Task 1: Construct fixed-length and variable-length records

Imagine that we are working with the Room table from the university database that we used in Lab 2. That table has the following schema:

```
Room(id CHAR(4), name VARCHAR(20), capacity INT)
```

Consider the following tuple from that table:

```
('0123', 'PHO 206', 199)
```

1. If we use the fixed-length record format discussed in lecture, what would the record look like for this tuple?

Task 1: Construct fixed-length and variable-length records

Imagine that we are working with the Room table from the university database that we used in Lab 2. That table has the following schema:

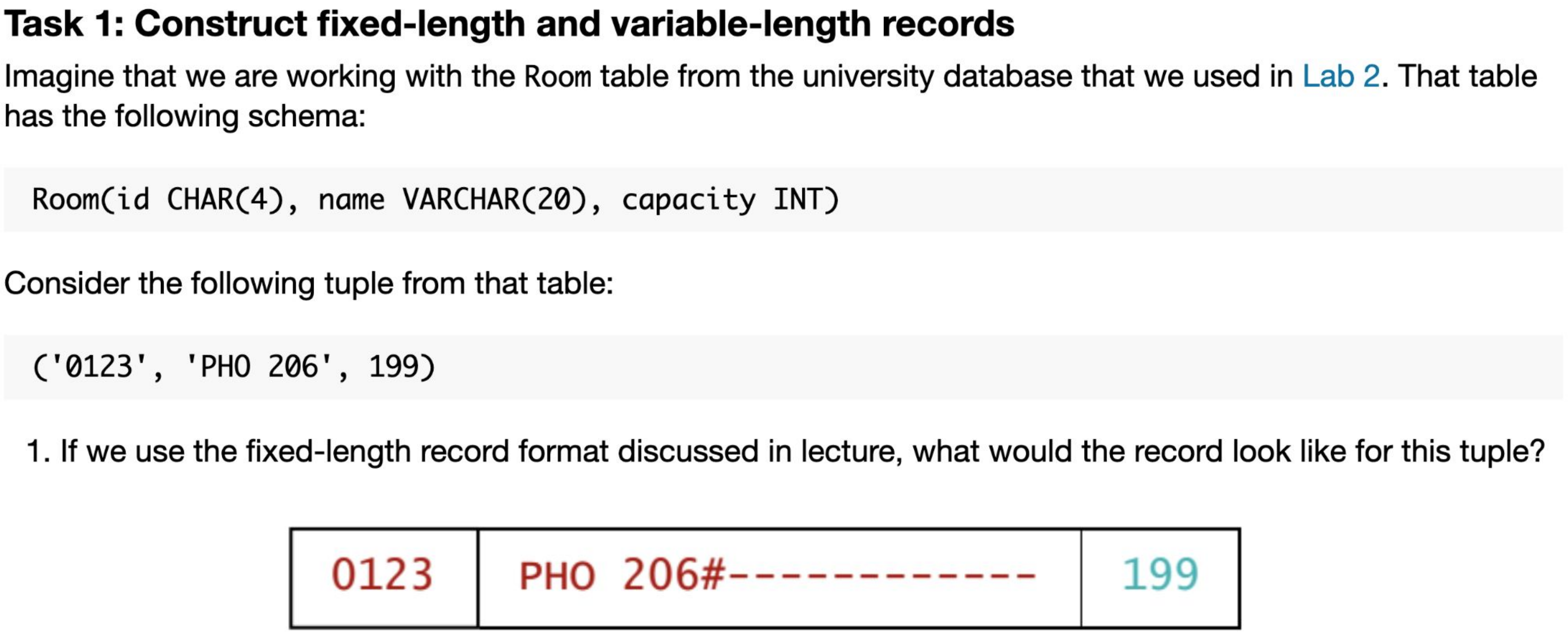
```
Room(id CHAR(4), name VARCHAR(20), capacity INT)
```

Consider the following tuple from that table:

```
('0123', 'PHO 206', 199)
```

1. If we use the fixed-length record format discussed in lecture, what would the record look like for this tuple?

0123	PHO 206#------------	199



Task 1: Construct fixed-length and variable-length records

Imagine that we are working with the Room table from the university database that we used in Lab 2. That table has the following schema:

```
Room(id CHAR(4), name VARCHAR(20), capacity INT)
```

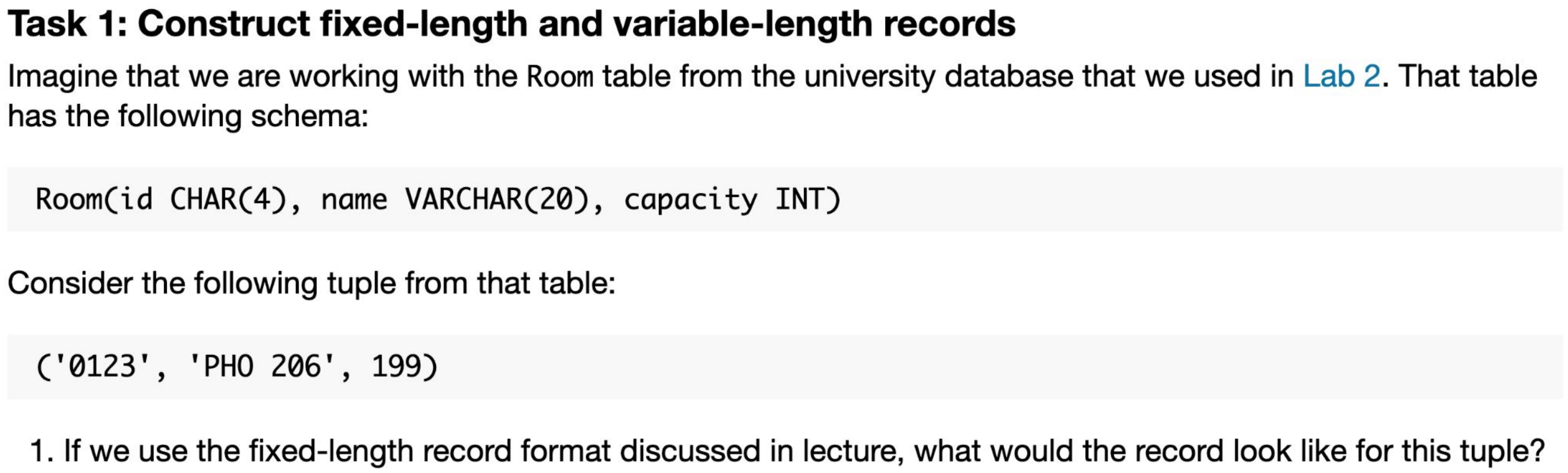
Consider the following tuple from that table:

```
('0123', 'PHO 206', 199)
```

1. If we use the fixed-length record format discussed in lecture, what would the record look like for this tuple?

0123	PHO 206#------------	199

Do we need per-record metadata?



Task 1: Construct fixed-length and variable-length records

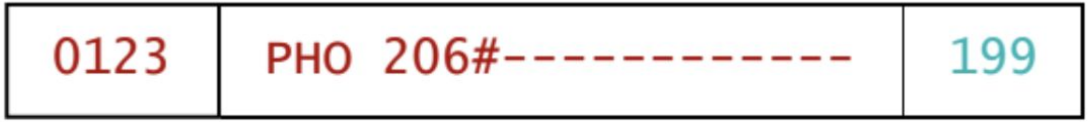
Imagine that we are working with the Room table from the university database that we used in Lab 2. That table has the following schema:

```
Room(id CHAR(4), name VARCHAR(20), capacity INT)
```

Consider the following tuple from that table:

```
('0123', 'PHO 206', 199)
```

1. If we use the fixed-length record format discussed in lecture, what would the record look like for this tuple?



0123	PHO 206#------------	199

Do we need metadata?

No. Each field has the same length for every record.

Task 1: Construct fixed-length and variable-length records

Imagine that we are working with the Room table from the university database that we used in Lab 2. That table has the following schema:

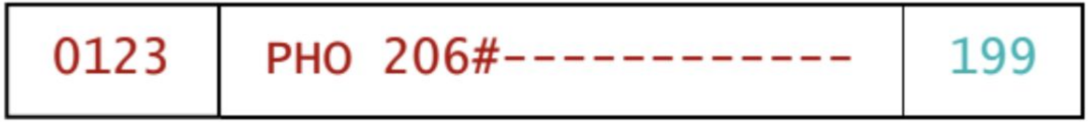
```
Room(id CHAR(4), name VARCHAR(20), capacity INT)
```

Consider the following tuple from that table:

```
('0123', 'PHO 206', 199)
```

2. What is the length in bytes of this record if we assume that:

   - characters are one byte each
   - integer data values are four bytes each.



0123	PHO 206#------------	199

```
Room(id CHAR(4), name VARCHAR(20), capacity INT)
```

0123	PHO 206#------------	199

2. What is the length in bytes of this record if we assume that:

- characters are one byte each
- integer data values are four bytes each.

```
Room(id CHAR(4), name VARCHAR(20), capacity INT)
```

0123	PHO 206#-------------	199

2. What is the length in bytes of this record if we assume that:

- characters are one byte each
- integer data values are four bytes each.

```
     len(id) + max_len(name) + len(capacity)
=       4    +      20       +      4         = 28
```

```
Room(id CHAR(4), name VARCHAR(20), capacity INT)
```

```
('0123', 'PHO 206', 199)
```

3. Now assume that we are 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? In addition to one-byte characters and four-byte integer *data* values, you should assume that we use **two-byte** integers for integer *metadata* like lengths and offsets.

```
Room(id CHAR(4), name VARCHAR(20), capacity INT)
```

```
('0123', 'PHO 206', 199)
```

3. Now assume that we are 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? In addition to one-byte characters and four-byte integer *data* values, you should assume that we use **two-byte** integers for integer *metadata* like lengths and offsets.

```
Room(id CHAR(4), name VARCHAR(20), capacity INT)
```

```
('0123', 'PHO 206', 199)
```

3. Now assume that we are 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? In addition to one-byte characters and four-byte integer *data* values, you should assume that we use **two-byte** integers for integer *metadata* like lengths and offsets.

```
Room(id CHAR(4), name VARCHAR(20), capacity INT)
```

```
('0123', 'PHO 206', 199)
```

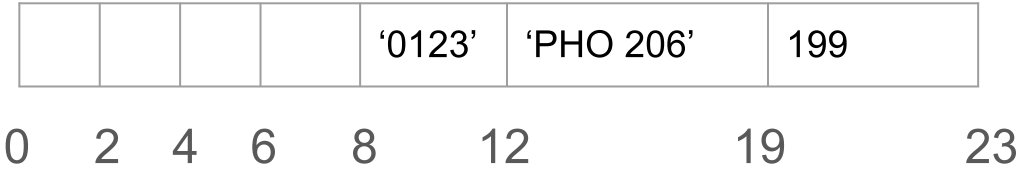
3. Now assume that we are using the second type of variable-length record discussed in lecture, in which each record begins with a header of field offsets. What will the record look like for the above tuple? In addition to one-byte characters and four-byte integer *data* values, you should assume that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

				‘0123’	‘PHO 206’	199

```
Room(id CHAR(4), name VARCHAR(20), capacity INT)
```

```
('0123', 'PHO 206', 199)
```

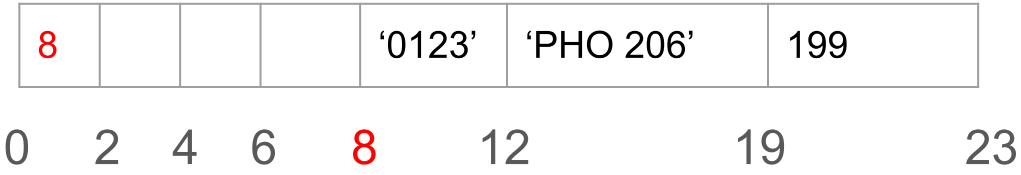
3. Now assume that we are using the second type of variable-length record discussed in lecture, in which each record begins with a header of field offsets. What will the record look like for the above tuple? In addition to one-byte characters and four-byte integer *data* values, you should assume that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



```
Room(id CHAR(4), name VARCHAR(20), capacity INT)
```

```
('0123', 'PHO 206', 199)
```

3. Now assume that we are 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? In addition to one-byte characters and four-byte integer *data* values, you should assume that we use **two-byte** integers for integer *metadata* like lengths and offsets.



```
Room(id CHAR(4), name VARCHAR(20), capacity INT)
```

```
('0123', 'PHO 206', 199)
```

3. Now assume that we are 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? In addition to one-byte characters and four-byte integer *data* values, you should assume that we use **two-byte** integers for integer *metadata* like lengths and offsets.

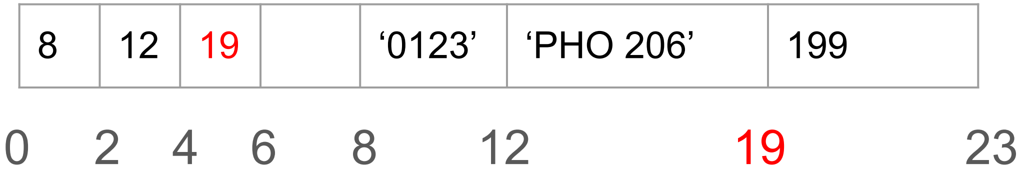
8	12			‘0123’	‘PHO 206’	199

0 2 4 6 8 12 19 23

```
Room(id CHAR(4), name VARCHAR(20), capacity INT)
```

```
('0123', 'PHO 206', 199)
```

3. Now assume that we are 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? In addition to one-byte characters and four-byte integer *data* values, you should assume that we use **two-byte** integers for integer *metadata* like lengths and offsets.



8	12	19		‘0123’	‘PHO 206’	199

0 2 4 6 8 12 19 23

```
Room(id CHAR(4), name VARCHAR(20), capacity INT)
```

```
('0123', 'PHO 206', 199)
```

3. Now assume that we are 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? In addition to one-byte characters and four-byte integer *data* values, you should assume that we use **two-byte** integers for integer *metadata* like lengths and offsets.

8	12	19	23	‘0123’	‘PHO 206’	199

0 2 4 6 8 12 19 23

Length of bytes? 23

```
Room(id CHAR(4), name VARCHAR(20), capacity INT)
```

```
('0123', NULL, 199)
```

5. Now imagine that the room didn’t have a name and we used a value of `NULL` to indicate this:

```
Room(id CHAR(4), name VARCHAR(20), capacity INT)
```

```
('0123', NULL, 199)
```

5. Now imagine that the room didn’t have a name and we used a value of NULL to indicate this:

a.

8	-1	12	16	‘0123’	199

b.

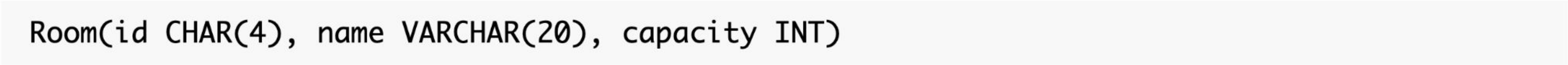
8	12	-1	16	‘0123’	199

c.

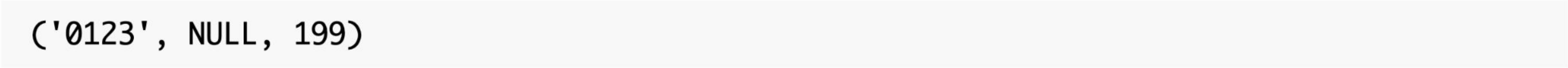
8	-1	12	16	‘0123’	NULL	199

d.

8	12	-1	16	‘0123’	NULL	199



```
Room(id CHAR(4), name VARCHAR(20), capacity INT)
```



```
('0123', NULL, 199)
```

5. Now imagine that the room didn’t have a name and we used a value of NULL to indicate this:

[✓ a.]

8	-1	12	16	‘0123’	199

b.

8	12	-1	16	‘0123’	199

c.

8	-1	12	16	‘0123’	NULL	199

d.

8	12	-1	16	‘0123’	NULL	199

```
Room(id CHAR(4), name VARCHAR(20), capacity INT)
```

```
('0123', NULL, 199)
```

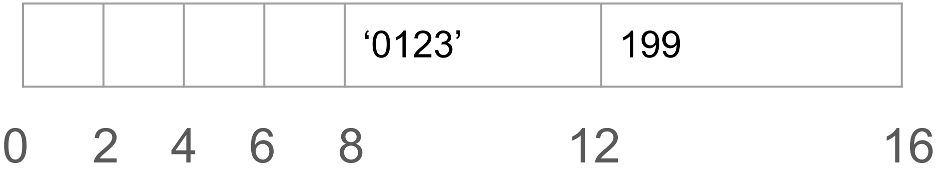
5. Now imagine that the room didn’t have a name and we used a value of NULL to indicate this:

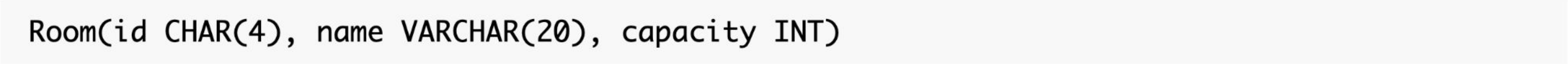
				‘0123’	199

```
Room(id CHAR(4), name VARCHAR(20), capacity INT)
```

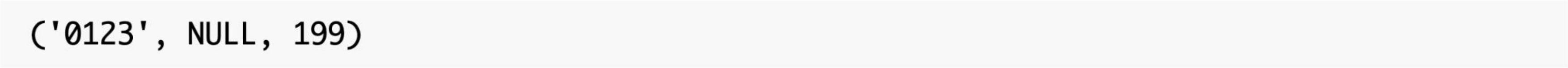
```
('0123', NULL, 199)
```

5. Now imagine that the room didn't have a name and we used a value of NULL to indicate this:



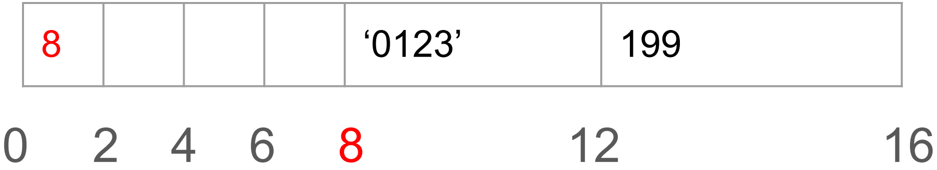


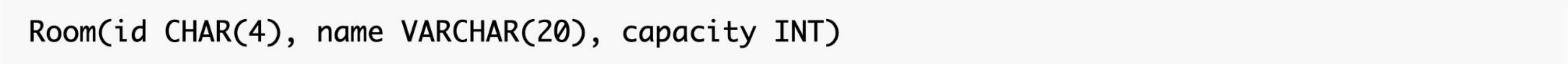
```
Room(id CHAR(4), name VARCHAR(20), capacity INT)
```



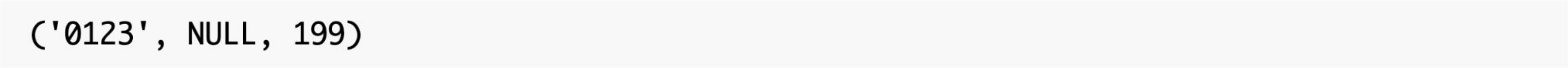
```
('0123', NULL, 199)
```

5. Now imagine that the room didn't have a name and we used a value of NULL to indicate this:



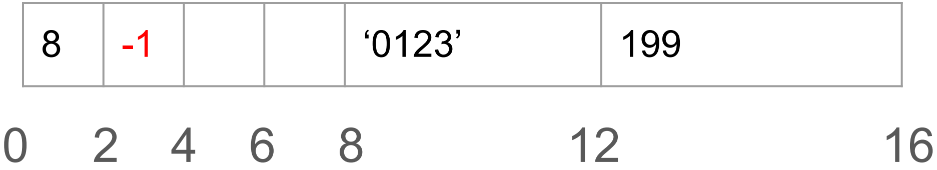


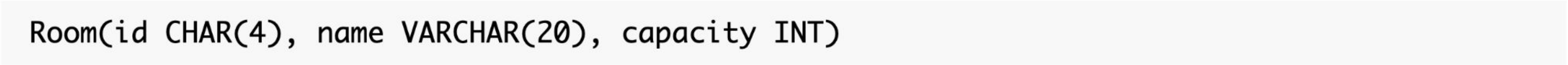
```
Room(id CHAR(4), name VARCHAR(20), capacity INT)
```



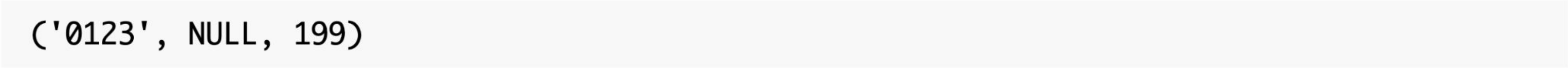
```
('0123', NULL, 199)
```

5. Now imagine that the room didn’t have a name and we used a value of NULL to indicate this:



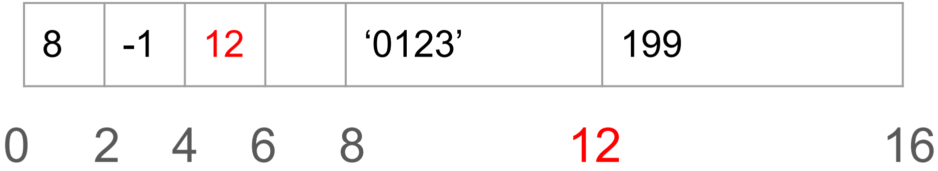


```
Room(id CHAR(4), name VARCHAR(20), capacity INT)
```



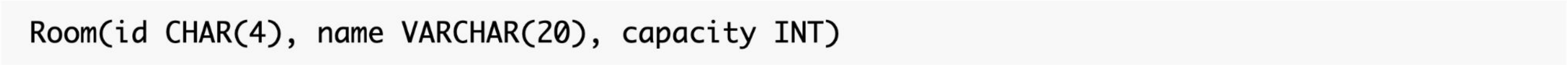
```
('0123', NULL, 199)
```

5. Now imagine that the room didn't have a name and we used a value of NULL to indicate this:

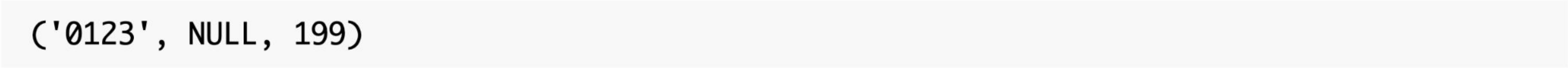


8	-1	12		‘0123’	199

0 2 4 6 8 12 16

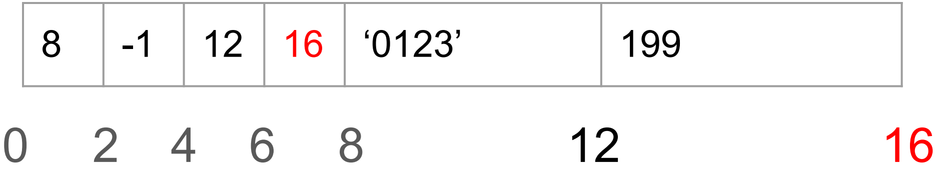


```
Room(id CHAR(4), name VARCHAR(20), capacity INT)
```

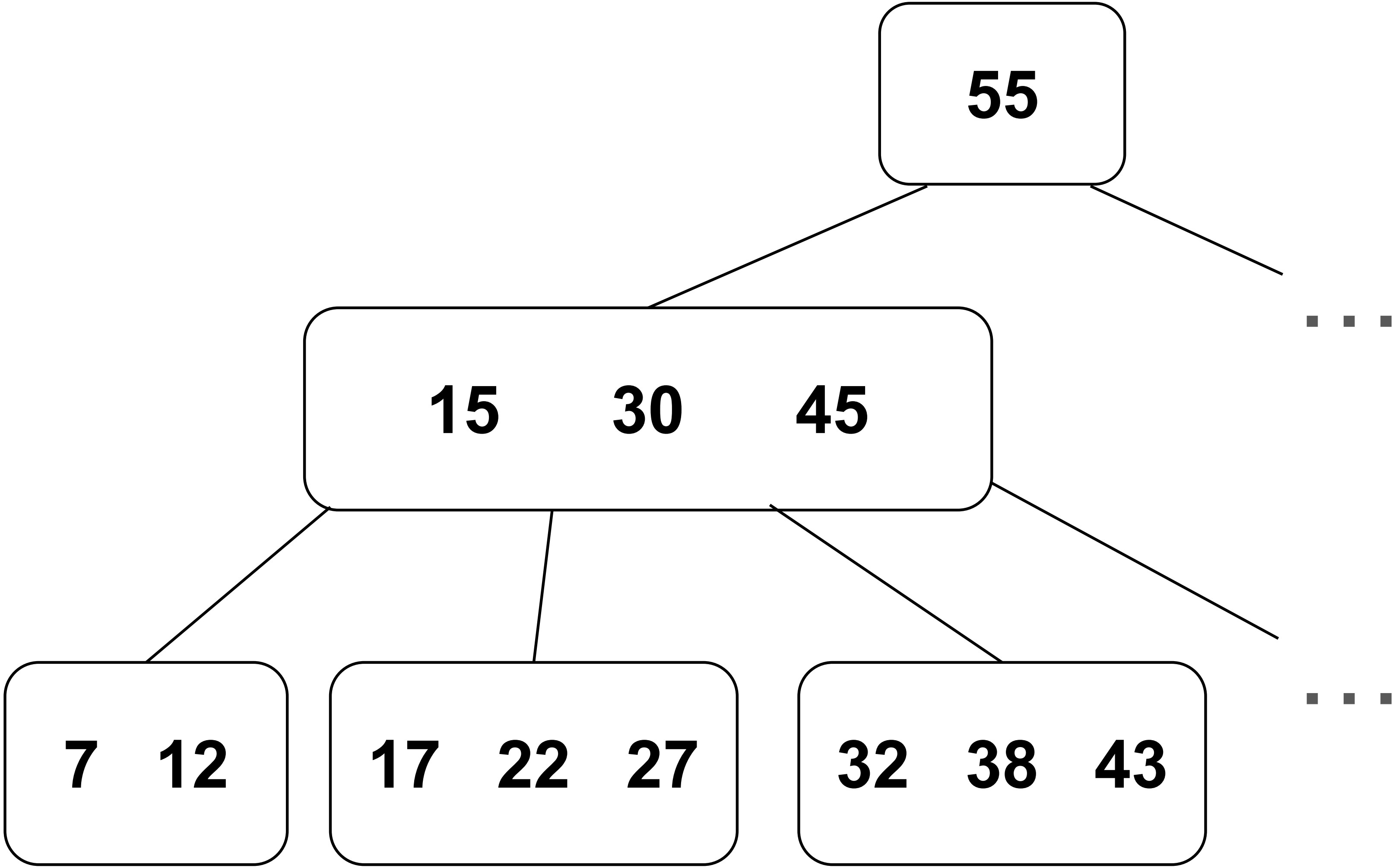


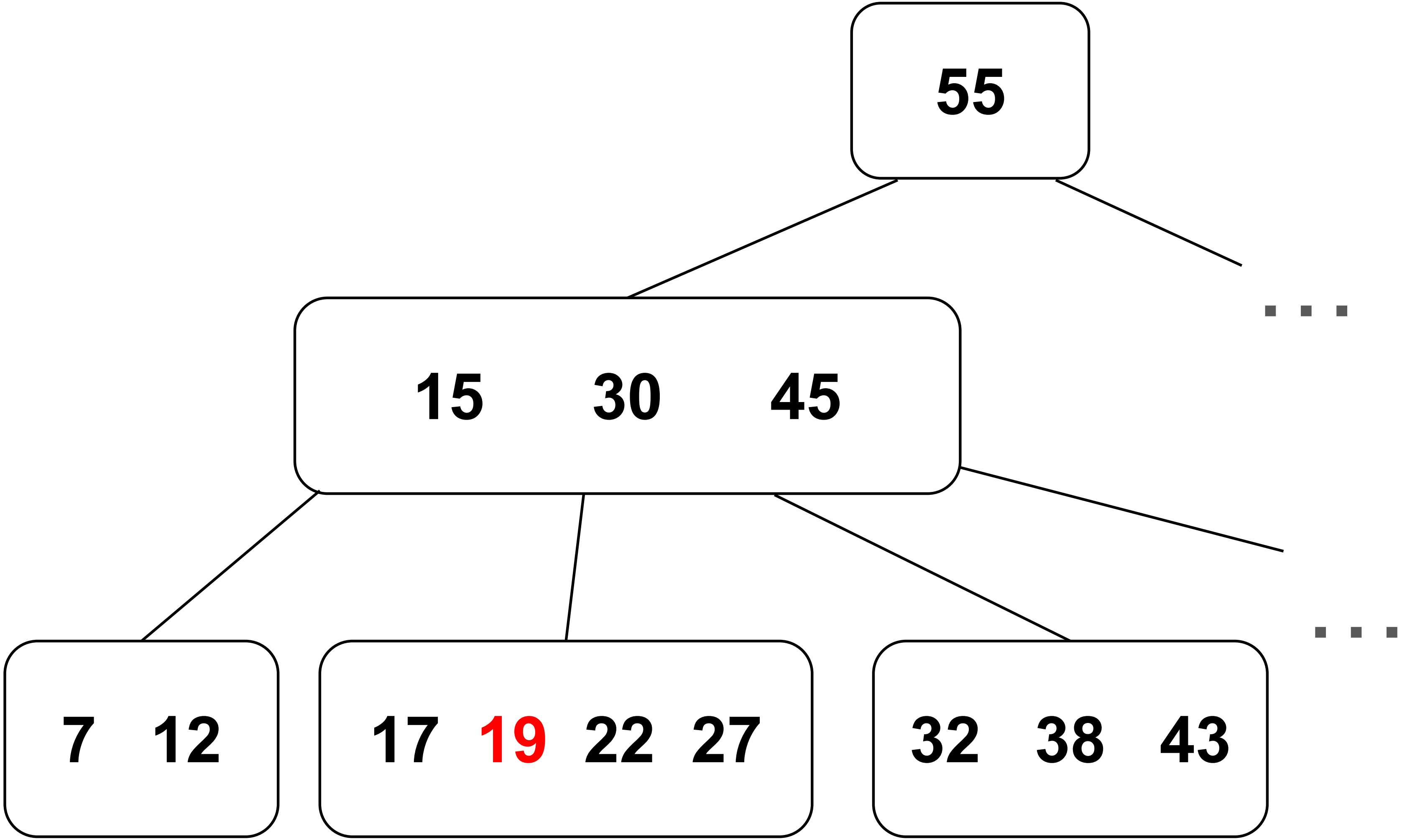
```
('0123', NULL, 199)
```

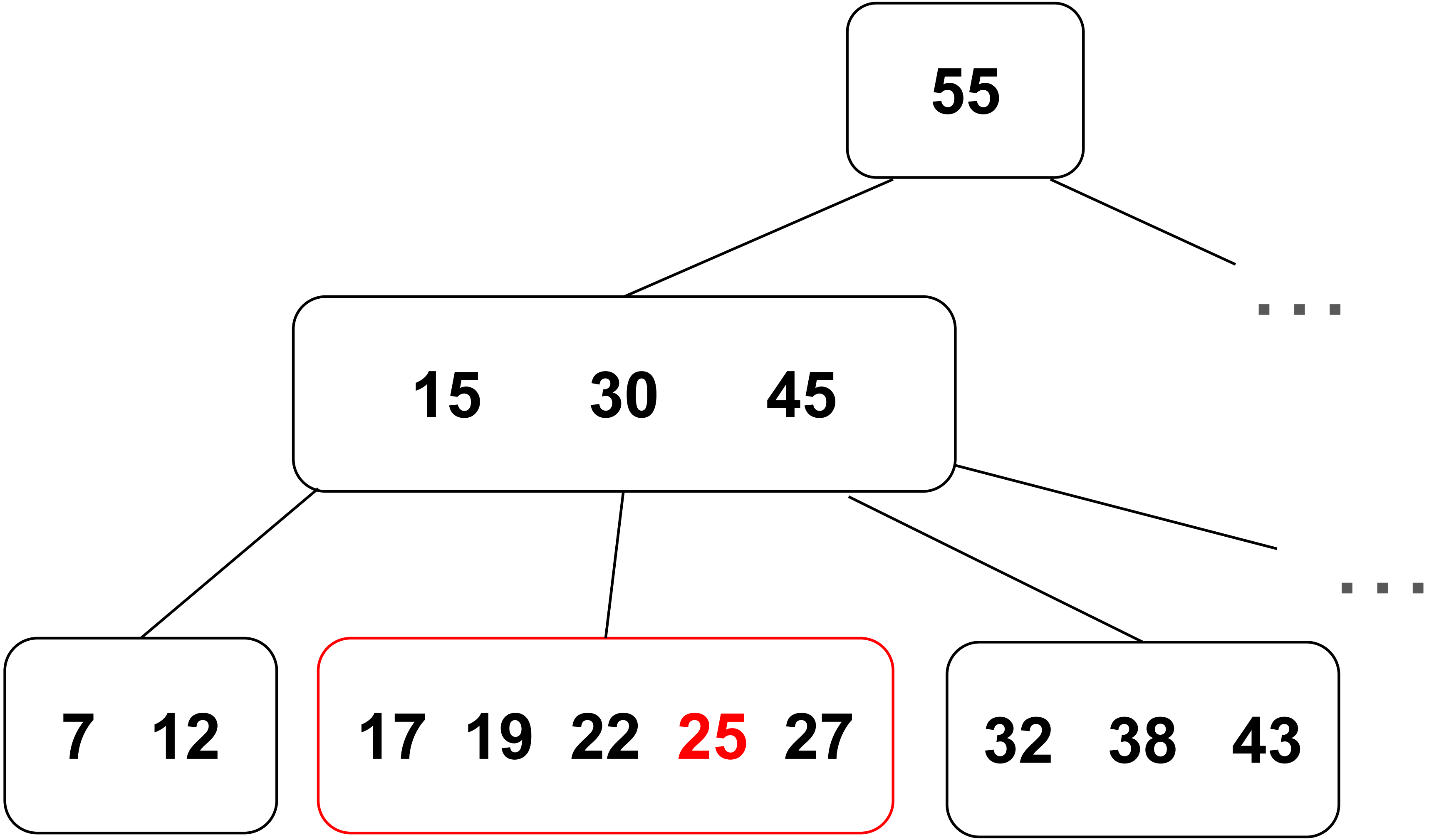
5. Now imagine that the room didn't have a name and we used a value of NULL to indicate this:

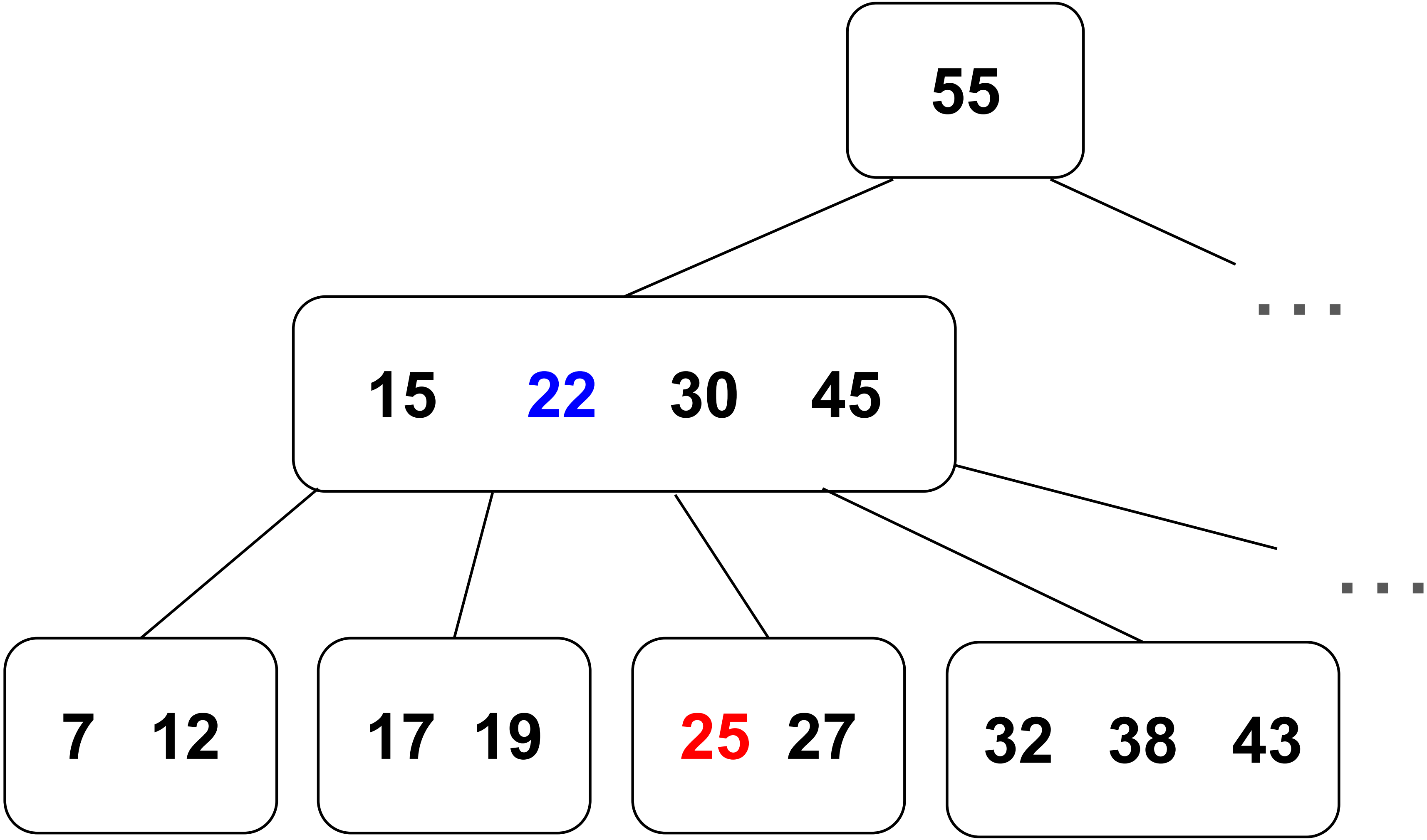


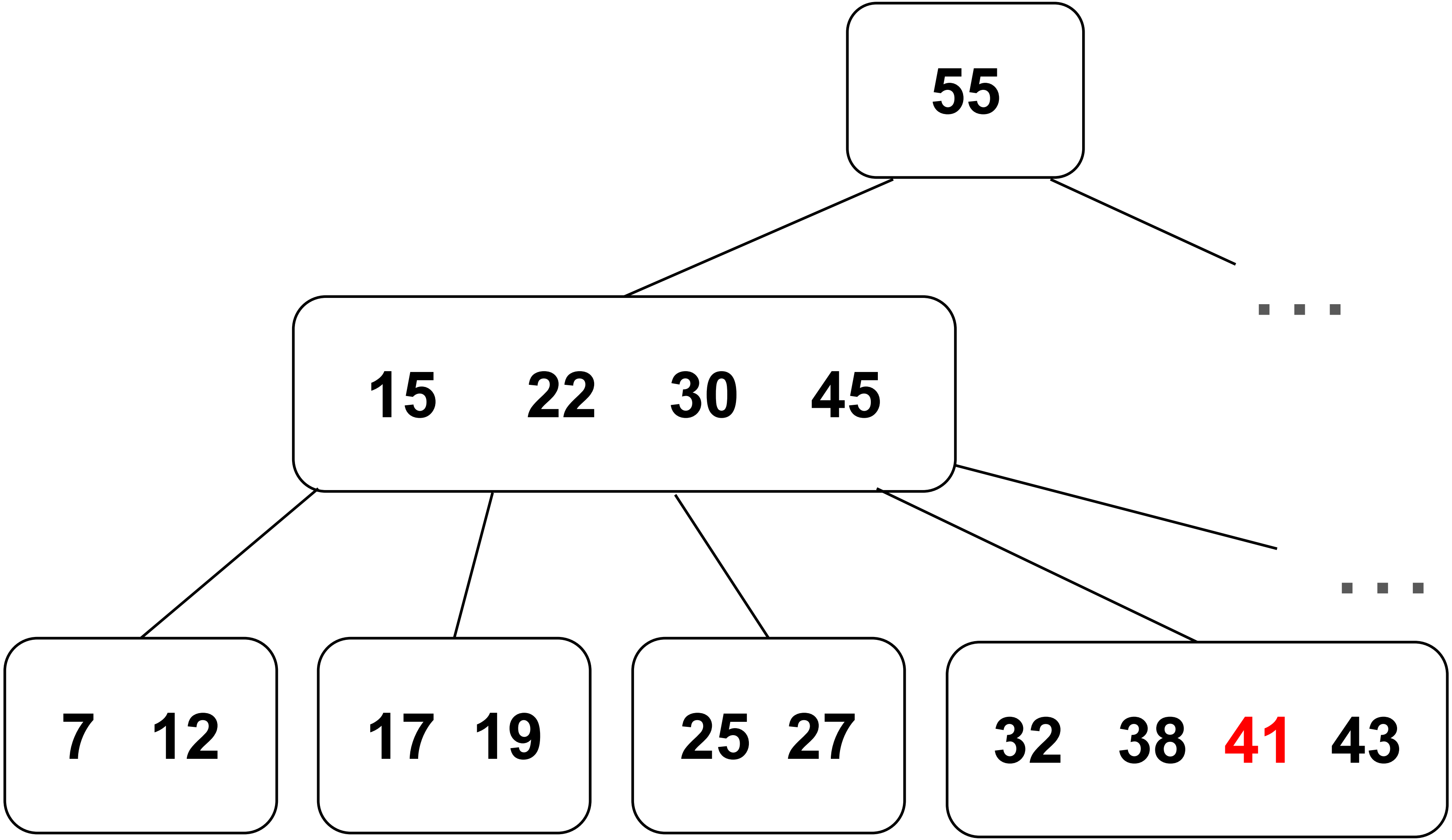
8	-1	12	16	‘0123’	199

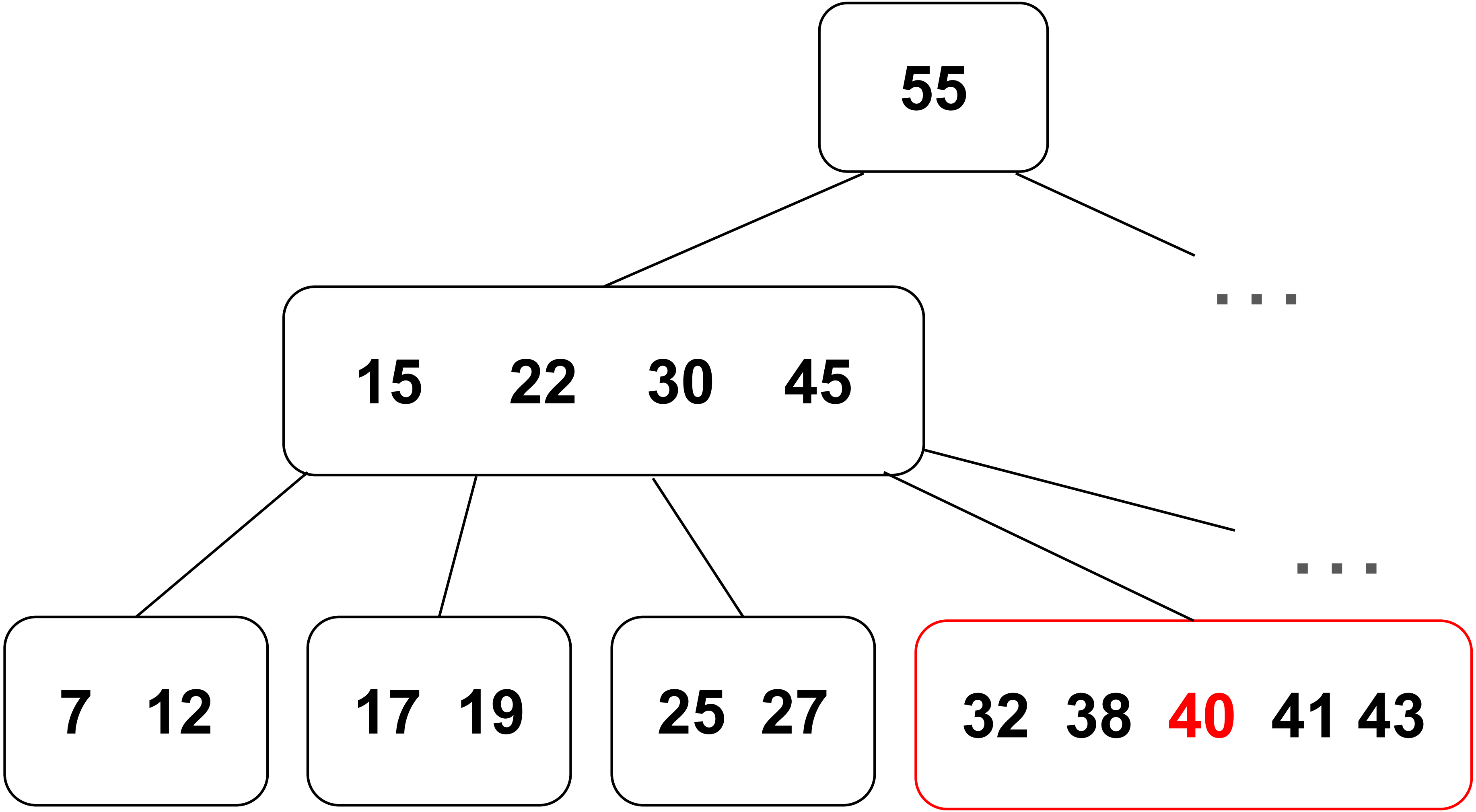

55
15 30 45
...
7 12
17 22 27
32 38 43
...

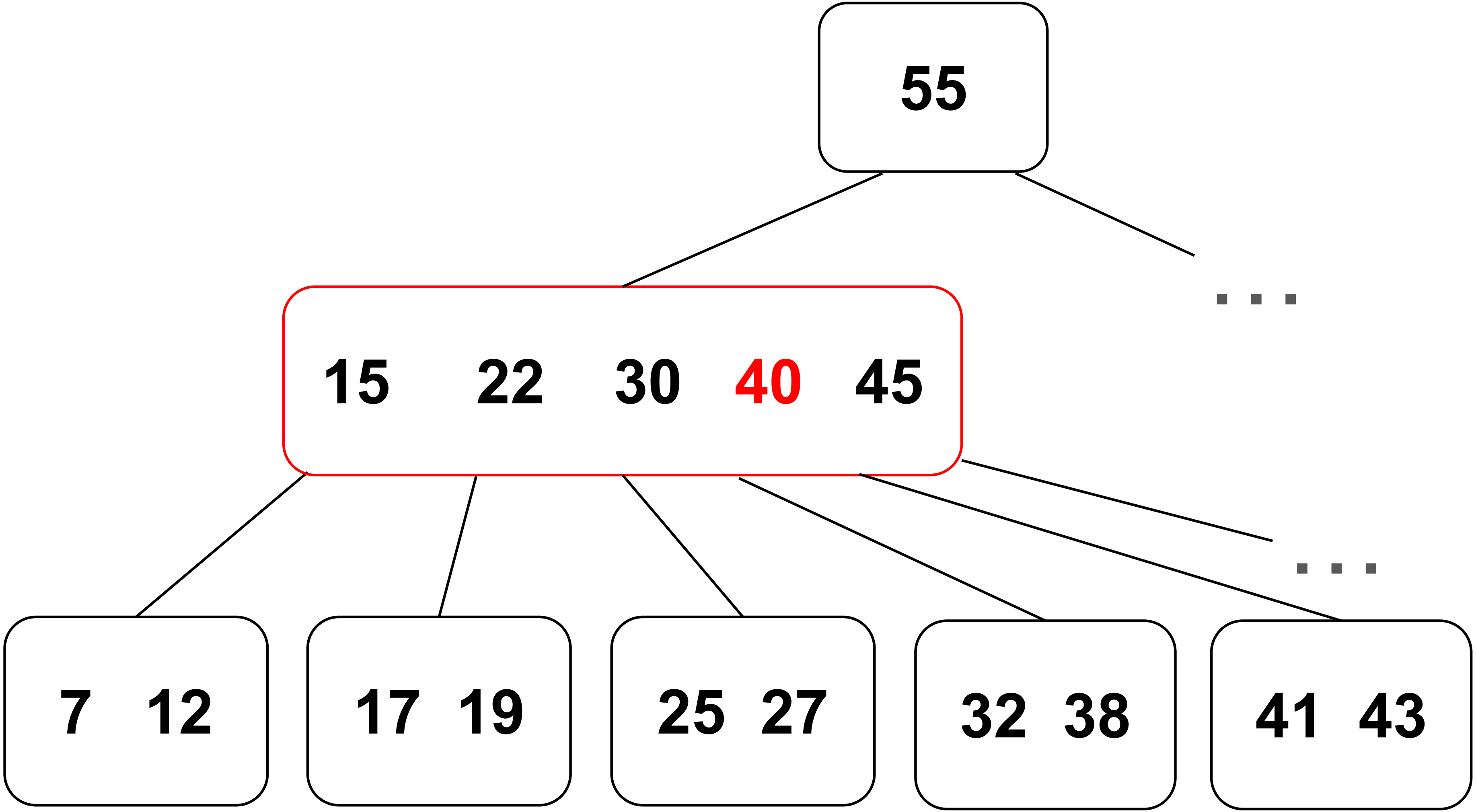

55
...
15 30 45
...
7 12
17 19 22 27
32 38 43

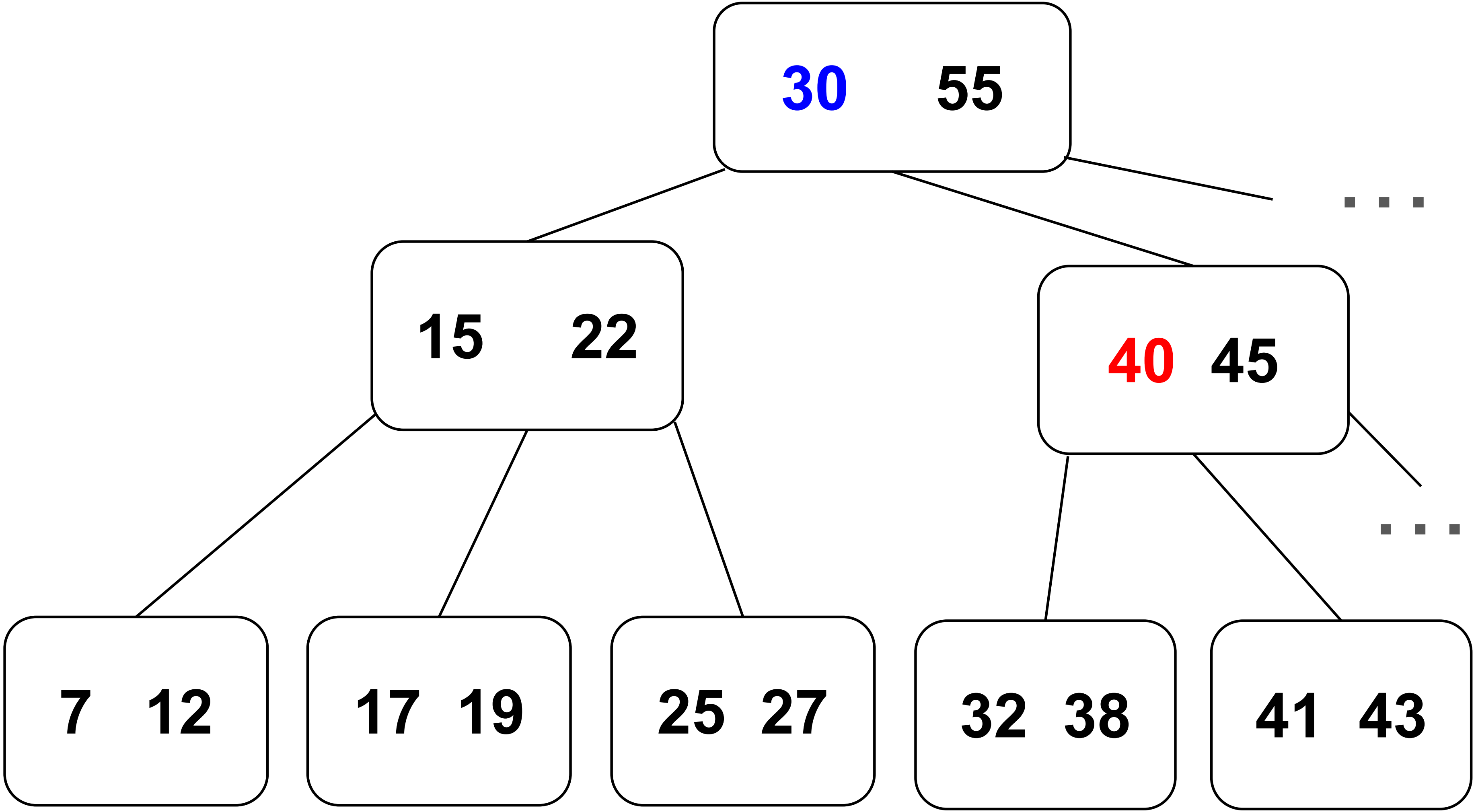

55
15 30 45
7 12
17 19 22 25 27
32 38 43
...
...


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

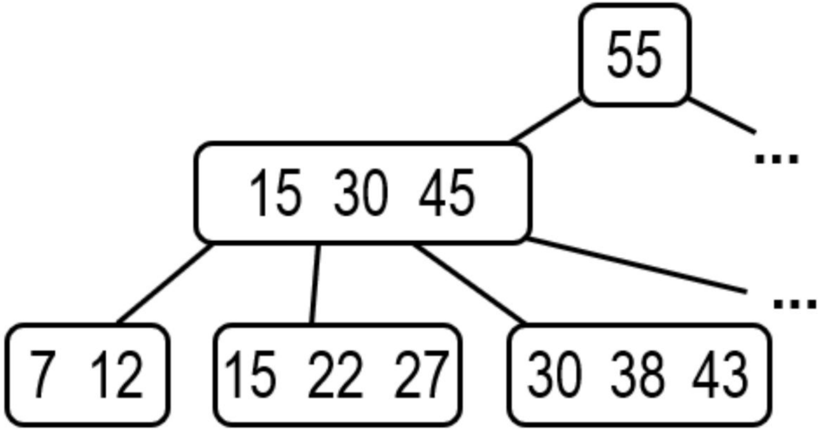

55
15 22 30 45
...
...
7 12
17 19
25 27
32 38 40 41 43


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:

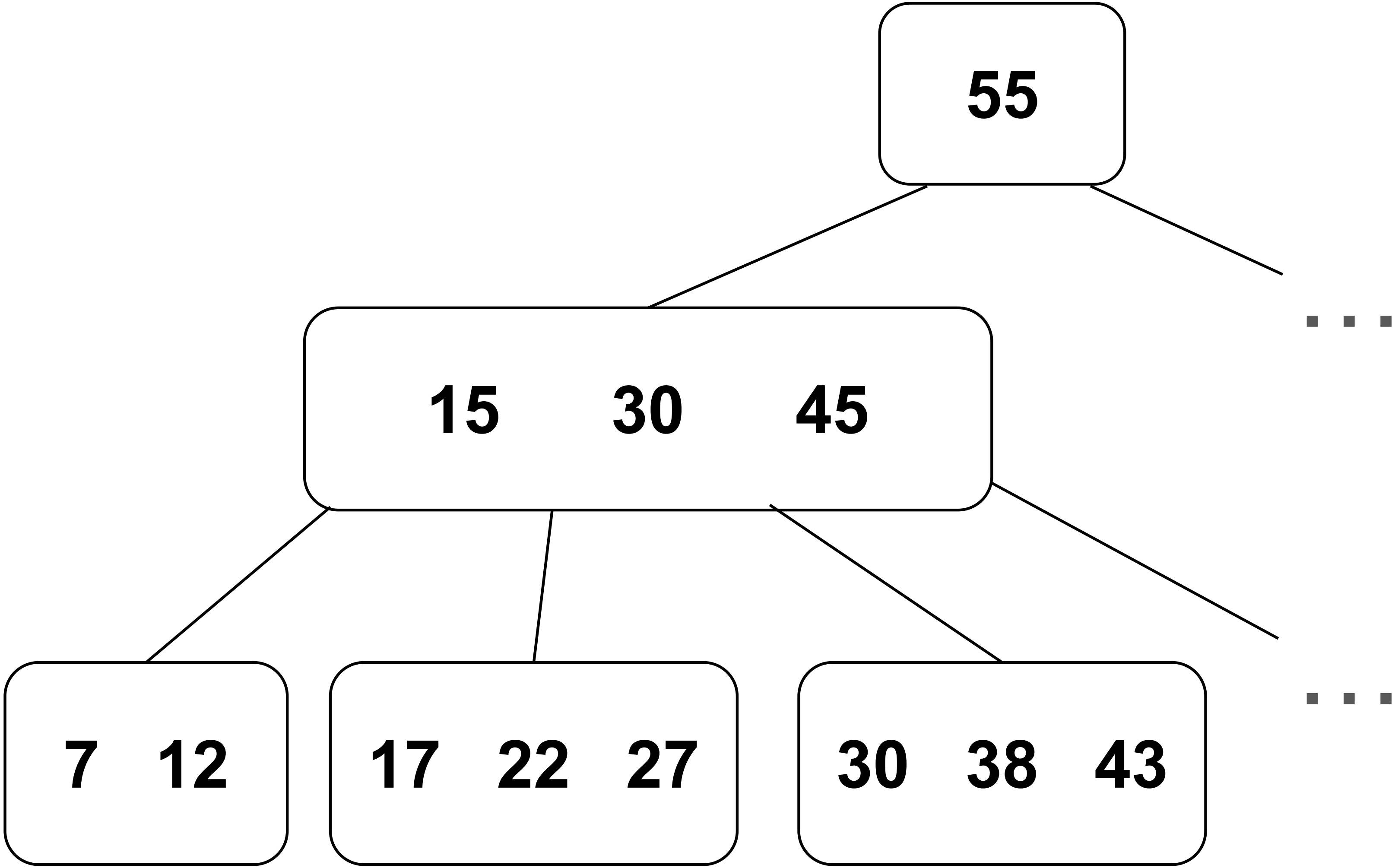


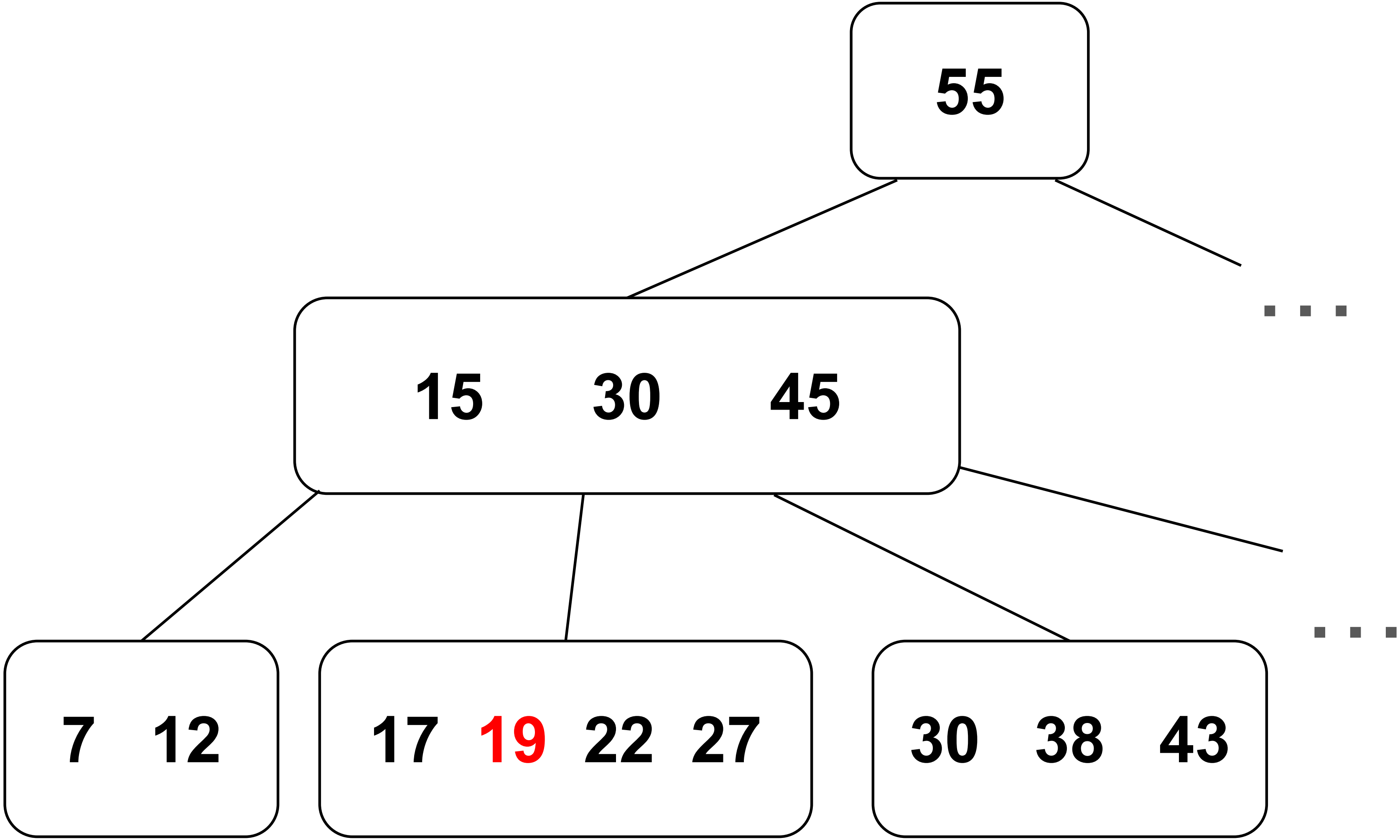
Note that this tree is similar to the initial tree in Task 2, but some keys are missing, and other keys appear in two places.

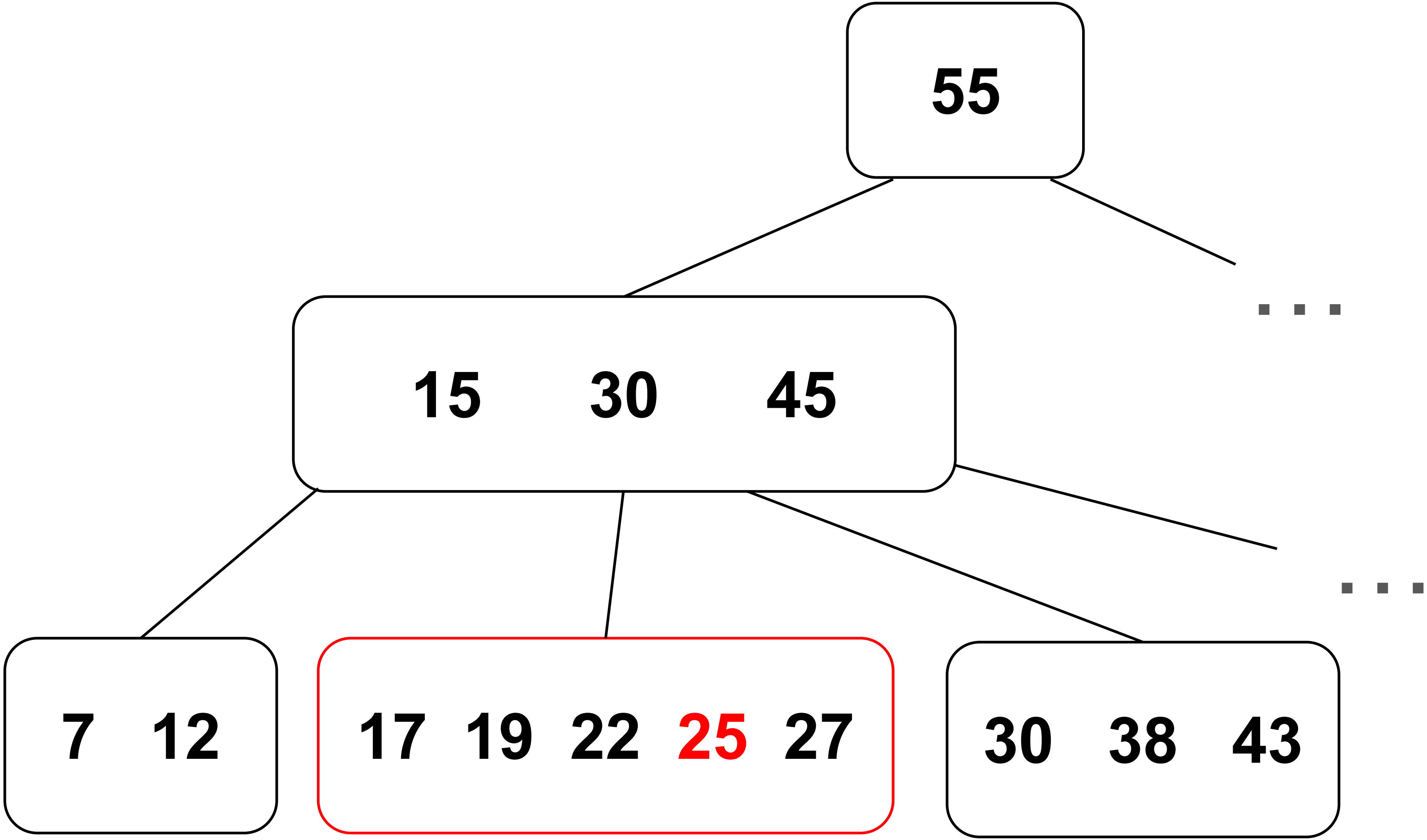
0. Why does it make sense that we have repeated keys?

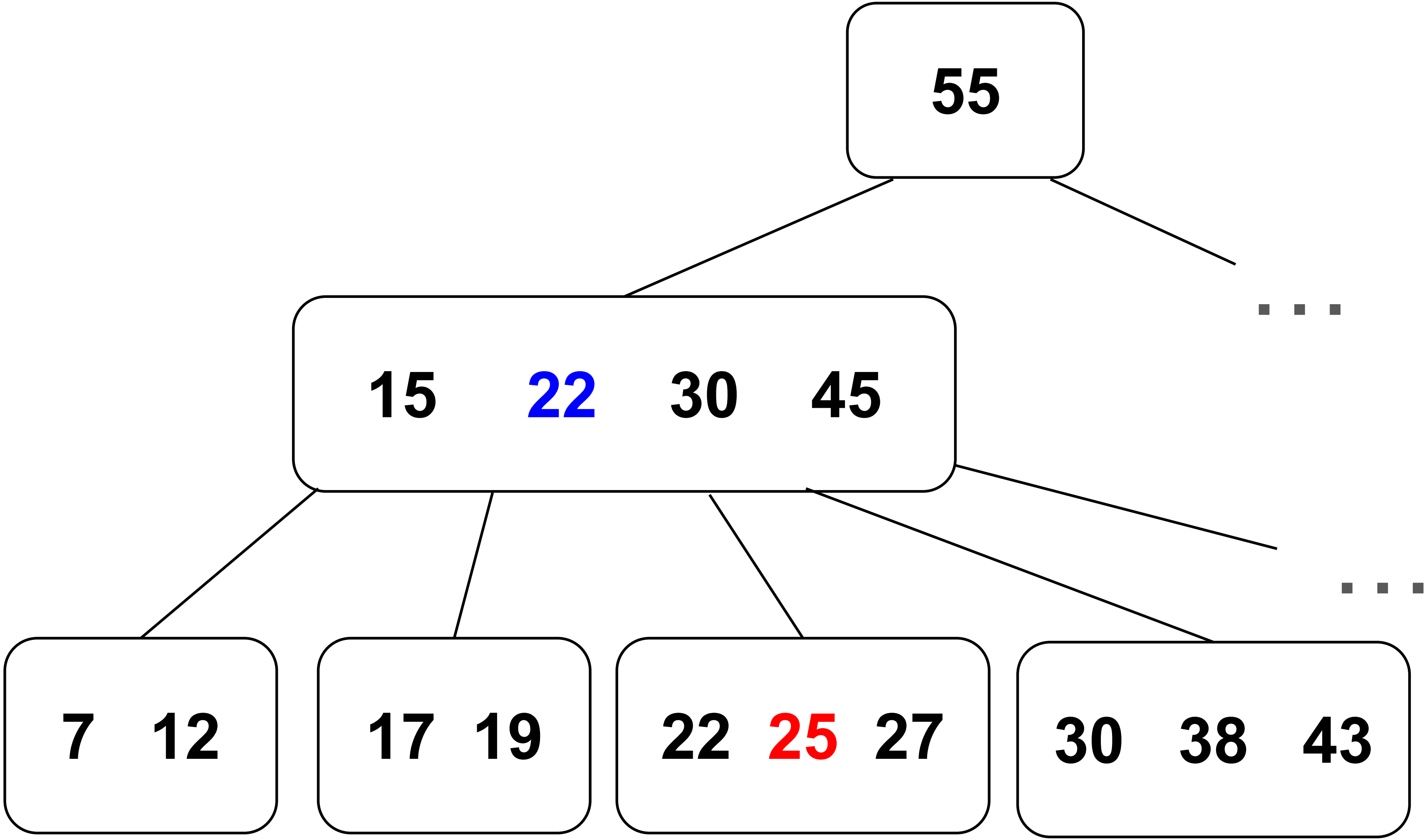
In a B+tree, the full key-value pairs are all stored in leaf nodes. The interior nodes only contain keys.

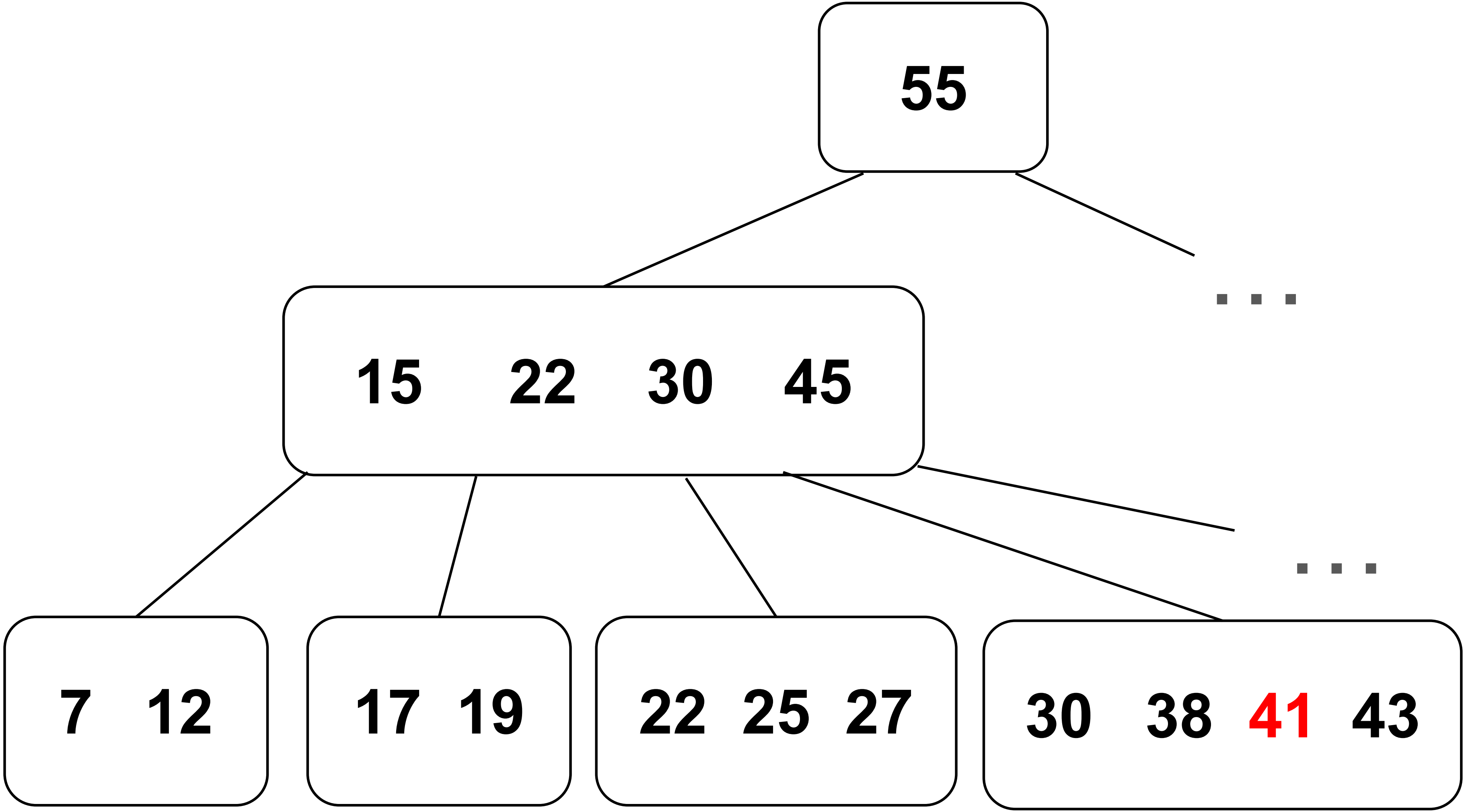
When a node is split, the middle key is sent up and inserted in the parent, but the corresponding key-value pair remains at the leaf level, in the new node that is created as part of the split.

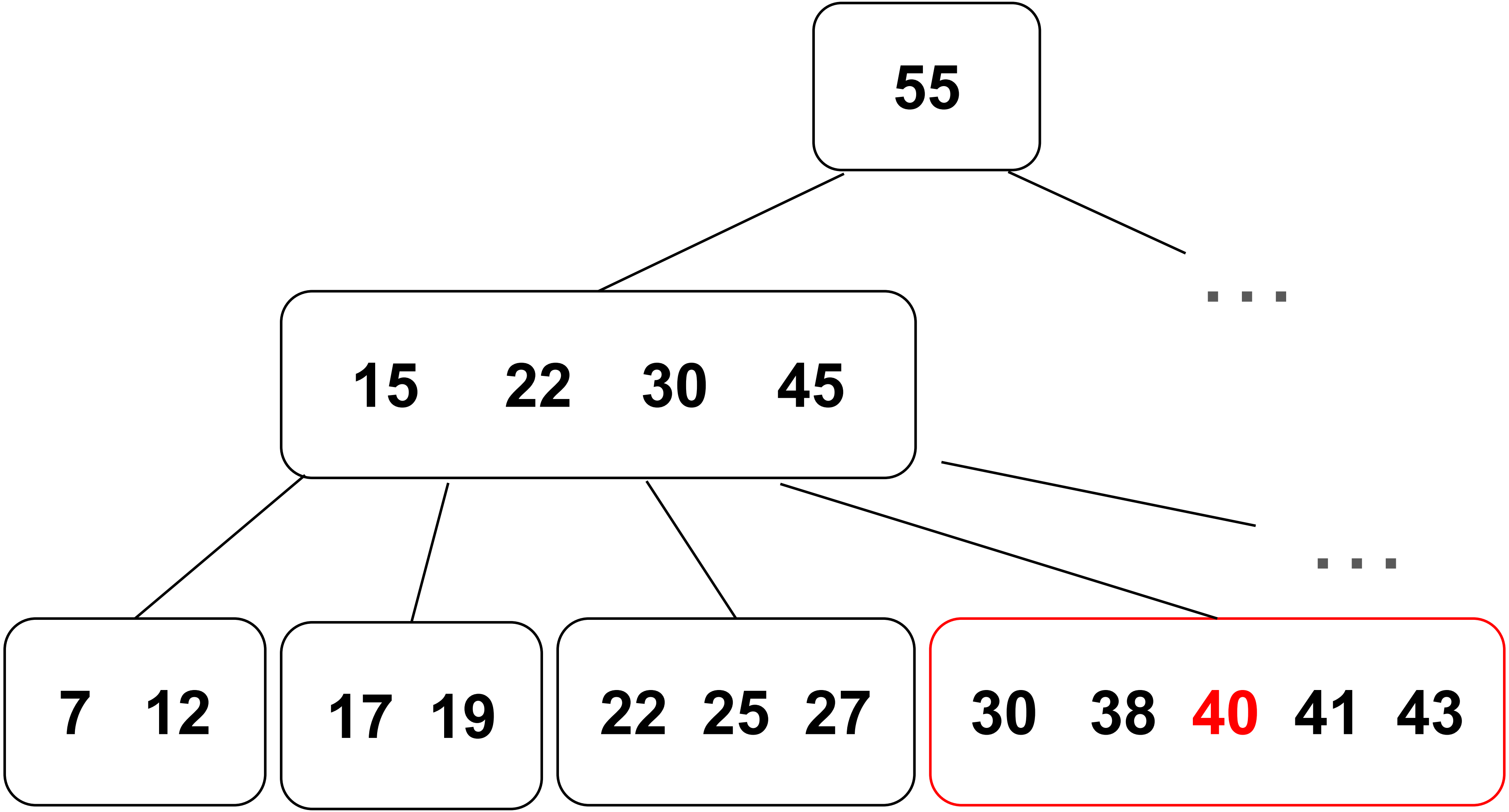

55
15 30 45
...
7 12
17 22 27
30 38 43
...

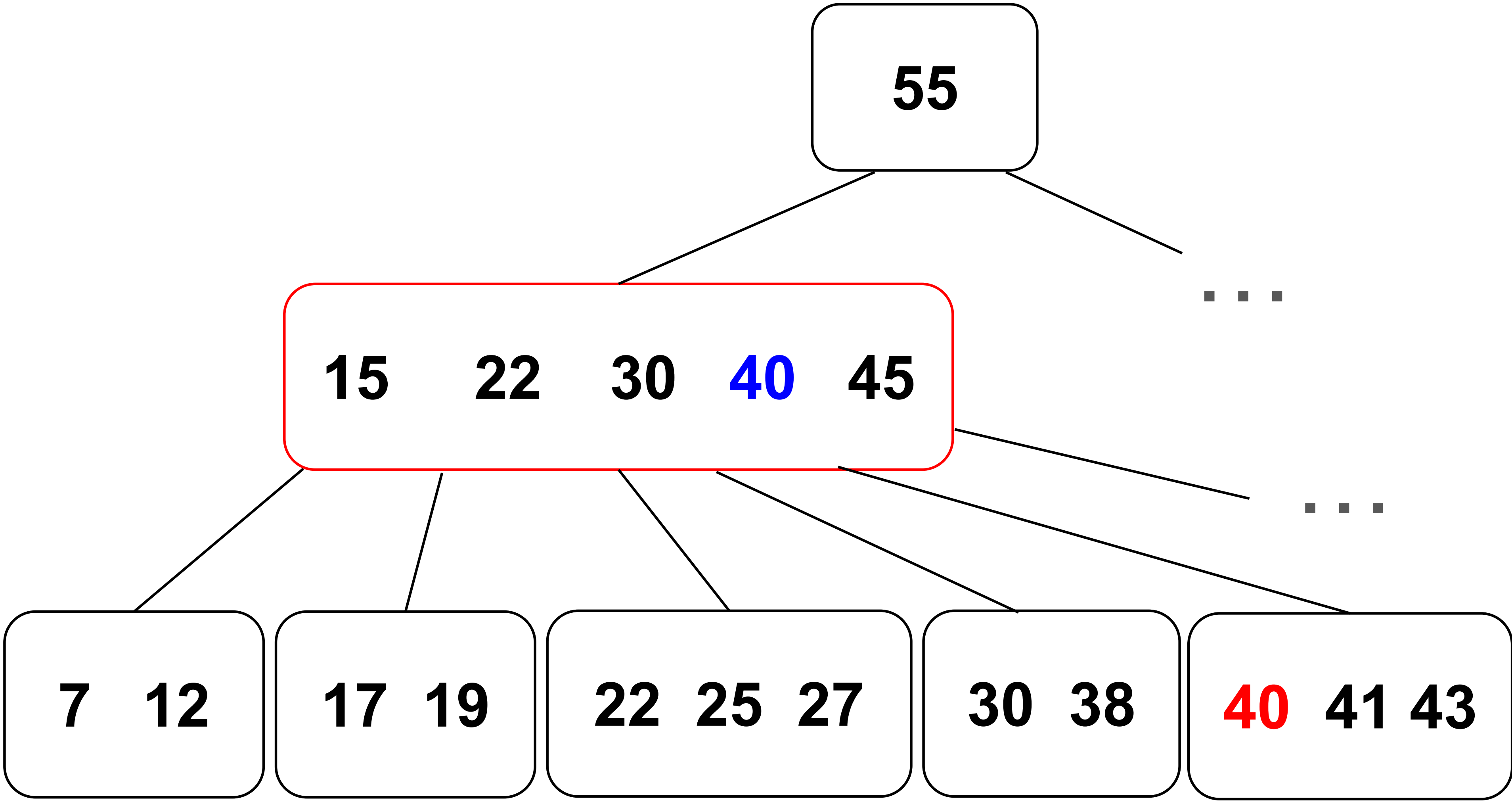

55
15 30 45
. . .
7 12
17 19 22 27
30 38 43
. . .

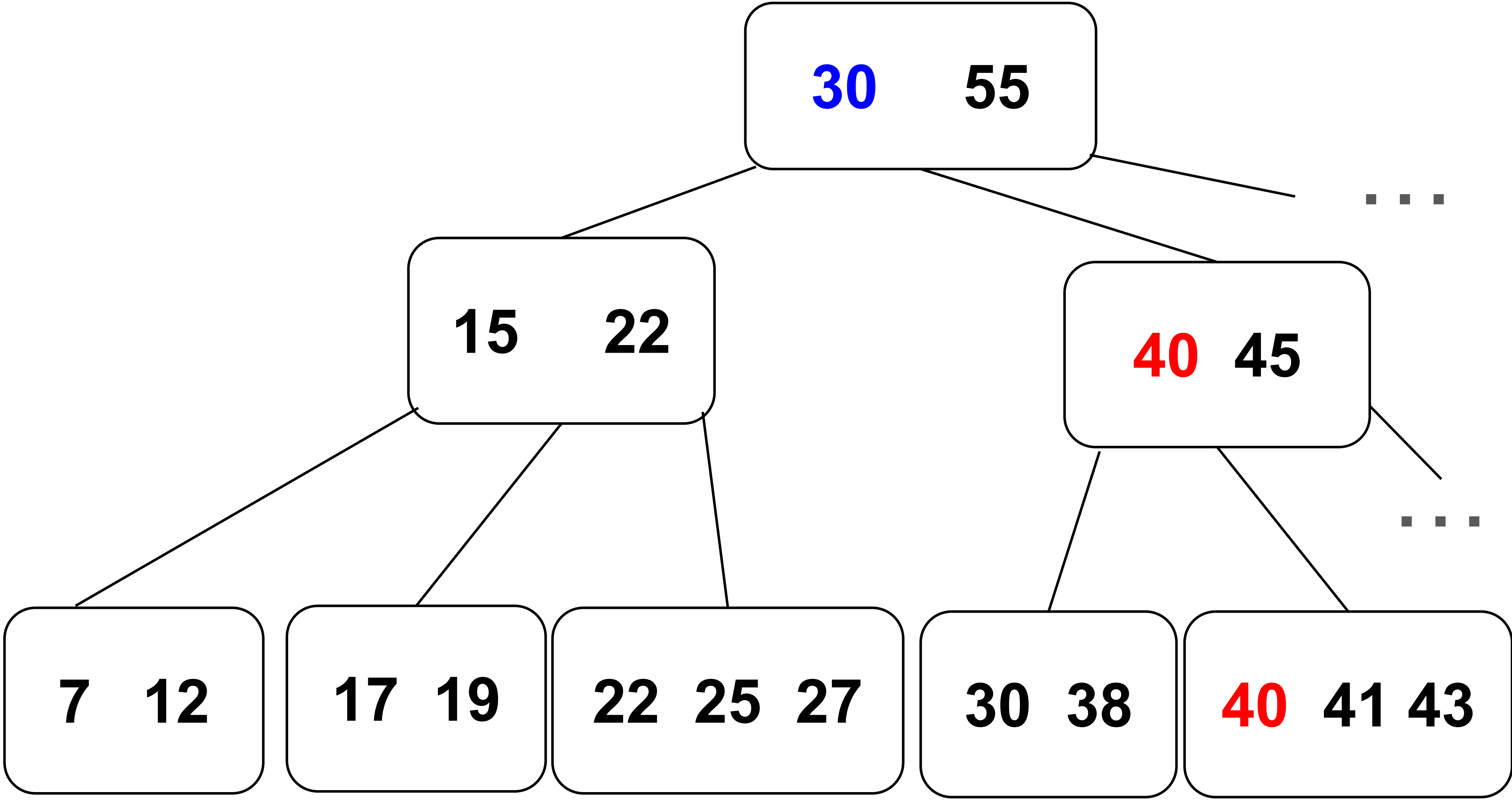

55
...
15 30 45
...
7 12
17 19 22 25 27
30 38 43

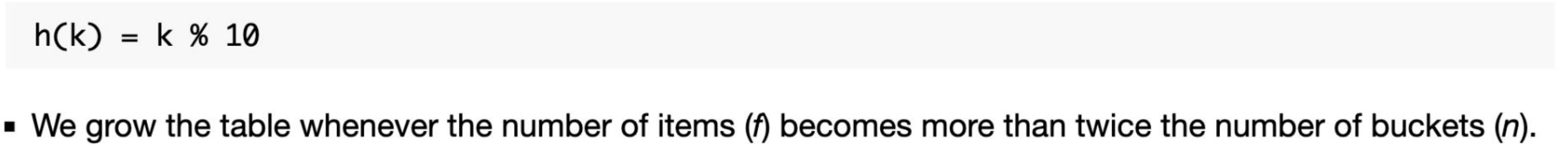

55
...
15 22 30 45
...
7 12
17 19
22 25 27
30 38 43


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

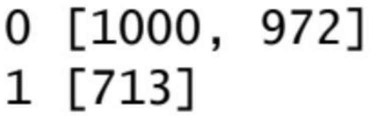

55
...
15 22 30 40 45
...
7 12
17 19
22 25 27
30 38
40 41 43


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

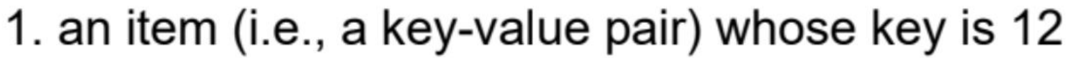


```
h(k) = k % 10
```

- 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
```

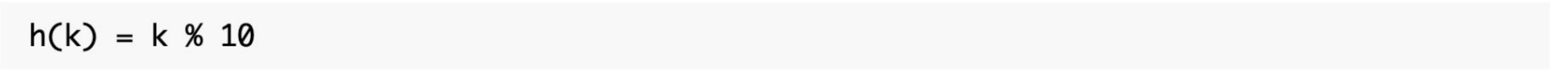
- We grow the table whenever the number of items (*f*) becomes more than twice the number of buckets (*n*).

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

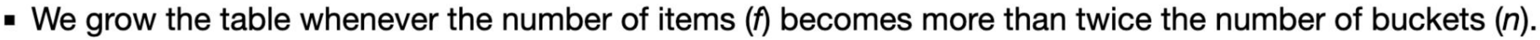
1. an item (i.e., a key-value pair) whose key is 12

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

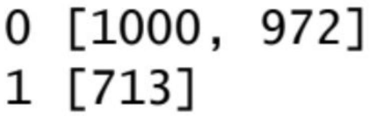
```
0 [1000, 972, 12]
1 [713]
```



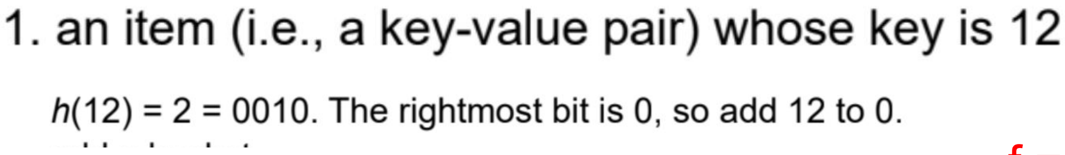
```
h(k) = k % 10
```



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

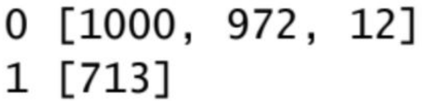


```
0 [1000, 972]
1 [713]
```



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]
```

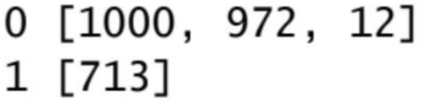
f = 4, n = 2

Add bucket when

f > 2n

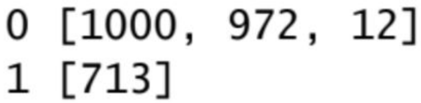
f > 4

*Add bucket on next insert



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

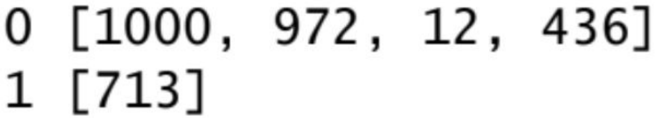
Insert 436



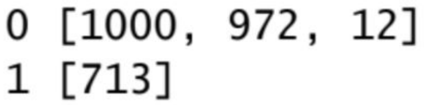
```
0 [1000, 972, 12]
1 [713]
```

Insert 436

436 % 10 = 6 = 0110



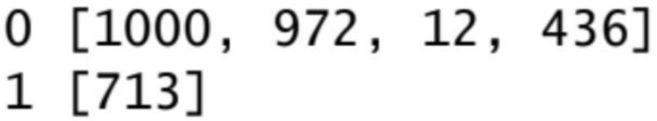
```
0 [1000, 972, 12, 436]
1 [713]
```



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

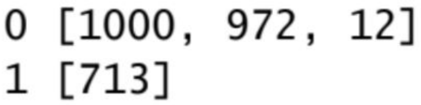
Insert 436

436 % 10 = 6 = 0110



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

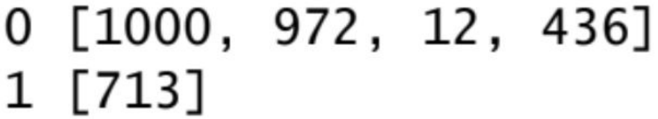
f > 2n, add bucket



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

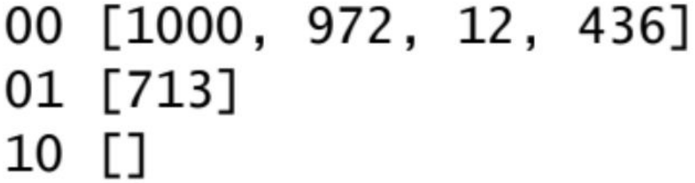
Insert 436

436 % 10 = 6 = 0110

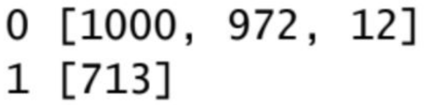


```
0 [1000, 972, 12, 436]
1 [713]
```

f > 2n, add bucket



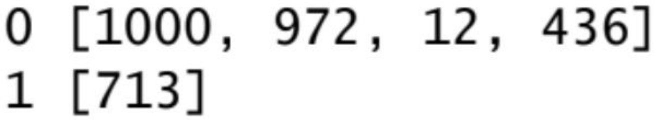
```
00 [1000, 972, 12, 436]
01 [713]
10 []
```



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

Insert 436

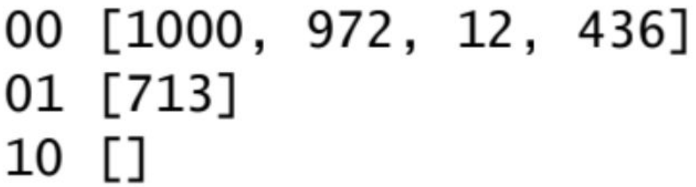
436 % 10 = 6 = 0110



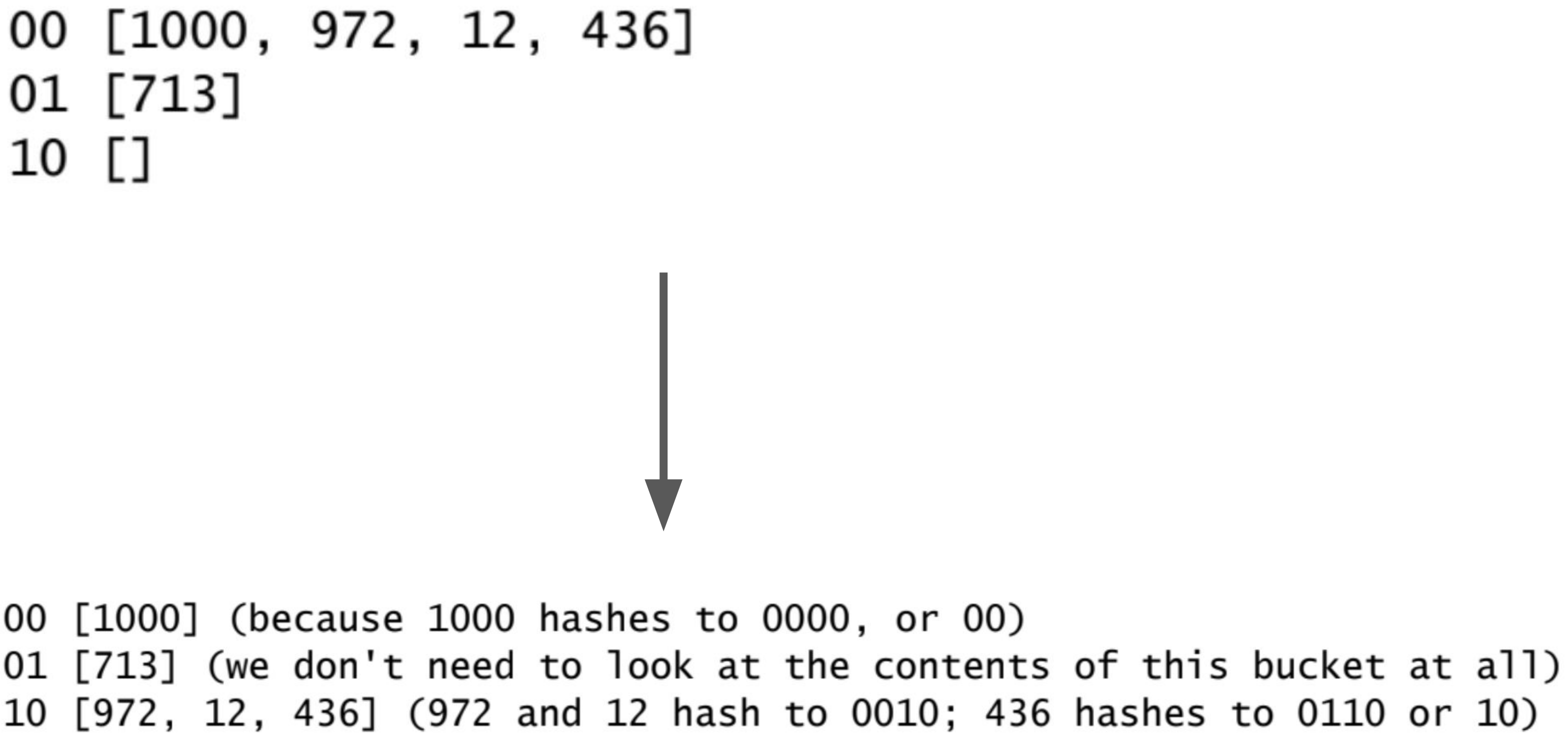
```
0 [1000, 972, 12, 436]
1 [713]
```

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]

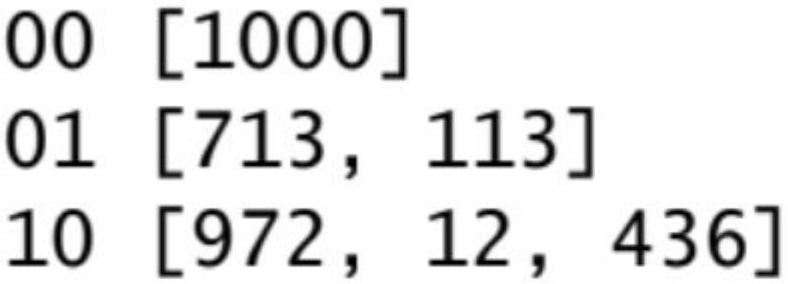
insert 113

```
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]
01 [713, 113]
10 [972, 12, 436]
```

insert 116

```
00 [1000]
01 [713, 113]
10 [972, 12, 436, 116]
```

Split

```
00 [1000]
01 [713, 113]
10 [972, 12, 436, 116]
11 []
```

rehash

```
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)
```