Midterm Review

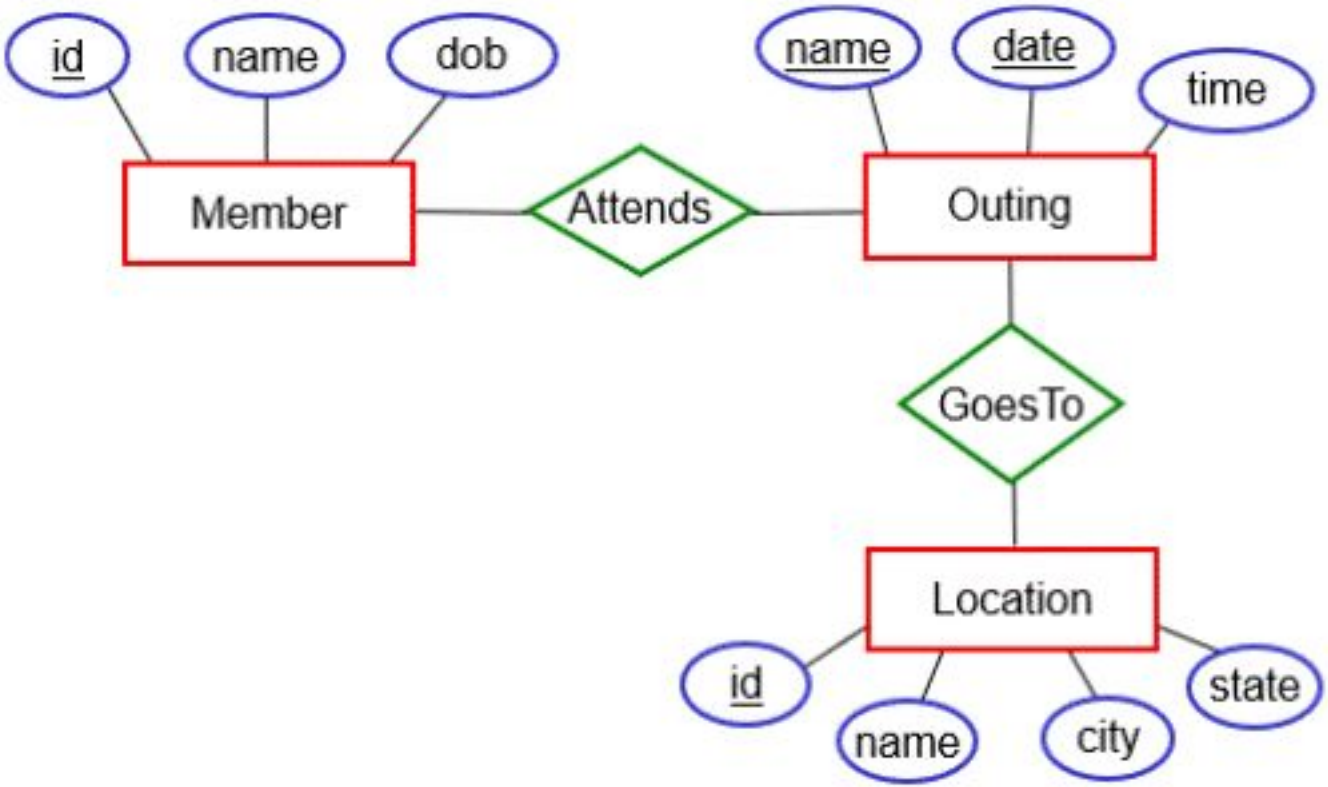
- Midterm: **Friday, October 10**
- The exam will cover everything up to and including the material on index structures (i.e., up to page 150 in the coursepack).
- **Resources**
  - Homework
  - Lab
  - Course Pack
  - Lecture Recordings
  - Pre-lecture Videos
  - **Midterm Review Problems**
  - **Midterm <u>Additional</u> Review Problems**

Which topic do you want to review?

- ER Diagrams
- Relational Schema
- Relational Algebra
- SQL Queries
- B-Tree
- B+Tree
- Record Formats
- Dynamic Linear Hashing

ER Diagrams and Relational Schema

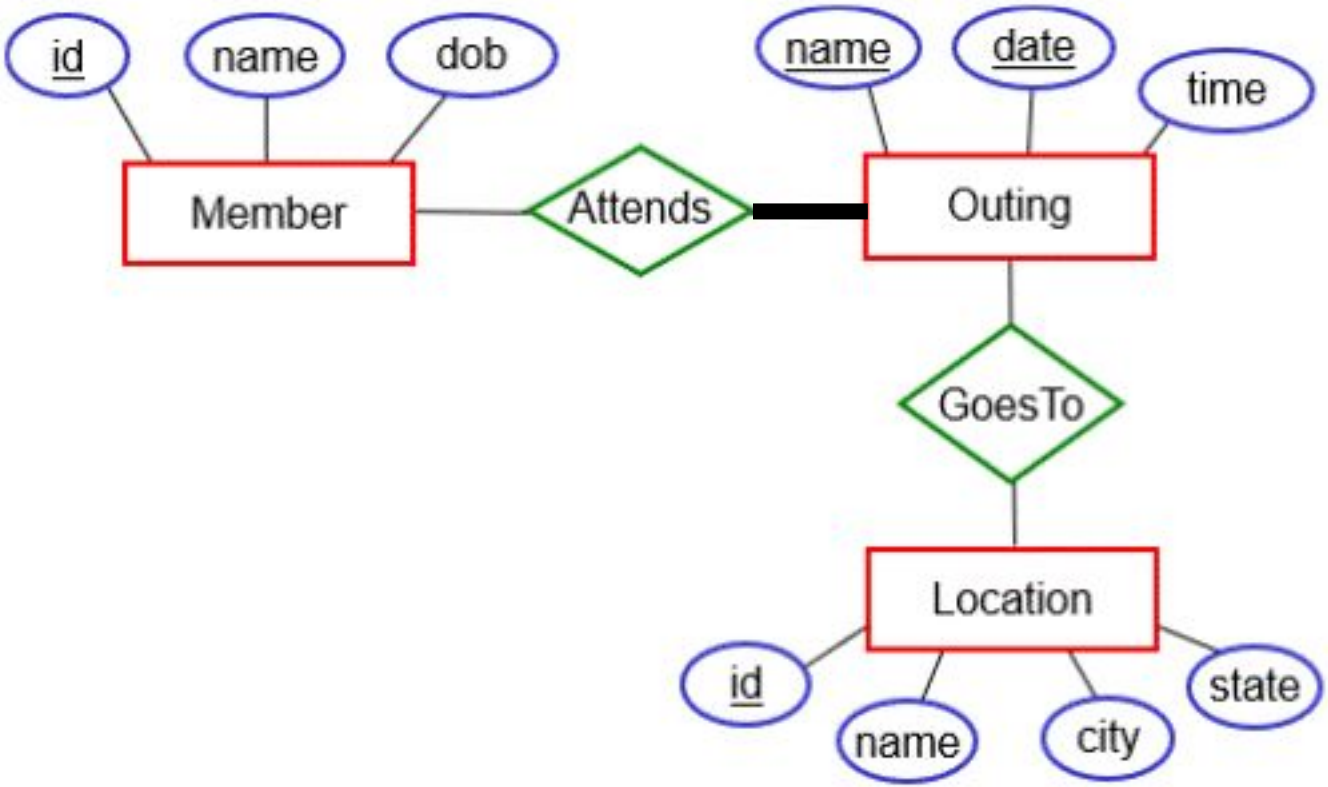
ER diagrams and the relational model



Represent:

Every outing is attended by **at least one** member

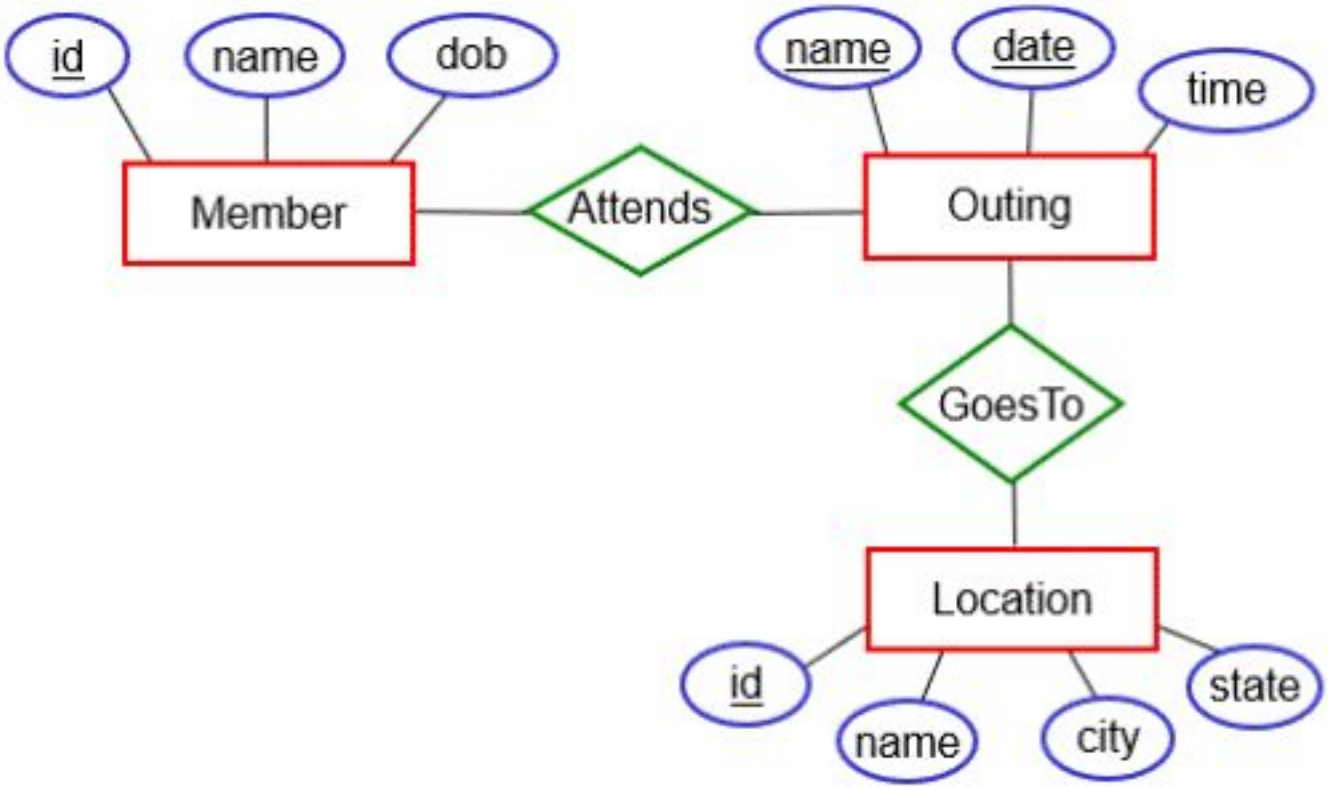
ER diagrams and the relational model



Represent:

Every outing is attended by **at least one** member

ER diagrams and the relational model



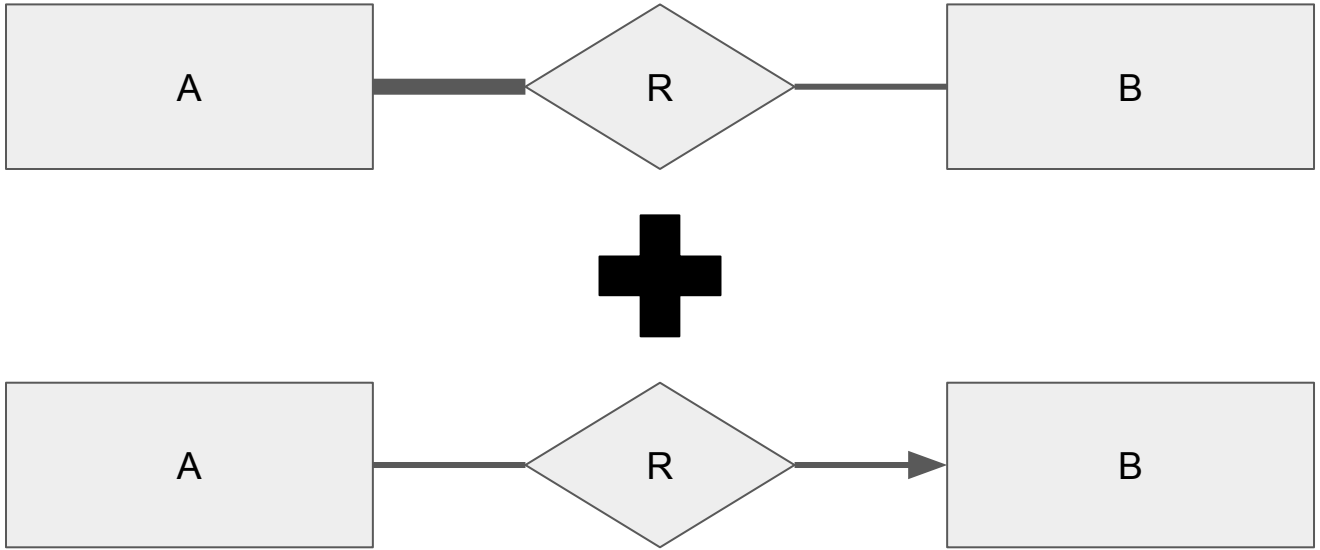
Represent:

Every outing goes to **exactly one** location.

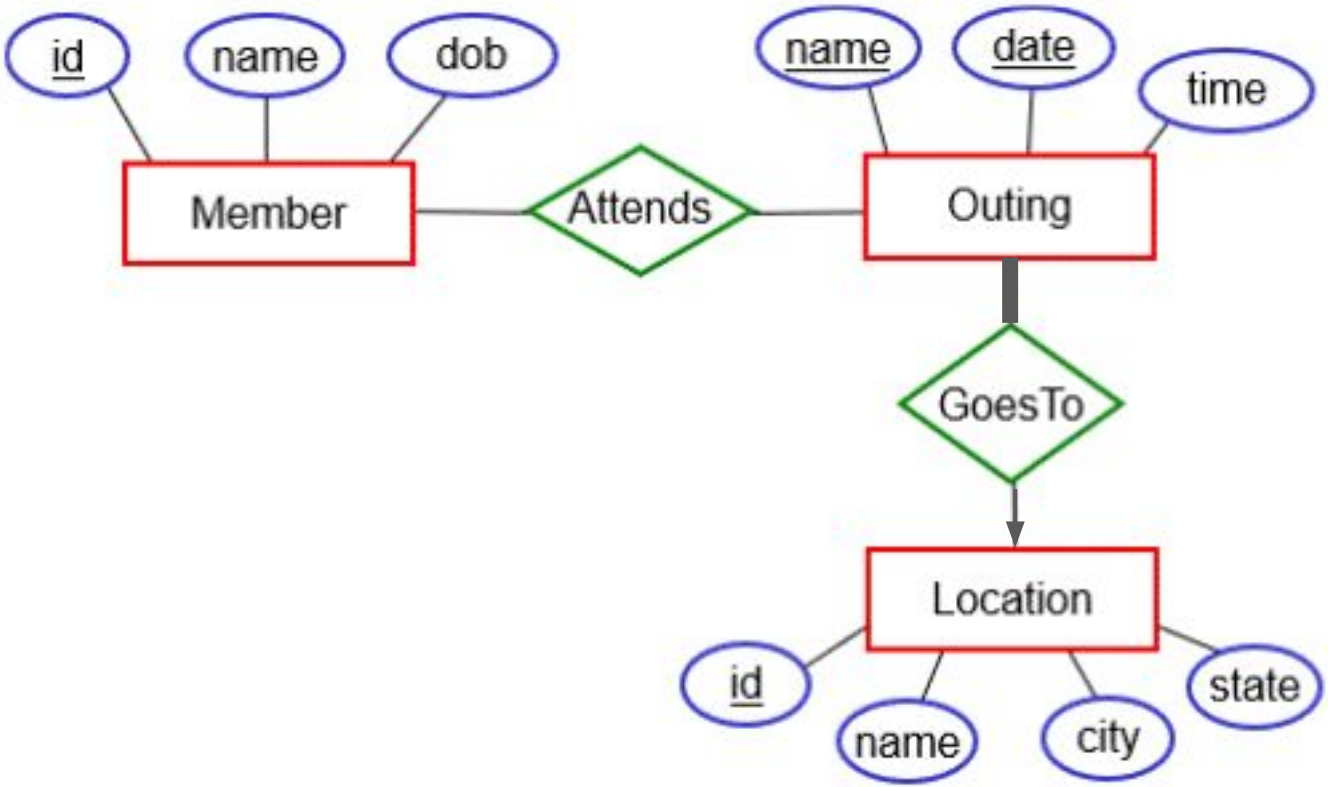
Exactly One

Every A is related to **exactly one** B =

Every A is related to **at least one** B + Every A is related to **at most one** B



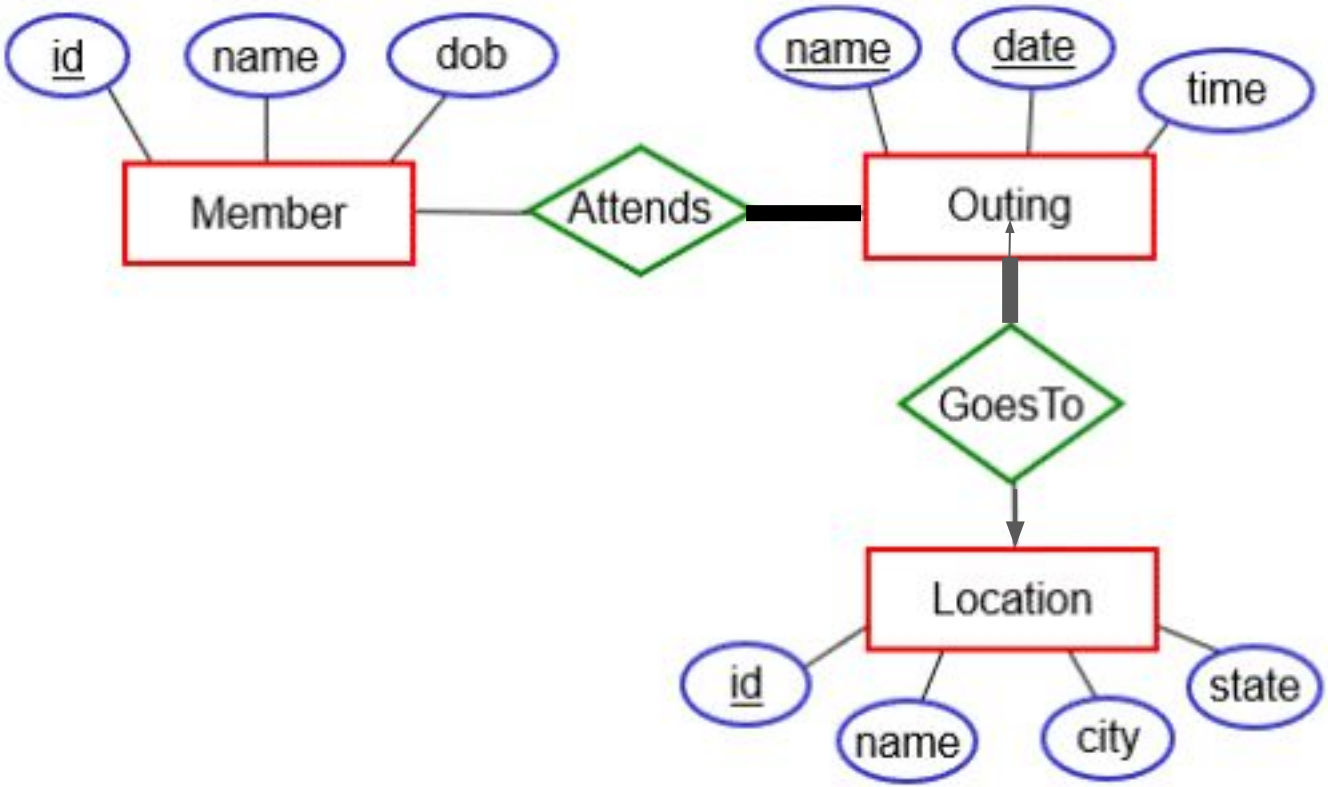
ER diagrams and the relational model



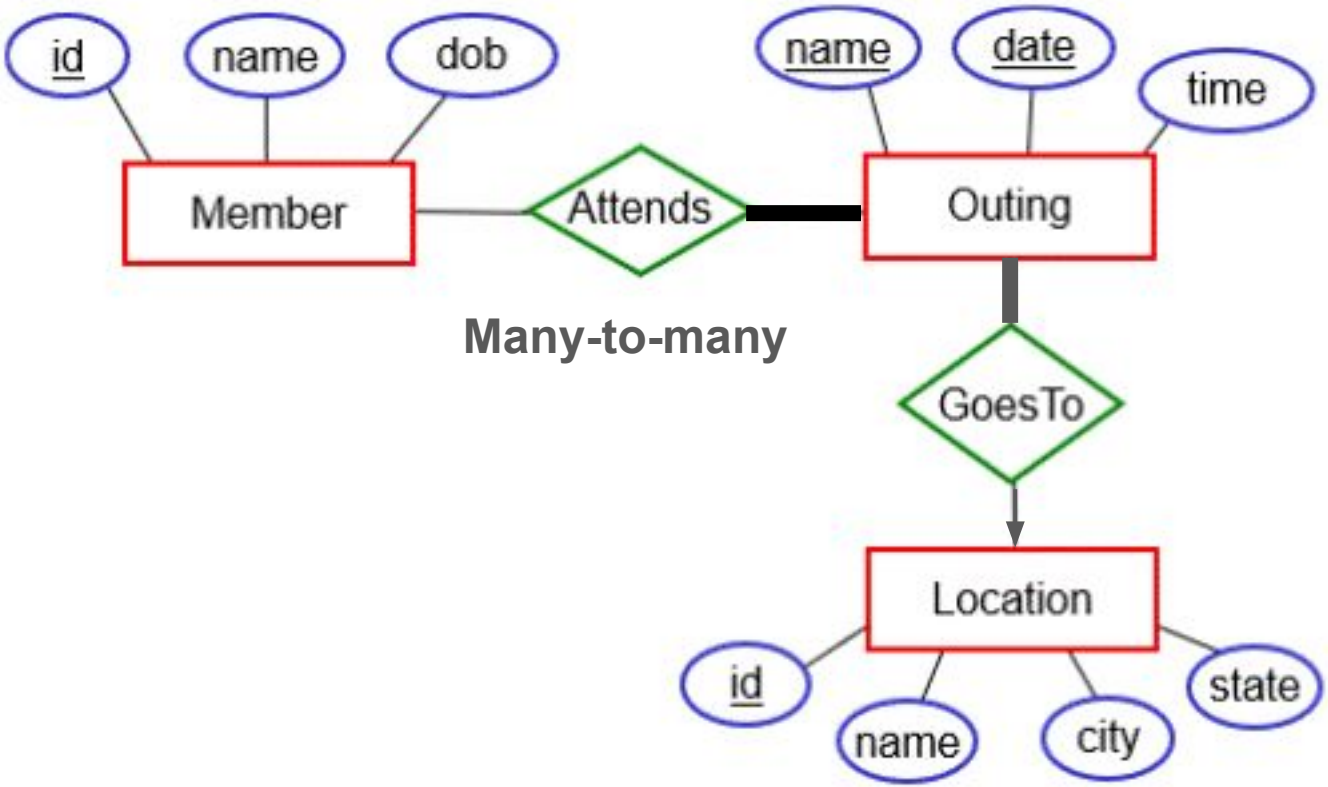
Represent:

Every outing goes to exactly one location.

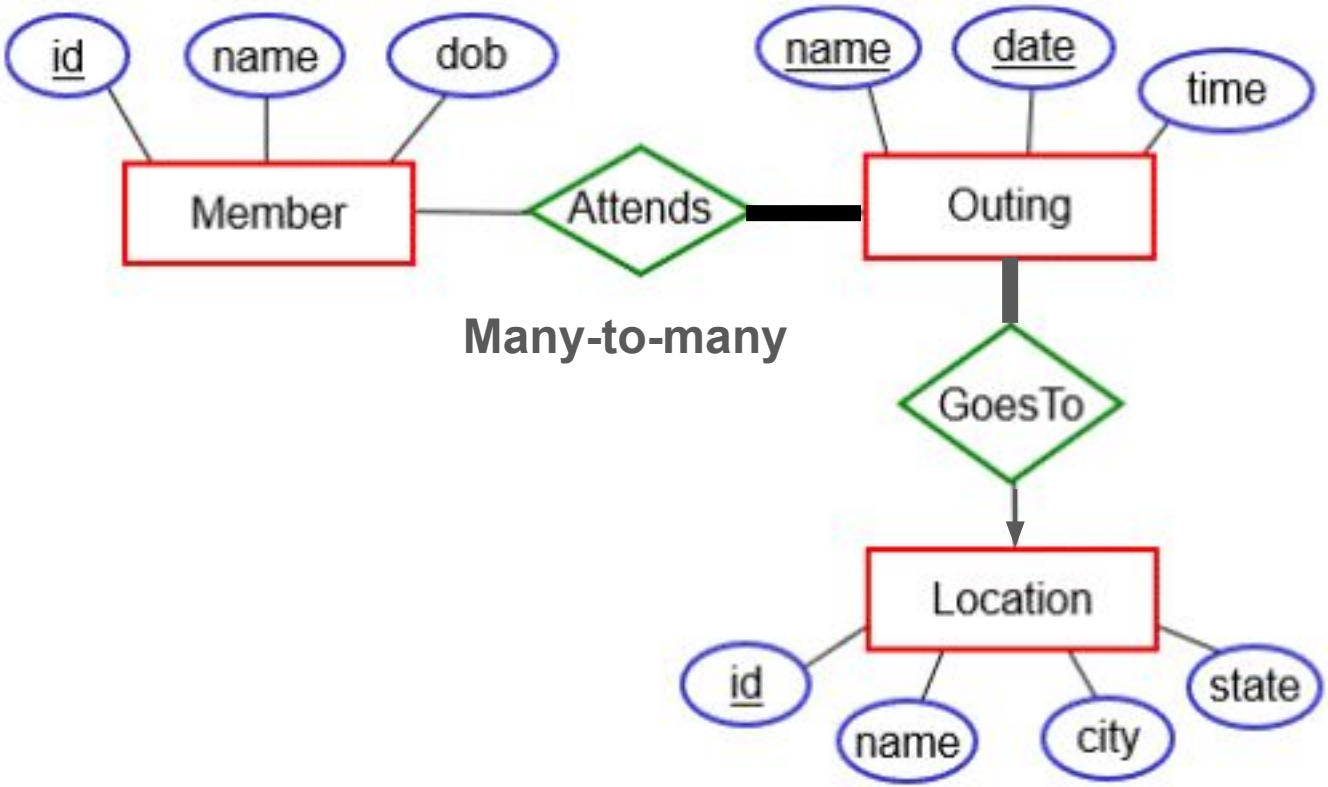
What type of relationship set is Attends?



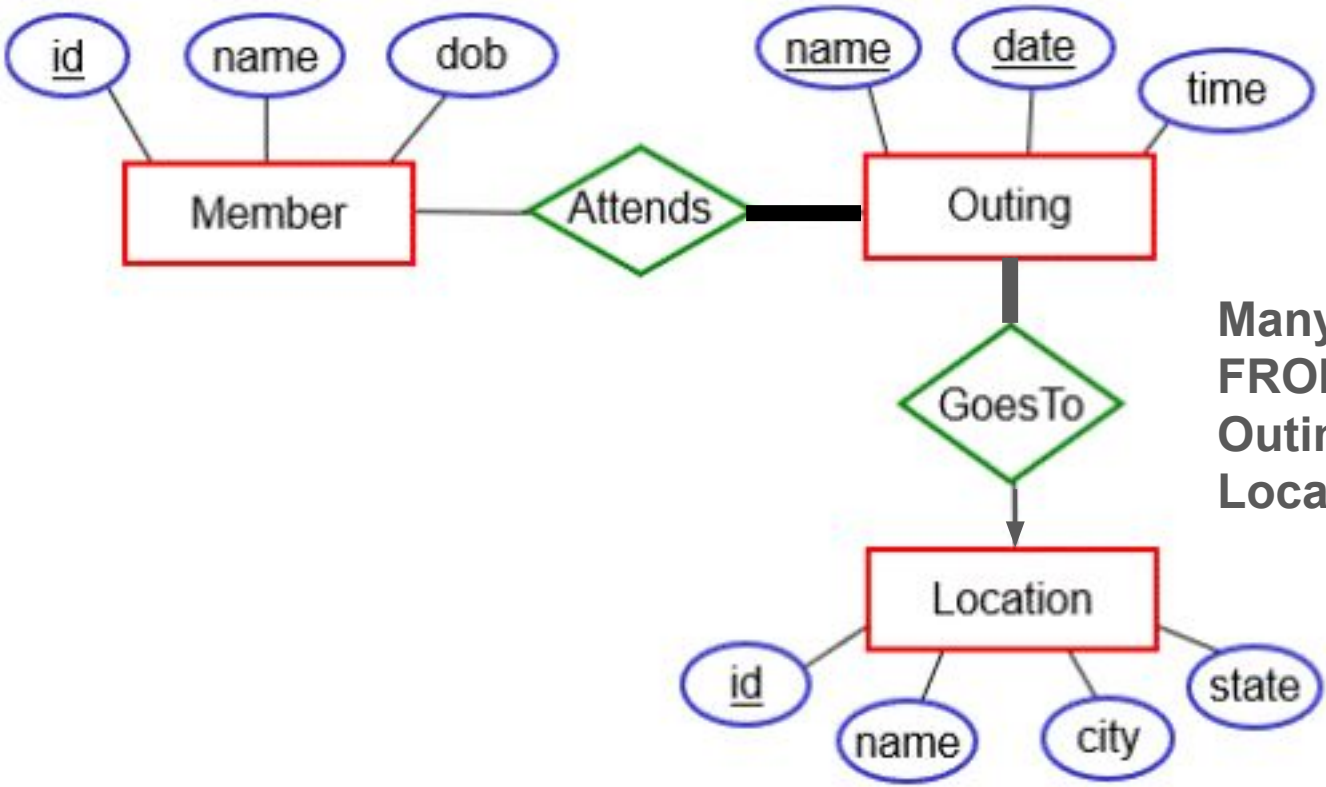
What type of relationship set is **Attends**?



What type of relationship set is **GoesTo**?

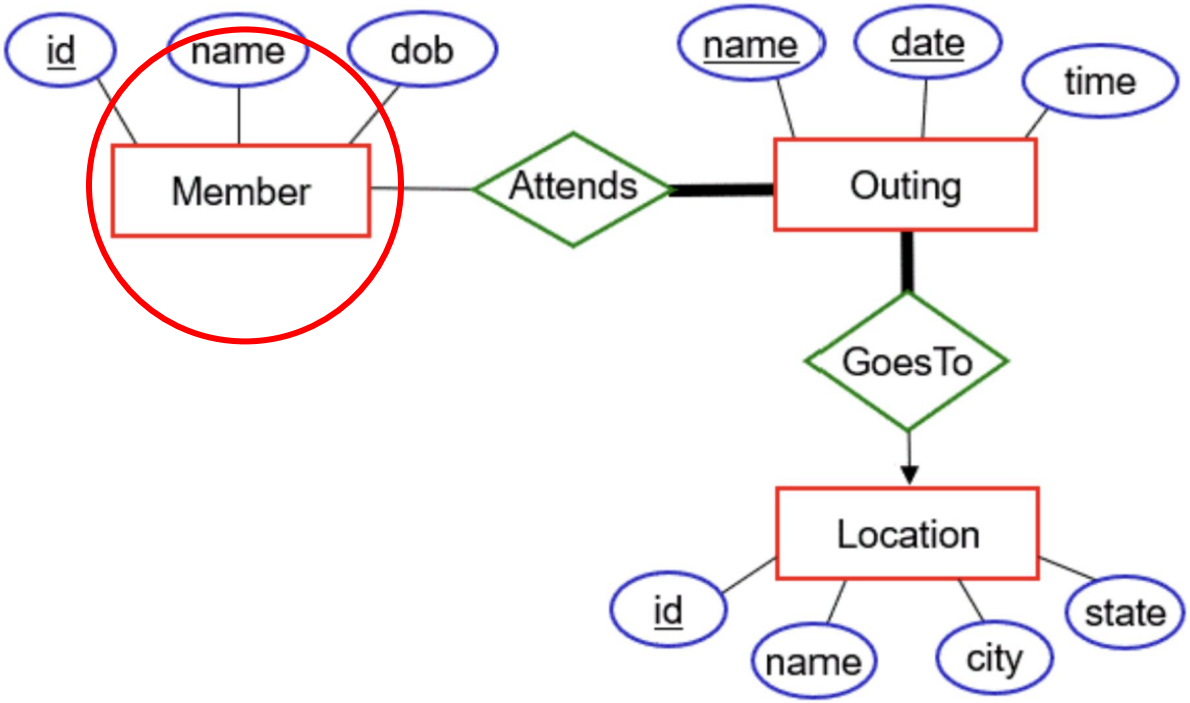


What type of relationship set is GoesTo?

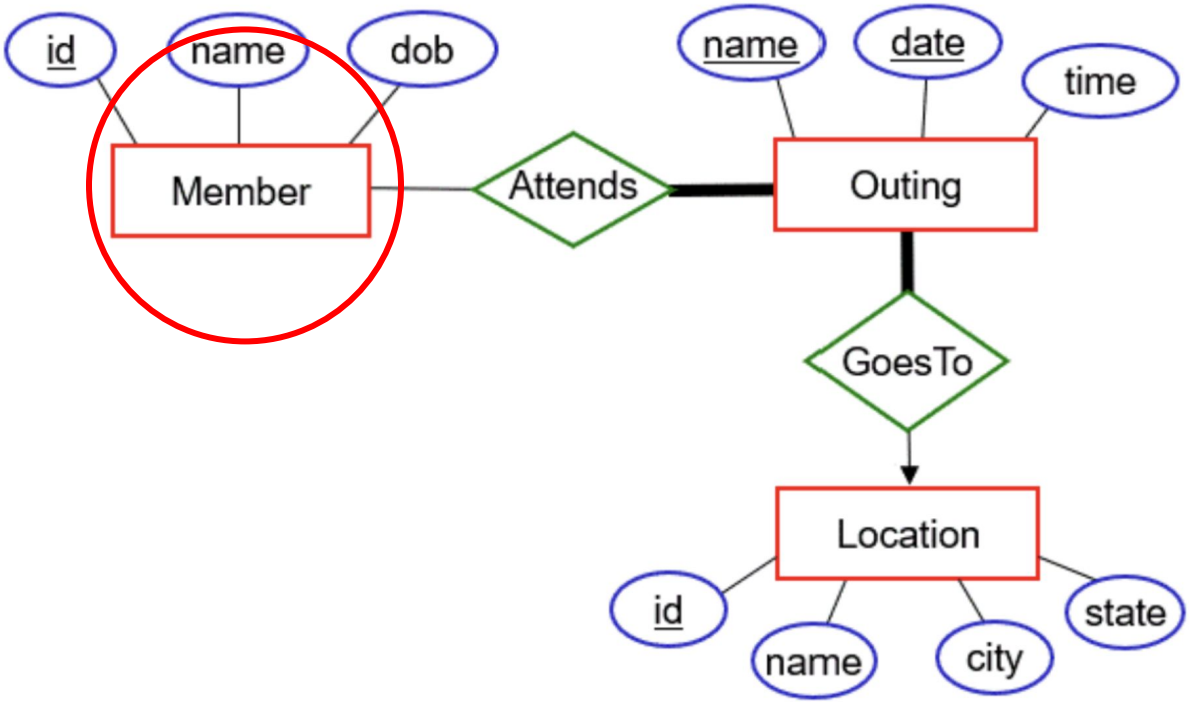


Many-to-one FROM Outing To Location

Turning ER Diagrams to Relational Schema

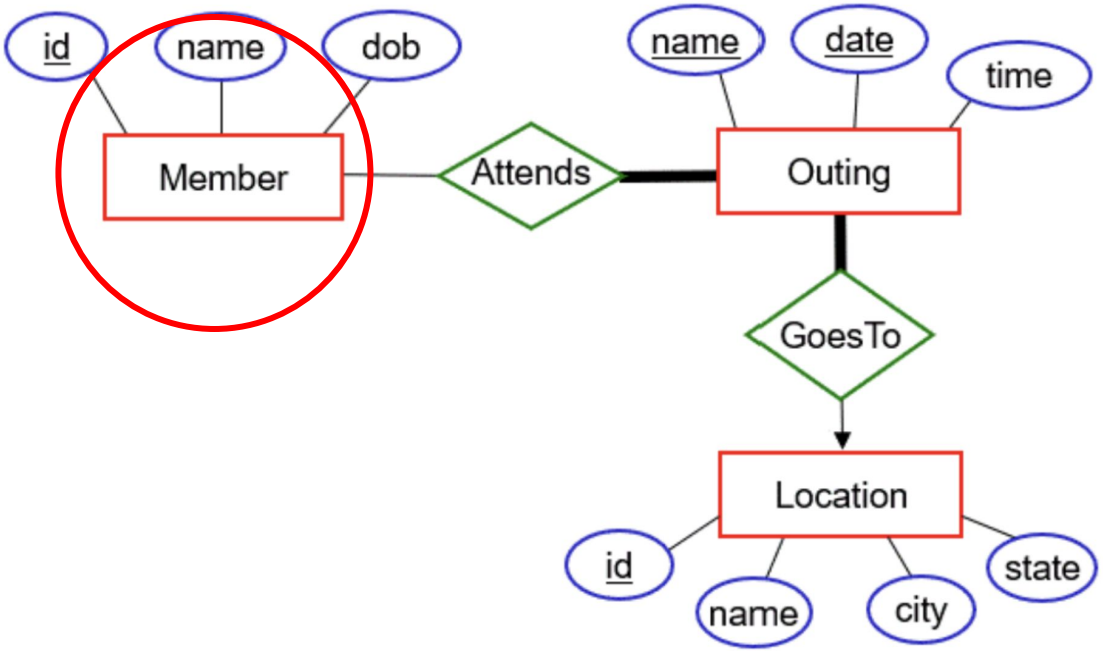


Turning ER Diagrams to Relational Schema



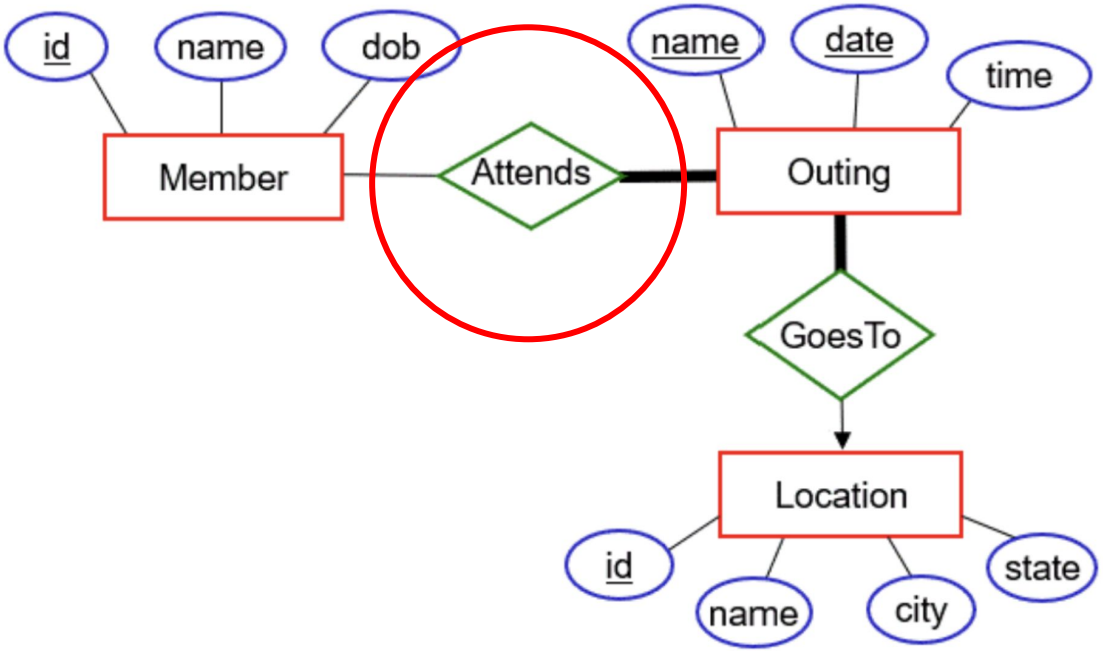
Can you capture Attends in member?

Turning ER Diagrams to Relational Schema



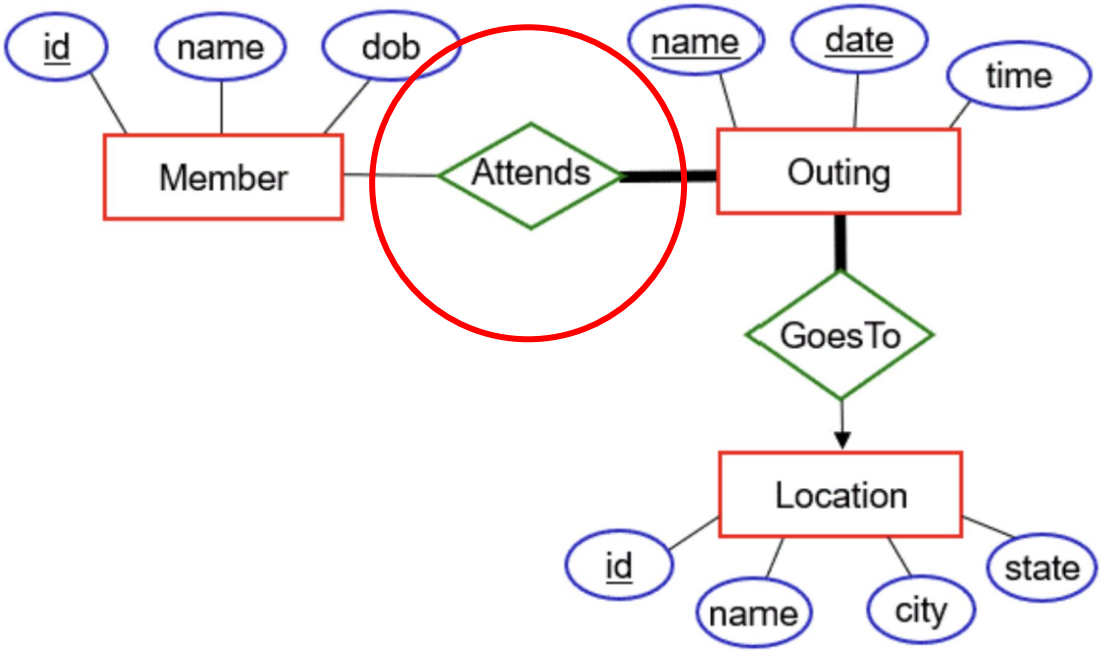
Member(id, name, dob)

Turning ER Diagrams to Relational Schema



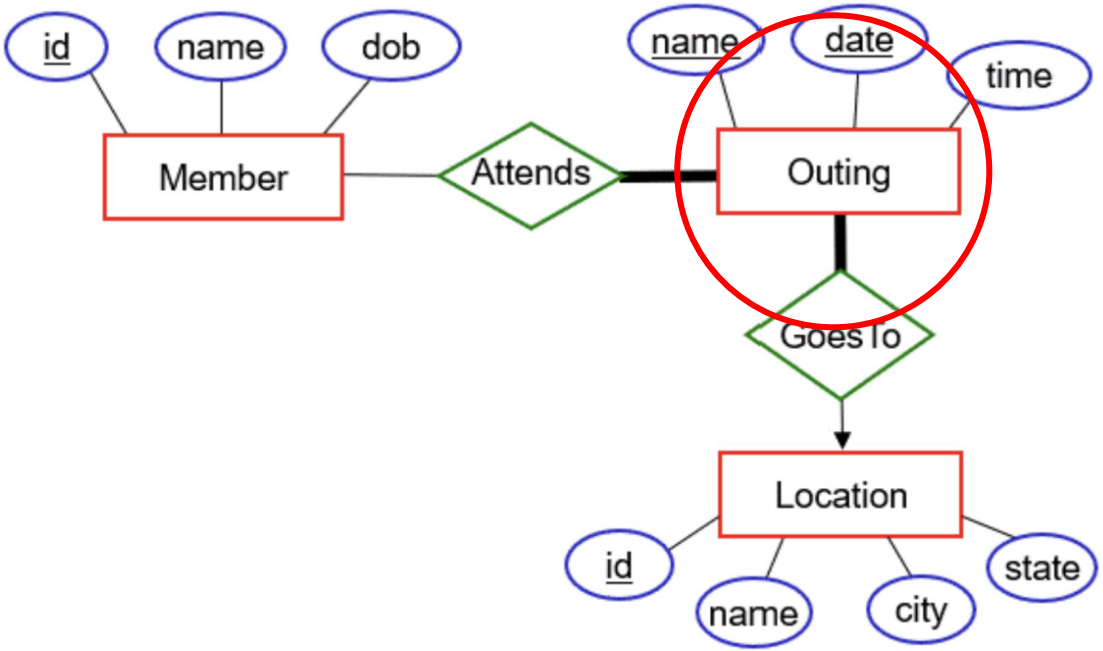
Member(id, name, dob)
Can Attends be captured in Outing?

Turning ER Diagrams to Relational Schema



Member(<u>id</u>, name, dob)
Attends(<u>Member_id,</u>
<u>Outing_name,</u>
<u>Outing_date</u>)

Turning ER Diagrams to Relational Schema

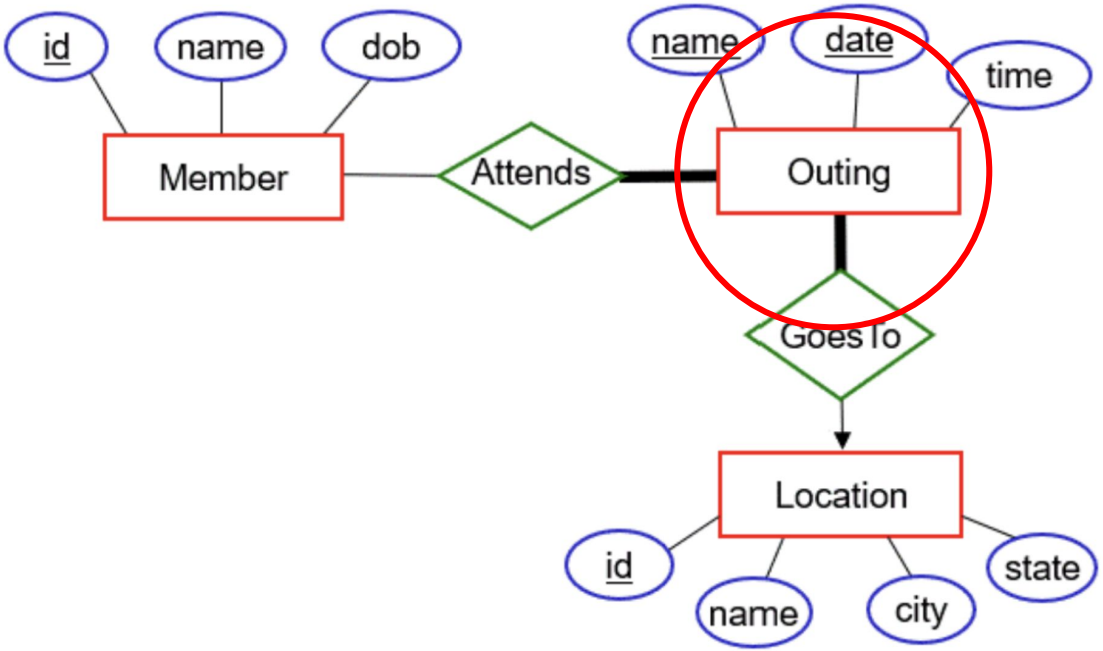


Member(id, name, dob)

Attends(Member_id, Outing_name, Outing_date)

Can we capture GoesTo in Outing?

Turning ER Diagrams to Relational Schema

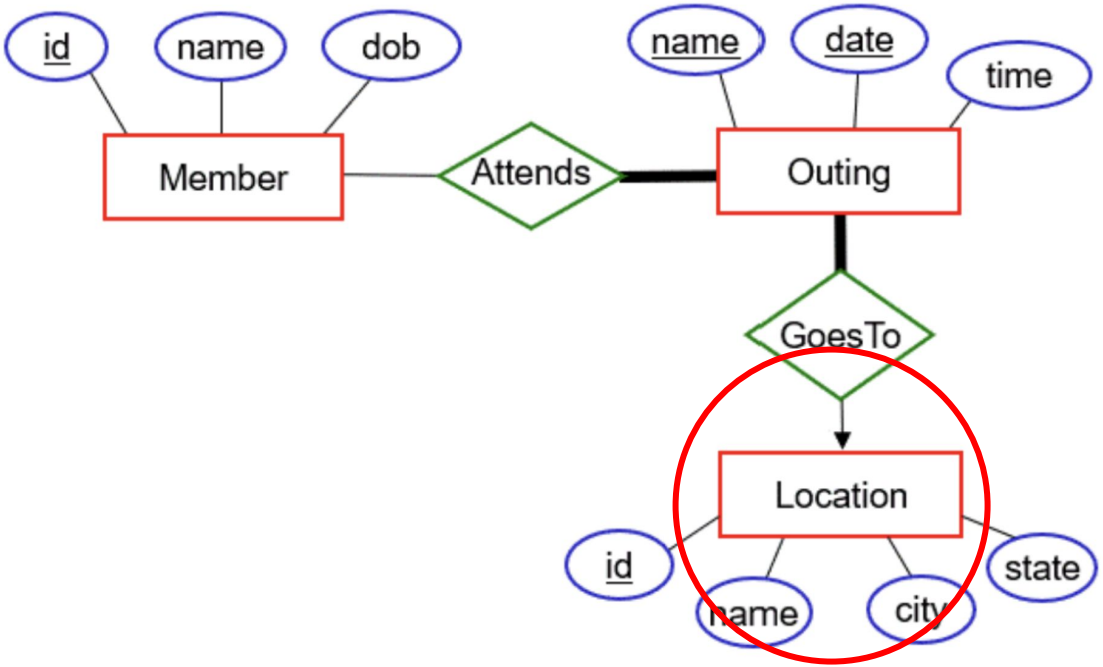


Member(id, name, dob)

Attends(Member_id, Outing_name, Outing_date)

Outing(name, date, time, Location_id)

Turning ER Diagrams to Relational Schema

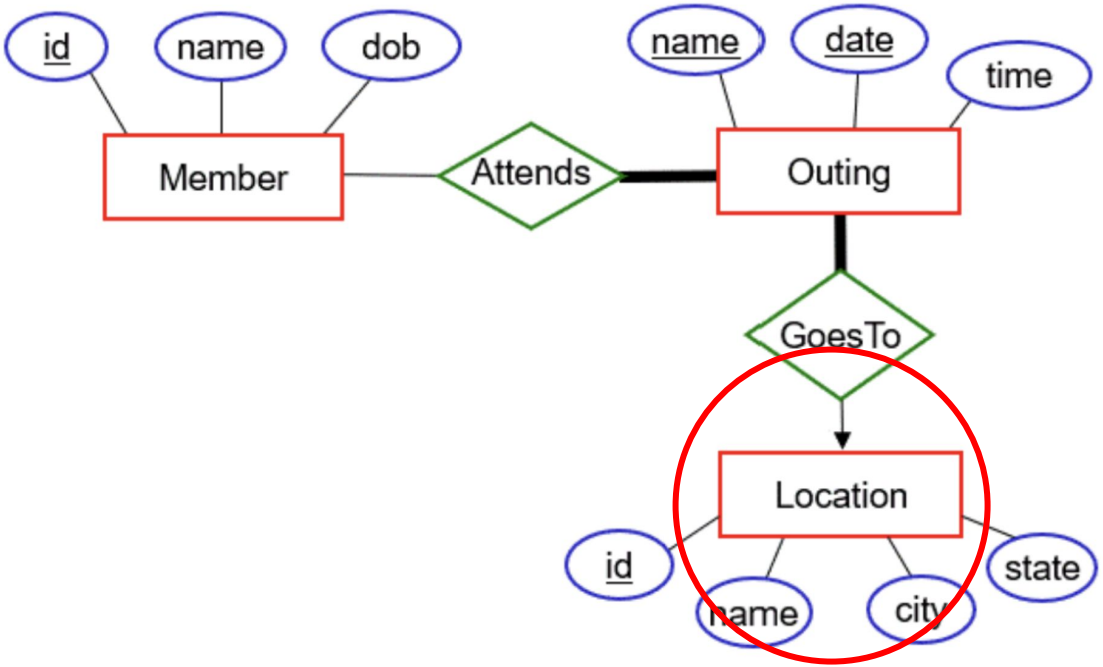


Member(id, name, dob)

Attends(Member_id, Outing_name, Outing_date)

Outing(name, date, time, Location_id)

Turning ER Diagrams to Relational Schema



Member(id, name, dob)

Attends(Member_id, Outing_name, Outing_date)

Outing(name, date, time, Location_id)

Location(id, name, city, state)

SQL Queries

Now Write the Queries!

Person(id, name, dob, pob)
Movie(id, name, year, rating, runtime, genre, earnings_rank)
Actor(actor_id, movie_id) Director(director_id, movie_id)
Oscar(movie_id, person_id, type, year)

2) Find the number of Oscars won by each person that has won an Oscar. Produce tuples of the form (name, num Oscars).

3) Find the number of Oscars won by each person, including people who have not won an Oscar.

Now Write the Queries!

Person(id, name, dob, pob)
Movie(id, name, year, rating, runtime, genre, earnings_rank)
Actor(actor_id, movie_id) Director(director_id, movie_id)
Oscar(movie_id, person_id, type, year)

2) Find the number of Oscars won by each person that has won an Oscar. Produce tuples of the form (name, num Oscars).

```
SELECT P.name, COUNT(O.type)
FROM Person P, Oscar O
WHERE P.id = O.person_id
GROUP BY P.name
```

3) Find the number of Oscars won by each person, including people who have not won an Oscar.

```
SELECT P.name, COUNT(O.type)
FROM Person P LEFT OUTER JOIN Oscar
ON P.id = Oscar.person_id
GROUP BY P.name
```

Don't do COUNT(*) if LEFT OUTER JOIN!

Practice Writing Queries

Student(id, name) Department(name, office) Room(id, name, capacity)
Course(name, start_time, end_time, room_id) MajorsIn(student_id, dept_name)
Enrolled(student_id, course_name, credit_status)

1) Find all rooms that can seat at least 100 people.

2) Find the course or courses with the earliest start time.

Practice Writing Queries

Student(id, name) Department(name, office) Room(id, name, capacity)
Course(name, start_time, end_time, room) MajorsIn(student, dept)
Enrolled(student, course, credit_status)

1) Find all rooms that can seat at least 100 people.

```
SELECT name
FROM Room
WHERE capacity >= 100;
```

2) Find the course or courses with the earliest start time.

```
SELECT name
FROM Course
WHERE start_time = (SELECT MIN(start_time)
                    FROM Course);
```

Practice Writing Queries (cont.)

Student(id, name) Department(name, office) Room(id, name, capacity)
Course(name, start_time, end_time, room_id) MajorsIn(student_id, dept_name)
Enrolled(student_id, course_name, credit_status)

3) Find the number of majors in each department.

4) Find all courses taken by CS ('comp sci') majors.

Practice Writing Queries (cont.)

Student(id, name) Department(name, office) Room(id, name, capacity)
Course(name, start_time, end_time, room) MajorsIn(student, dept)
Enrolled(student, course, credit_status)

3) Find the number of majors in each department.

```
SELECT dept, COUNT(*)
FROM MajorsIn
GROUP BY dept;
```

4) Find all courses taken by CS ('comp sci') majors.

```
SELECT DISTINCT course
FROM Enrolled E, MajorsIn M
WHERE E.student = M.student
  AND dept = 'comp sci';
```

Extra Practice Writing Queries (cont.)

Person(id, name, dob, pob)
Movie(id, name, year, rating, runtime, genre, earnings_rank)
Actor(actor_id, movie_id) Director(director_id, movie_id)
Oscar(movie_id, person_id, type, year)

5) Which movie ratings have an average runtime that is greater than 120 minutes, and what are their average runtimes?

Extra Practice Writing Queries (cont.)

Person(id, name, dob, pob)
Movie(id, name, year, rating, runtime, genre, earnings_rank)
Actor(actor_id, movie_id) Director(director_id, movie_id)
Oscar(movie_id, person_id, type, year)

5) Which movie ratings have an average runtime that is greater than 120 minutes, and what are their average runtimes?

```
SELECT rating, AVG(runtime)
FROM Movie
GROUP BY rating
HAVING AVG(runtime) > 120;
```

Relational Algebra

Practice Writing ~~Queries (cont.)~~ Relational Algebra

Student(id, name) Department(name, office) Room(id, name, capacity)
Course(name, start_time, end_time, room_id) MajorsIn(student_id, dept_name)
Enrolled(student_id, course_name, credit_status)

5) Create a list of all Students who are *not* enrolled in a course.

σ
$\pi_{id}(Student) - \pi_{student\_id}(Enrolled)$
⋈
ᐅ⋈
-

Practice Writing ~~Queries (cont.)~~ Relational Algebra

Student(id, name) Department(name, office) Room(id, name, capacity)
Course(name, start_time, end_time, room_id) MajorsIn(student_id, dept_name)
Enrolled(student_id, course_name, credit_status)

5) Create a list of all Students who are *not* enrolled in a course.

$$\text{NonEnrolledIDs} \leftarrow \pi_{id}(\text{Student}) - \pi_{student}(\text{Enrolled})$$

$$\pi_{name}(\text{Student} \bowtie \text{NonEnrolledIDs})$$

Practice Writing Queries (cont.)

Student(id, name) Department(name, office) Room(id, name, capacity)
Course(name, start_time, end_time, room_id) MajorsIn(student_id, dept_name)
Enrolled(student_id, course_name, credit_status)

7) Find the number of majors that each student has declared.

Instead of producing the count, produce tuples of the form (student id, student name, majoring department), including students without any major

Record Formats

Record formats

Imagine that we are working with a `Course` table that has the following schema:

```
Course(name VARCHAR(10) PRIMARY KEY, start_time CHAR(8), end_time CHAR(8), room_id CHAR(4))
```

Consider the following tuple from that table:

```
('CS 460', '13:25:00', '14:15:00', '0003')
```

1. If we use the fixed-length record format discussed in lecture, what would the record look like for this tuple? What would its length be? Assume that we're using one-byte characters.

Record formats

Imagine that we are working with a `Course` table that has the following schema:

```
Course(name VARCHAR(10) PRIMARY KEY, start_time CHAR(8), end_time CHAR(8), room_id CHAR(4))
```

Consider the following tuple from that table:

```
('CS 460', '13:25:00', '14:15:00', '0003')
```

1. If we use the fixed-length record format discussed in lecture, what would the record look like for this tuple? What would its length be? Assume that we're using one-byte characters.

Record formats

Imagine that we are working with a `Course` table that has the following schema:

```
Course(name VARCHAR(10) PRIMARY KEY, start_time CHAR(8), end_time CHAR(8), room_id CHAR(4))
```

Consider the following tuple from that table:

```
('CS 460', '13:25:00', '14:15:00', '0003')
```

2. Now assume that we're using the second type of variable-length record discussed in lecture, in which each record begins with a header of field offsets. What will the record look like for the above tuple, and what will its length be? In addition to one-byte characters, you should assume that we use **two-byte** integers for integer *metadata* like offsets.

Record formats

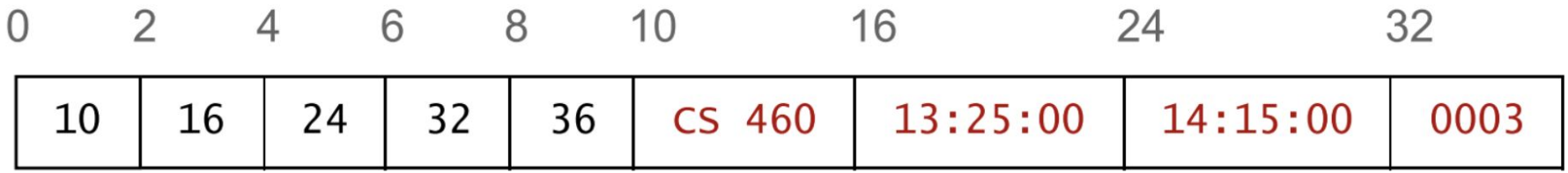
Imagine that we are working with a `Course` table that has the following schema:

```
Course(name VARCHAR(10) PRIMARY KEY, start_time CHAR(8), end_time CHAR(8), room_id CHAR(4))
```

Consider the following tuple from that table:

```
('CS 460', '13:25:00', '14:15:00', '0003')
```

2. Now assume that we're using the second type of variable-length record discussed in lecture, in which each record begins with a header of field offsets. What will the record look like for the above tuple, and what will its length be? In addition to one-byte characters, you should assume that we use **two-byte** integers for integer *metadata* like offsets.



Dynamic Linear Hashing

```
h(k) = k % 10
```

- We grow the table whenever the number of items (*f*) becomes more than twice the number of buckets (*n*).

```
0 [1000, 972]
1 [713]
```

1. an item (i.e., a key-value pair) whose key is 12

```
h(k) = k % 10
```

- We grow the table whenever the number of items (*f*) becomes more than twice the number of buckets (*n*).

```
0 [1000, 972]
1 [713]
```

1. an item (i.e., a key-value pair) whose key is 12

   h(12) = 2 = 0010. The rightmost bit is 0, so add 12 to 0.

```
0 [1000, 972, 12]
1 [713]
```

```
h(k) = k % 10
```

- We grow the table whenever the number of items (*f*) becomes more than twice the number of buckets (*n*).

```
0 [1000, 972]
1 [713]
```

1. an item (i.e., a key-value pair) whose key is 12

h(12) = 2 = 0010. The rightmost bit is 0, so add 12 to 0.

```
0 [1000, 972, 12]
1 [713]
```

f = 4, n = 2
Add bucket when
f > 2n
f > 4
*Add bucket on
next insert

```
0 [1000, 972, 12]
1 [713]
```

Insert 436

```
0 [1000, 972, 12]
1 [713]
```

Insert 436

436 % 10 = 6 = 011<u>0</u>

```
0 [1000, 972, 12, 436]
1 [713]
```

```
0 [1000, 972, 12]
1 [713]
```

Insert 436

436 % 10 = 6 = 011<u>0</u>

```
0 [1000, 972, 12, 436]
1 [713]
```

f > 2n, add bucket

```
0 [1000, 972, 12]
1 [713]
```

Insert 436

436 % 10 = 6 = 011<u>0</u>

```
0 [1000, 972, 12, 436]
1 [713]
```

f > 2n, add bucket

```
00 [1000, 972, 12, 436]
01 [713]
10 []
```

```
0 [1000, 972, 12]
1 [713]
```

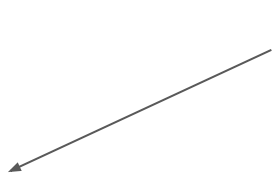
Insert 436

436 % 10 = 6 = 011<u>0</u>

```
0 [1000, 972, 12, 436]
1 [713]
```

f > 2n, add bucket

rehash!



```
00 [1000, 972, 12, 436]
01 [713]
10 []
```

```
00 [1000, 972, 12, 436]
01 [713]
10 []
```

```
00 [1000] (because 1000 hashes to 0000, or 00)
01 [713] (we don't need to look at the contents of this bucket at all)
10 [972, 12, 436] (972 and 12 hash to 0010; 436 hashes to 0110 or 10)
```

00 [1000]
01 [713]
10 [972, 12, 436]

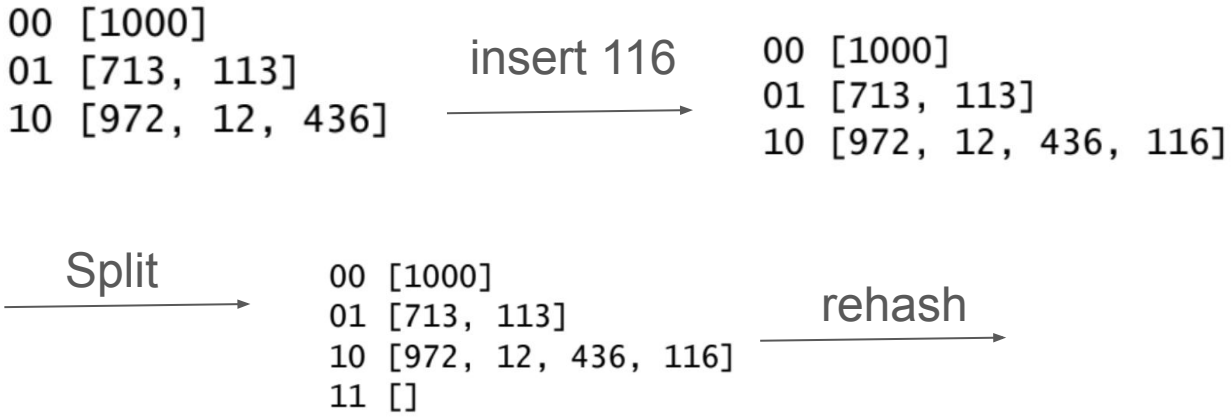
insert 113

00 [1000]
01 [713]
10 [972, 12, 436]

insert 113

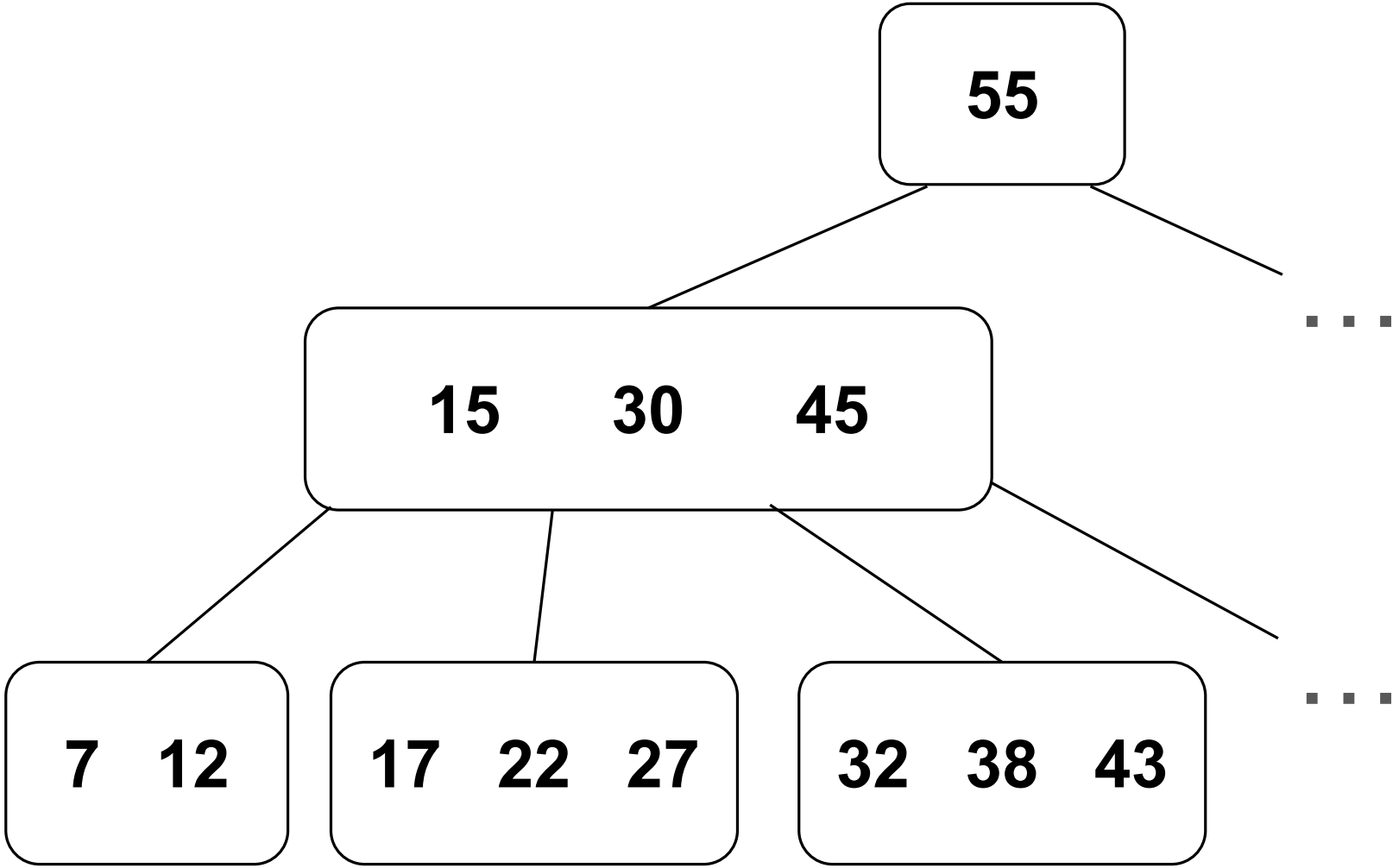
113 % 10 = 3
3 = 0011

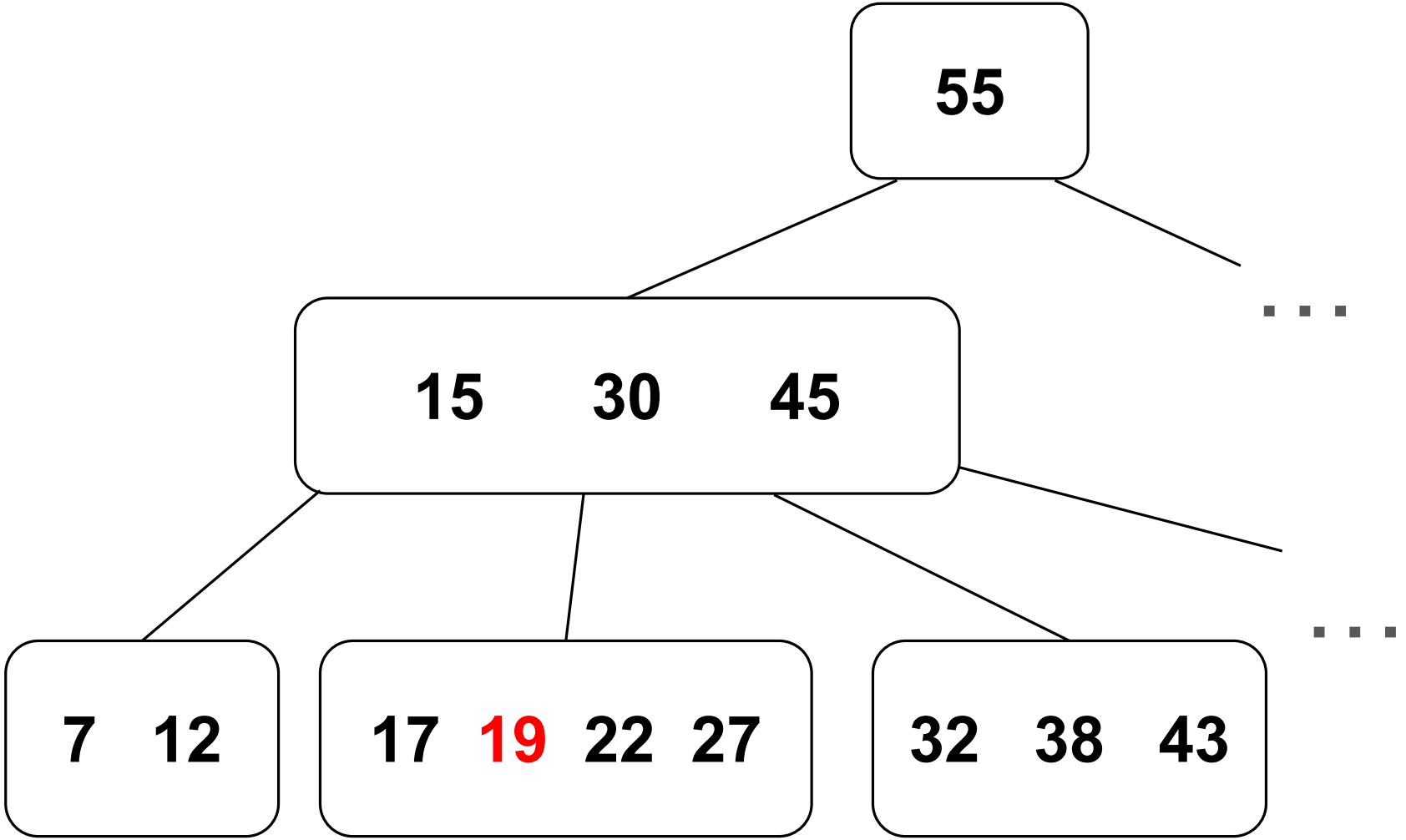
00 [1000]
01 [713, 113]
10 [972, 12, 436]

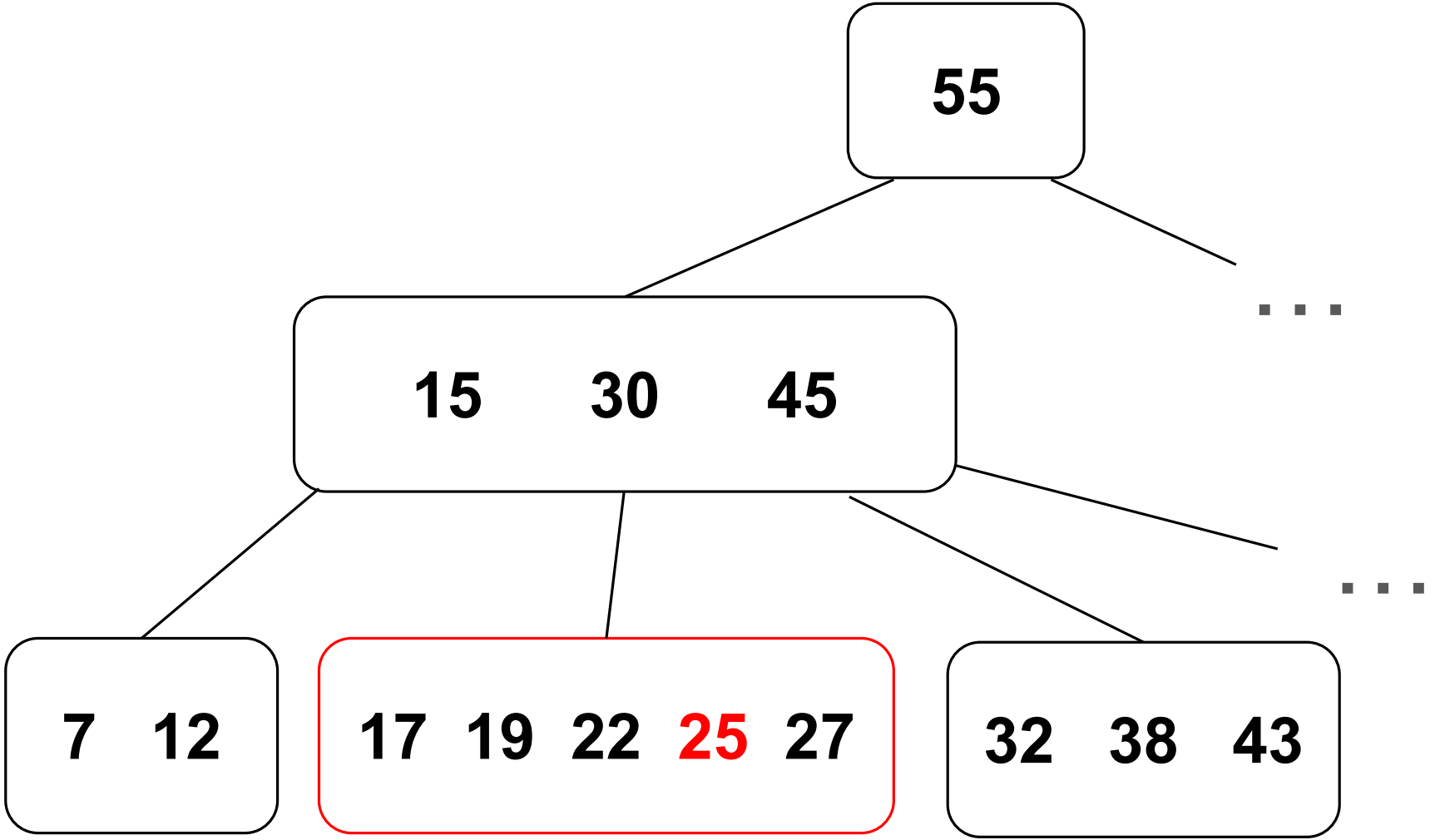


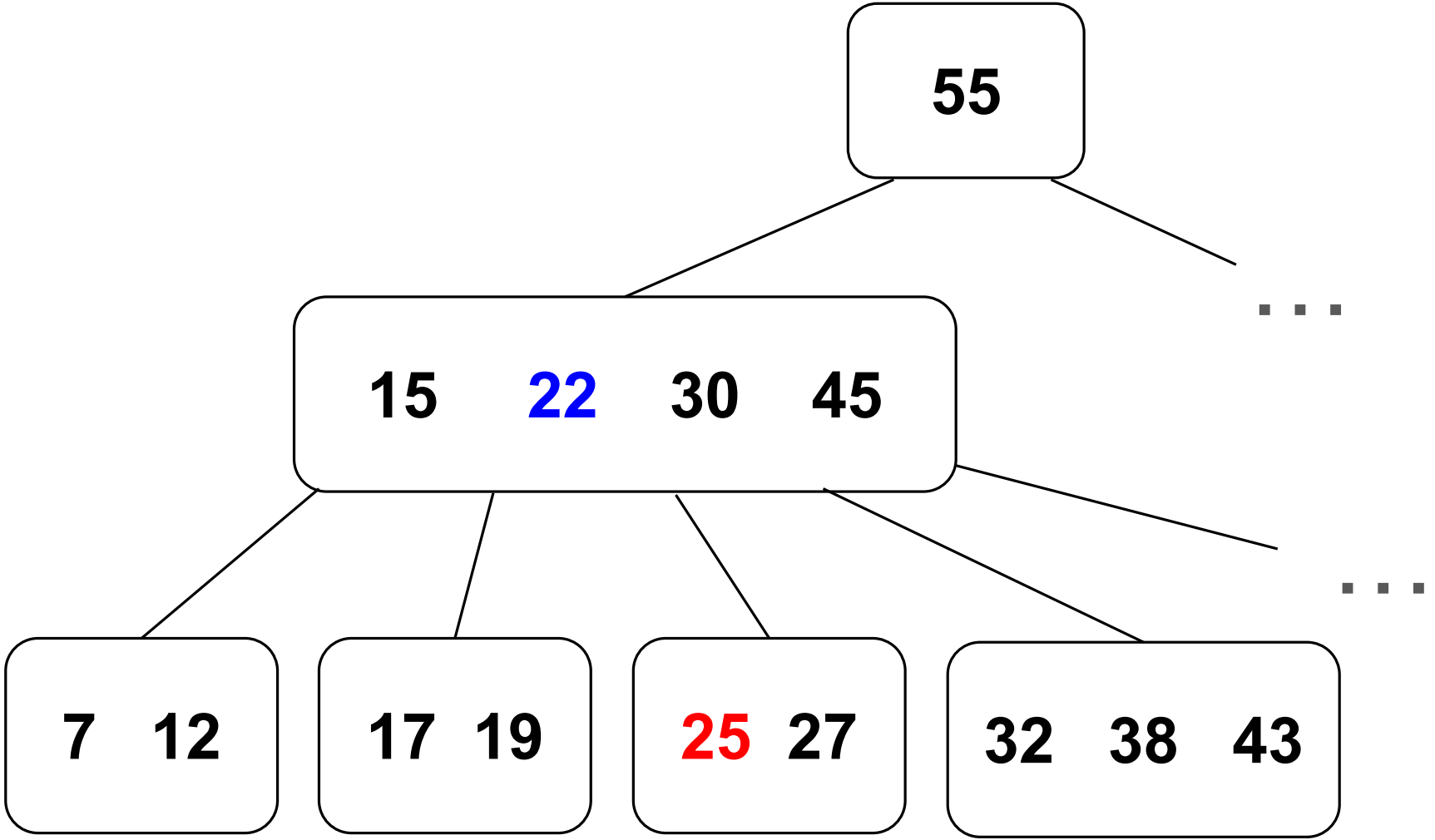
00 [1000] (we don't need to look at the contents of this bucket at all)
01 [] (nothing remains after rehashing!)
10 [972, 12, 436, 116] (we don't need to look at the contents of this bucket at all)
11 [713, 113] (they both hash to 0011 or 11)

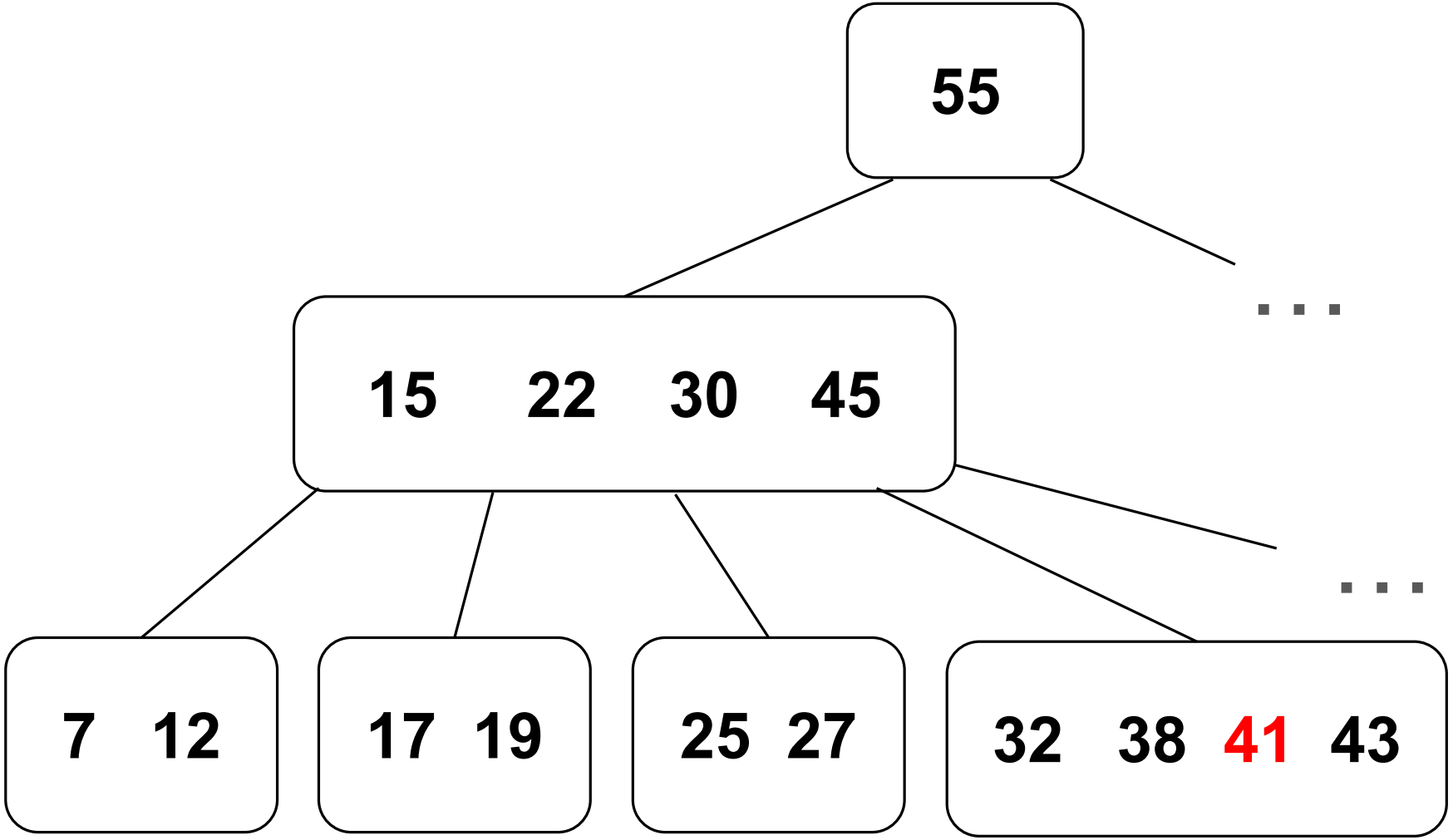
B-Trees and B+Trees

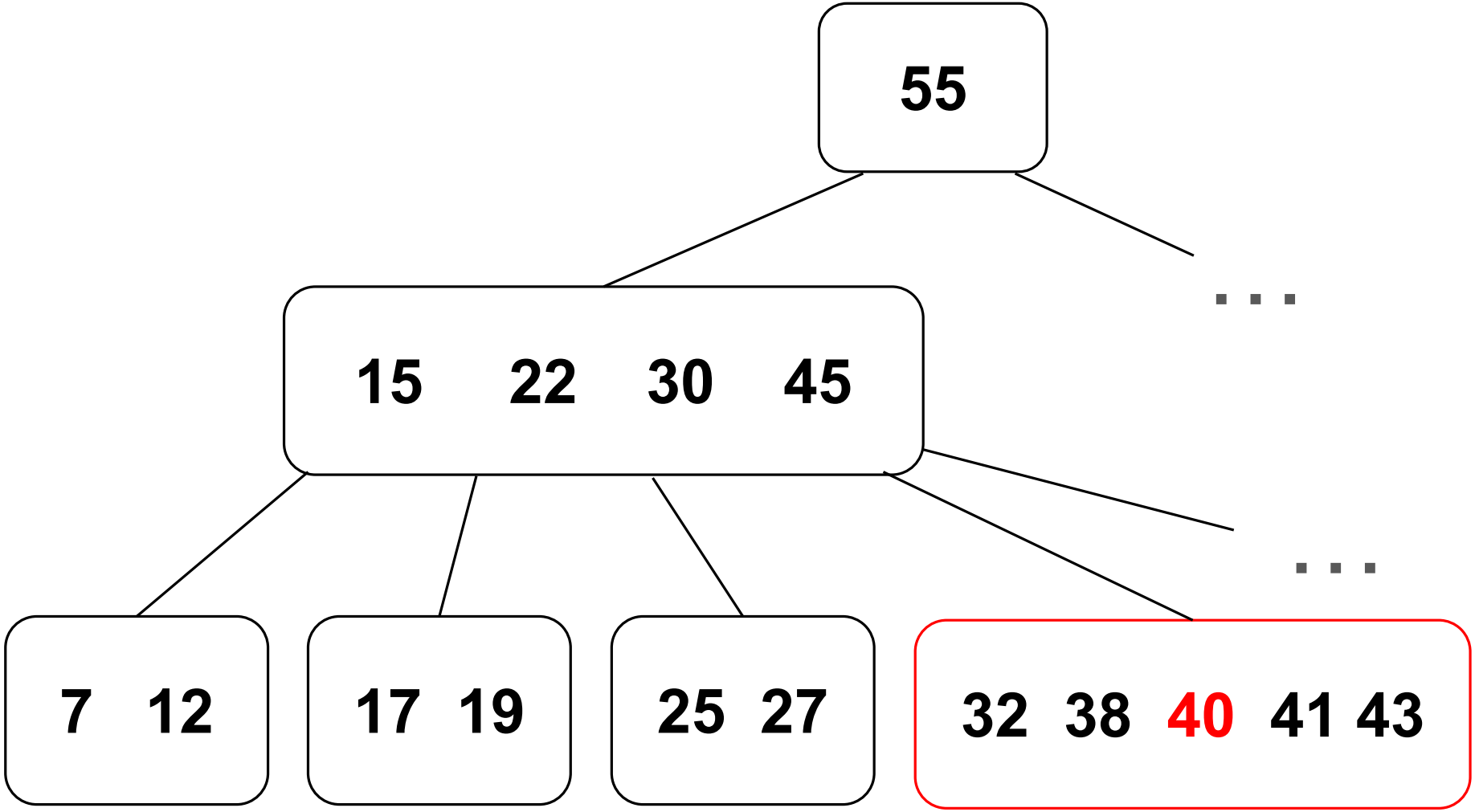

55
15 30 45
7 12
17 22 27
32 38 43

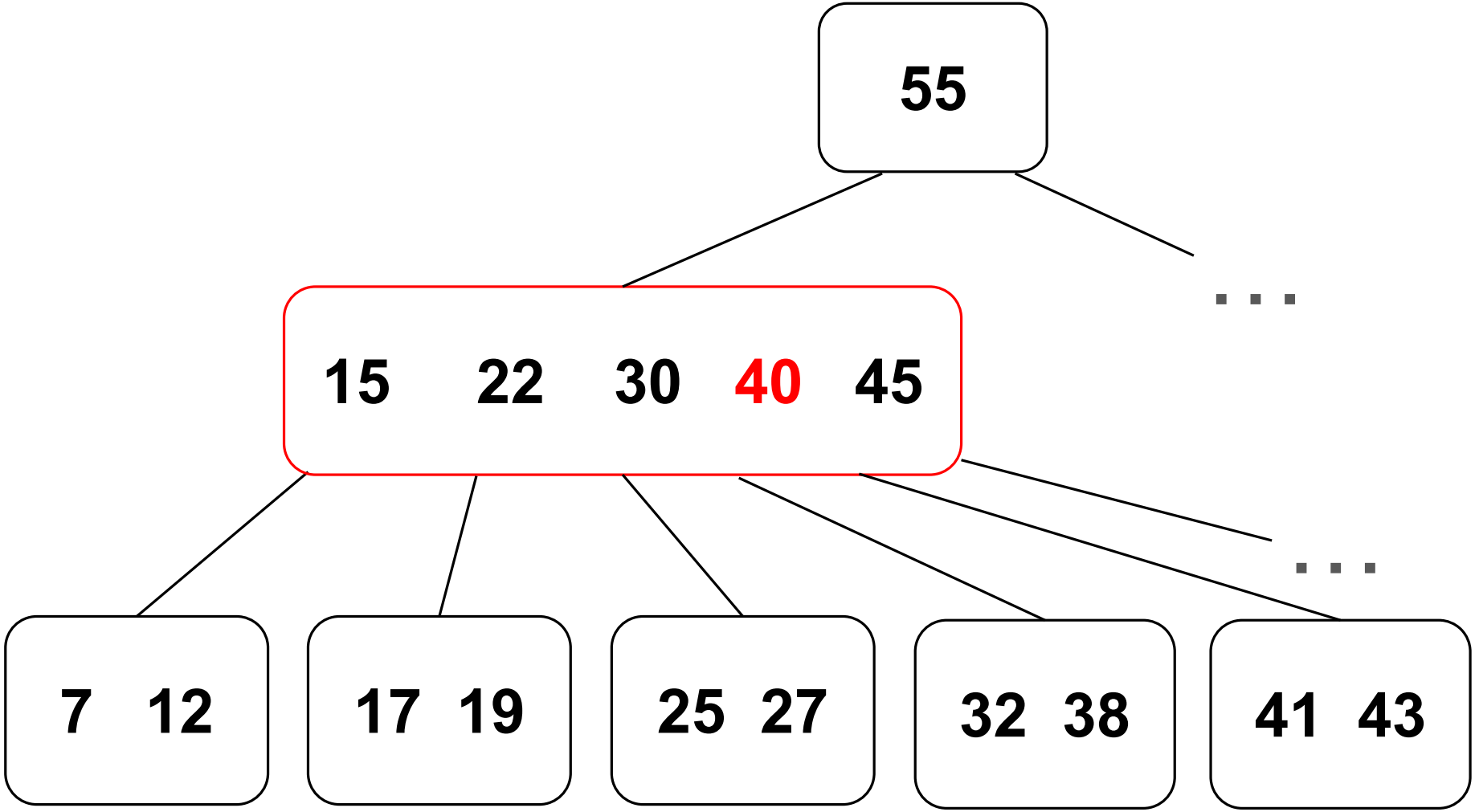

55
15 30 45
7 12
17 19 22 27
32 38 43

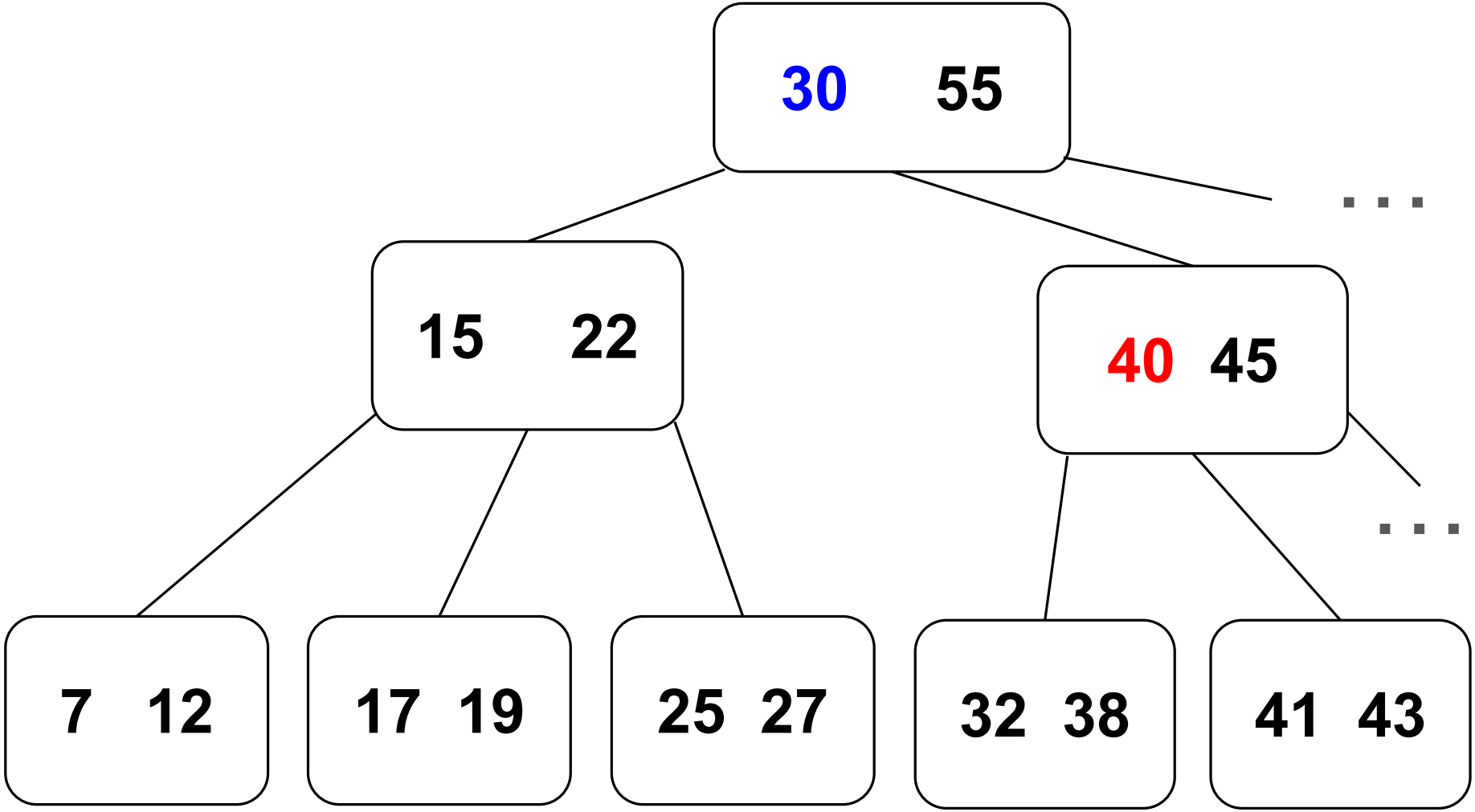

55
15 30 45
7 12
17 19 22 25 27
32 38 43


55
15 22 30 45
7 12
17 19
25 27
32 38 43


55
15 22 30 45
7 12
17 19
25 27
32 38 41 43

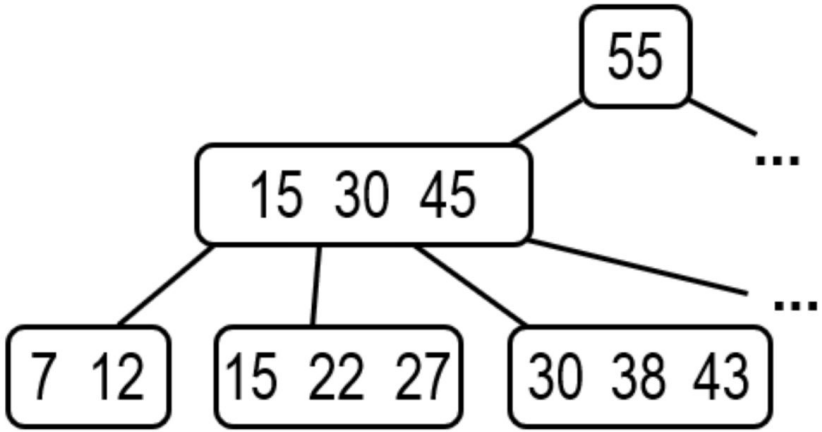

55
15 22 30 45
7 12
17 19
25 27
32 38 40 41 43


55
15 22 30 40 45
7 12
17 19
25 27
32 38
41 43


30 55
. . .
15 22
40 45
. . .
7 12
17 19
25 27
32 38
41 43

Task 3: Perform insertions in a B+tree

Consider the following diagram, which shows a portion of a B+tree (note the + symbol!) of order 2:

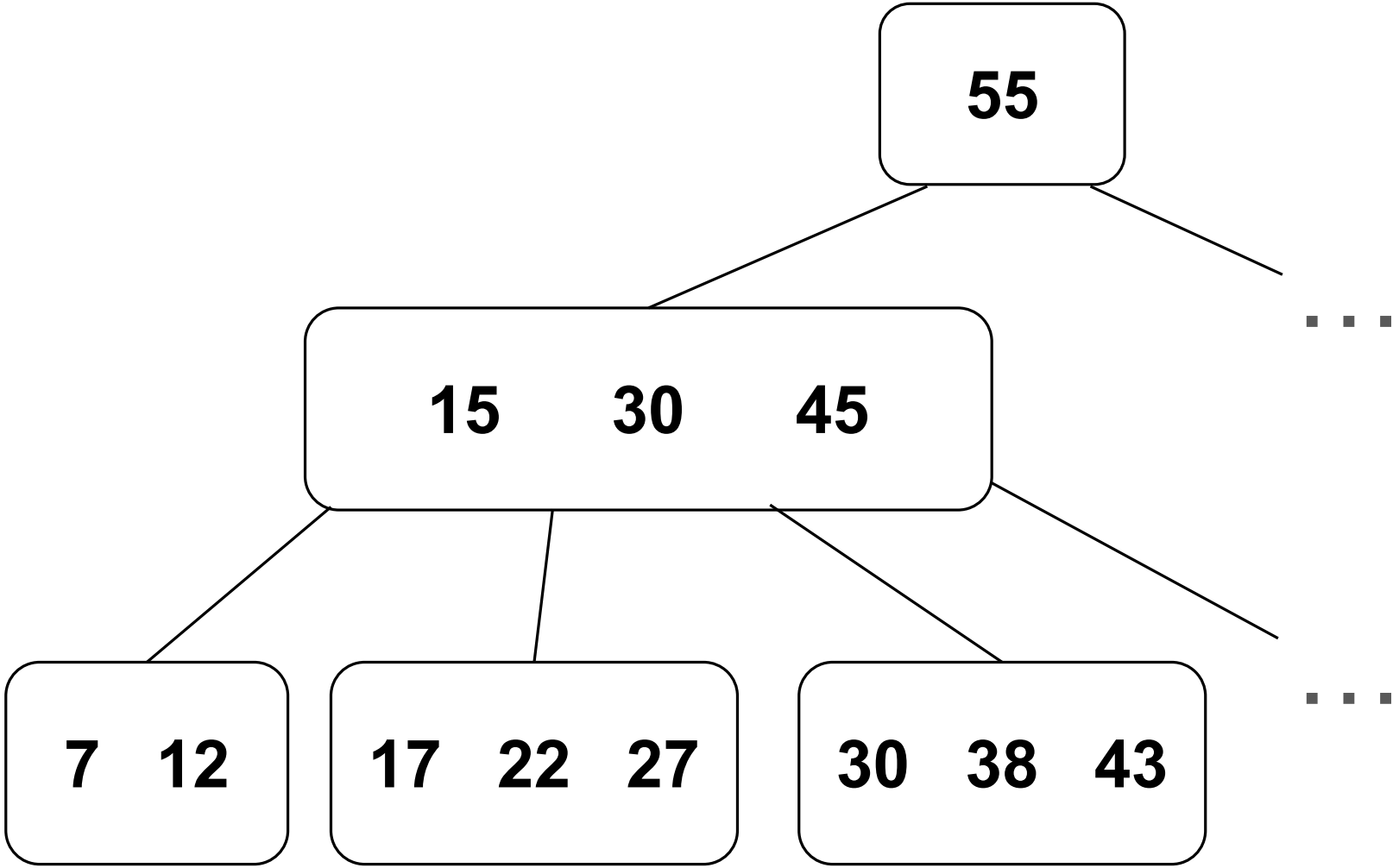


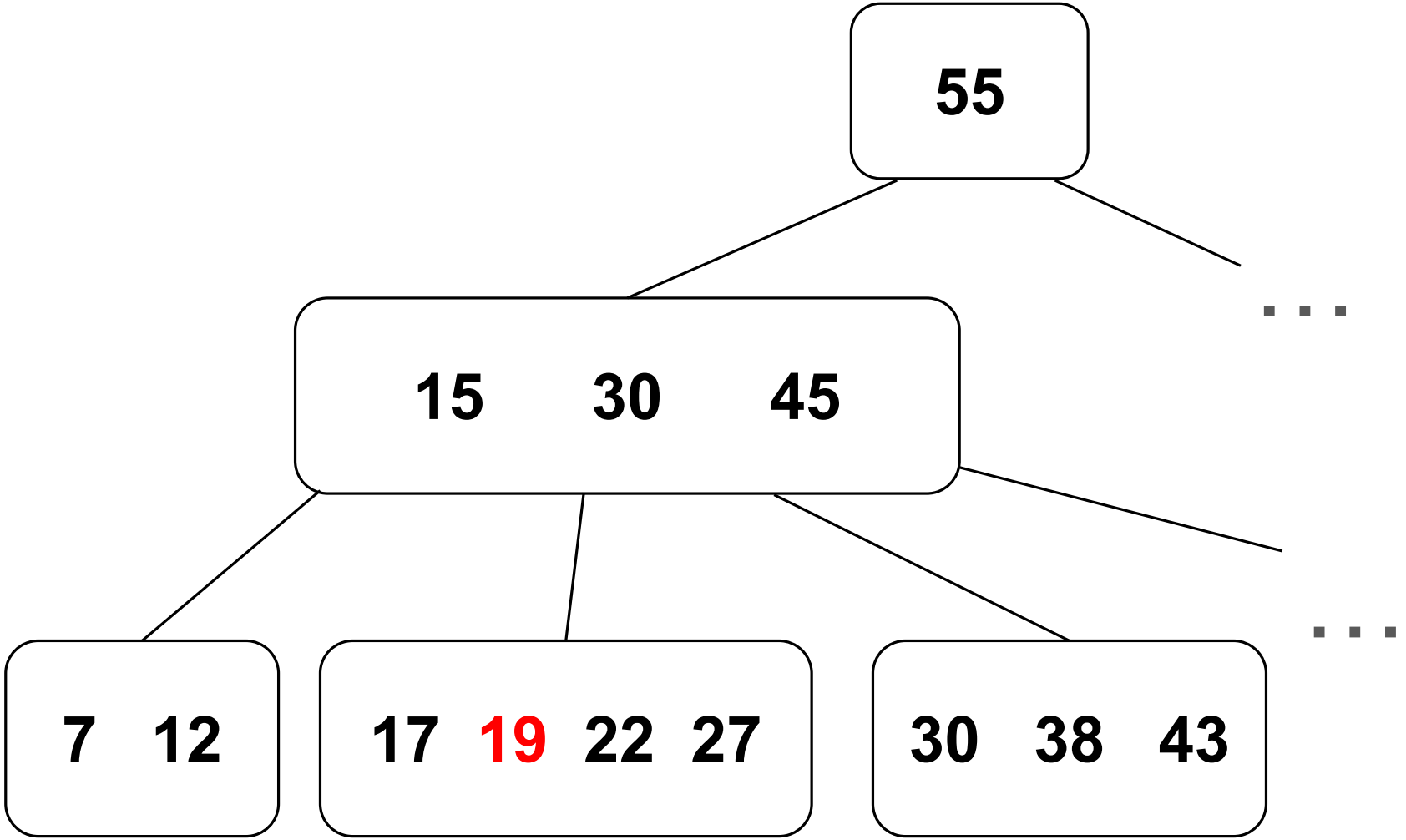
Note that this tree is similar to the initial tree in Task 2, but some keys are missing, and other keys appear in two places.

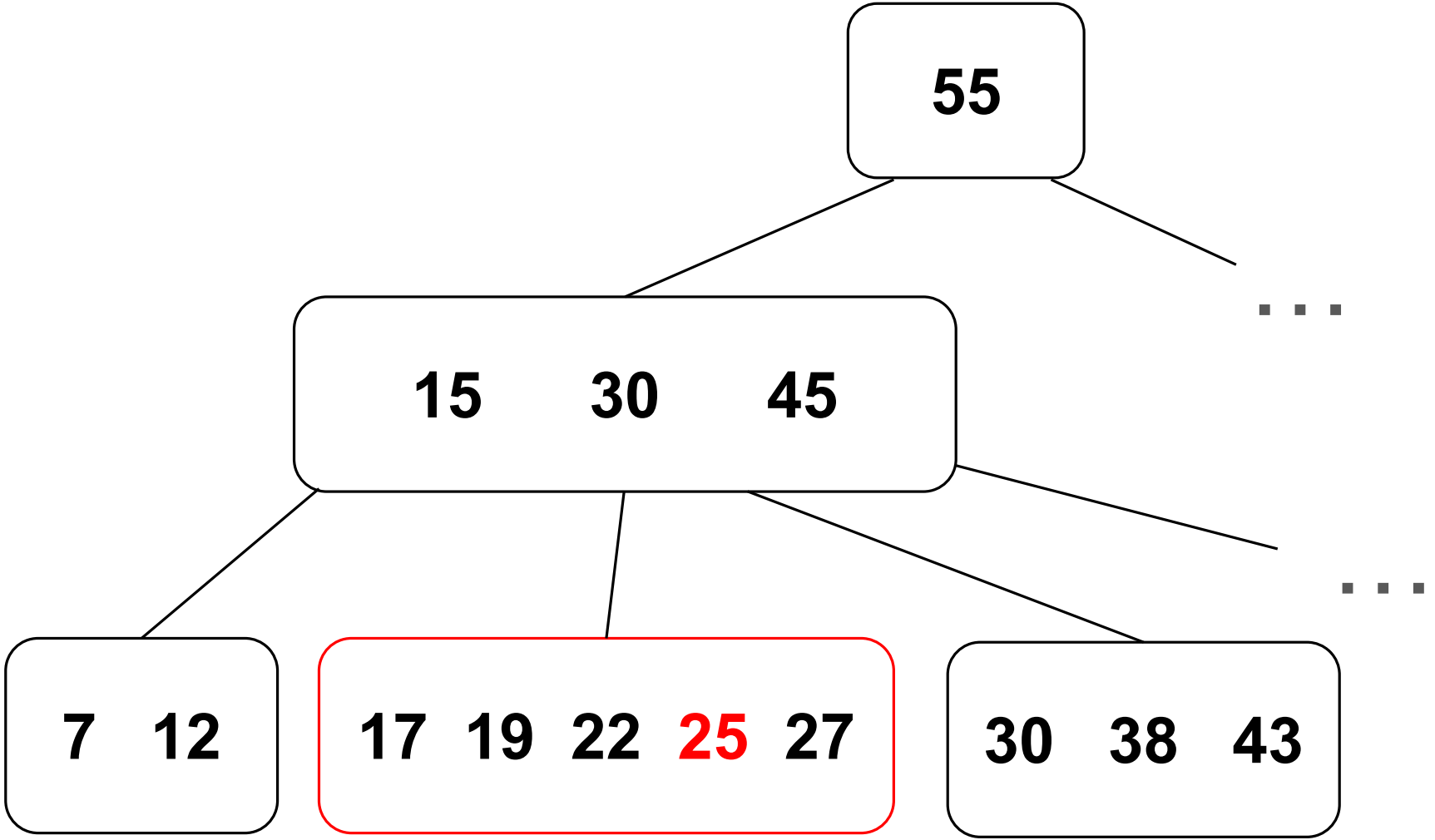
0. Why does it make sense that we have repeated keys?

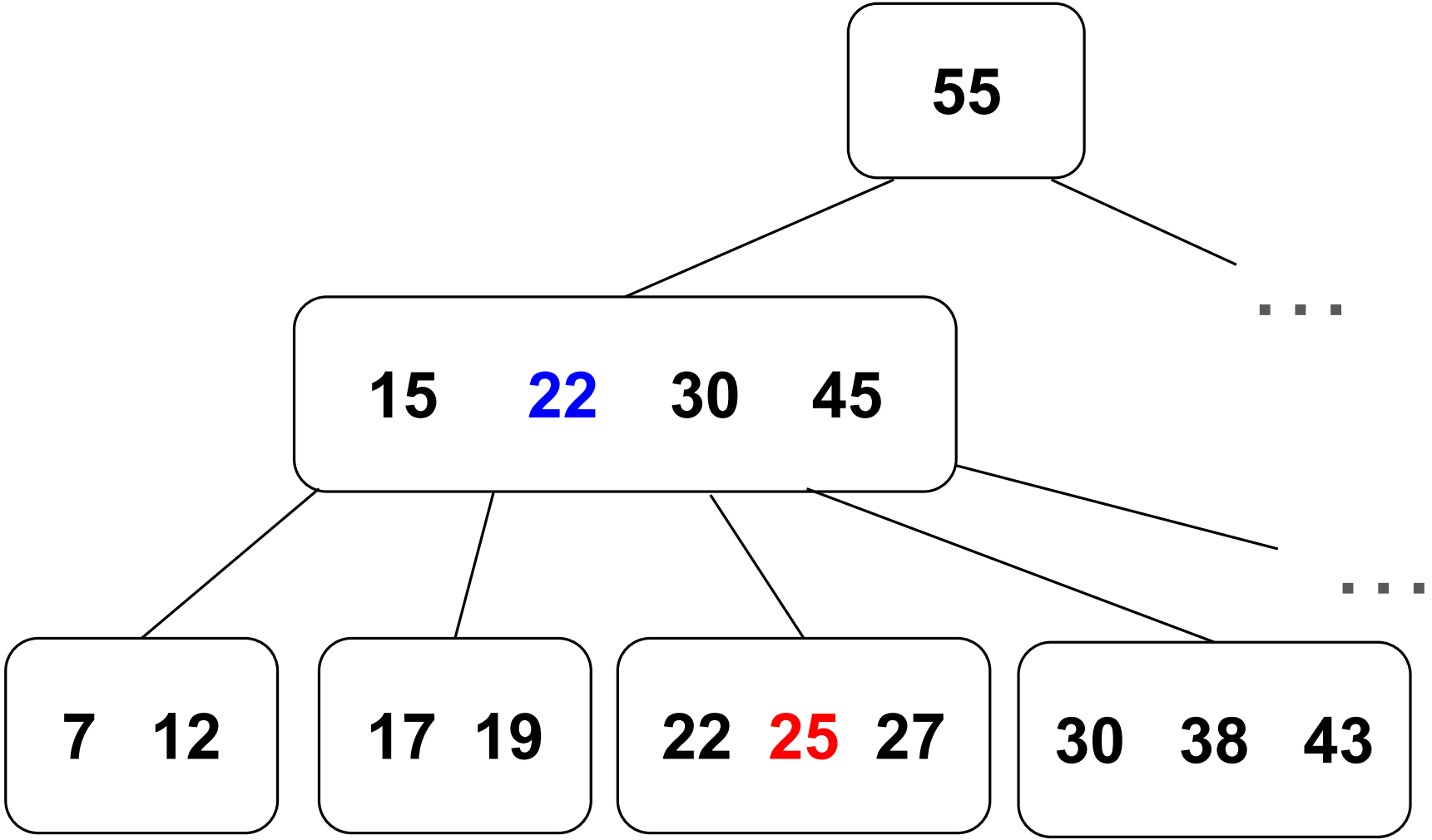
In a B+tree, the full key-value pairs are all stored in leaf nodes. The interior nodes only contain keys.

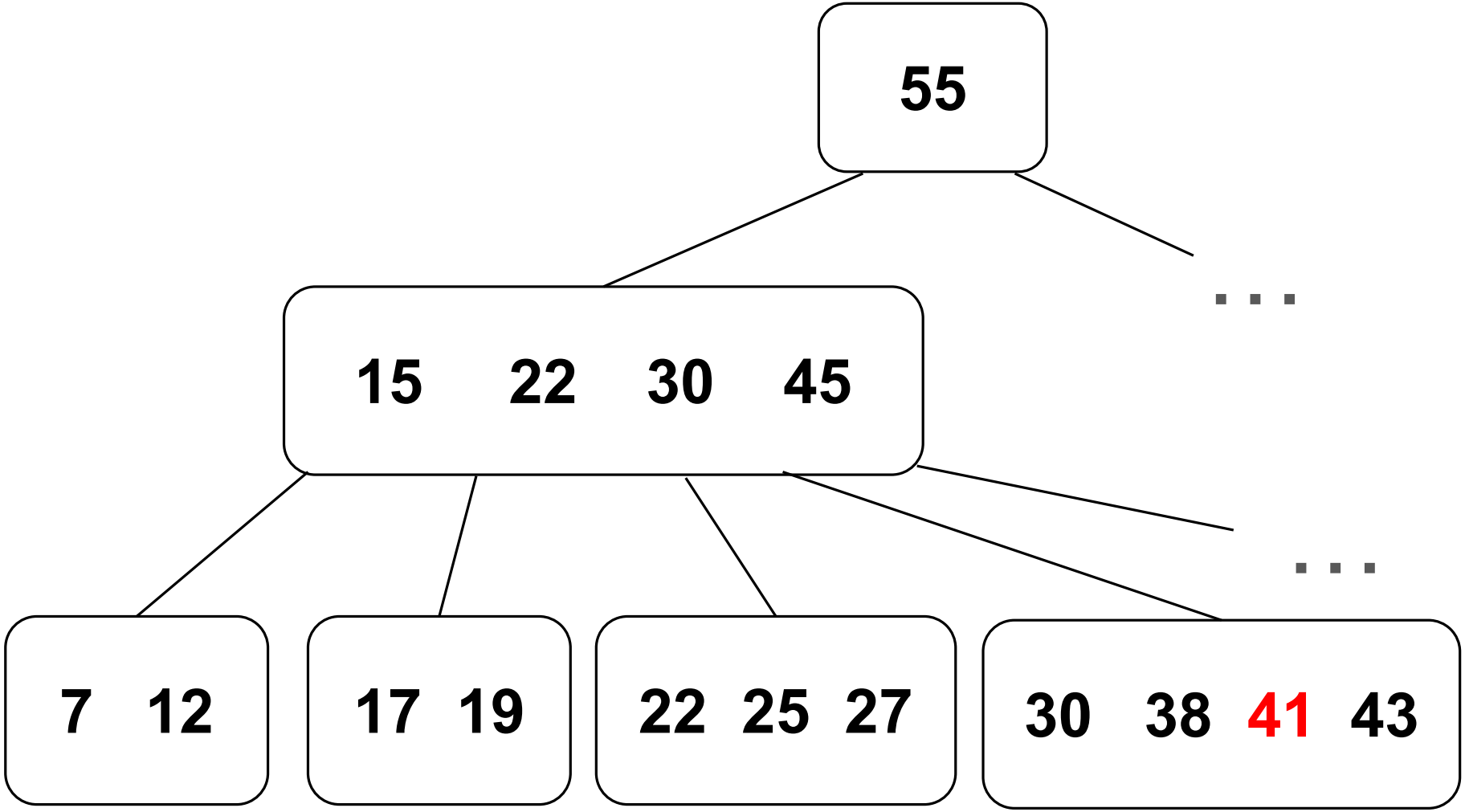
When a node is split, the middle key is sent up and inserted in the parent, but the corresponding key-value pair remains at the leaf level, in the new node that is created as part of the split.

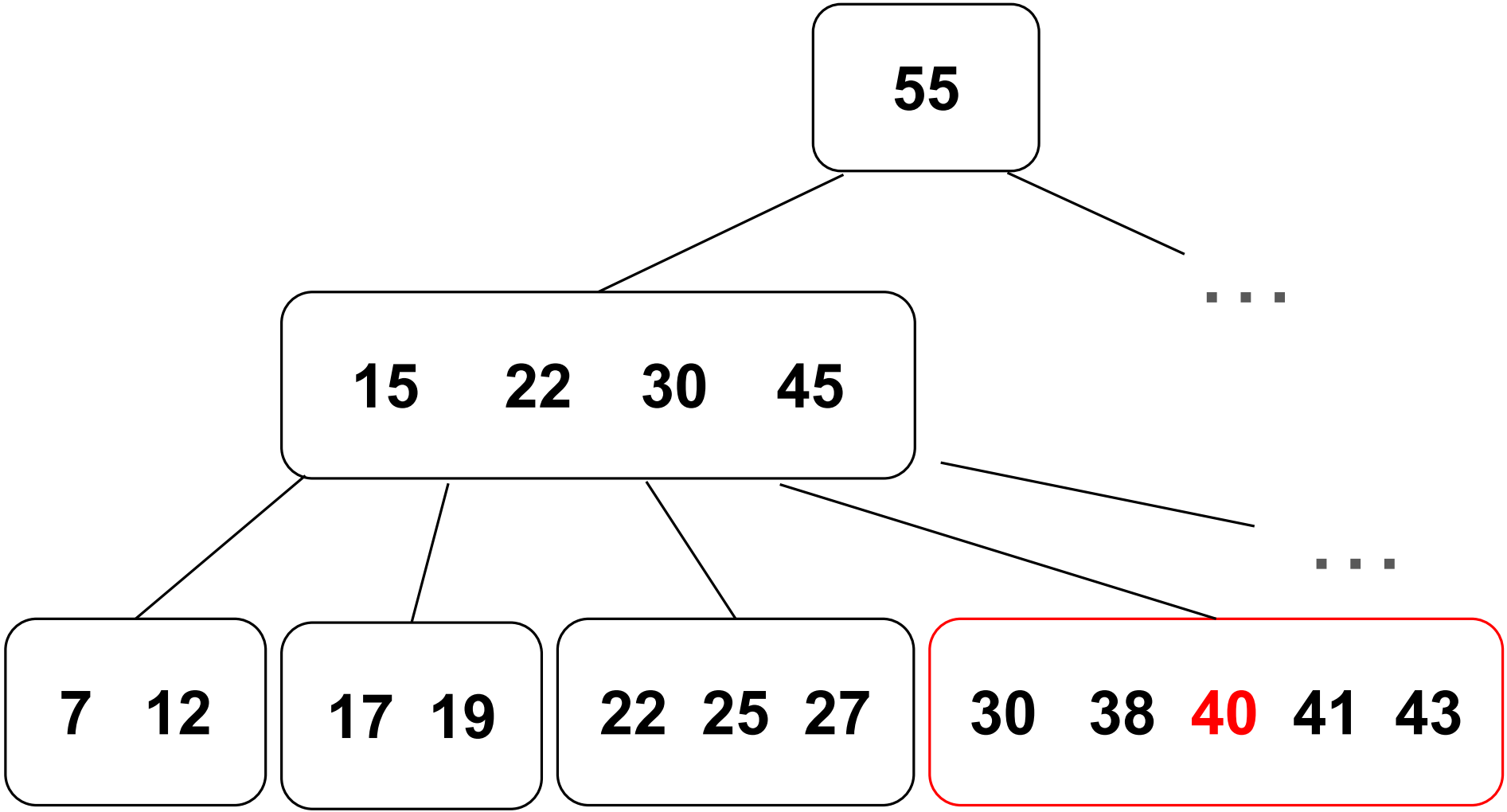

55
15 30 45
7 12
17 22 27
30 38 43

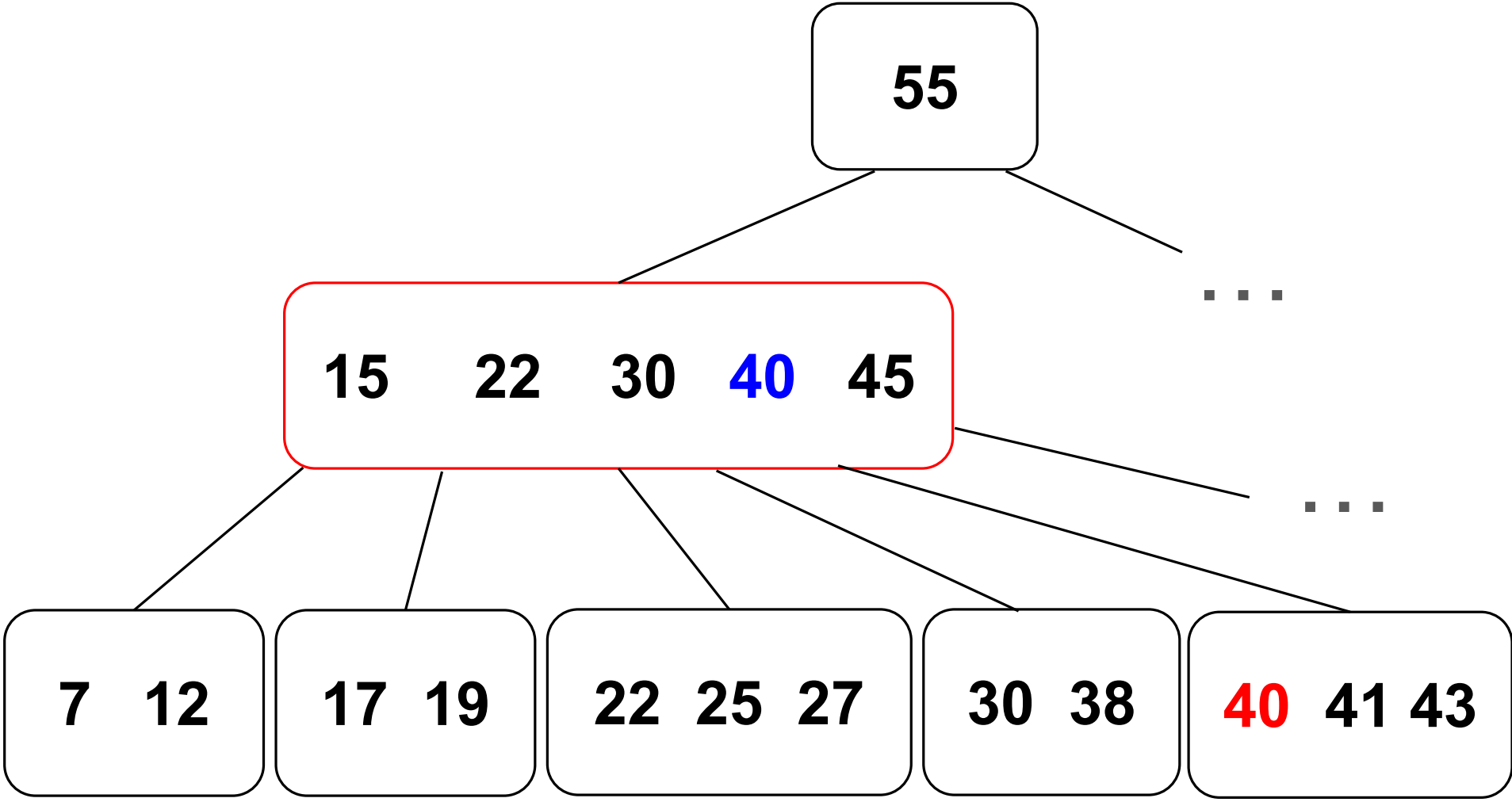

55
15 30 45
7 12
17 19 22 27
30 38 43

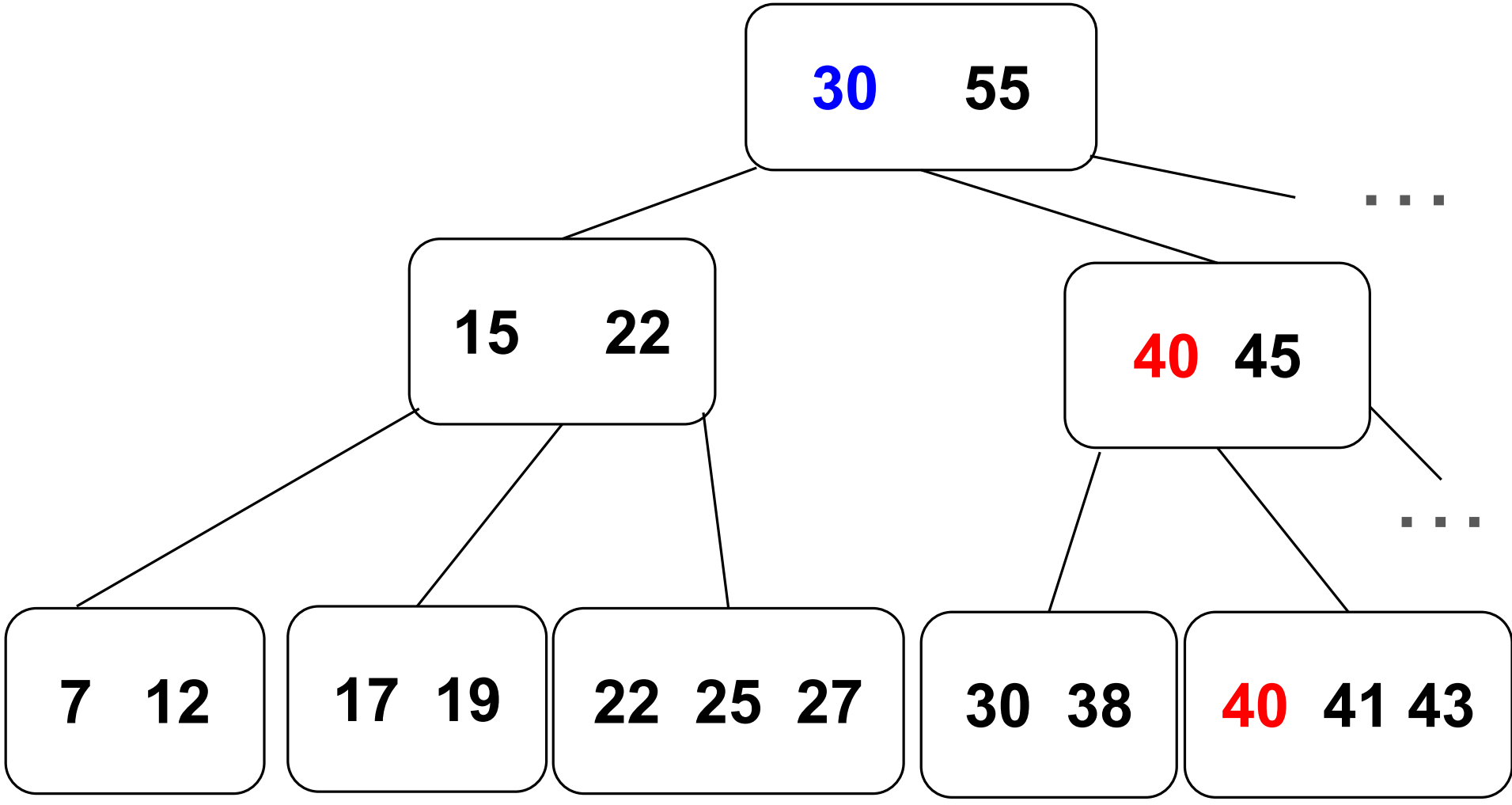

55
15 30 45
7 12
17 19 22 25 27
30 38 43


55
15 22 30 45
7 12
17 19
22 25 27
30 38 43


55
15 22 30 45
7 12
17 19
22 25 27
30 38 41 43


55
. . .
15 22 30 45
. . .
7 12
17 19
22 25 27
30 38 40 41 43


55
15 22 30 40 45
7 12
17 19
22 25 27
30 38
40 41 43


30 55
. . .
15 22
40 45
. . .
7 12
17 19
22 25 27
30 38
40 41 43