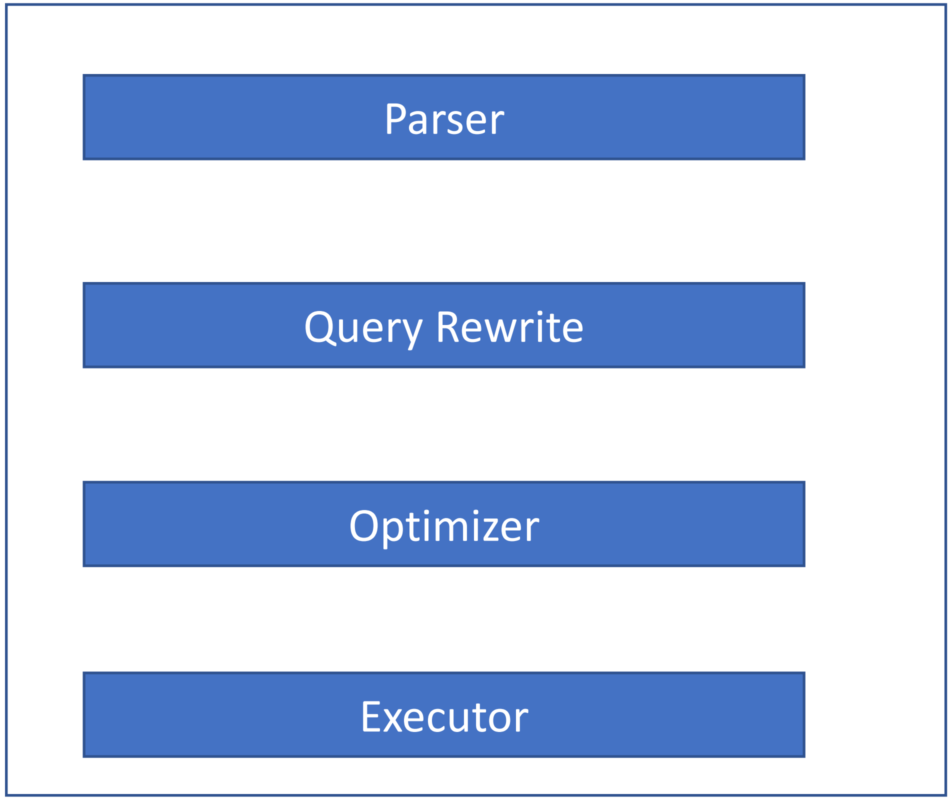
CSC553 Advanced Database Concepts

Tanu Malik

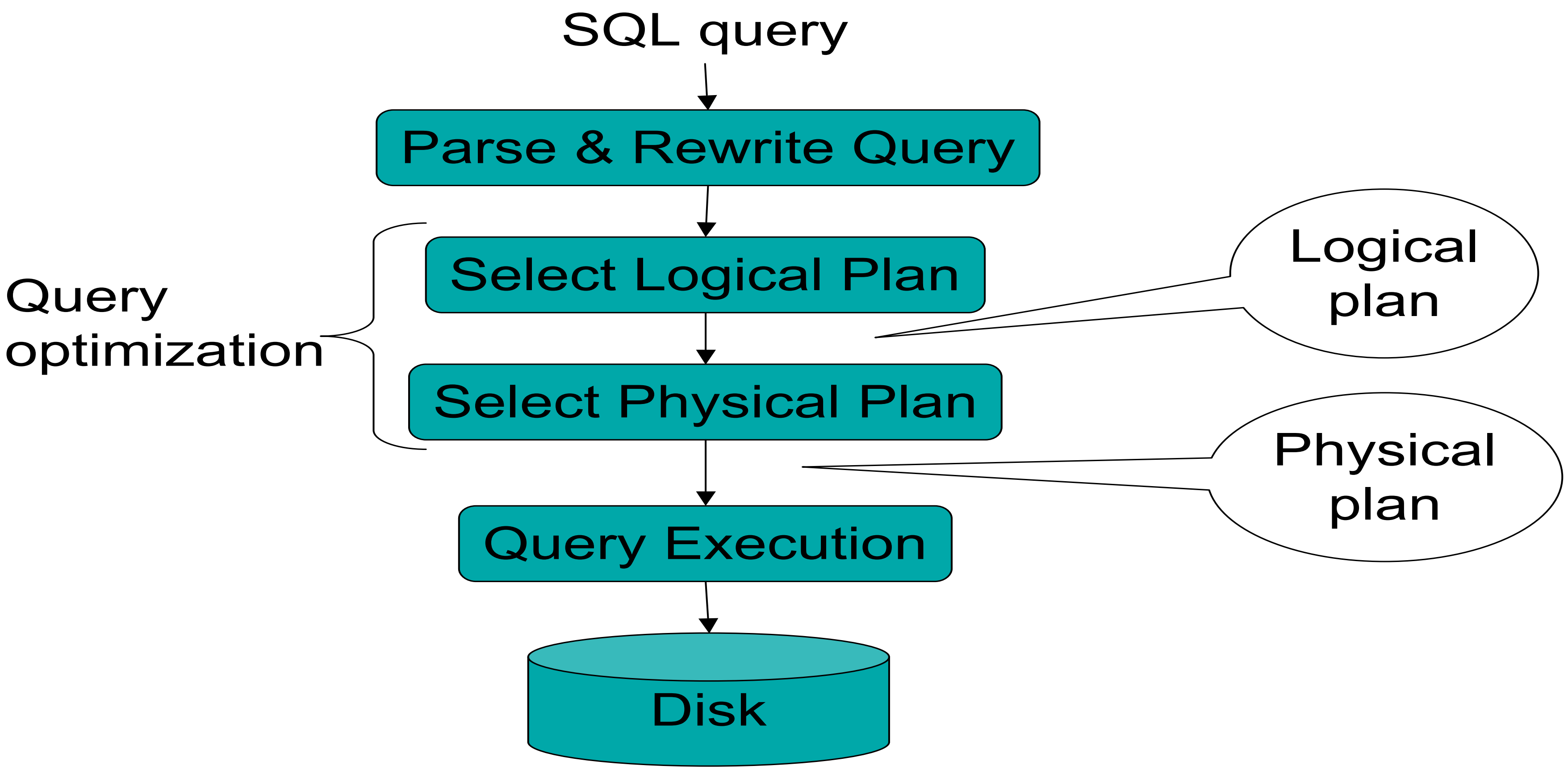
School of Computing

DePaul University

DBMS Architecture: Query Processor



Query Evaluation Steps



Steps in Query Evaluation

- **Step 0: admission control**
  - User connects to the db with username, password
  - User sends query in text format
- **Step 1: Query parsing: Syntax Check**
  - Parses query into an internal format
  - Performs various checks using catalog
- **Step 2: Query rewrite: Simplify queries**
  - View rewriting, flattening, etc.

Continue with Query Evaluation

- **Step 3: Query optimization**
  - Find an efficient query plan for executing the query
- A **query plan** is
  - **Logical query plan**: an *extended relational algebra* tree
  - **Physical query plan**: with additional annotations at each node
    - *Access method* to use for each relation
    - Implementation to use for each *relational operator*
- Next 5 lectures devoted to query processor.

Example Database Schema

- Supplier(sno, sname, scity, sstate)
- Part(pno, pname, psize, pcolor)
- Supply(sno, pno, price)

- View: Find suppliers in Chicago, IL

```
CREATE VIEW NearbySupp As
SELECT sno, sname
FROM Supplier
Where scity = ‘Chicago’ and sstate = ‘IL’
```

Example Query

- Find the names of all suppliers in Chicago who supply Part 2

```
SELECT sname FROM NearbySupp
WHERE sno IN ( SELECT sno
               FROM Supplies
               WHERE pno = 2 )
```

Rewritten Version of Our Query

- Original Query:

```
SELECT sname
FROM NearbySupp
WHERE sno IN ( SELECT sno
               FROM Supplies
               WHERE pno = 2
```

SELECT sno, sname
FROM Supplier
WHERE scity = ‘Chicago’ and sstate = ‘IL’

- Rewritten Query:

SELECT S.sname
FROM Supplier S, Supplies U
Where S.scity = ‘Chicago’ AND S.sstate = ‘IL’
AND s.sno = U.sno AND U.pno = 2

Example in SQLDeveloper

Relational Algebra

- Relational algebra (RA) is a query language for the relational model with a solid theoretical foundation.
- Relational algebra is not visible at the user interface level (not in any commercial RDBMS, at least).
- However, almost any RDBMS uses RA to represent queries internally (for query optimization and execution).
- Knowledge of relational algebra will help in understanding SQL and relational database systems in general.

Classic Relational Operators

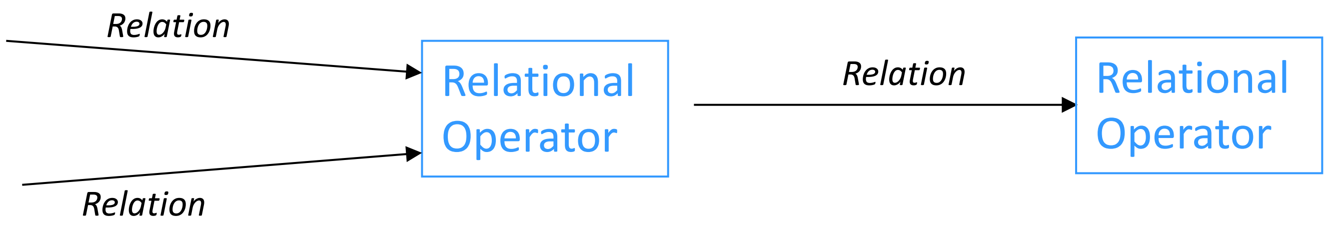
- Basic Operators
  1. select (σ)
  2. project (π)
  3. union ($\cup$)
  4. set difference (–)
  5. cartesian product ($\times$)
  6. rename (ρ)

Extended Relational Operators

- Group By/Aggregation (γ)
- Order By/Sort (τ)
- Distinct/Duplicate Elimination (δ)

Algebra equivalence

- Closure Property



- In mathematics, an algebra is a
  - set (the carrier), and
  - operations that are closed with respect to the set.
  - Example: (N, {$*$, +}) forms an algebra.
- In case of RA,
  - the carrier is the set of all finite relations.

Bank Database Schema

Account

bname	acct_no	balance

Branch

bname	bcity	assets

Depositor

cname	acct_no

Borrower

cname	lno

Customer

cname	cstreet	ccity

Loan

bname	lno	amt

Bank Database

Account

bname	acct_no	balance
Downtown	A-101	500
Mianus	A-215	700
Perry	A-102	400
R.H.	A-305	350
Brighton	A-201	900
Redwood	A-222	700
Brighton	A-217	750

Branch

bname	bcity	assets
Downtown	Brooklyn	9
Redwood	Palo Alto	2.1
Perry	Horseneck	1.7
Mianus	Horseneck	0.4
R.H.	Horseneck	8
Pownel	Bennington	0.3
N. Town	Rye	3.7
Brighton	Brooklyn	7.1

Depositor

cname	acct_no
Johnson	A-101
Smith	A-215
Hayes	A-102
Turner	A-305
Johnson	A-201
Jones	A-217
Lindsay	A-222

Borrower

cname	lno
Jones	L-17
Smith	L-23
Hayes	L-15
Jackson	L-14
Curry	L-93
Smith	L-11
Williams	L-17
Adams	L-16

Customer

cname	cstreet	ccity
Jones	Main	Harrison
Smith	North	Rye
Hayes	Main	Harrison
Curry	North	Rye
Lindsay	Park	Pittsfield
Turner	Putnam	Stanford
Williams	Nassau	Princeton
Adams	Spring	Pittsfield
Johnson	Alma	Palo Alto
Glenn	Sand Hill	Woodside
Brooks	Senator	Brooklyn
Green	Walnut	Stanford

Loan

bname	lno	amt
Downtown	L-17	1000
Redwood	L-23	2000
Perry	L-15	1500
Downtown	L-14	1500
Mianus	L-93	500
R.H.	L-11	900
Perry	L-16	1300

Select (σ)

Notation: $\sigma_{predicate}(Relation)$

$\sigma_{\text{bcity = "Brooklyn"}}$ (branch) =

bname	bcity	assets
Downtown	Brooklyn	9
Brighton	Brooklyn	7.1

$\sigma_{\text{assets > \$8M}}$ ($\sigma_{\text{bcity = "Brooklyn"}}$ (branch)) =

bname	bcity	assets
Downtown	Brooklyn	9

(same as $\sigma_{\text{assets > \$8M AND bcity = "Brooklyn"}}$ (branch))

Project (π)

Notation: $\pi_{A1, \ldots, An}$ (*Relation*)

- Relation: name of a table or result of another query
- Each A_i is an attribute
- Idea: π selects columns (vs. σ which selects rows)

Customer		
cname	**cstreet**	**ccity**

$\pi_{\text{cstreet, ccity}}$ (customer) =

cstreet	ccity
Main	Harrison
North	Rye
Park	Pittsfield
Putnam	Stanford
Nassau	Princeton
Spring	Pittsfield
Alma	Palo Alto
Sand Hill	Woodside
Senator	Brooklyn
Walnut	Stanford

Union (∪)

Notation: $Relation_1 \cup Relation_2$

R ∪ S valid only if:

1. *R, S have same number of columns (arity)*
2. *R, S corresponding columns have same domain (compatibility)*

Example:

$$(\pi_{cname}(\text{depositor})) \cup (\pi_{cname}(\text{borrower})) =$$

cname
Johnson
Smith
Hayes
Turner
Jones
Lindsay
Jackson
Curry
Williams
Adams

Schema:

Depositor	
cname	acct_no

Borrower	
cname	lno

Set Difference (–)

Notation: $Relation_1$ - $Relation_2$

R - S valid only if:

1. *R, S have same number of columns (arity)*
2. *R, S corresponding columns have same domain (compatibility)*

Example:

$$(\pi_{bname}(\sigma_{amount \geq 1000}(loan))) - (\pi_{bname}(\sigma_{balance < 800}(account))) =$$

bname	lno	amount
Downtown	L-17	1000
Redwood	L-23	2000
Perry	L-15	1500
Downtown	L-14	1500
Perry	L-16	1300

–

bname	acct_no	balance
Mianus	A-215	700
Brighton	A-201	900
Redwood	A-222	700
Brighton	A-217	850

=

bname
Downtown
Perry

Cartesian Product (×)

Notation: *Relation*$_1$ × *Relation*$_2$

R × S like cross product for mathematical relations:

- *every tuple of R appended to every tuple of S*

Example:

depositor × borrower =

depositor. cname	acct_no	borrower. cname	lno
Johnson	A-101	Jones	L-17
Johnson	A-101	Smith	L-23
Johnson	A-101	Hayes	L-15
Johnson	A-101	Jackson	L-14
Johnson	A-101	Curry	L-93
Johnson	A-101	Smith	L-11
Johnson	A-101	Williams	L-17
Johnson	A-101	Adams	L-16
Smith	A-215	Jones	L-17
…	…	…	…

How many tuples in the result?

*A: depositor (7) * borrower (8) = 56*

Rename (ρ)

Notation: $\rho_{\ \text{identifier}_0\ (\text{identifier}_1,\ \ldots,\ \text{identifier}_n)}\ (\mathit{Relation})$

Example:

$\rho_{\ \text{res (dcname, acctno, bcname, lno)}}\ (\text{depositor} \times \text{borrower}) =$

dccname	**acctno**	**bcname**	**lno**
Johnson	A-101	Jones	L-17
Johnson	A-101	Smith	L-23
Johnson	A-101	Hayes	L-15
Johnson	A-101	Jackson	L-14
Johnson	A-101	Curry	L-93
Johnson	A-101	Smith	L-11
Johnson	A-101	Williams	L-17
Johnson	A-101	Adams	L-16
Smith	A-215	Jones	L-17
…	…	…	…

Distinct (δ)

Notation: δ (*Relation*)

Example:

δ (π_{bcity}(Branch) =

bname	bcity	assets
Downtown	Brooklyn	9M
Brighton	Brooklyn	7.1M
Kenwood	Queens	40M
Manhattan	NY City	3M

=

bcity
Brooklyn
Queens
NY City

Grouping (γ)

Notation: $\gamma_{A_0\, A_1, \ldots, A_n}$ (*Relation*)

Example:

γ_{city}(Branch) =

bname	bcity	assets
Downtown	Brooklyn	9
Brighton	Brooklyn	7.1
Kenwood	Queens	40
Manhattan	NY City	3

=

bcity	Sum(assets)
Brooklyn	15.1
Queens	40
NY City	3

Sort (τ)

Notation: $\tau_{A_0 A_1, \ldots, A_n}$ (*Relation*)

Group By/Aggregation (γ)
Order By/Sort (τ)
Distinct/Duplicate Elimination (δ)

Example:

$\tau_{(sum(assets))}(\gamma_{city}(\text{Branch})) =$

bcity	Sum(assets)
Brooklyn	15.1
Queens	40
NY City	3

=

bcity	Sum(assets)
Queens	40
Brooklyn	15.1
NY City	3

Example Query1 in RA

- *Determine* ***lno*** *for loans that are for an amount that is larger than the amount of some other loan. (i.e.* ***lno*** *for all non-minimal loans)*

Example Query in RA

- *Determine **lno** for loans that are for an amount that is larger than the amount of some other loan. (i.e. **lno** for all non-minimal loans)*

*SELECT * FROM LOAN L1, LOAN L2*
WHERE L1.amount > L2.amount

*SELECT * FROM LOAN L1 WHERE*
amount > ANY (select amount from Loan L2)

*SELECT * FROM Loan L1 WHERE*
amount > (SELECT min(amount) FROM LOAN)

Can do in steps:

$Temp_1 \leftarrow \ldots$

$Temp_2 \leftarrow \ldots Temp_1 \ldots$

…

Example Query in RA

1. Find the base data we need

$$Temp_1 \leftarrow \pi_{\text{lno,amt}} (\text{loan})$$

lno	amt
L-17	1000
L-23	2000
L-15	1500
L-14	1500
L-93	500
L-11	900
L-16	1300

2. Make a copy of (1)

$$Temp_2 \leftarrow \rho_{\text{Temp2 (lno2,amt2)}} (Temp_1)$$

lno2	amt2
L-17	1000
L-23	2000
L-15	1500
L-14	1500
L-93	500
L-11	900
L-16	1300

Example Query in RA

3. Take the cartesian product of 1 and 2

$$Temp_3 \leftarrow Temp_1 \times Temp_2$$

lno	amt	lno2	amt2
L-17	1000	L-17	1000
L-17	1000	L-23	2000
…	…	…	…
L-17	1000	L-16	1300
L-23	2000	L-17	1000
L-23	2000	L-23	2000
…	…	…	…
L-23	2000	L-16	1300
…	…	…	…

Example Query in RA

4. Select non-minimal loans

$$Temp_4 \leftarrow \sigma_{amt > amt2} (Temp_3)$$

5. Project on lno

$$Result \leftarrow \pi_{lno} (Temp_4)$$

… or, if you prefer…

- $\pi_{lno} (\sigma_{amt > amt2} (\pi_{lno,amt} (loan) \times (\rho_{Temp2\ (lno2,amt2)} (\pi_{lno,amt} (loan)))))$

Example Query in RA

- *Determine **lno** for loans that are for an amount that is larger than the amount of some other loan. (i.e. **lno** for all non-minimal loans)*

*SELECT * FROM LOAN L1, LOAN L2*
WHERE L1.amount > L2.amount

*SELECT * FROM LOAN L1 WHERE*
amount > ANY (select amount from Loan L2)

*SELECT * FROM Loan L1 WHERE*
amount > (SELECT min(amount) FROM LOAN)

Can do in steps:

$Temp_1 \leftarrow \ldots$

$Temp_2 \leftarrow \ldots Temp_1 \ldots$

…

Example Query2 in RA

Branch		
bname	**bcity**	**assets**
Downtown	Brooklyn	9
Redwood	Palo Alto	2.1
Perry	Horseneck	1.7
Mianus	Horseneck	0.4
R.H.	Horseneck	8
Pownel	Bennington	0.3
N. Town	Rye	3.7
Brighton	Brooklyn	7.1

- Find branch name and assets in Brooklyn and Horseneck

Combining Operators to Form RA expressions

- **Relational Algebra (RA) expressions**: A SQL query in term of RA operators.
- A RA expression gives a step-by-step procedure

- Multiple SQL may map to the same RA expression. There can be multiple RA expressions for the same SQL.

- **RA equivalence**: Two expressions that will result in the same answer, but one of the expressions can be more *quickly evaluated*

- **RA Expression Tree:** RA maintained as a tree structure inside the DBMS

Example Query3 in RA

Express the following query in the RA:

Find the names of customers who have both accounts and loans

Depositor	
cname	**acct_no**
Johnson	A-101
Smith	A-215
Hayes	A-102
Turner	A-305
Johnson	A-201
Jones	A-217
Lindsay	A-222

Borrower	
cname	**lno**
Jones	L-17
Smith	L-23
Hayes	L-15
Jackson	L-14
Curry	L-93
Smith	L-11
Williams	L-17
Adams	L-16

Customer		
cname	**cstreet**	**ccity**
Jones	Main	Harrison
Smith	North	Rye
Hayes	Main	Harrison
Curry	North	Rye
Lindsay	Park	Pittsfield
Turner	Putnam	Stanford
Williams	Nassau	Princeton
Adams	Spring	Pittsfield
Johnson	Alma	Palo Alto
Glenn	Sand Hill	Woodside
Brooks	Senator	Brooklyn
Green	Walnut	Stanford

Review

Express the following query in the RA:

Find the names of customers who have both accounts and loans

$T_1 \leftarrow \rho_{T1\ (cname2,\ lno)}\ (borrower)$

$T_2 \leftarrow depositor \times T_1$

$T_3 \leftarrow \sigma_{cname\ =\ cname2}\ (T_2)$

$Result \leftarrow \pi_{\ cname}\ (T_3)$

Above sequence of operators (ρ, ×, σ) very common.

Motivates additional (redundant) RA operators.

Relational Algebra

Redundant Operators

1. Natural Join ($\bowtie$)
2. Generalized Projection (π)
3. Outer Joins (⟗ ⟕ ⟖)
4. Subqueries
5. Nested Correlation

Natural Join

Notation: $Relation_1 \bowtie Relation_2$

Idea: combines ρ, ×, σ

A	B	C	D
1	α	+	10
2	α	-	10
2	α	-	20
3	β	+	10

r

⋈

E	B	D
‘a’	α	10
‘a’	α	20
‘b’	β	10
‘c’	β	10

s

=

A	B	C	D	E
1	α	+	10	‘a’

depositor ⋈ borrower

≡

$$\pi_{cname,acct\_no,lno} (\sigma_{cname=cname2} (depositor \times \rho_{t(cname2,lno)} (borrower)))$$

Generalized Projection

Notation: $\pi_{e_1,\ldots,e_n}(Relation)$

$e_1, \ldots, e_n$ can include arithmetic expressions – not just attributes

Example

credit =

cname	limit	balance
Jones	5000	2000
Turner	3000	2500

Then…

$\pi_{\text{cname, limit - balance}}(\text{credit}) =$

cname	limit-balance
Jones	3000
Turner	500

Outer Joins

Motivation:

loan =

bname	lno	amt
Downtown	L-170	3000
Redwood	L-230	4000
Perry	L-260	1700

borrower =

cname	lno
Jones	L-170
Smith	L-230
Hayes	L-155

=

loan $\bowtie$ borrower =

bname	lno	amt	cname
Downtown	L-170	3000	Jones
Redwood	L-230	4000	Smith

Join result loses…

→ *any record of Perry*

→ *any record of Hayes*

Outer Joins

loan =

bname	lno	amt
Downtown	L-170	3000
Redwood	L-230	4000
Perry	L-260	1700

borrower =

cname	lno
Jones	L-170
Smith	L-230
Hayes	L-155

1. Left Outer Join (⟕)

- *preserves all tuples in <u>left</u> relation*

loan ⟕ borrower =

bname	lno	amt	cname
Downtown	L-170	3000	Jones
Redwood	L-230	4000	Smith
Perry	L-260	1700	$\perp$

$\perp$ = NULL

Outer Joins

loan =

bname	lno	amt
Downtown	L-170	3000
Redwood	L-230	4000
Perry	L-260	1700

borrower =

cname	lno
Jones	L-170
Smith	L-230
Hayes	L-155

1. Left Outer Join (⟕)

- *preserves all tuples in left relation*

loan ⟕ borrower =

bname	lno	amt	cname
Downtown	L-170	3000	Jones
Redwood	L-230	4000	Smith
Perry	L-260	1700	⊥

⊥ = NULL

$$R ⟕ S \equiv (R \bowtie S) \cup ((R - \pi_{A,B}(R \bowtie S)) \times \{(C{:}\text{null})\})$$

Outer Joins

loan =

bname	lno	amt
Downtown	L-170	3000
Redwood	L-230	4000
Perry	L-260	1700

borrower =

cname	lno
Jones	L-170
Smith	L-230
Hayes	L-155

2. Right Outer Join (⟖)

- *preserves all tuples in* *<u>right</u>* *relation*

loan ⟖ borrower =

bname	lno	amt	cname
Downtown	L-170	3000	Jones
Redwood	L-230	4000	Smith
⊥	L-155	⊥	Hayes

⊥ = NULL

$$R ⟖ S \equiv (R \bowtie S) \cup ((S - \pi_{A,B}(R \bowtie S)) \times \{(C{:}\text{null})\})$$

Outer Joins

loan =

bname	lno	amt
Downtown	L-170	3000
Redwood	L-230	4000
Perry	L-260	1700

borrower =

cname	lno
Jones	L-170
Smith	L-230
Hayes	L-155

3. Full Outer Join (⟗)

- *preserves all tuples in* ***both*** *relations*

loan ⟗ borrower =

bname	lno	amt	cname
Downtown	L-170	3000	Jones
Redwood	L-230	4000	Smith
Perry	L-260	1700	⊥
⊥	L-155	⊥	Hayes

⊥ = NULL

Subqueries (IN, NOT IN, ALL, ANY, EXISTS)

- Find all customers who have loans greater than 1M.

Borrower

cname	lno
Jones	L-17
Smith	L-23
Hayes	L-15
Jackson	L-14
Curry	L-93
Smith	L-11
Williams	L-17
Adams	L-16

Customer

cname	cstreet	ccity
Jones	Main	Harrison
Smith	North	Rye
Hayes	Main	Harrison
Curry	North	Rye
Lindsay	Park	Pittsfield
Turner	Putnam	Stanford
Williams	Nassau	Princeton
Adams	Spring	Pittsfield
Johnson	Alma	Palo Alto
Glenn	Sand Hill	Woodside
Brooks	Senator	Brooklyn
Green	Walnut	Stanford

Loan

bname	lno	amt
Downtown	L-17	1000
Redwood	L-23	2000
Perry	L-15	1500
Downtown	L-14	1500
Mianus	L-93	500
R.H.	L-11	900
Perry	L-16	1300

Subqueries (IN, NOT IN, ALL, ANY, EXISTS)

- Find all customers who have loan no between 50 and 100.

Select cname from Customer

Where Cname In

(Select cname from Borrower

Where lno LIKE L-[5-9][0-9])

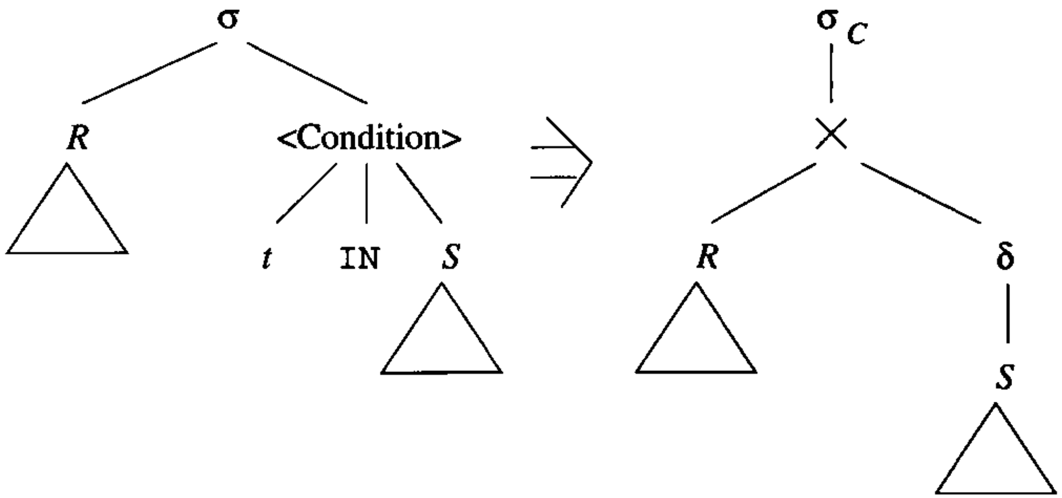
Customer

cname	cstreet	ccity
Jones	Main	Harrison
Smith	North	Rye
Hayes	Main	Harrison
Curry	North	Rye
Lindsay	Park	Pittsfield
Turner	Putnam	Stanford
Williams	Nassau	Princeton
Adams	Spring	Pittsfield
Johnson	Alma	Palo Alto
Glenn	Sand Hill	Woodside
Brooks	Senator	Brooklyn
Green	Walnut	Stanford

Borrower

cname	lno
Jones	L-17
Smith	L-23
Hayes	L-15
Jackson	L-14
Curry	L-93
Smith	L-11
Williams	L-17
Adams	L-16

Subqueries (IN, NOT IN, ALL, ANY, EXISTS)



- Two argument selection operator (σ)
- Duplicate elimination (δ) is necessary since the original query assumes set comparison between t and elements of S.
- Selection (σ) is replaced by (σ_C) where C is the join condition and any other condition.

EXISTS

- Select customers who have loans in all the branches.

Borrower

cname	lno
Jones	L-17
Smith	L-23
Hayes	L-15
Jackson	L-14
Curry	L-93
Smith	L-11
Williams	L-17
Adams	L-16

Branch

bname	bcity	assets
Downtown	Brooklyn	9
Redwood	Palo Alto	2.1
Perry	Horseneck	1.7
Mianus	Horseneck	0.4
R.H.	Horseneck	8
Pownel	Bennington	0.3
N. Town	Rye	3.7
Brighton	Brooklyn	7.1

Customer

cname	cstreet	ccity
Jones	Main	Harrison
Smith	North	Rye
Hayes	Main	Harrison
Curry	North	Rye
Lindsay	Park	Pittsfield
Turner	Putnam	Stanford
Williams	Nassau	Princeton
Adams	Spring	Pittsfield
Johnson	Alma	Palo Alto
Glenn	Sand Hill	Woodside
Brooks	Senator	Brooklyn
Green	Walnut	Stanford

Loan

bname	lno	amt
Downtown	L-17	1000
Redwood	L-23	2000
Perry	L-15	1500
Downtown	L-14	1500
Mianus	L-93	500
R.H.	L-11	900
Perry	L-16	1300

EXISTS

- Select customers who have loans in all the branches.

```
SELECT cname FROM Customer C
WHERE
NOT EXISTS (
SELECT DISTINCT bname FROM Branch
MINUS
(SELECT bname FROM Loan L, Borrower B
WHERE C.cname = B.cname AND
B.lno = L.lno))
```

Branch

bname	bcity	assets
Downtown	Brooklyn	9
Redwood	Palo Alto	2.1
Perry	Horseneck	1.7
Mianus	Horseneck	0.4
R.H.	Horseneck	8
Pownel	Bennington	0.3
N. Town	Rye	3.7
Brighton	Brooklyn	7.1

Borrower

cname	lno
Jones	L-17
Smith	L-23
Hayes	L-15
Jackson	L-14
Curry	L-93
Smith	L-11
Williams	L-17
Adams	L-16

Customer

cname	cstreet	ccity
Jones	Main	Harrison
Smith	North	Rye
Hayes	Main	Harrison
Curry	North	Rye
Lindsay	Park	Pittsfield
Turner	Putnam	Stanford
Williams	Nassau	Princeton
Adams	Spring	Pittsfield
Johnson	Alma	Palo Alto
Glenn	Sand Hill	Woodside
Brooks	Senator	Brooklyn
Green	Walnut	Stanford

Loan

bname	lno	amt
Downtown	L-17	1000
Redwood	L-23	2000
Perry	L-15	1500
Downtown	L-14	1500
Mianus	L-93	500
R.H.	L-11	900
Perry	L-16	1300

Without EXISTS-Attempt 1

- Select customers who have loans in all the branches.

SELECT cname FROM Borrower B, Loan L
WHERE B.lno = L.lno
AND L.bname IN (SELECT distinct bname from Branch)
GROUP BY cname
HAVING count(lno) =
(SELECT count(bname) from
Branch)

Branch

bname	bcity	assets
Downtown	Brooklyn	9
Redwood	Palo Alto	2.1
Perry	Horseneck	1.7
Mianus	Horseneck	0.4
R.H.	Horseneck	8
Pownel	Bennington	0.3
N. Town	Rye	3.7
Brighton	Brooklyn	7.1

Borrower

cname	lno
Jones	L-17
Smith	L-23
Hayes	L-15
Jackson	L-14
Curry	L-93
Smith	L-11
Williams	L-17
Adams	L-16

Customer

cname	cstreet	ccity
Jones	Main	Harrison
Smith	North	Rye
Hayes	Main	Harrison
Curry	North	Rye
Lindsay	Park	Pittsfield
Turner	Putnam	Stanford
Williams	Nassau	Princeton
Adams	Spring	Pittsfield
Johnson	Alma	Palo Alto
Glenn	Sand Hill	Woodside
Brooks	Senator	Brooklyn
Green	Walnut	Stanford

Loan

bname	lno	amt
Downtown	L-17	1000
Redwood	L-23	2000
Perry	L-15	1500
Downtown	L-14	1500
Mianus	L-93	500
R.H.	L-11	900
Perry	L-16	1300

Without EXISTS-Attempt 2

- Select customers who have loans in all the branches.

```
SELECT Cname FROM Customer
MINUS
Select Cname FROM
(SELECT cname, bname FROM Borrower, Loan
MINUS
(SELECT cname,bname FROM Borrower, Loan
WHERE B.lno=L.lno))
```

Branch		
bname	**bcity**	**assets**
Downtown	Brooklyn	9
Redwood	Palo Alto	2.1
Perry	Horseneck	1.7
Mianus	Horseneck	0.4
R.H.	Horseneck	8
Pownel	Bennington	0.3
N. Town	Rye	3.7
Brighton	Brooklyn	7.1

Borrower	
cname	**lno**
Jones	L-17
Smith	L-23
Hayes	L-15
Jackson	L-14
Curry	L-93
Smith	L-11
Williams	L-17
Adams	L-16

Customer		
cname	**cstreet**	**ccity**
Jones	Main	Harrison
Smith	North	Rye
Hayes	Main	Harrison
Curry	North	Rye
Lindsay	Park	Pittsfield
Turner	Putnam	Stanford
Williams	Nassau	Princeton
Adams	Spring	Pittsfield
Johnson	Alma	Palo Alto
Glenn	Sand Hill	Woodside
Brooks	Senator	Brooklyn
Green	Walnut	Stanford

Loan		
bname	**lno**	**amt**
Downtown	L-17	1000
Redwood	L-23	2000
Perry	L-15	1500
Downtown	L-14	1500
Mianus	L-93	500
R.H.	L-11	900
Perry	L-16	1300

Continue with Query Evaluation

- **Step 3: Query optimization (**finding cheaper, equivalent expressions**)**
  - Find an efficient query plan for executing the query
- A **query plan** is
  - **Logical query plan**: an *extended relational algebra* tree
  - **Physical query plan**: with additional annotations at each node
    - *Access method* to use for each relation
    - Implementation to use for each *relational operator*
- Next 5 lectures devoted to query processor.

Logical Query Plan

Find loans in branches which have assets greater than 1M

SELECT lno FROM Loan, Branch

WHERE B.bname = L.bname AND assets > 1.0

Branch

bname	bcity	assets
Downtown	Brooklyn	9
Redwood	Palo Alto	2.1
Perry	Horseneck	1.7
Mianus	Horseneck	0.4
R.H.	Horseneck	8
Pownel	Bennington	0.3
N. Town	Rye	3.7
Brighton	Brooklyn	7.1

Borrower

cname	lno
Jones	L-17
Smith	L-23
Hayes	L-15
Jackson	L-14
Curry	L-93
Smith	L-11
Williams	L-17
Adams	L-16

Customer

cname	cstreet	ccity
Jones	Main	Harrison
Smith	North	Rye
Hayes	Main	Harrison
Curry	North	Rye
Lindsay	Park	Pittsfield
Turner	Putnam	Stanford
Williams	Nassau	Princeton
Adams	Spring	Pittsfield
Johnson	Alma	Palo Alto
Glenn	Sand Hill	Woodside
Brooks	Senator	Brooklyn
Green	Walnut	Stanford

Loan

bname	lno	amt
Downtown	L-17	1000
Redwood	L-23	2000
Perry	L-15	1500
Downtown	L-14	1500
Mianus	L-93	500
R.H.	L-11	900
Perry	L-16	1300

Logical Query Plan

Find customers who live in Palo Alto and have loans in branches with assets greater than 1M

SELECT lno FROM Loan L, Branch R, Borrower B, Customer C

WHERE R.bname = L.bname

AND ccity = 'Palo Alto'

AND assets > 1.0

AND B.cname = C.cname

AND B.lno = L.lno

Branch

bname	**bcity**	**assets**
Downtown	Brooklyn	9
Redwood	Palo Alto	2.1
Perry	Horseneck	1.7
Mianus	Horseneck	0.4
R.H.	Horseneck	8
Pownel	Bennington	0.3
N. Town	Rye	3.7
Brighton	Brooklyn	7.1

Borrower

cname	**lno**
Jones	L-17
Smith	L-23
Hayes	L-15
Jackson	L-14
Curry	L-93
Smith	L-11
Williams	L-17
Adams	L-16

Customer

cname	**cstreet**	**ccity**
Jones	Main	Harrison
Smith	North	Rye
Hayes	Main	Harrison
Curry	North	Rye
Lindsay	Park	Pittsfield
Turner	Putnam	Stanford
Williams	Nassau	Princeton
Adams	Spring	Pittsfield
Johnson	Alma	Palo Alto
Glenn	Sand Hill	Woodside
Brooks	Senator	Brooklyn
Green	Walnut	Stanford

Loan

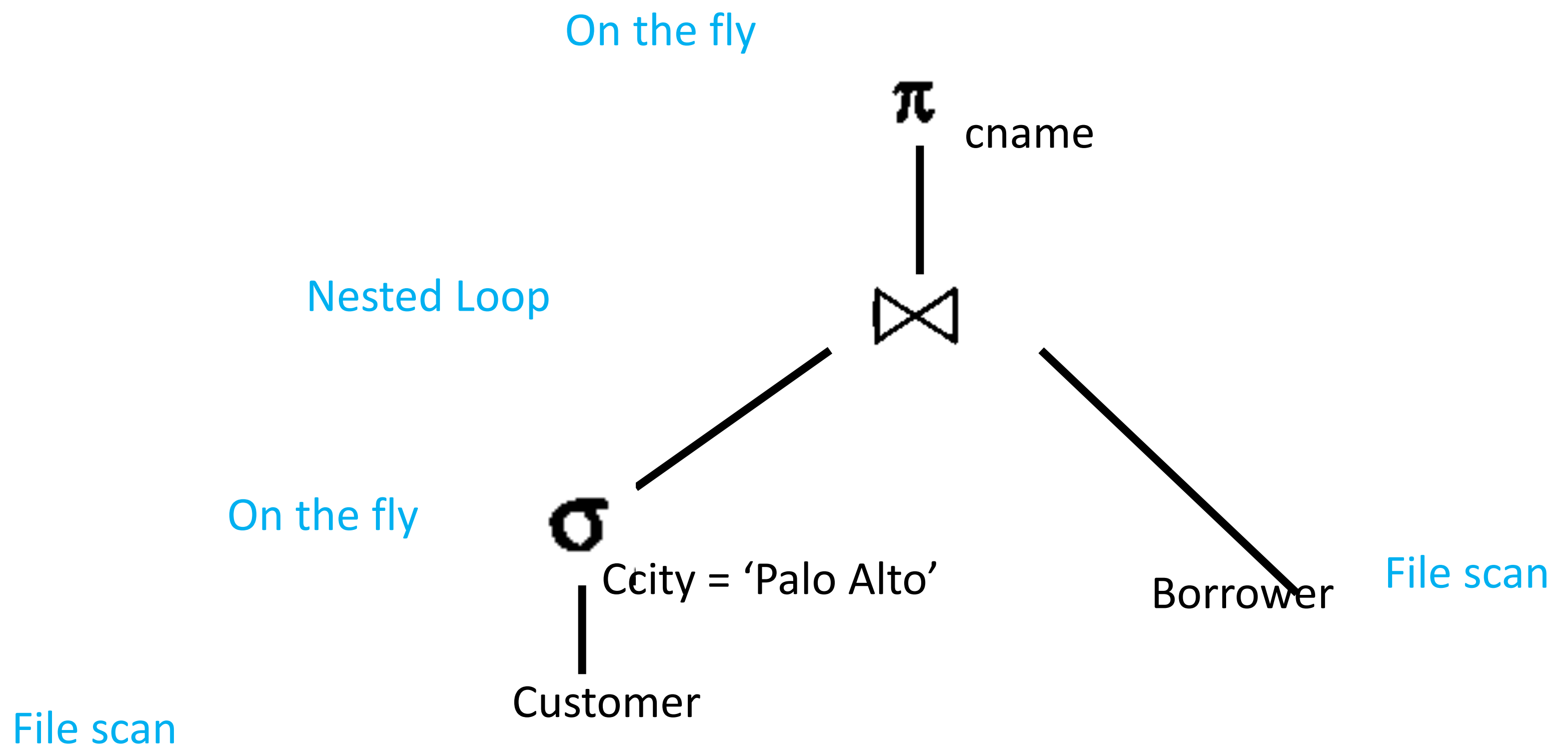
bname	**lno**	**amt**
Downtown	L-17	1000
Redwood	L-23	2000
Perry	L-15	1500
Downtown	L-14	1500
Mianus	L-93	500
R.H.	L-11	900
Perry	L-16	1300

Physical Query Plan

- Logical query plan with extra annotations
- **Implementation choice** for each operator
- **Access path selection** for each relation
  - Bottom of tree = read from disk
  - Use a file scan or use an index

Pipelining

- Physical plan aims to support means the tuples are processed one-by-one as they pass through the operator

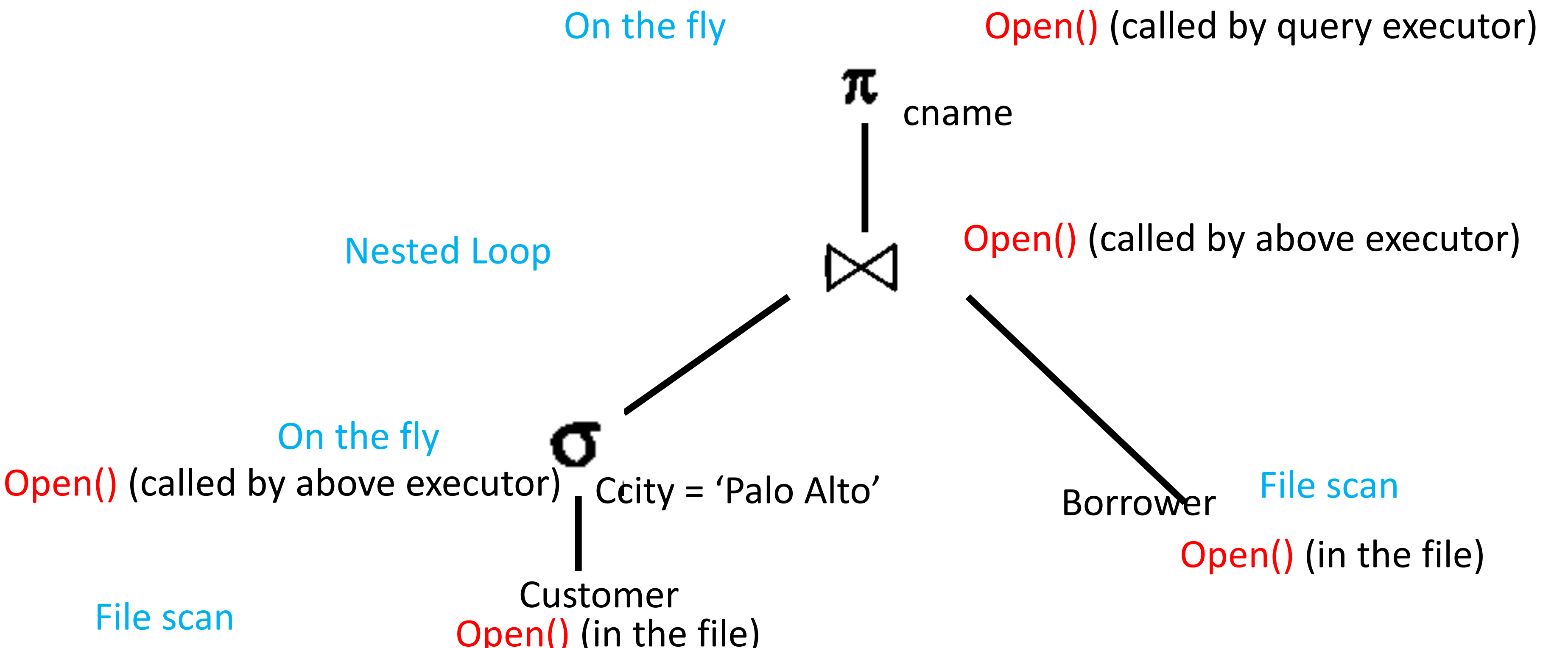


Query Executor

- Each operator implements OpIterator.java
- **open**()
  - Initializes operator state
  - Sets parameters such as selection predicate
- **next**()
  - Returns a Tuple!
  - Operator invokes next() recursively on its inputs
  - Performs processing and produces an output tuple
- **close**():
  - clean-up state
- Operators also have reference to their **child** operator in the query plan

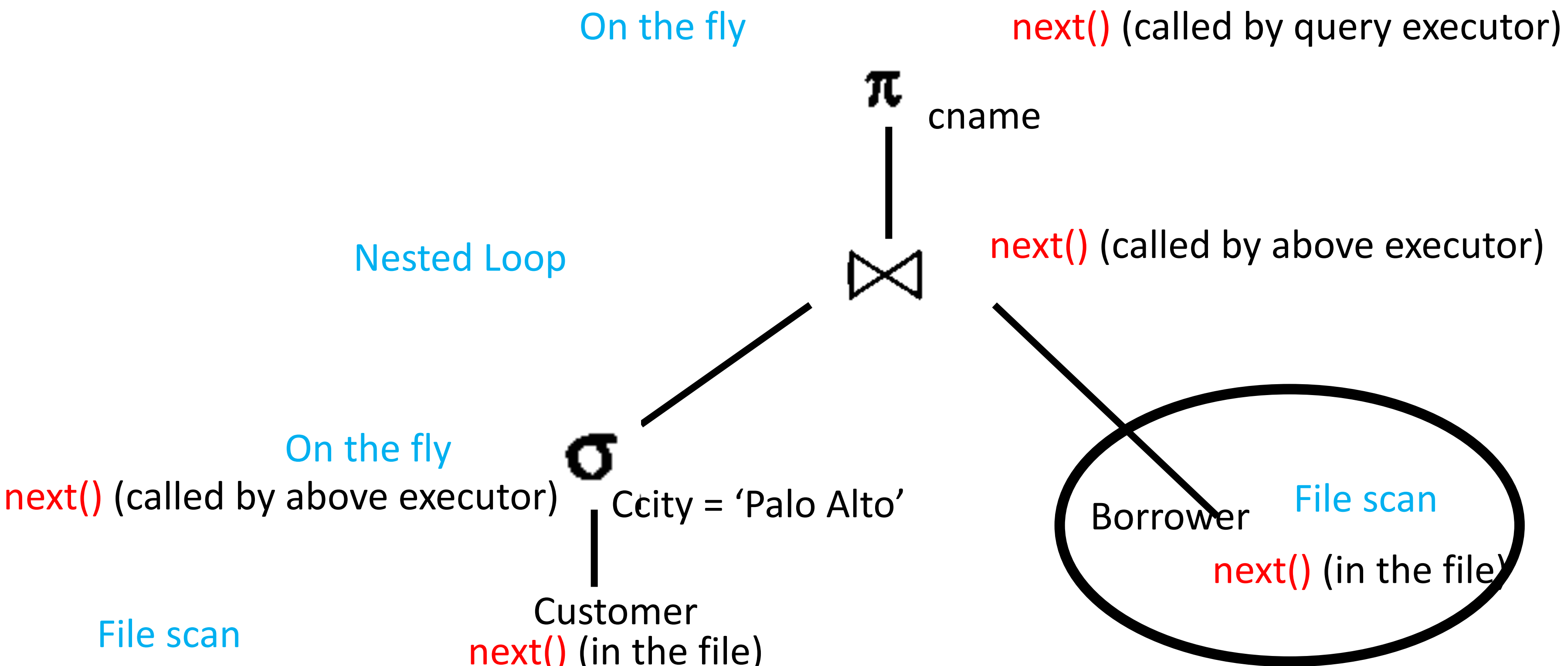
Pipelining

- Physical plan is pipelined i.e., the tuples are processed one-by-one as they pass through the operator



Pipelining

- Pull-based execution



Query Execution In SimpleDB

open()

next()

SeqScan

Operator at bottom of plan

open()

next()

In SimpleDB, SeqScan can find HeapFile in Catalog

Heap File Access Method

Offers iterator interface

- open()
- next()
- close()

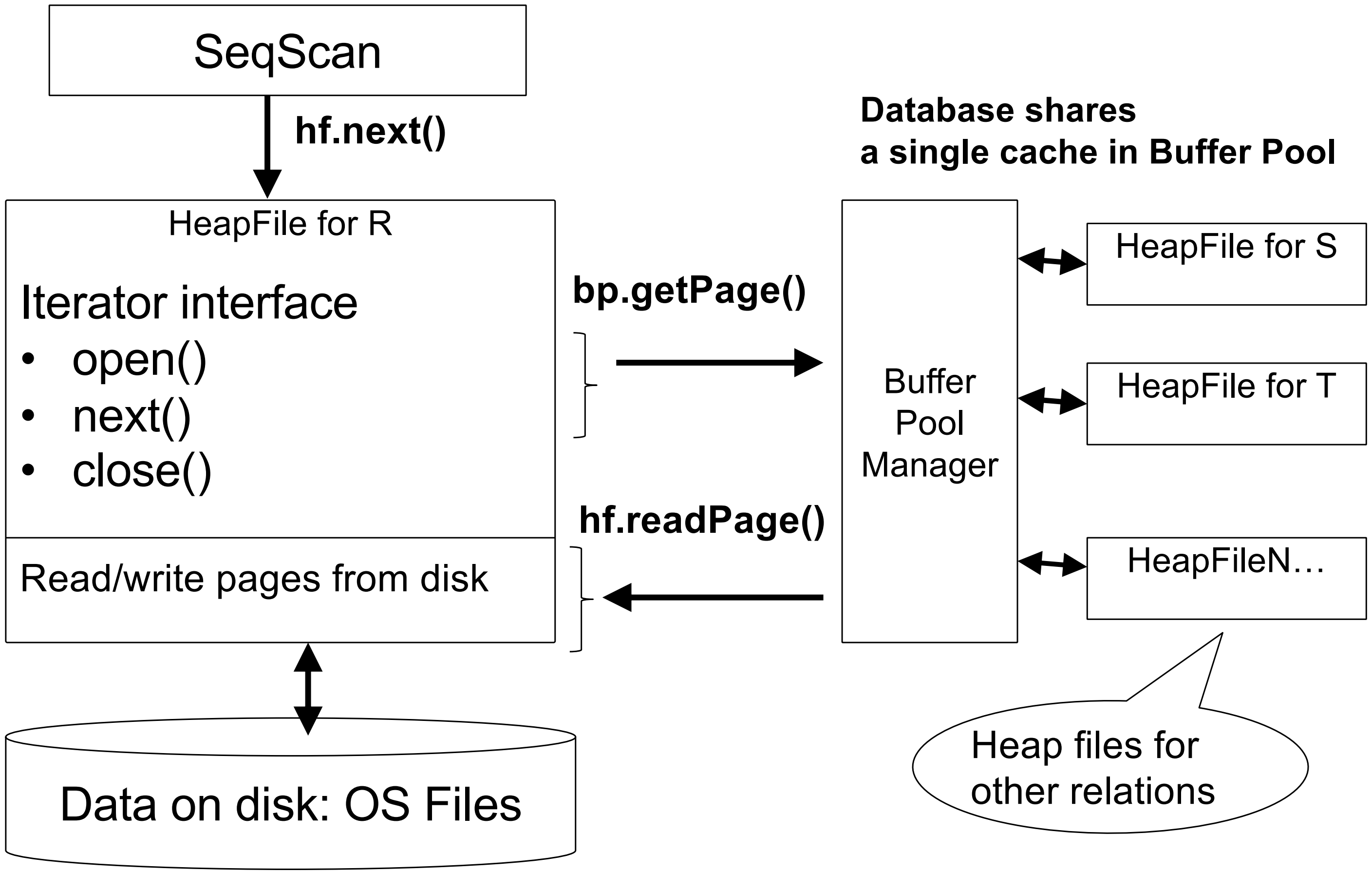
Knows how to read/write pages from disk

But if Heap File reads data directly from disk, it will not stay cached in Buffer Pool!

Iterators in SimpleDB

- SeqScan.java
- DbFileIterator.java
- Both have this method:
  - public Tuple next()
- How does DbFileIterator.java get its tuples?
- Needs pages from buffer pool
- Buffer pool has this method: getPage()

Query Execution In SimpleDB



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
- 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

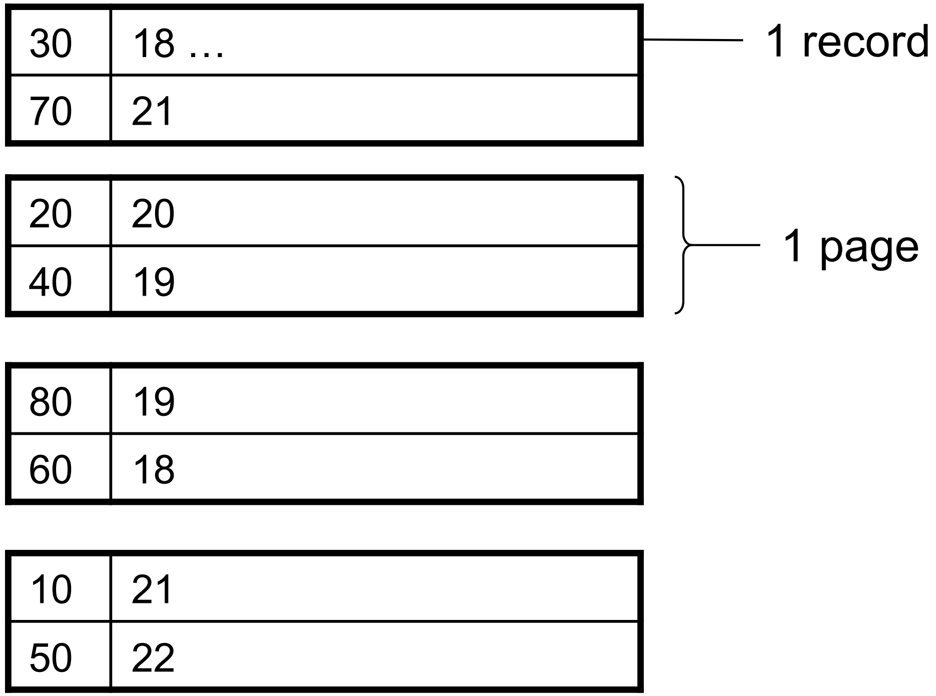
Motivation for Indexing

- 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
  - 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
  - Could do binary search, read log2(1,000) ≈ 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
```

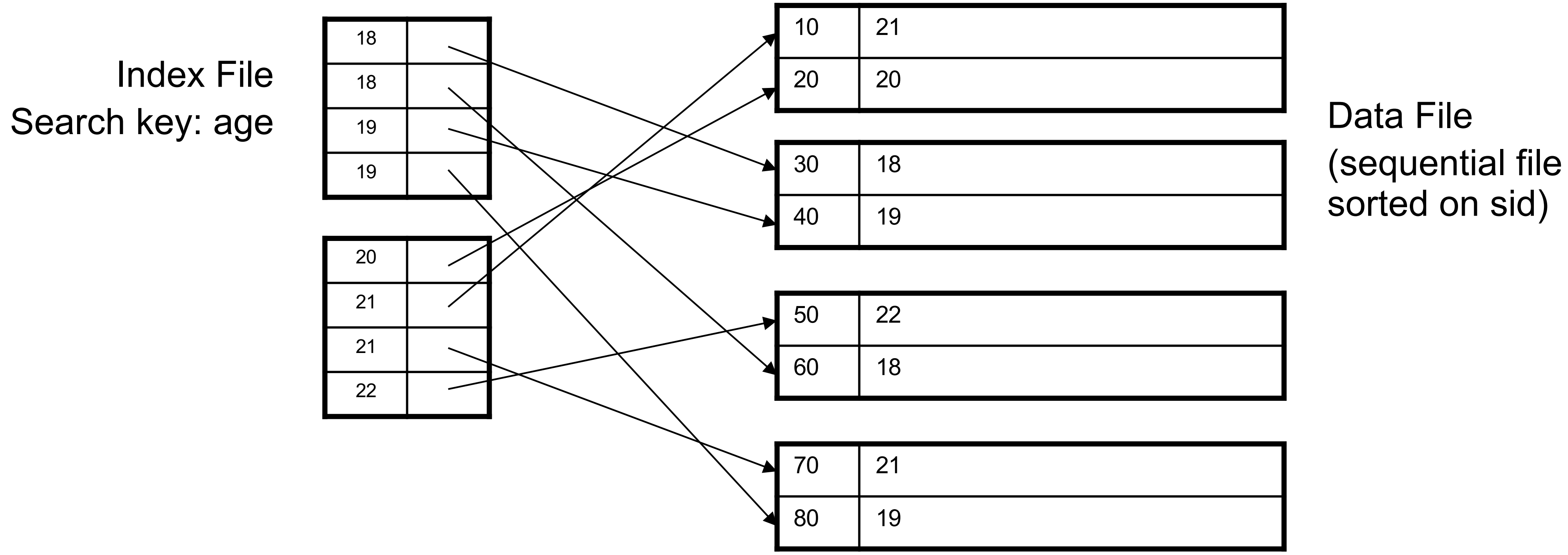
Outline

- 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

Index on a Sequential Data File



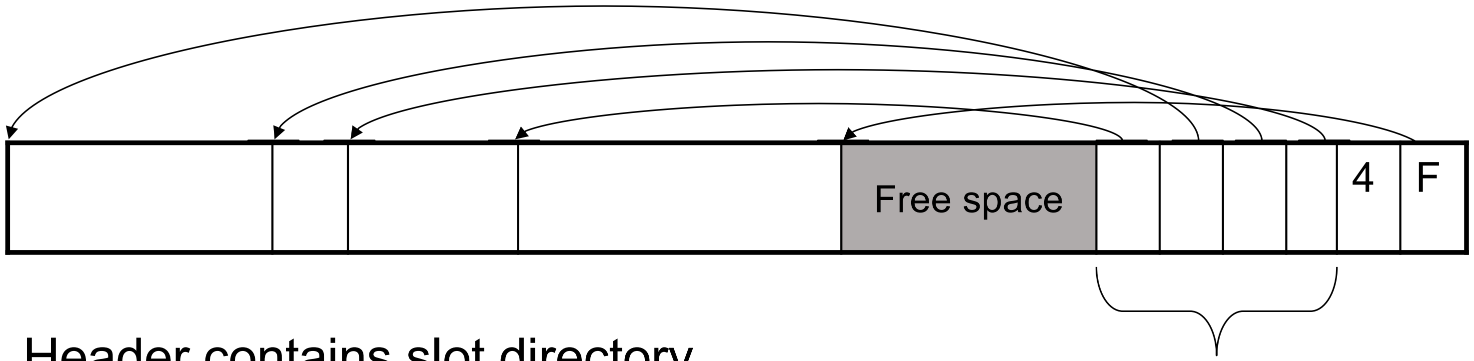
Example

- Total number of pages: 1,000 pages
- Find student whose sid is 80
  - Could do binary search, read log2(1,000) ≈ 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
  - not the same as the primary key, nor a 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

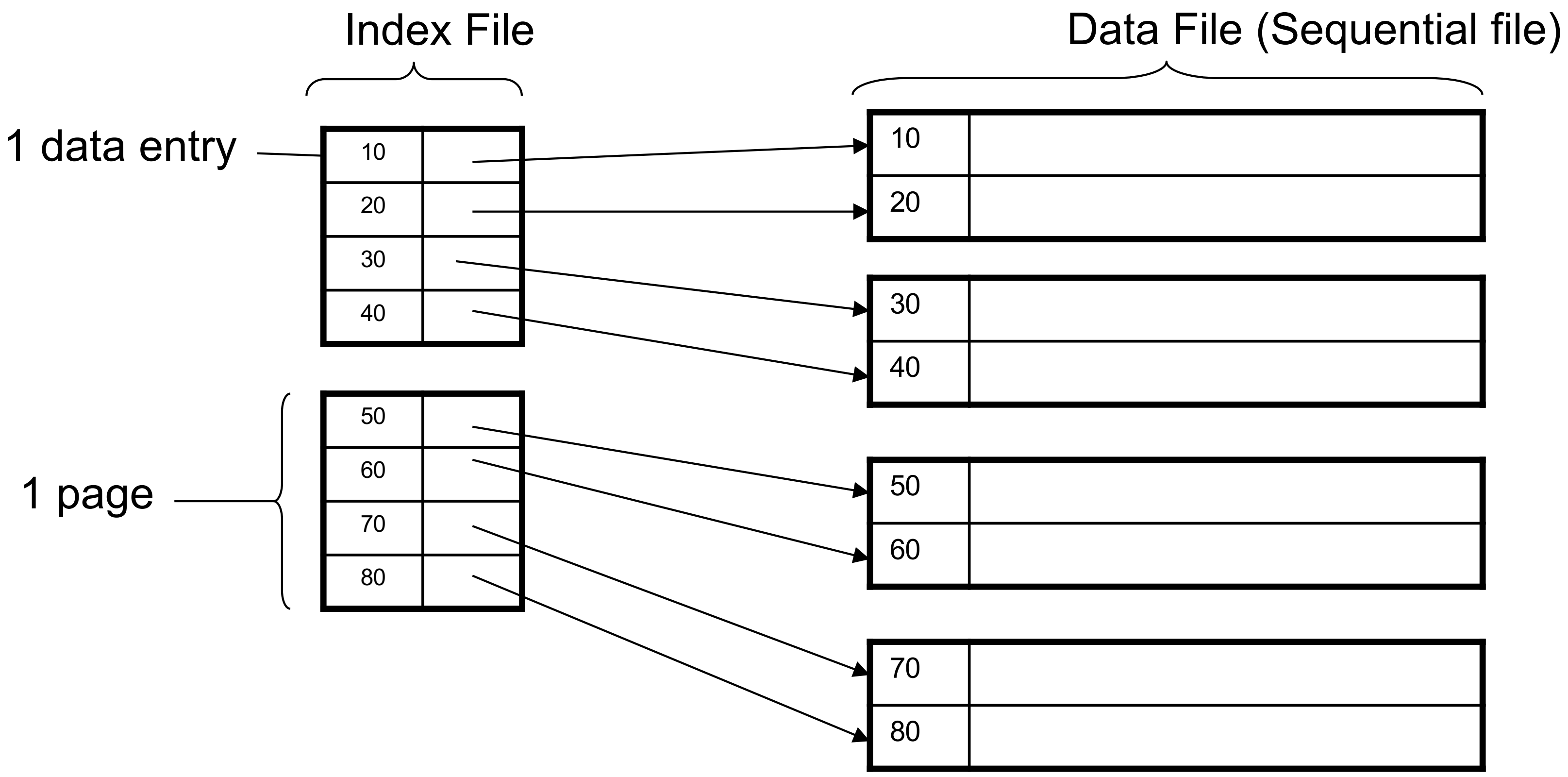
RID is (PageID, SlotID) combination

Different Types of Files

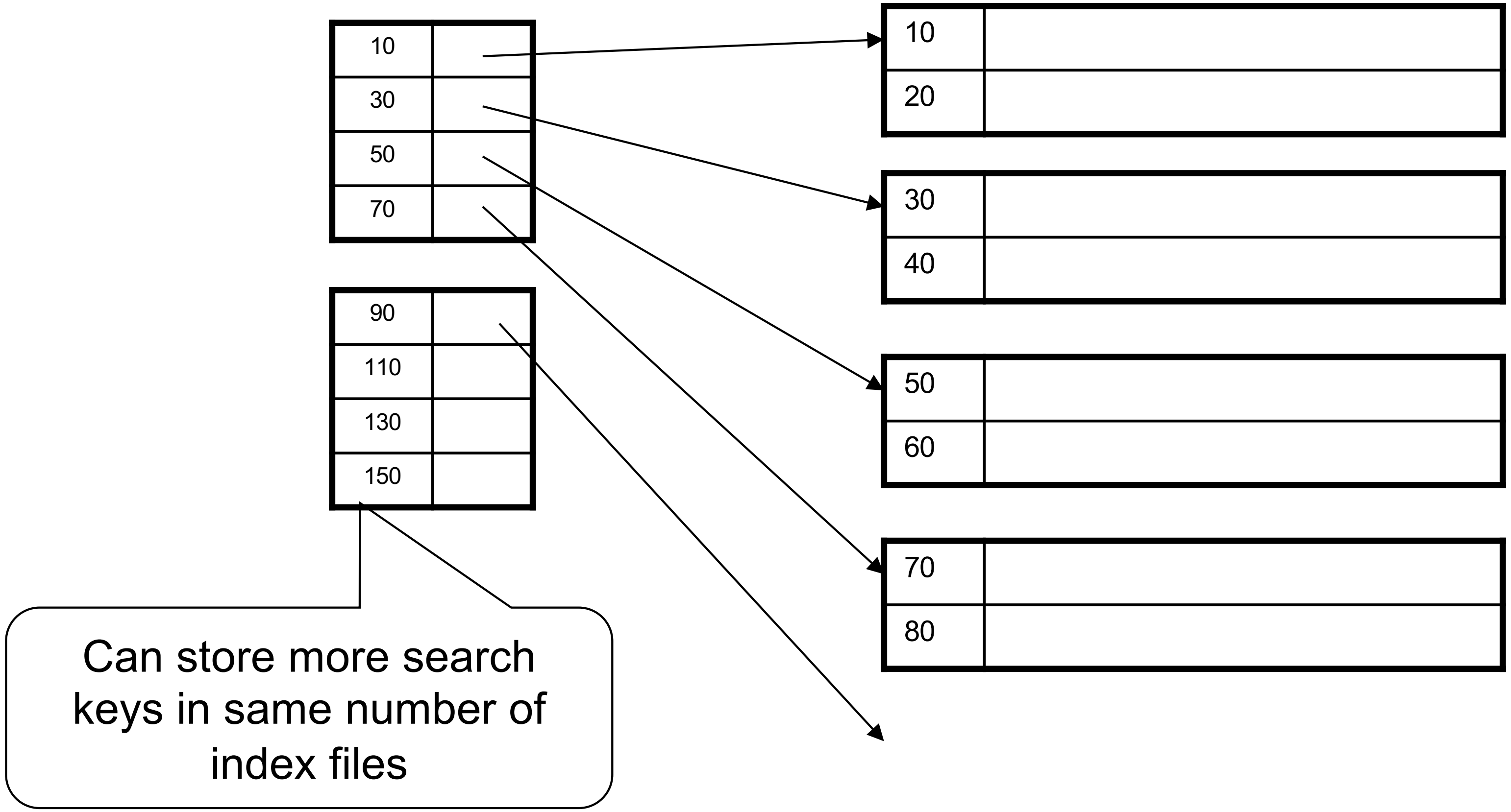
- For the data inside base relations:
  - Heap file (tuples stored without any order)
  - Sequential file (tuples sorted on some attribute(s))
  - Indexed file (tuples organized following an index)
- Then we can have additional index files that store (key,rid) pairs
- Index can also be a "covering index"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



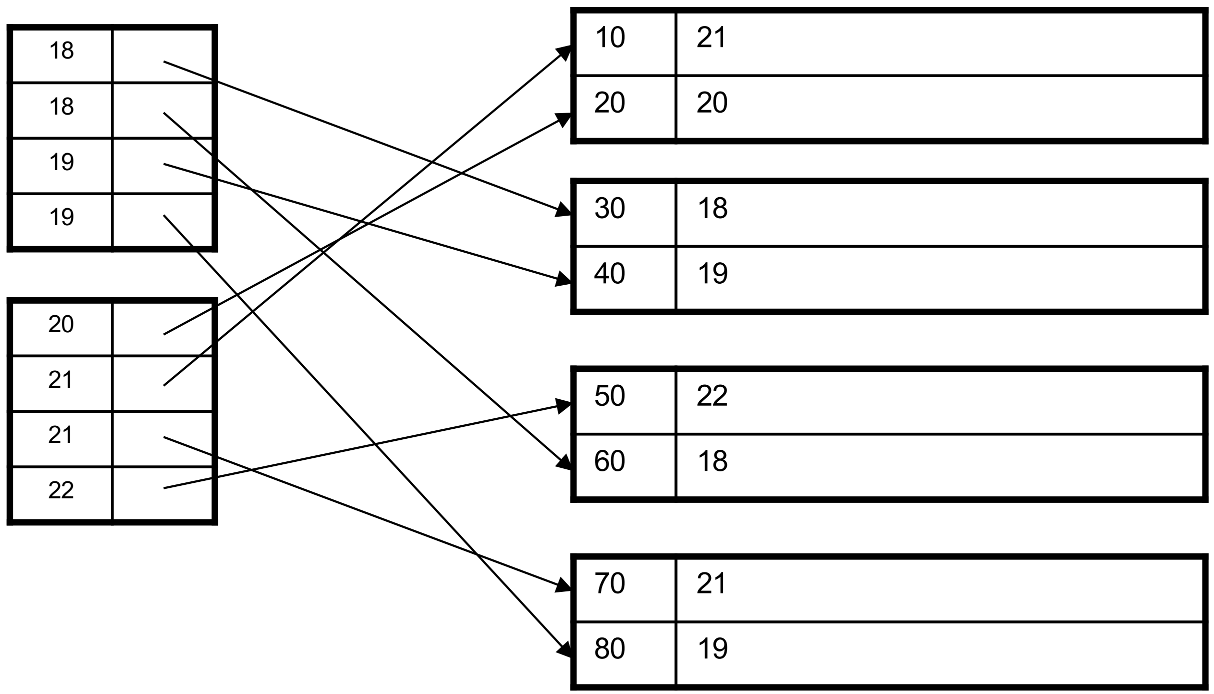
Example

- 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?

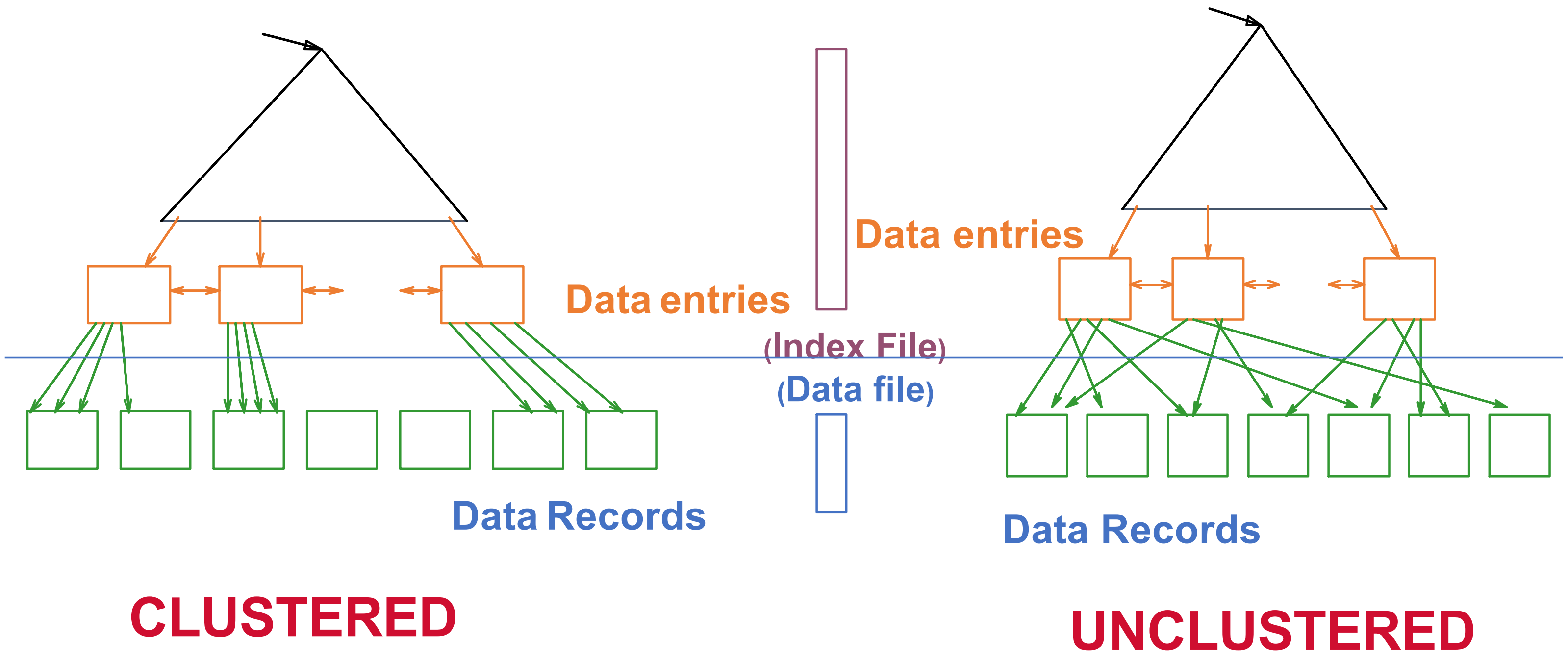
- How can we make *both* queries fast?

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

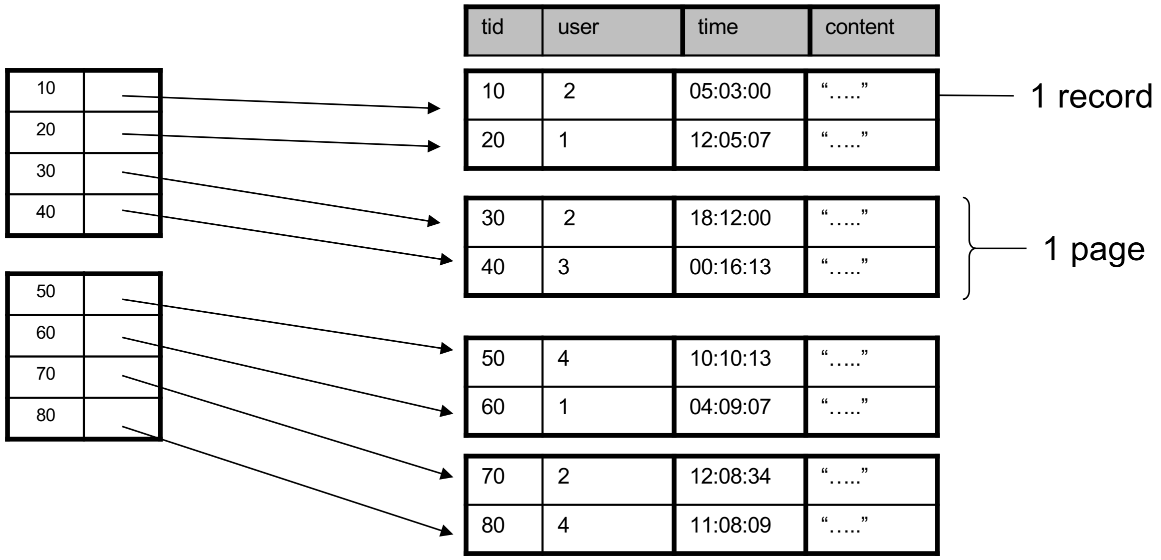
Secondary Index

- 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
  - 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 / ...

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

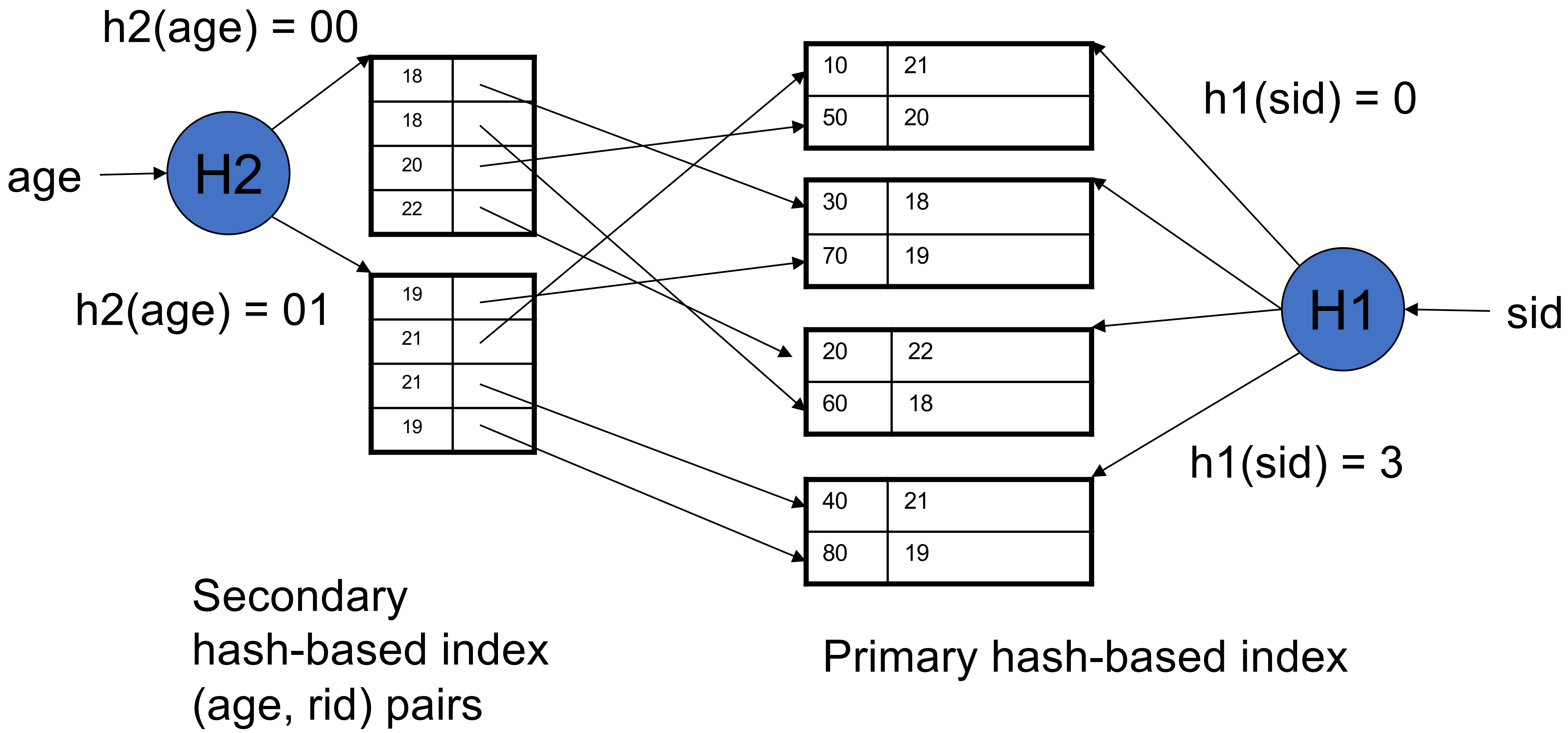
Large Indexes

- 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
- log2 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!

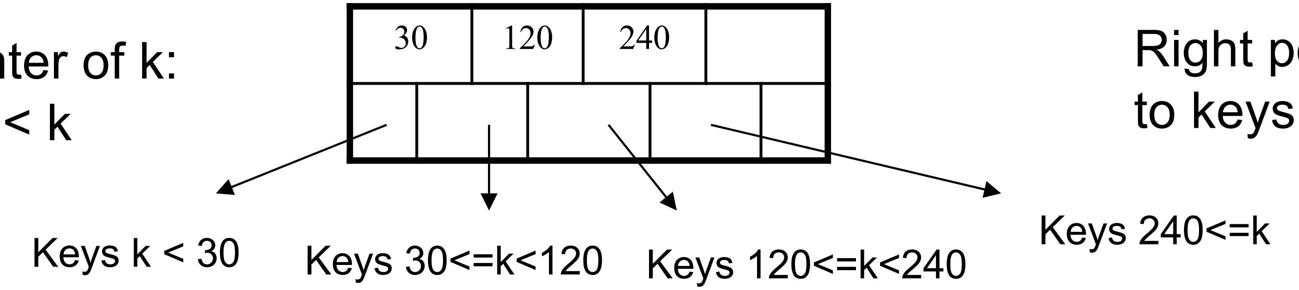
B-tree Vs B+-tree

- Search trees
- Idea in B Trees
  - Make1node=1page(=1block)
- Idea in B+ Trees
  - Keep tree balanced in height – dynamic rather than static
  - Make leaves into a linked list : facilitates range queries

Basics

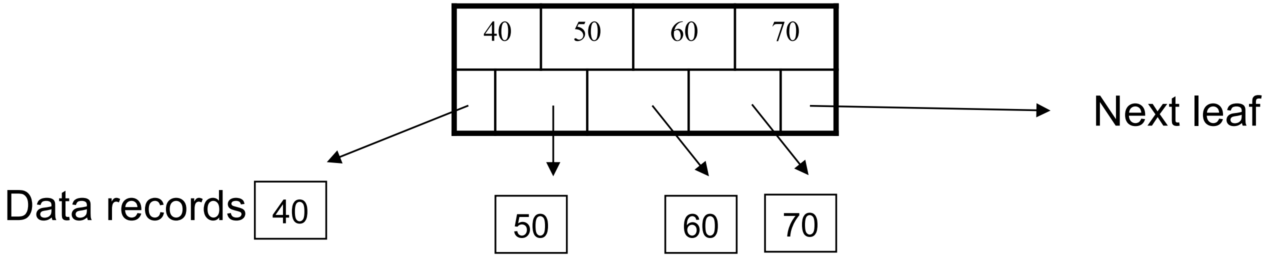
- Parameter d = the *degree*
- Each node has **d <= m <= 2d keys** (except root)
- Each node also has **m+1 pointers**

Left pointer of k:
to keys < k



Right pointer of k:
to keys >= k

- 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

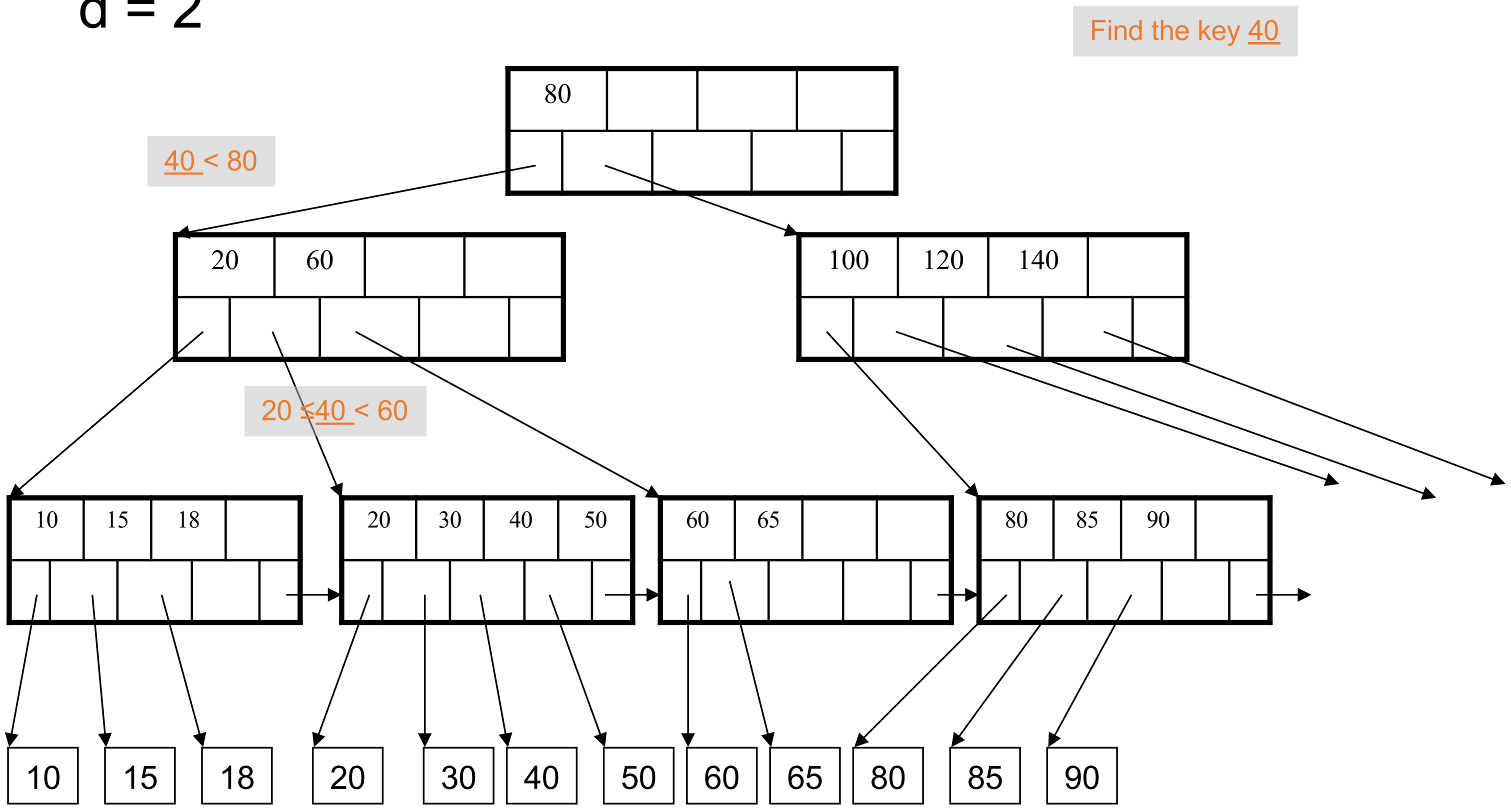
- Search
  - Exact key values:
    - Start at the root
    - Proceed down, to the leaf
  - Range queries:
    - Find lowest bound as above
    - Then sequential traversal

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

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

Example

d = 2



Find the key 40
80
40 < 80
20
60
100
120
140
20 ≤40 < 60
10
15
18
20
30
40
50
60
65
80
85
90
10
15
18
20
30
40
50
60
65
80
85
90

- 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

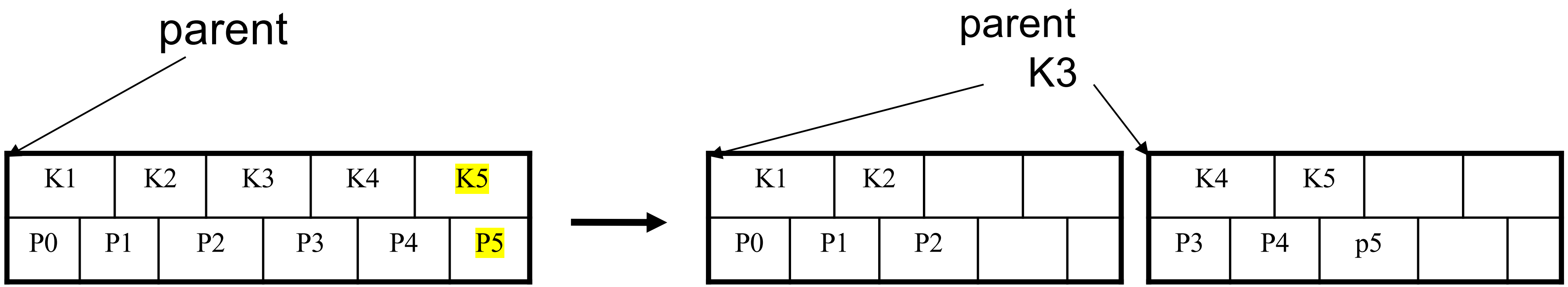
- **d = 170**

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

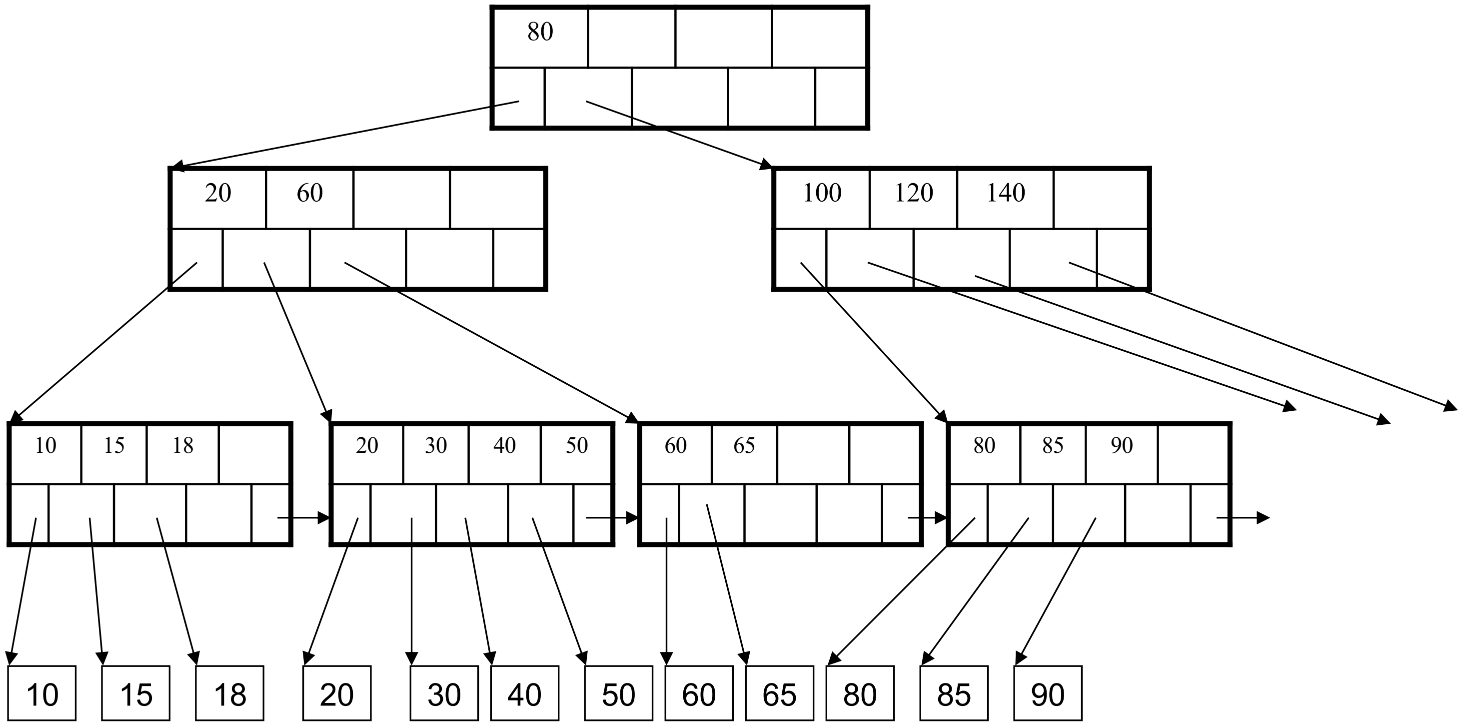
- Insert (K, P)
- Find leaf where K belongs, insert
  If no overflow (2d keys or less), halt
  If overflow (2d+1 keys), split node, insert in parent:



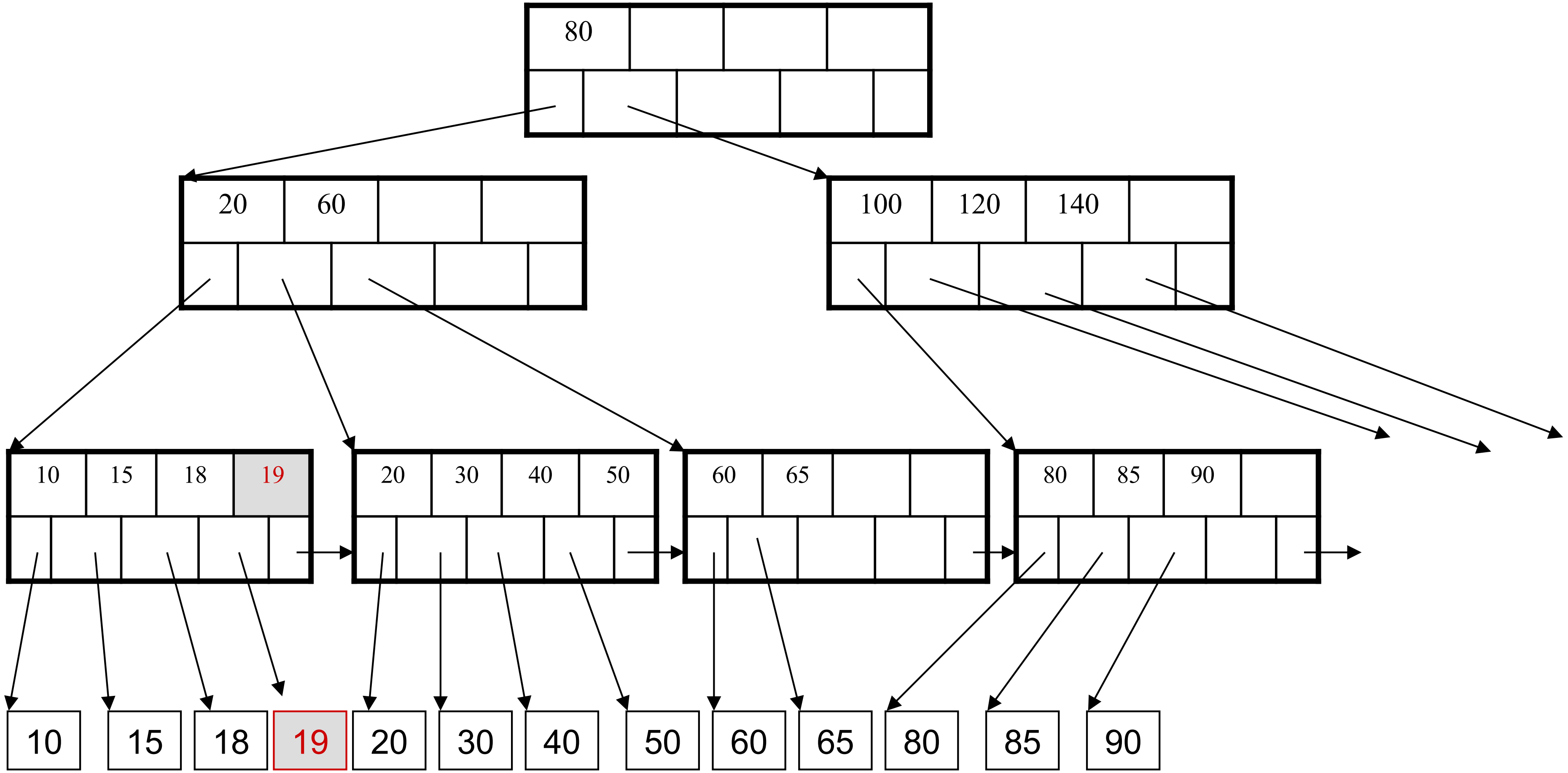
If leaf, also keep K3 in right node
When root splits, new root has 1 key only

Insert

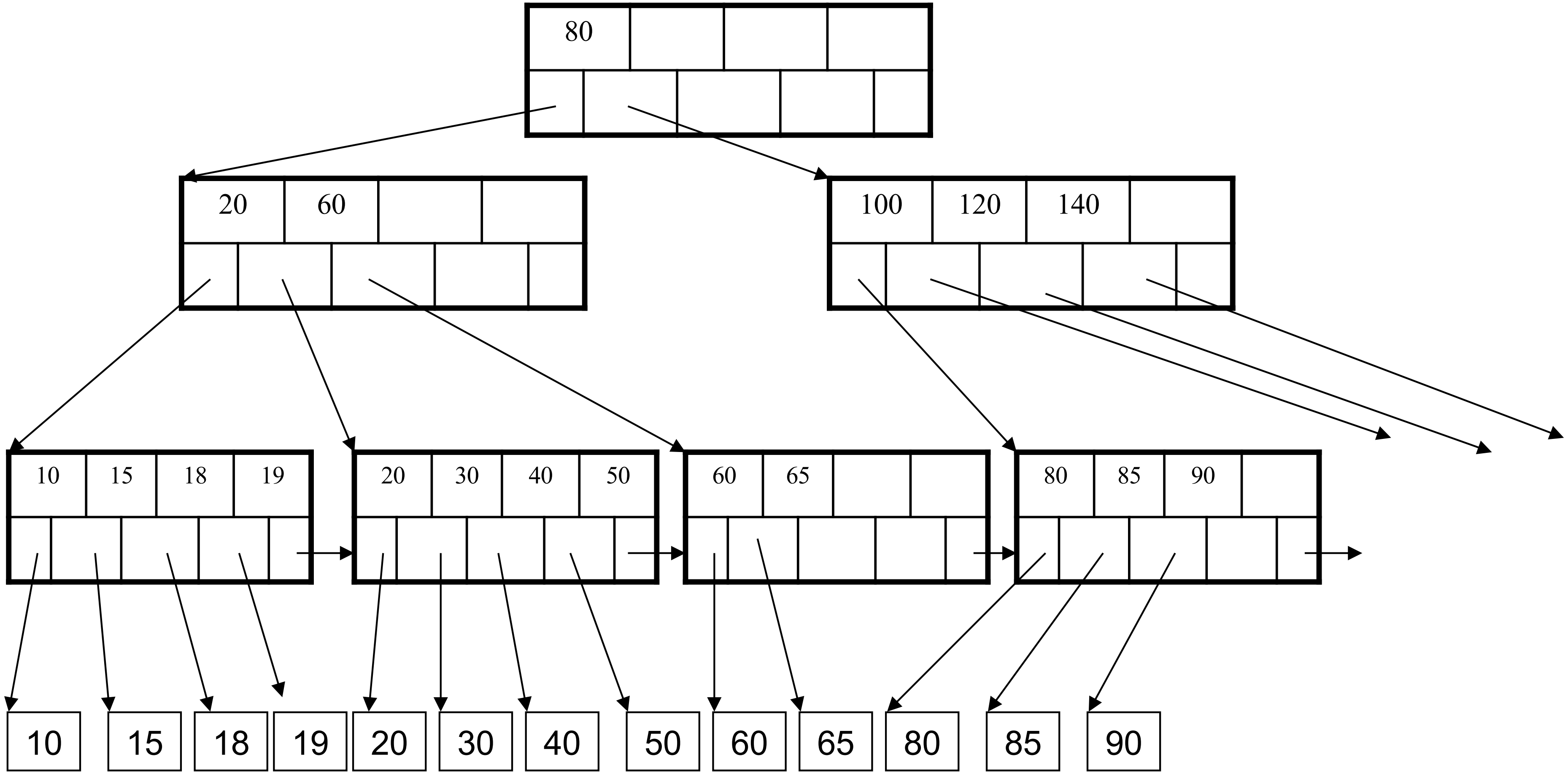
Insert K=19



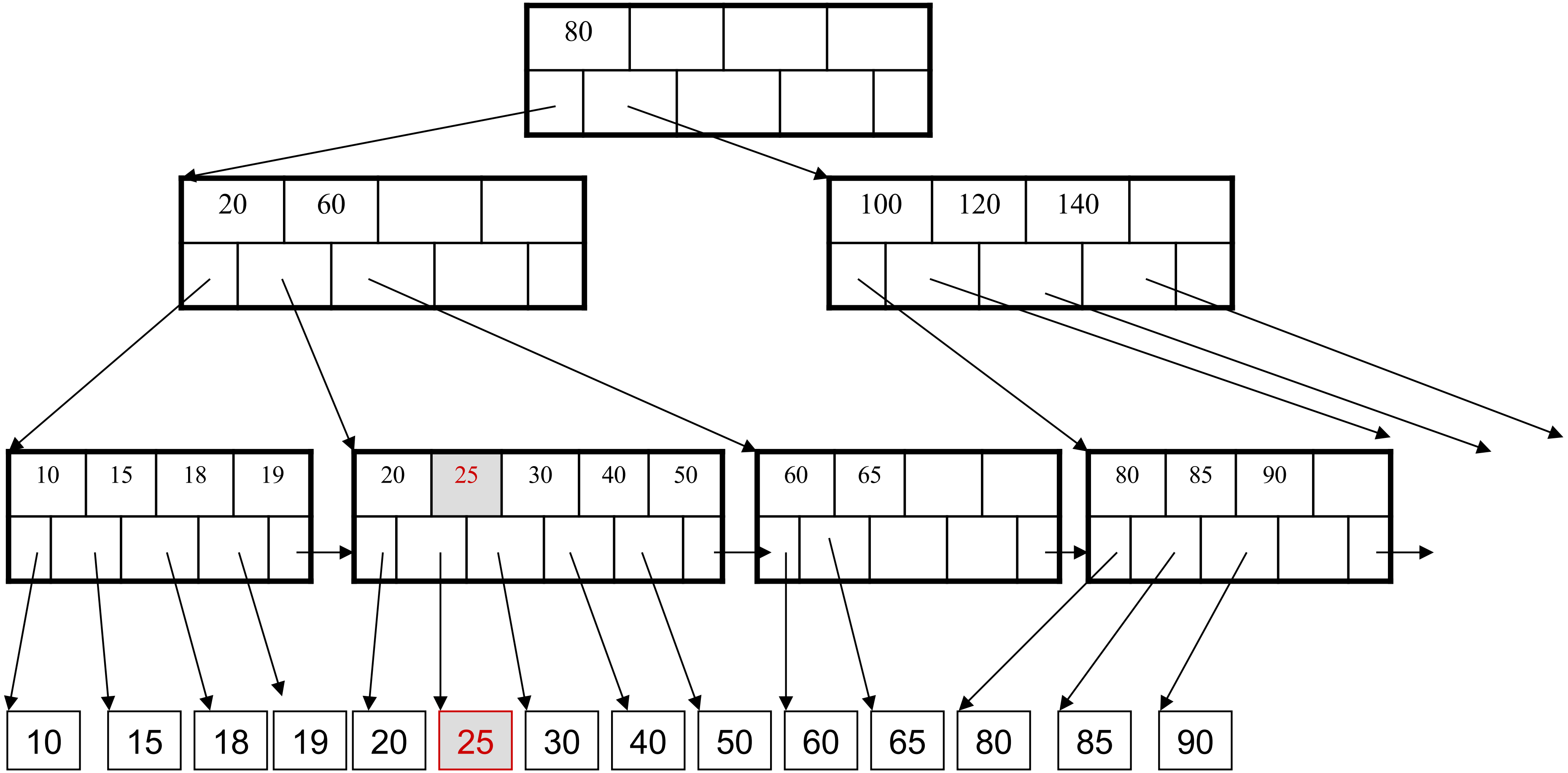
After insertion



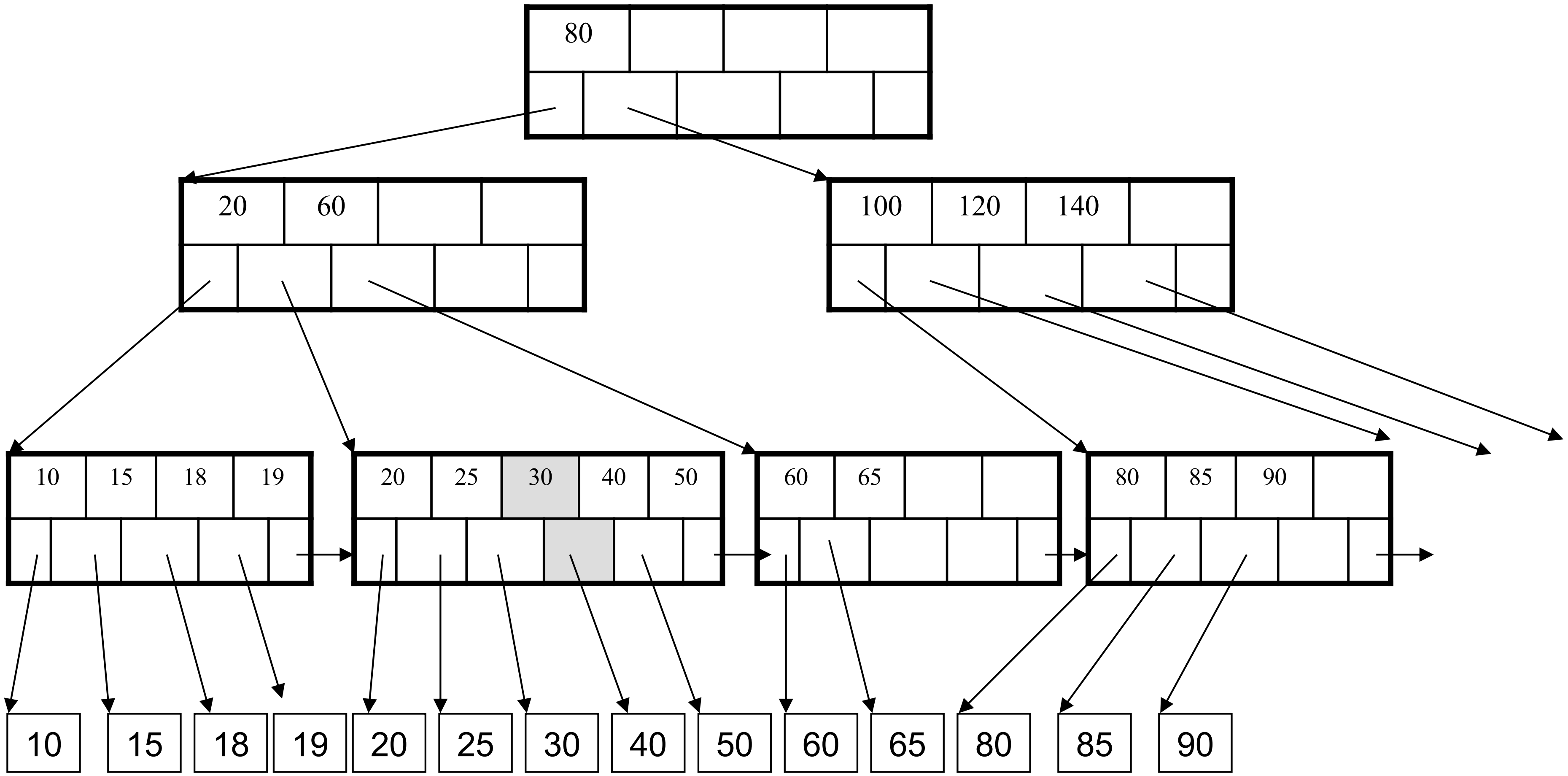
Now insert 25



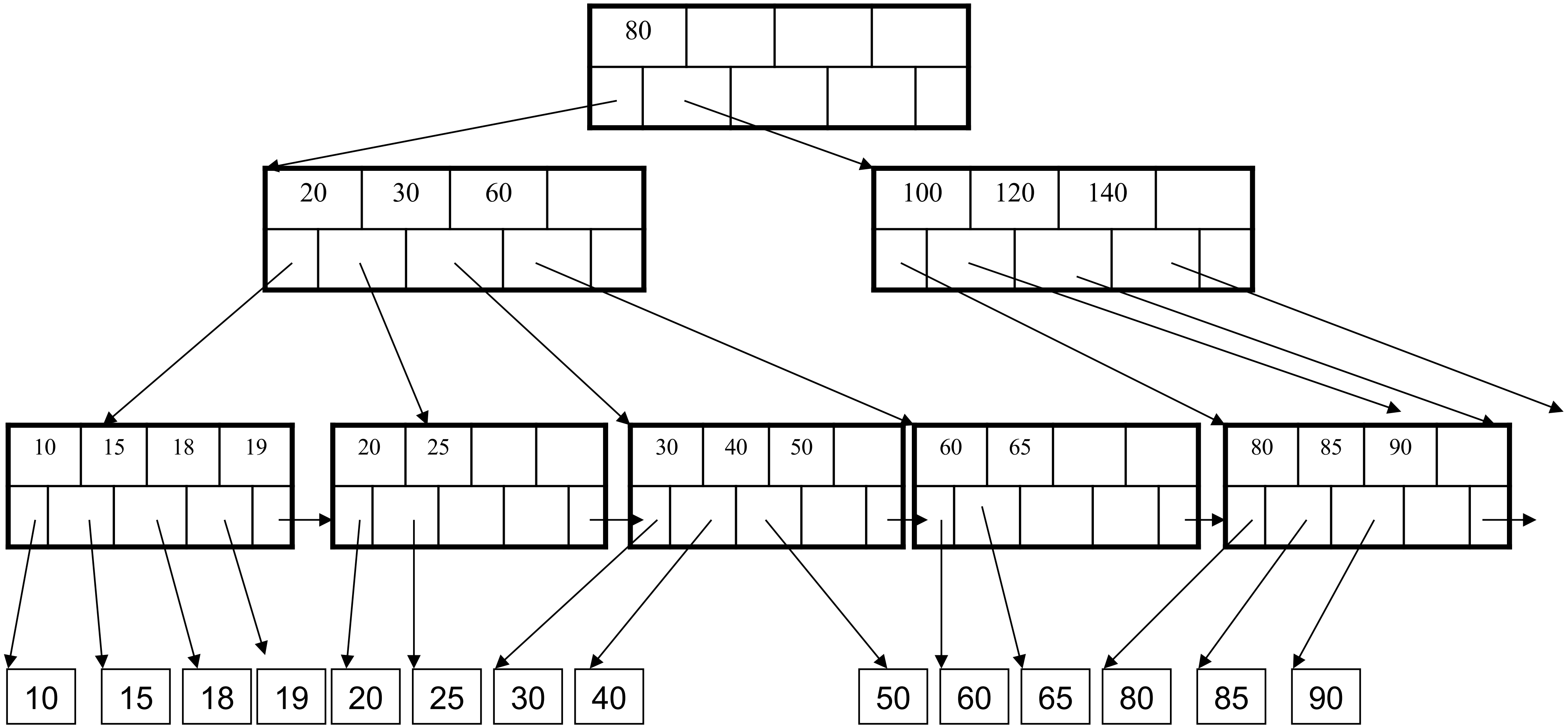
After insertion



But now have to split !



After the split



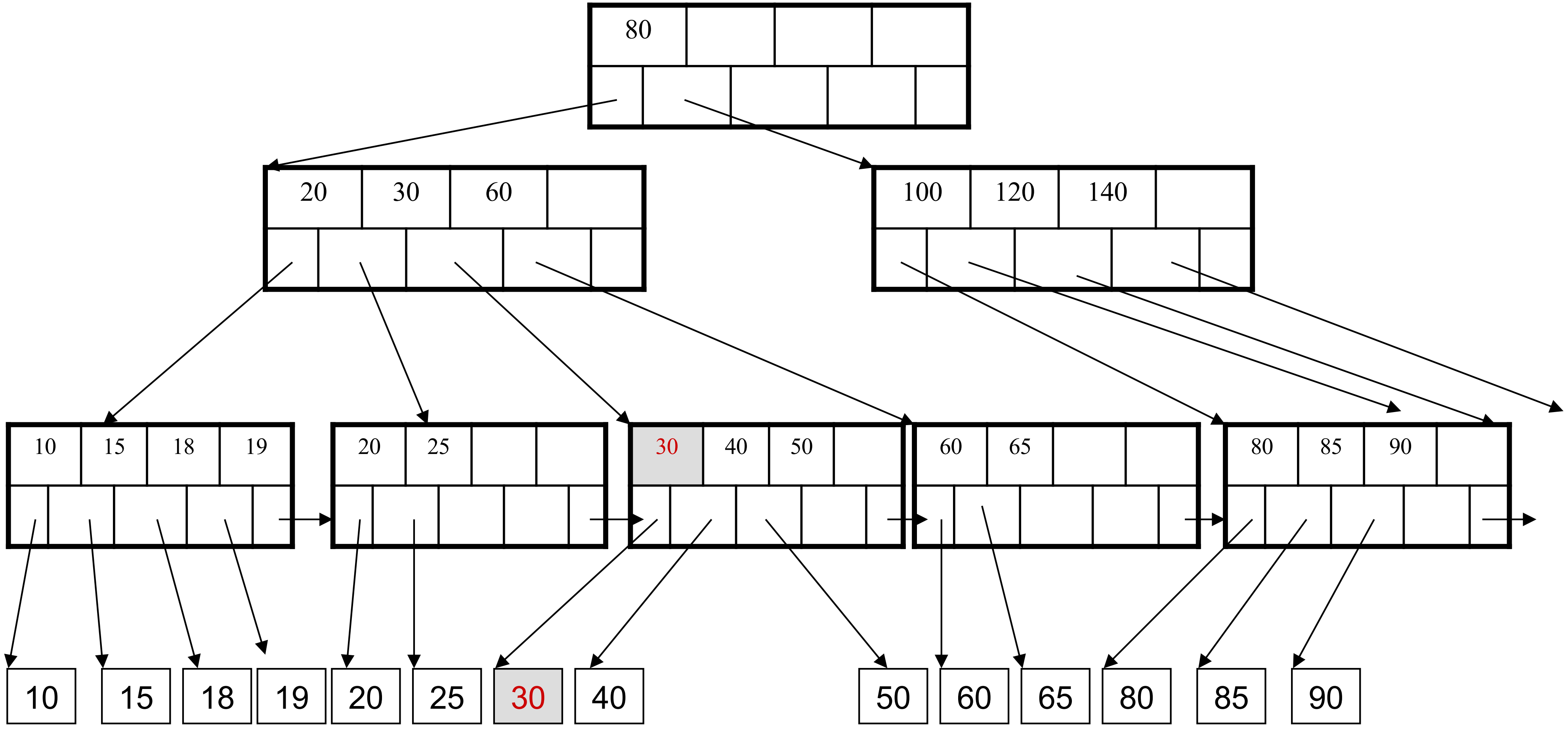
- Note: when a leaf is split, the middle key is copied to the new leaf on **right** (and also inserted in parent)
- Since we assumed the right pointer from key = k points to keys **>=** k

Delete

Delete (K, P)

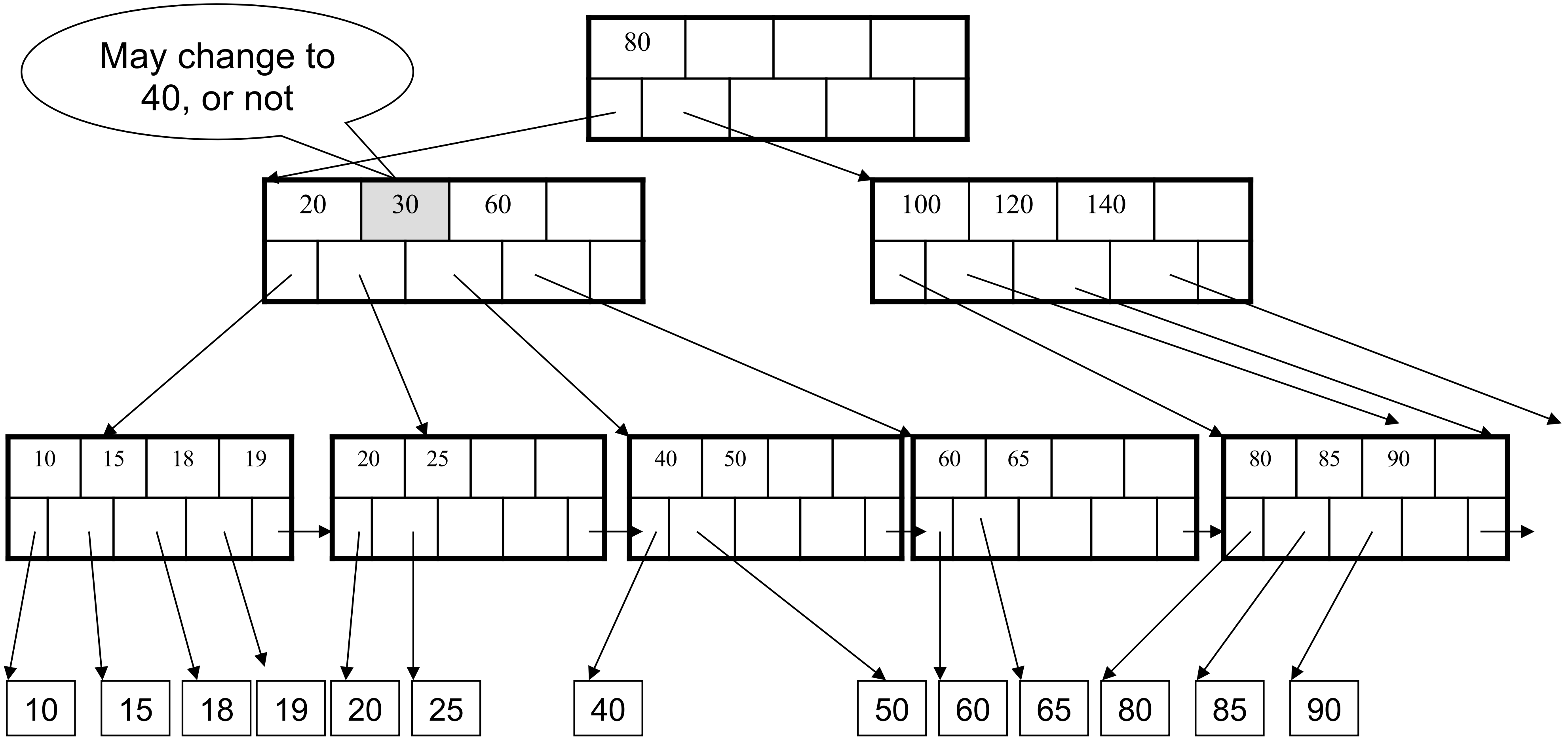
- Find leaf where K belongs, delete
- Check for capacity
- If leaf below capacity, search adjacent nodes (left first, then right) for extra tuples and rotate them to new leaf
- If adjacent nodes at 50% full, merge
- Update and repeat algorithm on parent nodes if necessary

Delete 30

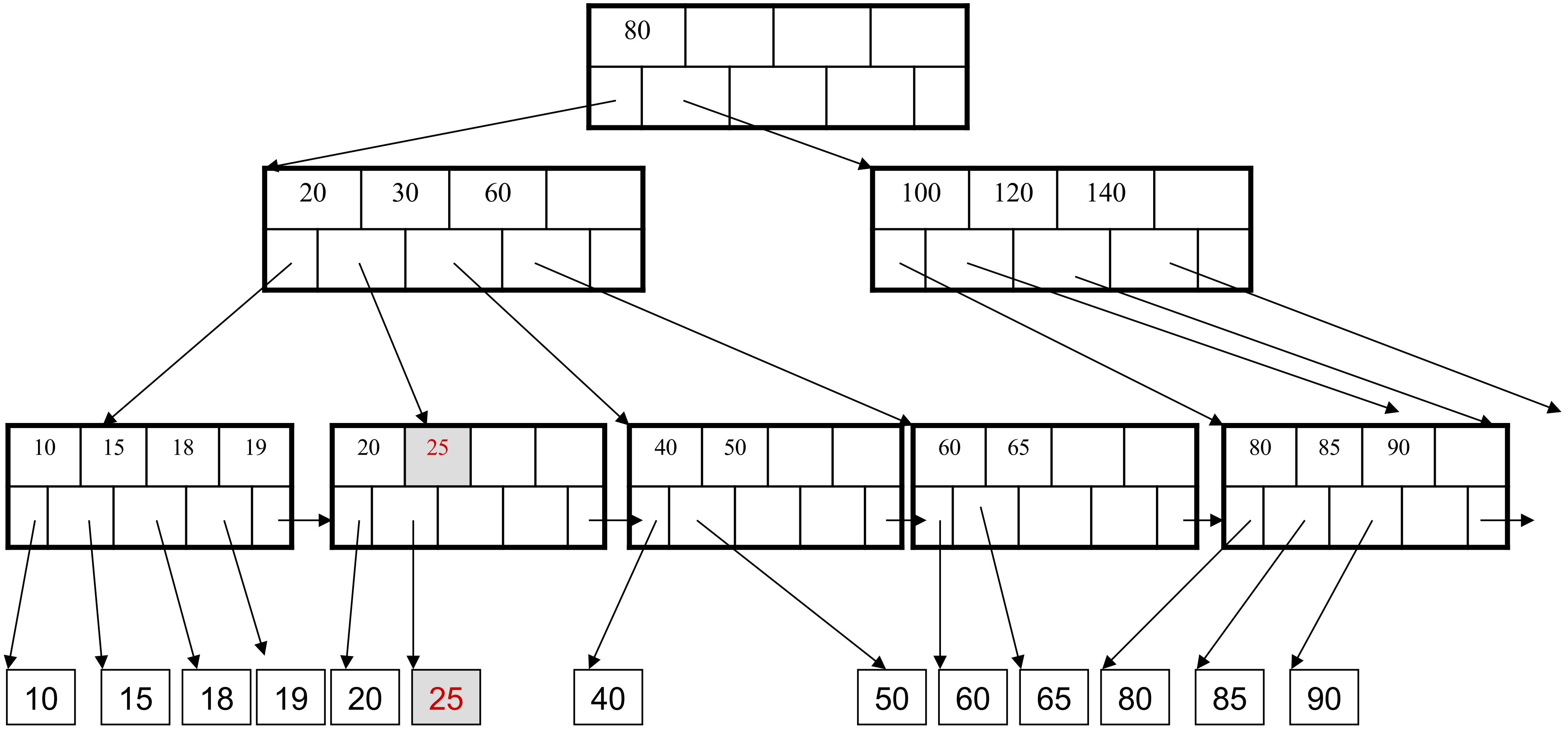


80
20 30 60
100 120 140
10 15 18 19
20 25
30 40 50
60 65
80 85 90
10
15
18
19
20
25
30
40
50
60
65
80
85
90

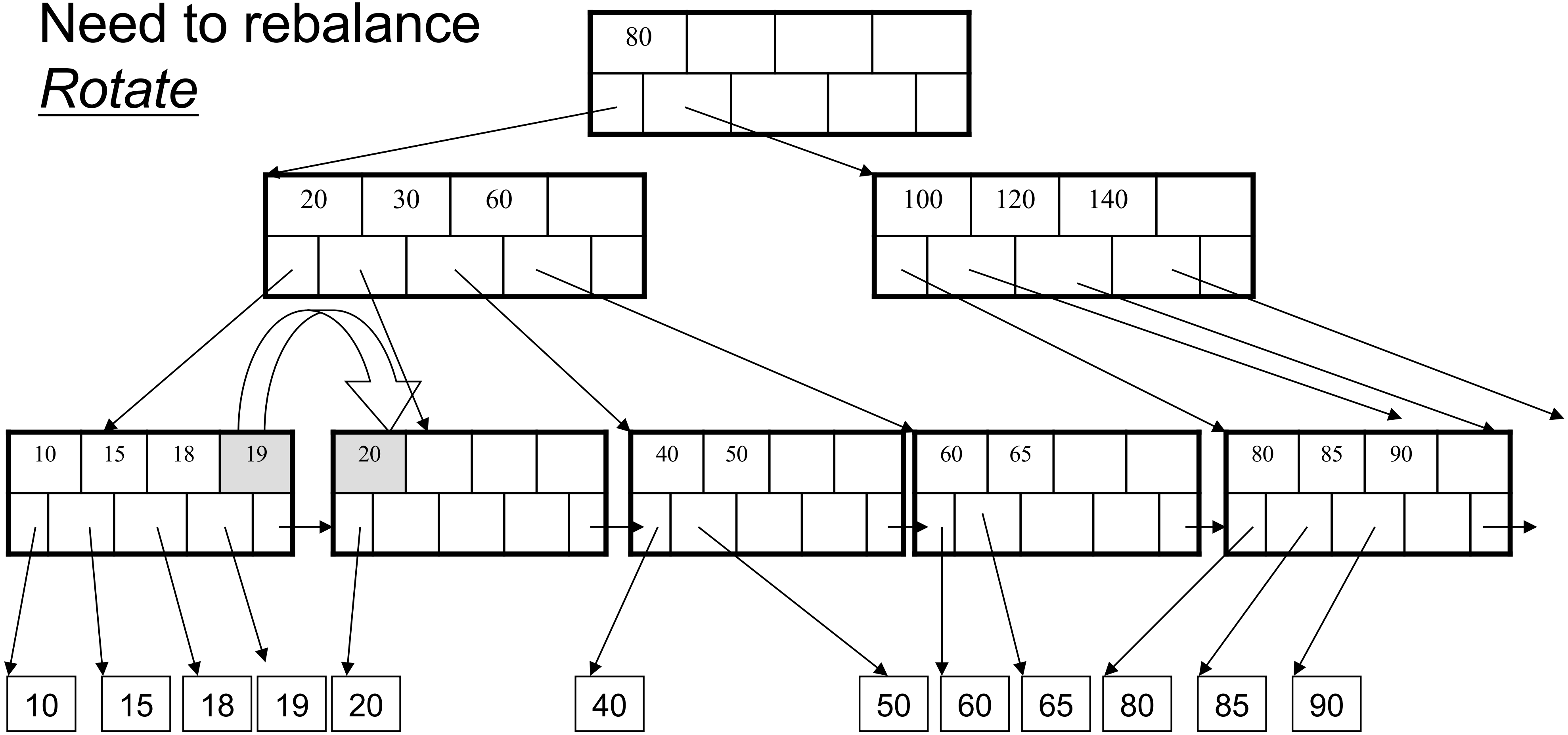
After deleting 30



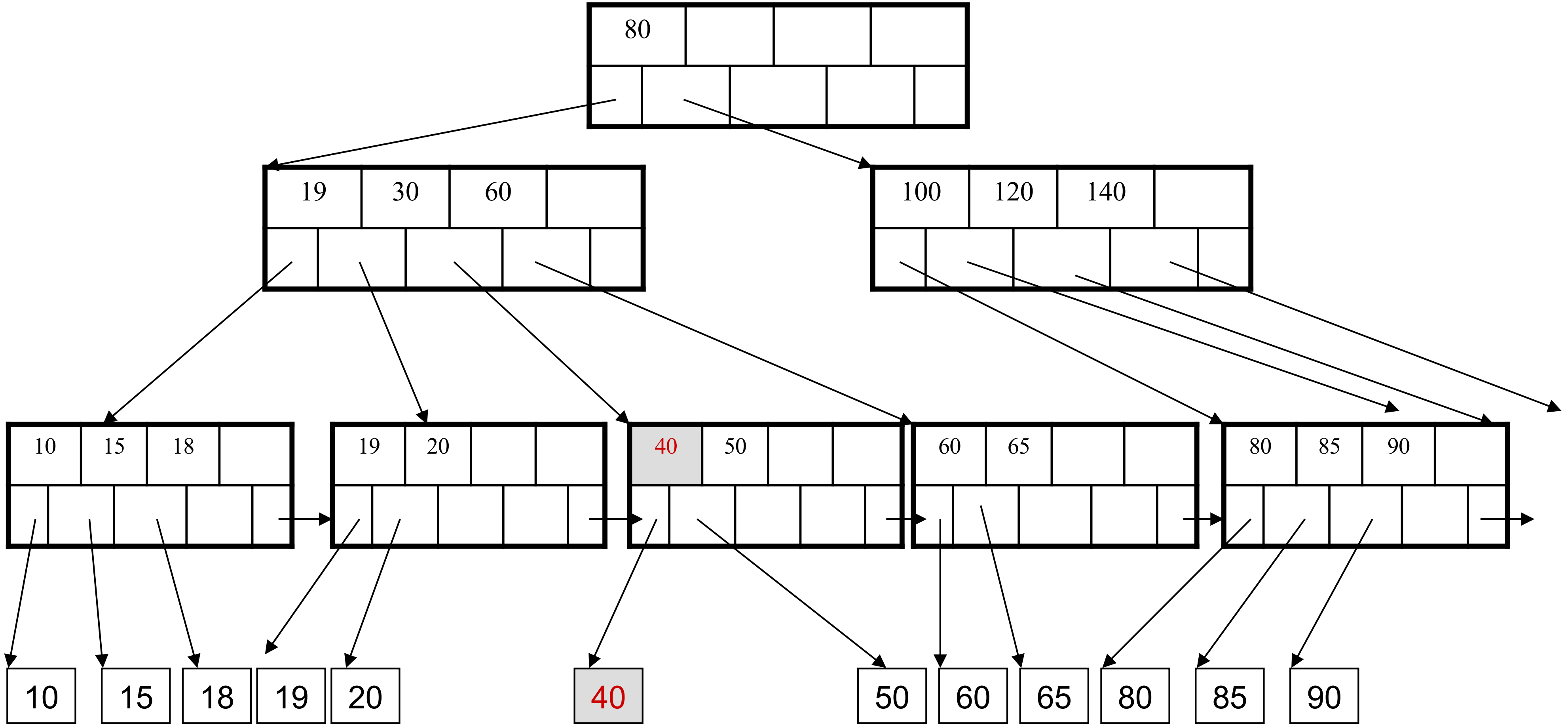
Now delete 25



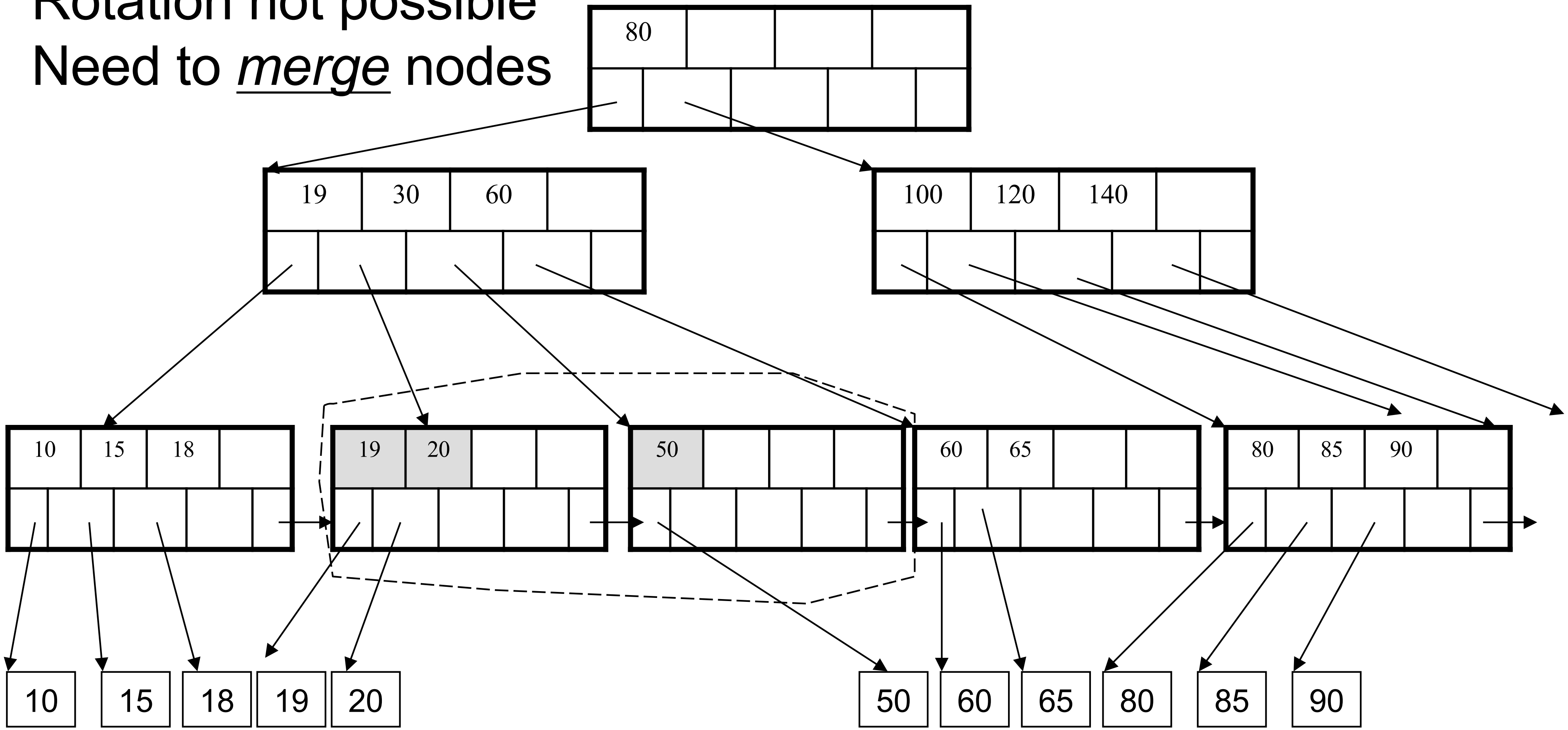
After deleting 25
Need to rebalance
Rotate



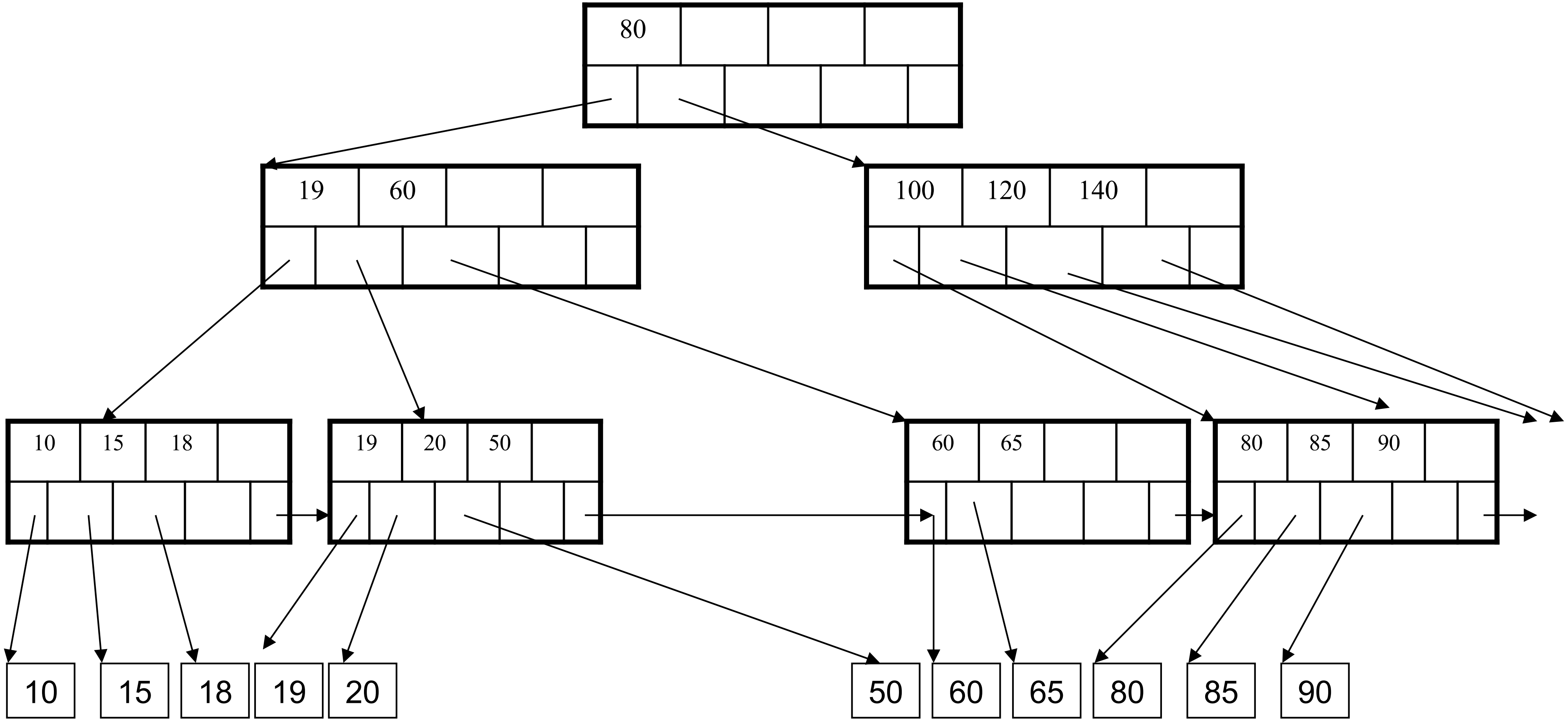
Now delete 40



After deleting 40
Rotation not possible
Need to <u>*merge*</u> nodes



Final tree


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

- 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