CSC553 Advanced Database Concepts

Tanu Malik

School of Computing

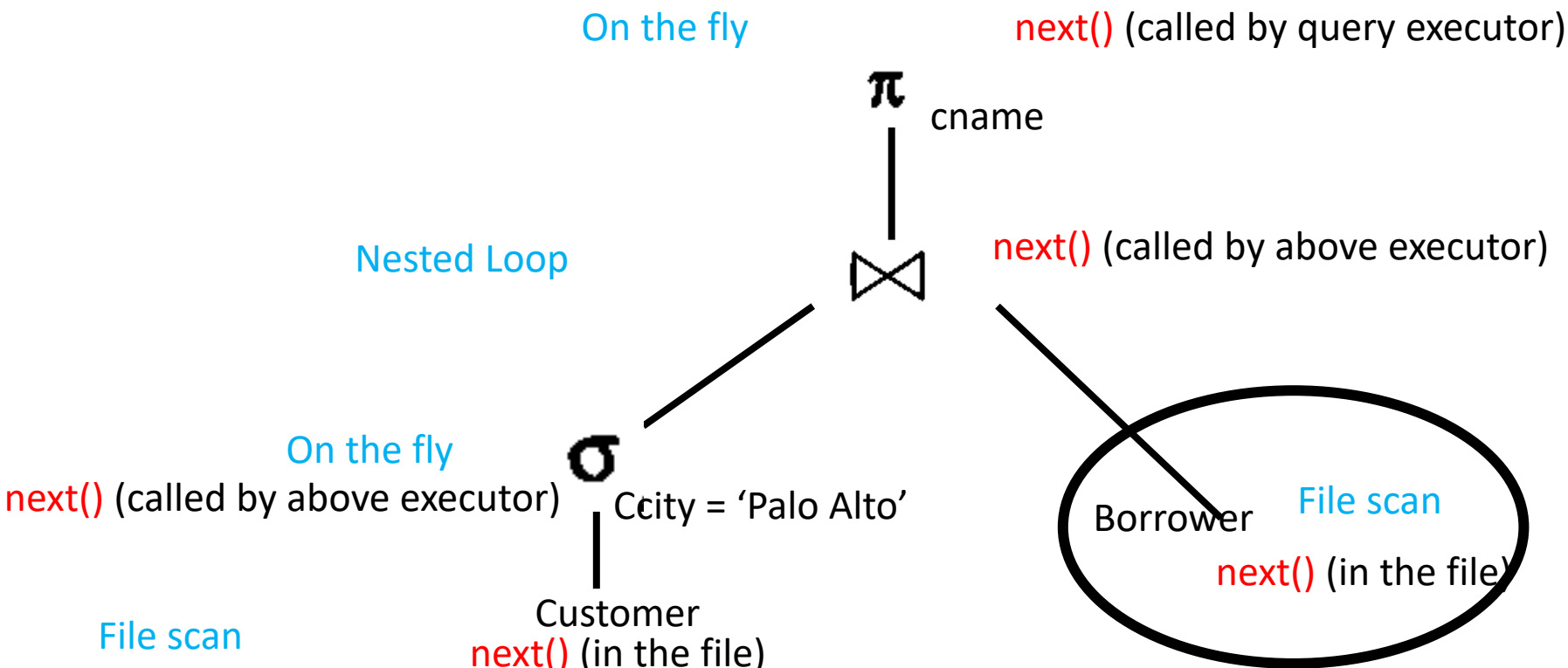
DePaul University

What we have learned so far

- Overview of the architecture of a DBMS
  - Reading: https://dice.cs.depaul.edu/553/courses/readings/anatomyofadatabase.pdf
- Access methods (scan)
  - Heap files
- Role of buffer manager
- Practiced the concepts in hw1 and lab1
- SQL to Relational algebra and some rules of query equivalence

Query Execution

- Pull-based execution



Next Lectures

- How to answer queries efficiently!
  - Physical query plans and operator algorithms
- How to automatically find good query plans
  - How to compute the cost of a complete plan
  - How to pick a good query plan for a query i.e., query optimization

- Lab 2 &3:
  - How to implement basic operator?
  - How to parse and optimize queries?

Index-based Access Methods

HeapFile In SimpleDB

- Data is stored on disk in an OS file. HeapFile class knows how to “decode” its content
- Control flow:
  - SeqScan calls methods such as "iterate" on the HeapFile Access Method
  - During the iteration, the HeapFile object needs to call the BufferManager.getPage() method to ensure that necessary pages get loaded into memory.
  - The BufferManager will then call HeapFile .readPage()/writePage() page to actually read/write the page.

HeapFile Access Method

API

- Create or destroy a file
- Insert a record
  - Requires a free vs full data structure
- Delete a record with a given rid (rid)
  - rid: unique tuple identifier (more later)
  - O(n)
- Search: Get a record with a given rid
  - O(n)
- Scan all records in the file

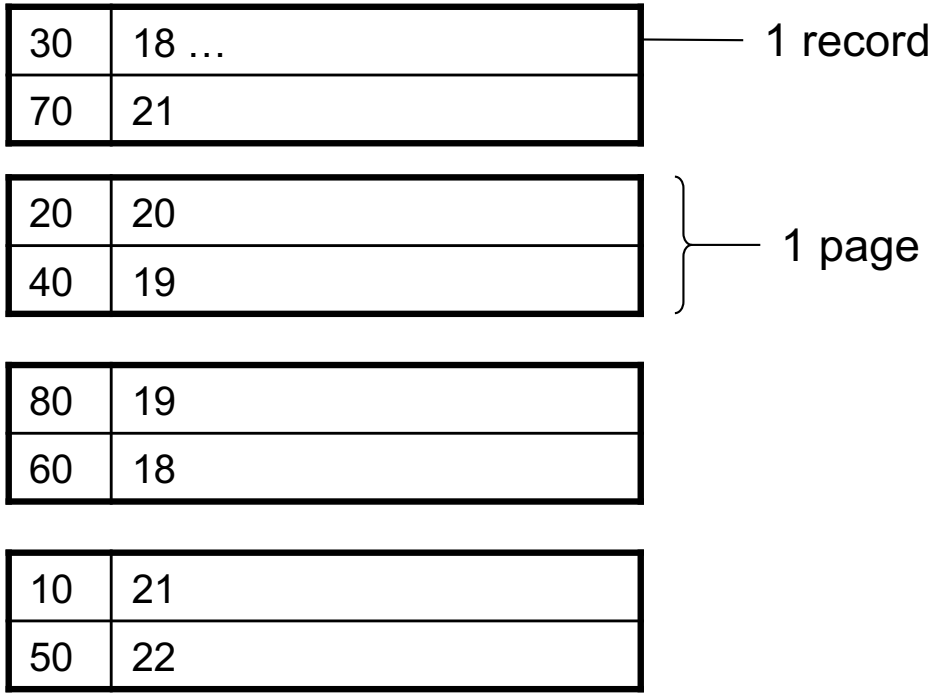
Motivation for Indexing

- Scan all records in the file that match a predicate of the form **attribute op value**
  - Example: Find all students with GPA > 3.5
- Critical to support such requests efficiently
- Why read all data form 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, ...)



Example

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

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

Example

- Total number of pages: 1,000 pages
- Find student whose sid is 80
  - ?
- Find all students older than 20
  - ?
- Can we do even better?

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

Creating Indexes in SQL

```
CREATE  TABLE    V(M int,   N varchar(20),    P int);
```

```
CREATE  INDEX V1 ON V(N)
```

```
CREATE  INDEX V2 ON V(P, M)
```

```
select *
from V
where P=55 and M=77
```

```
select *
from V
where P=55
```

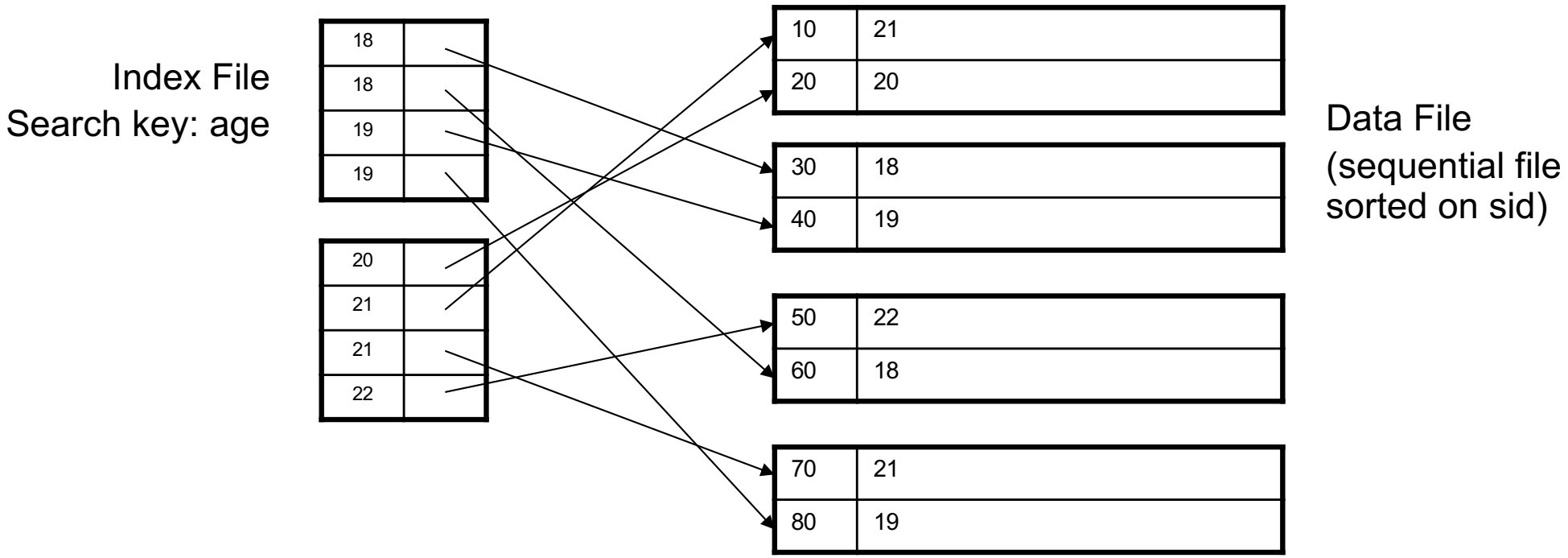
Outline

- Index structures
- Hash-based indexes
- B+ trees

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
  - Indexs can be in-memory or disk-based

Index on a Sequential Data File



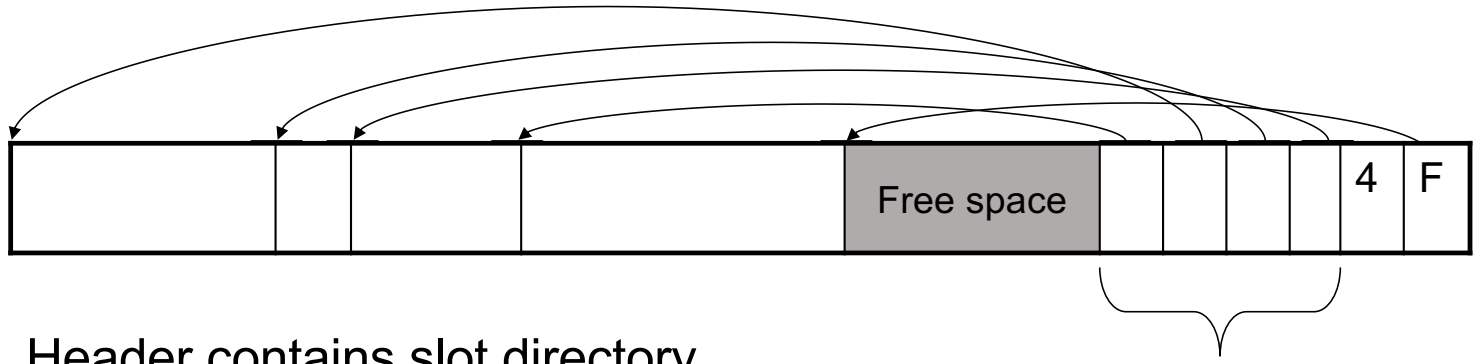
Example

- 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
  - Depends on index size
    - If in memory one disk record
    - Else $\log_2$(pages for an index)

Indexes

- **Search key** = can be any set of fields on which a query may specify a predicate
  - not the same as the primary key
- **Index** = collection of data entries
- **Data entry** for key k can be:
  - (k, RID)
  - (k, list-of-RIDs)
  - The actual record with key k
    - In this case, **the index is also a special file organization**
    - Called: “indexed file organization”

Indexed File Organization



Header contains slot directory
+ Need to keep track of # of slots
+ Also need to keep track of free space (F)

Slot directory

Each slot contains
<record offset, record length>

Can handle variable-length records

Can move tuples inside a page without changing RIDs

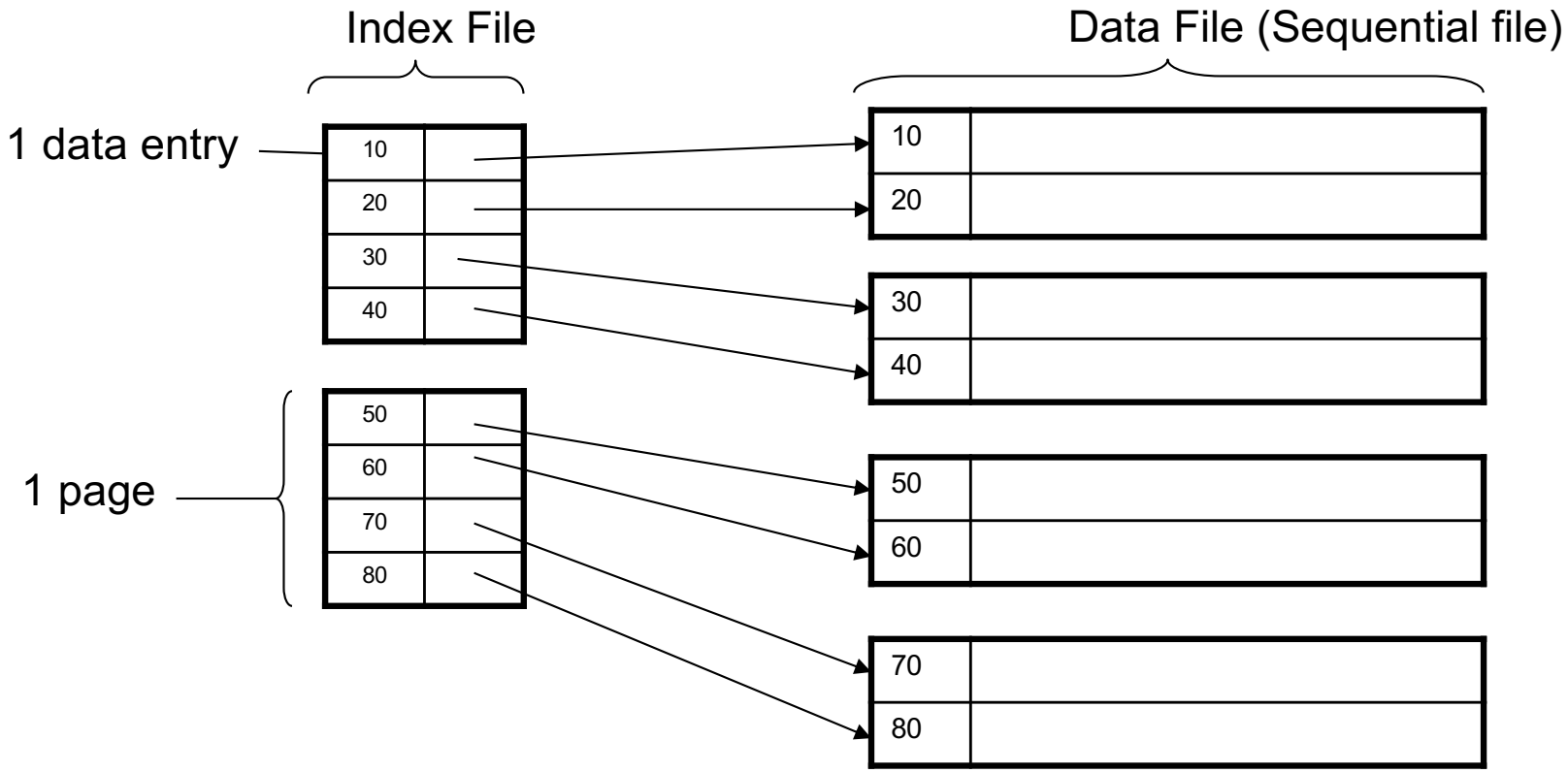
RID is (PageID, SlotID) combination

Different Types of Files

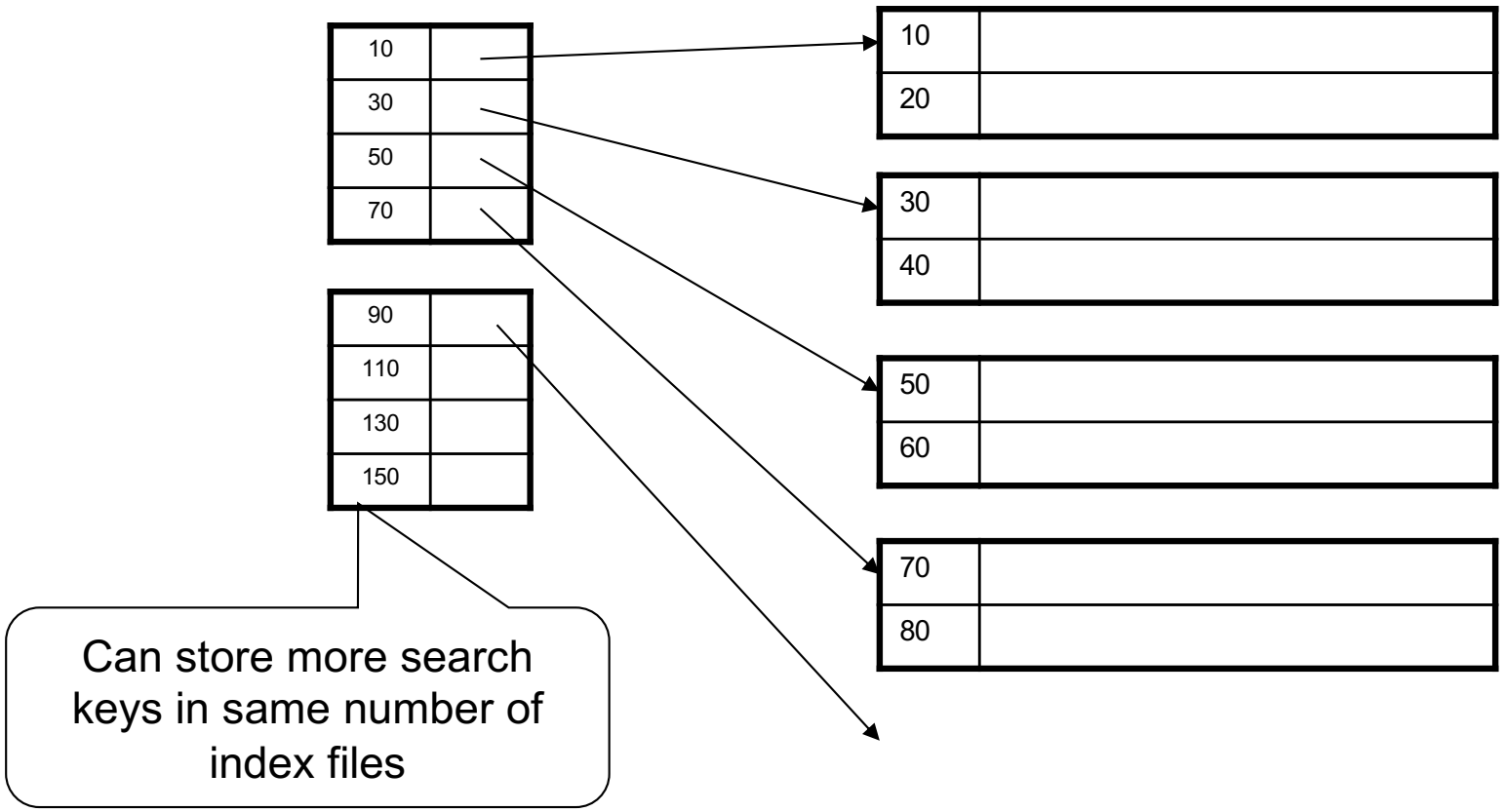
- For the data inside base relations:
  - Heap file (tuples stored without any order)
  - Sequential file (tuples sorted on some attribute(s))
  - Indexed file (tuples organized following an index)
- Then we can have additional index files that store (key,rid) pairs
- Index can also be a “covering index”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



- Sparse Index



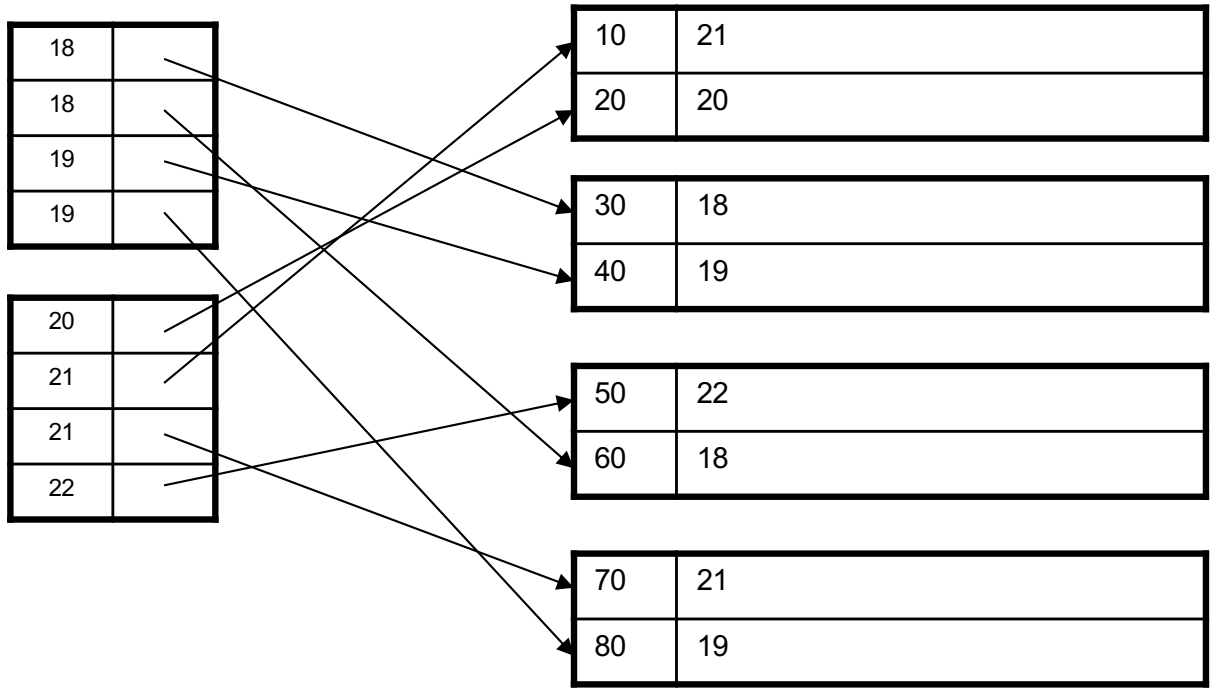
Example

- 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?

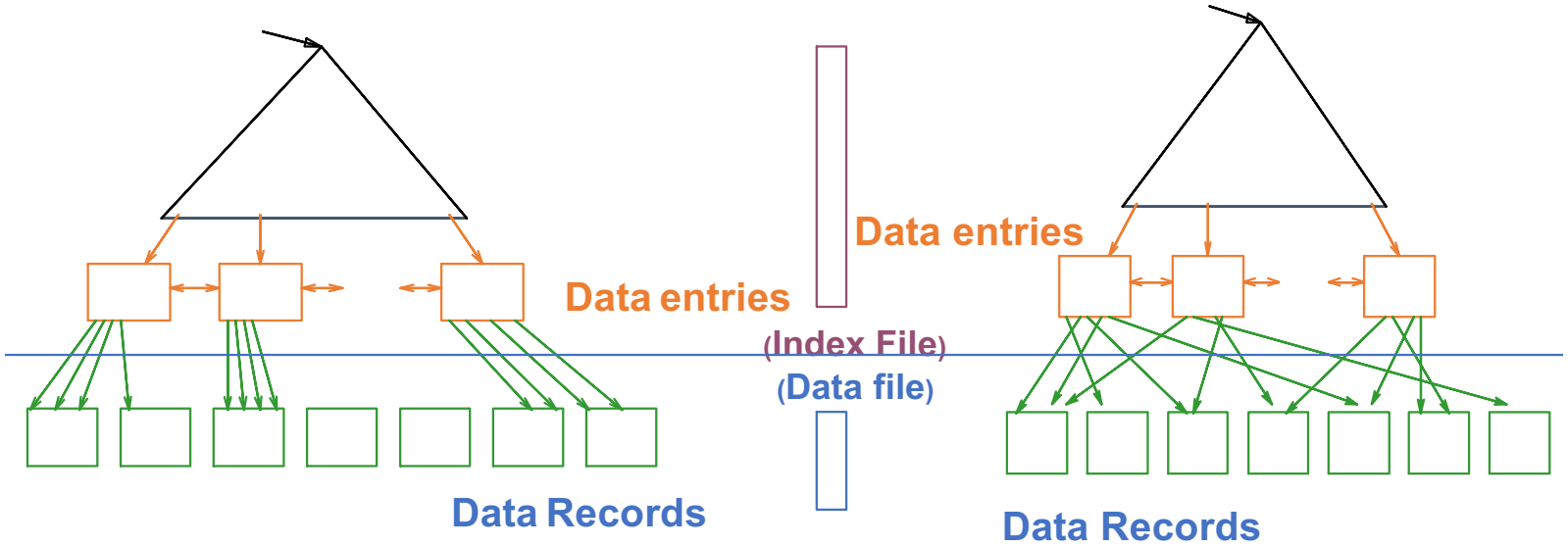
- How can we make *both* queries fast?

Secondary Index

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



Clustered Vs Unclustered Index



Clustered = records close in index are close in data

Clustered/Unclustered

- Primary index = clustered by definition
- Secondary indexes = usually unclustered
  - Possible that sorted order of the secondary index matches that of primary index, but hardly ever the case

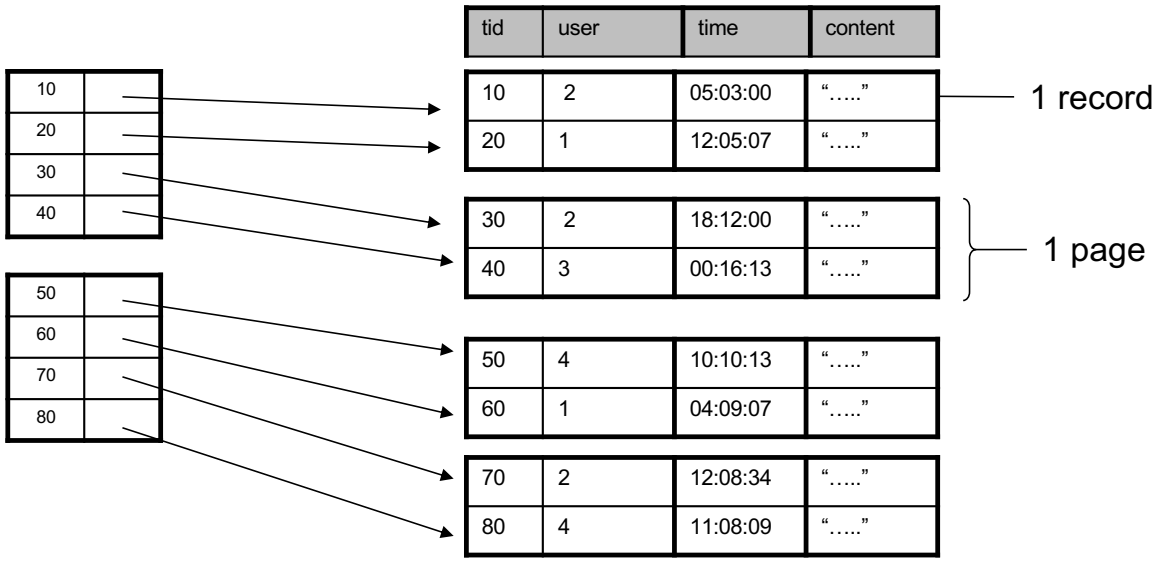
Secondary Index

- Applications
  - Index unsorted files (heap files)
  - When necessary to have multiple indexes
  - Index files that hold data from two relations

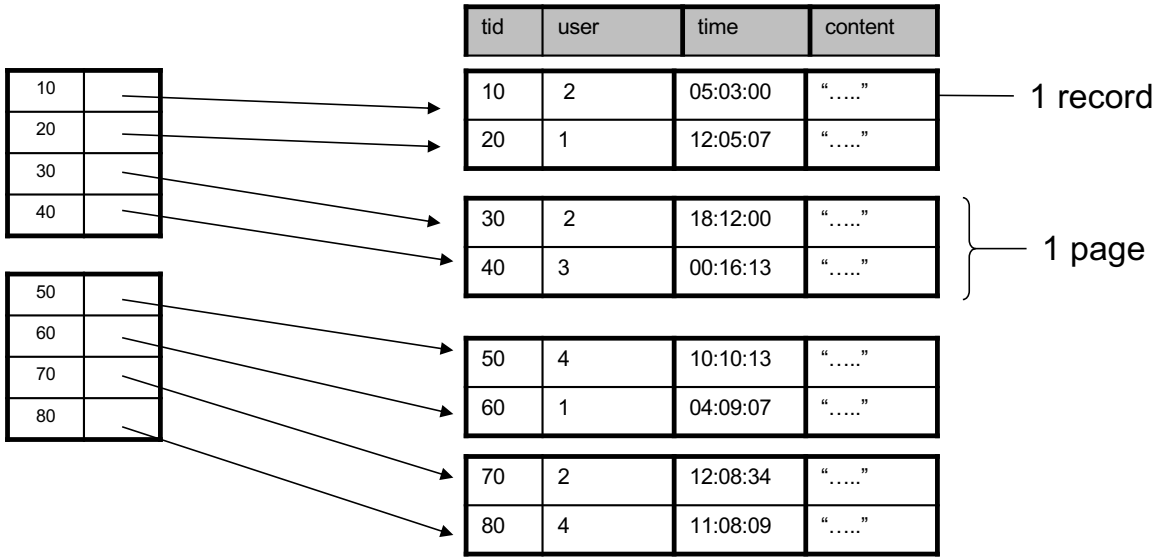
Index Classification Summary

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

What type of index?



Ex1: Primary Dense Index



- Dense: an "index key" for every database record
  - (In this case) every "database key" appears as an "index key"
- Primary: determines the location of indexed records
- Also, Clustered: records close in index are close in data

Improve further? Clustered Index can be made Sparse (normally one key per page)

Ex2. Draw a primary sparse index on “tid”

tid	user	time	content
10	2	05:03:00	“…..”
20	1	12:05:07	“…..”
30	2	18:12:00	“…..”
40	3	00:16:13	“…..”
50	4	10:10:13	“…..”
60	1	04:09:07	“…..”
70	2	12:08:34	“…..”
80	4	11:08:09	“…..”

1 record

1 page

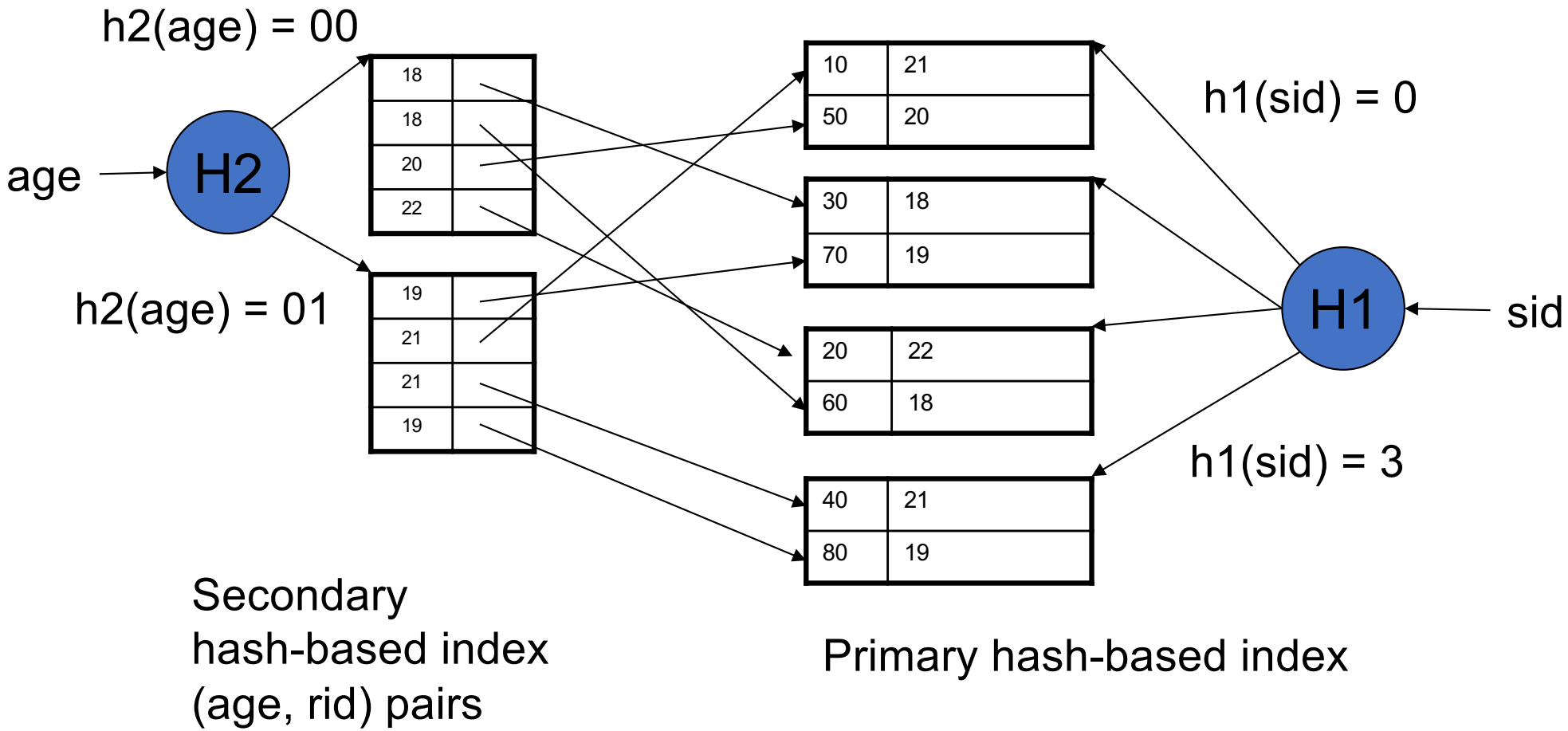
Large Indexes

- What if index does not fit in memory?

- Index the index itself!
  - Tree-based index
  - Hash-based index

Hash-based index

- Good for point queries but not range queries



Example

- Consider the following database schema:

Field Name	Data Type	Size on disk
Id (primary key)	INT	4 bytes
firstName	Char(50)	50 bytes
lastName	Char(50)	50 bytes
emailAddress	Char(100)	100 bytes

Compute

- Let default block size is **1024** bytes.

Let total records in the database = **5,000,000**

- Length of each record =
- How many disk blocks are needed to store this data set =

- Suppose you want to find the person with a
- particular **id** (say 5000)
  Assume data file sorted on primary key
- What is the cost of doing so with:
  - Linear search:
  - Binary search:
  - Index search with index pointer taking 4 bytes.

- Now, suppose you want to find the person having **firstName = ‘Alexa’** Here, the column isn’t sorted and does not hold a unique value.

- What is the cost of searching for the records?

- Solution: Create an index on the **firstName** column
- The schema for an index on **firstName** is:
- **Field Name Data Type Size on disk**
- **firstName** Char(50) 50 bytes
- **(record pointer)** Special 4 bytes

- Total records in the database = **5,000,000**
- Length of each index record = 4+50 = **54 bytes** Let the default block size be **1,024 bytes**
- Therefore,
  We will have 1024/54 = **18 records** per disk block
- Also, No. of blocks needed for the entire table = 5000000/18 = **277,778 blocks**

- Now, a binary search on the index will result in
- $\log_2 277778 = 18.08 =$ **19 block accesses**.
- Also, to find the address of the actual record, which requires a further block access to read, bringing the total to 19 + 1 = **20 block accesses**.
- Thus, indexing results in a much better performance as compared to searching the entire database.

B+ Tree Index

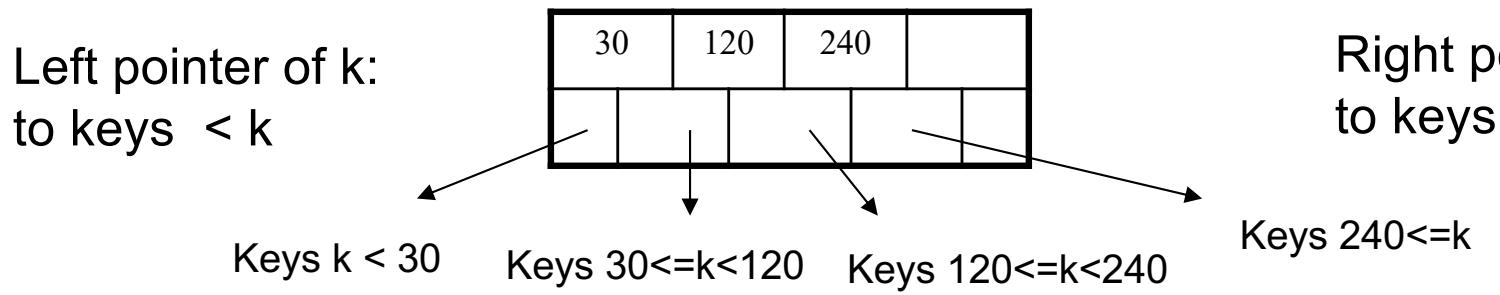
- How many index levels do we need?
- Can we create them automatically? Yes!
- Can do something even more powerful!

B-tree Vs B+-tree

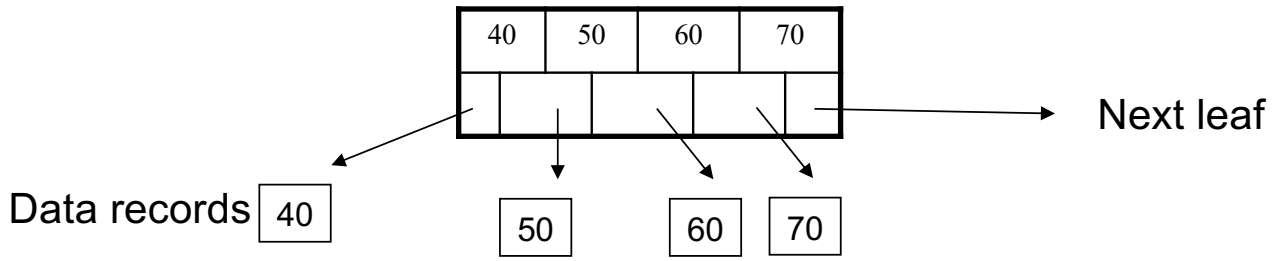
- Search trees
- Idea in B Trees
  - Make 1 node = 1 page (= 1 block)
- Idea in B+ Trees
  - Keep tree balanced in height – dynamic rather than static
  - Make leaves into a linked list: facilitates range queries

Basics

- Parameter d = the *degree*
- Each node has **d <= m <= 2d keys** (except root)
- Each node also has **m+1 pointers**



- Each leaf has **d <= m <= 2d keys**:



Leaf node:

- Left pointer from key = k: to the block containing data with value k in that attribute
- Last remaining pointer on right: To the next leaf on right

B+ Tree Properties

- For each node except the root, maintain 50% occupancy of keys
- Insert and delete must rebalance to maintain constraints

Operations

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

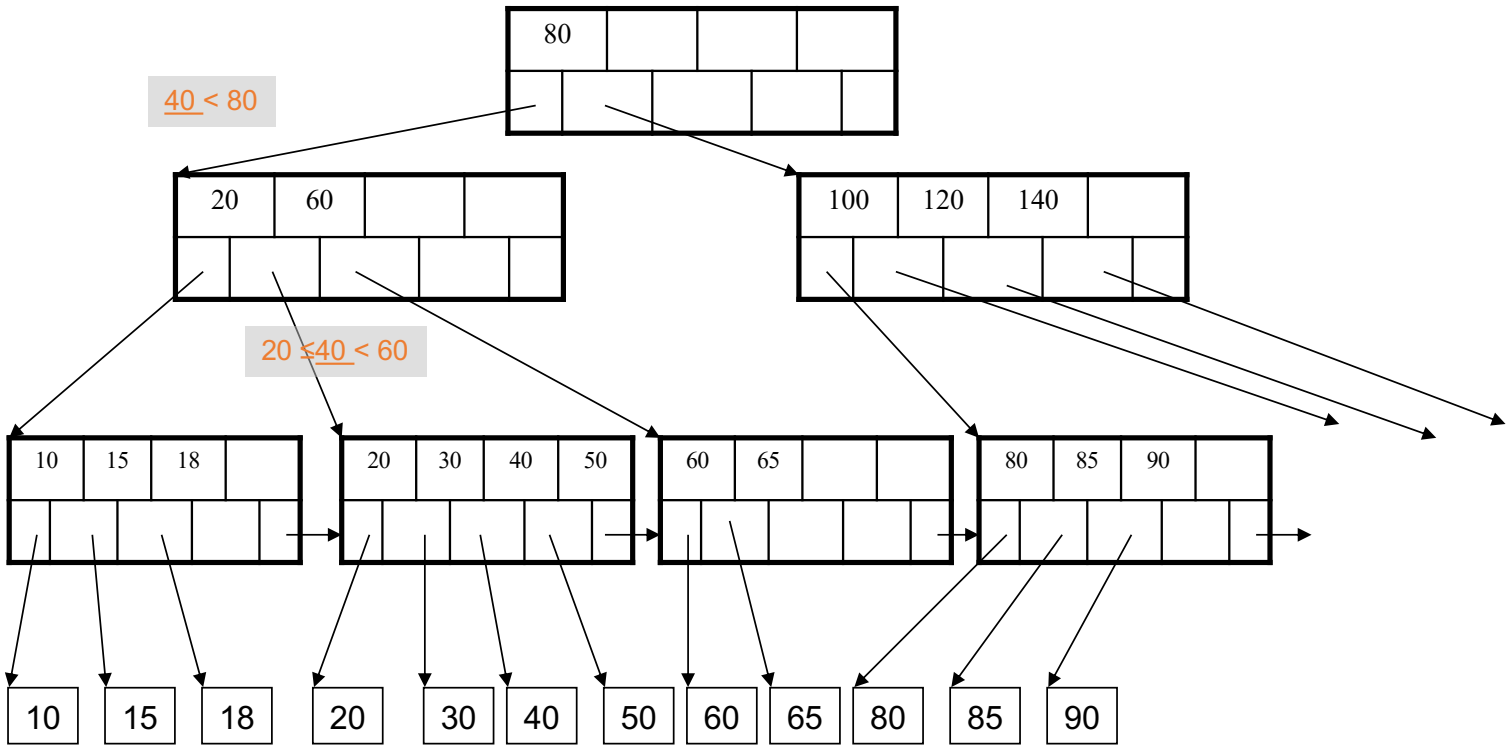
```
Select name
From Student
Where age = 25
```

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

Example

d = 2

Find the key 40



- How large d ? One B+ tree node fits on one block
- Example:
  Key size = 4 bytes , Pointer size = 8 bytes, Block size = 4096 bytes

- $2dx4 +(2d+1)x8 <= 4096$

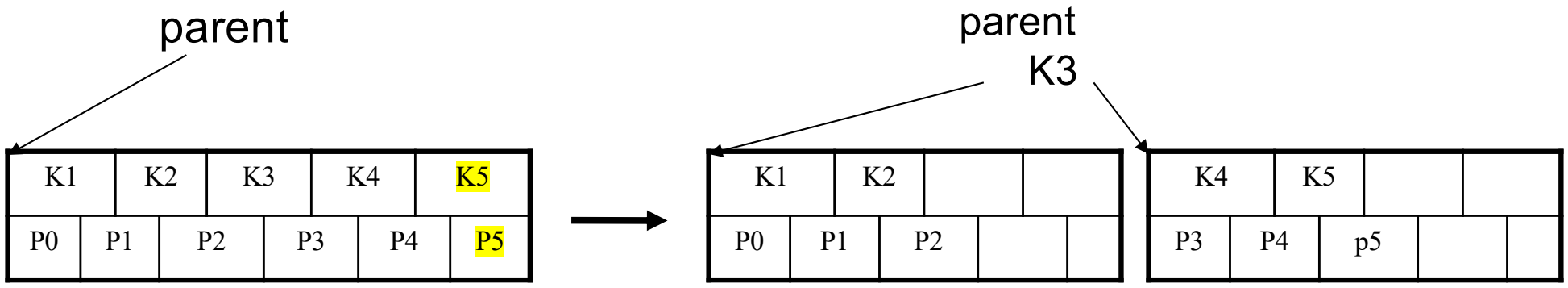
- **d = 170**

Space consumption of B+ tree in practice

- Typical order: 100. Typical fill-factor: 67%.
  - average fanout = 133
- Typical capacities
  - Height 4: 1334 = 312,900,700 records
  - Height 3: 1333 = 2,352,637 records
- Can often hold top levels in buffer pool
  - Level1= 1page = 8Kbytes
  - Level2= 133pages= 1Mbyte
  - Level 3 = 17,689 pages = 133 Mbytes

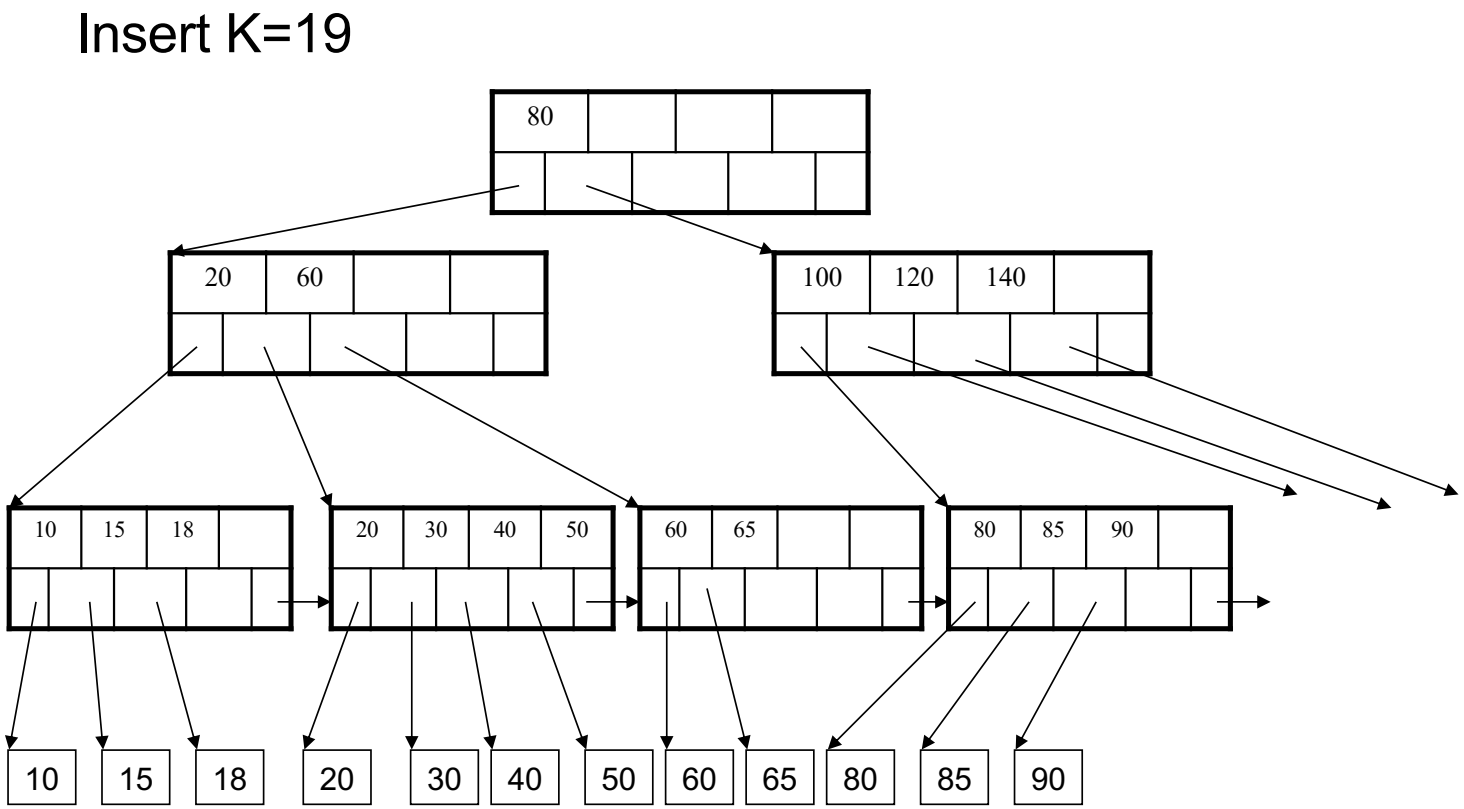
Insert

- 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:

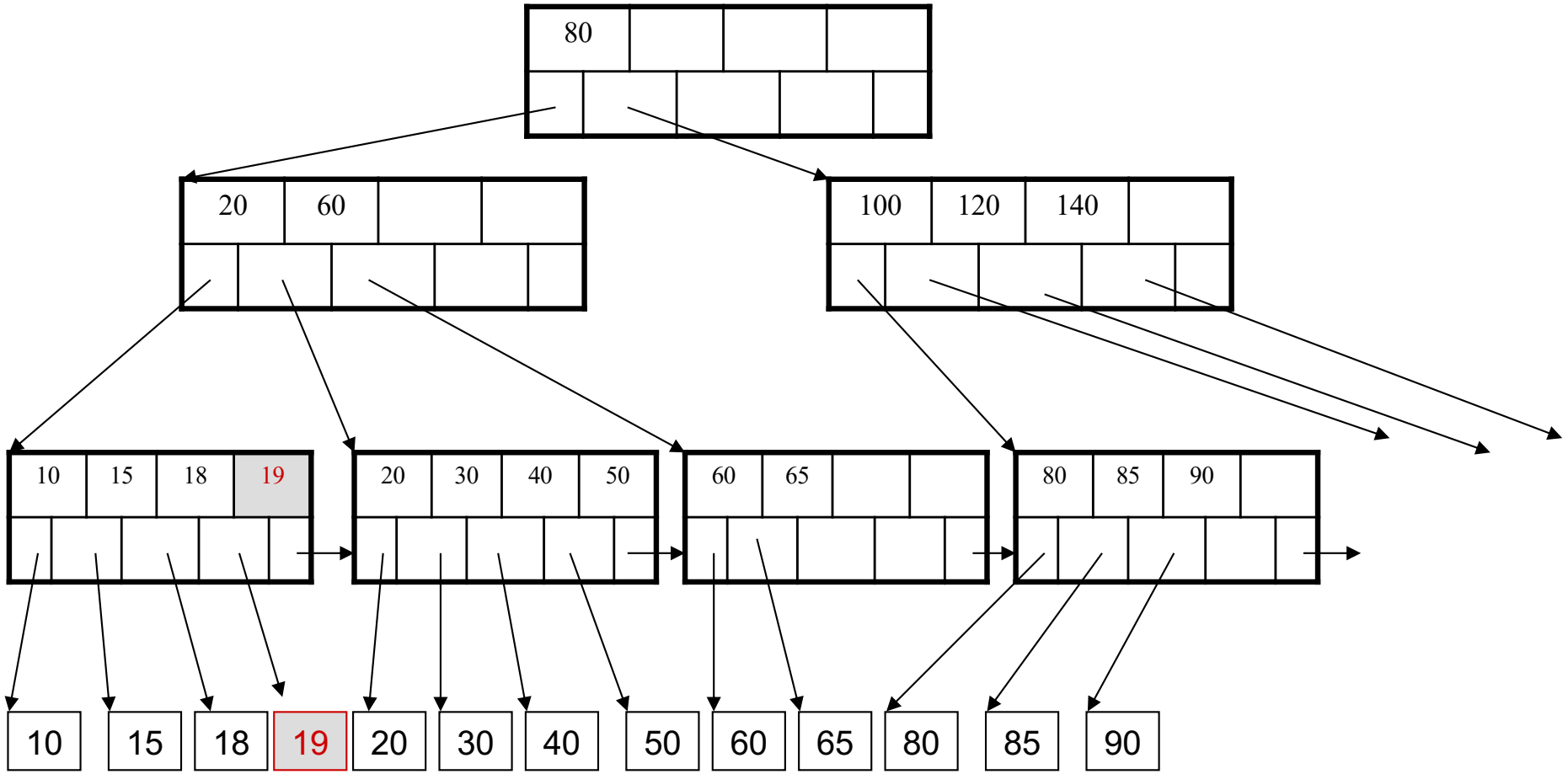


If leaf, also keep K3 in right node
When root splits, new root has 1 key only

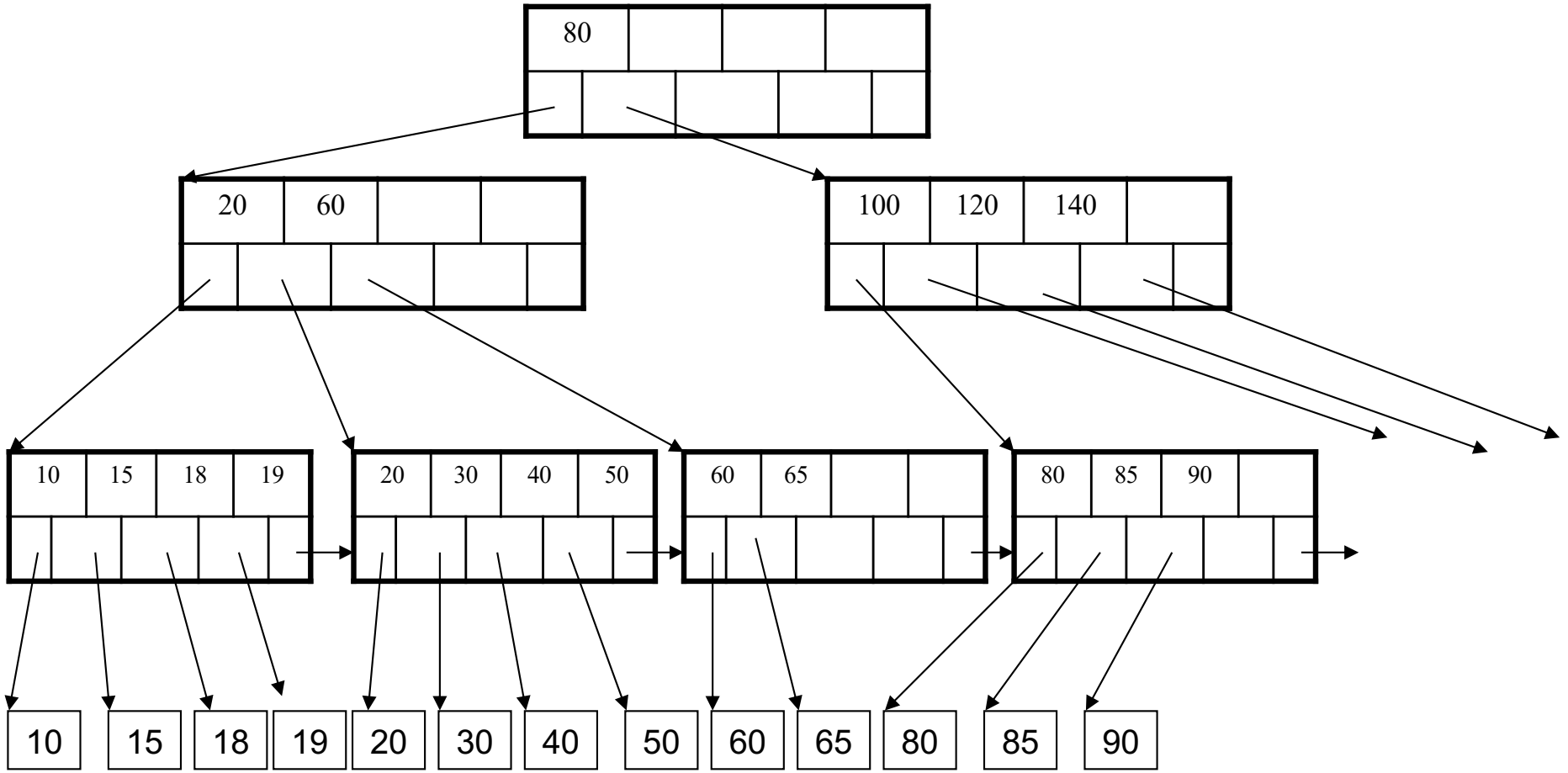
Insert


Insert K=19
80
20
60
100
120
140
10
15
18
20
30
40
50
60
65
80
85
90
10
15
18
20
30
40
50
60
65
80
85
90

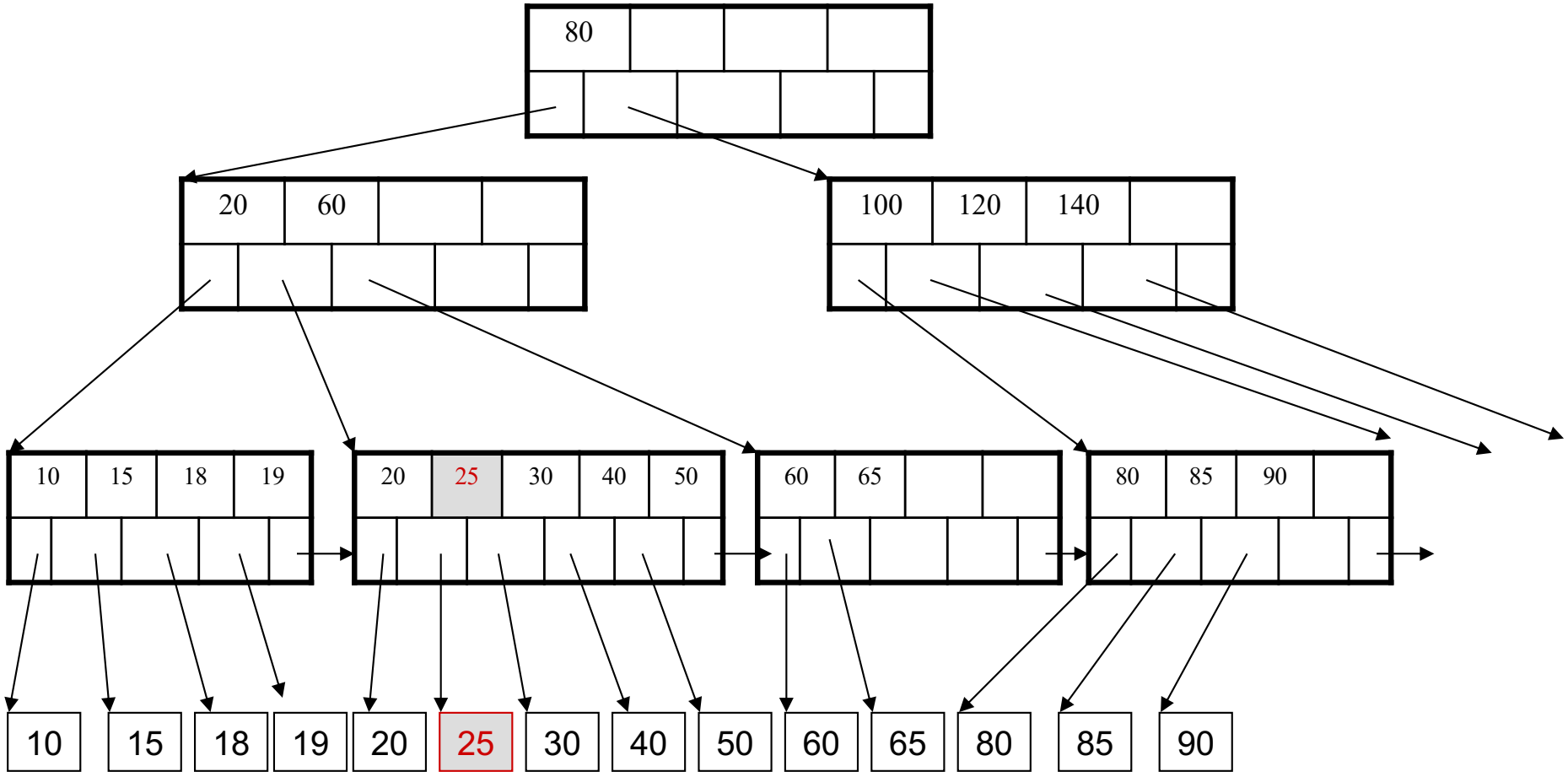
After insertion



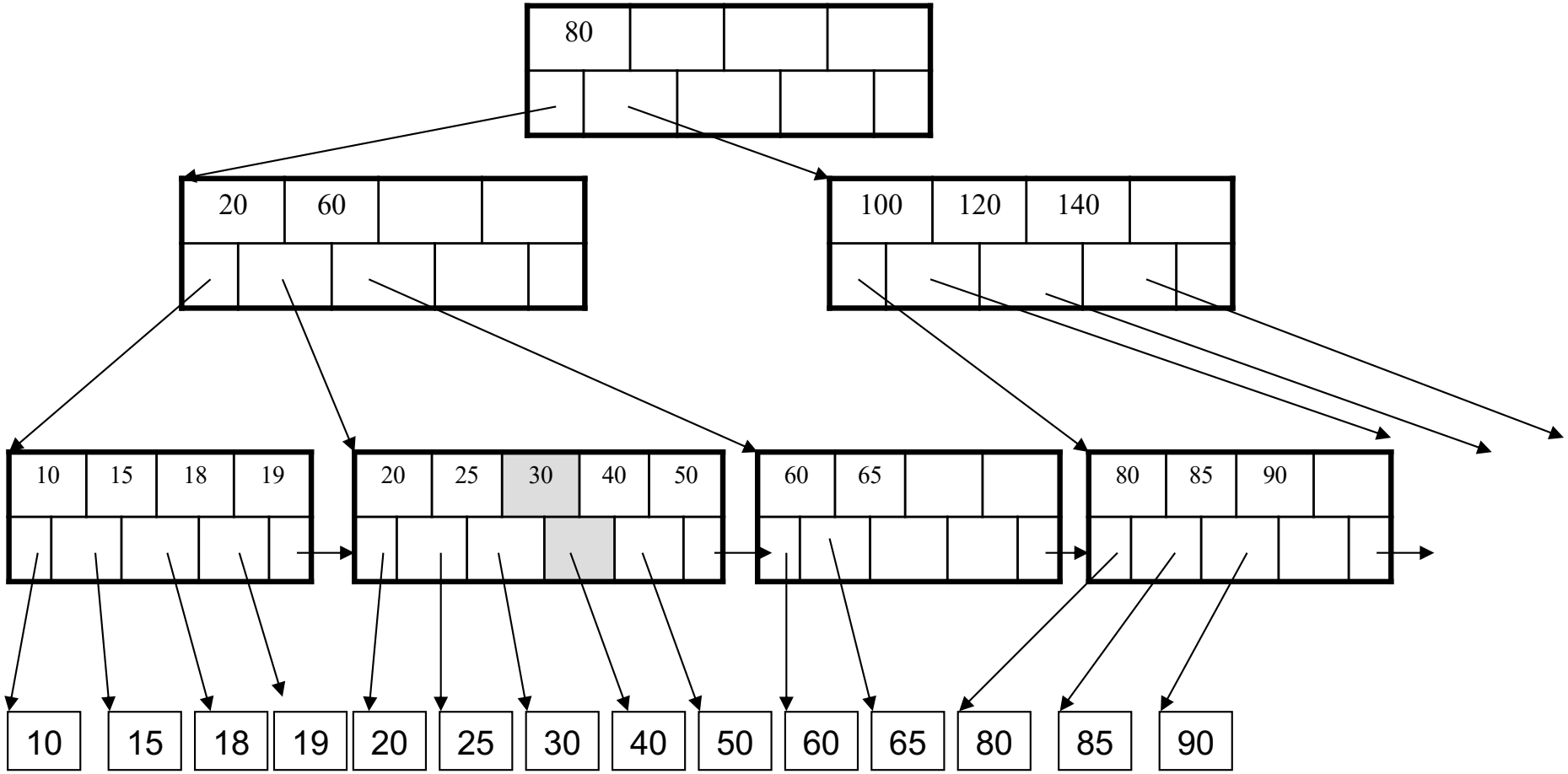
Now insert 25



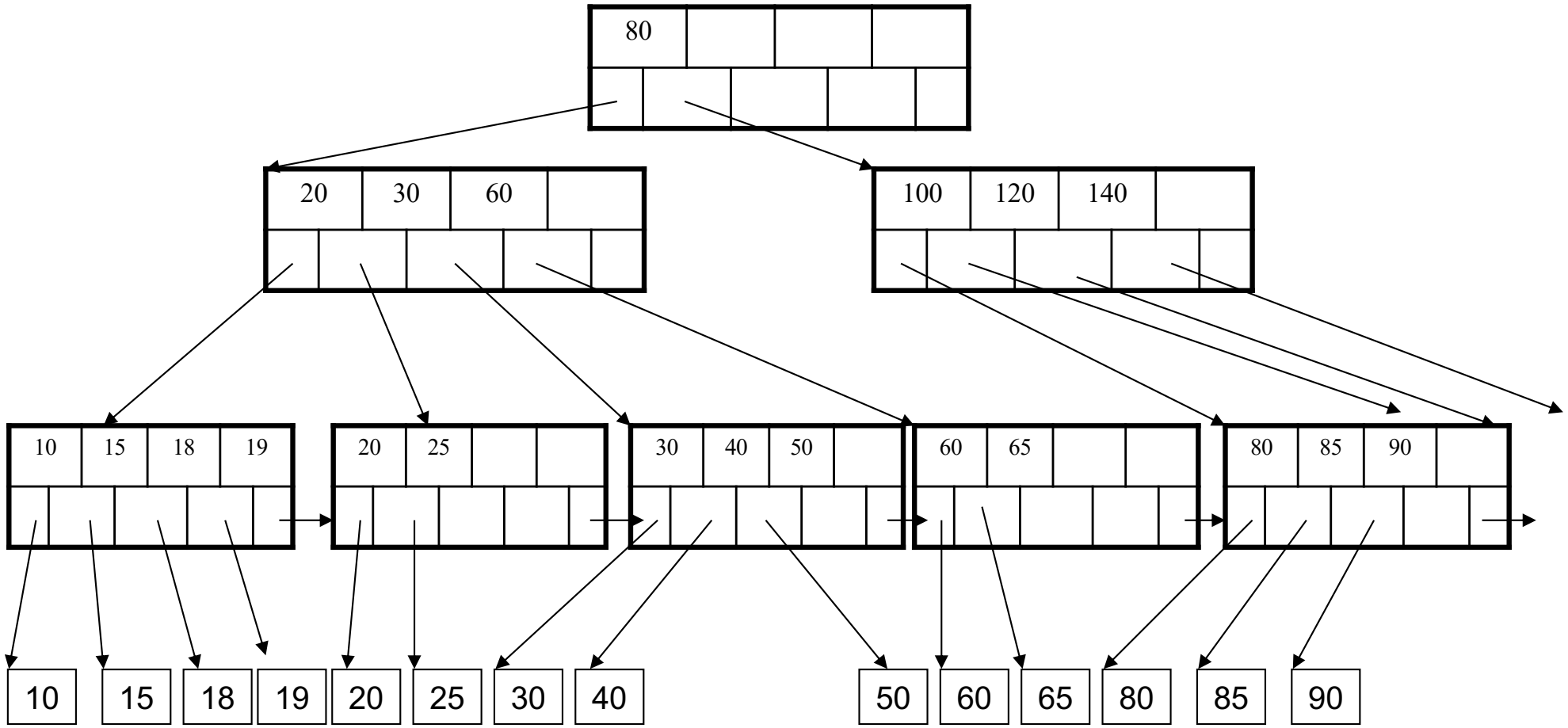
After insertion



But now have to split !



After the split



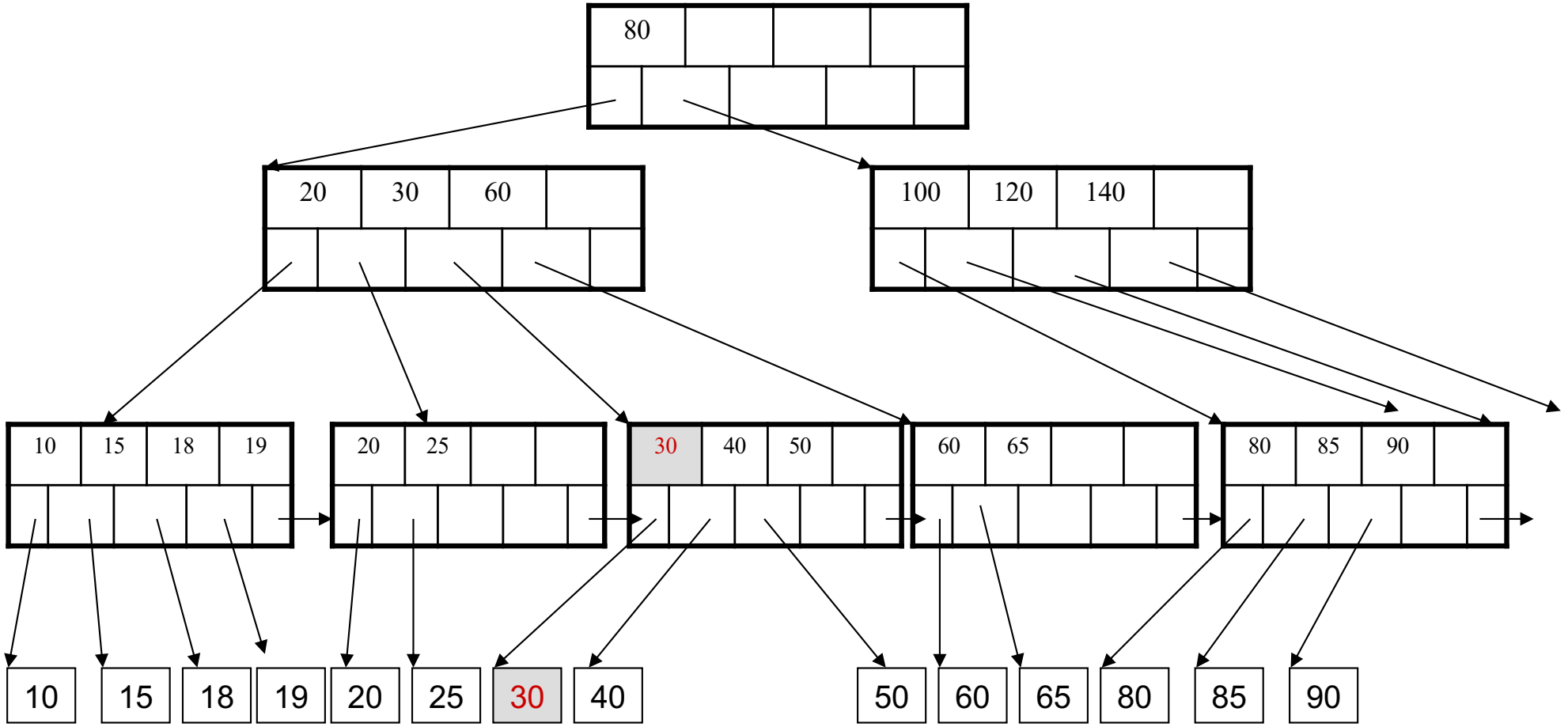
- Note: when a leaf is split, the middle key is copied to the new leaf on **right** (and also inserted in parent)
- Since we assumed the right pointer from key = k points to keys **>=** k

Delete

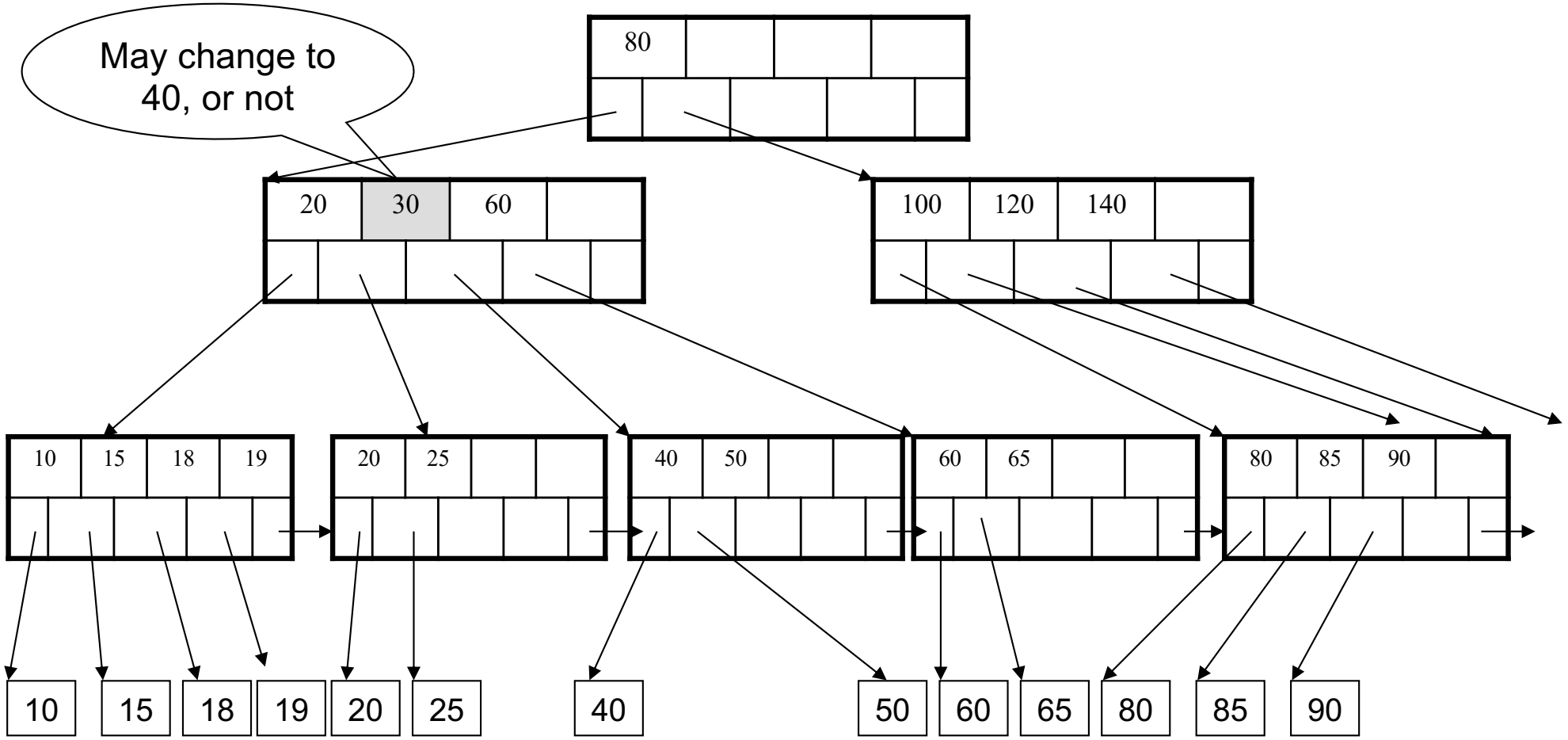
Delete (K, P)

- Find leaf where K belongs, delete
- Check for capacity
- If leaf below capacity, search adjacent nodes (left first, then right) for extra tuples and rotate them to new leaf
- If adjacent nodes at 50% full, merge
- Update and repeat algorithm on parent nodes if necessary

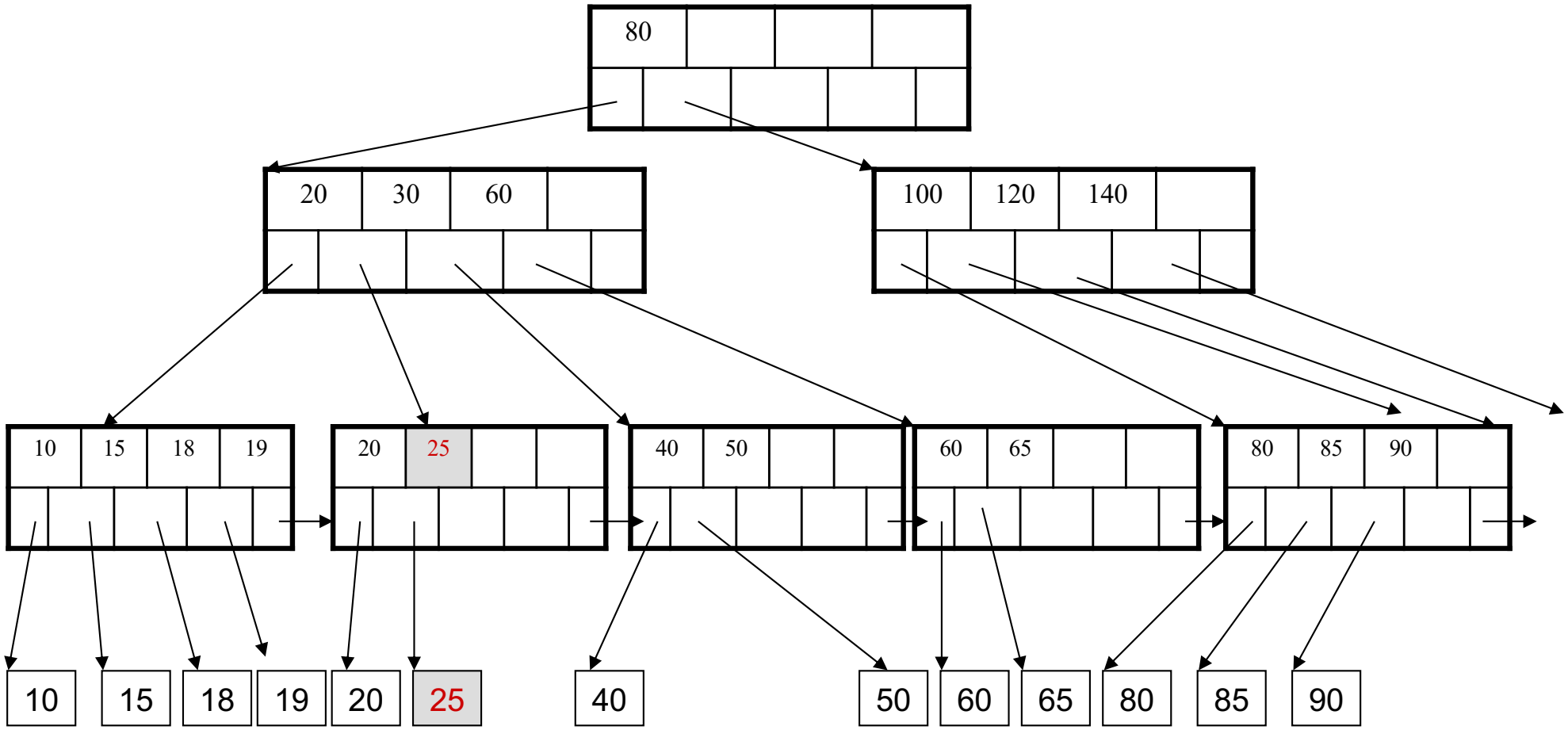
Delete 30


80
20 30 60
100 120 140
10 15 18 19
20 25
30 40 50
60 65
80 85 90
10 15 18 19 20 25 30 40 50 60 65 80 85 90

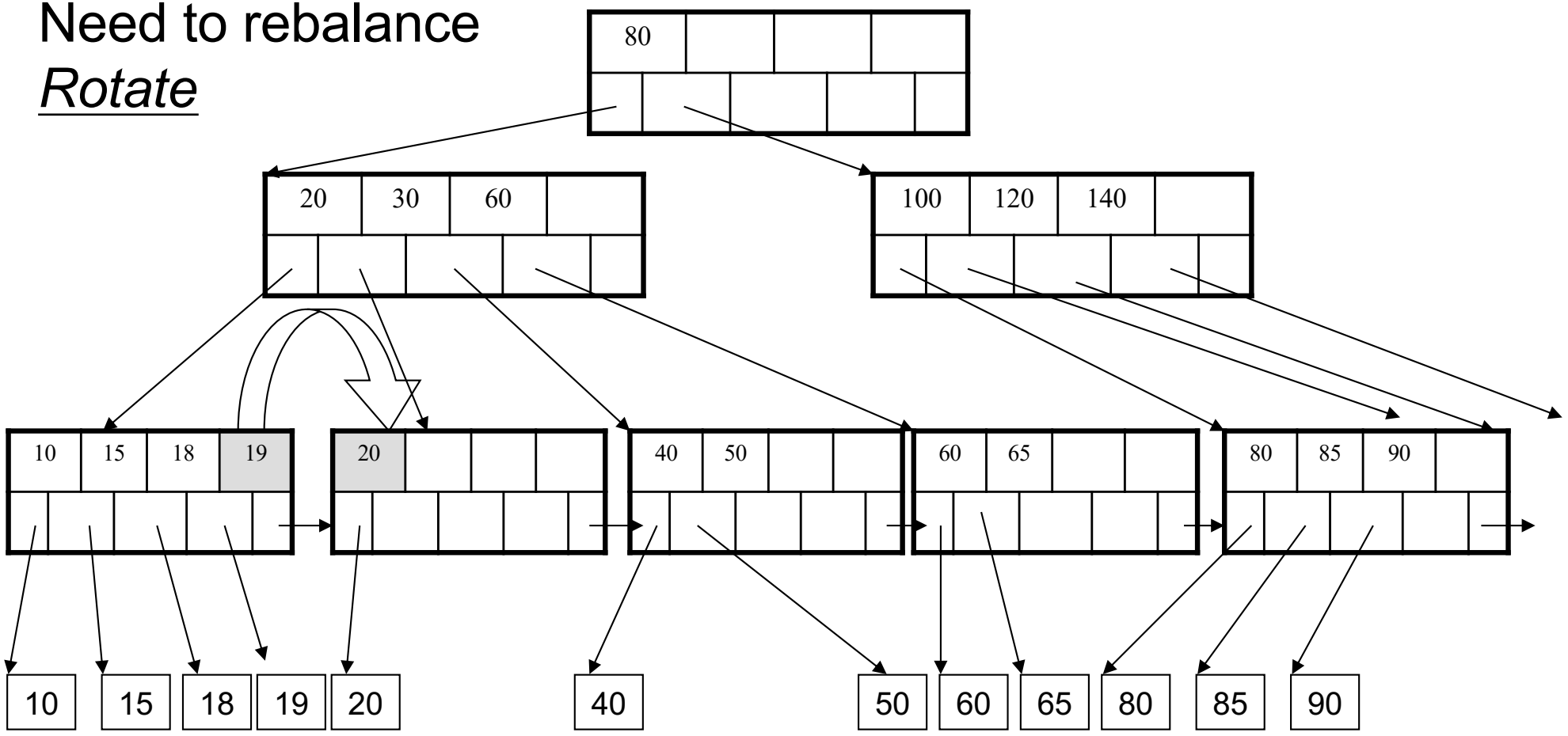
After deleting 30



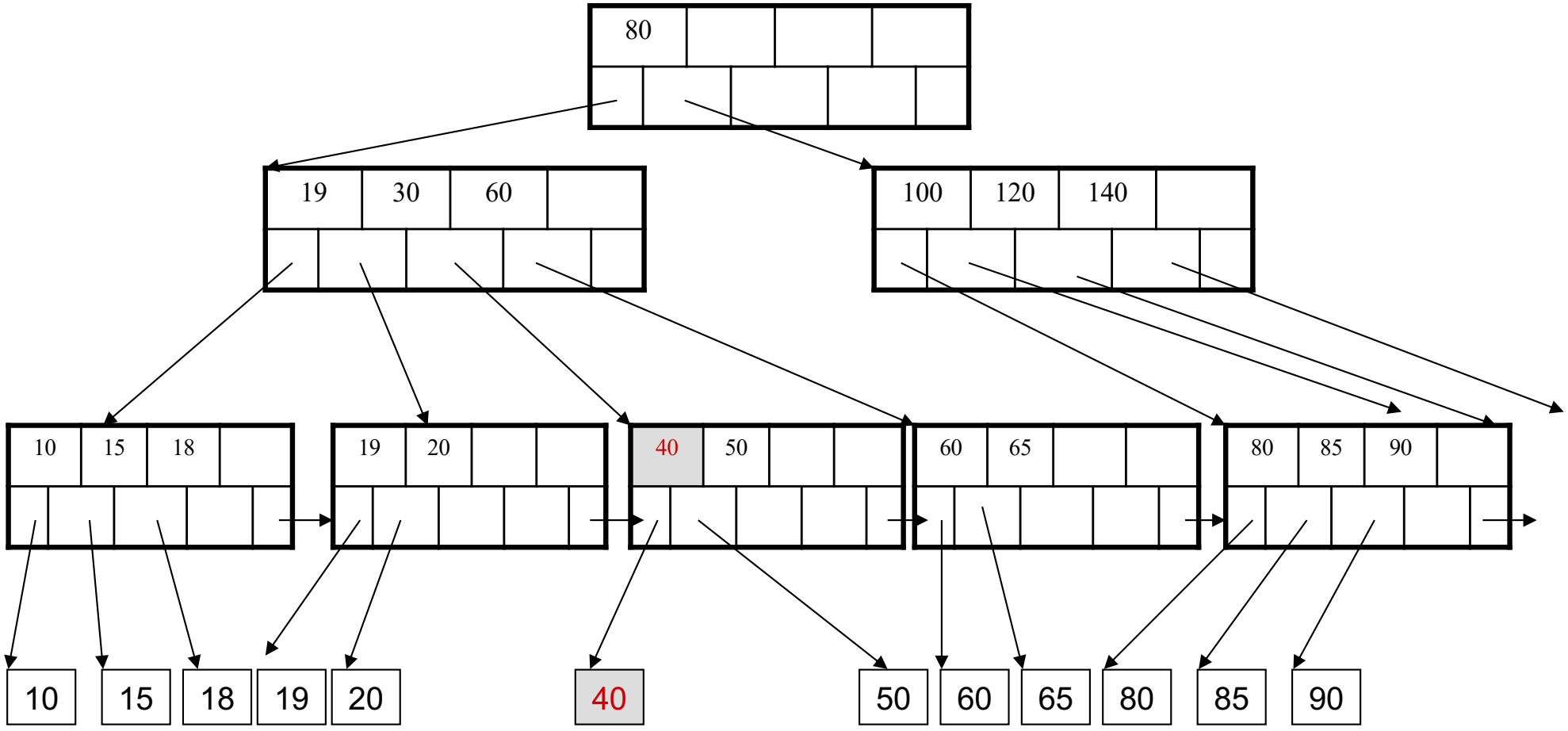
Now delete 25



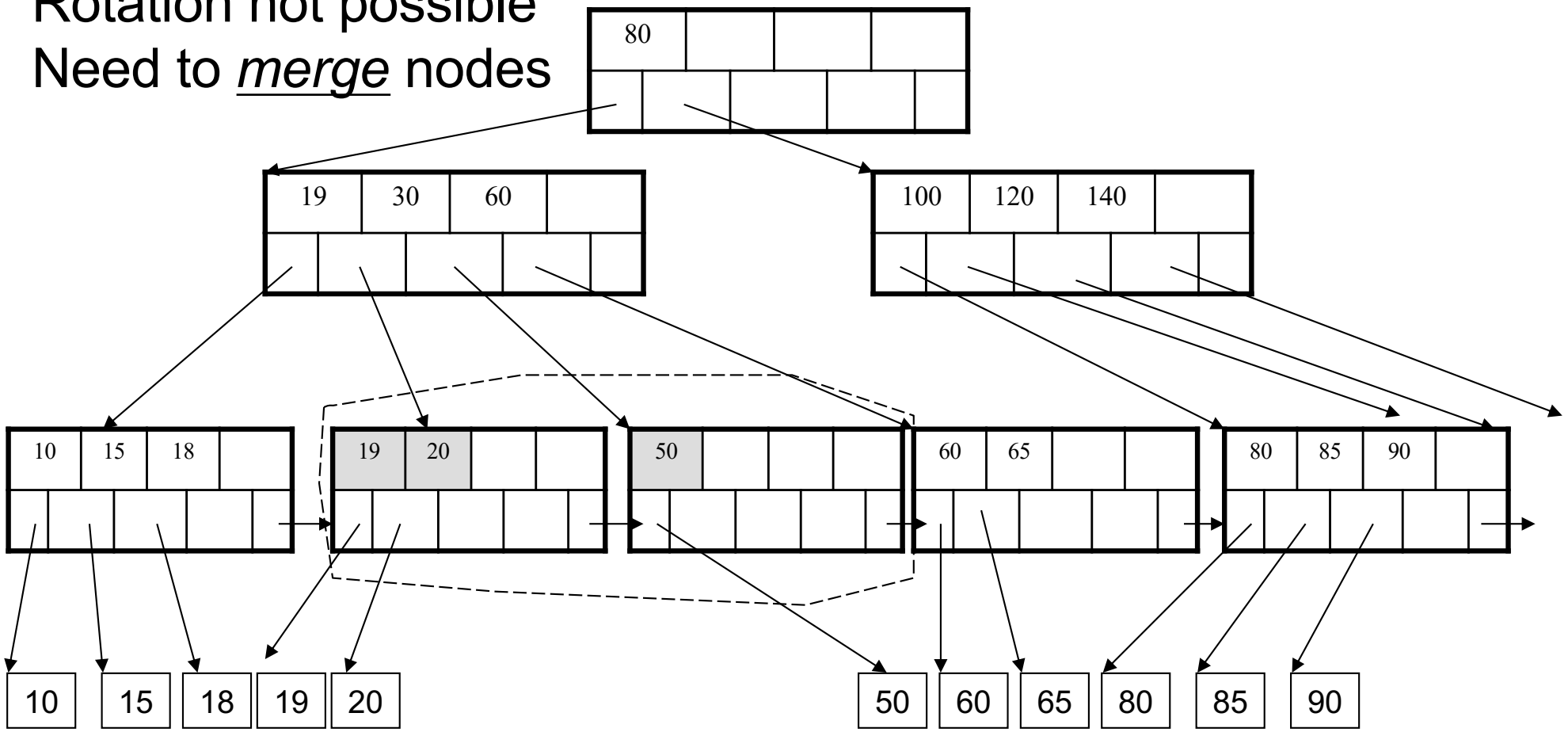
After deleting 25
Need to rebalance
Rotate



Now delete 40

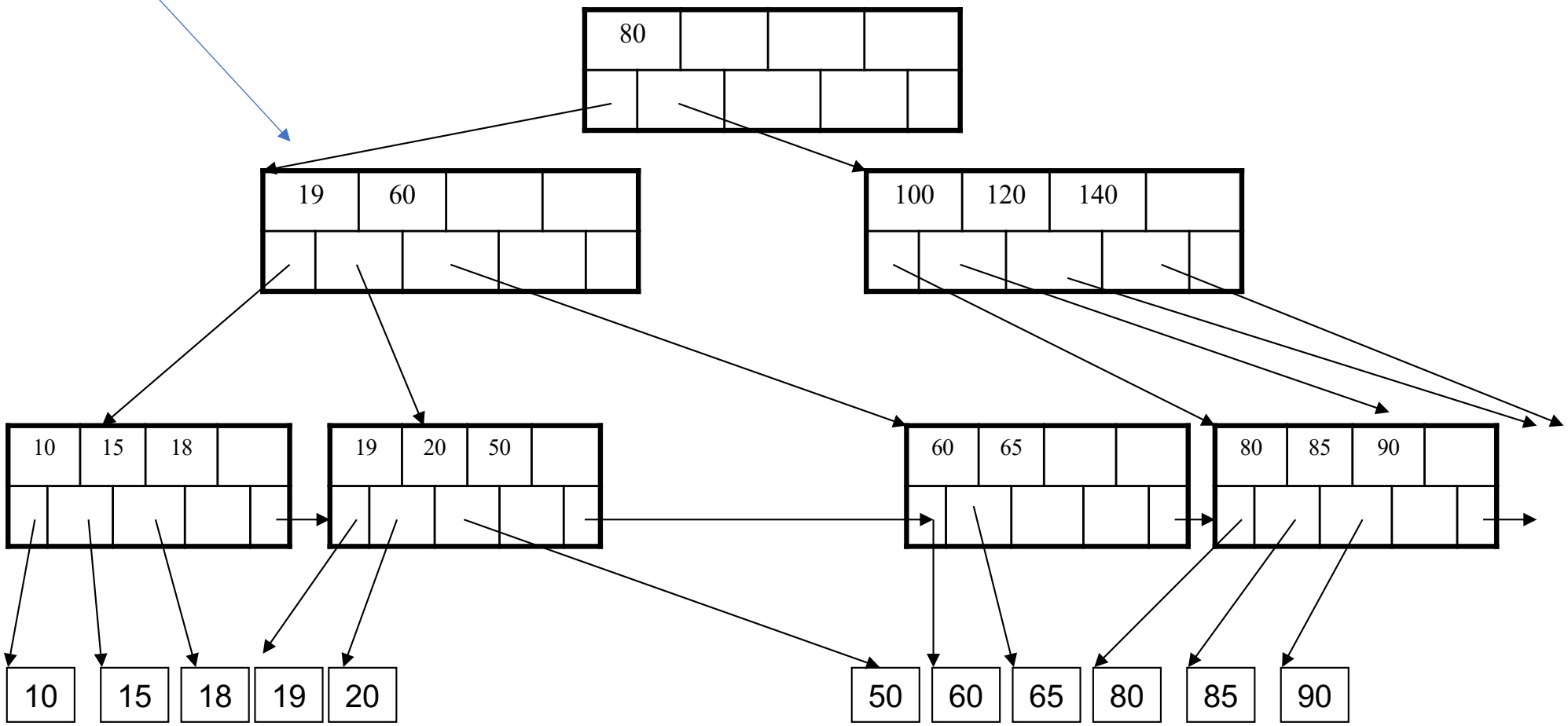


After deleting 40
Rotation not possible
Need to *merge* nodes



Parent must be deleted

Final tree



- Default index structure on most DBMSs
- Very effective at answering ‘point’ queries: sid = 80
- Effective for range queries: 50 < age AND age < 100
- Less effective for multirange: 50<age<100 AND 2018<started<2020

Another example

- Start with an empty B+ tree, d=2
- Insert 17, 3, 25, 95, 8, 57, 69
- Then insert 29, 91, 78, 80, 92, 99, 97

Delete

- Now delete all nodes in the following order:
- 57, 3, 99, 29, 17, 25, 95, 8, 78, 92, 69, 97, 91

Implementation of Physical Operators

- Iterator method
  - A group of three methods that allows a consumer of the results of the physical operator to get one tuple at a time.
  - Methods: open(), getnext(), close().

Union Operator with Iterator interface

```
Open() {
    b := the first block of R;
    t := the first tuple of block b;
}

GetNext() {
    IF (t is past the last tuple on block b) {
        increment b to the next block;
        IF (there is no next block)
            RETURN NotFound;
        ELSE /* b is a new block */
            t := first tuple on block b;
    }  /* now we are ready to return t and increment */
    oldt := t;
    increment t to the next tuple of b;
    RETURN oldt;
}

Close() {
}
```

Cost Parameters

- Cost = total number of I/Os
- This is a simplification that ignores CPU, network
- Parameters:
  - B(R) = # of blocks (i.e., pages) for relation R
  - T(R) = # of tuples in relation R
  - V(R, a) = # of distinct values of attribute a
    - When a is a key, V(R,a) = T(R)
    - When a is not a key, V(R,a) can be anything < T(R)

Cost Convention

- Cost = the cost of reading operands from disk
- Cost of writing the final result to disk is not included; need to count it separately when applicable

- **Assumption:** Arguments to operator are on disk but result is in main memory.
  - If final answer then result is written to disk and the cost of doing so depends on the size of the answer and not how it was computed.

Types of Algorithms

- One-pass algorithms
    - Reading data from disk only once.
    - One argument to fit in memory except select project
- Index-based algorithms
    - Use indexes to reduce the amount of data fetched.
    - Sort-scan: means sorting while scanning. If R is ot be sorted on a and B_tree on a, then scan B_tree
- Two-pass algorithms
    - Data too large to fit in main memory
    - Reading data a first time from disk, processing it is some way, then reading again from disk.
- Note about readings:
    - In class, we discuss only algorithms for joins
    - Other operators are easier: book has extra details

Types of Operators

- Tuple-at-a-time unary operators
  - Do not require the entire relation to be in memory at once.
  - Read one block at a time and produce output.
  - Select, project
- Full-relation, unary operators
  - See all tuples at once.
  - One-pass algorithms must limit to buffer size M.
  - Distinct, Group By, Order By
- Full-relation, binary operators.
  - For one-pass algorithm, one argument must be limited to size M

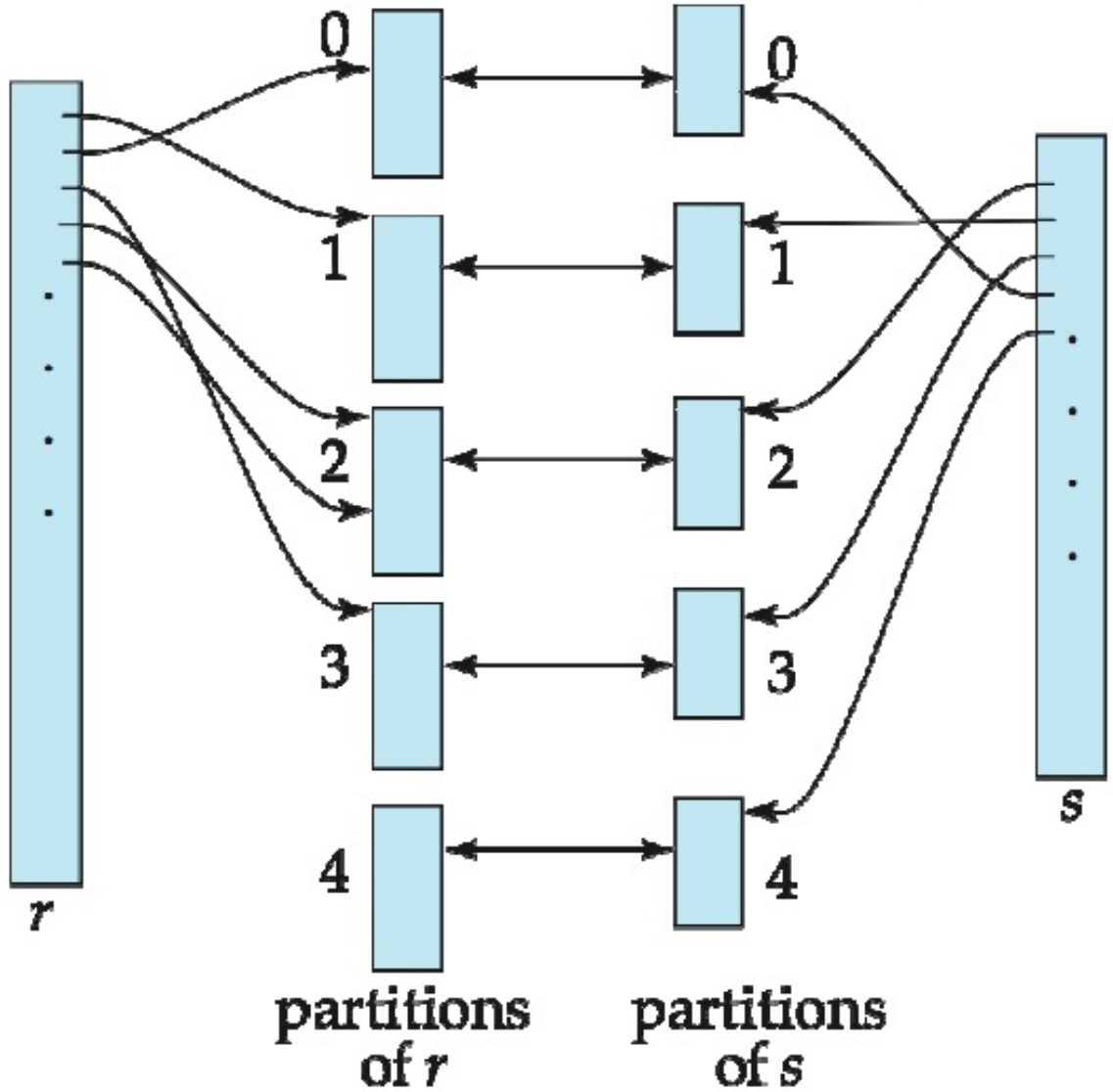
Join Algorithms

- Hash join
- Nested loop join
- Sort-merge join

Hash Join

- Hash join: R ⋈ S
- Scan R, build buckets in main memory; Then scan S, hash with same function, and join
- Cost: B(R) + B(S)

Hash-Join (Cont.)



What if there is not sufficient memory to store both relations?

- One-pass algorithm when $B(R)-1 \leq M$ or approximately $B(R) \leq M$
- In other words, all pages of R must fit into the memory of the join operator.

Example

- Open()
  - Scan R and build buckets
- GetNext()
  - Scan one block of S, join with hash table of R, and output.
  - Till S.next() is not found.
- Close()
  - Close R and S.

Nested Loop Join

- Tuple-based nested loop R ⋈ S
- R is the outer relation, S is the inner relation

```
for each tuple t1 in R do
    for each tuple t2 in S do
        if t1 and t2 join then output (t1,t2)
```

Cost in terms of I/O?

Nested Loop Join

- Tuple-based nested loop R ⋈ S
- R is the outer relation, S is the inner relation

for each tuple t1 in R do

    for each tuple t2 in S do

        if t1 and t2 join then output (t1,t2)

§ Cost: B(R) + T(R) B(S) § Multiple-pass since S is read many times

Block refinement

```
for each block of tuples r in R do
    for each block of tuples s in S do
        for all pairs of tuples t1 in r, t2 in s
            if t1 and t2 join then output (t1,t2)
```

What is the cost?

Block refinement

for each block of tuples r in R do

    for each block of tuples s in S do

        for all pairs of tuples t1 in r, t2 in s

            if t1 and t2 join then output (t1,t2)

Cost: B(R) + B(R)B(S)

Group-Block refinement

- for each group of M-1 blocks r in R do
  - for each block of tuples s in S do
    - for all pairs of tuples t1 in r, t2 in s
      - if t1 and t2 join then output (t1,t2)

What is the cost?

Group-Block refinement

- for each group of M-1 blocks r in R do
  - for each block of tuples s in S do
    - for all pairs of tuples t1 in r, t2 in s
      - if t1 and t2 join then output (t1,t2)

Cost: B(R) + B(R)B(S)/(M-1)

Iterator implementation for tuple-based NLJ

```
Open() {
    R.Open();
    S.Open();
    s := S.GetNext();
}

GetNext() {
    REPEAT {
        r := R.GetNext();
        IF (r = NotFound) { /* R is exhausted for
                the current s */
            R.Close();
            s := S.GetNext();
            IF (s = NotFound) RETURN NotFound;
                /* both R and S are exhausted */
            R.Open();
            r := R.GetNext();
        }
    }
    UNTIL (r and s join);
    RETURN the join of r and s;
}

Close() {
    R.Close();
    S.Close();
}
```

```
FOR each chunk of M-1 blocks of S DO BEGIN
    read these blocks into main-memory buffers;
    organize their tuples into a search structure whose
        search key is the common attributes of R and S;
    FOR each block b of R DO BEGIN
        read b into main memory;
        FOR each tuple t of b DO BEGIN
            find the tuples of S in main memory that
                join with t;
            output the join of t with each of these tuples;
        END;
    END;
END;
```

Index-based selection

- Selection on equality: σ_a=v(R)
- B(R)= size of R in blocks
- T(R) = number of tuples in R
- V(R, a) = # of distinct values of attribute a

Index-based selection

- Selection on equality: σ_a=v(R)
- B(R)= size of R in blocks
- T(R) = number of tuples in R
- V(R, a) = # of distinct values of attribute a

- What is the cost in each case?
  - Clustered index on a:
  - Unclustered index on a

Index-based selection

- Selection on equality: σ_a=V(R)
- B(R)= size of R in blocks
- T(R) = number of tuples in R
- V(R, a) = # of distinct values of attribute a

- What is the cost in each case?
  - Clustered index on a: B(R)/V(R,a)
  - Unclustered index on a: T(R)/V(R,a)
- Note: we ignore I/O cost for index pages

Index-based selection

- Example:
  - B(R) = 2000
  - T(R) = 100,000
  - V(R, a) = 20
- Table scan:
- Index based selection:

Index-based selection

- Example:
  - B(R) = 2000
  - T(R) = 100,000
  - V(R, a) = 20
- Table scan: B(R) = 2,000 I/Os
- Index-based selection:

Index-based selection

- Example:
  - B(R) = 2000
  - T(R) = 100,000
  - V(R, a) = 20
- Table scan: B(R) = 2,000 I/Os
- Index-based selection:

Index-based selection

- Example:
  - B(R) = 2000
  - T(R) = 100000
  - V(R, a) = 20
- Table scan: B(R) = 2000 I/Os
- Index-based selection:
  - If index is clustered: 2000/20 = 100
  - If index is unclustered: 100000/20 = 5000
- Lesson: Don’t build unclustered indexes when V(R,a) is small!

Nested Loop Join

- $R \bowtie S$
- Assume S has an index on the join attribute
- Iterate over R, for each tuple fetch corresponding tuple(s) from S
- Cost:
- If index on S is clustered: B(R) + T(R)B(S)/V(S,a)
- If index on S is unclustered: B(R) + T(R)T(S)/V(S,a)