Süleyman GÖLBOL
76210123
UP I FAMNIT

Introduction to database systems

Relational model and SQL

Relational data model - terminology

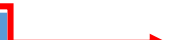
- **Relation** – two-dimensonal table with columns and rows

Name	Gender	Age
Mark	Male	22
Ivo	Male	18
Tadeja	Female	21
Meta	Female	18

Relational data model - terminology

- **Atrribute** – named column of the relation

Name	Gender	Age
Mark	Male	22
Ivo	Male	18
Tadeja	Female	21
Meta	Female	18



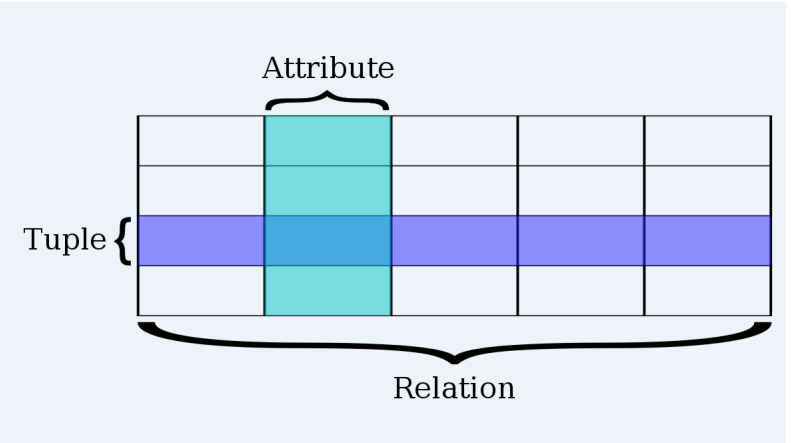
attribute

↳ arity = # of attributes in a relation

- **Domain** – a set of „allowed“ values of one or more attributes
  - Examples: colour: {yellow, red, green, blue}
    date of birth: date

Relational data model - terminology

- **Tuple** – one line in the relation.
- **Cardinality** – number of tuples in a relation.
- **Degree or arity** – number of attributes in a relation.



cardinality

Name	Gender	Age
Mark	Male	22
Ivo	Male	18
Tadeja	Female	21
Meta	Female	18

tuple

Degree of a relation

Standard SQL types in PostgreSQL database

- PostgreSQL supports the following types (and some others):
  - Int (-2 147 483 648 do 2 147 483 647)
  - Smallint (-32 768 do 32 767)
  - real
  - char(N)
  - varchar(N)
  - date
  - time
  - Timestamp

  …

  More: https://www.postgresql.org/docs/9.5/datatype.html

Groups of SQL commands

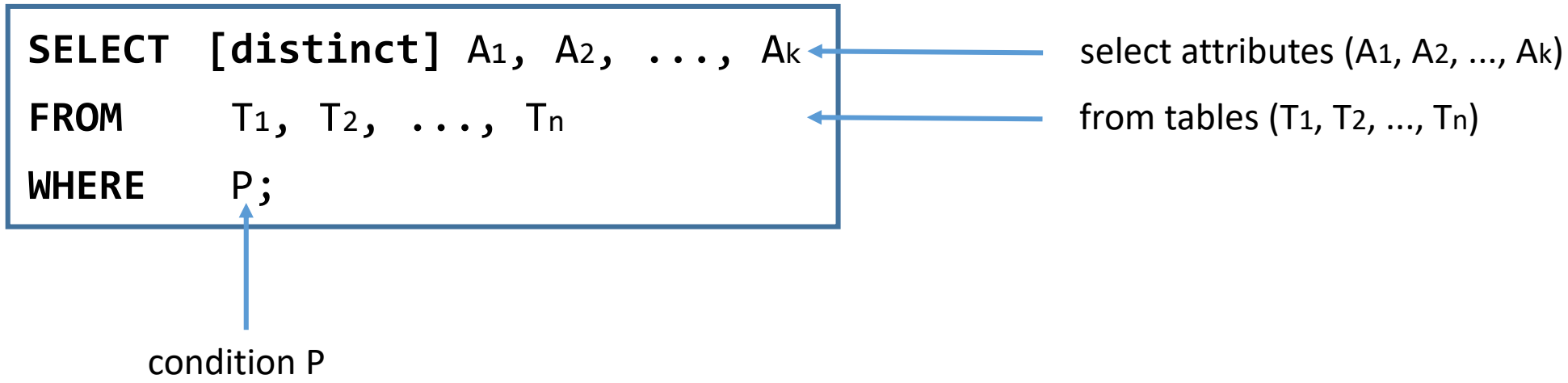
- **DDL (Data Definition Language)**
  - Table creation - CREATE (TABLE, USER, VIEW,...)
  - Table alteration (ALTER TABLE)
  - Table deletion (DROP TABLE)
- **DML (Data Manipulation Language)**
  - **Record selection (SELECT)**
  - Record insertion (INSERT)
  - Record alteration (UPDATE)
  - Record deletion (DELETE)

Groups of SQL commands

- **DCL (Data Control Language)**
  - (GRANT)
  - (REVOKE)
- **TPO (Transaction Processing Option)**
  - (COMMIT)
  - (ROLLBACK)
- **Keys**
  - PRIMARY KEY
    - UNIQUE and NOT NULL
  - FOREIGN KEY
    - REFERENCES

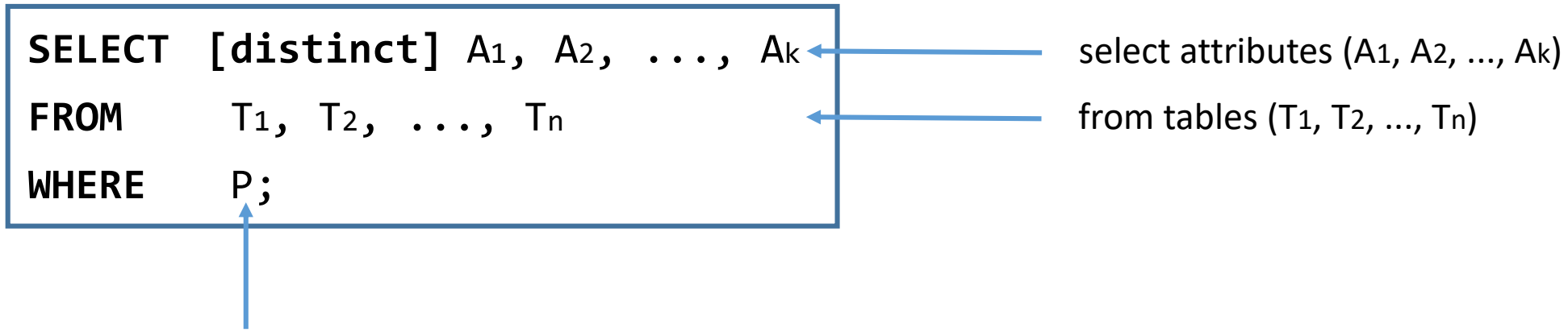
DML group - selections

- **SELECT** for inquiry



DML group - selections

- **SELECT** for inquiry



Selects all the customers from the country "Mexico"

```
SELECT *
FROM Customers
WHERE Country='Mexico';
```

DML group - selections

- **SELECT** for inquiry

```
SELECT   [distinct] A1, A2, ..., Ak
FROM     T1, T2, ..., Tn
WHERE    P
GROUP BY attributes
HAVING   group conditions
ORDER BY attributes, expressions [asc|desc]
LIMIT    number of lines;
```

For selecting all attributes, use *

Number of lines in the output

2472

`Group By X` means **put all those with the same value for X in the one group**.

`Group By X, Y` means **put all those with the same values for both X and Y in the one group**.

To illustrate using an example, let's say we have the following table, to do with who is attending what subject at a university:

```
Table: Subject_Selection

+---------+----------+----------+
| Subject | Semester | Attendee |
+---------+----------+----------+
| ITB001  |        1 | John     |
| ITB001  |        1 | Bob      |
| ITB001  |        1 | Mickey   |
| ITB001  |        2 | Jenny    |
| ITB001  |        2 | James    |
| MKB114  |        1 | John     |
| MKB114  |        1 | Erica    |
+---------+----------+----------+
```

When you use a `group by` on the subject column only; say:

```
select Subject, Count(*)
from Subject_Selection
group by Subject
```

You will get something like:

```
+---------+-------+
| Subject | Count |
+---------+-------+
| ITB001  |     5 |
| MKB114  |     2 |
+---------+-------+
```

...because there are 5 entries for ITB001, and 2 for MKB114

If we were to `group by` two columns:

```
select Subject, Semester, Count(*)
from Subject_Selection
group by Subject, Semester
```

we would get this:

```
+---------+----------+-------+
| Subject | Semester | Count |
+---------+----------+-------+
| ITB001  |        1 |     3 |
| ITB001  |        2 |     2 |
| MKB114  |        1 |     2 |
+---------+----------+-------+
```

DML group - selections

- **SELECT** for inquiry

Selects all the customers from the country "Mexico"

```
SELECT   *
FROM     Customers
WHERE    Country = 'Mexico‘;
```

DML group - selections

- **SELECT** for inquiry

Lists the number of customers in each Mexican city

```
SELECT   COUNT(CustomerID), City
FROM     Customers
WHERE    Country = 'Mexico'
GROUP BY City;
```

DML group - selections

- **SELECT** for inquiry

Lists the number of customers in each Mexican city.

```
SELECT   COUNT(CustomerID), City
FROM     Customers
WHERE    Country = 'Mexico'
GROUP BY City
HAVING COUNT(CustomerID) > 5
ORDER BY City DESC;
```

Include only cities with more than 5 customers

Return a sorted descending by the City column

DML group - selections

- **SELECT** for inquiry

Lists the number of customers in each Mexican city.

```
SELECT   COUNT(CustomerID), City
FROM     Customers
WHERE    Country = 'Mexico'
GROUP BY City
HAVING COUNT(CustomerID) > 5
ORDER BY City DESC
LIMIT 5;
```

Include only with more than 5 customers

Return a sorted descending by the City column

Display only top 5

Operators for comparison

=	equals
!= or <>	differs
> and >=	it is greater or it is greater or equal
< and <=	it is lower or it is lower or equal
IN (...)	corresponds to any element of the set
BETWEEN ... AND ...	is among the given values
LIKE '...%...'	corresponds to a pattern of a string of characters
IS NULL	is an unknown value
NOT ...	operator negation

Operators for comparison

IN (...)

list details of all customers that live in one of the following countries: Mexico, Italy, Spain

```
SELECT    *
FROM      Customers
WHERE     Country IN ('Mexico', 'Italy', 'Spain');
```

BETWEEN ... AND ...

list details of all customers aged between 20 – 30

```
SELECT    *
FROM      Customers
WHERE     age BETWEEN 20 AND 30;
```

LIKE '...%...'

list details of all customers whose name begins with K

```
SELECT    *
FROM      Customers
WHERE     name LIKE 'K%';
```

ORDER by
GROUP by

SQL solutions

EXERCISES 1

Go to postgreSQL: http://www.student.famnit.upr.si/phppgadmin/
Login as:

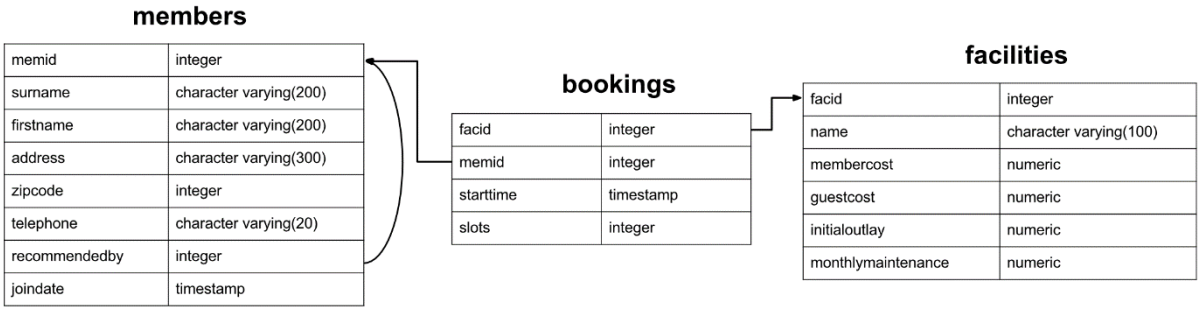
user: OPBvaje
pass: OPBvaje

Click on the server "PostgreSQL" → on the top right click on the "SQL"
Create an username with a password

Execute each of these two lines separately.

```
CREATE USER user_name WITH PASSWORD 'set_password' CREATEDB SUPERUSER;
CREATE DATABASE name_database WITH OWNER user_name TEMPLATE opb_19_20;
```

Logout and login with a new created username.



1. Retrieve all the information from the facilities table

```
SELECT *
FROM facilities;
```

facid	name	membercost	guestcost	initialoutlay	monthlymaintenance
0	Tennis Court 1	5	25	10000	200
1	Tennis Court 2	5	25	8000	200
2	Badminton Court	0	15.5	4000	50
3	Table Tennis	0	5	320	10
4	Massage Room 1	35	80	4000	3000
5	Massage Room 2	35	80	4000	3000
6	Squash Court	3.5	17.5	5000	80
7	Snooker Table	0	5	450	15
8	Pool Table	0	5	400	15

2. Print out a list of all of the facilities and their cost to members.

```
SELECT name, membercost
FROM facilities;
```

name	membercost
Tennis Court 1	5
Tennis Court 2	5
Badminton Court	0
Table Tennis	0
Massage Room 1	35
Massage Room 2	35
Squash Court	3.5
Snooker Table	0
Pool Table	0

3. **Produce a list of facilities that charge a fee to members.**

```
SELECT name
FROM facilities
WHERE membercost > 0;
```

name
Tennis Court 1
Tennis Court 2
Massage Room 1
Massage Room 2
Squash Court

4. **Produce a list of facilities that charge a fee to members, and that fee is less than 1/50th of the monthly maintenance cost.**
   Return the facid, facility name, member cost, and monthly maintenance of the facilities.

```
SELECT facid, name, membercost, monthlymaintenance
FROM facilities
WHERE (membercost > 0) AND (membercost < monthlymaintenance/50.0);
```

facid	name	membercost	monthlymaintenance
4	Massage Room 1	35	3000
5	Massage Room 2	35	3000

5. **Produce a list of all facilities with the word 'Tennis' in their name.**

```
SELECT name
FROM facilities
WHERE name LIKE '%Tennis%';
```

name
Tennis Court 1
Tennis Court 2
Table Tennis

6. **Retrieve the details of facilities with ID 1 and 5? *Do no use the OR operator***

```
select *
from facilities
where facid in (1,5);
```

facid	name	membercost	guestcost	initialoutlay	monthlymaintenance
1	Tennis Court 2	5	25	8000	200
5	Massage Room 2	35	80	4000	3000

7. **Produce a list of members who joined after the start of September 2012. Return the memid, surname, firstname, and joindate of the members.**

```
select memid, surname, firstname, joindate
from members
where joindate >= '2012-09-01';
```

memid	surname	firstname	joindate
24	Sarwin	Ramnaresh	2012-09-01 08:44:42
26	Jones	Douglas	2012-09-02 18:43:05
27	Rumney	Henrietta	2012-09-05 08:42:35
28	Farrell	David	2012-09-15 08:22:05
29	Worthington-Smyth	Henry	2012-09-17 12:27:15
30	Purview	Millicent	2012-09-18 19:04:01
33	Tupperware	Hyacinth	2012-09-18 19:32:05
35	Hunt	John	2012-09-19 11:32:45
36	Crumpet	Erica	2012-09-22 08:36:38
37	Smith	Darren	2012-09-26 18:08:45

8. **Produce an ordered list of the first 10 surnames in the members table? The list must not contain duplicates.**

```
select distinct surname
from members
order by surname
limit 10;
```

surname
Bader
Baker
Boothe
Butters
Coplin
Crumpet
Dare
Farrell
Genting
GUEST

9. Return a combined list of all surnames and all facility names.

```
select surname as "combined"
from members
union
select name
from facilities;
```

combined
Tennis Court 2
Worthington-Smyth
Badminton Court
Pinker
Dare
Bader
Mackenzie
Crumpet
Massage Room 1
Squash Court
Tracy
Hunt
Tupperware
Smith
Butters
Rownam
Baker
Genting
Purview
Coplin
Massage Room 2
Joplette
Stibbons
Rumney
Pool Table
Sarwin
Boothe
Farrell
Tennis Court 1
Snooker Table
Owen
Table Tennis
GUEST
Jones

10. Retrieve the signup date of your last member.

```
select max(joindate) as latest
from members;
```

latest
2012-09-26 18:08:45

11. Figure out how many facilities exist – (produce a total count).

```
select count(*)
from facilities;
```

count
9

12. Produce a count of the number of facilities that have a cost to guests of 10 or more.

```
select count(*)
from facilities
where guestcost >= 10;
```

count
6

13. Produce a count of the number of recommendations each member has made. Order by number of recommendations.

```
select recommendedby as id, count(*)
from members
where recommendedby is not null
group by recommendedby
order by recommendedby;
```

number of recommendations	count
1	5
2	3
3	1
4	2
5	1
6	1
9	2
11	1
13	2
15	1
16	1
20	1
30	1

14. Produce a list of the total number of slots booked per facility.

For now, just produce an output table consisting of facility id and slots, sorted by facility id.

```
select facid, sum(slots) as "Tot Slots"
from bookings
group by facid
order by facid;
```

facid	Tot Slots
0	1320
1	1278
2	1209
3	830
4	1404
5	228
6	1104
7	908
8	911

15. **Produce a list of the total number of slots booked per facility in the month of September 2012. Produce an output table consisting of facility id and slots, sorted by the number of slots.**

```
select facid, sum(slots) as "Tot Slots"
from bookings
where starttime >= '2012-09-01' and starttime < '2012-10-01'
group by facid
order by sum(slots);
```

facid	Tot Slots
5	122
3	422
7	426
8	471
6	540
2	570
1	588
0	591
4	648

16. **Produce a list of the total number of slots booked per facility per month in the year of 2012. Produce an output table consisting of facility id and slots, sorted by the id and month.**
    Hint: Extract month from the start time using 'extract()'

```
select facid, extract(month from starttime) as month, sum(slots) as
"Tot Slots"
from bookings
where starttime >= '2012-01-01' and starttime < '2013-01-01'
group by facid, month
order by facid, month;
```

facid	month	Tot Slots
0	7	270
0	8	459
0	9	591
1	7	207
1	8	483
1	9	588
2	7	180
2	8	459
2	9	570
3	7	104
3	8	304
3	9	422
4	7	264
4	8	492
4	9	648
5	7	24
5	8	82
5	9	122
6	7	164
6	8	400
6	9	540
7	7	156
7	8	326
7	9	426
8	7	117
8	8	322
8	9	471

17. **Print out the total number of members who have made at least one booking.**

```
select count(distinct memid)
from bookings
```

count
30

18. **Produce a list of facilities with more than 1000 slots booked. Produce an output table consisting of facility id and hours, sorted by facility id.**

```
select facid, sum(slots) as "Tot Slots"
from bookings
group by facid
having sum(slots) > 1000
order by facid
```

facid	Tot Slots
0	1320
1	1278
2	1209
4	1404
6	1104

19. Output the facility id that has the highest number of slots booked.

```
select facid, sum(slots) as "Tot Slots"
from bookings
group by facid
order by sum(slots) desc
LIMIT 1;
```

facid	Tot Slots
4	1404

Introduction to database systems

Relational algebra

Intro

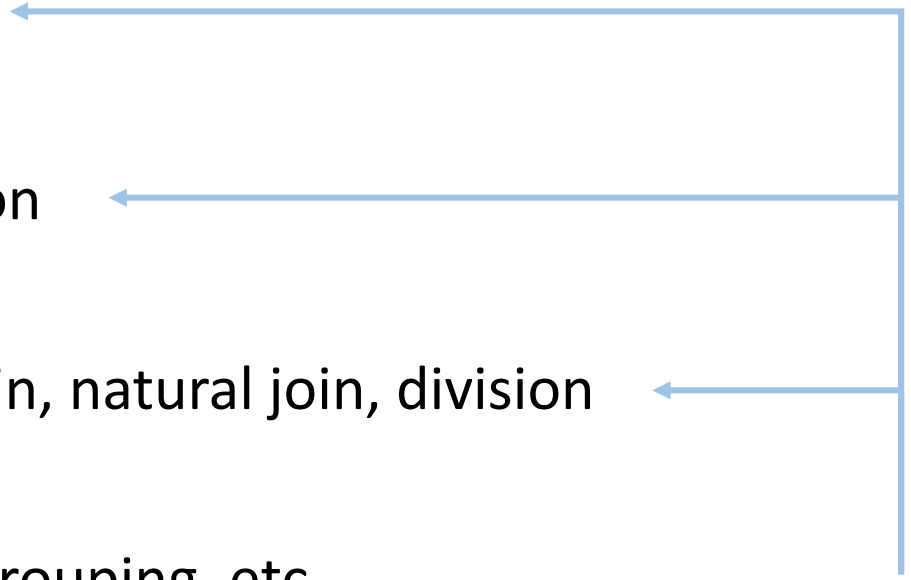
- Relational algebra defines a **sequence of operations** that are carried out over a set of relations.
- The **results** and operands are **relations**.

Groups of relational algebra operations

- **simple** operations:
  - projection, selection, renaming
- **set** operations:
  - union, set-difference, intersection
- **product** operations:
  - cartesian product, θ-join, equijoin, natural join, division
- **other** operations:
  - semijoin, outer join, aggregate, grouping, etc.

Groups of relational algebra operations

- **simple** operations:
  - projection, selection, renaming
- **set** operations:
  - union, set-difference, intersection
- **product** operations:
  - cartesian product, θ-join, equijoin, natural join, division
- **other** operations:
  - semijoin, outer join, aggregate, grouping, etc.

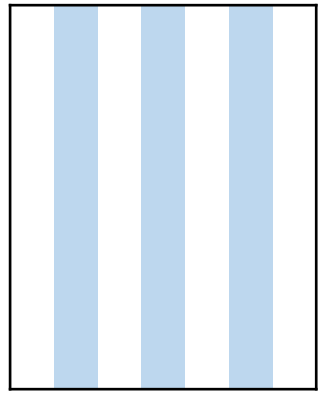


Basic operations with which we can derive others.

Simple operations

Projection

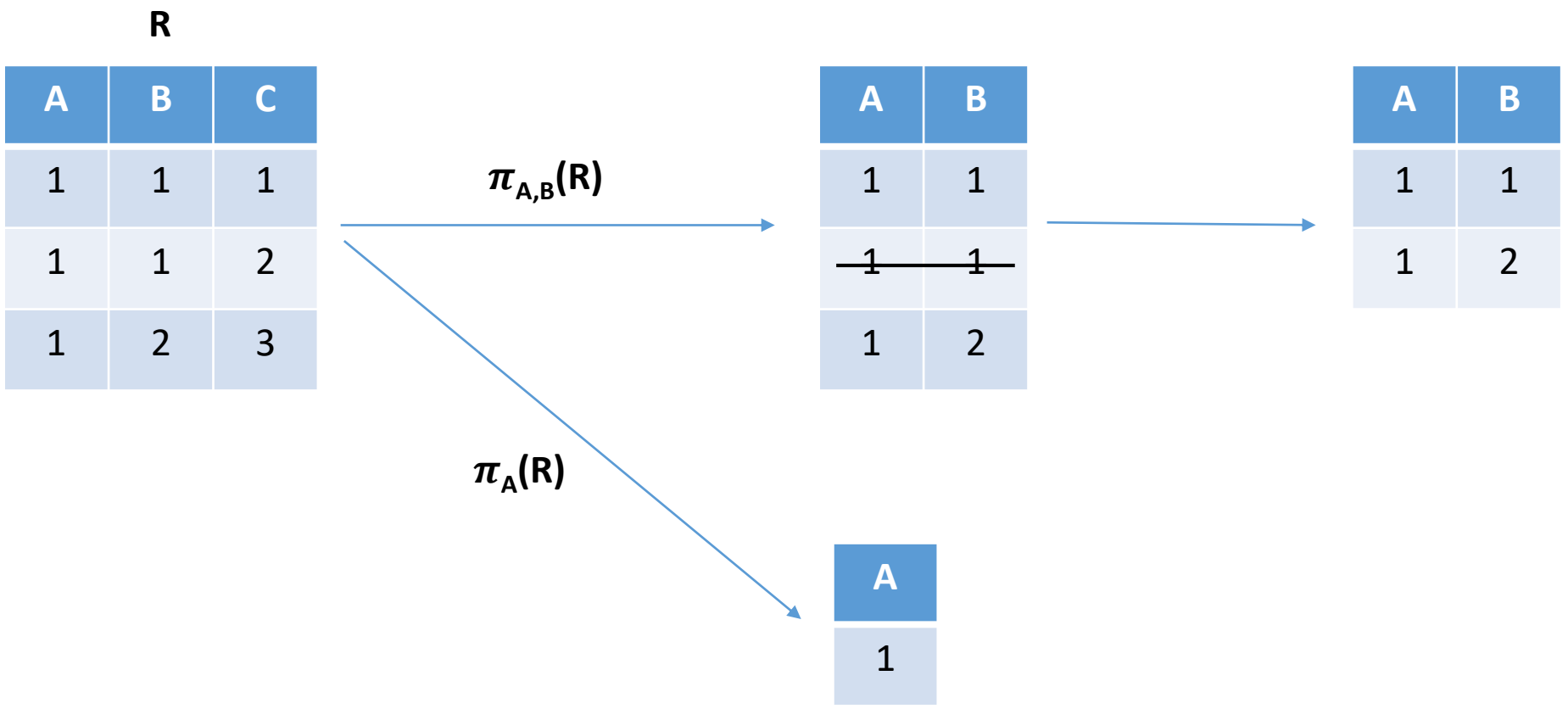
$$\pi_S(R)$$



- Works on a single relation R; returns a relation, which contains only those attributes (columns) that are defined by the list S.
- The projection eliminates duplicates.

Simple operations

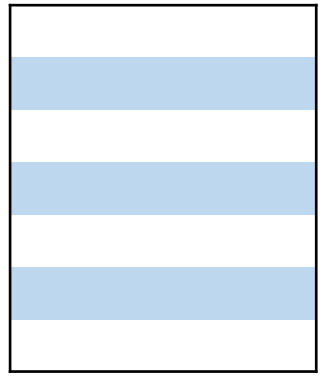
Projection (example)



Simple operations

Selection

$$\sigma_{\text{predicate}}(R)$$



- Works on a single relation R; returns a relation, which contains only those tuples (rows) from the relation R that satisfy a given condition (predicate)

Simple operations

Selection (example)

R

A	B	C
1	1	1
1	1	2
1	2	3

$\sigma_{A \neq C \wedge B<2}(R)$

A	B	C
1	1	2

Simple operations

Renaming

$$\rho_{s(S)}(R)$$

- Renaming of a relation R;
  - s is a new relation name,
  - S is a list of new attribute names.

Simple operations

Renaming (example)

R

A	B	C
1	1	1
1	1	2
1	2	3

S

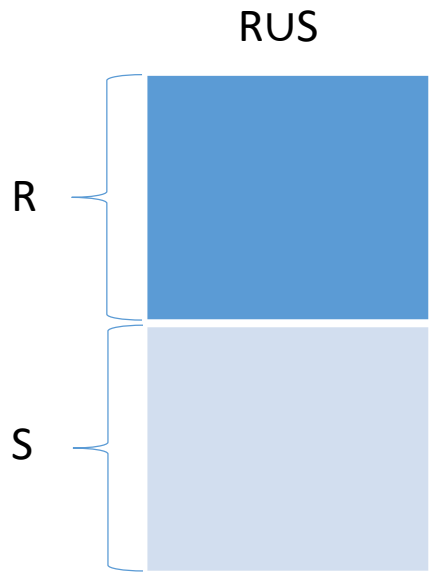
D	E	F
1	1	1
1	1	2
1	2	3

Set operations

Union

$$R \cup S$$

- Union of the relations R and S
- The condition for the execution of all set operations is that the relations are compatible with one another (the same number of attributes and that the same attributes have the same domains).



Set operations

Union (example)

R

A	B	C
1	2	3
4	5	6
7	8	9

S

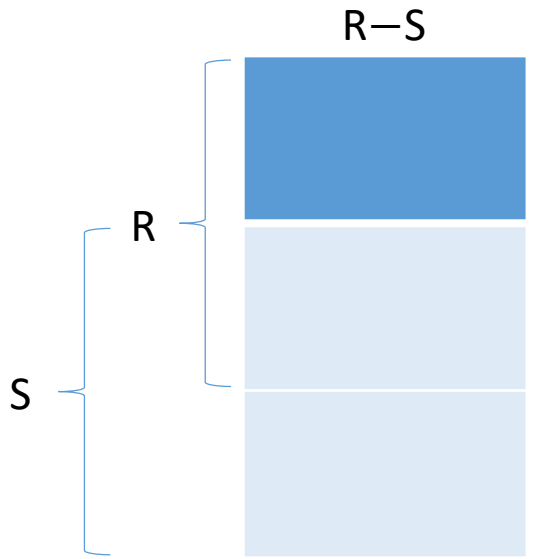
A	B	C
1	2	3
2	4	6
3	6	9

A	B	C
1	2	3
4	5	6
7	8	9
2	4	6
3	6	9

Set operations

Set-difference

$$\mathbf{R - S}$$



- Set-difference of the relations R and S.
- The condition for the execution of all set operations is that the relations are compatible with one another (the same number of attributes and that the same attributes have the same domains).

Set operations

Set-difference (example)

R

A	B	C
1	2	3
4	5	6
7	8	9

S

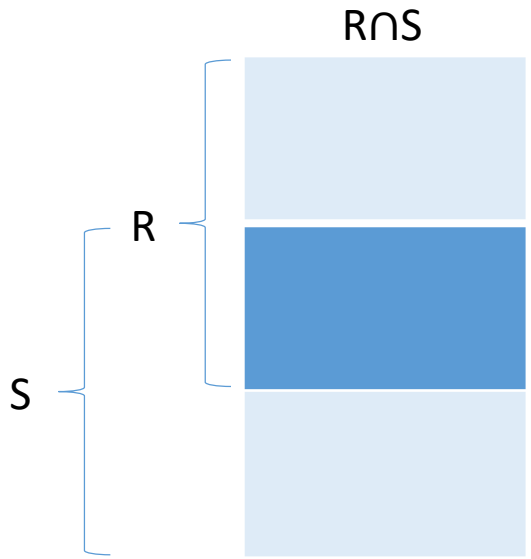
A	B	C
1	2	3
2	4	6
3	6	9

A	B	C
4	5	6
7	8	9

Set operations

Intersection

R∩S



- Intersection of the relations R and S.
  - Also: R ∩ S = R – (R – S)
- The condition for the execution of all set operations is that the relations are compatible with one another (the same number of attributes and that the same attributes have the same domains).

Set operations

Intersection (example)

R

A	B	C
1	2	3
4	5	6
7	8	9

S

A	B	C
1	2	3
2	4	6
3	6	9

A	B	C
1	2	3

Product operations

Cartesian product

R×S

R
a
b

S
1
2
3

=

R×S	
a	1
a	2
a	3
b	1
b	2
b	3

- Cartesian product of the relations R and S is a relation, which contains one tuple for each pair of tuples from the relations R and S.

Product operations

Cartesian product (example)

R

A	B	C
1	2	3
4	5	6
7	8	9

S

D	E
1	2
3	4

A	B	C	D	E
1	2	3	1	2
1	2	3	3	4
4	5	6	1	2
4	5	6	3	4
7	8	9	1	2
7	8	9	3	4

Product operations

Theta join

$$\mathbf{R} \bowtie_{\theta} \mathbf{S}$$

- $\boldsymbol{\theta}$-join of the relations R and S is a cartesian product where we keep only those tuples which satisfy the condition θ.

Product operations

Theta join (example)

R

A	B	C
1	2	3
4	5	6
7	8	9

S

D	E
1	2
3	4

$$R \bowtie_{C \geq D \wedge A = E} S$$

A	B	C	D	E
~~1~~	~~2~~	~~3~~	~~1~~	~~2~~
~~1~~	~~2~~	~~3~~	~~3~~	~~4~~
~~4~~	~~5~~	~~6~~	~~1~~	~~2~~
4	5	6	3	4
~~7~~	~~8~~	~~9~~	~~1~~	~~2~~
~~7~~	~~8~~	~~9~~	~~3~~	~~4~~

Product operations

Equijoin

$$R \bowtie_{\theta_=} S$$

- Equijoin of the relations R and S is a θ-join, where condition $\theta_=$ contains only equalities.

Product operations

Equijoin (example)

R

A	B	C
1	2	3
4	5	6
7	8	9

S

D	E
1	2
3	4

$R \bowtie_{A = E} S$

A	B	C	D	E
~~1~~	~~2~~	~~3~~	~~1~~	~~2~~
~~1~~	~~2~~	~~3~~	~~3~~	~~4~~
~~4~~	~~5~~	~~6~~	~~1~~	~~2~~
4	5	6	3	4
~~7~~	~~8~~	~~9~~	~~1~~	~~2~~
~~7~~	~~8~~	~~9~~	~~3~~	~~4~~

Product operations

Natural join

$$R \bowtie S$$

R

A	B
a	1
b	2

S

B	C
1	x
1	y
3	z

=

R ⋈ S

A	B	C
a	1	x
a	1	y

- Natural join of the relations R and S is an equijoin over all common attributes, where we keep only one occurrence of the common attributes (no duplicate attributes).
  - If relations R and S do not have common attributes, natural join equals cartesian join.

Product operations

Natural join (example)

R

A	B	C
1	2	3
4	5	6
7	8	9

S

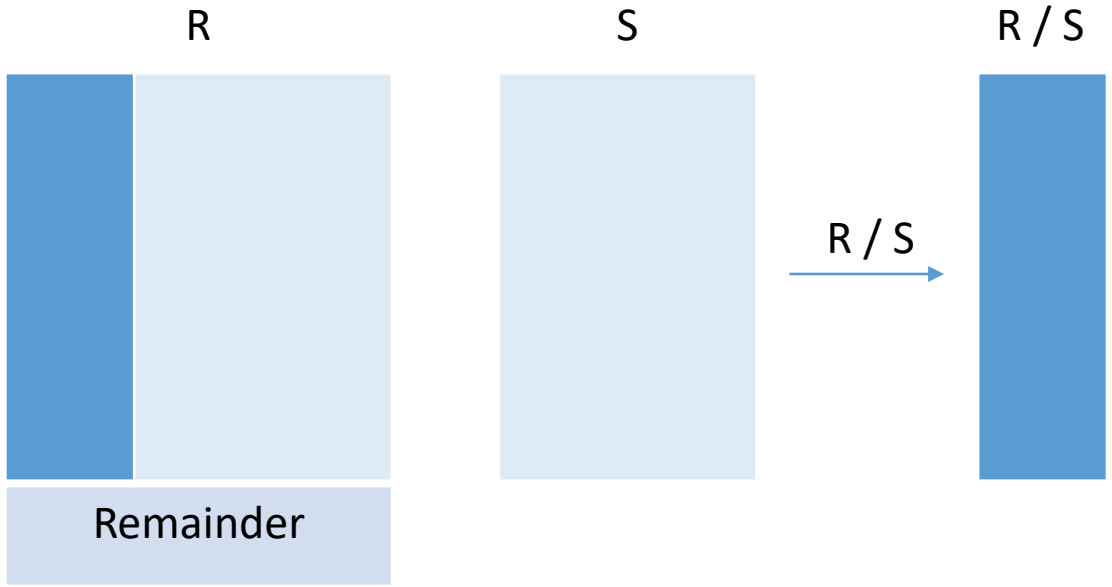
C	D	E
6	1	2
9	3	4

A	B	C	D	E
4	5	6	1	2
7	8	9	3	4

Product operations

Division

R / S



- The quotient of the relations R and S includes only those tuples which cover the relation S.
- The new scheme is obtained by subtracting the R and S schemes.

Product operations

Division (example)

R

A	B	C	D
1	1	2	1
1	3	4	1
2	3	4	1
3	1	2	1
4	1	2	1
4	3	4	2
5	5	6	1
6	7	8	1

S

B	C
1	2
3	4

R / S

A	D
1	1

Product operations

Division (example – Who has passed all the exams?**)**

passed

Student ID	Subject ID	...
10000	101	...
10001	101	...
10001	102	...
10002	101	...
10003	102	...
...	...	...

exams

Subject ID	...
101	...
102	...

$\pi_{\text{Student ID,Subject ID}}(\text{passed}) \,/\, \pi_{\text{Subject ID}}(\text{exams})$

Student ID
10001

Other operations

Semijoin, semi-θ-join

$$R \rhd S, \quad R \rhd_{\theta} S$$

- Semijoin of relations R and S is equal to natural join where we keep only the attributes of the left relation R.

Other operations

Semijoin (example)

R

A	B	C
1	2	3
4	5	6
7	8	9

S

C	D	E
6	1	2
9	3	4

$R \triangleright S$

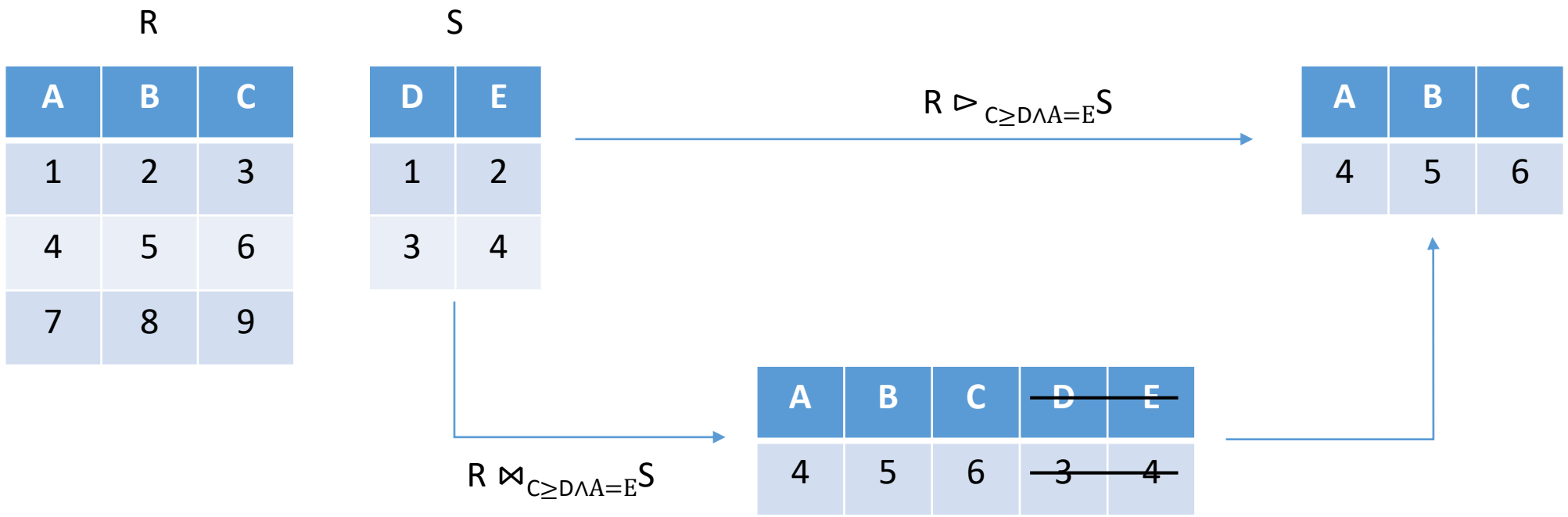
A	B	C
4	5	6
7	8	9

$R \bowtie S$

A	B	C	~~D~~	~~E~~
4	5	6	~~1~~	~~2~~
7	8	9	~~3~~	~~4~~

Other operations

Semi-θ-join (example)



Other operations

Outer joins (left, right and full)

$$R \mathbin{⟕} S, \quad R \mathbin{⟖} S, \quad R \mathbin{⟗} S$$

- Left outer join of R and S returns a natural join where tuples of R having no matching values in common attributes of S are also included in the result. Unknown values of the attributes are set to NULL.
  - Similarly goes for right outer join and full outer join.

Other operations

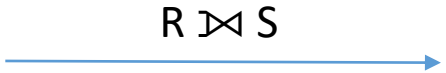
Left outer join (example)

R

A	B	C
1	2	3
4	5	6
7	8	9

S

C	D	E
6	1	2
9	3	4
12	5	6



A	B	C	D	E
1	2	3	NULL	NULL
4	5	6	1	2
7	8	9	3	4

- All tuples of R are included in the result.

Other operations

Right outer join (example)

R

A	B	C
1	2	3
4	5	6
7	8	9

S

C	D	E
6	1	2
9	3	4
12	5	6

$R \bowtie S$

A	B	C	D	E
4	5	6	1	2
7	8	9	3	4
NULL	NULL	12	5	6

- All tuples of S are included in the result.

Other operations

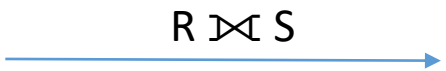
Full outer join (example)

R

A	B	C
1	2	3
4	5	6
7	8	9

S

C	D	E
6	1	2
9	3	4
12	5	6



A	B	C	D	E
1	2	3	NULL	NULL
4	5	6	1	2
7	8	9	3	4
NULL	NULL	12	5	6

- All tuples from R and S are included in the result.

Other operations

Aggregate

$$\tau_{AS}(R)$$

- Applies aggregate function list AS to the relation R.
- Aggregation functions are:
  - COUNT – counts all non-NULL values
  - SUM – sums the values
  - AVG – computes the average
  - MIN – finds the lowest number
  - MAX – finds the highest number

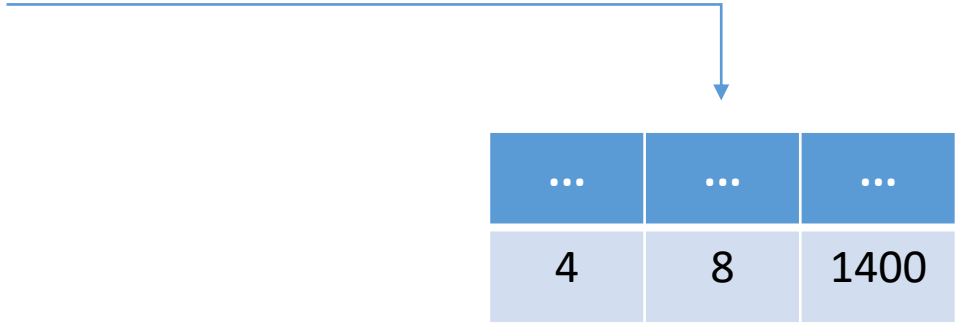
Other operations

Aggregate (example)

R

A	B	C	D
1	a	100	1
1	a	200	1
1	b	200	1
2	a	100	1
2	b	100	1
3	a	100	1
3	NULL	100	1
4	b	100	1
4	b	200	1
4	NULL	200	1

$\tau_{\text{MAX A, COUNT B, SUM C}}(R)$



Other operations

Grouping

$$\mathbf{GA}\tau_{\mathbf{AS}}(\mathbf{R})$$

- Aggregation with grouping of relation R uses a list of aggregation functions AS upon groups of relation R which are determined with a list of grouping attributes.

Other operations

Grouping (example)

R

A	B	C	D
1	a	100	1
1	a	200	1
1	b	200	1
2	a	100	1
2	b	100	1
3	a	100	1
3	NULL	100	1
4	b	100	1
4	b	200	1
4	NULL	200	1

$A,D\ \tau_{\text{COUNT B, SUM C}}(R)$

A	D	…	…
1	1	3	500
2	1	2	200
3	1	1	200
4	1	2	500

Priority of operations

1. **Simple operations:**
   projection, selection, renaming

2. **Product operations**:
   cartesian product, θ-join, natural join, division

3. **Set operations**:
   intersection, union, set-difference

4. **Other operations**:
   aggregate (with grouping)

Pictorial Representation of Equivalences

RELATIONAL ALGEBRA EQUIVALENC

```
SELECT s.name, e.cid
  FROM student AS s, enrolled AS e
 WHERE s.sid = e.sid
   AND e.grade = 'A'
```

$$\pi_{name,\ cid}(\sigma_{grade='A'}(student \bowtie enrolled))$$

$$=$$

$$\pi_{name,\ cid}(student \bowtie (\sigma_{grade='A'}(enrolled)))$$

Relational algebra exercises

Exercise 1 - Operator

Write the following queries in a relational algebra.

1. Which customers are buying telephones at Telekom?
2. Which operators are selling Janez‘s prefered phone?
3. Which customers are buying from all operators? Assume that all the operators are listed in relation p.
4. Which operators are selling all Janez‘s prefered telephones? Assume that relation n contains more tuples which correspond to Janez.
5. Which customers are buying from one operator only?

p (sale)

Operator	Telephone
Telekom	Nokia
Telekom	Siemens
T2	Nokia
T2	Samsung
A1	Nokia
A1	Siemens
A1	Samsung

k (buying)

Customer	Operator
Marko	Telekom
Marko	A1
Meta	Telekom
Meta	T2
Meta	A1
Janez	A1
Petra	Telekom

n (prefers)

Customer	Telephone
Marko	Siemens
Meta	Nokia
Janez	Siemens
Petra	Nokia

1. $\Pi_{customer}(\sigma_{operator='Telekom'}(K))$

2. $K / \Pi_{operator}(P)$

3. $\Pi_{customer,operator} / \Pi_{operator}(P \bowtie K)$

4. $P / \Pi_{operator}(\sigma_{customer='Janez'}(N))$

5.

(5)

Exercise 2 - GSM

Relation	Relation schema
s customer	CUSTOMER (SID, SName, SSurname, SAge, SCity)
p seller	SELLER (PID, PName, PSurname, PAge, PDiscount)
g mobile phone	GSM (GID, GType, GPrice)
k purchase	PURCHASE (#SID, #PID, #GID, KDate, KPieces)

Write the following queries in a relational algebra.

1. Find names and surnames of customers who are from Kranj and are older than 18 years.
2. Find names and surnames of customers who have bought something.
3. Find names and surnames of customers who have never bought anything. Assume that the customers are stored in a relation CUSTOMER.
4. Find names and surnames of the sellers who have untill now sold only Nokia mobile phones.
5. For each type of mobile phone find the total number of sold pieces.

1. $\Pi_{name, surname}(\sigma_{scity='Kranj' \wedge \sigma_{age>18}}(S))$
2. $\Pi_{name, surname}(S \bowtie K)$
3. $\Pi_{name, surname}(S) - \Pi_{name, surname}(K \bowtie S)$ → find names of all customers and subtract those who have made a purchase
4. $\Pi_{pname, psurname}(\sigma_{gtype='Nokia'}(P \bowtie G \bowtie K)) - \Pi_{name, surname}(\sigma_{gtype \neq 'Nokia'}(P \bowtie G \bowtie K))$ → nađi prodavce koji su prodali Nokiu i oduzmi one koji nisu prodali Nokiu

???

4,5

Exercise 3 - Hotel

Relation	Relation schema
g guest	GUEST (GNo, GName, GAddress)
h hotel	HOTEL (HNo, HName, HCity)
r room	ROOM (RNo, #HNo, RType, RPrice)
b booking	BOOKING (#HNo, #RNo, #GNo, BFrom, BTo)

Write the following queries in a relational algebra.

1. Find the numbers of all single-bed rooms (Rtype = 1), where the price is lower than 80€ per day.
2. Find the numbers, prices and room types in hotel Bernardin.
3. Find names of the guests which are currently in hotel Bernardin (value today). Show also the prices and types of the rooms they are situated in.
4. For each hotel list its name, number of all rooms and an average room price.
5. List all the room data of hotel Bernardin (RNo, RType and RPrice), including the name of the guest in the room if it is currently occupied (otherwise value NULL).

1. $\Pi_{RNo}(\sigma_{Rtype=1 \wedge RPRICE<80€}(R))$
2. $\Pi_{RNO, RPrice, RType}(\sigma_{HName='Bernardin'}(H \bowtie R))$
3. $\Pi_{gname, RPrice, Rtype}((\sigma_{HName='Bernardin'}(H)) \bowtie (\sigma_{BFrom \leq today \wedge to > today}(B \bowtie R \bowtie G)))$
4. Π_{HName}

4. Π_{Hname}
5. $\Pi_{RNo, Rtype, RPrice, gname}(R \bowtie (\sigma_{Hname='Bernardin'(H))} \times$

Introduction to database systems

Relational calculus

Intro

Calculus → logic

Relational calculus is nonprocedural or declarative query language.

- With relational algebra we define the procedure how the answers should be computes. **Relational calculus** allows us to **describe the set of answers** without being explicit how they should be computed.

There are two types of relational calculus:

- **Tuple** Relational Calculus
  - Variables take on tuples as values.
- **Domain** Relational Calculus
  - The variables range over field values

Domain Relational Calculus

- A DRC query has the form:

  $\{ \langle d1, d2, \ldots, dn \rangle \mid F(\langle d1, d2, \ldots, dn \rangle) \}$

  domain variables — formula

  Where:

  $\langle d1, d2, \ldots, dn \rangle$ set of domain variables or constants and

  $F(\langle d1, d2, \ldots, dn \rangle)$ DRC formula (The result of this query is the set of all tuples $(d1, d2, \ldots, dn)$ for which the formula evaluates to true.)

Quantifiers

- Using quantifiers we determine how many examples the predicate refers to.
- There are two quantifiers:
  - **Existential** quantifier ∃ (we read: „there exists (at least one)“)
  - **Universal** quantifier ∀ (we read: „for all“)

Example

Names of suppliers, which supply a red part.

Relation	Relational schema
Suppliers	SUPPLIERS(sid, sname, address)
Parts	PARTS(pid, pname, color)
Catalog	CATALOG(#sid, #pid, cost)

Using relational algebra and relational calculus, find:

- Names of suppliers, which supply a red part.

RA:

$$\pi_{sname}(\pi_{sid}((\pi_{pid}\,\sigma_{color\ =\ 'red'}Parts) \bowtie Catalog) \bowtie Suppliers)$$

DRC:

$$\{\langle Y\rangle \mid \langle X,Y,Z\rangle \in Suppliers \wedge \exists P,Q,R(\langle P,Q,R\rangle \in Parts \wedge R =' red' \wedge \exists I,J,K(\langle I,J,K\rangle \in Catalog \wedge J = P \wedge I = X))\}$$

OR

$$\{\langle Y\rangle \mid \langle X,Y,Z\rangle \in Suppliers \wedge \exists\langle P,Q,R\rangle \in Parts(R =' red' \wedge \exists\langle I,J,K\rangle \in Catalog(J = P \wedge I = X))\}$$

↳ use shorter form

Exercises

Relational algebra and relational calculus

TASK 1: Suppliers

Using **relational algebra** and **relational calculus**, find:

1. Find the *sids* of suppliers who supply some red or green part.
2. Find the *sids* of suppliers who supply some red part or are at „Koprska cesta 25“.
3. Find the *sids* of suppliers who supply some red part and some green part.
4. Find the *sids* of suppliers who supply every part.
5. Find the *sids* of suppliers who supply every red part.
6. Find the *sids* of suppliers who supply every red or green part.
7. Find the *pids* of parts supplied by at least two different suppliers.

same

Relation	Relational schema
Suppliers = S	SUPPLIERS(<u>sid</u>, sname, address)
Parts = P	PARTS(<u>pid</u>, pname, color)
Catalog = C	CATALOG(<u>#sid</u>, <u>#pid</u>, cost)

EXERCISE 1

1. RA: $\Pi_{sid}(\sigma_{color='red' \vee color='green'}(C \bowtie P))$ → we need catalog to interconnect them (catalog has sid & pid)
   RC: $\{<s> \mid <s,p,c> \in catalog \wedge \exists <p1,pn1,c1> \in PARTS\ (p=p1 \wedge (c1='red' \vee c1='green'))\}$

2. RA: $\Pi_{sid}(\sigma_{color='red'}(P \bowtie C)) \cup \Pi_{sid}(\sigma_{address='koperska\ cesta\ 25'}(S))$
   RC: $\{<s> \mid <s,p,c> \in catalog \wedge \exists <p1,pn1,c1> \in PARTS\ (p=p1 \wedge c1='red') \vee \exists <s2,sn2,a2> \in supplies\ (a2='koperska\ cesta\ 25')\}$

3. RA: $\Pi_{sid}(\sigma_{color='red'}(P \bowtie C)) \cap \Pi_{sid}(\sigma_{color='green'}(P \bowtie C))$
   RC: $\{<s> \mid <s,p,c> \in catalog \wedge \exists <p1,pn1,c1> \in PARTS\ (p=p1 \wedge c1='red') \wedge \exists <s2,p2,c2> \in catalog\ (\exists <p3,pn3,c3> \in PARTS\ (p2=p3 \wedge c3='green'))\}$

4. RA: $\Pi_{sid,pid}(C) / \Pi_{pid}(P)$ → when we divide pid by pid we get sid as a result
   RC: $\{<s> \mid <s,p,c> \in catalog \wedge \forall <p1,pn1,c1> \in PARTS\ (\exists <s2,p2,c2> \in catalog\ (p1=p2 \wedge s=s2))\}$

5. RA: $\Pi_{sid,pid}(C) / \Pi_{pid}(\sigma_{color='red'}(P))$
   RC: $\{<s> \mid <s,p,c> \in catalog \wedge \forall <p1,pn1,c1> \in PARTS\ (c1='red' \wedge \exists <s2,p2,c2> \in catalog\ (p1=p2 \wedge s=s2))\}$ → we can add c1='red' at the end brackets too

6. RA: $\Pi_{sid,pid}(C) / \Pi_{pid}(\sigma_{color='red' \vee color='green'}(P))$
   RC: $\{<s> \mid <s,p,c> \in catalog \wedge \forall <p1,pn1,c1> \in PARTS\ ((c1='red \vee c1='green' \wedge \exists <s2,p2,c2> \in catalog\ (p1=p2 \wedge s=s2))\}$

7. RA: $\rho(c1, catalog)\ \rho(c2, catalog)\ \Pi_{c1.pid}((\sigma_{c1.pid=c2.pid \wedge c1.sid \neq c2.sid})(c1 \times c2))$ (1st copy, 2nd copy; the same part, different supplier) → perform a cartesian product; compare that it goes for the same part and different supplier
   RC: $\{<s,p,c> \in catalog \wedge \exists <s1,p1,c1> \in catalog\ (p=p1 \wedge s \neq s1)\}$

TASK 2: Airline flight information

Using **relational algebra** and **relational calculus**, find:

1. Find the *eids* of pilots certified for some Boeing aircraft.
2. Find the names of pilots certified for some Boeing aircraft.
3. Find the *aids* of all aircraft that can be used on non-stop flights from Paris to Vancouver.
4. Find the names of pilots who can operate planes with a range greater than 3,000 miles but are not certified on any Boeing aircraft.
5. Find the *eids* of employees who make the highest salary.

Relation	Relational schema
Flights = F	FLIGHTS(flno, from, to, distance, departs, arrives)
Aircraft = A	AIRCRAFT(aid, aname, cruisingrange)
Certified = C	CERTIFIED(#eid, #aid)
Employees = E	EMPLOYEES(eid, ename, salary)

EXERCISE 2

1. RA: $\Pi_{eid}\ (\sigma_{aname='Boeing'}\ (A \bowtie C)$

   RC: $\{<E> \mid <E,A> \in certified \wedge \exists <A1,AN1,CR1> \in aircraft\ (A=A1 \wedge AN1='Boeing')\}$

2. RA: $\Pi_{name}(\sigma_{aname='Boeing'}(E \bowtie C \bowtie A))$

   RC: $\{<EN> \mid <E,EN,S> \in employees \wedge \exists <E1,A1> \in certified\ (E=E1 \wedge \exists <A2,AN2,CR2> \in aircraft\ (A1=A2 \wedge AN2='Boeing'))\}$

3. RA: $\int(F1,\ \sigma_{from='Paris' \wedge to='Vancouver'}(F))\ \ \Pi_{A.aid}(\sigma_{A.cruisingrange > F1.distance}(A \times F1))$

   RC: $\{<A> \mid <A,AN,CR> \in AIRCRAFT \wedge \exists <F1,FR1,TO1,D1,DE1,AR1> \in flights\ (CR>D1)\}$

4. RA: $\Pi_{ename}(E \bowtie (\Pi_{eid}(\sigma_{cruisingrange>3000}(A \bowtie C)) - \Pi_{eid}(\sigma_{aname='Boeing'}(A \bowtie C))))$

   (set difference)

   (using negation)

   RC: $\{<EN> \mid <E,EN,S> \in employees \wedge \exists <E1,A1> \in CERTIFIED\ (E=E1 \wedge \exists <A2,AN2,CR2> \in aircraft\ (A1=A2 \wedge CR2>3000)) \wedge \neg(\exists <E3,EN3,S3> \in employees\ (\exists <E4,A4> \in certified\ (E3=E4 \wedge \exists <A5,AN5,CR5> \in aircraft \wedge A4=A5 \wedge AN5='Boeing' \wedge E=E3)))\}$

5. RA: $\int(E1, employees)\ \int(E2, employees)\ \ \Pi_{employees.eid} - \Pi_{E2.eid}(E1 \bowtie_{E1.salary>E2.salary} E2)$

   RC: $\{<E> \mid <E,EN,S> \in employees \wedge \neg(\exists <E1,EN1,S1> \in employees\ (S<S1))\}$

TASK 3: Oil company

Relation	Relational schema
t tank	TANK (IdC, serialNo, capacity, dtService)
o orders	ORDERS (IdN, no, date, quantity)
d delivery	DELIVERY (#IdN, #IdC, date, quantity)

Relation DELIVERY contains information about the deliveries of supplies (or tank transports) to fulfill individual orders. In oder to fulfill one order, you may also need more deliveries of supplies.

Write the following queries using the **relational calculus**.

1. Find all order numbers where a tank with a serial number 123456 participated in the fuel delivery.
2. Find all *IdN* of orders, delivered by the tank with serial number 123456 or 123457.
3. Find all *IdN* of orders, which either require a quantity of 10,000 units or have been delivered by a tank with a serial number 123456.
4. Find all *IdN* of orders, which were delivered by a tank with a serial number 123456 and a tank with a serial number 123457.

EXERCISE 3

Notebook

Introducion to Databases

Exercises: SQL

Nested SELECT

- The **result** of a SELECT statement may be used **as a value** in another statement.
- For example the statement:

```
SELECT continent
FROM world
WHERE name = 'Brazil'
```

evaluates to 'South America' so we can **use this value** to obtain a list of all countries in the same continent as 'Brazil':

```
SELECT name
FROM world
WHERE continent = (SELECT continent
                   FROM world
                   WHERE name = 'Brazil')
```

Nested SELECT – multiple results

- The subquery may return **more than one result** - if this happens the query above will fail as you are testing one value against more than one value.
- It is safer to use **IN** to cope with this possibility.
- The phrase:

```
SELECT continent
FROM world
WHERE name = 'Brazil' OR name = 'Mexico'
```

will return two values ('North America' and 'South America'). You should use:

```
SELECT name, continent
FROM world
WHERE continent IN (SELECT continent
                    FROM world
                    WHERE name = 'Brazil' OR name = 'Mexico')
```

Subquery on the SELECT line

- If you are certain that **only one value** will be returned you can use that query on the SELECT line:

```
SELECT population/(SELECT population
                   FROM world
                   WHERE name='United Kingdom')
FROM world
WHERE name = 'China'
```

Operators over a set

- These operators are *binary* - they normally take two parameters:

```
=     equals
>     greater than
<     less than
>=    greater or equal
<=    less or equal
```

You can use the words **ALL** or **ANY** where the right side of the operator might have multiple values.

Example:

Show each country that has a population greater than the population of ALL countries in Europe. (Note that we mean greater than every single country in Europe; not the combined population.)

```
SELECT name
FROM world
WHERE population > ALL (  SELECT population
                          FROM world
                          WHERE continent='Europe')
```

JOIN

- We can combine attibutes from more than one table, to get the results.
- Example:

```
SELECT *
FROM game JOIN goal ON (id=matchid)
```

The **`FROM`** clause says to **merge data** from the **goal table** with that from the **game table**.

The **`ON`** says **how to figure out which rows in game go with which rows in goal** - the **`matchid`** from goal must match **`id`** from game.

(If we wanted to be more clear/specific we could say `ON (game.id=goal.matchid)`)

JOIN

Instead of using ON, we can also match the attributes in WHERE clause:

```
SELECT player, teamid, stadium, mdate
FROM game, goal
WHERE teamid = 'GER' AND game.id=goal.matched
```

Gives the same results as:

```
SELECT player, teamid, stadium, mdate
FROM game JOIN goal ON (id=matchid)
WHERE teamid='GER'
```

CASE

- CASE allows you to **return different values under different conditions**.
- If there no conditions match (and there is not ELSE) then NULL is returned.

```
CASE   WHEN condition1 THEN value1
       WHEN condition2 THEN value2
       ELSE def_value
END
```

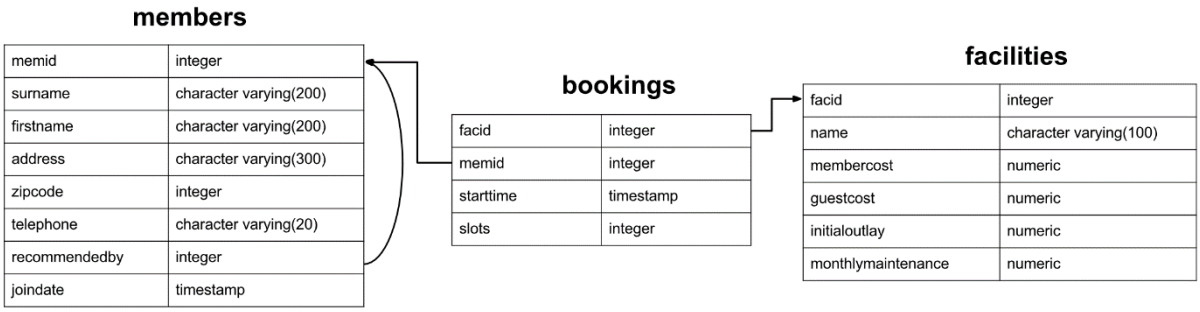
```
SELECT name, population,
       CASE   WHEN population<1000000    THEN 'small'
              WHEN population<10000000   THEN 'medium'
              ELSE 'large'
       END
FROM bbc
```

Exercises

http://sqlzoo.net/wiki/SELECT_within_SELECT_Tutorial

SQL solutions

Exercises – SQL 2



members

memid	integer
surname	character varying(200)
firstname	character varying(200)
address	character varying(300)
zipcode	integer
telephone	character varying(20)
recommendedby	integer
joindate	timestamp

bookings

facid	integer
memid	integer
starttime	timestamp
slots	integer

facilities

facid	integer
name	character varying(100)
membercost	numeric
guestcost	numeric
initialoutlay	numeric
monthlymaintenance	numeric

1. **Find surnames of all members who have made at least one booking.**

for distinct surnames

```
SELECT DISTINCT mem.surname
FROM cd.members mem, cd.bookings book
WHERE mem.memid = book.memid;
```

→ Natural join = Inner Join

surname
Worthington-Smyth
Joplette
Stibbons
Pinker
Dare
Rumney
Bader
Mackenzie
Sarwin
Boothe
Crumpet
Farrell
Tracy
Hunt
Owen
Tupperware
Smith
Butters
Rownam
Baker
GUEST
Genting
Jones
Purview
Coplin

2. **Find surnames of all memebers, who have made more than 100 bookings.**

```
SELECT mem.surname, count(*)
FROM cd.members mem, cd.bookings book
WHERE mem.memid = book.memid
GROUP BY mem.surname
HAVING COUNT(*) > 100;
```

surname	count
Joplette	159
Stibbons	104
Dare	117
Bader	120
Mackenzie	126
Boothe	188
Farrell	114
Tracy	176
Owen	131
Smith	560
Butters	164
Rownam	408
Baker	284
GUEST	883
Jones	129

3. **Find memid of all members, who have booked either “Squash Court” or “Pool Table”.**

```
SELECT DISTINCT book.memid
FROM cd.facilities fac, cd.bookings book
WHERE fac.facid = book.facid and
        (fac.name = 'Squash Court' OR fac.name = 'Pool Table');
```

memid
14
27
8
12
17
28
1
15
10
26
11
4
30
0
16
33
6
29
2
21
3
35
20
5
13
22
9
7
24

4. **Find memid of all members, who have booked “Squash Court” or their surname begins with the letter ‘R’.**

```
SELECT DISTINCT book.memid
FROM cd.facilities fac, cd.bookings book
WHERE fac.facid = book.facid AND (fac.name = 'Squash Court' )
UNION
SELECT mem.memid
FROM cd.members mem
WHERE surname LIKE 'R%' ;
```

here we're including members who haven't booked anything at all

memid
14
27
8
12
17
28
1
15
10
26
11
4
30
0
16
33
6
2
21
3
35
5
13
22
9
24

5. **Find memid of all members, who have booked “Squash Court” and “Pool Table”.**

```
SELECT DISTINCT book.memid
FROM cd.facilities fac, cd.bookings book
WHERE fac.facid = book.facid AND (fac.name = 'Squash Court')
INTERSECT
SELECT DISTINCT book.memid
FROM cd.facilities fac, cd.bookings book
WHERE fac.facid = book.facid AND (fac.name = 'Pool Table');
```

memid
14
27
8
12
17
28
1
15
10
26
11
4
30
0
16
33
6
2
21
5
13
22
9
24

6. **Upgrade the solution of the previous task (Task 5) by displaying the names and surnames of all members, who have booked “Squash Court” and “Pool Table”.** ***HINT: Nested SELECT***

```
SELECT mem.firstname, mem.surname
FROM cd.members mem
WHERE mem.memid IN (
        SELECT DISTINCT book.memid
        FROM cd.facilities fac, cd.bookings book
        WHERE fac.facid = book.facid AND (fac.name = 'Squash Court')
        INTERSECT
        SELECT DISTINCT book.memid
        FROM cd.facilities fac, cd.bookings book
        WHERE fac.facid = book.facid AND (fac.name = 'Pool Table')
);
```

firstname	surname
Jack	Smith
Henrietta	Rumney
Tim	Boothe
Anne	Baker
David	Pinker
David	Farrell
Darren	Smith
Florence	Bader
Charles	Owen
Douglas	Jones
David	Jones
Janice	Joplette
Millicent	Purview
GUEST	GUEST
Timothy	Baker
Hyacinth	Tupperware
Burton	Tracy
Tracy	Smith
Anna	Mackenzie
Gerald	Butters
Jemima	Farrell
Joan	Coplin
Ponder	Stibbons
Ramnaresh	Sarwin

7. **Find surnames of all members, who have joined after the member whose surname is “Owen”.**

```
SELECT DISTINCT mem.surname, mem.joindate
FROM  cd.members mem
WHERE  mem.joindate > ANY (SELECT mem.joindate
                           FROM cd.members mem
                           WHERE mem.surname = 'Owen'
) ;
```

surname	joindate
Tupperware	2012-09-18 19:32:05
Purview	2012-09-18 19:04:01
Hunt	2012-09-19 11:32:45
Jones	2012-09-02 18:43:05
Baker	2012-08-10 14:23:22
Bader	2012-08-10 17:52:03
Mackenzie	2012-08-26 09:32:05
Crumpet	2012-09-22 08:36:38
Jones	2012-08-06 16:32:55
Farrell	2012-08-10 14:28:01
Sarwin	2012-09-01 08:44:42
Farrell	2012-09-15 08:22:05
Rumney	2012-09-05 08:42:35
Pinker	2012-08-16 11:32:47
Smith	2012-08-10 16:22:05
Baker	2012-08-15 10:34:25
Genting	2012-08-19 14:55:55
Coplin	2012-08-29 08:32:41
Worthington-Smyth	2012-09-17 12:27:15
Smith	2012-09-26 18:08:45

8. **Find names and surnames of all members, who have booked one of the tennis courts (»Tennis Court 1«, »Tennis Court 2« …).**

```
SELECT DISTINCT mem.firstname, mem.surname, fac.name
FROM cd.members mem, cd.bookings book, cd.facilities fac
WHERE name LIKE 'Tennis%' AND
        mem.memid = book.memid AND
        book.facid=fac.facid;
```

firstname	surname	name
Ramnaresh	Sarwin	Tennis Court 2
Charles	Owen	Tennis Court 1
John	Hunt	Tennis Court 1
Florence	Bader	Tennis Court 2
Jemima	Farrell	Tennis Court 1
Ponder	Stibbons	Tennis Court 2
Tim	Boothe	Tennis Court 1
Burton	Tracy	Tennis Court 2
David	Farrell	Tennis Court 1
Erica	Crumpet	Tennis Court 1
Tim	Rownam	Tennis Court 2
Douglas	Jones	Tennis Court 1
GUEST	GUEST	Tennis Court 2
Janice	Joplette	Tennis Court 2
Charles	Owen	Tennis Court 2
Ramnaresh	Sarwin	Tennis Court 1
Nancy	Dare	Tennis Court 2
Timothy	Baker	Tennis Court 2
Ponder	Stibbons	Tennis Court 1
Matthew	Genting	Tennis Court 1
Jemima	Farrell	Tennis Court 2
Florence	Bader	Tennis Court 1
Jack	Smith	Tennis Court 2
John	Hunt	Tennis Court 2
Gerald	Butters	Tennis Court 2

9. **List the facility numbers (facid) that have the highest monthly maintenance**

```
SELECT fac.facid, fac.monthlymaintenance
FROM cd.facilities fac
WHERE fac.monthlymaintenance = (
                        SELECT MAX(fac1.monthlymaintenance)
                        FROM cd.facilities fac1);
```

facid	monthlymaintenanc
4	300
5	300

10. List the facility numbers (facid) that have the second highest monthly maintenance

```
SELECT fac.facid, fac.monthlymaintenance
FROM cd.facilities fac
WHERE fac.monthlymaintenance =
(SELECT MAX(fac1.monthlymaintenance)
 FROM cd.facilities fac1
 WHERE fac1.monthlymaintenance <> (SELECT
                                   MAX(fac2.monthlymaintenance)
                                   FROM cd.facilities fac2));
```

facid	monthlymaintenanc
0	20
1	20

11. List the names of all members who have reserved 6 slots.

```
SELECT DISTINCT mem.firstname
FROM cd.members mem
WHERE EXISTS (
                SELECT book.memid
                FROM cd.bookings book
                WHERE slots = 6 AND mem.memid = book.memid
);
```

firstname
Henry
Ramnaresh
Charles
David
Timothy
Anne
Darren
Gerald
Tracy
Anna
Ponder
Tim
GUEST
Jack

12. List the name and surname of all visitors who have booked Badminton court more times than any person named "Timothy".

```
SELECT mem.firstname, mem.surname, COUNT(*)
FROM cd.members mem, cd.bookings book, cd.facilities fac
WHERE mem.memid = book.memid AND book.facid = fac.facid AND
fac.name = 'Badminton Court'
GROUP BY mem.memid
HAVING COUNT(*) > ALL (
                SELECT COUNT(*)
                FROM cd.members mem,
                        cd.bookings book,
                        cd.facilities fac
                WHERE mem.memid = book.memid
                                AND book.facid = fac.facid
                                AND fac.name = 'Badminton Court'
                                AND mem.firstname = 'Timothy'
);
```

firstname	surname	count
Jack	Smith	12
Tim	Boothe	12
Anne	Baker	10
Darren	Smith	132
Florence	Bader	9
David	Jones	8
GUEST	GUEST	39
Tracy	Smith	32
Anna	Mackenzie	30
Gerald	Butters	20
Ponder	Stibbons	16
Nancy	Dare	10

13. List the name and surname of all visitors who have booked Badminton court more times than any person named "Timothy").

Solve in alternative way (use "INNER JOIN" or do it in the following way "R1.id = R2.id")

```
SELECT mem.firstname, mem.surname, COUNT(*)
FROM  cd.bookings book
INNER JOIN cd.members mem ON book.memid = mem.memid
INNER JOIN cd.facilities fac ON book.facid = fac.facid
WHERE fac.name = 'Badminton Court'
GROUP BY mem.memid
HAVING COUNT(*) > ALL (
            SELECT COUNT(*)
            FROM cd.bookings book1
                INNER JOIN cd.members mem1 ON book1.memid =
mem1.memid
                INNER JOIN cd.facilities fac1 ON book1.facid = fac1.facid
            WHERE  fac1.name = 'Badminton Court' AND
                        mem1.firstname = 'Timothy'
);
```

firstname	surname	count
Jack	Smith	12
Tim	Boothe	12
Anne	Baker	10
Darren	Smith	132
Florence	Bader	9
David	Jones	8
GUEST	GUEST	39
Tracy	Smith	32
Anna	Mackenzie	30
Gerald	Butters	20
Ponder	Stibbons	16
Nancy	Dare	10

14. List the names of the facilities that charge the highest price to guests.

```
SELECT fac.name, fac.guestcost
FROM cd.facilities fac
WHERE fac.guestcost = (
                SELECT MAX(fac1.guestcost)
                FROM cd.facilities fac1
);
```

name	guestcost
Massage Room 1	80
Massage Room 2	80

15. List start times of all bookings made by 'David Farrell' (Use INNER JOIN)

```
SELECT book.starttime
FROM cd.bookings book
            INNER JOIN cd.members mem ON mem.memid = book.memid
WHERE mem.firstname = 'Darren'  AND mem.surname = 'Farrell' ;
```

starttime
2012-07-03 11:00:00
2012-07-03 08:00:00
2012-07-03 19:00:00
2012-07-03 10:00:00
2012-07-03 15:00:00
2012-07-04 15:30:00
2012-07-05 09:30:00
2012-07-05 19:00:00
2012-07-05 14:30:00
2012-07-06 17:00:00
2012-07-06 11:00:00
2012-07-06 14:00:00
2012-07-07 09:00:00
2012-07-07 11:30:00
2012-07-07 16:00:00
2012-07-07 10:30:00
2012-07-07 14:30:00
2012-07-08 15:00:00
2012-07-08 17:30:00
2012-07-08 11:30:00
2012-07-08 19:30:00
2012-07-08 16:30:00
2012-07-09 19:00:00
2012-07-09 09:00:00
2012-07-09 15:30:00
2012-07-10 11:00:00
2012-07-10 18:00:00
2012-07-11 08:00:00
2012-07-12 11:30:00
2012-07-12 09:00:00
2012-07-12 18:30:00

16. List start times (“starttime”) of all bookings for facilities with IDs 0 and 1, for a date '2012-09-21'? Return a list of start times (starttime and object names). The list should be ordered by “starttime”.

```
SELECT book.starttime as START, fac.name as NAME
FROM cd.facilities fac
            INNER JOIN cd.bookings book ON fac.facid = book.facid
WHERE fac.facid in (0,1) AND
            book.starttime >= '2012-09-21' AND
            book.starttime < '2012-09-22'
ORDER BY book.starttime;
```

start	name
2012-09-21 08:00:00	Tennis Court 1
2012-09-21 08:00:00	Tennis Court 2
2012-09-21 09:30:00	Tennis Court 1
2012-09-21 10:00:00	Tennis Court 2
2012-09-21 11:30:00	Tennis Court 2
2012-09-21 12:00:00	Tennis Court 1
2012-09-21 13:30:00	Tennis Court 1
2012-09-21 14:00:00	Tennis Court 2
2012-09-21 15:30:00	Tennis Court 1
2012-09-21 16:00:00	Tennis Court 2
2012-09-21 17:00:00	Tennis Court 1
2012-09-21 18:00:00	Tennis Court 2

17. List all members who have recommended another member at any time? (recommendedby). Get rid of all duplicates, the list should be sorted by name and surname.

```
SELECT DISTINCT mem.firstname , mem.surname
FROM cd.members mem
    INNER JOIN cd.members mems ON mems.memid =
mem.recommendedby
ORDER BY surname, firstname;
```

firstname	surname
Florence	Bader
Anne	Baker
Timothy	Baker
Tim	Boothe
Gerald	Butters
Joan	Coplin
Erica	Crumpet
Nancy	Dare
Matthew	Genting
John	Hunt
David	Jones
Douglas	Jones
Janice	Joplette
Anna	Mackenzie
Charles	Owen
David	Pinker
Millicent	Purview
Henrietta	Rumney
Ramnaresh	Sarwin
Jack	Smith
Ponder	Stibbons
Henry	Worthington-Smyth

18. Print a list of all members, including the members they have recommended (If none were recommended, return NULL). The list should be arranged by name and surname.

```
SELECT mem.firstname, mem.surname,
            rec.firstname as recFirstname,
            rec .surname as recSurname
FROM cd.members mem
         LEFT OUTER JOIN cd.members rec
                ON rec.memid = mem.recommendedby
ORDER BY firstname, surname;
```

firstname	surname	recfirstname	recsurname
Anna	Mackenzie	Darren	Smith
Anne	Baker	Ponder	Stibbons
Burton	Tracy	*NULL*	*NULL*
Charles	Owen	Darren	Smith
Darren	Smith	*NULL*	*NULL*
Darren	Smith	*NULL*	*NULL*
David	Farrell	*NULL*	*NULL*
David	Jones	Janice	Joplette
David	Pinker	Jemima	Farrell
Douglas	Jones	David	Jones
Erica	Crumpet	Tracy	Smith
Florence	Bader	Ponder	Stibbons
Gerald	Butters	Darren	Smith
GUEST	GUEST	*NULL*	*NULL*
Henrietta	Rumney	Matthew	Genting
Henry	Worthington-Smyth	Tracy	Smith
Hyacinth	Tupperware	*NULL*	*NULL*
Jack	Smith	Darren	Smith
Janice	Joplette	Darren	Smith
Jemima	Farrell	*NULL*	*NULL*
Joan	Coplin	Timothy	Baker
John	Hunt	Millicent	Purview
Matthew	Genting	Gerald	Butters
Millicent	Purview	Tracy	Smith
Nancy	Dare	Janice	Joplette
Ponder	Stibbons	Burton	Tracy
Ramnaresh	Sarwin	Florence	Bader
Tim	Boothe	Tim	Rownam
Tim	Rownam	*NULL*	*NULL*
Timothy	Baker	Jemima	Farrell
Tracy	Smith	*NULL*	*NULL*

19. Print a list of all members so that their last names and first names are displayed in the following format <'Surname, Firstname'> (Apply function CONCAT ())

```
SELECT CONCAT(surname , ', ', firstname ) AS name
FROM cd.members;

# alternatively

SELECT surname || ', ' || firstname AS name
FROM cd.members;
```

name
GUEST, GUEST
Smith, Darren
Smith, Tracy
Rownam, Tim
Joplette, Janice
Butters, Gerald
Tracy, Burton
Dare, Nancy
Boothe, Tim
Stibbons, Ponder
Owen, Charles
Jones, David
Baker, Anne
Farrell, Jemima
Smith, Jack
Bader, Florence
Baker, Timothy
Pinker, David
Genting, Matthew
Mackenzie, Anna
Coplin, Joan
Sarwin, Ramnaresh
Jones, Douglas
Rumney, Henrietta
Farrell, David
Worthington-Smyth, Henry
Purview, Millicent
Tupperware, Hyacinth
Hunt, John
Crumpet, Erica
Smith, Darren

20. Print the initials of all members (assume that each member has only one first name and one last name).

```
SELECT CONCAT(LEFT(surname ,1),'.',LEFT(firstname ,1),'.') AS name
FROM cd.members;
```

name
G.G.
S.D.
S.T.
R.T.
J.J.
B.G.
T.B.
D.N.
B.T.
S.P.
O.C.
J.D.
B.A.
F.J.
S.J.
B.F.
B.T.
P.D.
G.M.
M.A.
C.J.
S.R.
J.D.
R.H.
F.D.
W.H.
P.M.
T.H.
H.J.
C.E.
S.D.

Introduction to Databases

Exercises: QBE

↳ query by example

QBE basics (1)

- A user writes queries by creating **example tables.**
- QBE uses **domain variables**, as in the DRC, to create example tables.
- The domain of a variable is determined by the column in which it appears, and **variable symbols** are prefixed with **underscore (_)** to distinguish them from constants.
- The fields that should appear in the answer are specified by using the command **P.**, which stands for **print**.
  - To print the names and ages of all sailors, we would create the following example table:

Sailors	*sid*	*sname*	*rating*	*age*
		P._N		P._A

  - Query in DRC: $\{\langle N,A\rangle \mid \exists\, I,N,T,A(\langle I,N,T,A\rangle \in \text{Sailors})\}$

QBE basics (2)

- If we want to print **all fields** in some relation, we can **place P. under the name of the relation**.
  - This notation is like the SELECT * convention in SQL. It is equivalent to placing a P. in every field:

Sailors	*sid*	*sname*	*rating*	*age*
P.				

- Selections are expressed by placing a constant in some field:

Sailors	*sid*	*sname*	*rating*	*age*
P.			> 10	

  - Placing a constant, say 10, in a column is the same as placing the condition =10. We can use other comparison operations (<, >, <=, >=, ¬) as well. For example, we could say < 10 to retrieve sailors with a rating less than 10.

QBE basics (3)

- We can explicitly specify whether **duplicate** tuples in the answer are to be **eliminated** (or ***not***) by putting **UNQ**. (respectively ***ALL***.) under the relation name.
- We can **order** the presentation of the answers through the use of the **.AO** (for **ascending order**) and **.DO** commands **in conjunction with P.** .
  An **optional integer argument** allows us to **sort on more than one field**.
  - For example, we can display the names, ages, and ratings of all sailors in ascending order by age, and for each age, in ascending order by rating as follows:

Sailors	*sid*	*sname*	*rating*	*age*
UNQ. → no duplicates ALL. → yes duplicates		P.	AO(2). then ordered by rating	AO(1). first ordered by age

Queries over multiple relations (1)

- To find **sailors with a reservation**, we have to combine information from the Sailors and the Reserves relations.
  In particular we have to **select tuples** from the **two relations with the same value** in the join **column *sid***. We do this by **placing the same variable in the *sid* columns** of the two example relations.

Sailors	***sid***	***sname***	***rating***	***age***
	_Id	P._S		

Reserves	**sid**	**bid**	**day**
	_Id		

Queries over multiple relations (2)

- To find sailors who have reserved a boat for **8/24/96** and who are **older than 25**, we could write:

Sailors	*sid*	*sname*	*rating*	*age*
	_Id	P._S		> 25

Reserves	sid	bid	day
	_Id		'8/24/96'

connected

- The following query prints the names and ages of sailors who have reserved some boat that is also reserved by the sailor with id 22:

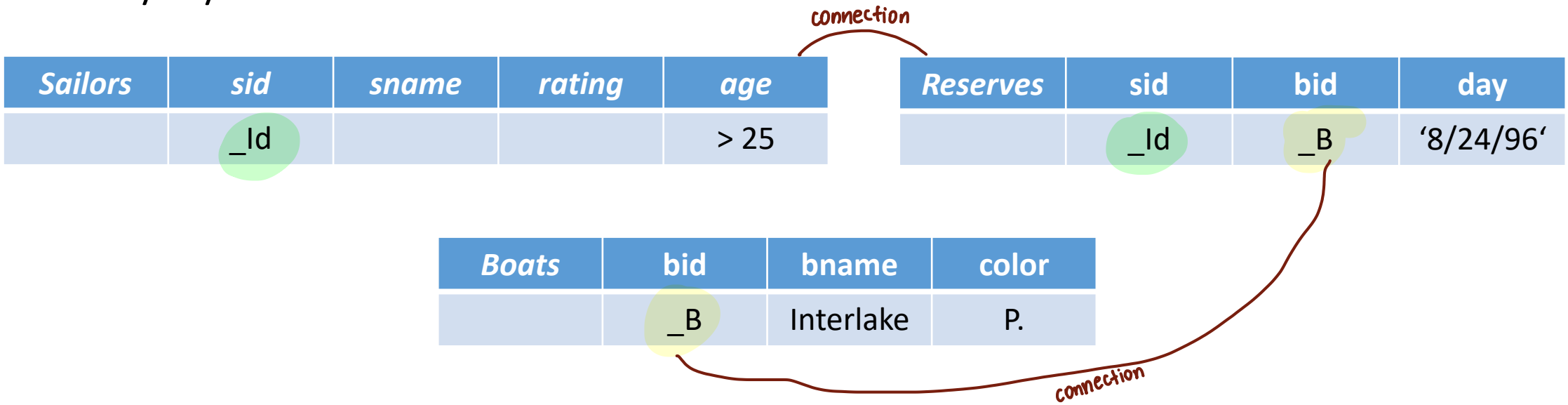
Sailors	*sid*	*sname*	*rating*	*age*
	_Id	P._N		P._A

Reserves	sid	bid	day
	_Id	_B	
	22	_B	

interconnect

Queries over multiple relations (3)

- Find the colors of Interlake boats reserved by sailors who have reserved a boat for 8/24/96 and who are older than 25:



Negation

- We can print the names of sailors who **do not** have a reservation by using the ¬ command in the relation name column:

Sailors	*sid*	*sname*	*rating*	*age*
	_Id	P._S		

Reserves	sid	bid	day
¬	_Id		

Aggregates

- Like SQL, QBE supports the aggregate operations **AVG.**, **COUNT.**, **MAX.**, **MIN.**, and **SUM.**
  - By default, these aggregate operators **do not eliminate duplicates**, with the exception of COUNT., which does eliminate duplicates.
  - To eliminate duplicate values, the variants AVG.UNQ. and SUM.UNQ. must be used.
  - Curiously, there is no variant of COUNT. that does not eliminate duplicates.
- The following query prints the average age of sailors:

Sailors	*sid*	*sname*	*rating*	*age*	
				_A	P.AVG._A

- QBE supports **grouping** through the use of the **G.** command. To print average ages by rating, we could use:

Sailors	*sid*	*Sname*	*rating*	*age*	
			G.	_A	P.AVG._A

Conditions box

- Simple conditions can be expressed directly in columns of the example tables. For more complex conditions QBE provides a feature called a **conditions box**. Conditions boxes are used to do the following:
  - *Express a condition involving two or more columns*, such as _R/_A > 0.2
  - *Express a condition involving an aggregate operation on a group*, for example, AVG. A > 30. In conjunction with G., only columns with either G. or an aggregate operation can be printed!
    - The following query prints those ratings for which the average age is more than 30:

Sailors	*sid*	*sname*	*rating*	*age*	*Conditions*
			G.P.	_A	AVG._A > 30

  - *Express conditions involving the **AND** and **OR** operators*.
    - We can print the names of sailors who are younger than 20 or older than 30 as follows:

Sailors	*sid*	*sname*	*rating*	*age*	*Conditions*
		P.		_A	_A < 20 OR 30 < _A

AND/OR queries

- AND and OR can be expressed in QBE **without** using a **conditions box**.
  - We can print the names of sailors who are younger than 30 or older than 20 by simply creating two example rows:

Sailors	*sid*	*sname*	*rating*	*age*
		P.		< 30
		P.		> 20

  - To print the names of sailors who are **both** younger than 30 and older than 20, we use the same variable in the key fields of both rows:

Sailors	*sid*	*sname*	*rating*	*age*
	_Id	P.		< 30
	_Id			> 20

Unnamed columns

- If we want to display some information **in addition** to fields retrieved from a relation, we can create unnamed columns for display.
  - We can print the name of each sailor along with the ratio rating/age as follows:

Sailors	*sid*	*sname*	*rating*	*age*	
		P.	_R	_A	P._R / _A

- All our examples have included P. commands in exactly **one** table. This is a **QBE restriction**. If we want to display fields from more than one table, we have to use unnamed columns.
  - To print the names of sailors along with the dates on which they have a boat reserved, we could use the following:

Sailors	*sid*	*sname*	*rating*	*age*	
	_Id	P.			P._D

Reserves	sid	bid	day
	_Id		_D

Updates - insert

- **Insertion**, **deletion**, and **modification** of a tuple are specified through the commands **I.**, **D.**, and **U.**, respectively.
  - We can **insert a new tuple** into the Sailors relation as follows:

Sailors	*sid*	*sname*	*rating*	*age*
I.	74	Jana	7	41

  - We can insert several tuples, computed essentially through a query, into the Sailors relation as follows:

Sailors	*sid*	*sname*	*rating*	*age*
I.	_Id	_N		_A

Students	*sid*	*name*	*login*	*age*
	_Id	_N		_A

Conditions
_A > 18 OR _N LIKE ‘C%‘

We insert one tuple for each student older than 18 or with a name that begins with C. The rating field of every inserted tuple contains a null value.

Updates - delete

- We can **delete all tuples** with rating > 5 from the Sailors relation as follows:

Sailors	*sid*	*sname*	*rating*	*age*
D.			> 5	

- We can delete all reservations for sailors with rating < 4 by using:

Sailors	*sid*	*sname*	*rating*	*age*
	_Id		< 4	

Reserves	sid	bid	day
D.	_Id		

Updates - modify

- We can **update** the age of the sailor with *sid* 74 to be 42 years by using:

Sailors	*sid*	*sname*	*rating*	*age*
	74			U.42

- The fact that sid is the key is significant here; **we cannot update the key field**, but we can use it to identify the tuple to be modified (in other fields).
  We can also change the age of sailor 74 from 41 to 42 **by incrementing the age value**:

Sailors	*sid*	*sname*	*rating*	*age*
	74			U._A+1

QBE exercises

1. Print the names of all employees who work on the 10th floor and make less than $50,000.

Emp	eid	enam e	salary	
	_id	P.	<50000	

OR _S

Works	eid	did	
	_id	_did	

Dept	did	dnam e	manageri d	floornu m	
	_did			10	

Conditions
_S < 50000 ↘ we can use the condition, but we don't have to

2. Print the names of all managers who manage three or more departments on the same floor.

Emp	eid	enam e	salary	
	_E	P.		

Works	eid	did	
		_did	

Dept	did	dnam e	manageri d	floornu m	
	_D1		_E	_F	
	_D2		_E	_F	
	_D3		_E	_F	

Conditions
D1!=D2 and D2!=D3 and D1!=D3

3. Print the names of all managers who manage ten or more departments on the same floor.

Emp	eid	enam e	salary	
	_E	P.		

Works	eid	did	

Dept	did	dnam e	manageri d	floornu m	
	_D		G._E	G._F	

Conditions
COUNT._D >=10

↓ count departments eliminates duplicates

Count the departments → group by managers and floor

4. Give every employee who works in the Toy department a 10% raise.

Emp	eid	enam e	salary	
	_E		U._S·1.1	

Works	eid	did	
	-E	_D	

Dept	did	dnam e	managerid	floornum	
	_D	'Toy'			

Conditions

5. Print the names of the departments (in ascending order) that employee Santa works in.

Emp	eid	enam e	salary	
	-E	'Santa'		

Works	eid	did	
	_E	- Did	

Dept	did	dnam e	managerid	floornum	
	- Did	P.A0.			

Conditions

6. Print the names and salaries of employees who work in both the Toy department and the Candy dept.

Emp	eid	enam e	salary	
	_E	P.	P.	

Works	eid	did	
	_E	_D	

Dept	did	dnam e	managerid	floornum	
	_D	'Toy'			
	_D	'Candy'			

Conditions

7. Print the names of employees who earn a salary that is either less than $10,000 or more than $100,000.

Emp	eid	enam e	salary	
		P.	<10000	
		P.	>100000	

Conditions

Works	eid	did	

Dept	did	dnam e	manageri d	floornu m	

8. Print all of the attributes for employees who work in some department that employee Santa also works in.

Emp	eid	enam e	salary	
	_E	'Santa'		
P.	_E1			

Conditions
_E!=_E1

because we don't want to print Santa

Works	eid	did	
	_E	_D	
	_E1	_D	

join with works

Dept	did	dnam e	manageri d	floornu m	
	_D				

9. Fire Santa.

Emp	eid	enam e	salary	
D.	_E	'Santa'		

Conditions

Works	eid	did	
	_E	_D	

Dept	did	dnam e	manageri d	floornu m	
	_D				

10. Print the names of employees who make more than 20,000 and work in either the Video department or the Toy department.

Emp	eid	enam e	salary	
	_E	P.	>20000	

Works	eid	did	
	_E	_D	

Dept	did	dnam e	managerid	floornum	
	_D	'Video'			
	_D	'Toy'			

Conditions
_DN = 'Video' OR _DN = 'Toy'

tek satır yapmak istersen bunu kullan.

11. Print the name of each employee who earns more than the manager of the department that he or she works in.

Emp	eid	enam e	salary	
	_E	P.	>_S	
	_M		_S	

Works	eid	did	
	_E	_D	

Dept	did	dnam e	managerid	floornum	
	_D		_M		

Conditions

connection için ortaya yaz.

12. Print the names of all employees who work on the floor(s) where John works.

Emp	eid	enam e	salary	
	_E1	P.		
	_E	'John'		

Works	eid	did	
	_E1	_D1	
	_E	_D	

Dept	did	dnam e	managerid	floornum	
	_D1			_F	
	_D			_F	

Conditions
_E1!=_E

if we don't want to include John

Introduction to Database Systems

Exercises: indexing

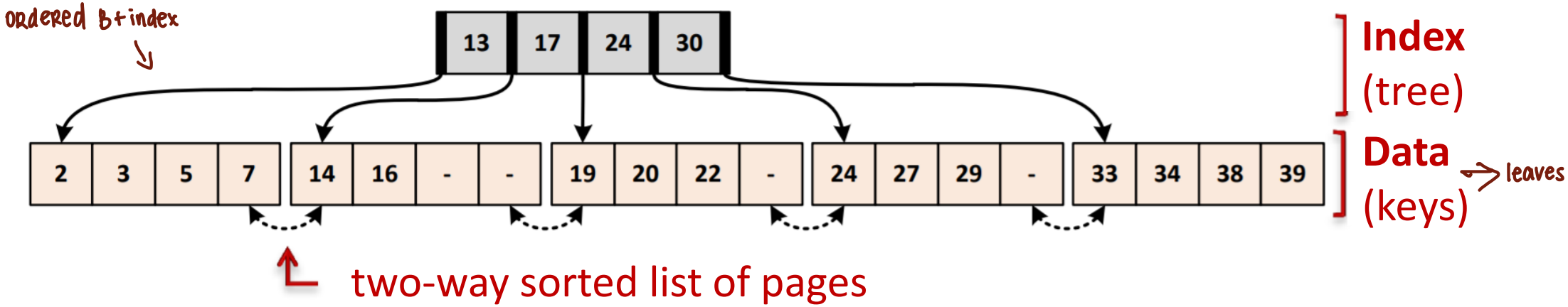
Source: D. Lavbič, Osnove podatkovnih baz - vaje, UL FRI

Introduction

- The purpose of indexing is to **increase the efficiency of the system**.
  - We usually want to improve the efficiency (speed) of the queries.
- Indexes are **pointers** (shortcuts) **to records** so that they can be found faster and we do not need to search the complete database.
- We will cover the following indexing methods, which are commonly found in databases:
  - B+ index (most commonly used in databases),
  - ISAM index and
  - Hash index

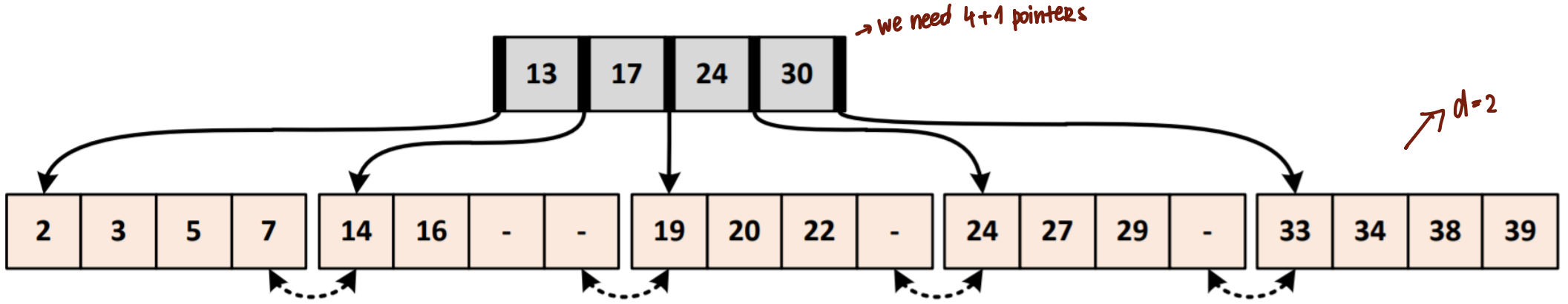
B+ index

- Dynamic = The structure is adjusted to the changes in the file.
  - Balanced tree where nodes direct search, and data (keys) is stored in leafs.
  - The insert and delete operations keep the tree balanced.



B+ index

- Condition of the node: **d ≤ m ≤ 2d** (at least 50% full)
  - **m** … number of records (occupancy) of the node
  - **d** … order of the tree (**2d** = capacity of the node)
  - For the root: **1 ≤ m ≤ 2d**

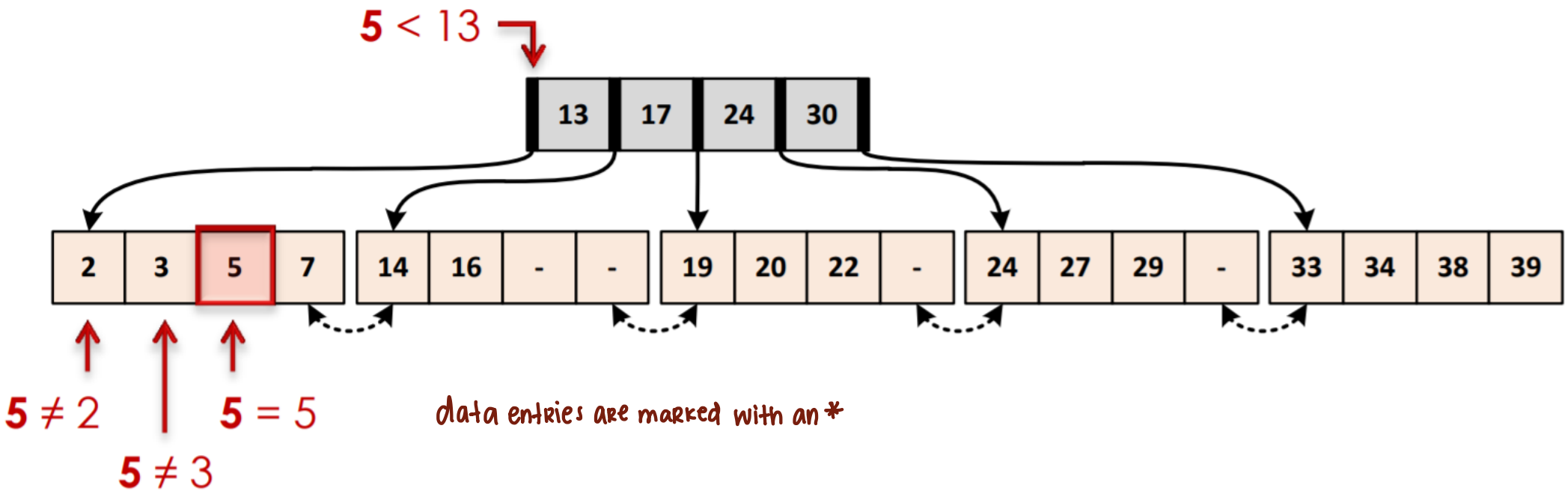


Order of the tree **d=2**.

Node **occupancy** (index) is **at least 2** and **up to 4** records.

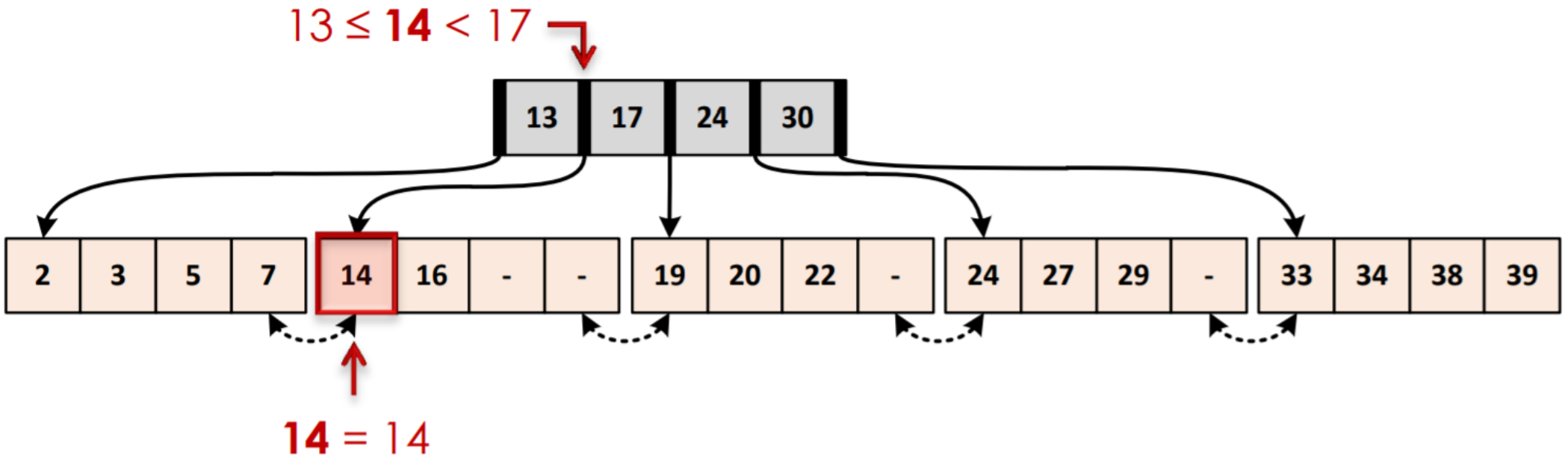
B+ index – example

- **Find** the entry with key **5**.



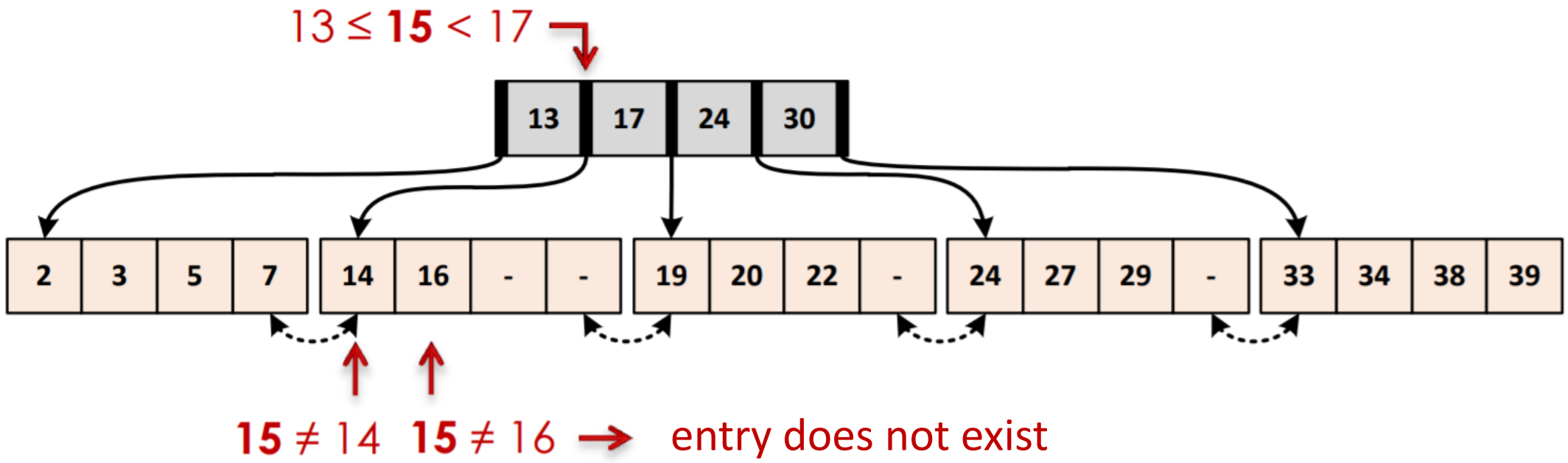
B+ index – example

- **Find** the entry with key **14**.



B+ index – example

- **Find** the entry with key **15**.



B+ index – inserting data

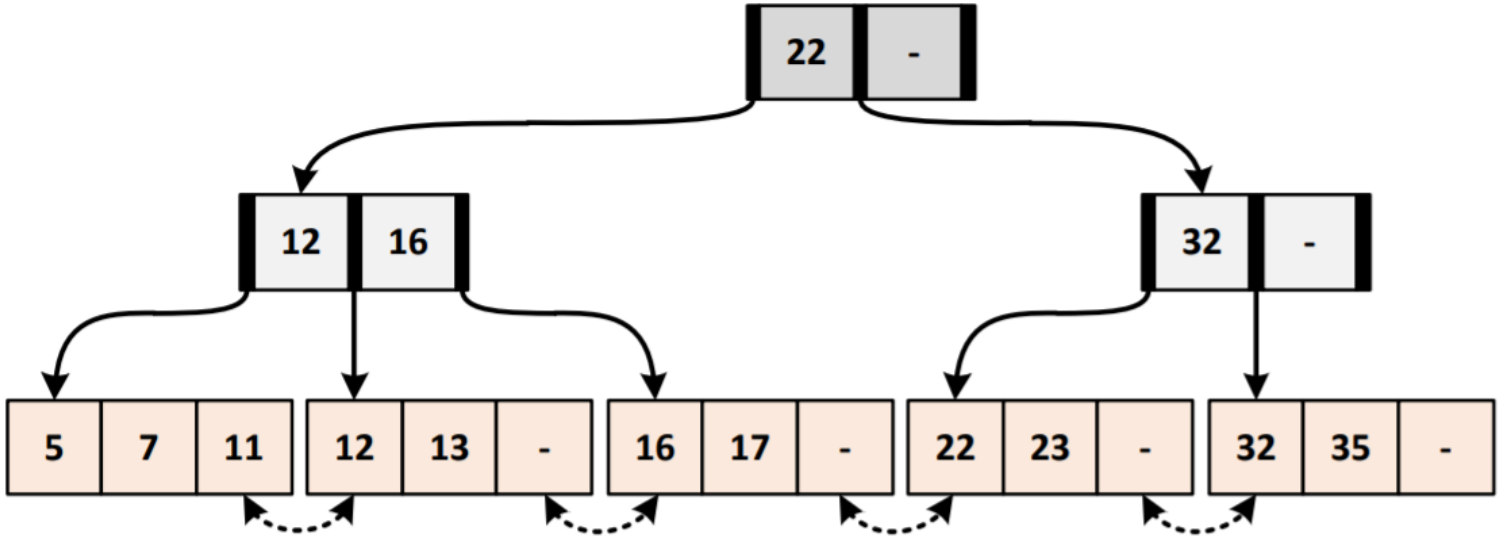
- Algorithm for adding a record k*:
  - **FIND** the leaf where k* belongs
  - **IF** there si free space in the leaf, insert k*
  - **ELSE** divide the leaf in 2 parts:
    - d elements go to the left leaf → 50% occupancy
    - the rest go to the right leaf
  - If necessary, correct the split key up the index tree. ↗ update
  - There are 2 versions: **with** and **without redistribution**
    - If not specified, by default it is used without redistribution.

when we split index levels, we push d+1 key one level up

without redistribution → split the page
with redistribution → borrowing entry from neighboring page (don't split!!!)

B+ index – example 1

- The constructed B+ index is shown on the figure below:

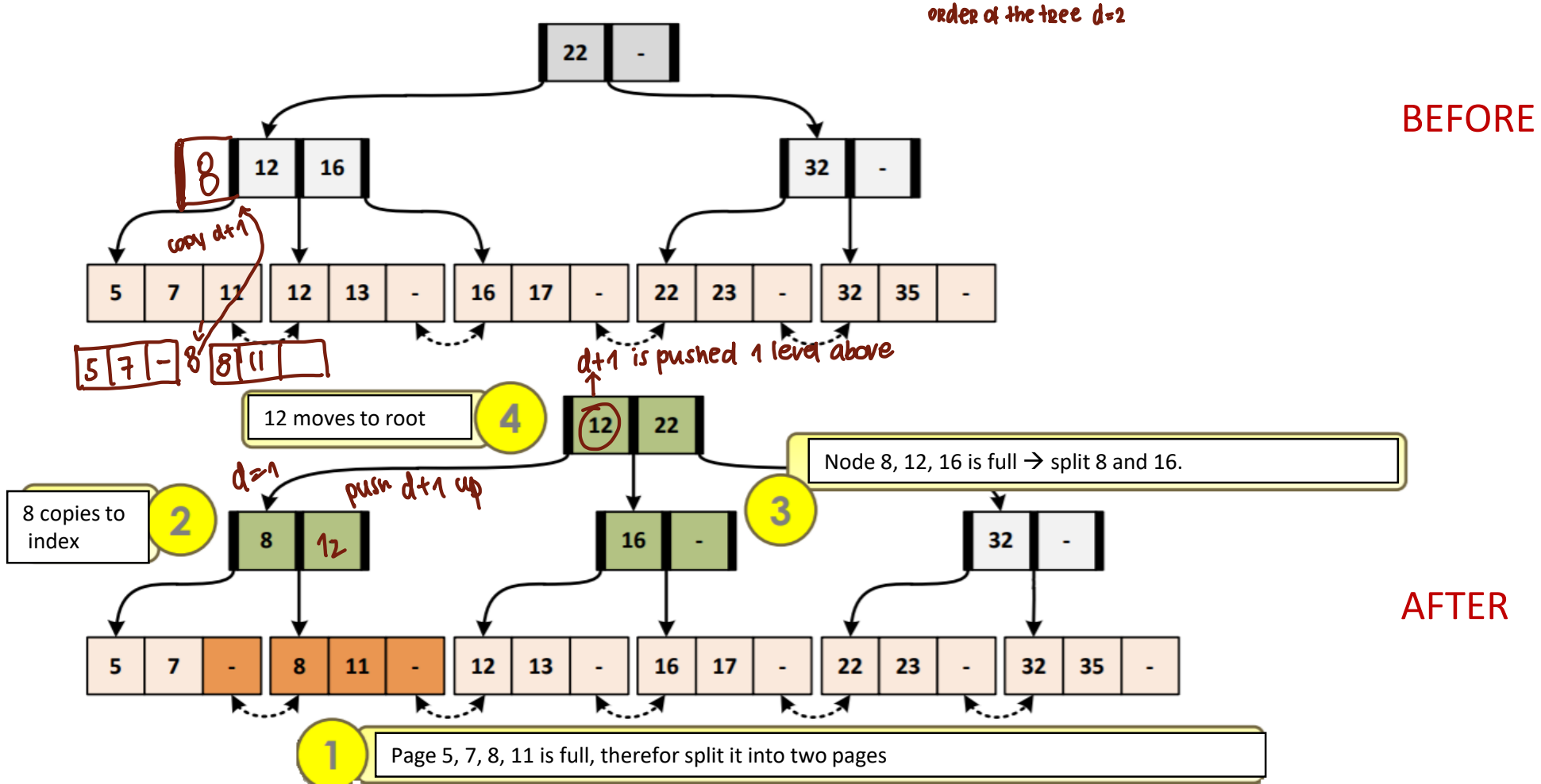


- Insert the entries with keys **8**, **24** and **25** once **without redistribution** and once **with redistribution**.

B+ index – example 1

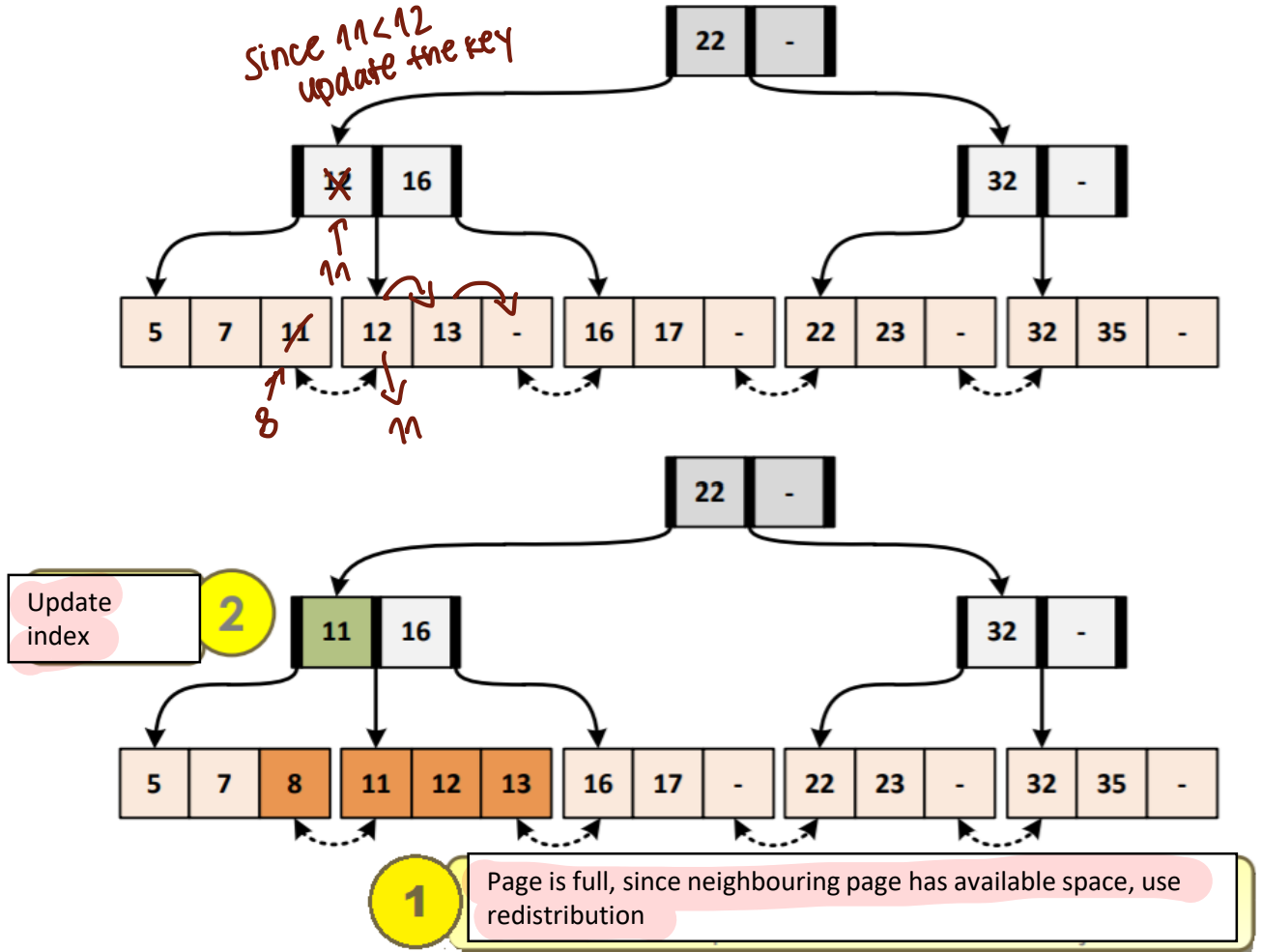
splitting the page

- Inserting the entry with key **8 (without redistribution)**:



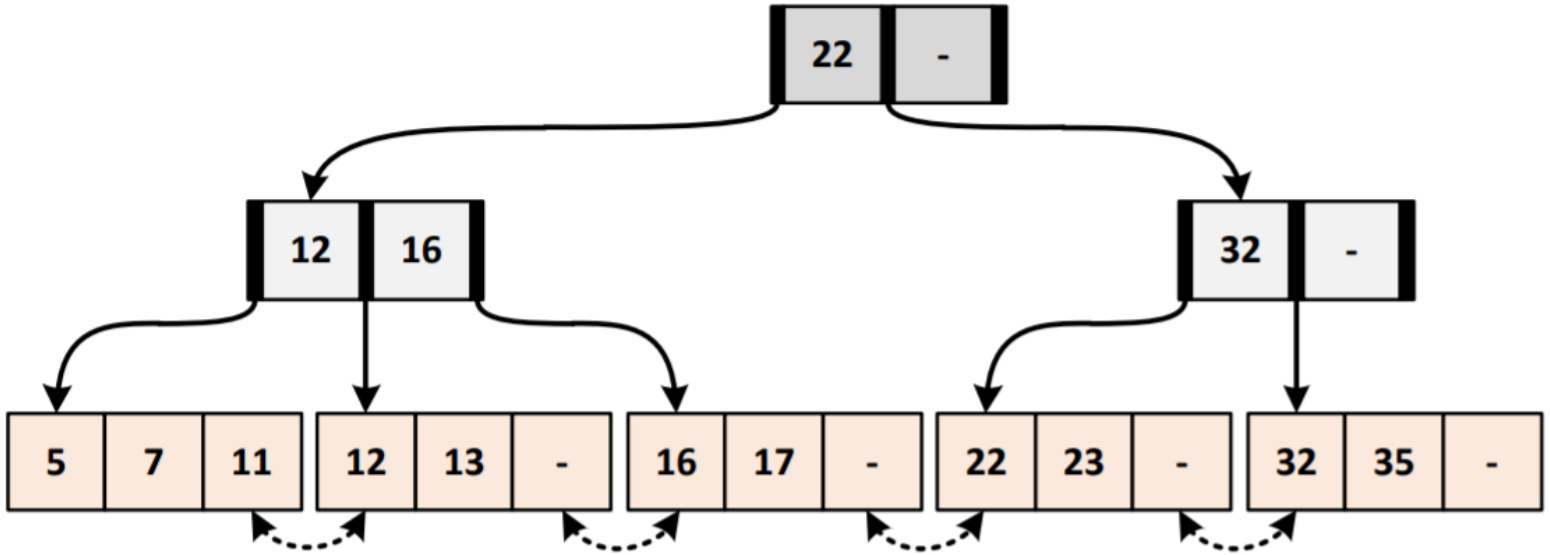
B+ index – example 1

- Inserting the entry with key **8 (with redistribution)**::



B+ index – example 2

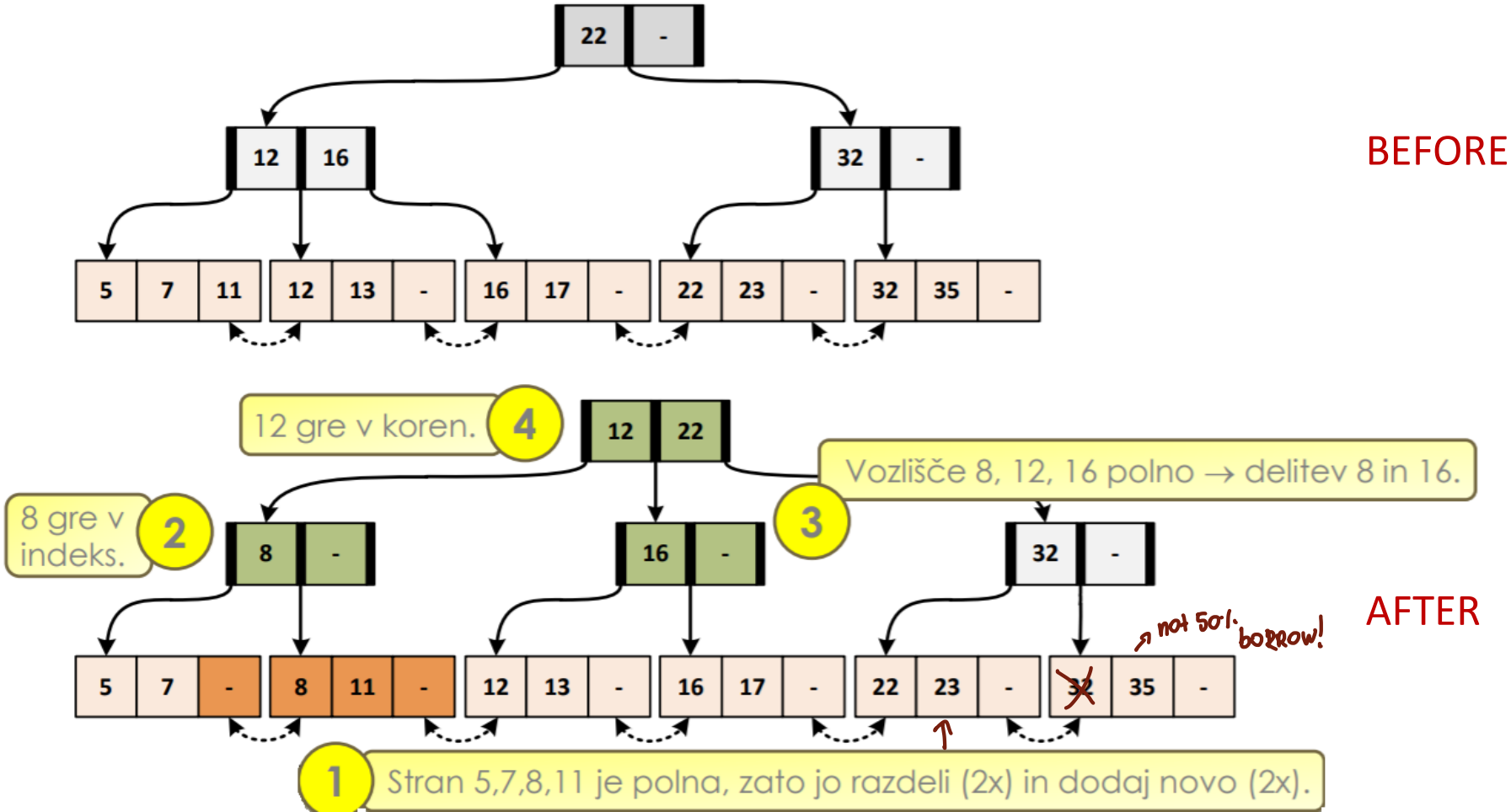
- The constructed B+ index is shown on the figure below:



- First **insert** the record with **key 8**, then **delete** the record with **key 32**.

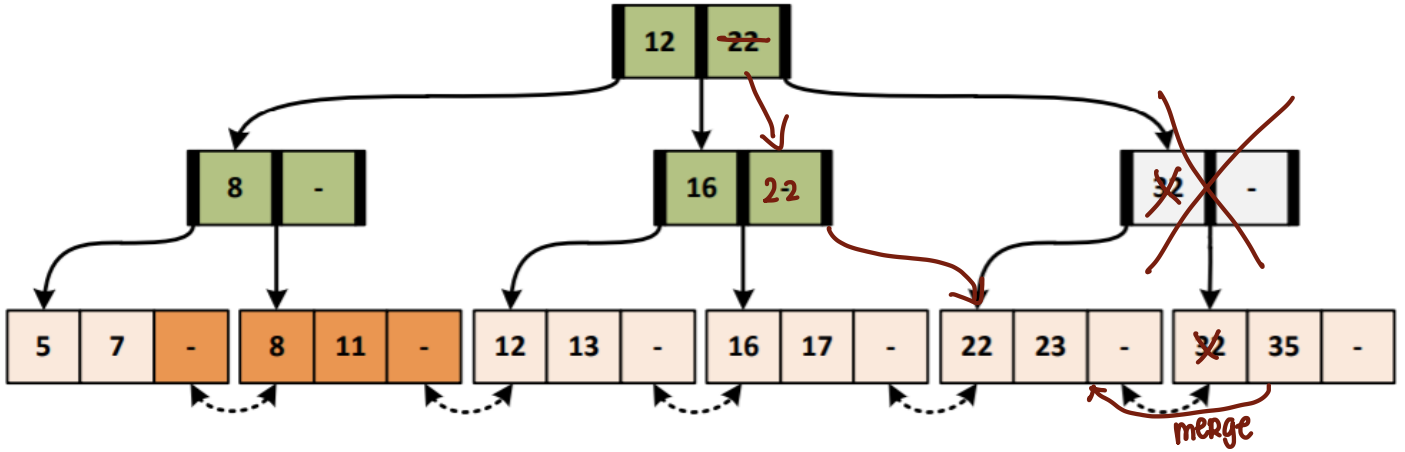
B+ index – example 2

- **Inserting** the entry with key **8**.

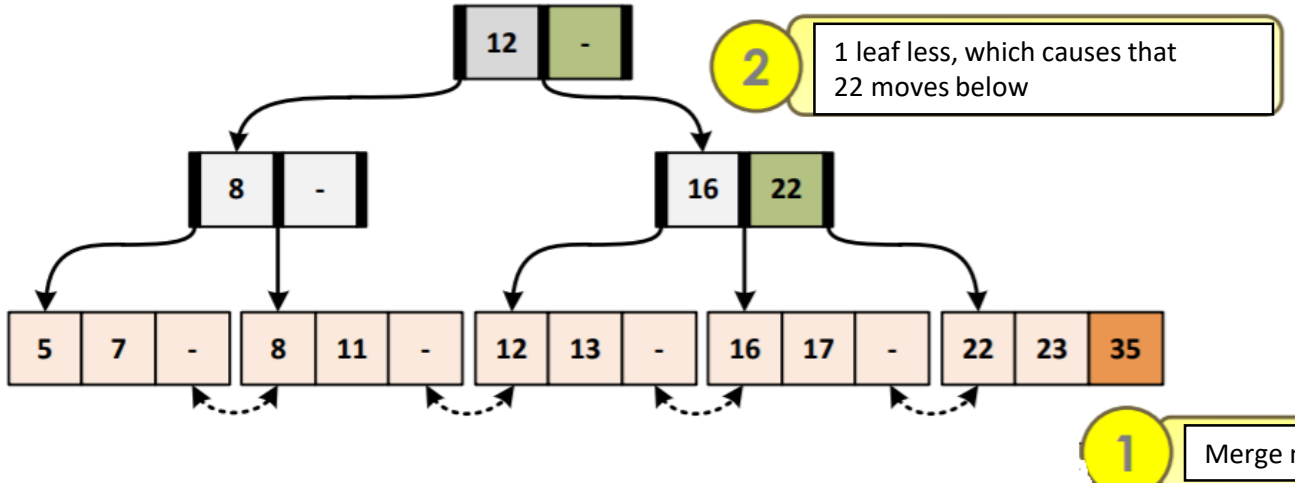


B+ index – example 2

- **Delete** the entry with key **32**.



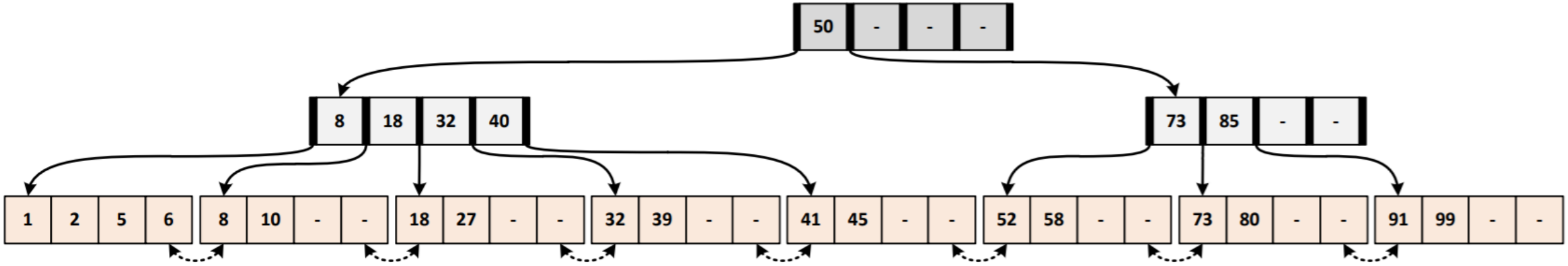
PRED operacijo



PO operaciji

B+ index – Exercise 1

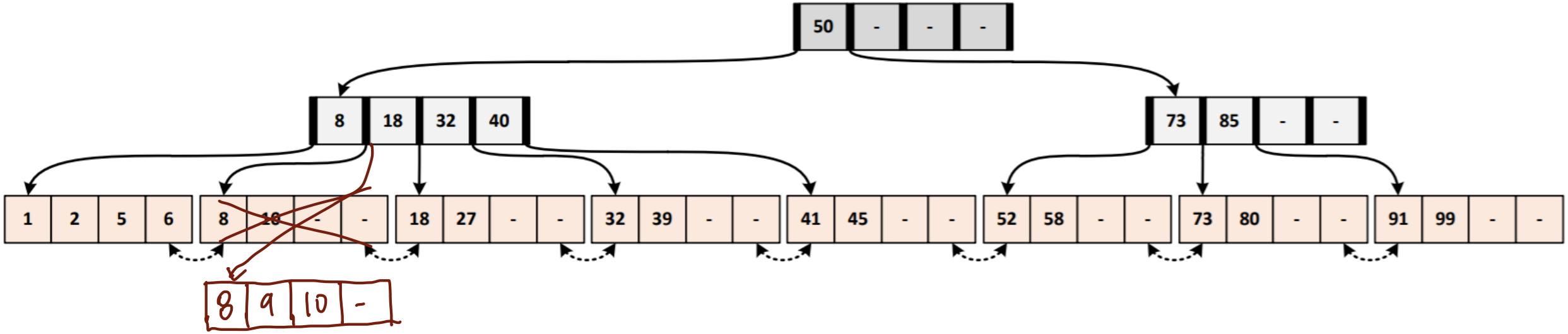
- The constructed B+ index is shown on the figure below.
- You will have to perform certain operations and display the result in the form of a tree.



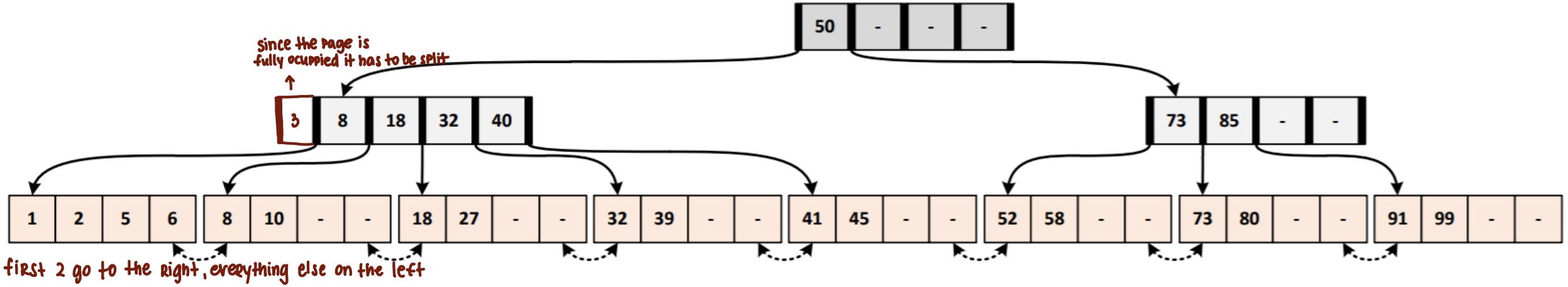
B+ index – Exercise 1

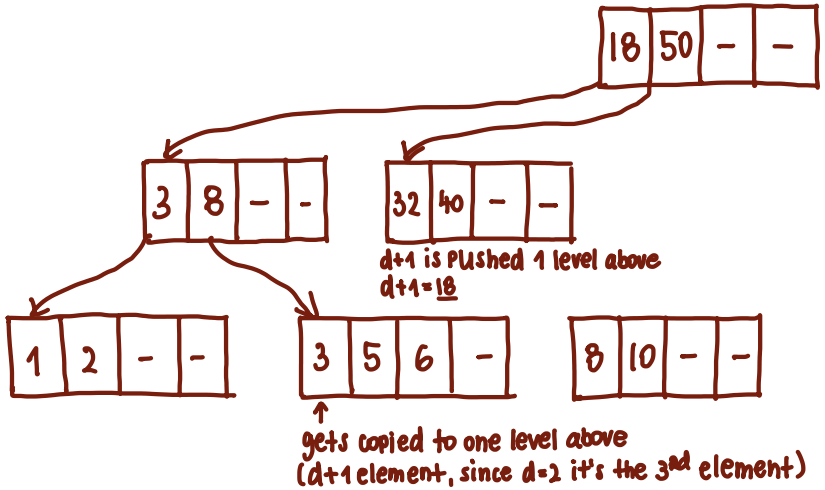
- **Insert** the entry with key **9**.
- **Insert** the entry with key **3**.
- **Delete** the entry with key **8** (redistribution with left side).
- **Delete** the entry with key **8** (redistribution with right side).
- **Insert** the entry with key **46** and then **delete** the entry with key **52**.
- **Delete** the entry with key **91**.
- **Delete** the entries with key **32**, **39**, **41**, **45, 73**.

Insert the entry with key **9**.

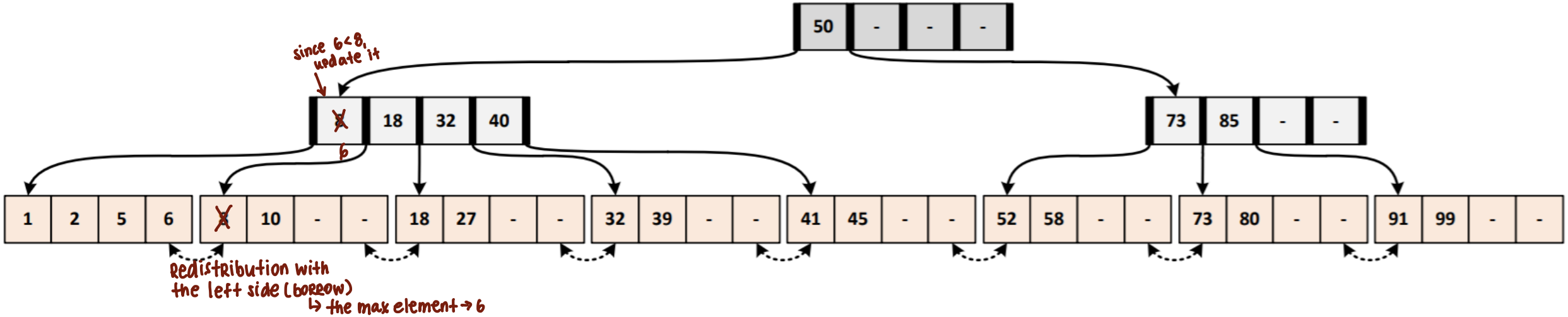


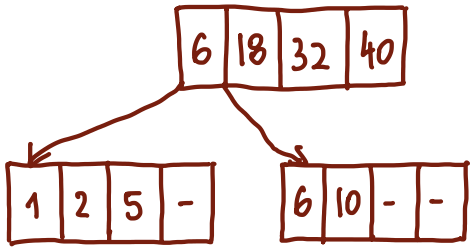
Insert the entry with key **3**.



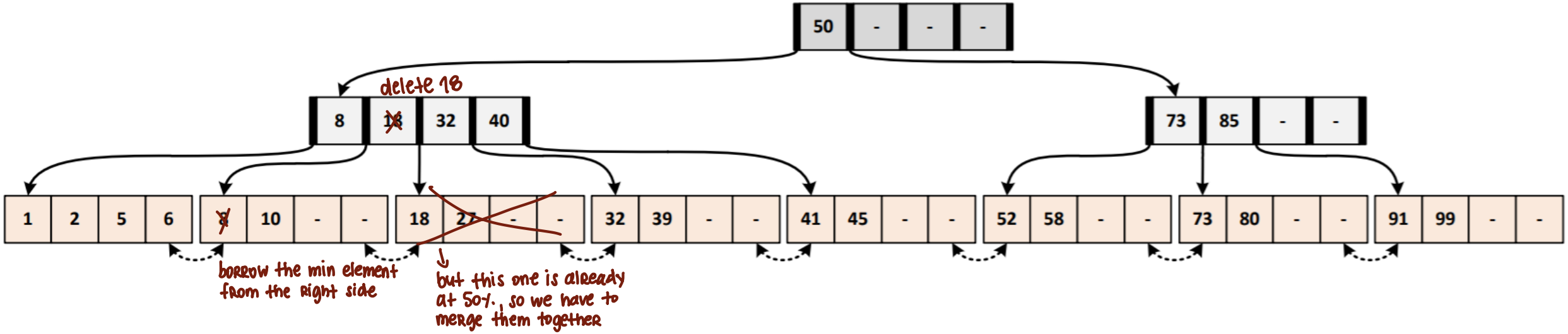


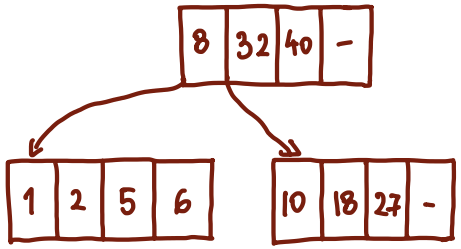
Delete the entry with key **8** (redistribution with left side).



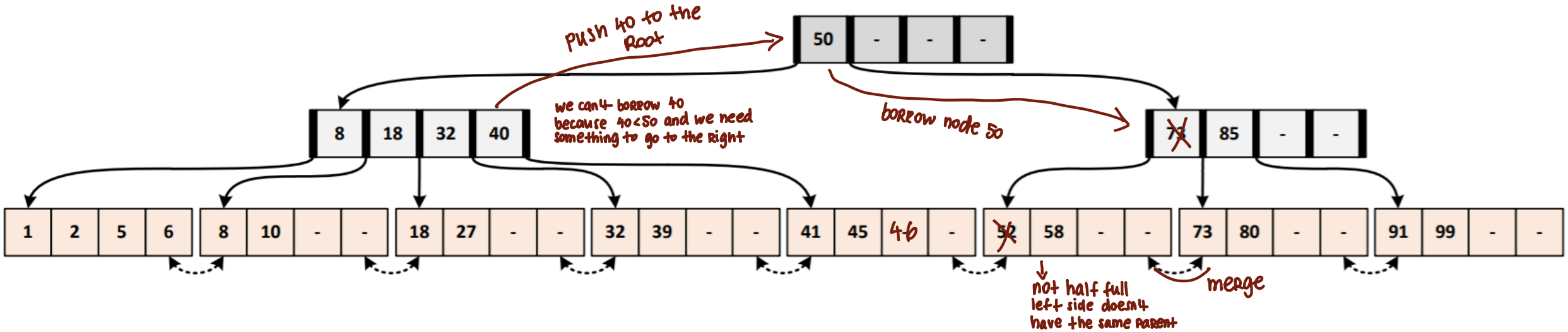


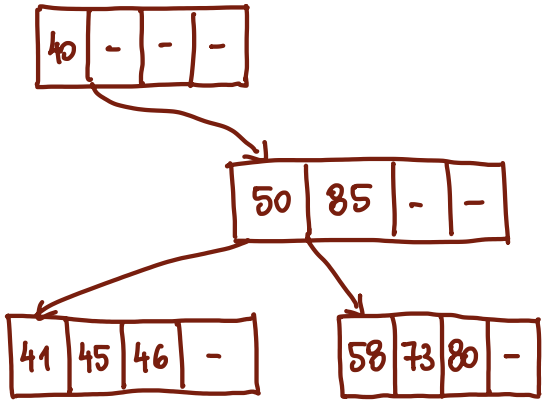
Delete the entry with key **8** (redistribution with right side).



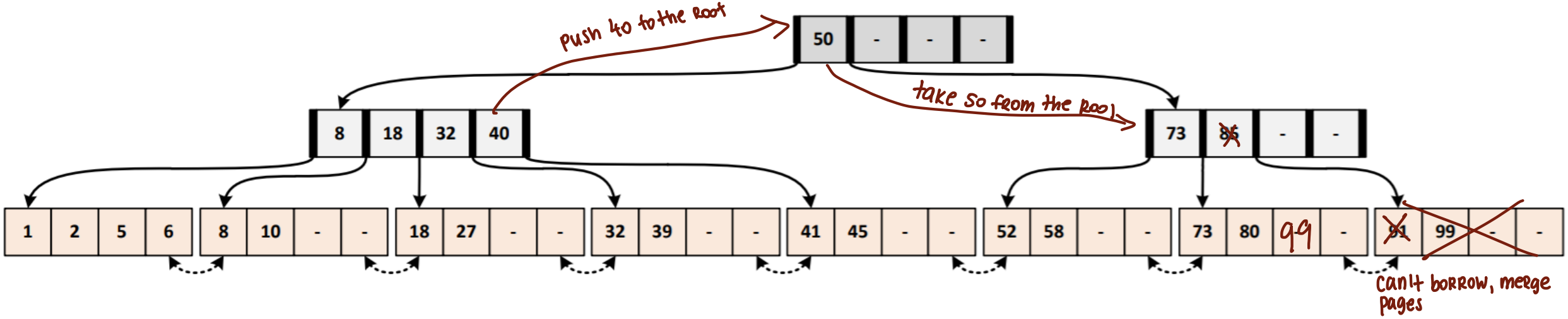


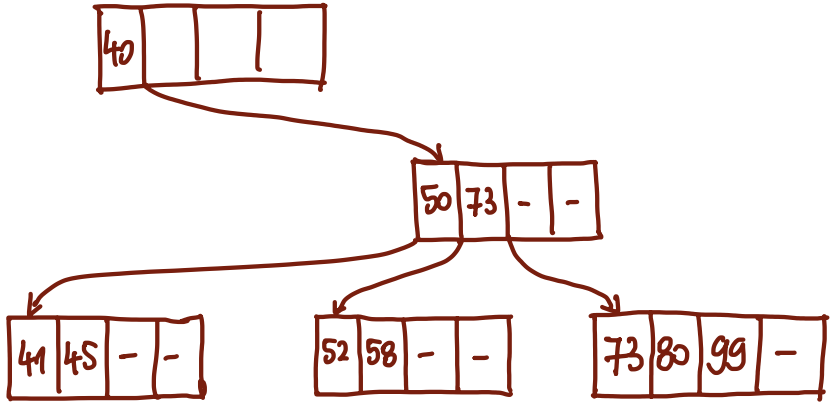
Insert the entry with key **46** and then **delete** the entry with key **52**.



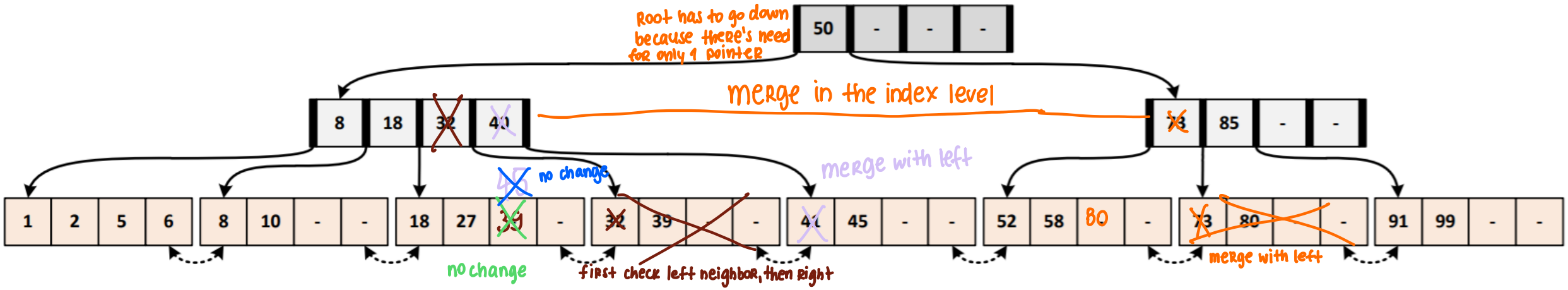


Delete the entry with key **91**.





Delete the entries with key 32, 39, 41, 45, 73.



End Root

B+ index – Exercise 2

a) Construct a B+-tree for the following set of key values:

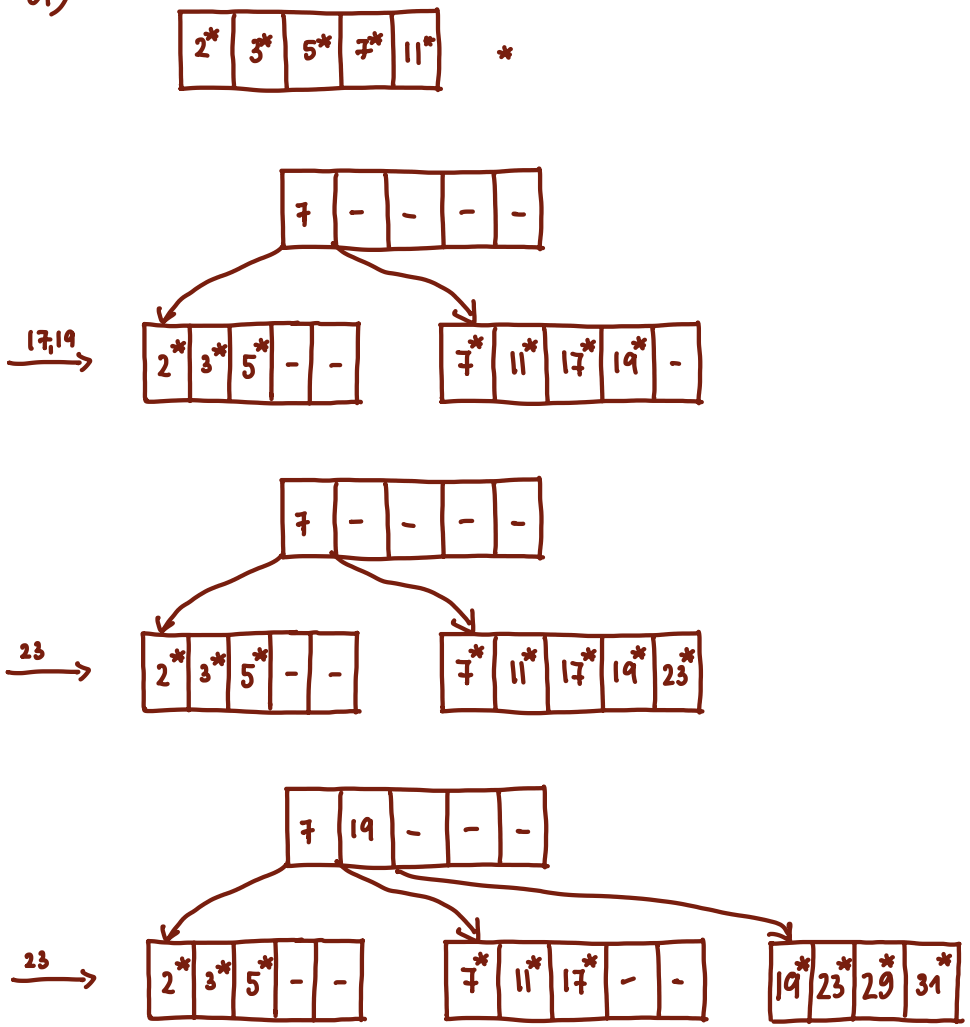
(2, 3, 5, 7, 11, 17, 19, 23, 29, 31)

Assume that the tree is initially empty and values are added in ascending order. Construct B+-tree where the number of pointers that will fit in one node is 6. = 5 entries per node

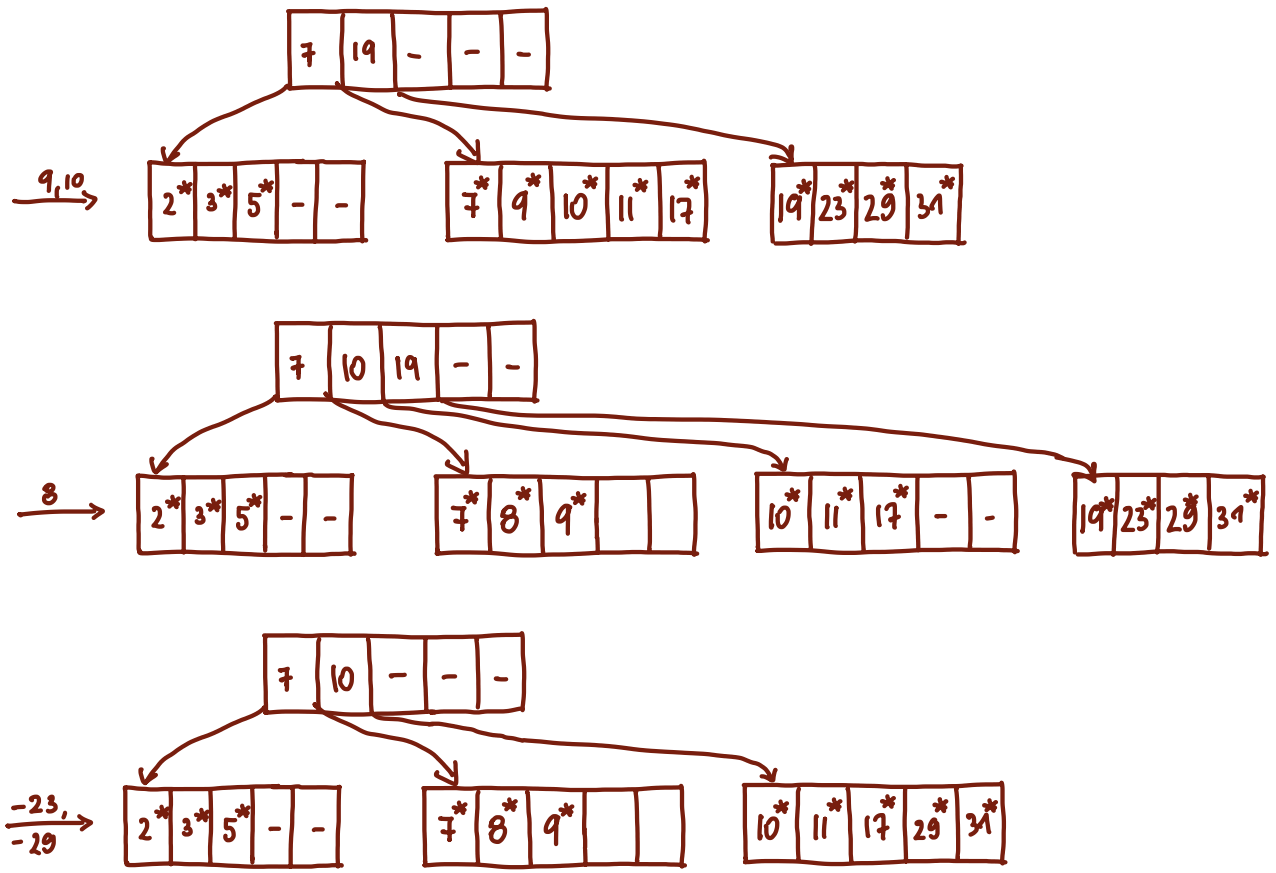
b) Show the form of the tree (constructed in the example (a)) after each of the following series of operations:

1. Insert 9.
2. Insert 10.
3. Insert 8.
4. Delete 23.
5. Delete 19.

a)

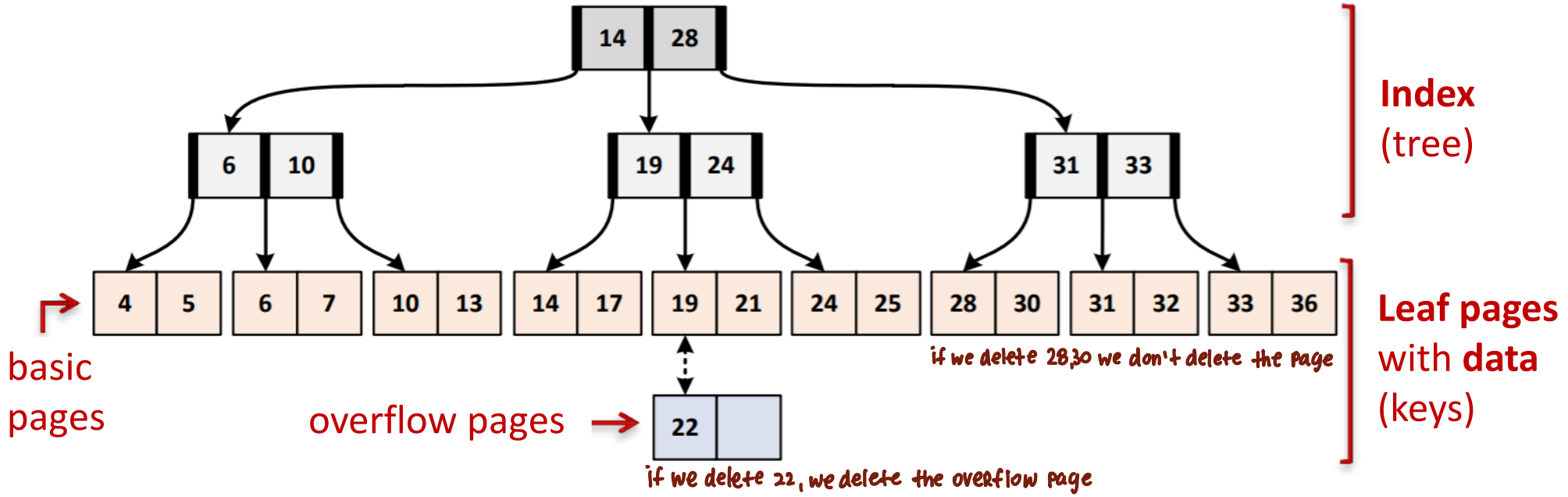

2* 3* 5* 7* 11*
7 - - - -
17,19
2* 3* 5* - -
7* 11* 17* 19* -
7 - - - -
23
2* 3* 5* - -
7* 11* 17* 19* 23*
7 19 - - -
23
2* 3* 5* - -
7* 11* 17* - -
19* 23* 29* 31*

b)


7 19 - - -
9,10
2* 3* 5* - -
7* 9* 10* 11* 17*
19* 23* 29* 31*
7 10 19 - -
8
2* 3* 5* - -
7* 8* 9*
10* 11* 17* - -
19* 23* 29* 31*
7 10 - - -
-23,
-29
2* 3* 5* - -
7* 8* 9*
10* 11* 17* 29* 31*

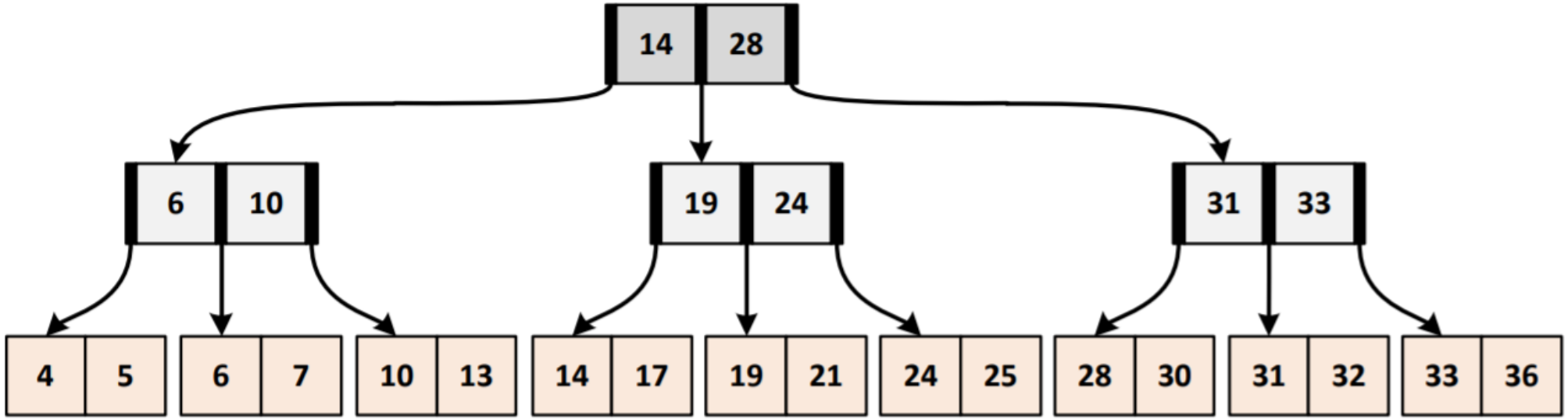
ISAM index

- **Static** (with the exception of overflow pages) = need to be reorganised.
- The data in the overflow pages are not sorted → due to the effectiveness of adding the records.



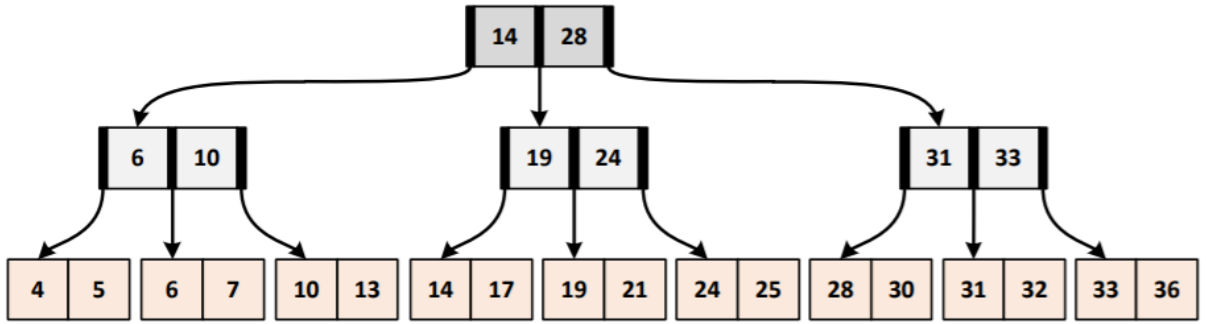
ISAM index – Exercise 3

- The constructed ISAM index is show on the figure below.
- You will have to perform certain operations and display the result in the form of a tree.

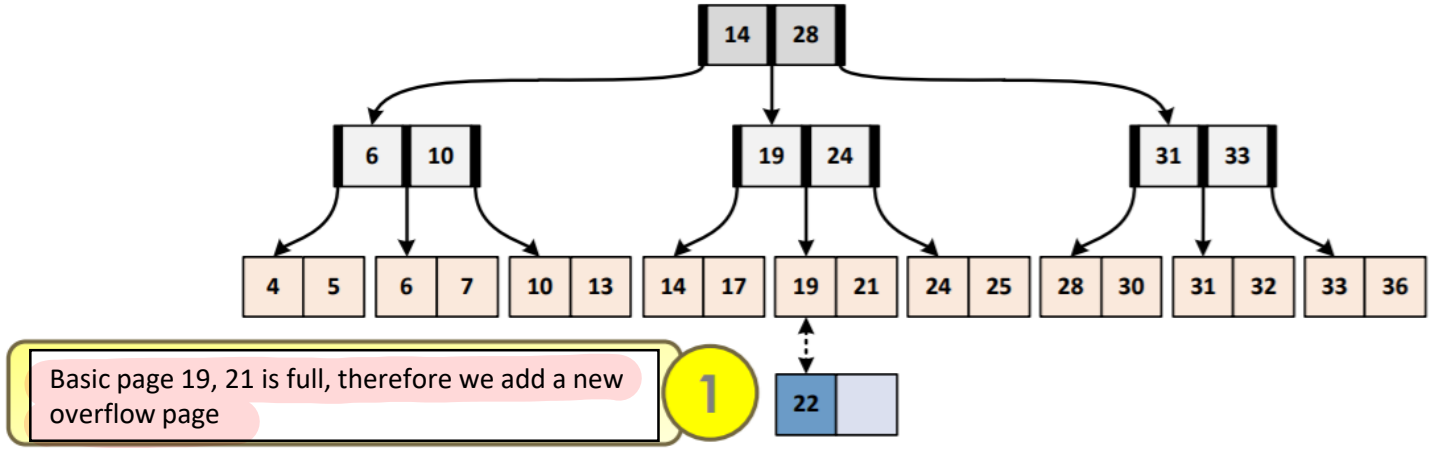


ISAM index – Exercise 3

- **Insert** entry with the key **22**.



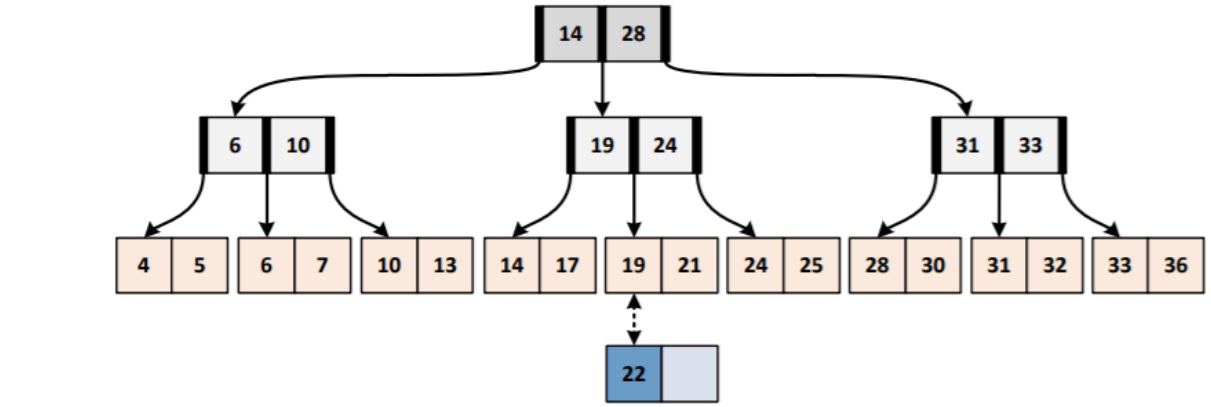
BEFORE



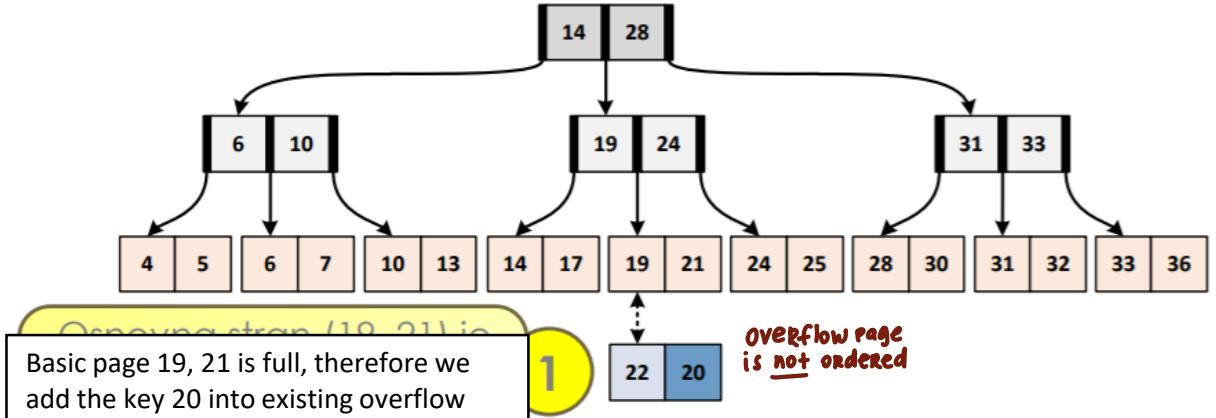
AFTER

ISAM index – Exercise 3

- **Insert** entry with key **20**.



BEFORE

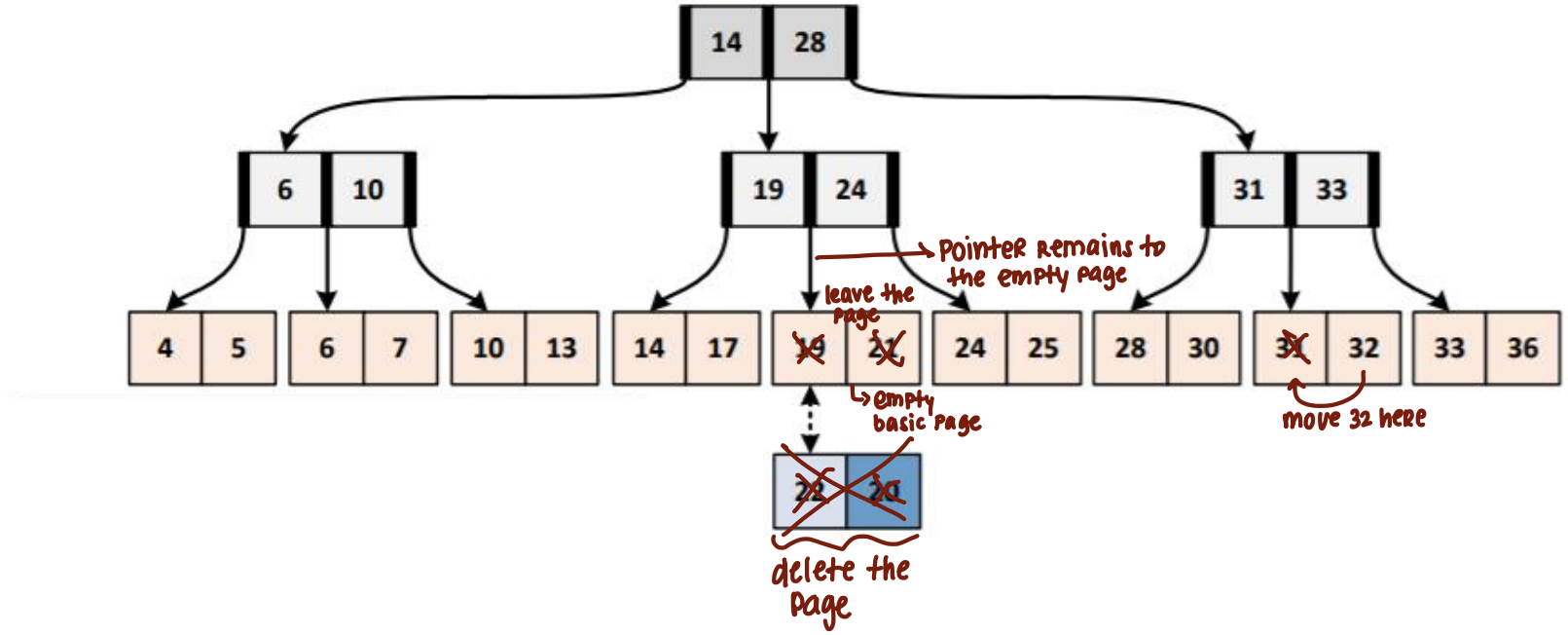


AFTER

Basic page 19, 21 is full, therefore we add the key 20 into existing overflow page. We do not order keys

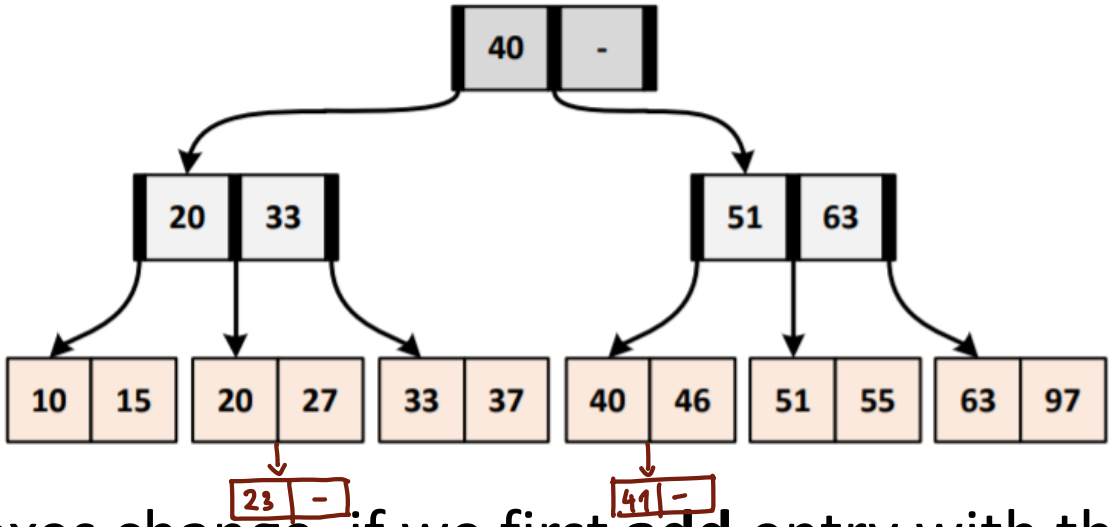
ISAM index – Exercise 3

- **Delete** entries with key **22**, **20, 21, 19, 31**



ISAM and B+ index – Exercise 4

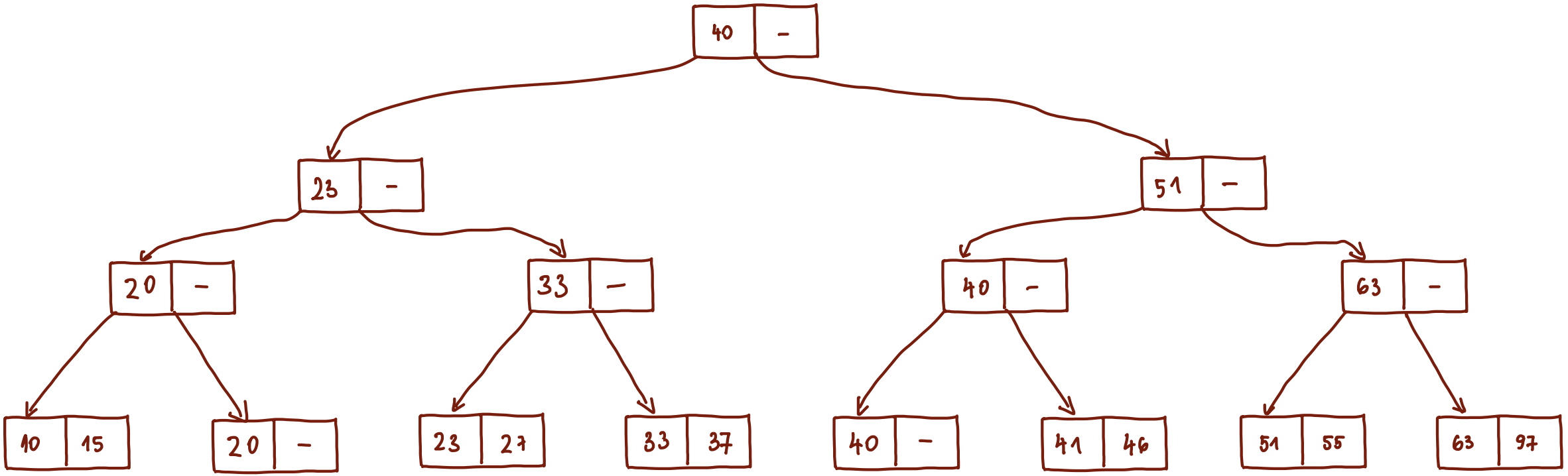
- Two constructed indexes (ISAM and B+) are shown in the figure below (they look the same at the beginning).



ISAM:

- How do the indexes change, if we first **add** entry with the key **41** and after that entry with the key **23**?

B+ tree



Introduction to Database Systems

Exercises: Disks, files and indexes (Part 2)

B+ index - exercise (4)

exam problem

The following schema is given:

5 attributes (6B, >6B, ~90B, 6B, 6B)

Cd(cid, author, title, abbreviation, year)
Member (lid, name, surname, address, email, phone)
Loan (iid, cid, lid, zid, date, loan_term, comentary)
Employees (zid, name, surname, address, email, phone)

Additional informations:

1 page on the disk = 8K

size ↑ ⇒ each attribute is the size of $\frac{400B}{5} = 80B$

|Cd| = 300.000 records, 400 bytes, 20 records/page, 15.000 pages
|Member| = 10.000 records, 200 bytes, 40 zapisov/ page, 250 pages
|Loan| = 300.000 records, 100 bytes, 80 zapisov/ page, 3750 pages
|Employees| = 100 records, 200 bytes, 40 zapisov/ page, 3 pages

Calculate the size of the B + index on the attribute Cd.title. Write down all the assumptions.

↳ we have to calculate the levels the B+ index has and per each level the number of pages (starting from the leaves to the root)

1° assumption: the size of Cd.title

|Cd.title| = 90B → assume based on the size of the attibutes

2° assumption: size of the pointers (on the leaf we have data entries (attribute + pointer = RECORD identifier); same size as page identifier)

|rid| = |pid| = assume it's 6B (it may be given)

↑ RECORD ident. ↑ PAGE ident.

based on 1° & 2° size of data entry => |Data entry| = |Index entry| = |Cd.title| + |rid| = 96B

calculate how many data entries can fit on one page => Number of data entries per page = $\frac{8000 \rightarrow \text{size of 1 page on a disk}}{96 \rightarrow \text{data entry}} = 83.3 \rightarrow 83$ (DEP) (abbrevation)

Number of index entries per page = $\frac{8000}{96} = 83$ (IEP)

Calculate for each level (starting from the leaves) how many pages it has :

leaf => $\frac{300000}{DEF} = 3614.45 \approx 3615$ pages

index level => $\frac{3615}{IEP} = 44$ pages

root level => $\frac{44}{IEP} = 0.\ldots = 1$ page

RESULT

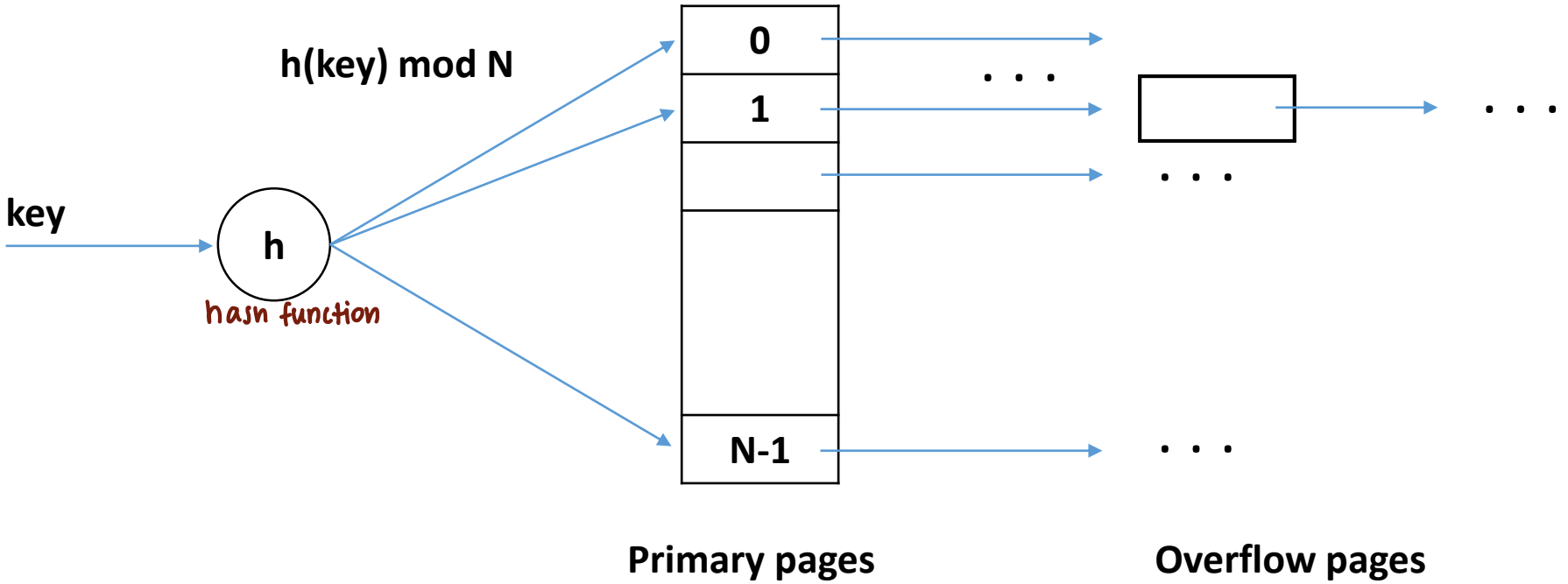
Introduction to Database Systems

Exercises: Hash-based indexing

Hash-based indexing

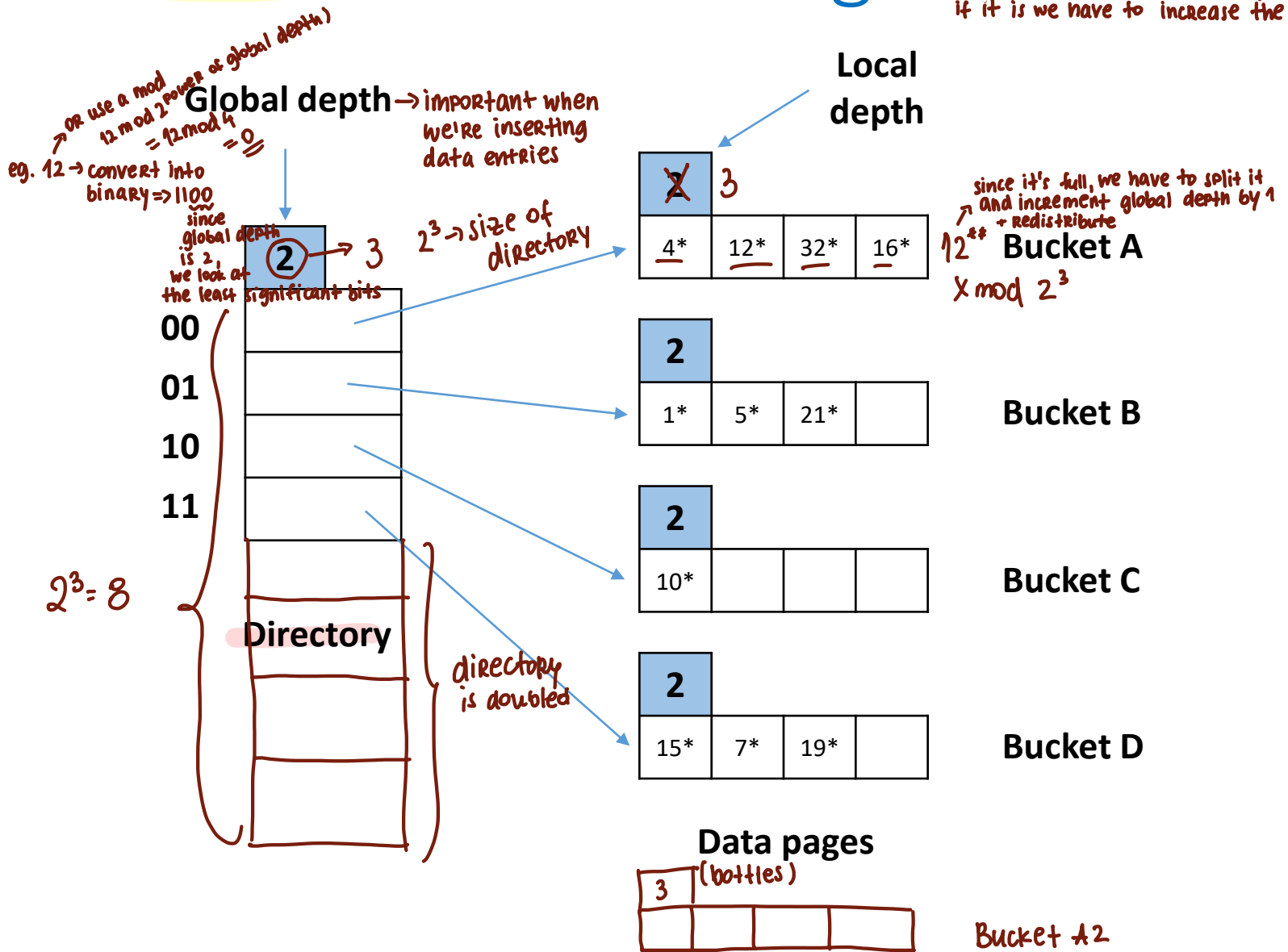
- They are excellent for **equality** selections.
  - We **do not** use it for **range** searches.
- We will have a look at **static** and **dynamic** (extendible) hash-based indexes.
  - Static indexes can lead to long overflow chains.
  - Two solutions to overcome this problem:
    - Extendible Hashing scheme, which doesn‘t use overflow pages,
    - Linear Hashing scheme, which rarely has more than two overflow pages in a chain.

Static hashing → not focusing on this



- The pages containing the data can be viewed as a collection of buckets, with one primary page and possibly additional overflow pages per bucket.
- We apply a hash function *h* to identify to which bucket the key belongs.
- The hash function must distribute values in the domain of the search field uniformly over the collection of buckets (on the interval 0 ... N-1).
  - h(k) = (a * k + b); a and b are constants
- We can get a long chain of overflow pages, resulting in poor performance.

Extendible hashing



- Directory with pointers to buckets
  - Number of buckets is doubled by doubling the directory;
  - Split only the the bucket that overflows;
    - Double directory if necessary.
  - Adjust the hash function if needed;
- Most of the times spliting the bucket does not demand doubling the directory (compare local and global depth);

- How do we know to which bucket the entry belongs to?
  - Binary format of new entry;
  - Last two bits tell us the appropriate bucket.

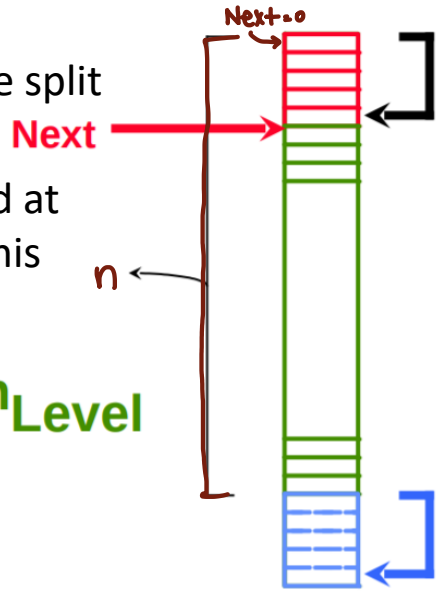
formula for insertion: key mod ($2^{level} \cdot N$) ↗ initial number of buckets at a certain level

formula for redistribution: key mod ($2^{level+1} \cdot N$)

Linear hashing

1st Round level = 0 N = n (# of buckets) Next = 0 (pointer points to the first bucket)

When a bucket is full it creates an overflow page, then the bucket is split and the new bucket is placed below the old one; elements are redistributed, pointer points to the next bucket



Bucket to be split

Buckets that existed at the beginning of this round:
this is the range of h_{Level}

Buckets split in this round:
If h_{Level} (*search key value*) is in this range, must use $h_{Level+1}$ (*search key value*) to decide if entry is in split image bucket

‚split image' buckets: created (through splitting of other buckets) in this round.

- Use family of hash functios h_0, h_1, h_2,...
- Select hash function *h* and number of buckets *N:*
  - $h_i(value)=h(value) mod (2^i N)$ insertion formula
  - d_0 – *number of bits needed to represent N*
  - $d_i=d_0+i$
- Round: Level
  - Use only functions h_{Level} and $h_{Level+1}$
  - Buckets are doubled in every round – one by one.
  - With *Next* we denote the bucket, which will be split next.

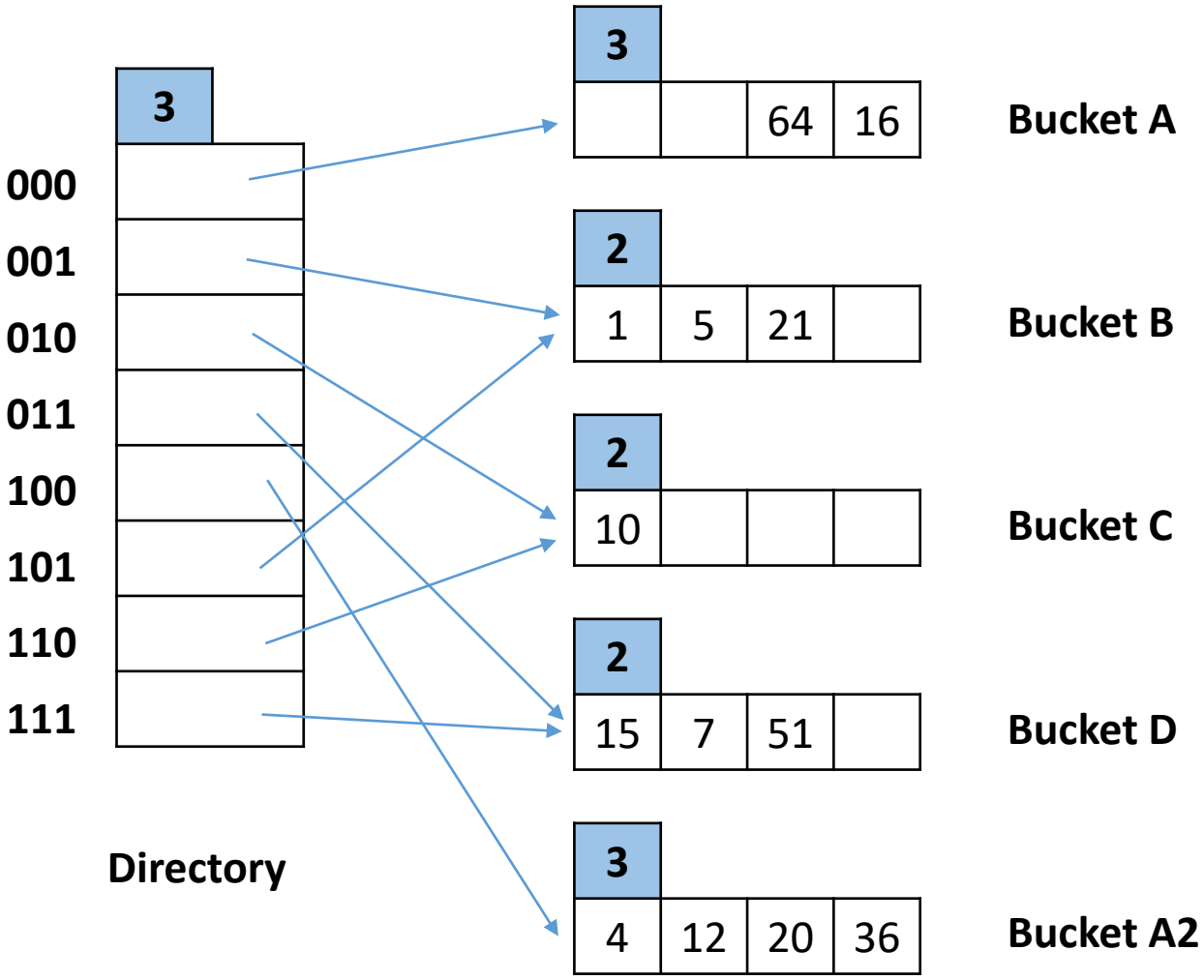
i denen şey = next

Search:
Apply function h_{Level} – if it points to unsplit bucket, look there. If it points to split bucket, check function $h_{Level+1}$

Insert:
If we have to insert the value in the bucket which is full and not split, add overflow pages. Split the next bucket (defined by *Next*) when the overflow page is added.
When *Next* equals N_{Level}*-1* and splitting is triggered, split all the buckets that have not yet been split. *Level=Level+1*

Exercises

Exercise 1.1 – Extendible hash index

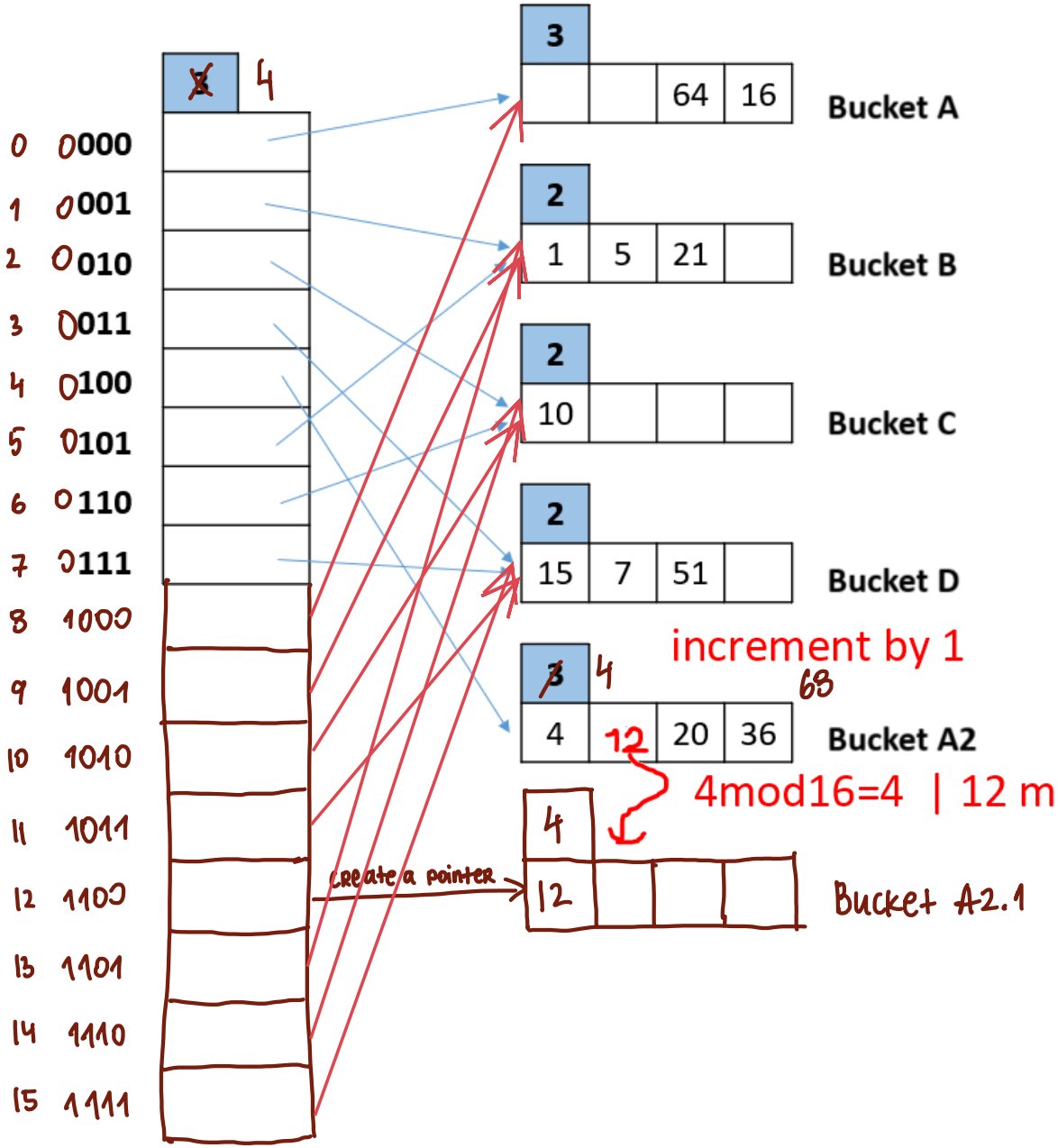


Consider the Extendible Hashing index shown in figure on the left.

Show the index after performing the following operations:

- Insert an entry with hash value 68
- Insert entries with hash values 17 and 69 (use the original index)
- Delete an entry with hash value 21 (use the original index)
- Delete an entry with hash value 10 (use the original index)

a) ADDING 68



increment by 1

68

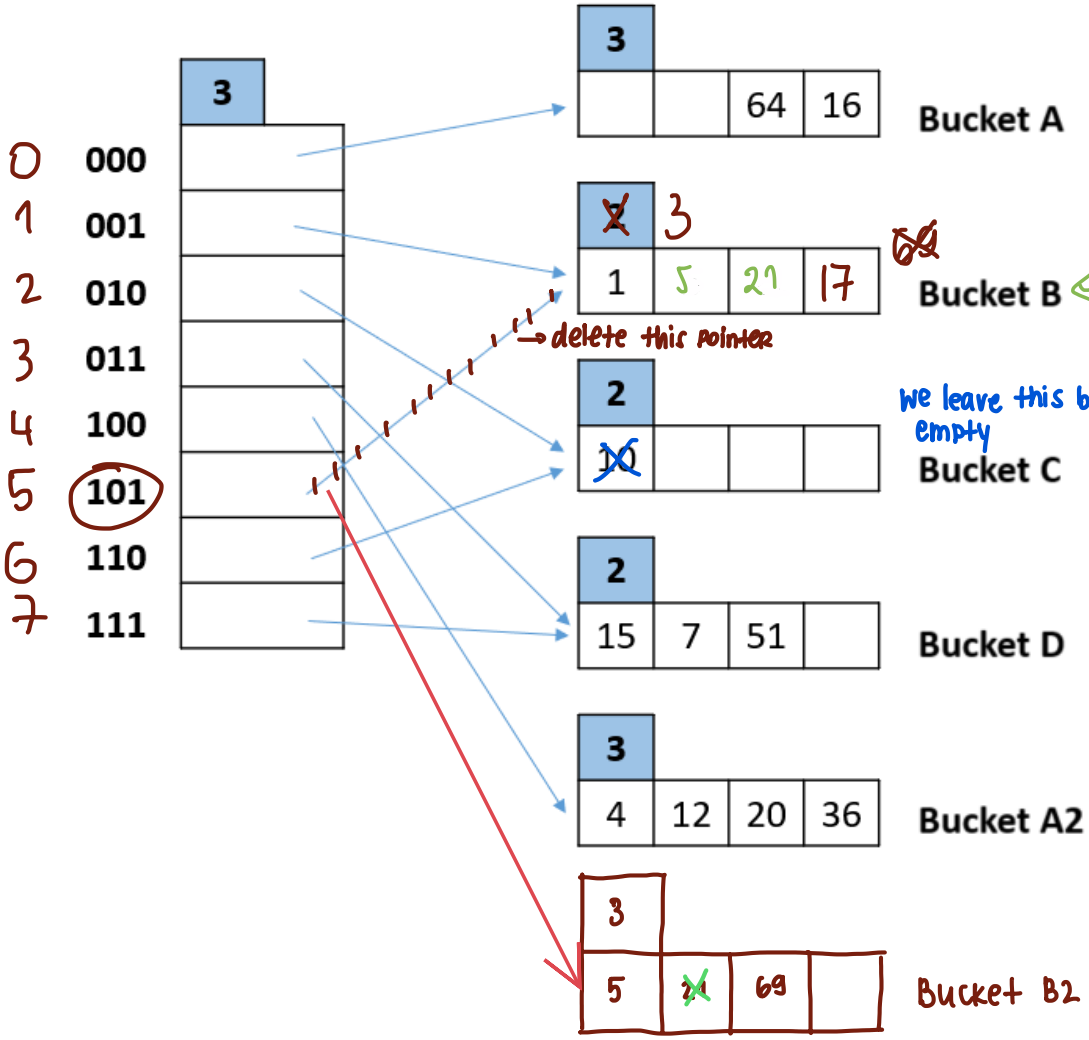
4mod16=4 | 12 mod 16= 12

also new bucket will have 4 .
so we need to compare last 4 bits.
(least significant)

a) • Insert an entry with hash value 68
b) • Insert entries with hash values 17 and 69 (use the original index)
c) • Delete an entry with hash value 21 (use the original index)
d) • Delete an entry with hash value 10 (use the original index)

a) $68 \bmod 2^3 = 68 \bmod 8 = 4$

Redistribute: $4 \bmod 2^4 = 4$
$12 \bmod 2^4 = 12$
$20 \bmod 2^4 = 4$
$36 \bmod 2^4 = 4$
$68 \bmod 2^4 = 4$



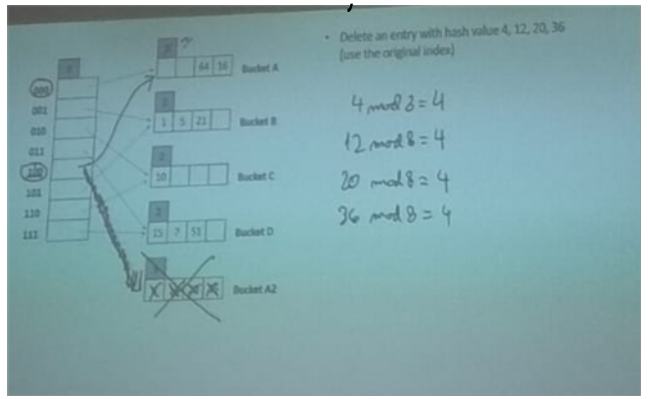
- Insert an entry with hash value 68
- Insert entries with hash values 17 and 69 (use the original index)
- Delete an entry with hash value 21 (use the original index)
- Delete an entry with hash value 10 (use the original index)

b) $17 \bmod 2^3 = 1$ → performed over the original example
$69 \bmod 2^3 = 5$

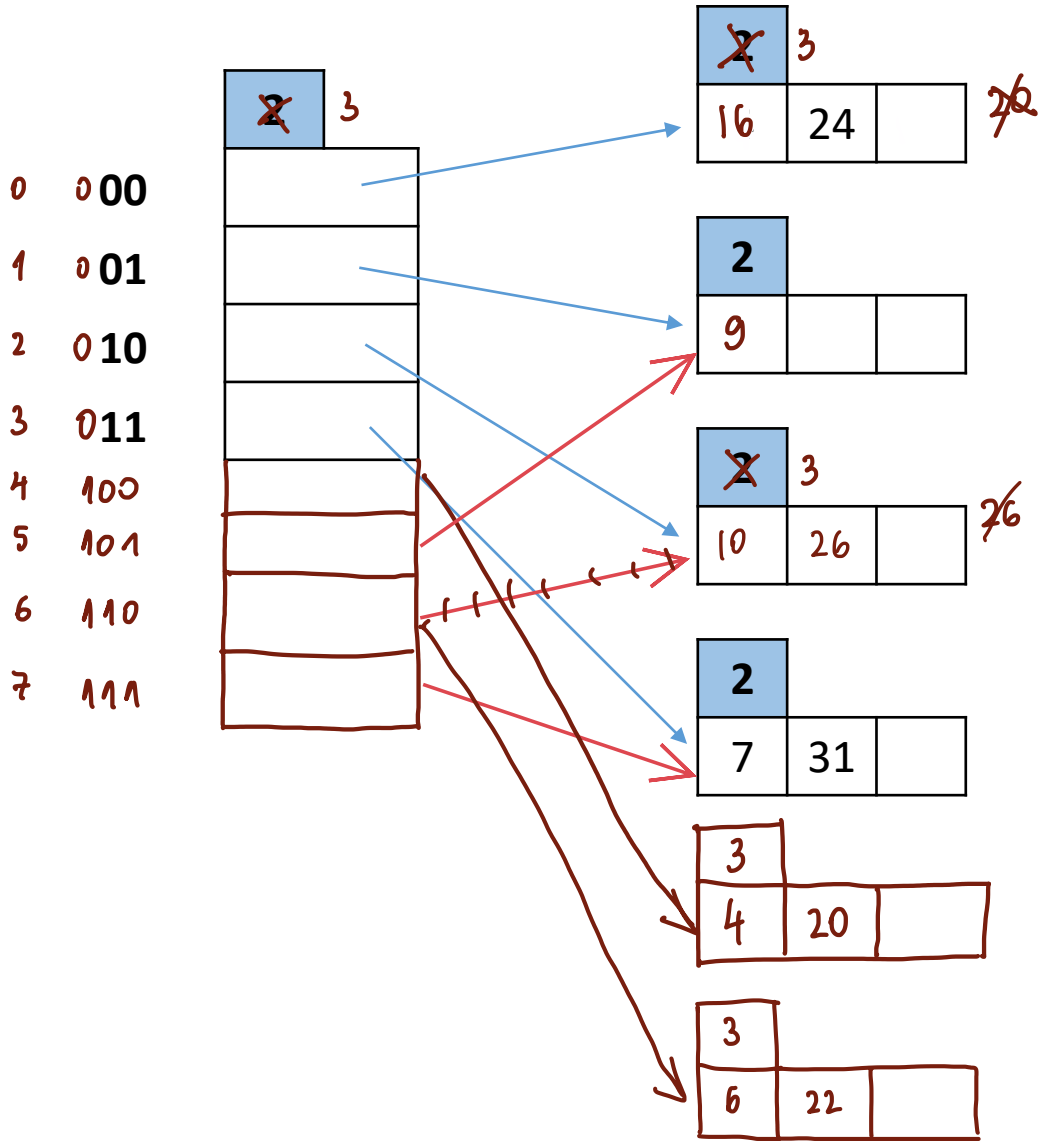
Redistribution: $1 \bmod 2^3 = 1$
Redistribute what was in this bucket
$5 \bmod 2^3 = 5$
$21 \bmod 2^3 = 5$ → new bucket
$17 \bmod 2^3 = 1$
$69 \bmod 2^3 = 5$

- use the formula to find the value $21 \bmod 8 = 5$
- locate 10 ⇒ $10 \bmod 8 = 2$

to locate a # ⇒ _ mod $2^{\text{global depth}}$ ↓ 23



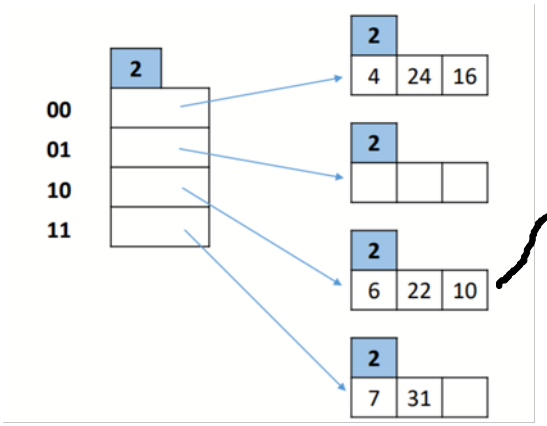
Exercise 1.2 – Extendible hash index



- Insert an entry with hash value 9, 20 and 26

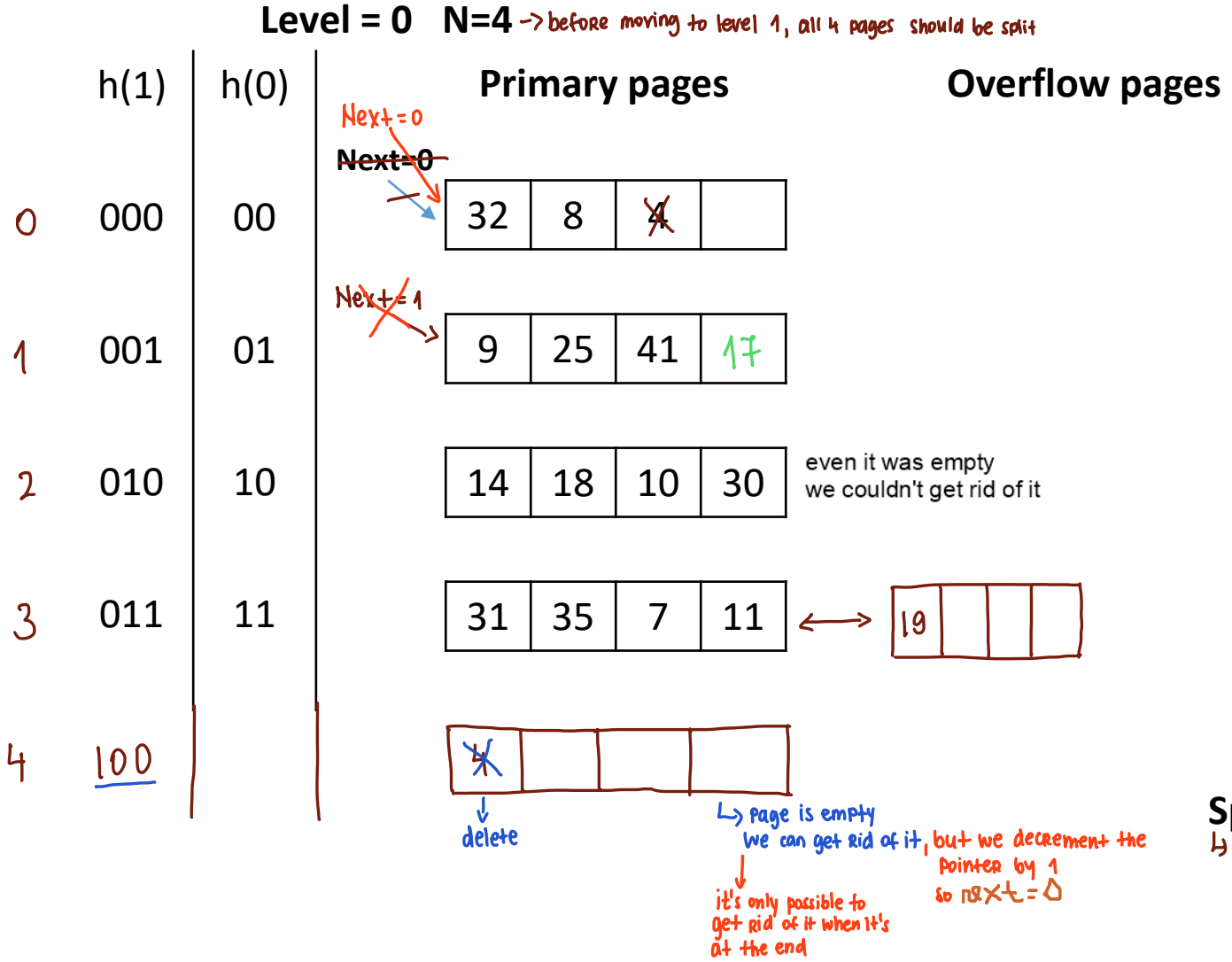
- $9 \bmod 2^2 = 1$
- $20 \bmod 2^2 = 0$
- Redistribution: $4 \bmod 2^3 = 4$, $24 \bmod 2^3 = 0$, $16 \bmod 2^3 = 0$, $20 \bmod 2^3 = 4$
- $26 \bmod 2^2 = 2$
- Redistribution: $6 \bmod 2^3 = 6$, $22 \bmod 2^3 = 6$, $10 \bmod 2^3 = 2$, $26 \bmod 2^3 = 2$

en başta 26yı c'ye koy. ama dolu. bu yüzden restribute.



başta böyleydi

Exercise 2.1 – Linear hashing index



Consider the Linear Hashing index shown in figure on the left. Assume that we split whenever an overflow page is created.

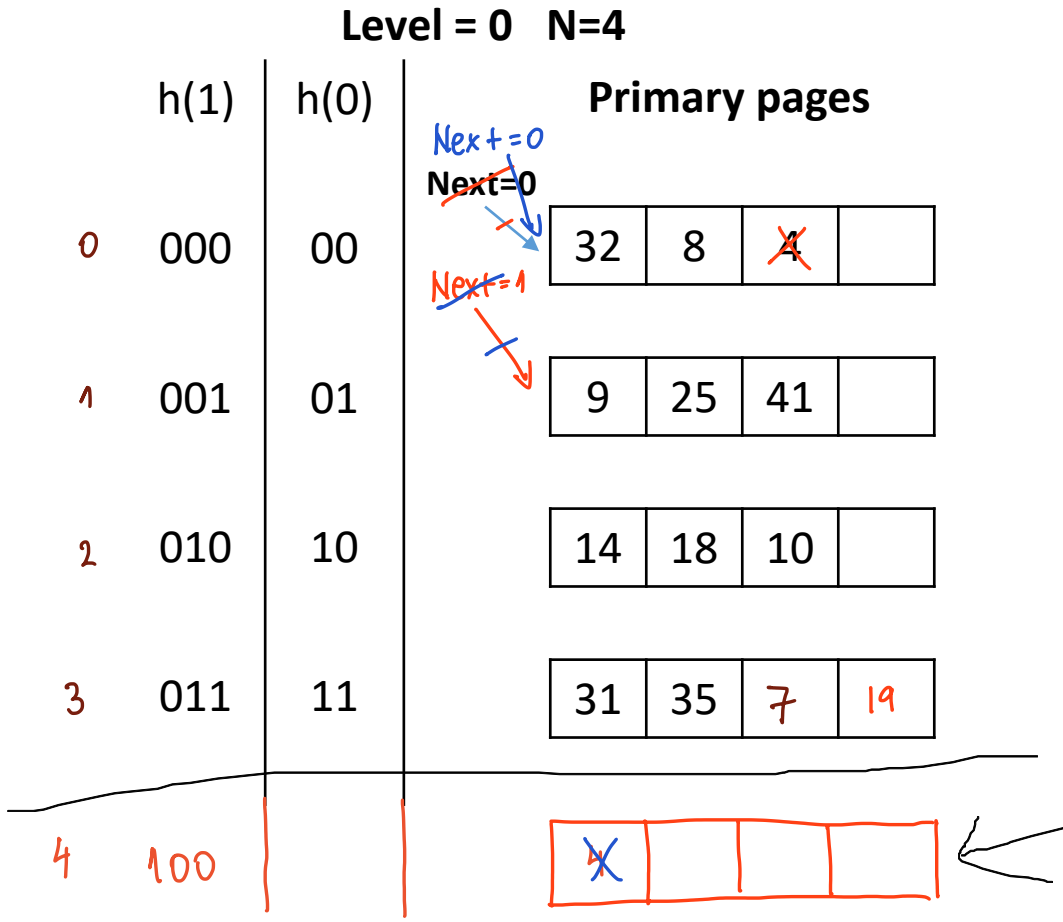
Show the index after performing the following operations:

- Insert an entry with hash value 19
- Insert an entry with hash value 17
- Delete the entry with hash value 4

- insert 19: $19 \bmod (2^0 \cdot 4) = 19 \bmod 4 = 3$
  redistribute: $32 \bmod (2^{0+1} \cdot 4) = 0$, $8 \bmod (2^1 \cdot 4) = 0$, $4 \bmod (2^1 \cdot 4) = 4$ → remain as they are
- insert 17: $17 \bmod (2^0 \cdot 4) = 1$
- delete 4: $4 \bmod (2^0 \cdot 4) = 0$ → can't locate it ⇒ check if $0 <$ Next=1 (pointer) ⇒ $4 \bmod (2^1 \cdot 4) = 4$
  ↳ true
  1. find 4

Split condition → when the overflow page is added
↳ when it's fullfilled, split the page with the pointer; increment the pointer

Exercise 2.2 – Linear hashing index



Consider the Linear Hashing index shown in figure on the left.
Show the index after performing the following operations:

- Insert an entry with hash value 7
- Insert an entry with hash value 19
- Delete the entry with hash value 4

- insert 7: $7 \bmod (2^0 \cdot 4) = 3 \Rightarrow$ occupancy = 75%
- insert 19: $19 \bmod (2^0 \cdot 4) = 3 \Rightarrow$ occupancy = 81%

13/(4.4) = 0.81

Redistribution: $32 \bmod (2^1 \cdot 4) = 0$, $8 \bmod (2^1 \cdot 4) = 0$ > remain, $4 \bmod (2^1 \cdot 4) = 4$

- delete 4: $4 \bmod (2^0 \cdot 4) = 0 \Rightarrow 0 < \text{Next} \Rightarrow 4 \bmod (2^1 \cdot 4) = 4$

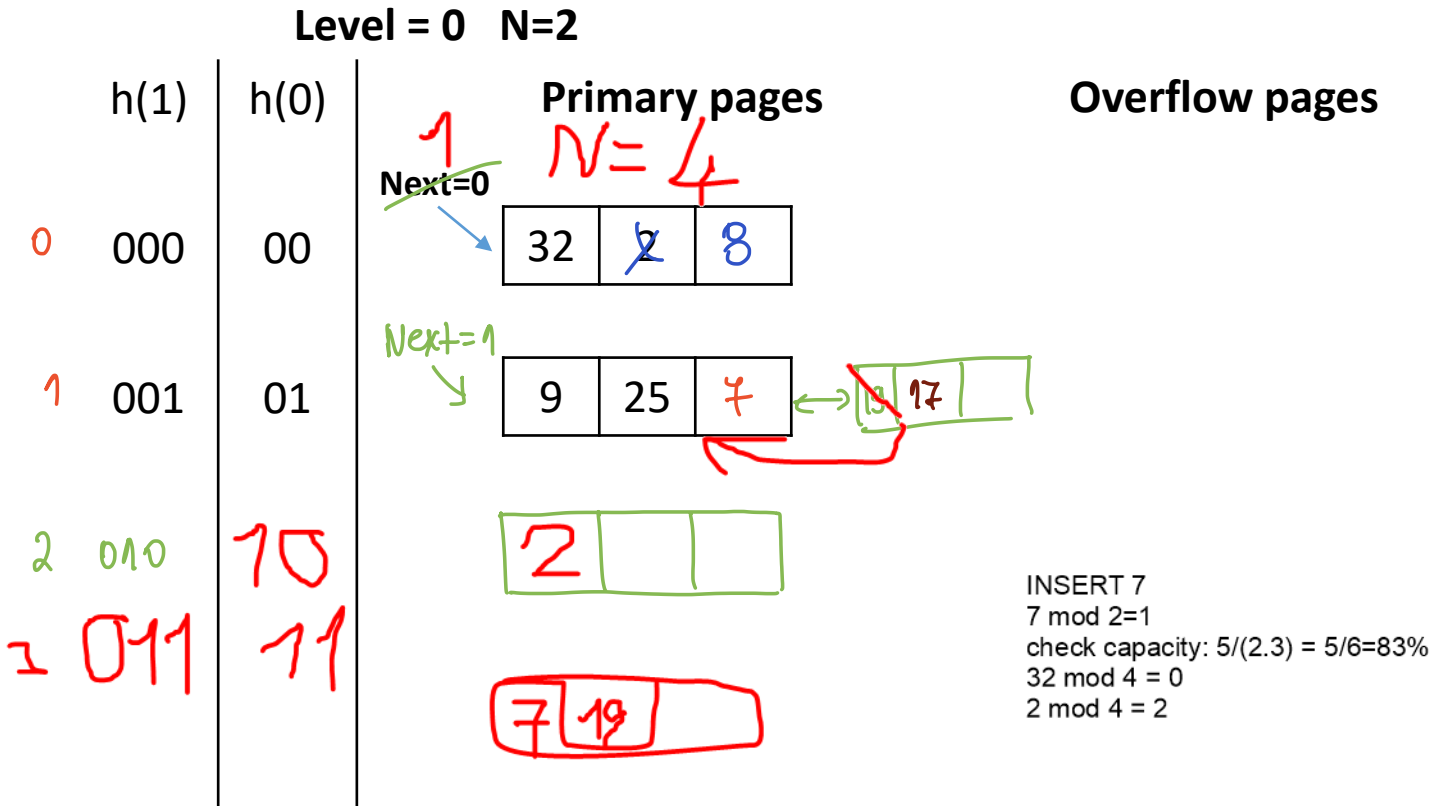
We don't take into account overflow pages

Split condition → more than 75% occupancy

Number of items/ (number of buckets * bucket capacity)

$\frac{11}{4 \cdot 4} = \frac{11}{16} = 0.68 = 68\%$

Exercise 2.3 – Linear hashing index



INSERT 7
7 mod 2=1
check capacity: 5/(2.3) = 5/6=83%
32 mod 4 = 0
2 mod 4 = 2

INSERT 19:
19 mod 2=1 overflow olarak eklendi.
check capacity: 6.
6/3.3 = 6/9 = 66%

INSERT 17:
17mod2=1
check capacity = 7
7/3.3=77% (we have to split)

Consider the Linear Hashing index shown in figure on the left.
Show the index after performing the following operations:

- Insert an entry with hash value 7
- Insert an entry with hash value 19
- Insert an entry with hash value 8
- Delete the entry with hash value 2

Split condition → more than 70% occupancy

Number of items/ (number of buckets * bucket capacity)

$\frac{4}{12} = 0.33$

OTHER EXAMPLE

32	8	4	
9	25	41	
14	18	10	
31	35	7	19

Consider the Linear Hashing index shown in figure on the left.
Show the index after performing the following operations:

- Insert an entry with hash value 7
- Insert an entry with hash value 19
- Insert an entry with hash value 17

insert 7

7 mod (2⁰·2) = 1

insert 19

19 mod 2 = 1

Redistribute:

9 mod (2¹·2) = 1

25 mod (2¹·2) = 1

7 mod (2¹·2) = 1

insert 8:

8 mod (2⁰·2) = 0

delete 2

2 mod (2⁰·2) = 0

insert 17

17 mod (2⁰·2) = 1

32	8		
9	25	41	17
14	18	10	
31	35	7	19

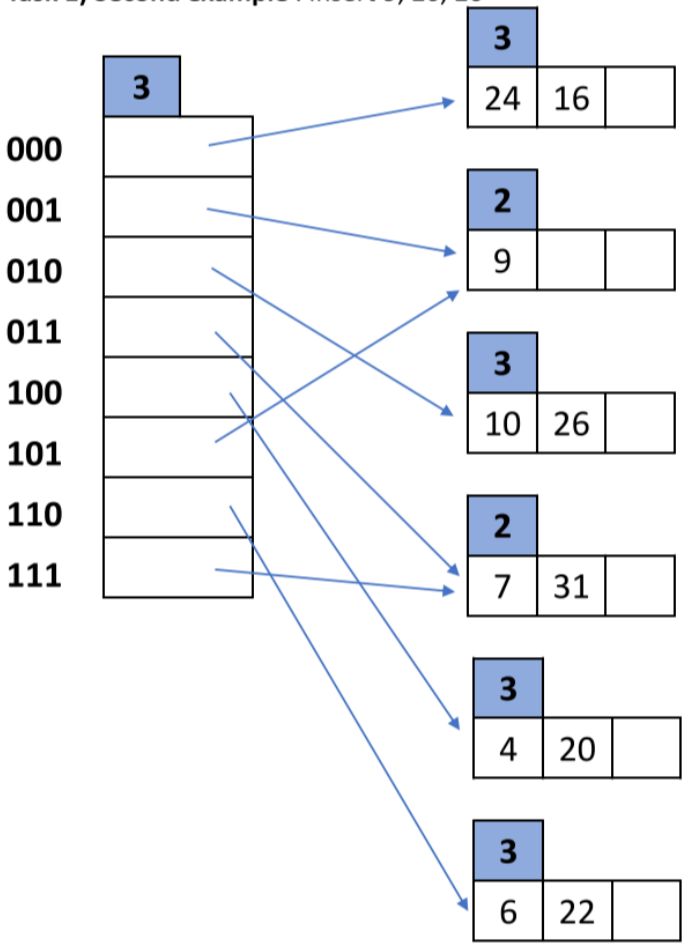
100

4			

17 % 4 = 1

capacity: 14/(5.4) < %75

Task 1) Second example : Insert 9, 20, 26



Introduction to database systems

Exercises: query evaluation and optimization

Task

we calculate how many pages are read and written to the disk

- estimate execution plan of SQL queries:

  we are counting the number of transfered pages from/to the disk (from a dynamic memory)

- we can have several different execution plans for each query
- If fe understand the plan, we can choose the optimal solution

What is query execution plan

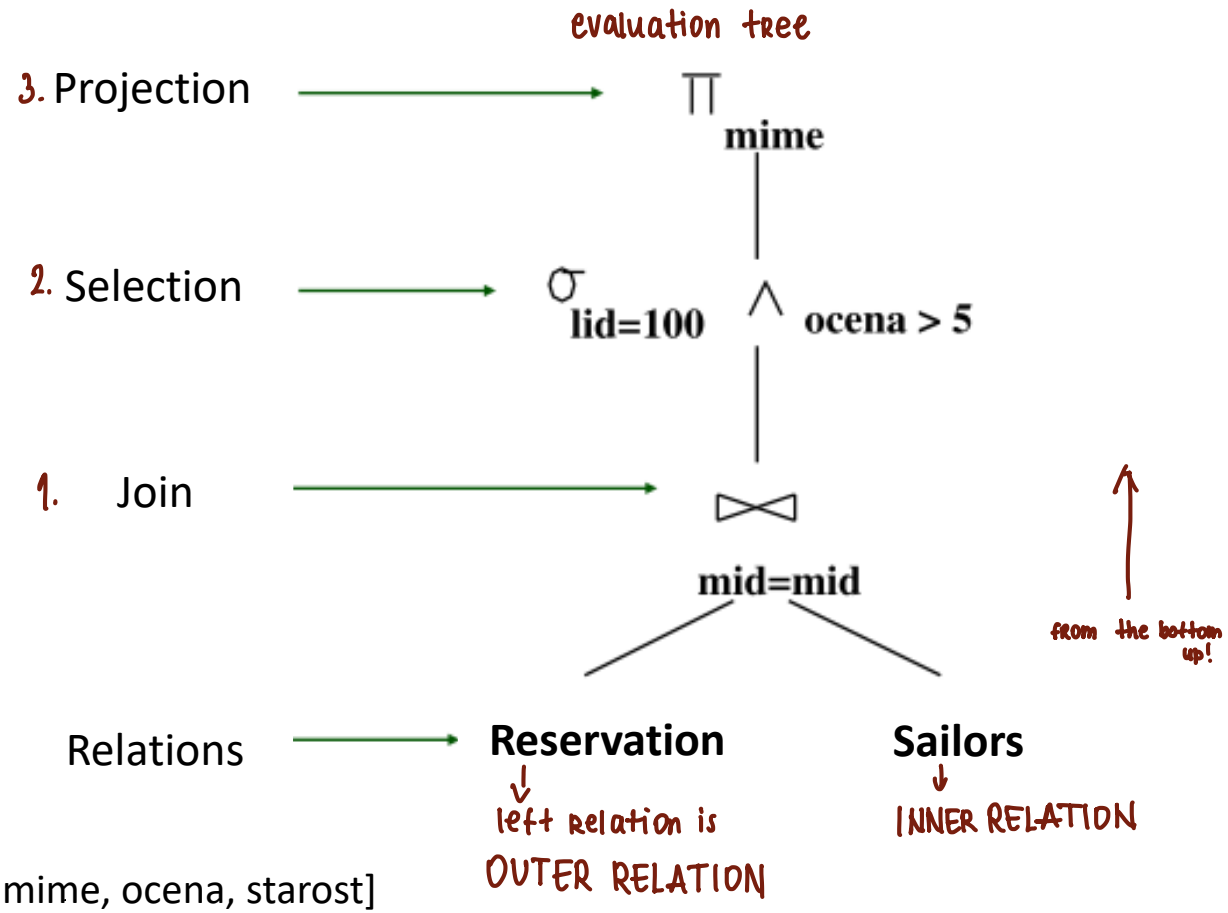
```
SELECT    M.ime
FROM      Reservation R, Sailors M
WHERE     R.mid = M.mid
          AND R.lid = 100 AND M.grade > 5
```

In relational algebra:

$\pi_{mime}(\sigma_{lid\ =\ 100\ \wedge\ grade\ >\ 5}(Reservation \bowtie_{mid\ =\ mid} Sailors))$

↑ block nested loop join

Cost: M + M * N = 1000 + 1000 * 500 = 501 000 pages



Reservation [mid, bid, dan, rime]
entry size 40 bytes
prm = page size 100 entries
M = 1000 pages

Sailors [mid, mime, ocena, starost]
entry size 50 bytes
prn = page size 80 entries
N = 500 pages

Formulas

Records per page

1. **Simple Nested Loops Join**: M + M*prm*N (M – outer relation, N- inner relation)
   N + N*prn*M (N – outer relation, M- inner relation)
   - For each tuple in the outer relation R1, we scan the entire inner relation R2.

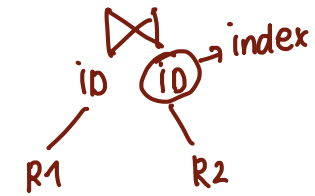
1.1 **Block Nested Loops Join**: M+M*N
   - For each page of M , get each page of N , and write out matching pairs of tuples
   - If we have enough space in the memory for N+2 pages, where N is smaller relation: M+N
   - If there is not enough space (lets say B=102):
     M+N*(M/(B-2)) = 1000+500*1000/100=6000

We join relations only on hash indexes

1.2 **Index Nested Loops Join** (if we have hash index on one of the join attributes, we put relation with index on the inner part of the join)

(right part)

Example for hash index: N+M*prn*1,2 M+N*prm*1,2



Formulas

2. Sort-Merge Join: M and N (example: B=100) (buffer size; pages)

(sort relation M; sort N; join them together)

$$2*|M|*(1+\log_{B-1}(|M|/B)) + 2*|N|*(1+\log_{B-1}(|N|/B)) + |M| + |N| =$$

$\sim \log_B(|M|)$

$$2*1000*(1+1) + 2*500*(1+1) + 1000 + 500 = 7500$$

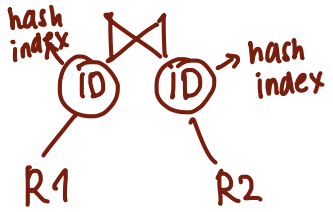
$$\log_{B-1}\left(\frac{|M|}{B}\right) = \frac{\log_{10}\frac{M}{B}}{\log_{10}(B-1)}$$

Formulas

3. Hash Join (hash index on both join attributes)

Both relations with hash index we split to B-1 partitions

- 3*(M+N), if B > √N (N is smaller relation)
- If not, we have the example of excesses and each partition of the smaller relationships, that doesn‘t fit into the memory is recursively divided.



hash — 80% full

the cost of creating an index: $1.5 \cdot 0.1 \cdot$ number of pages of T1

B+ : $1.2 \cdot 0.1 \cdot$ number of pages of T1 — 67% full

Exercises

Schema:

PILOTS (IdP int(11), Surname varchar(50), Name varchar(50), Emso varchar(13), Address varchar(250), EmploymentDate date, YearsOfExperience varchar(20), NoOfFlights int(11), NoOfHoursPiloting int(11), IdLD int(11))

FLIGHTS (IdLT int(11), IdP int(11), IdL int(11), IdLEFlight int(11), FlightDate date, FlightTime time, IdLELand int(11), LandDate date, LandTime time)

RESERVATIONS (IdR int(11), Number int(11), Price float, Surname varchar(50), Name varchar(50), Address varchar(250), Telephone varchar(20), SeatNo int(11), Class int(11), IdLT int(11))

Take the following information into account:

PILOTS: 10 records per page, 40 pages, each record represents a single pilot, relation stores 100 different values for NoOfHoursPiloting and the pilots are evenly distributed among these values, at the same time there are 70 different values over 500.

Always assume uniform distribution

FLIGHTS: 20 records per page, 150 pages

RESERVATIONS: 50 records per page, 30000 pages, there are 150 different reservation prices and 70 of those are higher than 80.000.

For each query, we have 10,000 pages in the buffer (SUBP).

Exercise 1

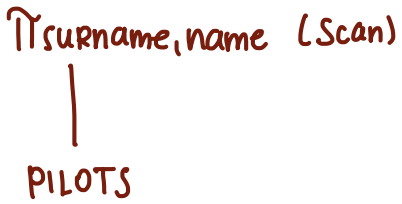
Build a query execution plan for each query and give a cost estimation of the execution plan:

a)
```
SELECT Surname, Name
FROM PILOTS
```

b)
```
SELECT Surname, Name
FROM Pilots
WHERE NoOfHoursPiloting>500
```

(on the fly) - during the running of a computer program without interrupting the run.

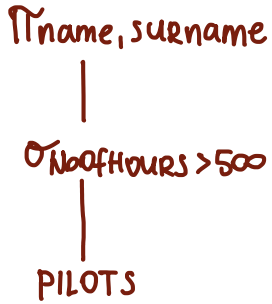
a) RA: $\Pi_{surname, name}$ (PILOTS)



we only have to scan the pilots

COST: 40 pages

b)



- there are no indexes, so the cost won't be reduced
- we scan the pilots and go on-the fly (we have to scan the whole relation and compare where the condition is satisfied)

COST: 40 pages

Exercise 2

What is the cost of the execution plan for the second query in the previous case if it uses a clustered tree index (height 2) on the attribute NoOfHoursPiloting?

- we don't read the entire relation

COST: B+ index (height) → 2 pages
(↗ we have to read 2 pages to reach the records)

- how many pilots satisfied the condition?
  - ↳ total number of pilots: 10 records/page · 40 pages = 400 records = 400 pilots
  - ↳ how many pilots fit per each different value of NoHoursPiloting: $\frac{400}{100} = 4$ pilots → since they are evenly distributed
  - ↳ number of pilots where condition is satisfied => 4·70 = 280 pilots
- Number of pages: $\frac{280}{10}$ + 2 pages = 30 pages FINAL COST
  (10 ↗ records/page)

SELECT Surname, Name
FROM Pilots
WHERE NoOfHoursPiloting>500

PILOTS: 10 records per page, 40 pages, each record represents a single pilot, relation stores 100 different values for NoOfHoursPiloting and the pilots are evenly distributed among these values, at the same time there are 70 different values over 500.

(if not given, we assume e.g. 10/.)

Exercise 3

On the exam

PILOTS: 10 records per page, 40 pages, each record represents a single pilot, relation stores 100 different values for NoOfHoursPiloting and the pilots are evenly distributed among these values, at the same time there are 70 different values over 500.

FLIGHTS: 20 records per page, 150 pages

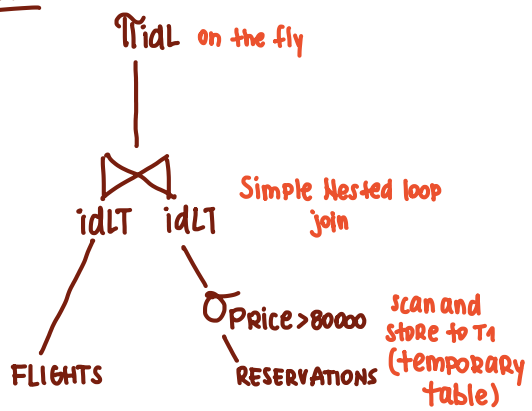
RESERVATIONS: 50 records per page, 30000 pages, there are 150 different reservation prices and 70 of those are higher than 80.000.

The following algorithms for the imlementation of joins are available: Sort-Merge Join and Simple Nested Loops Join.

a) Evaluate the implementation plan: Find all IdL (plane ids), where the price for reservation of their flights is higher than 80,000.

- assume were optimizing the plan before beginning the evaluation

PLAN 1:



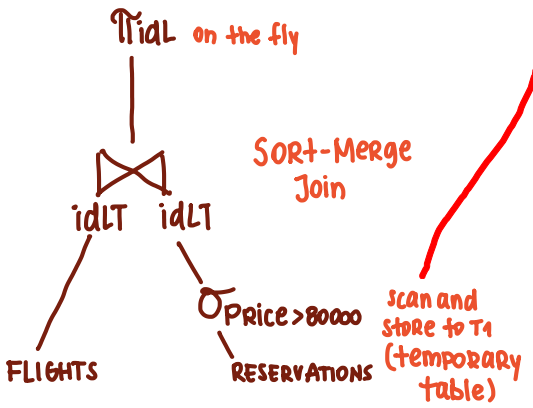
- scan RESERVATIONS (no indexes → go through the whole relation)
  ↳ 30000 pages
- T1:
  total number of reservations: 50 · 30000 = 1.500000 RESERVATIONS (50: Rec. per page)
  assume even distribution: $\frac{1500000}{150}$ = 10000 RESERVATIONS PER different PRICE (150: # of different RESERVATION PRICES)
  - number of PRICES > 80000 = 70 · 10000 = 700000
  - number of pages for RESERVATIONS: $\frac{700000}{50}$ = 14000 pages (50: RPP) → stored in T1
- Simple Nested Loop Join:
  FLIGHTS + FLIGHTS · RPP · T1 = 150 + 150 · 20 · 14000 = 42000150 pages (150: pages; 14000: pages)
- TOTAL COST: 42000150 + 14000 + 30000 = 42044150 pages

1.1 SORT MERGE JOIN
SORT FLIGHTS: 2.150.(1+log_9999(150/10000) = 300

SORT RESERVATIONS:
2.30000.(1+log_9999(30k/10k)=120k pages

MERGE FLIGHTS AND RESERVATIONS: 150+30000 = 30150 PAGES

PLAN 2:



- scan RESERVATIONS : 30000 pages

T1: 14000 pages

- SORT-MERGE JOIN: assume they're not sorted

$\log_{B-1} \frac{|M|}{B} = \frac{\log_{10} \frac{|M|}{B}}{\log_{10} B-1}$ B=buffer = 10000 pages

SORT flights: $2|\text{FLIGHTS}| \cdot (1+\log_{B-1}|M|/B) =$

$2 \cdot 150 \cdot \left(1+\frac{\log_{10}\left(\frac{150}{10000}\right)}{\log_{10}(10000-1)}\right) = 300 \cdot (1+0) =$ 300 pages

EXAMPLE OF TEACHER 10-> 9999

- SORT T1: $2 \cdot 14000 \cdot \left(1+\frac{\log_{10}\frac{14000}{10000}}{\log_{10}(10000-1)}\right) = 28000 \cdot (1+1) =$ 56000 pages

- merge FLIGHTS & T1 : 150 + 14000 = 14150 pages

TOTAL COST: 30000 + 14000 + 300 + 56000 + 14150 = 114450 pages

↓

better choice

Exercise 3

PILOTS: 10 records per page, 40 pages, each record represents a single pilot, relation stores 100 different values for NoOfHoursPiloting and the pilots are evenly distributed among these values, at the same time there are 70 different values over 500.

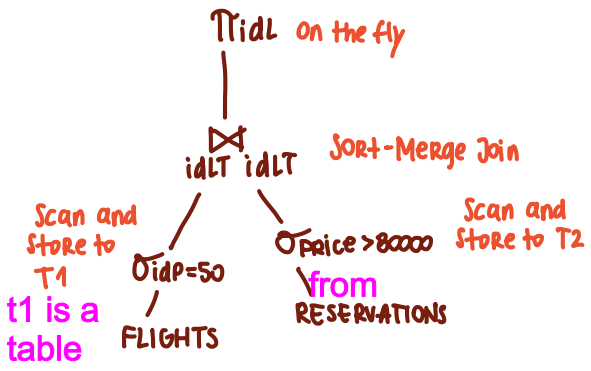
FLIGHTS: 20 records per page, 150 pages

RESERVATIONS: 50 records per page, 30000 pages, there are 150 different reservation prices and 70 of those are higher than 80.000.

The following algorithms for the imlementation of joins are available: Sort-Merge Join and Simple Nested Loops Join.

b) Evaluate the implementation plan: Find all IdL (plane ids), where the price for reservation of its flights pri is higher than 80,000 and was piloted by a pilot with id IdP=50.

PLAN 1:



- Scan RESERVATIONS: 30000 pages
- T2: 14000 pages (T1 from part a)
- Scan flights: 150 pages
- T1:
  - ↳ total number of flights: 20 · 150 = 3000 RECORDS (rpr = records per page)
  - ↳ number of flights per pilot : $\frac{3000}{400}$ = 7 records
    - 10 · 40 → pages
    - ↑ Records / Page
  - ↳ number of pages: $\frac{7}{20}$ = 1 page

- SORT-Merge Join:
  - ↳ SORT T2: 56000 pages (from part a)
  - ↳ SORT T1: 2 · 1 · (1 + 0) = 2 pages (because m=1)
- merge T1 ⋈ T2 = 14000 + 1 = 14001 pages

TOTAL COST: 30000 + 14000 + 150 + 1 + 56000 + 2 + 14001 = 114154 pages

PLAN 2: SIMPLE NESTED LOOPS JOIN (OUTER = T2)
1+1*20*14000 = 280001 PAGES
COST:
30K+14K+150+1+280001 = 324152 PAGES

In the exam, you gonna eliminate the worst and select the best

Exercise 4

PILOTS: 10 records per page, 40 pages, each record represents a single pilot, relation stores 100 different values for NoOfHoursPiloting and the pilots are evenly distributed among these values, at the same time there are 70 different values over 500.

FLIGHTS: 20 records per page, 150 pages

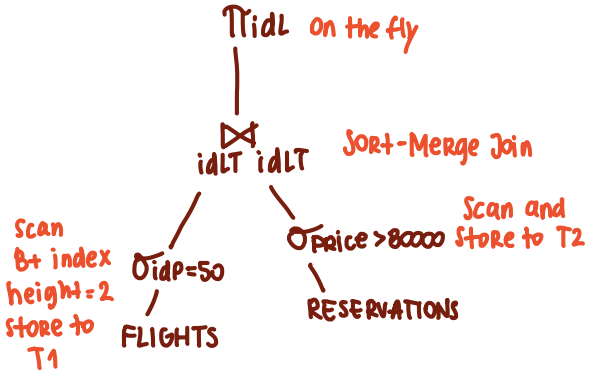
RESERVATIONS: 50 records per page, 30000 pages, there are 150 different reservation prices and 70 of those are higher than 80.000.

For how much is the cost of the execution plan for the second query in the previous exercise different if it uses in its execution:

- 2-level tree index on IdP in Flight relation and
- ~~Hash index on Price~~ and hash index on IdLT in Reservations relation → hash index can't be used because it's used for looking up equalities only
  ↳ since T2 doesn't have any indexes, we can't use this assumption that's why σ has to be above the join => PLAN 2

pointer+int =10 (exam)

PLAN 1:



We can use hash, because there is equals sign.

- scan flights: B+ index => 2 pages
- T1: 1 page (from b)
- scan reservations: 30000 pages
- T2: 14000 pages (from b)
- Sort-Merge Join:
  ↳ sort T1: 2 pages
  ↳ sort T2: 56000 pages
  merge T1 & T2: 14001 pages

from old example

- T1:
  ↳ total number of flights: 20·150 = 3000 records (pages)
  ↳ number of flights per pilot : 3000/400 = 7 records
  (10·40 → pages; records/page)
  ↳ number of pages: 7/20 = 1 page

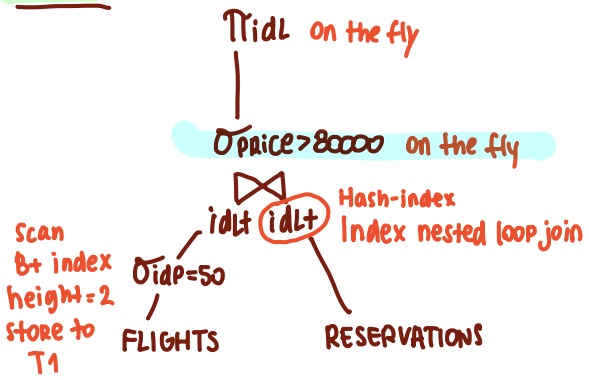
TOTAL COST: 2+1+30000+14000+2+56000+1 = 114006 pages

PILOTS: 10 records per page, 40 pages, each record represents a single pilot, relation stores 100 different values for NoOfHoursPiloting and the pilots are evenly distributed among these values, at the same time there are 70 different values over 500.

FLIGHTS: 20 records per page, 150 pages

RESERVATIONS: 50 records per page, 30000 pages, there are 150 different reservation prices and 70 of those are higher than 80.000.

PLAN 2:



- scan flights => 2 pages (B+ index)
- T1: 1 page
- Index nested loops join:

$7 \cdot (1.2 + \underline{\quad})$ → for ordered index: $7 \cdot (1.2 + 10) = 84$ pages

number of records per pilot

↑ pages for reservations per a single flight

total number of reservations: $50 \cdot 30000 = 150000$ reservations

number of flights: $20 \cdot 150 = 3000$ flights

↳ number of reservations per single flight: $\frac{1500000}{3000} = 500$ reservations per flight

↳ unordered index: each reservation takes 1 page => 500 pages

↳ assume we have an ordered index => $\frac{500}{50} = 10$ pages

left (outer side) = T1 => pushes above join 1 page →

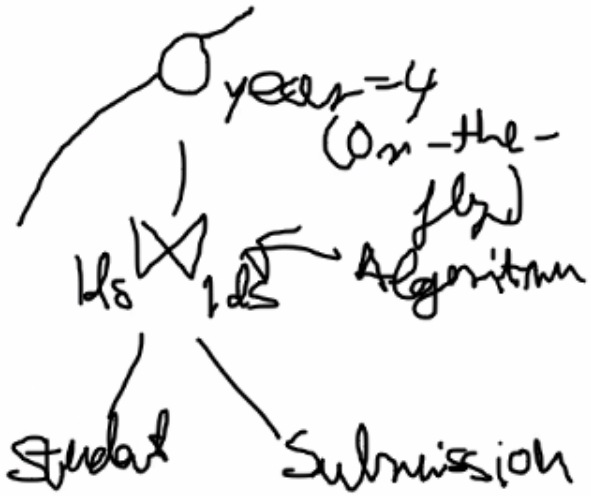
Right (inner side) => $\frac{\text{flights}}{\text{Pilot}} \cdot \frac{\text{reserv.}}{\text{flights}} = \frac{\text{Reservations}}{\text{pilot}}$

↳ $7 \cdot 500 = 3500 \frac{\text{Reservations}}{\text{pilot}}$ => $\frac{3500}{50} = 70$ pages

↑ records per page

TOTAL COST: $2+1+84+1+70 = 158$ pages

Query evaluation



The database presented in the first task has pages of size 4,000 B. (Buffer 8K)

|student| = 5.000 records, 110B, 35 records/page, 145 pages
|submission| = 50.000 records, 30B, 130 records/page, 385 pages
|task| = 500 records, 110B, 35 records/page, 5 pages

How many pages reads the following query? Assume no indexes are available.

Available algoritms: Page – oriented Nested Loops Join and Block Nested Loops Join

select[year=4](join[student.ids=submission.ids](student, submission))

Page oriented Nested Loops join

145 + 145·35 · 385 = ____

pages tuple

Block Nested loops Join

B-2 > M

7998 > 145 ✓

N+M = 145 + 385 = 530 pages

Introduction to Database Systems

Exercise: Transactions (Part 1)

Properties (ACID)

1. **Atomicity**
   The transaction is either executed or not.

2. **Consistency**
   The database has to be consistent before and after the transaction

3. **Isolation**
   Concurrently executed transactions do not affect each other.

4. **Durability**
   Transactions are durable upon confirmation

Types of transaction conflicts (concurrent execution of transactions)

WR – Write-Read conflict
Transaction T2 read an object that has been modified by another transaction T1 (T1 has not been committed yet)

RW – Read-Write conflict
Transaction T2 changed the value of an object that has been read by another transaction T1 (T1 has not been commited yet).

WW – Write-Write conflict
Transaction T2 overwrote the value of an object which has already been modified by another transaction T1 (T1 has not been committed yet)

Read-read conflict doesn't exist because it doesn't affect anything

Strict Two-Phase Locking

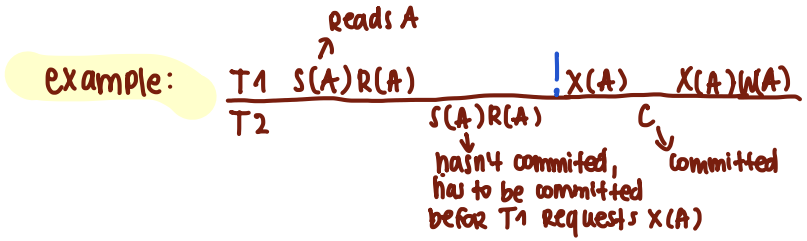
1. If a transaction T wants to read (or/and modify) an object, it first requests a shared S(A) or exclusive X(A) lock on the object.

Request when we want to read from A

Request if we want to write on A

2. All locks held by a transaction are released when the transaction is completed.

committed



Exercises

Task 1

Transaction T1 has the following arrangement over database objects A and B: R(A), W(A), R(B), W(B)

a) Give an example of another transaction T2, that, if executed concurrently (without some form of concurrency control), could interfere with T1.

we don't write locks

b) Explain how the use of Strict 2PL would prevent interference between the two transactions.

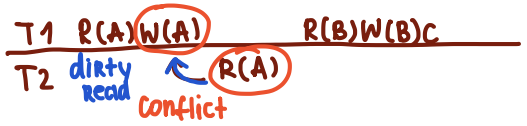
2 phase locking

transaction = schedule

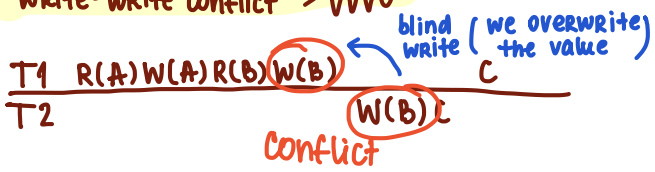
c means commit
s=shared lock
x=exclusive lock

a) • Read-Write conflict → RW

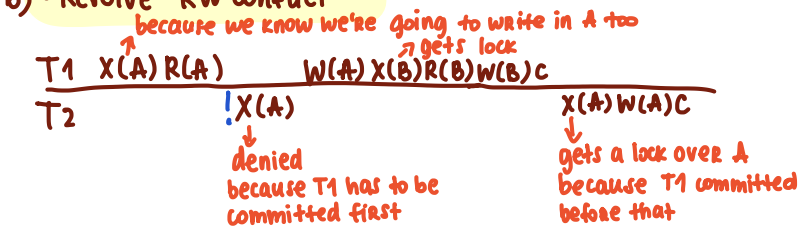
• Write-Read conflict → WR



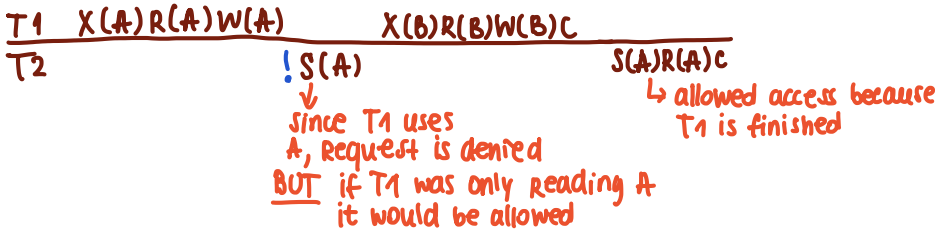
• Write-Write conflict → WW



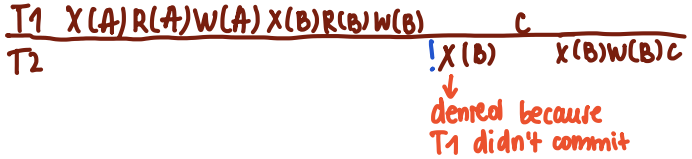
b) • Resolve RW conflict



• Resolve WR conflict



• Resolve WW conflict



Task 2

Consider the following transactions:

T1: R(A) R(B) W(A)
T2: R(A) R(B) W(A) W(B)

follow the same order

a) Give an example of a schedule with transactions T1 and T2 that results in a WR conflict.

b) Give an example of a schedule with transactions T1 and T2 that results in a RW conflict.

c) Give an example of a schedule with transactions T1 and T2 that results in a WW conflict.

d) For each of the three schedules, demonstrate how Strict 2PL resolves these conflicts.

a) T1 R(A) R(B) W(A) C

T2 ! R(A) R(B) W(A) W(B) C

WR conflict

Resolve the conflict:

T1 X(A) R(A) S(B) R(B) W(A) C

T2 ! X(A) X(A) R(A) X(B) R(B) W(B)

b) T1 R(A) R(B) W(A)C

T2 R(A) R(B) W(A) W(B) C

Resolve conflict:

T1 X(A) R(A) S(B) W(A) C

T2 ! X(A) X(A) R(A) X(B) R(B) W(A) W(B) C

there's also a RW conflict

c) T1 R(A) R(B) W(A) C

T2 R(A) R(B) W(A) W(B) C

Resolve conflict:

T1 ! X(A) X(A) S(B) W(A) C

T2 X(A) R(A) X(B) R(B) W(A) W(B) C

Task 3

For a given schedule determine whether there are conflicts between transactions.

```
T1  R(A) W(A)                    C
-------------------------------------------------------------------------
T2                          W(A) C
-------------------------------------------------------------------------
T3                R(A) C
```

a) If yes, explain how would Strict 2PL prevent interference between these transactions.

b) Where do we need to place the T3 „commit“ to cause a RW conflict between T2 and T3?

$\binom{3}{2} = \frac{3 \cdot 2}{2 \cdot 1} = 3$ combinations of conflicts

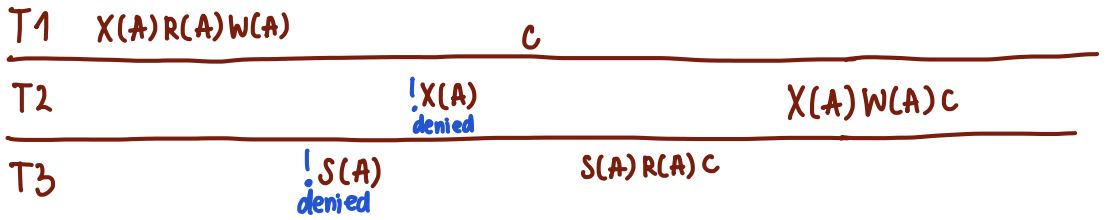
(3: 3 transactions; 2: pairs)

T1 and T2 => WW conflict

T1 and T3 => WR conflict

T2 and T3 => ✓ no conflicts

a)



b) Place the commit(C) after T2 W(A)

Task 4

Given are two transactions:

T1: R(A), R(B), if A = 0 then B:=B+1, W(B)
T2: R(B), R(A), if B = 0 then A:=A+1, W(A)

TRUE

The database consistency is guaranteed by ((A = 0) v (B = 0)) with initial values A = B = 0.

commit entire transaction before beginning with another transaction

a) Show that after both possible serial executions, the database maintains consistency.

A=1 and B=1

b) Provide an example where concurrent execution would result in inconsistency of the database.

c) Is there a case of concurrent implementation that would maintain consistency?

a)

		A	B
	START	0	0
1.	T1	0	1
2.	T2	0	1

it results in A=0 and B=1

condition: A=0 ∨ B=1 ⇒ it's satisfied

		A	B
	START	0	0
2.	T1	1	0
1.	T2	1	0

it results in A=1 and B=0

condition is satisfied

} database is consistent

b)

T1: R(A)=0 … R(B)=0 "if condition" B=1 … W(B)=1 C

T2: R(B)=0 R(A)=0 … "if condition" A=1 W(A)=1

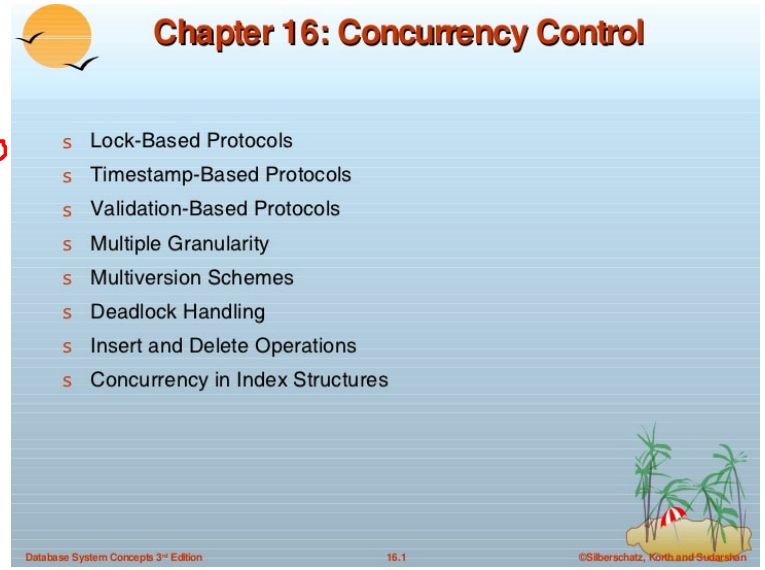
⇒ A=1 ∧ B=1 → inconsistent

c) No, but if we apply concurrency control it's possible

T1: S(A) R(A) X(B) R(B) … "if" W(B) C

T2: !S(B) … S(B) R(B) X(A) R(A) "if" W(B) C

unless if 2pl
(2 phase locking)



Chapter 16: Concurrency Control

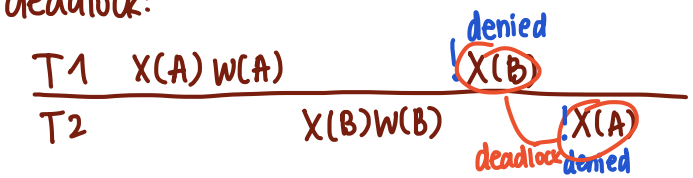
- Lock-Based Protocols
- Timestamp-Based Protocols
- Validation-Based Protocols
- Multiple Granularity
- Multiversion Schemes
- Deadlock Handling
- Insert and Delete Operations
- Concurrency in Index Structures

Task 5

What are the outputs of all SELECT statements?

time step	T1 (insert 4 into Relation square)	T2	
a.	INSERT INTO Squares VALUES (4);		
b.		SELECT * FROM Squares;	← Returns empty set {} conflict, Rerolve with 2PL
c.		INSERT INTO Squares VALUES (9);	
d.		INSERT INTO Squares VALUES (16);	
e.		SELECT * FROM Squares;	← returns {9,16} T1 can't see it until we commit but T2 sees it
f.	COMMIT;		
g.		SELECT * FROM Squares;	← returns {4,9,16}
h.	SELECT * FROM Squares;		← Returns {4} T2 isn't committed yet
i.		COMMIT;	
j.	SELECT * FROM Squares;		← returns {4,9,16}

deadlock:



Introduction to Database Systems

Exercise: Transactions (Part 2)

Exercises

Exercise 1

Transactions T1 and T2 have the following set of actions over objects A and B:

T1: R(A), W(A), R(B), W(B)

T2: W(B), R(A), R(B)

a) Show an example of concurrent implementation that will lead to a deadlock.

b) How would you handle a deadlock?

SUPB sets priorities **based on timestamps** (the smaller the timestamp, the higher the priority).

Ti wants a lock, which is held by Tj.

Two possible policies:

smaller timestamp

Wait-Die: If Ti has higher priority, Ti waits for Tj; otherwise Ti aborts.

Wound-Wait: If Ti has higher priority, Tj aborts; otherwise Ti waits.

2 phase locking

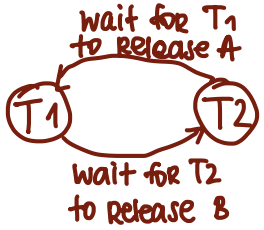
Conservative 2FL: Ti gets all (necessary) locks at the beginning.

c) How would you resolve a deadlock with Wound-Wait policy?

d) How would you resolve a deadlock with Wait-Die policy?

a) T1 X(A) R(A) W(A) !X(B)

T2 X(B) W(B) !S(A)



wait for graph

cycle ⇒ deadlock

ts = timestamp
! => is denied anlamına geliyor

timestamp ↑

c) T1 TS(T1) X(A) R(A) W(A) !X(B) X(B) R(B) W(B) C

T2 TS(T2) X(B) W(B) X(B) W(B) S(A) R(A) R(B) C

apply wound-wait policy
TS(T1) < TS(T2) → T1 has priority

ABORT (TS(T1) < TS(T2))
↳ entire T2 is aborted when T1 finishes, T2 might be restarted

d) T1 TS(T1) X(A) R(A) W(A) !X(B) X(B) R(B) W(B) C

T2 TS(T2) X(B) W(B) !S(A) ABORT (TS(T2) > TS(T1)) X(B) W(B) S(A) R(A) R(B) C

Wait-die policy
T1 has priority → wait for T2 to complete
WAIT (TS(1) < TS(T2)

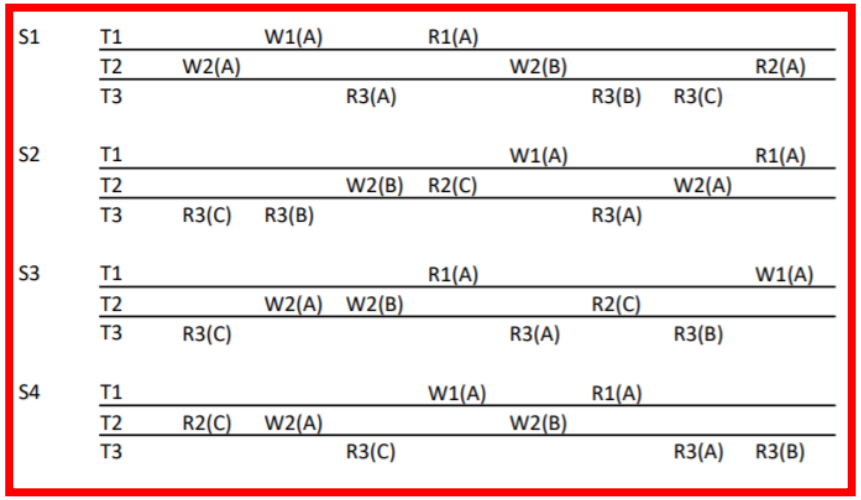
!S(A) ↓
T2 has to abort now locks held by T2 are available

↑
now T2 can restart

Exercise 2

Given are the following schedules:

S1 = W2(A), W1(A), R3(A), R1(A), W2(B), R3(B), R3(C), R2(A)
S2 = R3(C), R3(B), W2(B), R2(C), W1(A), R3(A), W2(A), R1(A)
S3 = R3(C), W2(A), W2(B), R1(A), R3(A), R2(C), R3(B), W1(A)
S4 = R2(C), W2(A), R3(C), W1(A), W2(B), R1(A), R3(A), R3(B)



a) Which of the above schedules is conflict equivalent?

Two schedules are conflict equivalent if:

- Involve the same actions of the same transactions
- Every pair of conflicting actions is ordered the same way

RW, WR, WW

b) Which of the above schedules is conflict serializable? (**include Dependency graph**)

Schedule S is conflict serializable if S is conflict equivalent to some serial schedule

a) S1-S2: S1: W2(A)→W1(A) } they're not conflict equivalent (not ordered in the same way)
S2: W1(A)→W2(A)

S1-S3: S1: W1(A)→R3(A) } they're not conflict equivalent
S3: R3(A)→W1(A)

S1-S4: S1: W1(A)→R2(A) } not conflict equivalent
S4: W1(A)→R2 is missing

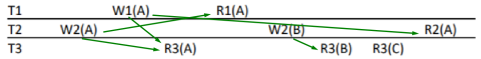
S2-S3: S2: W1(A)→R3(A) } not conflict equivalent
S3: R3(A)→W1(A)

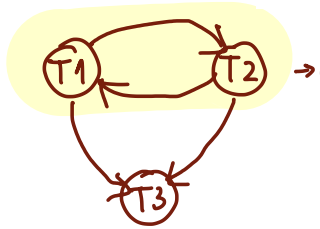
teacher:
s3: w2(a) -> w1(a)

S2-S4: S2: R3(B)→W2(B) } not conflict equivalent
S4: W2(B)→R3(B)

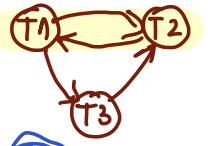
s2: w1(A) -> w2(A)
S4:w2(A) -> w1(A)

S3-S4: S3: R3(A)→W1(A) } not conflict equivalent
S4: W1(A)→R3(A)

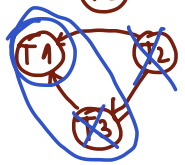


b) S1: 

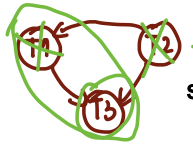
→cycle→so it's not conflict serializable

S2: 

→ cycle ; not serializable

S3: 

→ T2→T3→T1 serial schedule
it's conflict serializable

S4: 

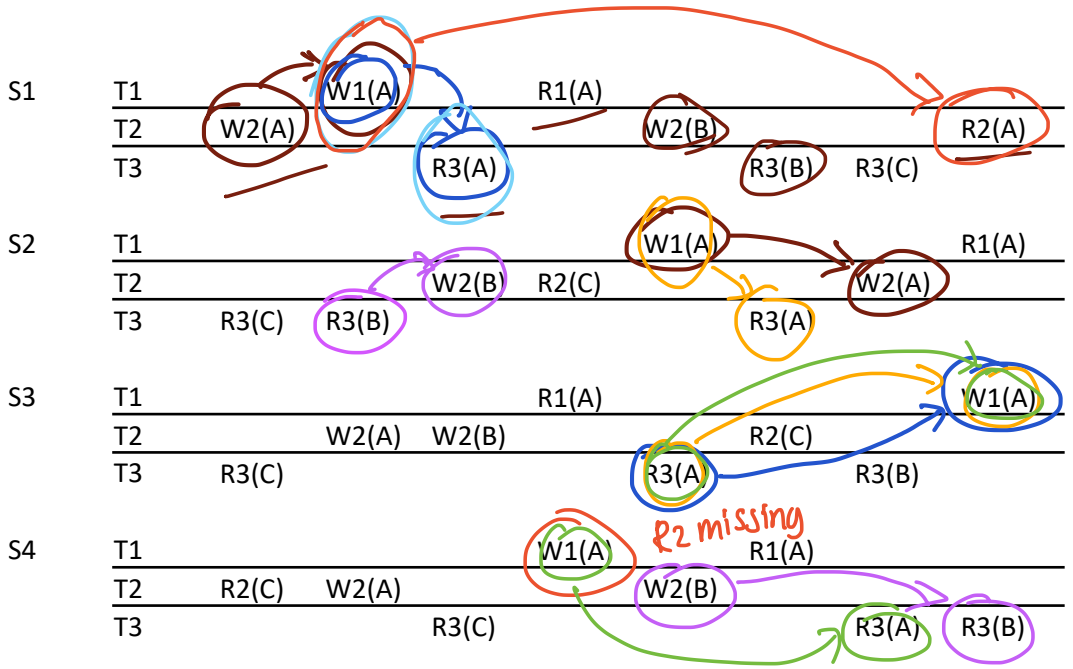
→T2→T1→T3 it's conflict serializable

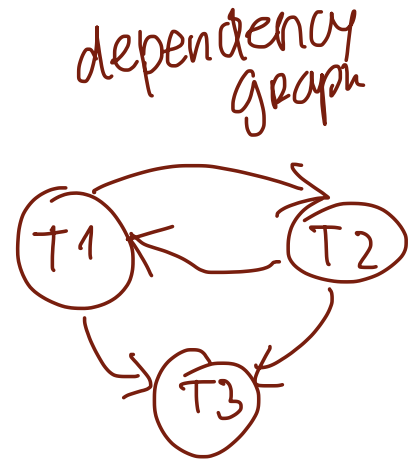
serial Schedule: t2->t1->t3

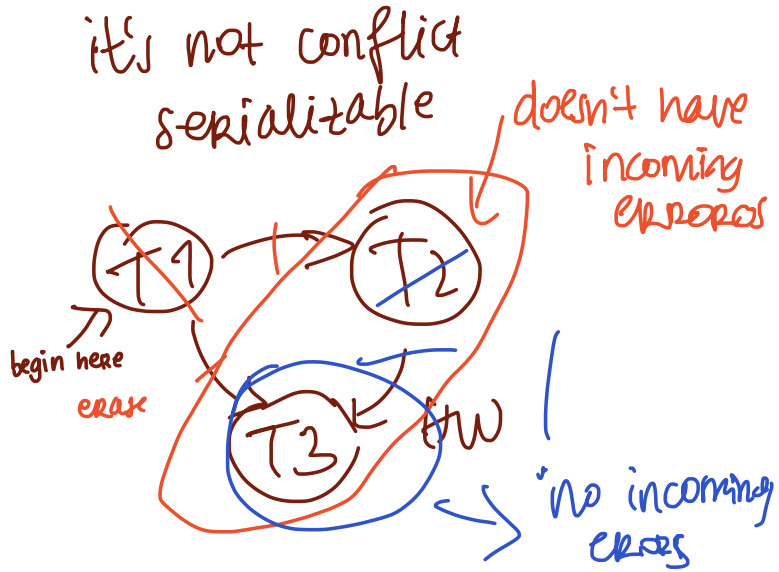
rule:
if no cycle it means
conflict serializable

cycle varsa
deadlock detected demektir.

Exercise 2







a) Which of the above schedules is conflict equivalent?

Two schedules are conflict equivalent if:

- Involve the same actions of the same transactions
- Every pair of conflicting actions is ordered the same way

b) Which of the above schedules is conflict serializable? (**include Dependency graph**)

Schedule S is conflict serializable if S is conflict equivalent to some serial schedule

Exercise 3

When to draw dependency graph arrows:

example:
T1 T2
R1(A)=read
W2(A)=write
+ commit

1. R1(A) → W2(A)
2. W1(A) → R2(A)
3. W1(A) → W2(A)

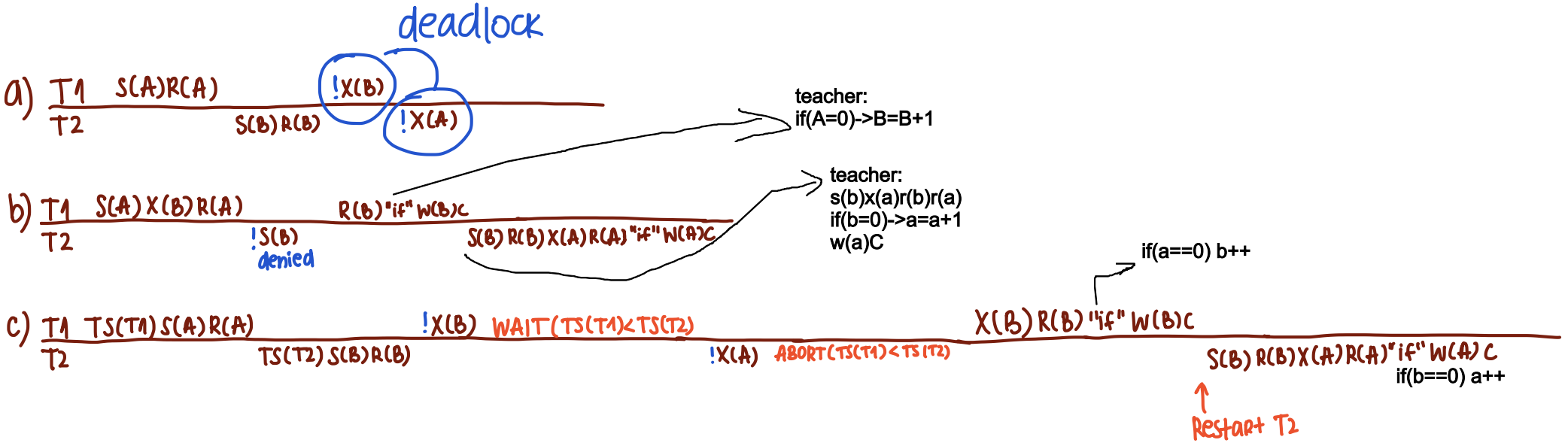
ignore commits!

The following transactions are given:

T1: R(A); R(B); if A = 0 then B=B+1; W(B).

T2: R(B); R(A); if B = 0 then A=A+1; W(A).

a) Can these two transactions lead to a deadlock?
b) How would you resolve a deadlock with a conservative 2PL (C2PL)?
c) How would you resolve a deadlock with of Wait-Die policy?


deadlock
a)
T1 S(A)R(A) !X(B)
T2 S(B)R(B) !X(A)
b)
T1 S(A)X(B)R(A) R(B)"if"W(B)C
T2 !S(B) denied S(B)R(B)X(A)R(A)"if"W(A)C
teacher:
if(A=0)->B=B+1
teacher:
s(b)x(a)r(b)r(a)
if(b=0)->a=a+1
w(a)C
c)
T1 TS(T1)S(A)R(A) !X(B) WAIT(TS(T1)<TS(T2)
T2 TS(T2)S(B)R(B) !X(A) ABORT(TS(T1)<TS(T2)
X(B)R(B)"if"W(B)C
if(a==0) b++
S(B)R(B)X(A)R(A)"if"W(A)C
if(b==0) a++
Restart T2

Exercise 4

S1	T1	R(A)						W(B)	C			
	T2			W(A)		W(B)					C	
	T3						W(B)					C

S2	T1	R(A)						W(B)	C			
	T2			W(B)		W(A)					C	
	T3						W(B)					C

The following sequences of actions are given:

S1: T1:R(A), T2:W(A), T2:W(B), T3:W(B), T1:W(B), T1:Commit, T2:Commit, T3:Commit

S2: T1:R(A), T2:W(B), T2:W(A), T3:W(B), T1:W(B), T1:Commit, T2:Commit, T3:Commit

For each schedule describe how the concurrency control mechanisms prevent a deadlock.

a. Strict 2PL with timestamps (Wait-Die) used for deadlock prevention.

b. Strict 2PL with deadlock detection (in case of deadlock display Wait-for graph).

c. Conservative 2PL.

S1

a) T1 TS(T1) S(A) R(A) ! X(B) WAIT (TS(T1)<TS(T3)) X(B) W(B) C

T2 ! TS(T2) X(A) ABORT (TS(T1)<TS(T2)) X(A) W(A) X(B) W(B) C

T3 TS(T3) X(B) W(B) C

S2

T1 TS(T1) S(A) R(A) ! X(B) WAIT (TS(T1)<TS(T3)) X(B) W(B) C

T2 ! TS(T2) X(B) W(B) X(A) ABORT (TS(T1)<TS(T2)) X(B) W(B) X(A) W(A) C

T3 TS(T3) X(B) W(B) C

S1

c) T1 S(A) X(B) R(A) because a is already locked by t1 W(B) C

T2 denied ! X(A) X(A) X(B) W(A) W(B) C

T3 ! X(B) denied X(B) W(B) C

S2

T1 S(A) X(B) R(A) W(B) C

T2 denied ! X(B) X(B) X(A) W(B) W(A) C

T3 ! X(B) denied X(A) W(A) C

bence x(b) w(b) c

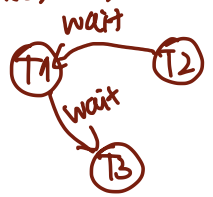
S1

b) T1 S(A) R(A) denied ! X(B) X(B) W(B) C

T2 denied ! X(A) X(A) W(A) X(B) W(B) C

T3 X(B) W(B) C

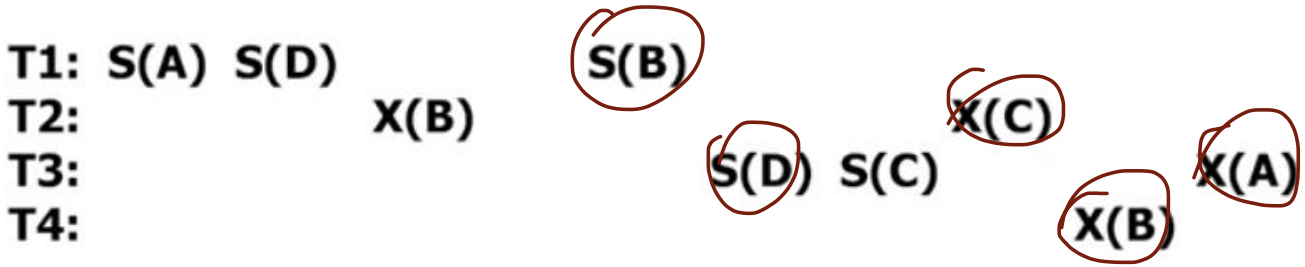
wait-for graph:

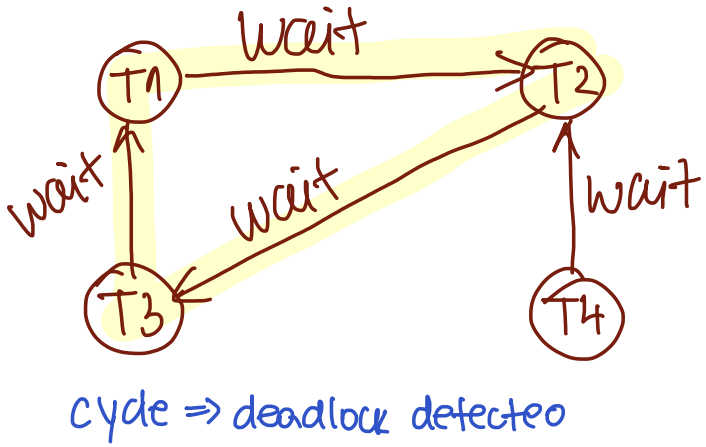


S1	T1	R(A)					W(B)	C		
	T2		W(A)	W(B)					C	
	T3				W(B)					C
S2	T1	R(A)					W(B)	C		
	T2		W(B)	W(A)					C	
	T3				W(B)					C

Exercise 5

Deadlock detection (with Wait-for graph)





Introduction to Database Systems

Functional dependencies and normalization

Functional dependencies

Dependencies between the attributes of the relational scheme define **which values in the relation are possible and which can not exist**.

Functional dependencies represent the relationship between the attributes of the relational schema. They limit the permissible values of the attributes in the relationship tuples.

Let X and Y be nonempty sets of attributes in relation schema R: $X,Y \subseteq R$.

The attributes $\mathbf{X}$ **functionally determine** attributes $\mathbf{Y}$ (denoted by $\mathbf{X \rightarrow Y}$) if there do not exist two tuples in any relation to R schema that would match the values of the X attributes and would not match the values of the Y attributes.

Identification of functional dependencies

Functional dependencies can be identified in two ways:

- based on the **understanding of the relational schema** (common sense, documentation, description)
- on the basis of a **representative set of data** (dependencies must be valid at all times).

In doing so, we want functional dependencies to be complete, which means that all determinants are minimal. For the sake of simplicity, we usually find a **minimum coverage of a set of functional dependencies**.

X → Y

- X determines Y → if the values of X are the same, then Y values are the same but not vice versa
- Primary keys are a special case of FD
- Superkey → a set of columns that determines all the columns in its table

example: relation Hourly_Emps

Hourly_Emps(ssn, name, lot, rating, wage_per_hr, hrs_per_week)

denote the schema by listing its attributes:

SNLRWH = {S, N, L, R, W, H}

FDs for Hourly_Emps:

- ssn is the primary key: S → SNLRWH (S determines the whole relation)
- Rating determines wage_per_hr: R → W
- lot determines lot: L → L (every FD determines itself, it's always true, we don't have to state it)

Exercise 1

Let *EMPLOYEE* be a relational schema of a relation *employed*. Identify functional dependencies based on understanding of the relational schema.

ID	NameSurname	Position	Salary	BranchID	BranchAddress
21	Janez Novak	CEO	4800	P002	Kranjčeva ulica 25, 1000 Ljubljana
37	Zvonko Nered	Programmer	1500	P007	Koprska 10, 1000 Ljubljana

Each employee has a unique employee number. Each branch is located in precisely one location and has at most one branch in each location. The salary of the employee depends on the position of the employee and the branch in which he is employed.

$F_{employed} = \{ ID \rightarrow \{NameSurname, Position, BranchID\},$
$BranchID \rightarrow BranchAddress,$
$BranchAddress \rightarrow BranchID,$
$\{Position, BranchID\} \rightarrow Salary\}$

Exercise 2

Let $R=ABCDE$ be a relational schema of relation r.

Identify functional dependencies based on representative set of data.

$F_r = \{E \rightarrow ABCD,$
$A \rightarrow C,$
$C \rightarrow A,$
$B \rightarrow D,$
$AB \rightarrow E,$
$BC \rightarrow E\}$

A	B	C	D	E
a	b	z	w	a
e	b	r	w	b
a	d	z	w	c
e	d	r	w	d
a	f	z	s	e
e	f	r	s	f

Reasoning about functional dependencies

Let X and Y be subsets of attributes of relational schema R: $X,Y \subseteq R$. A set of functional dependencies F_R **logically implies** dependency $X \rightarrow Y$, if every relation with schema R, that satisfies all dependencies in F_R, also meets the dependence $X \rightarrow Y$.

Denoted,

$$F_R \vDash X \rightarrow Y.$$

We use **Armstrong's Axioms** and **inferential rules** to derive functional dependencies.

Armstrong's Axioms

- **A1**: reflexivity $Y \subseteq X \Rightarrow X \rightarrow Y$ if x is a superset of y then x determines y
- **A2**: augmentation $\{X \rightarrow Y\} \vDash XZ \rightarrow YZ$
- **A3**: transitivity $\{X \rightarrow Y, Y \rightarrow Z\} \vDash X \rightarrow Z$

Armstrong's axioms or theorems are **sound** because they generate only valid dependencies and are **complete** because they generate all valid dependencies.

Union: if $x \rightarrow y$ and $y \rightarrow z$, then $x \rightarrow yz$

Decomposition: if $x \rightarrow yz$, then $x \rightarrow y$ and $x \rightarrow z$

Inferential rules

- **I1**: decomposition: $\mathbf{\{X \to YZ\} \vDash X \to Y}$

  $A1 : YZ \to Y$

  $A3 : \{X \to YZ, YZ \to Y\} \vDash X \to Y$

- **I2**: union: $\mathbf{\{X \to Y, X \to Z\} \vDash X \to YZ}$

  $A2 : \{X \to Y\} \vDash X \to YX$

  $A2 : \{X \to Z\} \vDash YX \to YZ$

  $A3 : \{X \to YX, YX \to YZ\} \vDash X \to YZ$

- **I3**: pseudotransitivity: $\mathbf{\{X \to Y, WY \to Z\} \vDash WX \to Z}$

  $A2 : \{X \to Y\} \vDash WX \to WY$

  $A3 : \{WX \to WY, WY \to Z\} \vDash WX \to Z$

Exercise 3

Let $R = ABCDXYZW$ be a relational schema and F_R and E_R corresponding sets of functional dependencies. Does the dependence set F_R logically imply all dependencies in the E_R set?

Justify the answer using Armstrong's axioms.

$$F_R = \{A \to BC, B \to D, C \to D, X \to ZW, X \to Y, YW \to Z\}$$

$$E_R = \{A \to BCD, B \to D, AC \to BD, X \to W\}$$

$W \subseteq ZW$ $ZW \to W$

$A \to BCD$:
A2: $\{X \to Y\} \models XZ \to YZ$
$\{B \to D\} \models BC \to BCD$
A3: $\{X \to Y, Y \to Z\} \models X \to Z$
$\{A \to BC, BC \to BCD\} \models A \to BCD$

$B \to D$: ✓

$AC \to BD$:
A2: $\{X \to Y\} \models XZ \to YZ$
$\{A \to BC\} \models AC \to BC$
A2: $\{C \to D\} \models BC \to BD$
A3: $\{AC \to BC, BC \to BD\} \models AC \to BD$

$X \to W$:
A1: $W \subseteq ZW \to ZW \to W$
A3: $\{X \to ZW, ZW \to W\} \models X \to W$

Exercise 4

Let $R = ABCDEFGH$ be a relational schema and F_R corresponding set of functional dependencies. Does the dependency set F_R logically imply dependency f ?

Justify the answer using Armstrong's axioms or inferential rules.

$$\mathbf{F_R = \{AB \to D, AC \to FG, B \to G, D \to B\}}$$

1. $f = AC \to C$
2. $f = AC \to G$
3. $f = AB \to DG$
4. $f = AD \to BG$

Exercise 4:

1. $f = AC \to C$
   A1: $C \subseteq AC \vDash AC \to C$
2. $f = AC \to G$
   A1: $G \subseteq FG \vDash FG \to G$
   A3: $\{AC \to FG, FG \to G\} \vDash AC \to G$
3. $f = AB \to DG$
   A2: $\{AB \to D\} \vDash AB \to BD$
   A2: $\{B \to G\} \vDash BD \to DG$
   A3: $\{AB \to BD, BD \to DG\} \vDash AB \to DG$
4. $f = AD \to BG$
   A3: $\{D \to B, B \to G\} \vDash D \to G$
   I2: $\{D \to B, D \to G\} \vDash D \to BG$
   A2: $\{D \to BG\} \vDash AD \to ABG$
   A1: $BG \subseteq ABG \vDash ABG \to BG$
   A3: $\{AD \to ABG, ABG \to BG\} \vDash AD \to BG$

A1: reflexivity $\mathbf{Y \subseteq X \Rightarrow X \to Y}$
A2: augmentation $\mathbf{\{X \to Y\} \vDash XZ \to YZ}$
A3: transitivity $\mathbf{\{X \to Y, Y \to Z\} \vDash X \to Z}$

I1: decomposition: $\mathbf{\{X \to YZ\} \vDash X \to Y}$
$A1 : YZ \to Y$
$A3 : \{X \to YZ, YZ \to Y\} \vDash X \to Y$

I2: union: $\mathbf{\{X \to Y, X \to Z\} \vDash X \to YZ}$
$A2 : \{X \to Y\} \vDash X \to YX$
$A2 : \{X \to Z\} \vDash YX \to YZ$
$A3 : \{X \to YX, YX \to YZ\} \vDash X \to YZ$

I3: pseudotransitivity: $\mathbf{\{X \to Y, WY \to Z\} \vDash WX \to Z}$
$A2 : \{X \to Y\} \vDash WX \to WY$
$A3 : \{WX \to WY, WY \to Z\} \vDash WX \to Z$

Key identification

Let F_R be a set of functional dependencies of relational schema R and let X be a subset of attributes of schema R: $X \subseteq R$. Attributes X are **key candidates** of schema R or corresponding relations if:

- attributes X functionally determine all the attributes of the R schema,
- There is no subset $X' \subset X$, which would functionally determine all attributes of schema R.

The relational schema may have several key candidates. We select one of them, which we call the **primary key**. A subset of the relational schema attributes, which is the primary key in another schema, is called a **foreign key**.

When identifying key candidates, we must check all possible subsets of the relational schema attributes. The first condition in the definition is checked by closing set of attributes, while the second condition gradually limits the space of possible key candidates.

Normalization

Normalization is the process of transforming relational schemes or corresponding relations into a form in which we cannot get anomalies.

Normalization uses a bottom-up approach. Planning a database thus starts with a single relationship that contains all the important attributes. The relationship is then gradually transformed into the desired normal form.

- **1NF**: first normal form
- **2NF**: second normal form
- **3NF**: third normal form
- **BCNF**: Boyce-Codd normal form
- **4(B)NF**: fourth (business) normal form
- **5(B)NF**: fifth (business) normal form

1NF, 2NF, 3NF and BCNF are based on functional dependencies, 4BNF is based on several value dependencies, 5BNF is based on contact dependencies.

First normal form (1NF)

A relation is in 1NF if it:

- has functional dependencies,
- has a primary key,
- contains only atomic values (no lists or sets).

We eliminate the superseded attributes (non-atomic value attributes) by entering the missing values or transferring the multivalued attributes together with the key to a new relation.

Second normal form (2NF)

The relation is in the second normal form 2NF, if:

- it is in first normal form 1NF and
- it doesn’t have partial dependencies.

Partial dependency means that the attributes, which are not part of the key, are functionally depend only on the part of the key.

We resolve the partial dependencies by breaking the relation to several new relations. The relation is automatically in 2NF if it is in 1NF and its key consists of only one or all of the schema attributes.

Third normal form (3NF)

Relation is in third normal form 3NF, if:

- it is in second normal form 2NF and
- it doesn‘t have transitive dependencies.

Transitive dependence is a functional dependency between attributes that are not part of the key.

We eliminate transitive dependencies by breaking the relationship into several new relations. The relation is automatically in 3NO if it is in 2NO and its key consists of all or all but one schema attribute.

Exercise 5

Let ENROLMENT be a relational schema, which represents enrollment of the students in each year. Gradually normalize the scheme to 3NF.

ENROLMENT(EnrolmentID, Name, Surname, (RegID, ProgrammeID, ProgrammeName, Year, RegistrationFee))

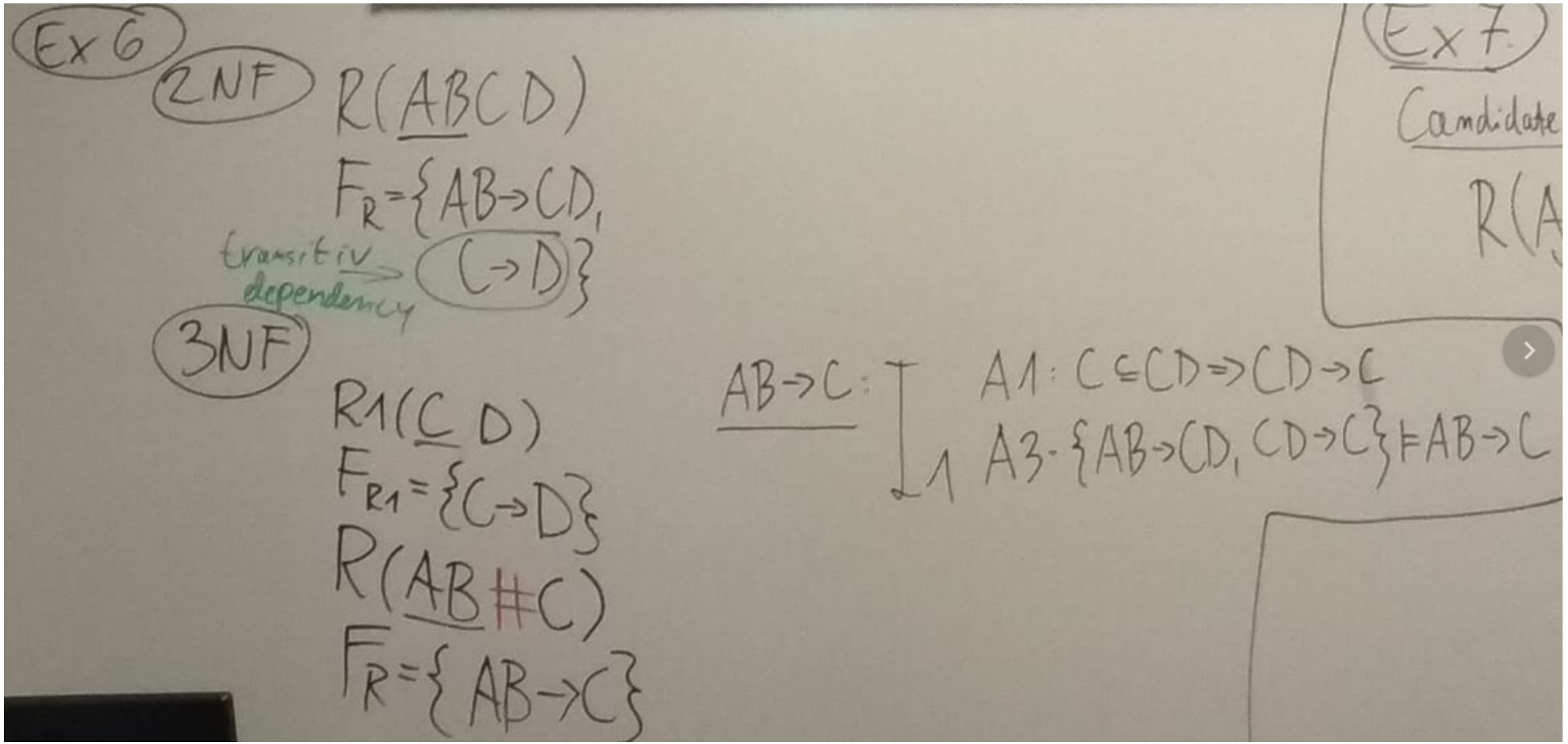
Exercise 6

Let $\mathrm{R}=\mathrm{ABCD}$ be a relational schema, $\mathrm{F_R}$ corresponding set of functional dependencies and $\mathrm{X}=\mathrm{AB}$ primary key for R.

In what normal form is the schema R?

Gradually normalize the scheme to 3NF, and for each new scheme, specify the keys and associated functional dependencies.

$$\mathrm{FR} = \{\mathrm{AB} \rightarrow \mathrm{CD}, \mathrm{C} \rightarrow \mathrm{D}\}$$


Ex 6
2NF
R(ABCD)
F_R={AB→CD,
transitiv dependency
C→D}
3NF
R1(C D)
F_{R1}={C→D}
R(AB#C)
F_R={AB→C}
AB→C:
A1: C⊆CD⇒CD→C
A3: {AB→CD, CD→C}⊨AB→C
Ex 7.
Candidate
R(A

none of normal forms because no primary key

Exercise 7

Let $\mathrm{R}=\mathrm{ABCDE}$ be a relational schema, $\mathrm{F_R}$ corresponding set of functional dependencies.

What normal form is the R scheme in?

Gradually normalize the scheme to 3NF, and for each new scheme, specify the keys and associated functional dependencies.

$$\mathrm{F_R} = \{\mathrm{A} \rightarrow \mathrm{D}, \mathrm{B} \rightarrow \mathrm{E}, \mathrm{E} \rightarrow \mathrm{B}\}$$

Ex 7. None of NFs

Candidate Key : ABC, AEC

R(ABCDE)

1NF R(ABCDE)

$F_R = \{A \to D, B \to E, E \to B\}$

$C\}$ ⊨ $AB \to C$

It is already in the 3NF

2NF

R1(A D)

$F_{R1} = \{A \to D\}$

R2(B E)

$F_{R2} = \{B \to E, E \to B\}$

R(#A#BC)

$F_R = \{\}$

Introduction to database systems

Exercises: BCNF + Physical design

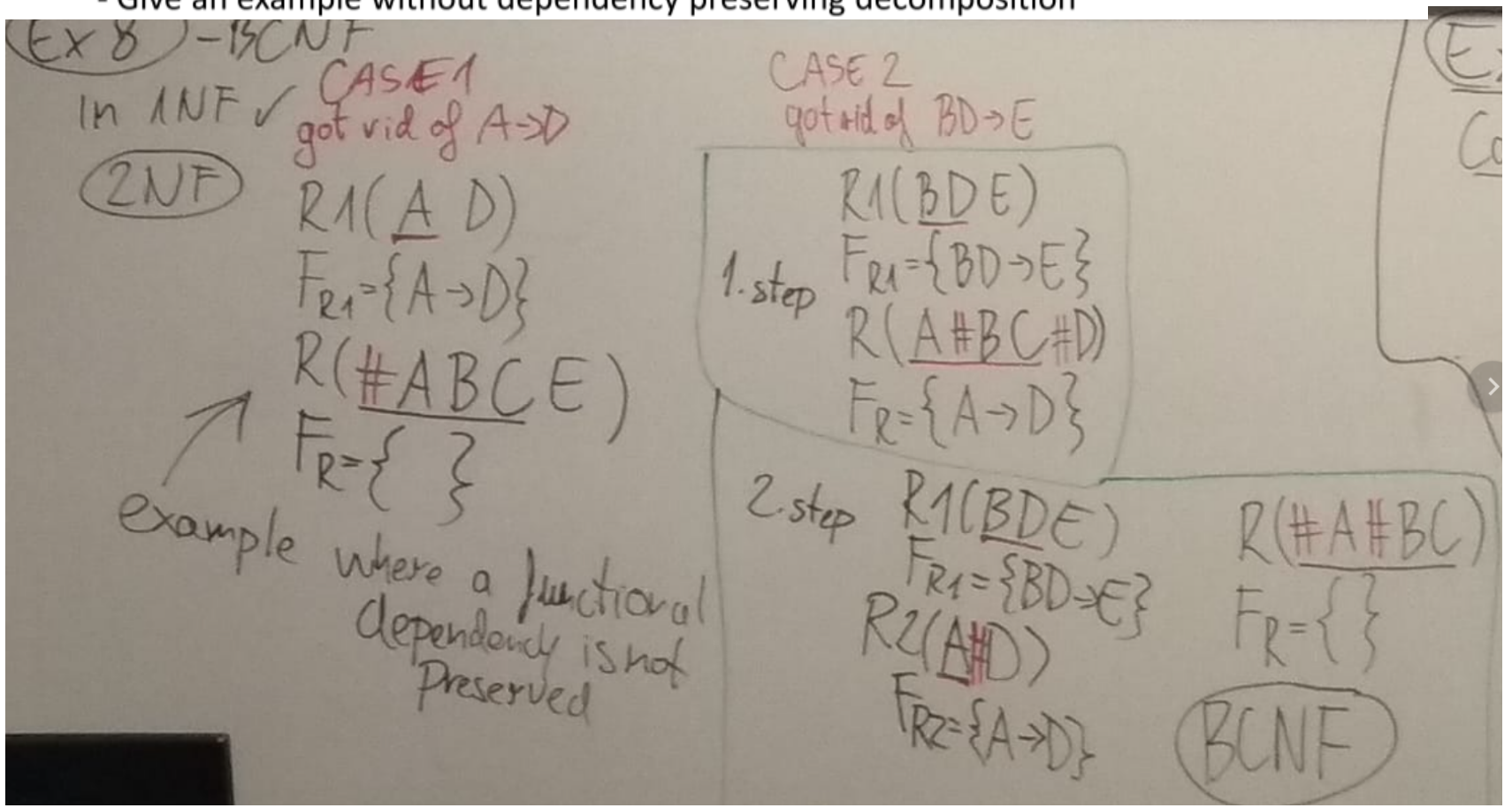
Functional dependencies – BCNF (3.5NF)

R (A, B, C, D, E)

FR = {A->D, BD->E}

Can we tranform to BCNF?

- Give an example with dependency preserving decomposition
- Give an example without dependency preserving decomposition



Optimizer

For each of the following queries, identify one possible reason why an optimizer might not find a good plan. Rewrite the query so that a good plan is likely to be found.

1. An index is available on the *age* attribute:

```
SELECT E.dno
FROM Employee E
WHERE E.age=20 OR E.age=10
```

```
SELECT E.dno
FROM Employee E
WHERE E.age = 20
UNION
SELECT E.dno
FROM Employee E
WHERE E.age = 10;
```

2. A B+ tree index is available on the *age* attribute:

```
SELECT E.dno
FROM Employee E
WHERE E.age<20 AND E.age>10
```

```
SELECT E.dno
FROM Employee E
WHERE E.age BETWEEN 10 AND 19;
```

Optimizer

3. An index is available on the age attribute:

```
SELECT E.eIno
FROM Enlployee E
WHERE 2*E.age<20
```

```
SELECT E.dno
FROM Employee E
WHERE E.age < 10;
```

4. No index is available:

```
SELECT DISTINCT *
FROM Enlployee E
```

```
SELECT *
FROM Employee;
```

5. No index is available:

```
SELECT AVG (E.sal)
FROM Employee E
GROUP BY E.dno
HAVING E.dno=22
```

```
SELECT AVG(E.sal)
FROM Employee E
WHERE E.dno = 22;
```

6. The sid in Reserves is a foreign key that refers to Sailors:

```
SELECT S.sid
FROM Sailors S, Reserves R
WHERE S.sid=R.sid
```

```
SELECT R.sid
FROM Reserves R
WHERE R.sid IS NOT NULL;
```

Because what if other deletes Reserve

Postgresql

Query plan (evaluation):

EXPLAIN select * from relation;

Returns only relative cost of a query.

```
explain select * from cd.facilities where facid = 4;
```

Data Output

	QUERY PLAN text
1	Index Scan using facilities_pk on facilities (cost=0.14..8.15 rows=1 width=350)
2	Index Cond: (facid = 4)

Query plan (evaluation) + execution :

EXPLAIN ANALYZE select * from relation;

Returns actual execution time.

Command is executed.

```
explain analyze select * from cd.facilities where facid = 4;
```

Data Output

	QUERY PLAN text
1	Index Scan using facilities_pk on facilities (cost=0.14..8.15 rows=1 width=350) (actual time=0.012..0.013 rows=1 loops=1)
2	Index Cond: (facid = 4)
3	Planning Time: 0.415 ms
4	Execution Time: 0.026 ms

Introduction of database systems

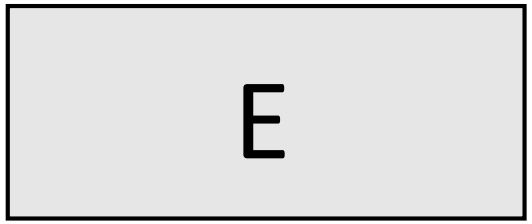
ER Model and

translation of ER to SQL

ER model - building blocks

ENTITY = real world object

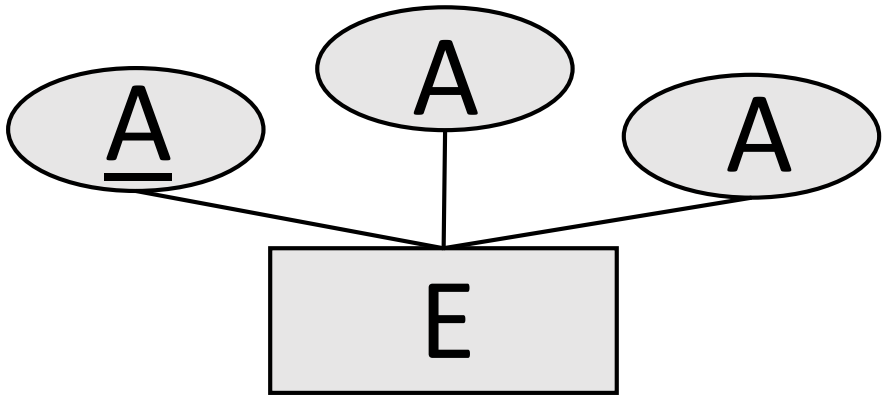
- represented with a rectangle



- examples of entities: house, car, table, person, dog, ...

ATTRIBUTE = entity property

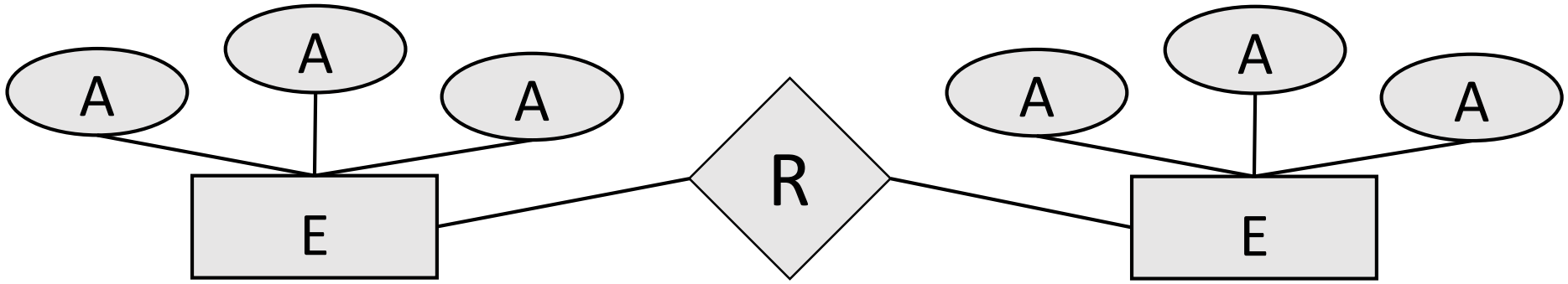
- represented with an ellipse
- example: table material, dog‘s name ...



ER model - building blocks

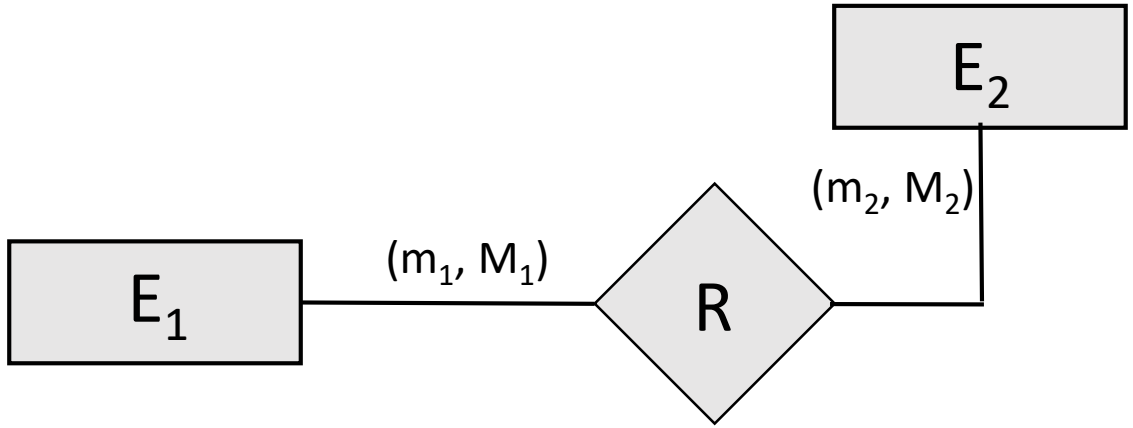
RELATIONSHIP = it defines a connection between two or more entities belonging to different entity sets

- presented with a diamond



- example: Janko owns a wooden desk, Ringo lives in a dog house ...

ER model - building blocks – cardinality of a relationship



Cardinality of the entity set E_i in the relationship R is a function :

card(E_i, R) = (min, max)

min is minimal cardinality of entity set E_i in relationship R,

max is maximal cardinality E_i in R.

Cardinality of an entity set E_i in relationship R describes in how many different relationships an entity from R can participate.

Possible values of minimal and maximal cardinality:
"0" (zero), "1" (one) in "N" (reads as "many"; in general more than one).

ER models building blocks – cardinality of a relationship

The classification of relationships is based on the **maximal cardinality** of entities in the relationship:

- Types of the entity set E roles in the relationship R :
  - **max-card(E,R) = 1**; E has a single-valued role in the relationship R.
  - **max-card(E,R) = N**; E has a multi-valued role in R.

- Binary relationship R between the entity sets E and F is denoted :
  - **1—1**; one-to-one: if E and F have single-valued role in R
  - **1—N**, **N—1**; one-to-many: if one entity set has single-valued and the other has multi-valued role in R
  - **N—N**; many-to-many: if E and F have multi-valued role in R

ER models building blocks – cardinality of a relationship

Minimal cardinality serves as the second type of relationship classification: participation constraint.

min-card(E,R) = 1

Each entity from the set E appears in at least one relationship instance of R.

Entities from E are mandatory in the relationship R.

min-card(E,R) = 0

Some entities from E are not part of any relationship from R.

Entities from E are optional in the relationship R.

ER models building blocks – cardinality of a relationship

An employee doesn‘t need to manage any departments, or can manage several deparments.

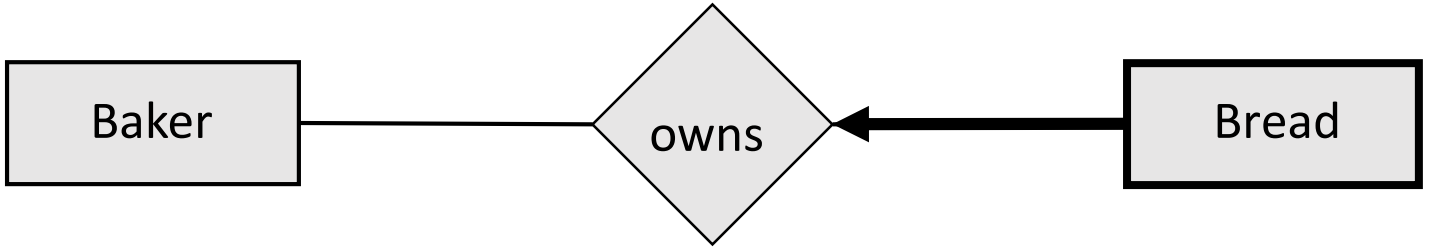
The department must be managed by at least 1 and at most 1 employee.

Alternatively

CHEN notation

ER models building blocks – weak entity

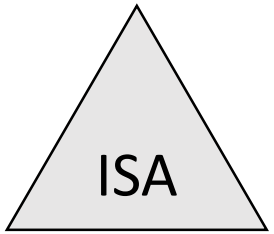
- the existence of weak entity is conditioned with the existence of some other entity.
- presented with an arrow in the direction of the dependence



- it is identified by using a key of another entity
- example: every bread has its own baker (entity bread is dependent on entity baker)

ER models building blocks – ISA hierarchy

- **ISA hierarchy**
  - entity inherits the attributes of the parent entity
  - presented with a triangle

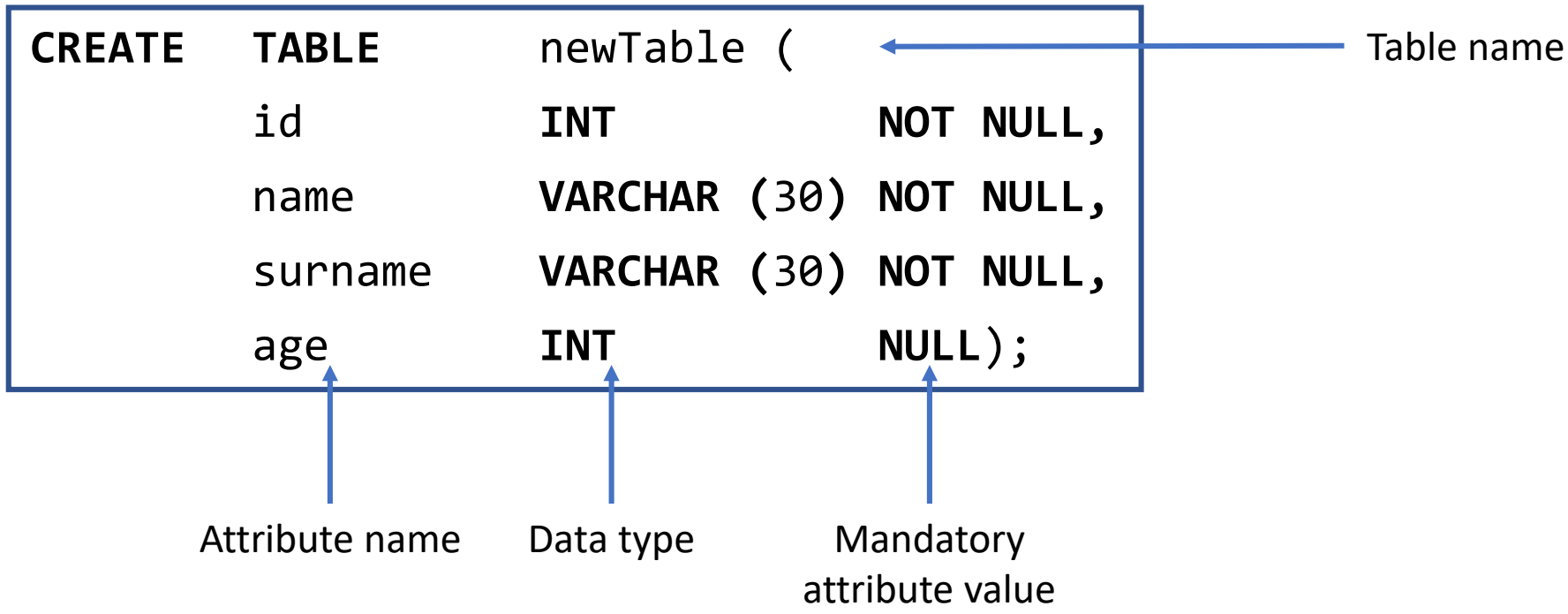


  - example: olive ISA tree

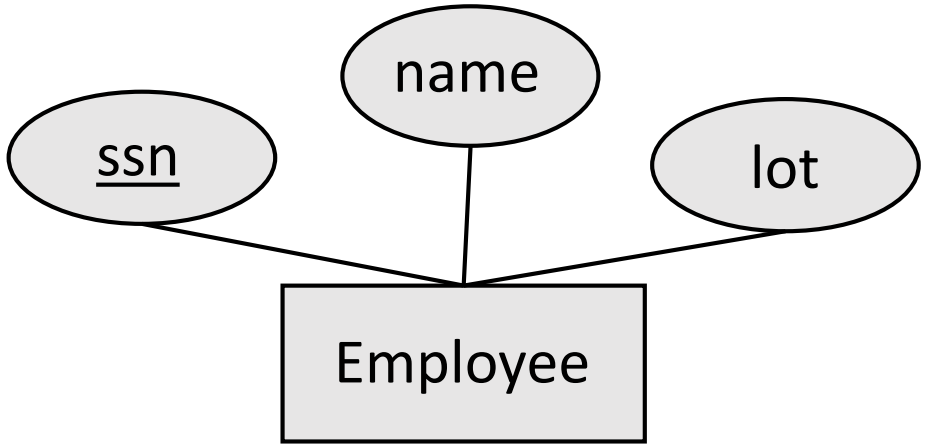
Translation of ER to SQL

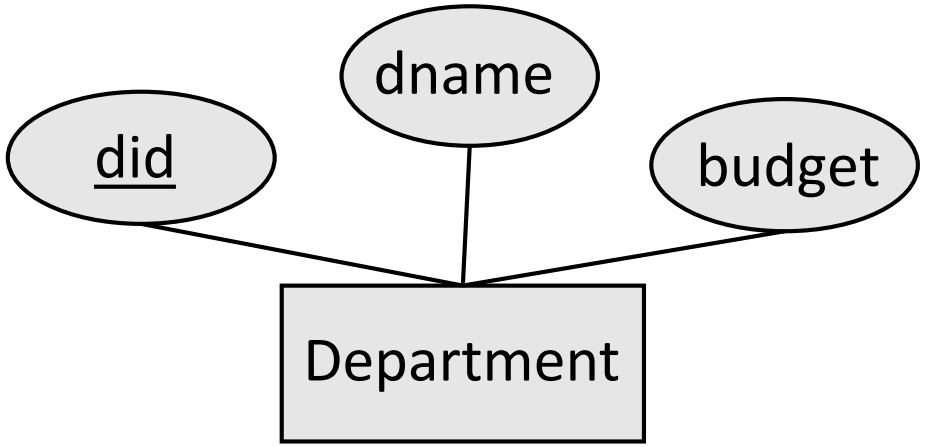
Creation of objects in SQL

CREATE for creation of objects (TABLE, USER, VIEW,...)



Translation of relationships

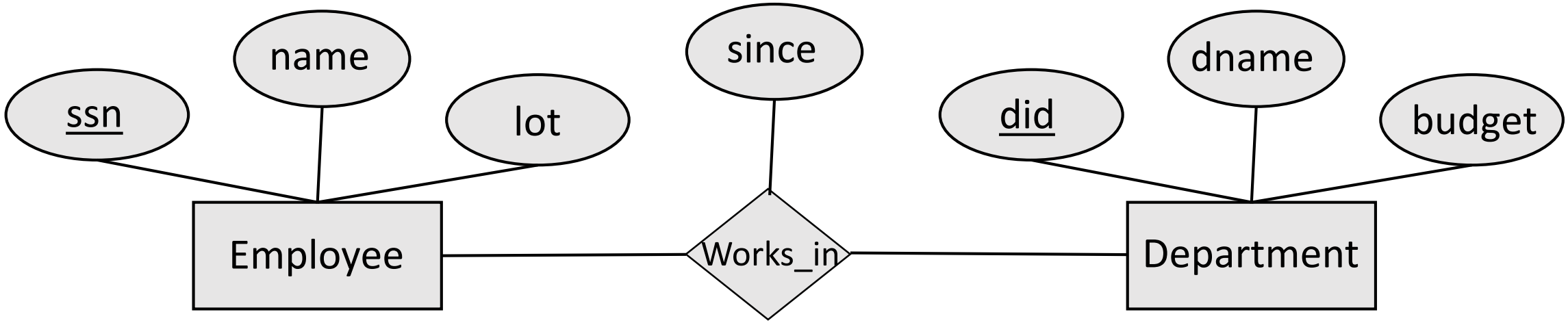





```
CREATE TABLE Employee (ssn    CHAR(11),
                       name   CHAR(30),
                       lot    INTEGER,
                       PRIMARY KEY (ssn))
```

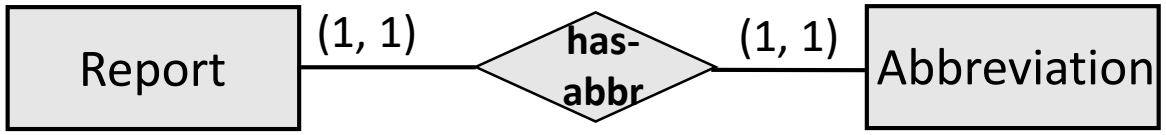
```
CREATE TABLE Department (did CHAR(11),
                         dname  CHAR(30),
                         budget INTEGER,
                         PRIMARY KEY (did))
```

Translation of relationships (N-N)




```
CREATE TABLE Works_in (    ssn    CHAR(11),
                           did    INTEGER,
                           since  DATE,
                           PRIMARY KEY (ssn, did),
                           FOREIGN KEY (ssn) REFERENCES Employee,
                           FOREIGN KEY (did) REFERENCES Department);
```

Binary relationship
1-1 (2 x mandatory)

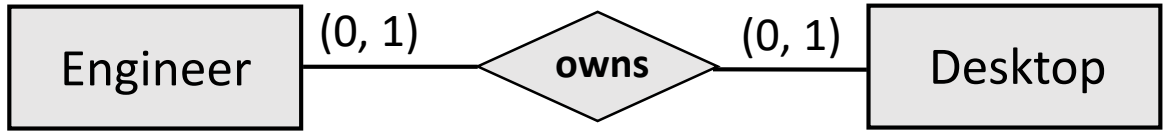



```
CREATE TABLE Report (      report_no    INTEGER,
                           report_name  VARCHAR(256),
                           PRIMARY KEY (report_no)
                     );
CREATE TABLE Abbreviation (abbr_no      CHAR(6),
                           report_no    INTEGER NOT NULL UNIQUE,
                           PRIMARY KEY (abbr_no),
                           FOREIGN KEY (report_no) REFERENCES Report
                                 ON DELETE CASCADE ON UPDATE CASCADE
                           );
```

Binary relationship
1-1 (1 x mandatory)

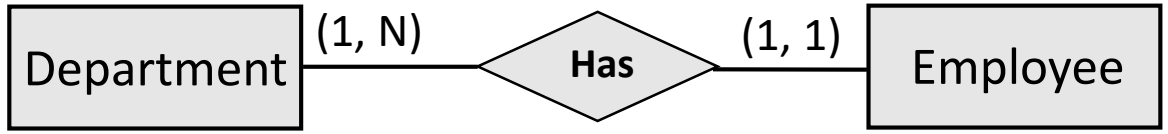

```
CREATE TABLE Department ( dept_no       INTEGER,
                          dept_name     CHAR(20),
                          emp_id        CHAR(10) NOT NULL UNIQUE,
                          PRIMARY KEY (dept_no),
                          FOREIGN KEY (emp_id) REFERENCES Employee
                                ON DELETE SET DEFAULT ON UPDATE CASCADE);

CREATE TABLE Employee (   emp_id        CHAR(10),
                          emp_name      CHAR(20),
                          PRIMARY KEY (emp_id));
```

Binary relationship
1-1 (2 x optional)



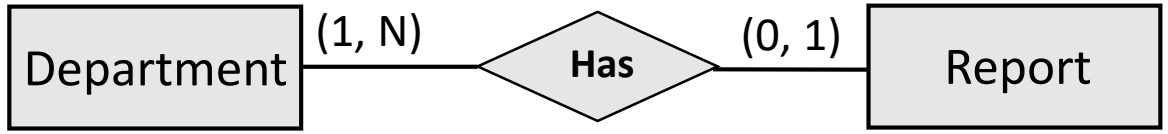

```
CREATE TABLE Engineer (    emp_id        CHAR(10),
                           desktop_no    INTEGER,
                           PRIMARY KEY (emp_id));

CREATE TABLE Desktop (     desktop_no    INTEGER,
                           emp_id        CHAR(10),
                           PRIMARY KEY   (desktop_no),
                           FOREIGN KEY (emp_id) REFERENCES Engineer ON
                           DELETE SET NULL ON UPDATE CASCADE)
```

Binary relationship
1-N (2 x mandatory)




```
CREATE TABLE Department ( dept_no       INTEGER,
                          dept_name     CHAR(20),
                          PRIMARY KEY (dept_no));


CREATE TABLE Employee (   emp_id        CHAR(10),
                          emp_name      CHAR(20),
                          dept_no       INTEGER NOT NULL,
                          PRIMARY KEY (emp_id),
                          FOREIGN KEY (dept_no) REFERENCES Department
                          ON DELETE SET DEFAULT ON UPDATE CASCADE);
```

Binary relationship
1-N (1 x mandatory)

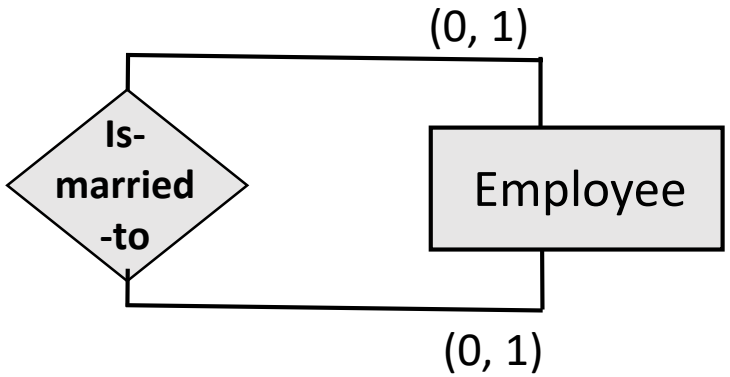



```
CREATE TABLE Department ( dept_no       INTEGER,
                          dept_name     CHAR(20),
                          PRIMARY KEY (dept_no));


CREATE TABLE Report (     report_no     INTEGER,
                          dept_no       INTEGER,
                          PRIMARY KEY (report_no),
                          FOREIGN KEY (dept_no) REFERENCES Department
                          ON DELETE SET NULL ON UPDATE CASCADE);
```

Binary relationship
N-N (2 x optional)

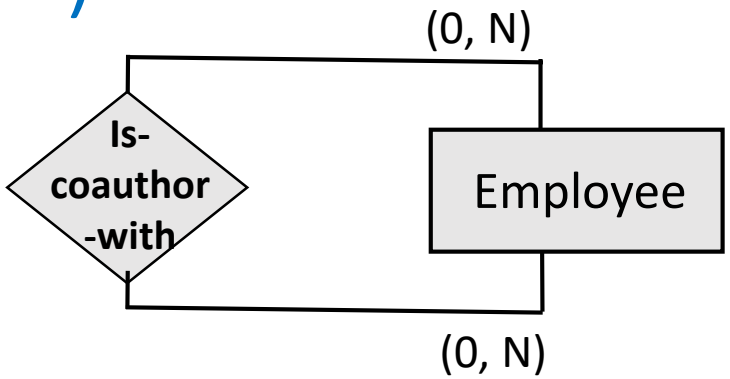

```
CREATE TABLE Engineer (     emp_id          CHAR(10),
                            PRIMARY KEY (emp_id));

CREATE TABLE Prof_assoc (   assoc_name      VARCHAR(256),
                            PRIMARY KEY (assoc_name));

CREATE TABLE Belongs_to (   emp_id          CHAR(10),
                            assoc_name      VARCHAR(256),
                            PRIMARY KEY (emp_id, assoc_name),
                            FOREIGN KEY (emp_id) REFERENCES Engineer ON
                            DELETE CASCADE ON UPDATE CASCADE,
                            FOREIGN KEY (assoc_name) REFERENCES Prof_assoc
                            ON DELETE CASCADE ON UPDATE CASCADE);
```

Recursive relationship: 1-1 (2 x optional)




```
CREATE TABLE Employee (    emp_id        CHAR(10),
                           emp_name      CHAR(20),
                           spouse_id     CHAR(10),
                           PRIMARY KEY (emp_id),
                           FOREIGN KEY (spouse_id) REFERENCES Employee
                           ON DELETE SET NULL ON UPDATE CASCADE);
```

Recursive relationship: N-N (2 x optional)

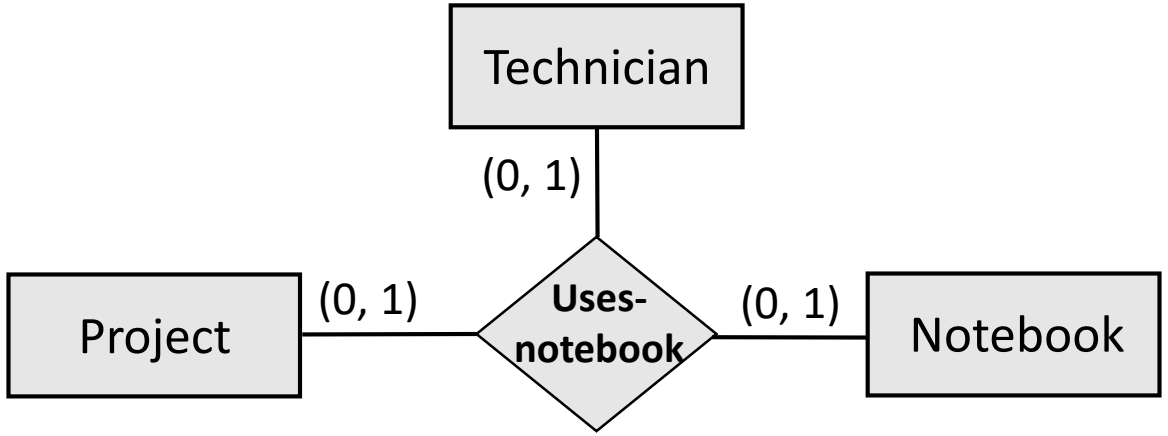



```
CREATE TABLE Employee (    emp_id        CHAR(10),
                           emp_name      CHAR(20),
                           PRIMARY KEY (emp_id));

CREATE TABLE Coauthor (    author_id     CHAR(10),
                           coauthor_id   CHAR(10),
                           PRIMARY KEY (author_id, coauthor_id),
                           FOREIGN KEY (author_id) REFERENCES Employee ON
                           DELETE CASCADE ON UPDATE CASCADE,
                           FOREIGN KEY (coauthor_id) REFERENCE Employee ON
                           DELETE CASCADE ON UPDATE CASCADE);
```

Ternary relationships

- Keys in ternary relationships are always NOT NULL.
- Keys have to be deleted and updated in cascade form.

Ternary relationships: 1-1-1




```
CREATE TABLE Technician (   emp_id         CHAR(10),
                            PRIMARY KEY (emp_id));
CREATE TABLE Project (      project_name   CHAR(20),
                            PRIMARY KEY (project_name));
CREATE TABLE Notebook (     notebook_no    INTEGER,
                            PRIMARY KEY (notebook_no));
CREATE TABLE Uses_notebook (        emp_id         CHAR(10),
                                    project_name   CHAR(20),
                                    notebook_no    INTEGER NOT NULL,
                                    PRIMARY KEY (emp_id, project_name),
                                    FOREIGN KEY (emp_id) REFERENCES Technician
                                    ON DELETE CASCADE ON UPDATE CASCADE,
                                    FOREIGN KEY (project_name) REFERENCES Project
                                    ON DELETE CASCADE ON UPDATE CASCADE,
                                    FOREIGN KEY (notebook_no) REFERENCES Notebook
                                    ON DELETE CASCADE ON UPDATE CASCADE,
                                    UNIQUE (emp_id, notebook_no),
                                    UNIQUE (project_name, notebook_no));
```

Exercises

ER models and translation of ER to SQL

Exercise 1 (CREATE/ALTER/DROP … TABLE)

- Create schema „bakery“
- Create a table „baker“ (id baker, name), and „bread“ (id bread, #id_baker, name)

  CREATE TABLE …

- We decide that the bread‘s attribute „name“ is no longer useful. Drop „name“.
- Instead, we want to add a new attribute „typee“. Add „typee“
- Accidentally, we made a typo … let‘s rename it to „type“

  ALTER TABLE … [DROP|ADD|RENAME] … [AFTER|TO]

- We want to add some instances to the database. **INSERT INTO … VALUES (…)**

baker

Id_baker	name
1	Jan
2	Peter

bread

Id_bread	Id_baker	type
1	1	„white“
2	2	„whole grain“

- Update the name of the first baker (id = 1) to ‚Janez‘ **UPDATE … SET … WHERE …**
- Remove a baker whose id is 2. **DELETE FROM … WHERE …**
- Drop both tables **DROP TABLE …**

```
create table baker (id_baker int,name varchar(15),primary key (id_baker));

create table bread(id_bread serial, id_baker int,name varchar (15),primary key (id_bread),foreign key
(id_baker) references baker   on delete cascade on update cascade);

ALTER TABLE bakery.bread DROP name;

ALTER TABLE bakery.bread ADD typee varchar(15);

ALTER TABLE bakery.bread RENAME typee TO type;

insert into baker(id_baker, name) VALUES (1, 'Jan'), (2, 'Peter');
SELECT * FROM baker;   -- just to see. Not so important to add this.

INSERT INTO bread(id_baker, type) VALUES (1, 'White'), (2, 'Whole grain');
SELECT * FROM bread;  -- this is a comment line.

UPDATE baker SET name = 'Janez' WHERE id_baker = 1;

DELETE FROM baker WHERE id_baker = 2;

DROP SCHEMA bakery cascade;
```

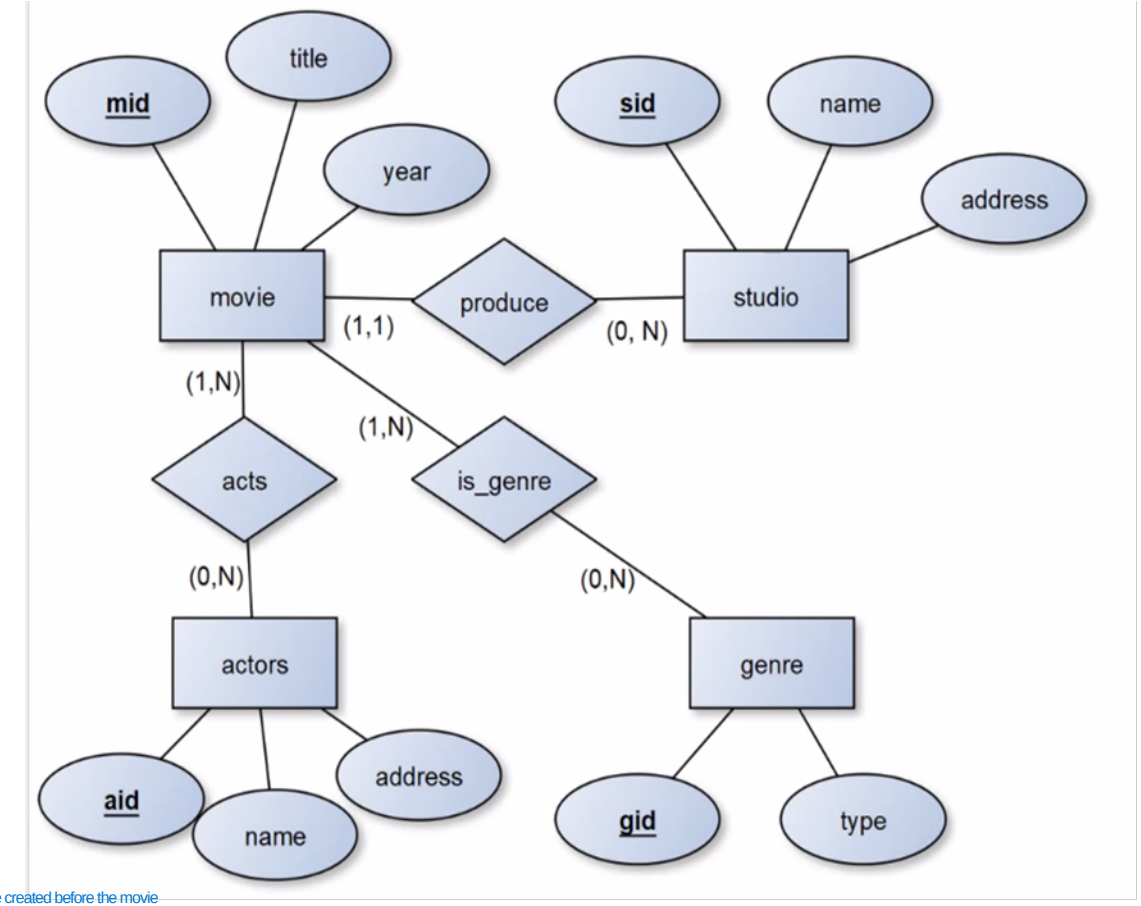
Exercise 2

Create a conceptual model of a simple database of movies you have at home.

For every movie we want to store at least the following:

- Title
- Year of issue
- Production house
- Actors
- Genre

Relations: Movie, Production house (Studio), Actors, Genre




```
/*
Studio should be created before the movie
because movie is (1,1) but studio is different.
*/

CREATE TABLE studio(
 sid int,
 name varchar(15) not null,
 adress varchar(49) not null,
 PRIMARY KEY (sid)
);

CREATE TABLE movie(
 mid int,
 title varchar(25) not null, --we want without any null
values.
 year date not null,
 sid int not null -- because mandatory
 PRIMARY KEY(mid),
 FOREIGN KEY(sid) REFERENCES studio
  ON DELETE SET DEFAULT ON UPDATE CASCADE
);

CREATE TABLE actors (
 aid int,
 name varchar(15) not null,
 adress varchar(25) not null,
 PRIMARY KEY (aid)
);

CREATE TABLE acts(
 mid int not null,
 aid int not null,
 PRIMARY KEY (mid, aid),
 FOREIGN KEY (mid) REFERENCES movie
  ON DELETE CASCADE ON UPDATE CASCADE
 FOREIGN KEY (aid) REFERENCES actors
  ON DELETE CASCADE ON UPDATE CASCADE
);

CREATE TABLE genre(
 gid int,
 type varchar(15) NOT NULL,
 PRIMARY KEY (gid)
);

CREATE TABLE is_genre(
 mid int not null,
 gid int not null,
 PRIMARY KEY (mid,gid),
 FOREIGN KEY (mid) REFERENCES movie
  ON DELETE CASCADE ON UPDATE CASCADE,
 FOREIGN KEY (gid) REFERENCES genre
  ON DELETE CASCADE ON UPDATE CASCADE
);
```

```
CREATE TABLE genre(
    gid int,
    type varchar(15) NOT NULL,
    PRIMARY KEY (gid)
);

CREATE TABLE is_genre(
    mid int not null,
    gid int not null,
    PRIMARY KEY (mid,gid),
    FOREIGN KEY (mid) REFERENCES movie
        ON DELETE CASCADE ON UPDATE CASCADE,
    FOREIGN KEY (gid) REFERENCES genre
        ON DELETE CASCADE ON UPDATE CASCADE
);
```

```
/*
Studio should be created before the movie
because movie is (1,1) but studio is different.
*/

CREATE TABLE studio(
    sid int,
    name varchar(15) not null,
    adress varchar(49) not null,
    PRIMARY KEY (sid)
);

CREATE TABLE movie(
    mid int,
    title varchar(25) not null, --we want without any null values.
    year date not null,
    sid int not null -- because mandatory
    PRIMARY KEY(mid),
    FOREIGN KEY(sid) REFERENCES studio
        ON DELETE SET DEFAULT ON UPDATE CASCADE
);

CREATE TABLE actors (
    aid int,
    name varchar(15) not null,
    adress varchar(25) not null,
    PRIMARY KEY (aid)
);

CREATE TABLE acts(
    mid int not null,
    aid int not null,
    PRIMARY KEY (mid, aid),
    FOREIGN KEY (mid) REFERENCES movie
        ON DELETE CASCADE ON UPDATE CASCADE
    FOREIGN KEY (aid) REFERENCES actors
        ON DELETE CASCADE ON UPDATE CASCADE
);
```

Exercise 3

A company database needs to store information about:

- employees (identified by ssn, with salary and phone as attributes),
- departments (identified by dno, with dname and budget as attributes), and
- children of employees (with name and age as attributes).
- Employees work in departments;
- each department is managed by an employee;
- a child must be identified uniquely by name when the parent (who is an employee; assume that only one parent works for the company) is known.
- We are not interested in information about a child once the parent leaves the company.

Draw an ER diagram that captures this information.