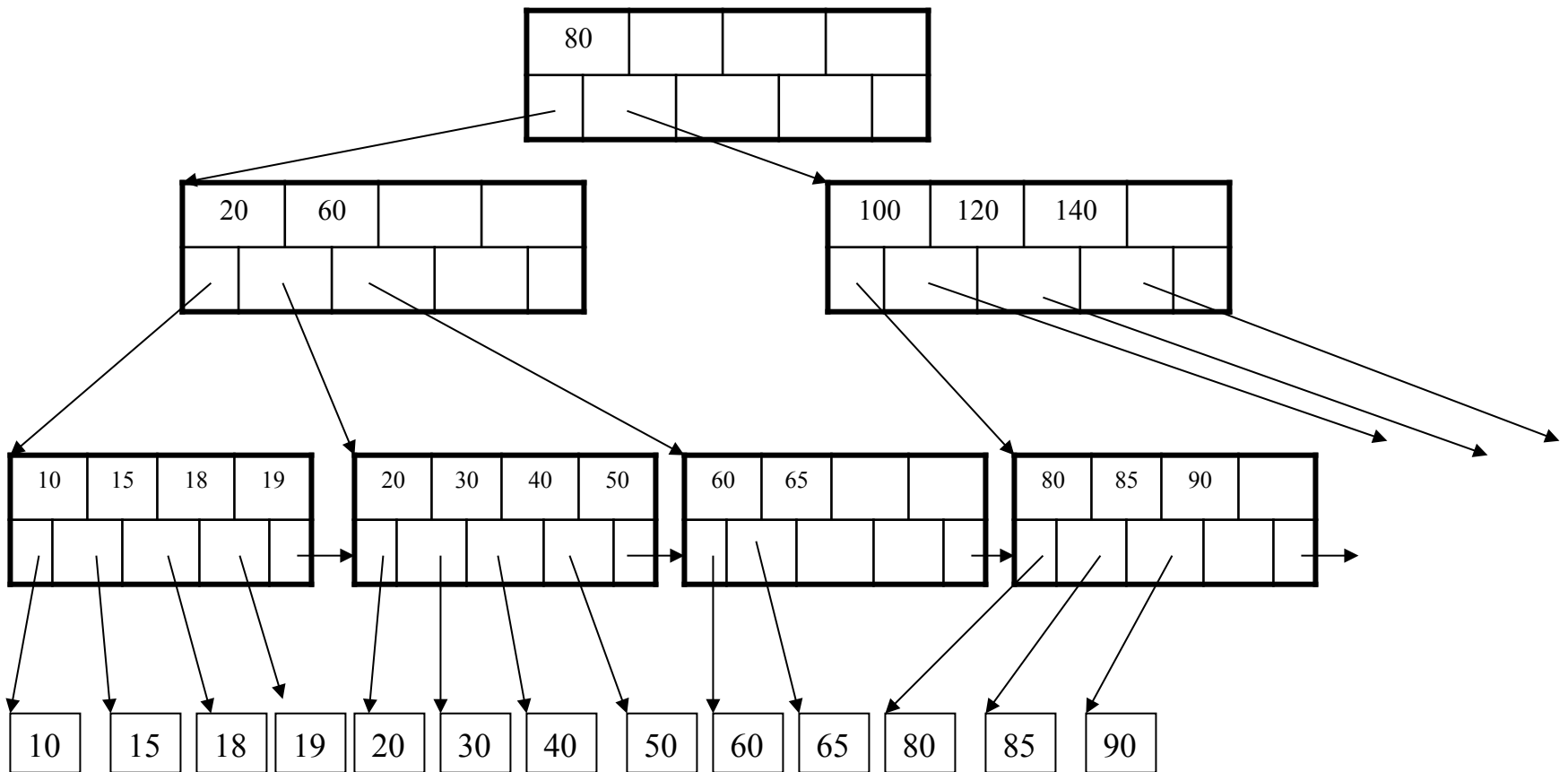
Insertion in a B+ Tree

After insertion



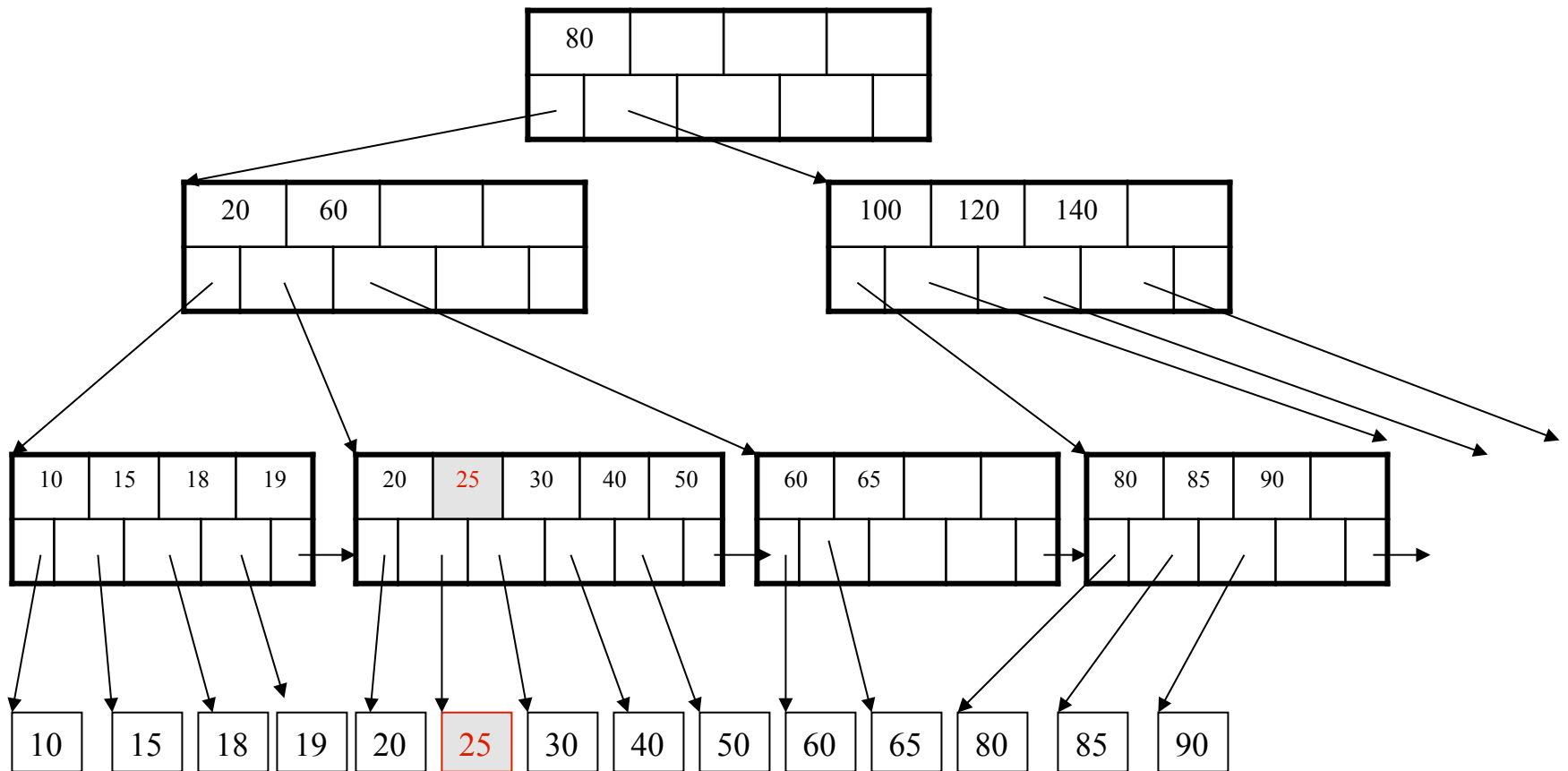
Insertion in a B+ Tree

Now insert 25



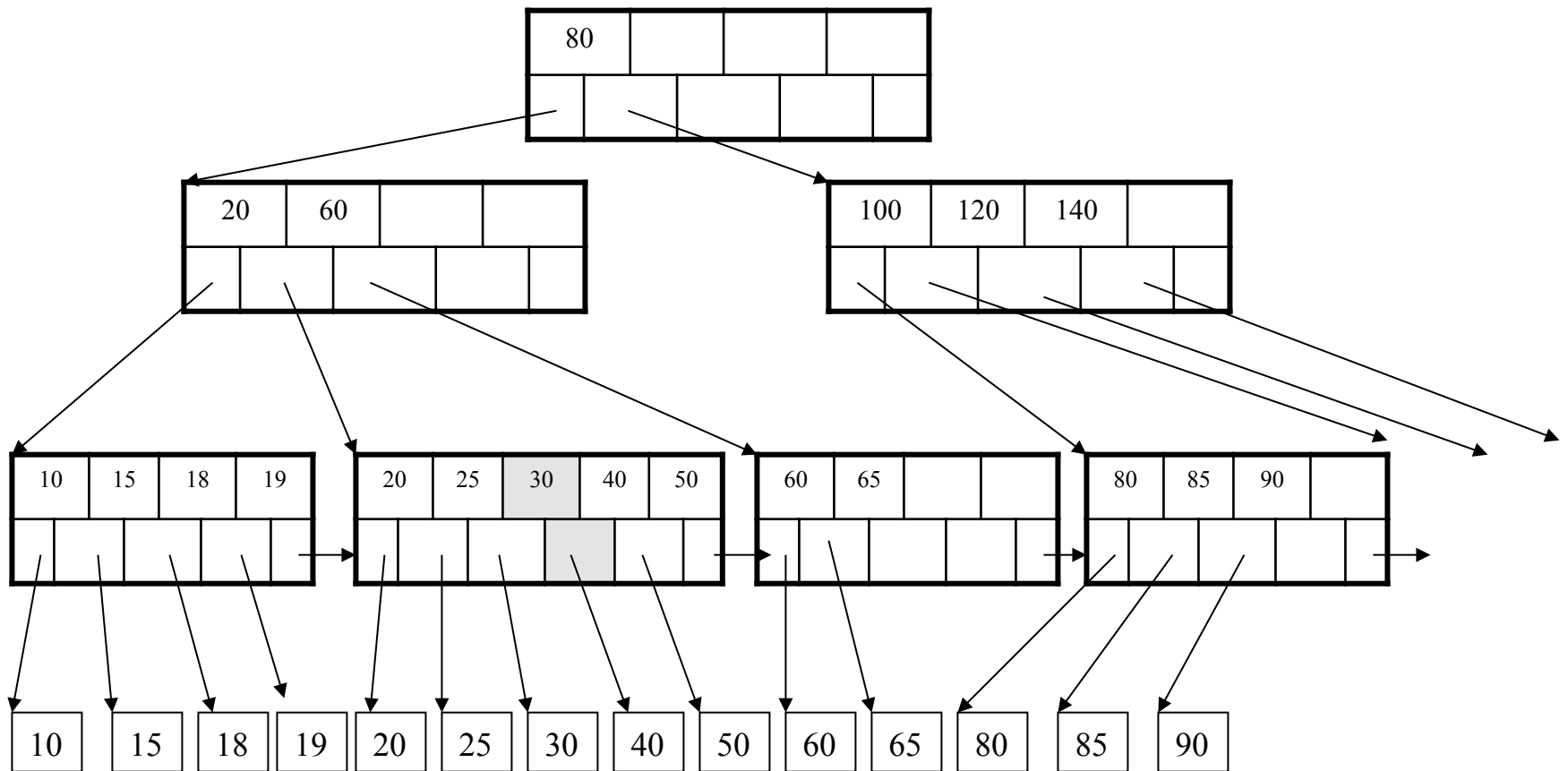
Insertion in a B+ Tree

After insertion



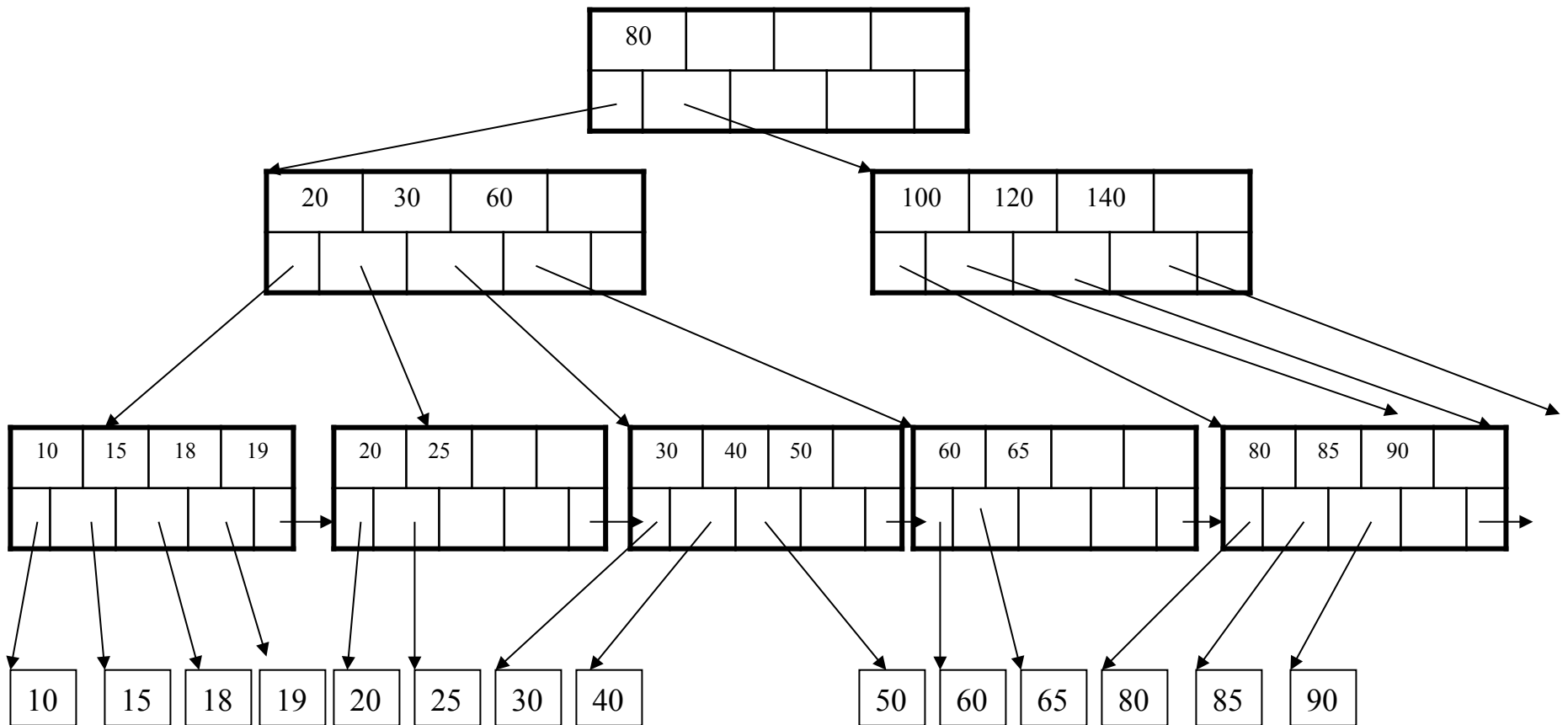
Insertion in a B+ Tree

But now have to split !



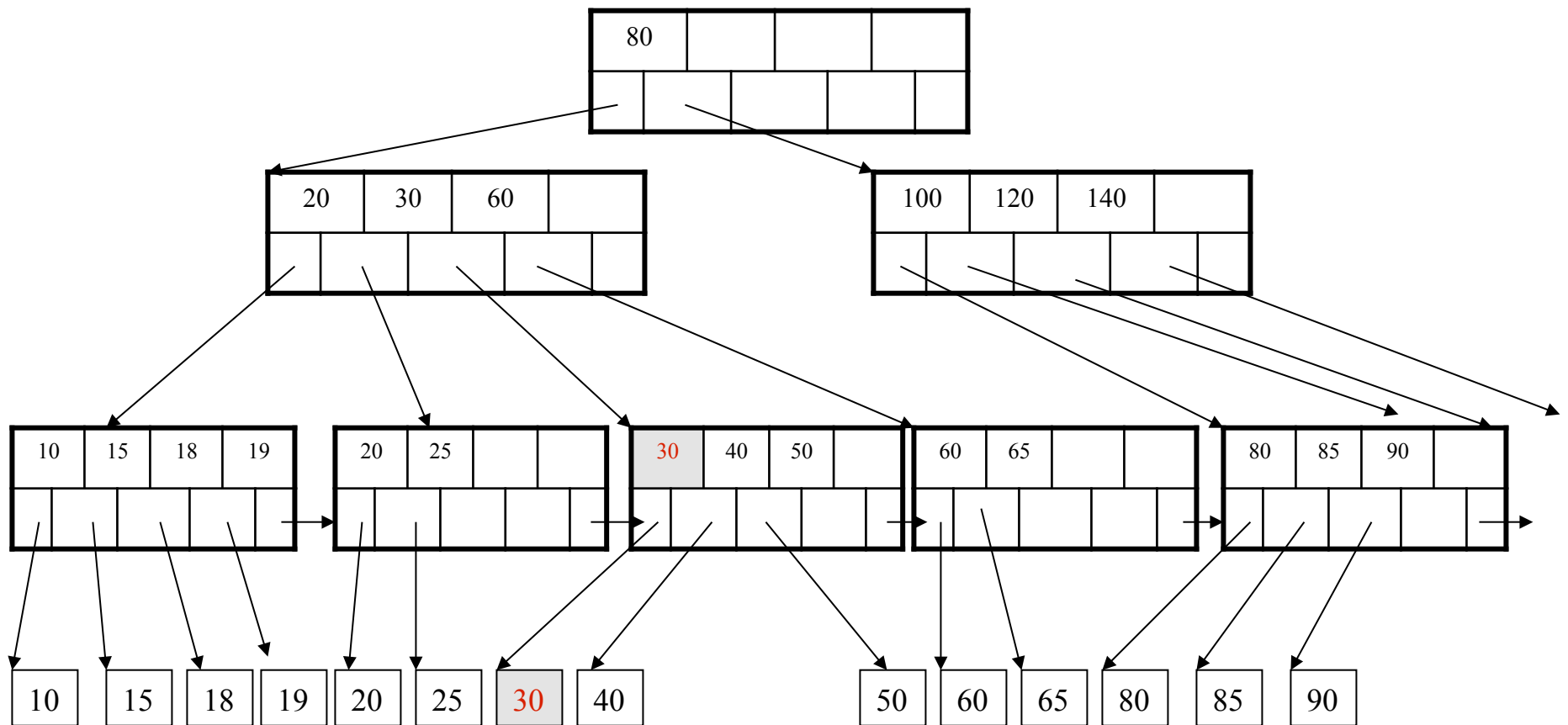
Insertion in a B+ Tree

After the split



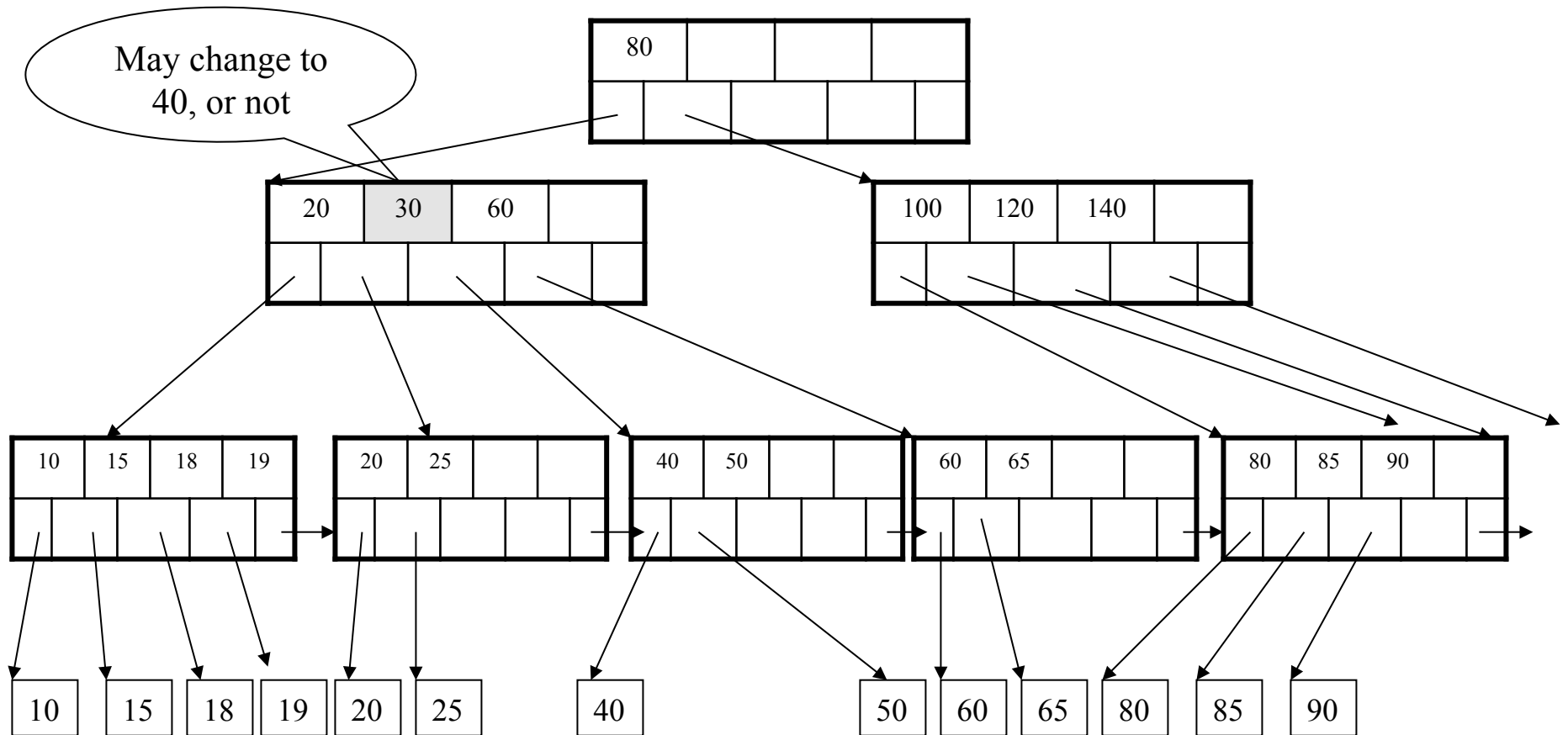
Deletion from a B+ Tree

Delete 30



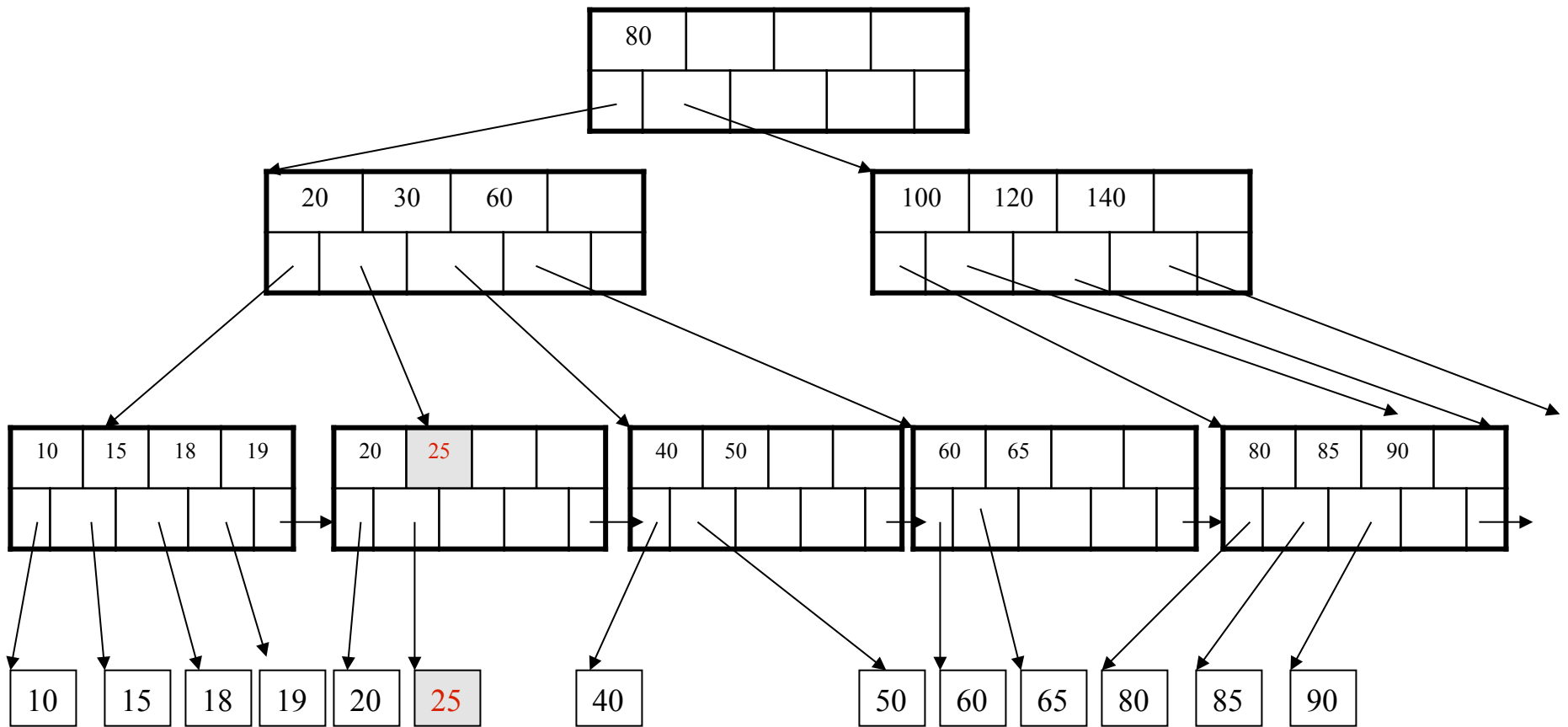
Deletion from a B+ Tree

After deleting 30



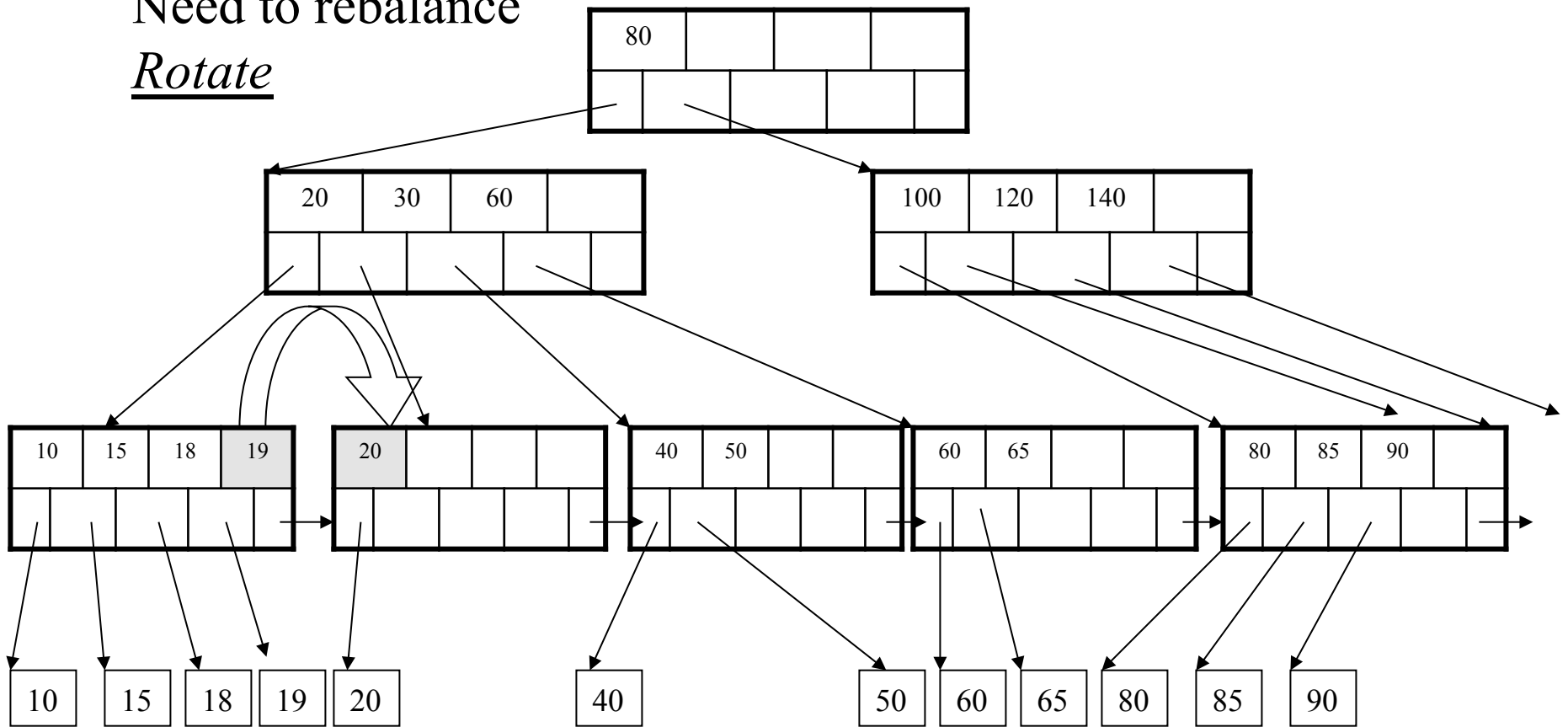
Deletion from a B+ Tree

Now delete 25



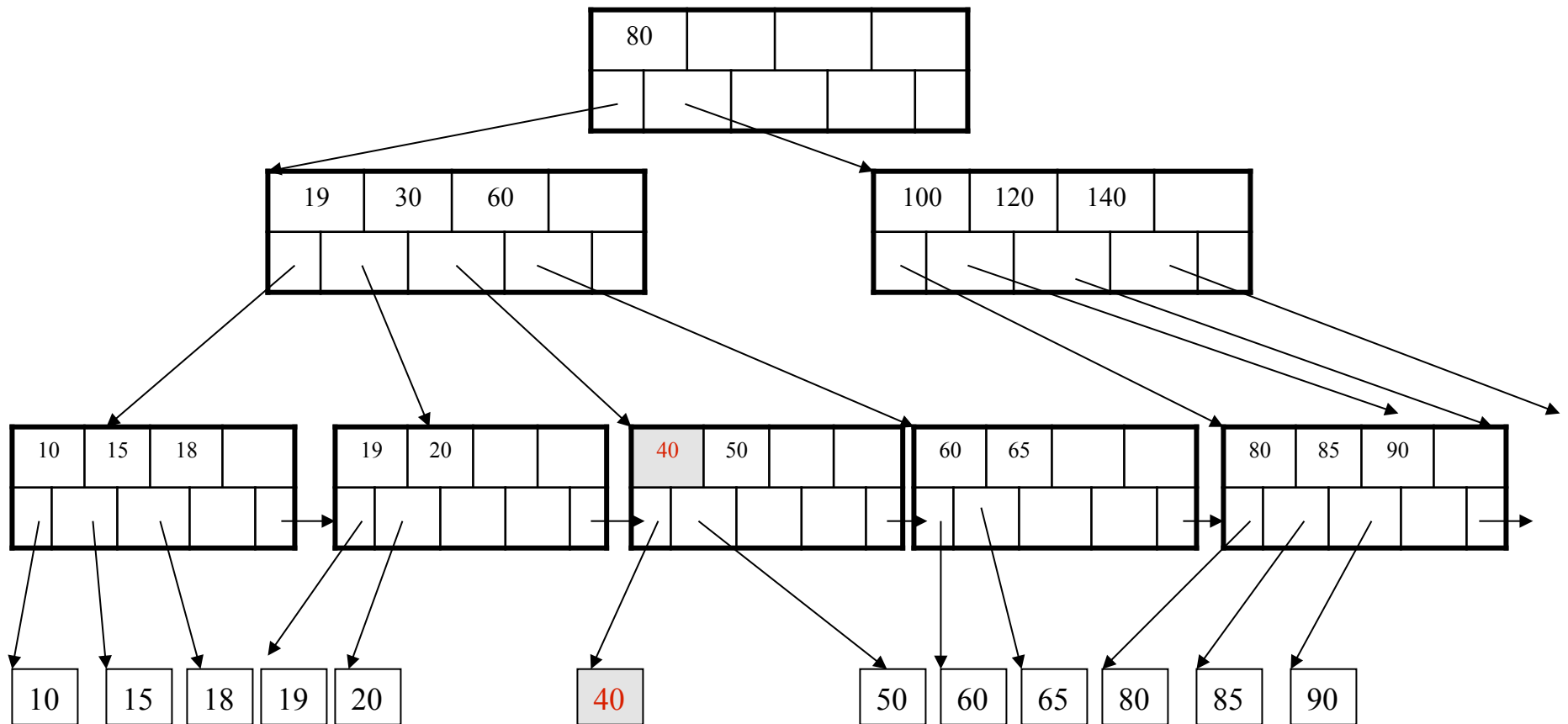
Deletion from a B+ Tree

After deleting 25
Need to rebalance
Rotate



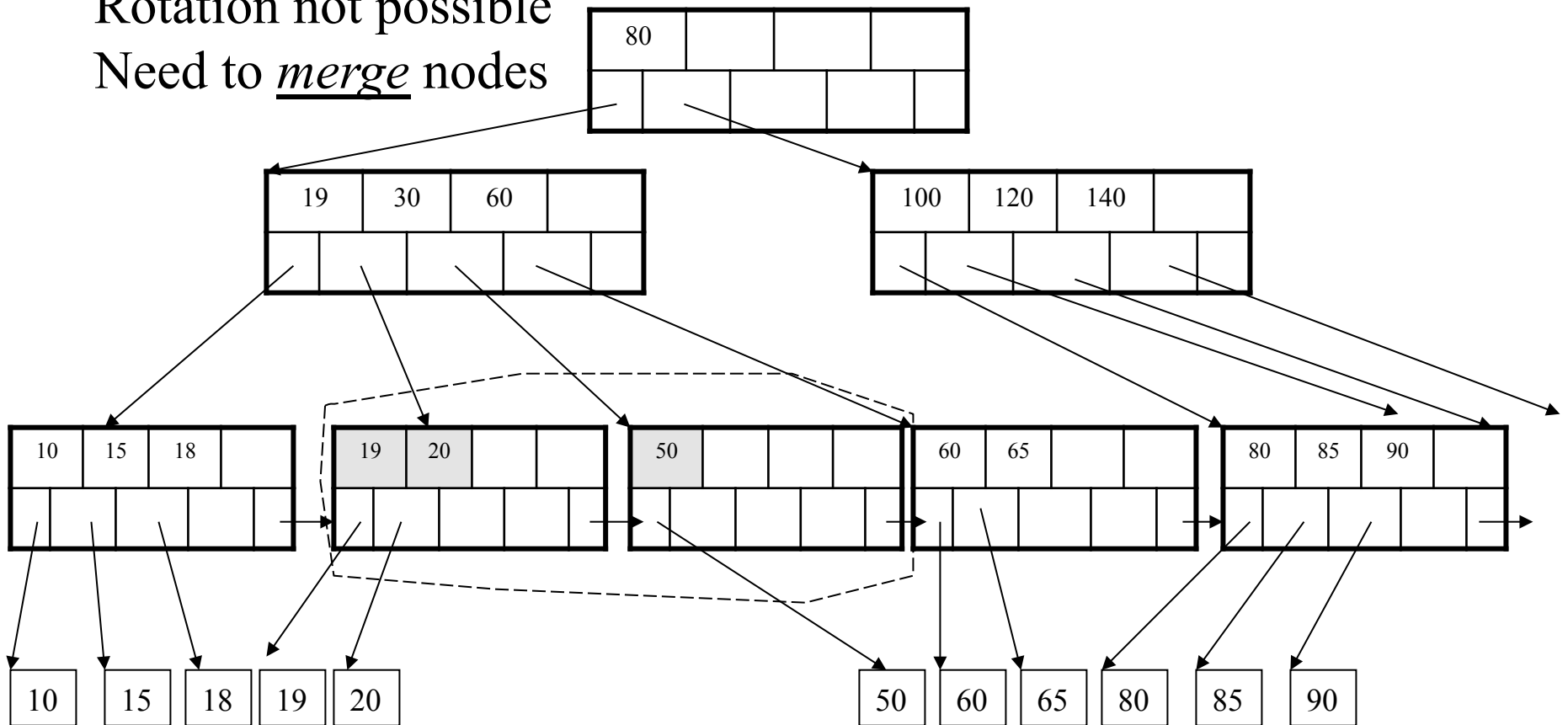
Deletion from a B+ Tree

Now delete 40



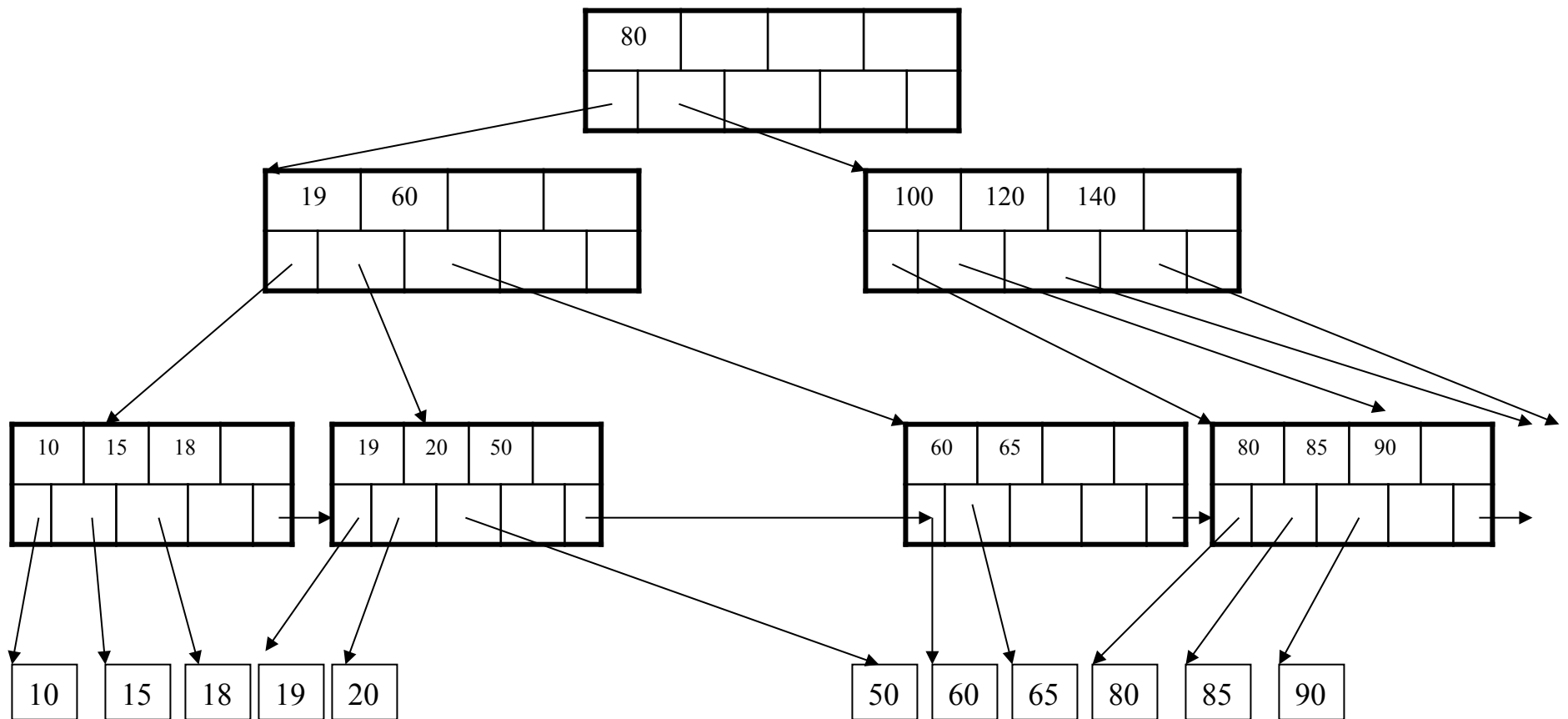
Deletion from a B+ Tree

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

Outline

- **Data storage**
 - Disk and files: Sections 9.3 through 9.7
 - Operations on files

- **Indexes**
 - Index structures: Section 8.3
 - Hash-based indexes: Section 8.3.1
 - B+ trees: Chapter 10
 - **GiST: Hellerstein et. al.'s VLDB'95**

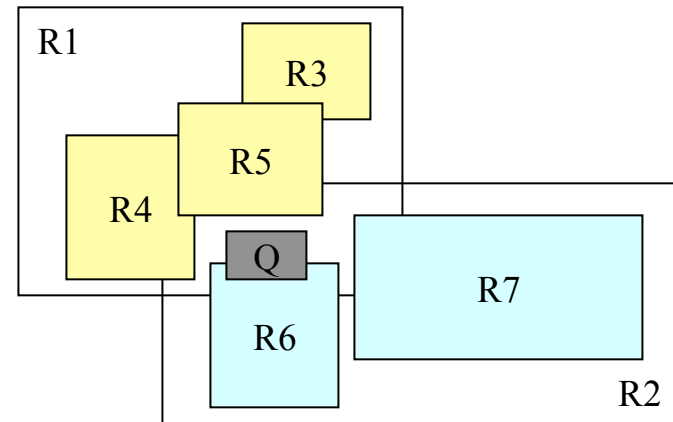
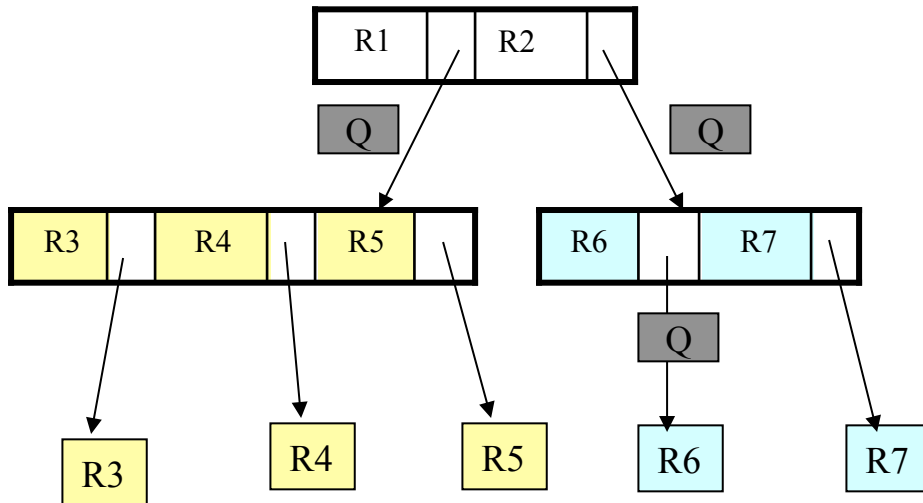
Motivation

- To better appreciate GiST, let's take a look at another type of index, the R tree
- R trees serve to index spatial data

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)

Generalized Search Tree (GiST)

- Goal: facilitate database extensibility
 - When adding a new data type
 - Want to add indexes for the data type
- Overview
 - GiST is an index structure
 - Basically, this is a **template** for indexes
 - Supports extensible set of queries and data types

GiST Key Insights

Canonical database search tree

- Balanced tree with high fanout
- Leaf nodes contain pointers to actual data
- Leaf nodes stored as a linked list
- Internal nodes used as a directory
 - Contain $\langle \text{key}, \text{pointers} \rangle$ pairs
 - If key consistent with query, data may be found if we follow pointer
 - **Generalized search key: predicate that holds for each entry below key**
 - B+-tree key is pair of integers $\langle a, b \rangle$ and predicate is $\text{Contains}([a, b], v)$
 - R-tree key is bounding box and predicate is also containment test
 - **Generalized search tree: hierarchy of partitions**

GiST Key Methods: Consistent

- Consistent(E,q)
 - Entry $E = (p, ptr)$ and query predicate q
 - Returns false if $p \wedge q$ can be guaranteed unsatisfiable
 - Returns true otherwise
- See Algo Search(R,q) [also FindMin(R,q) and Next(R,q,E)]

GiST Key Methods: Consistent

- In a B+-tree, query predicates q can be either
 - $\text{Contains}([x,y], v)$ returns true if $x \leq v < y$ and false otherwise
 - $\text{Equal}(x,v)$ returns true if $x = v$ and false otherwise
- In a B+-tree, $\text{Consistent}(E,q)$
 - $p = \text{Contains}([x_p, y_p], v)$
 - (1) $q = \text{Contains}([x_q, y_q], v)$ or (2) $q = \text{Equal}(x_q, v)$
 - For (1), return true if $(x_p < y_q) \wedge (y_p > x_q)$
 - For (2), return true if $x_p \leq x_q < y_p$
- In R-tree, Consistent returns true if bounding boxes overlap

GiST Key Methods

- **Penalty**
 - Used during insert operations to pick subtree where to insert
 - See algorithms **Insert(R,E,I)** and **ChooseSubtree(R,E,I)**
 - B+-tree: returns zero when value to insert falls within subtree range
 - R-tree: returns change in area
- **PickSplit**
 - Used to split nodes during insert operations
 - See algorithm **Split(R,N,E)**
 - B+-tree: half the entries go into left group and half into right group
 - R-tree: e.g., minimize total area of bounding boxes after split

GiST Key Methods

- Union
 - Once a key is inserted, need to adjust predicates at parent nodes
 - See algorithm `AdjustKeys(R,N)`
 - B+-tree: computes interval that covers all given intervals
 - R-tree: computes bigger bounding box
- `Compress/Decompress`: for storage performance