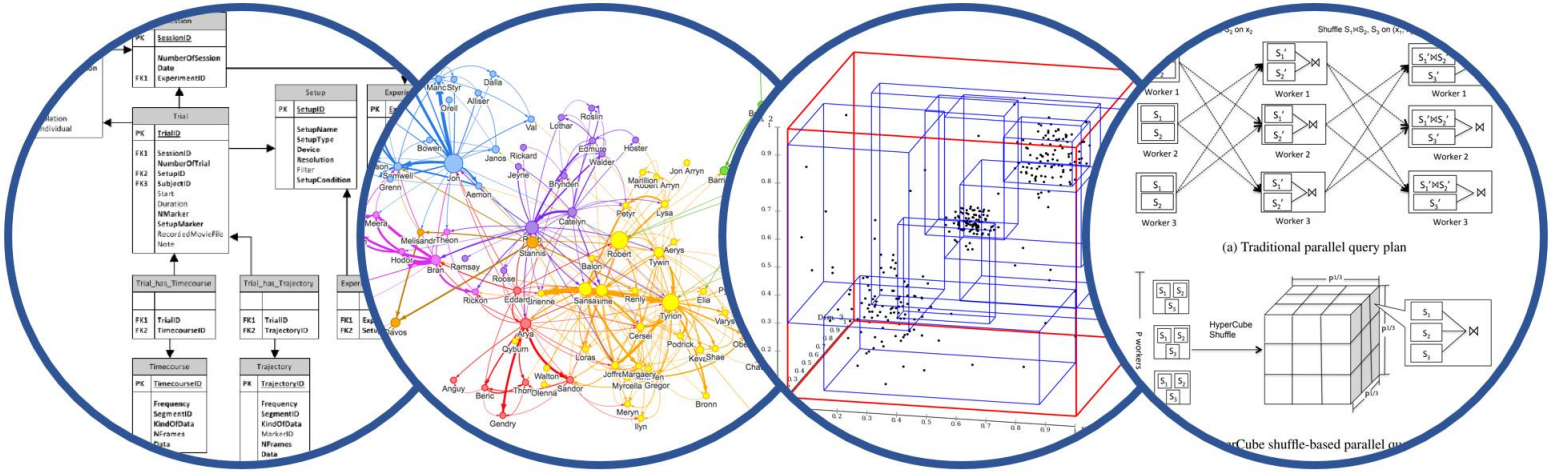


Introduction to Data Management

Indexes

Paul G. Allen School of Computer Science and Engineering
University of Washington, Seattle

Announcements

HW6: Part 2 due 11/27. More work than part 1

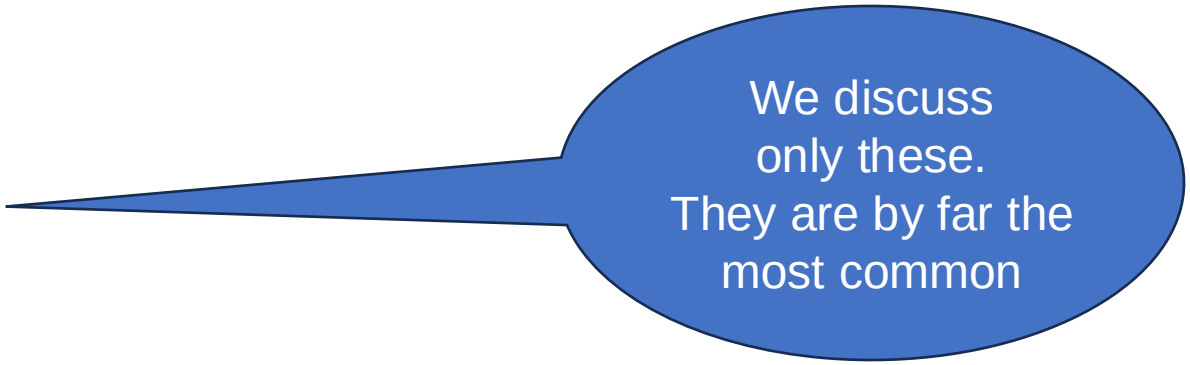
Announcements

- Lecture on Wednesday, Nov. 27:
  - No in-person lecture
  - Recording only
- Monday, Dec. 9, Final Review session
  - CSE2 G10
  - 10:30 – 12:20 (we may finish earlier)

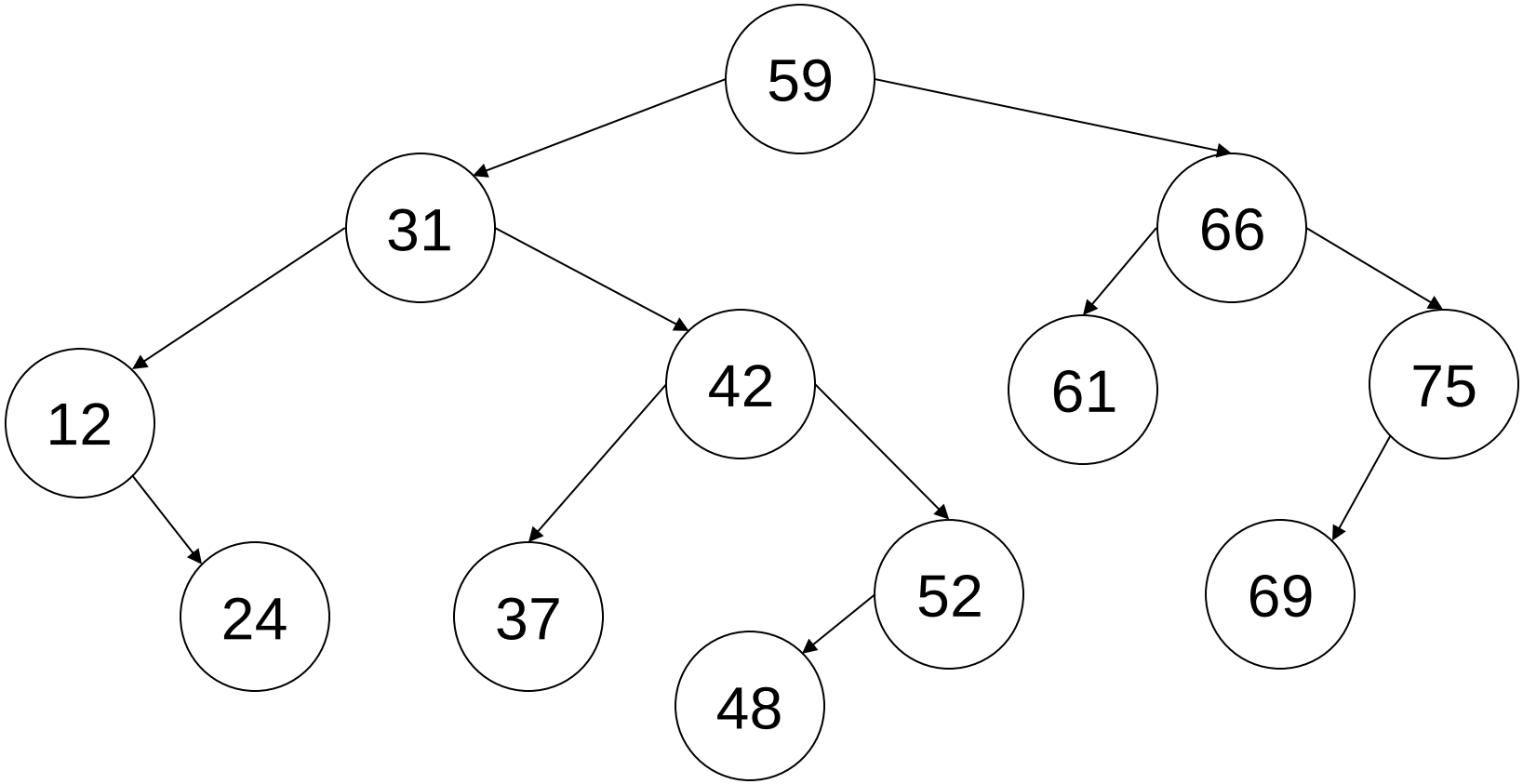
B+ Trees

Index Types

- B+ trees
- Hash tables
- Bitmaps (for attributes with few values)
- R+ tree (for 2D data)

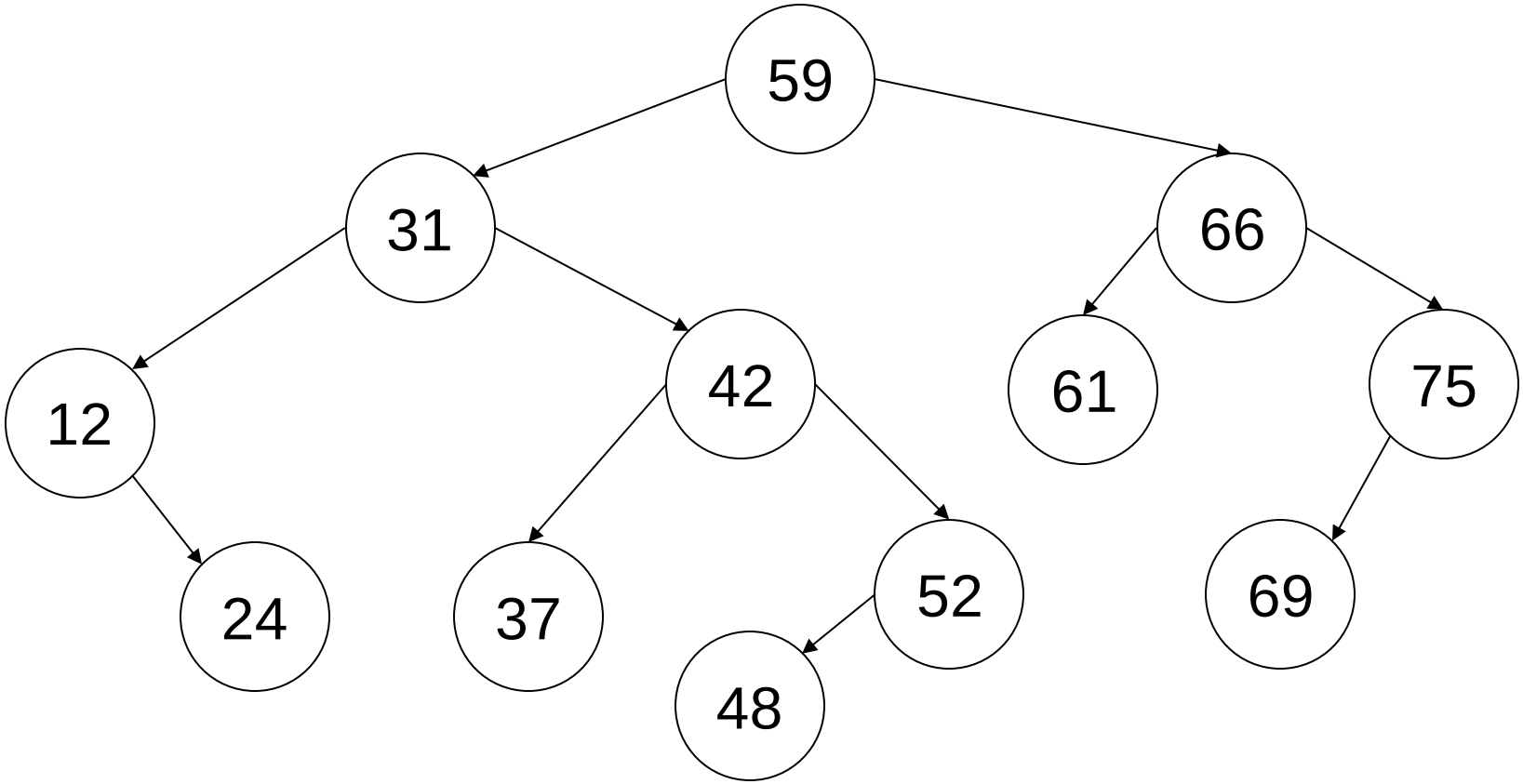


Review: Binary Search Tree



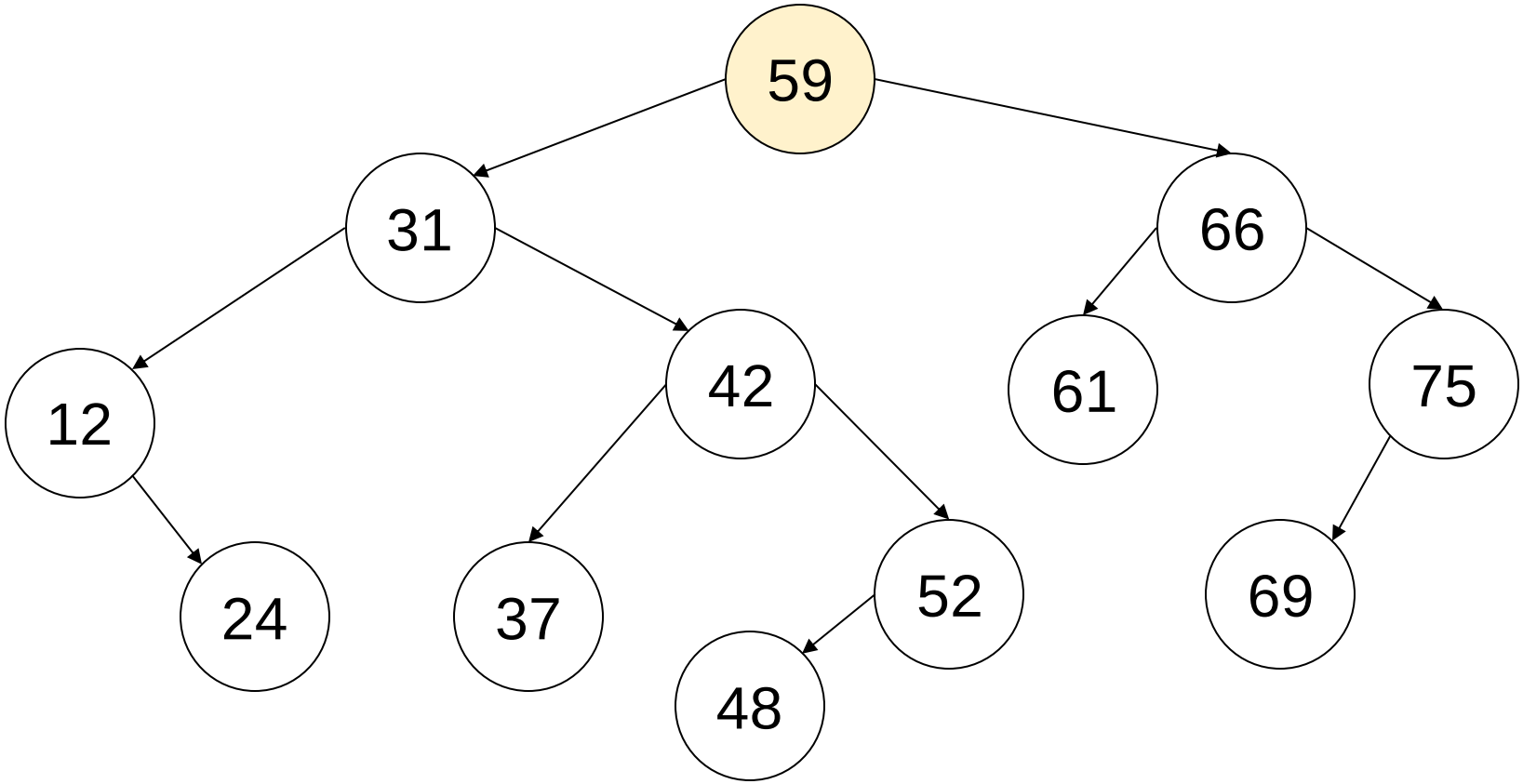
Review: Binary Search Tree

Find 48



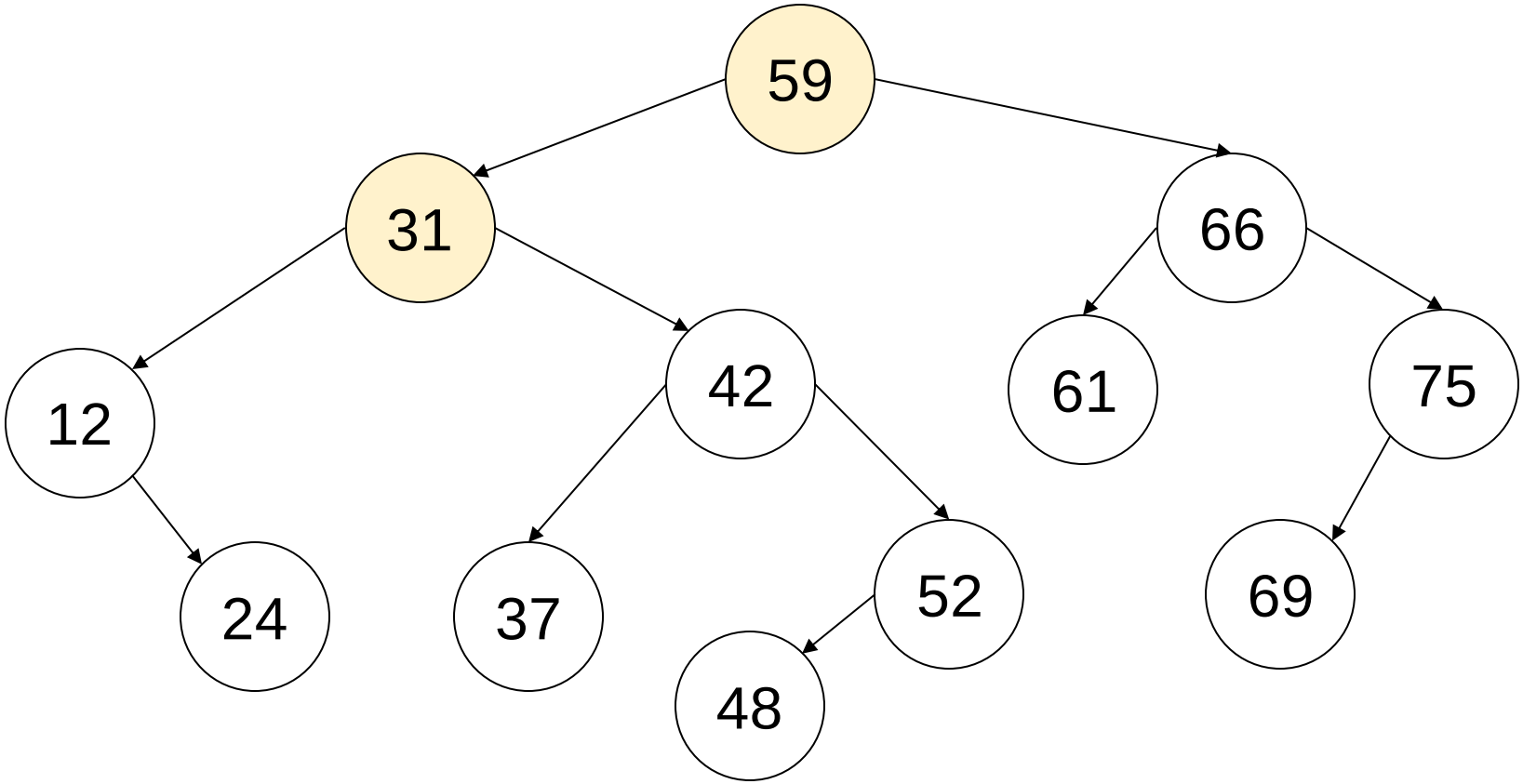
Review: Binary Search Tree

Find 48



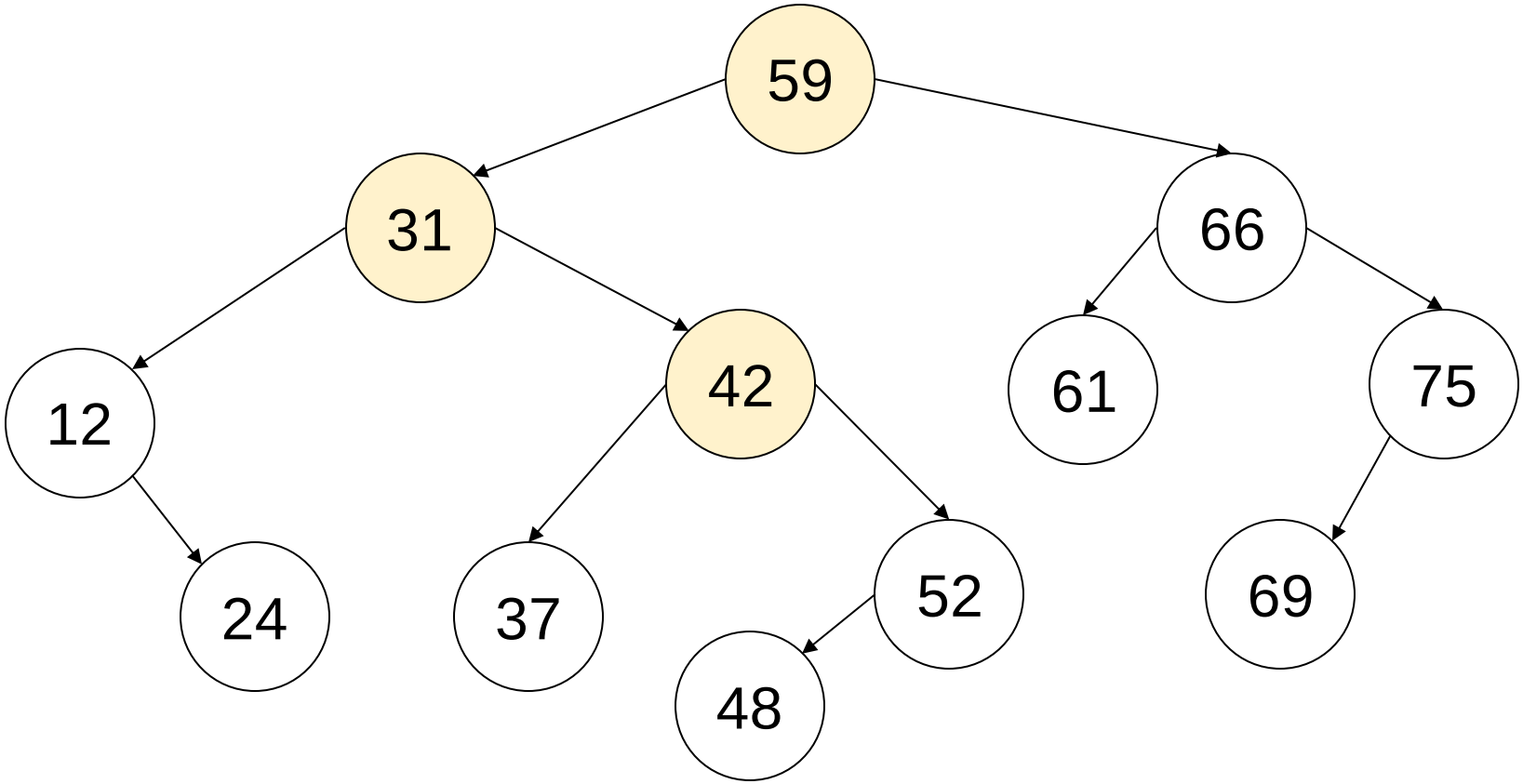
Review: Binary Search Tree

Find 48



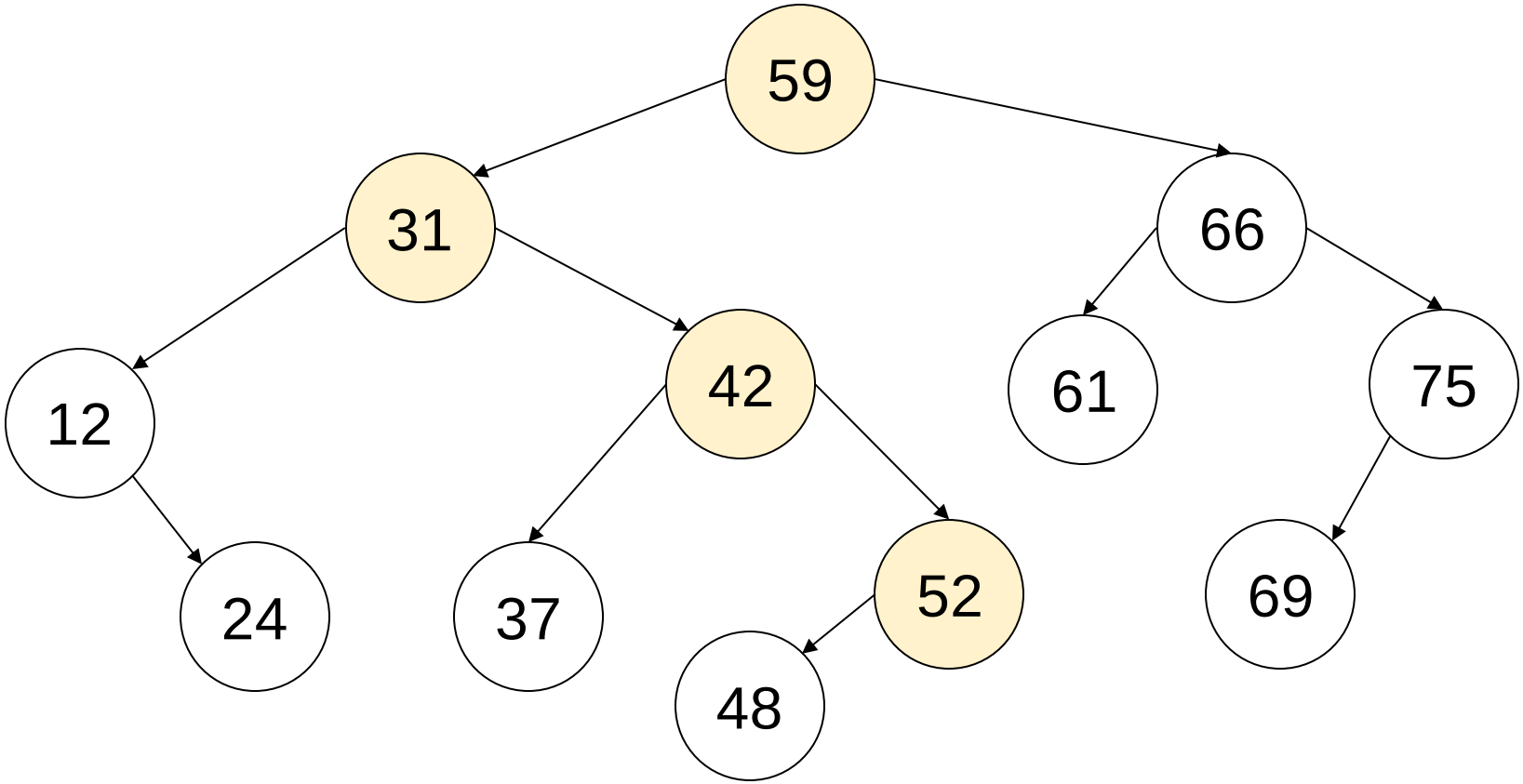
Review: Binary Search Tree

Find 48



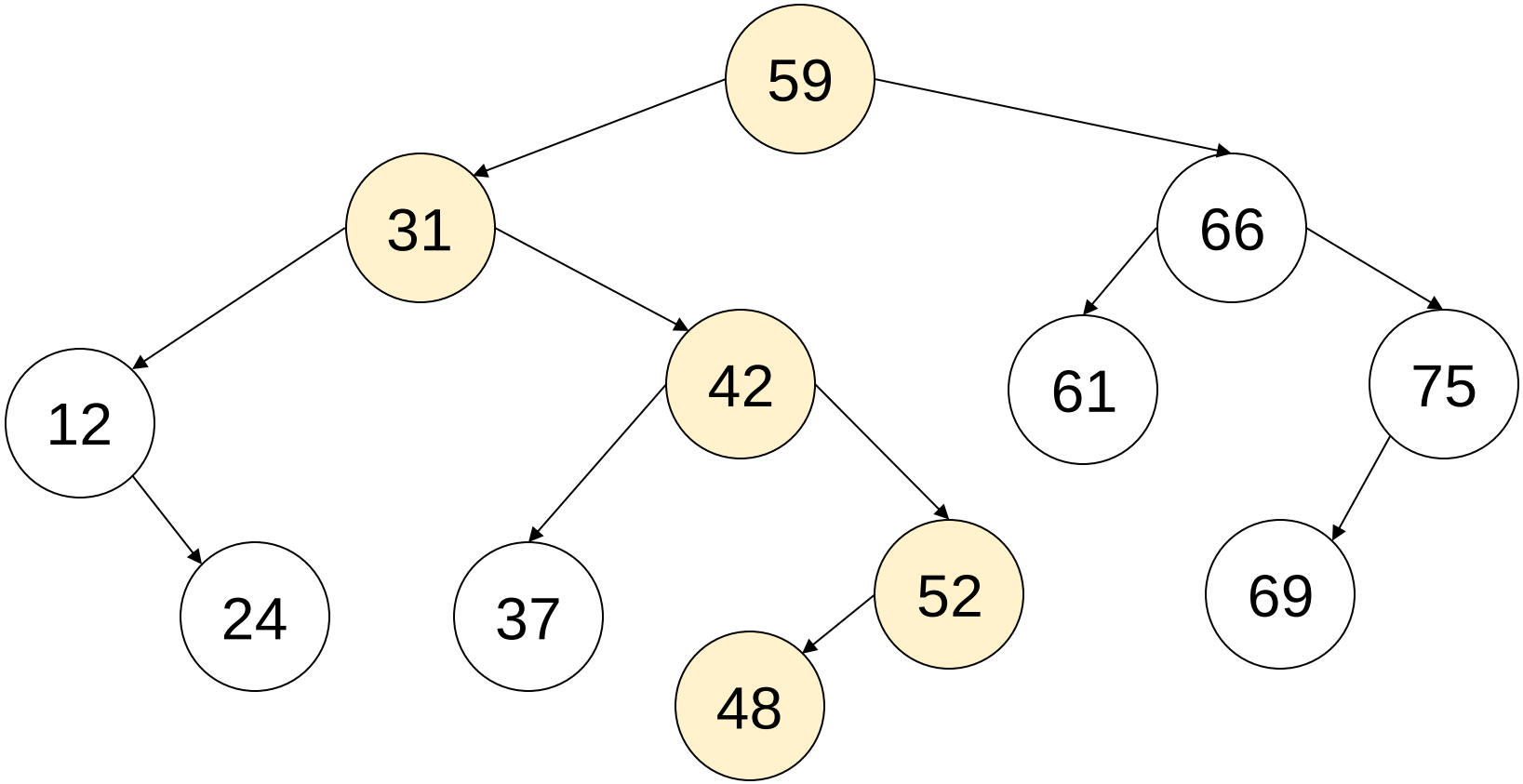
Review: Binary Search Tree

Find 48



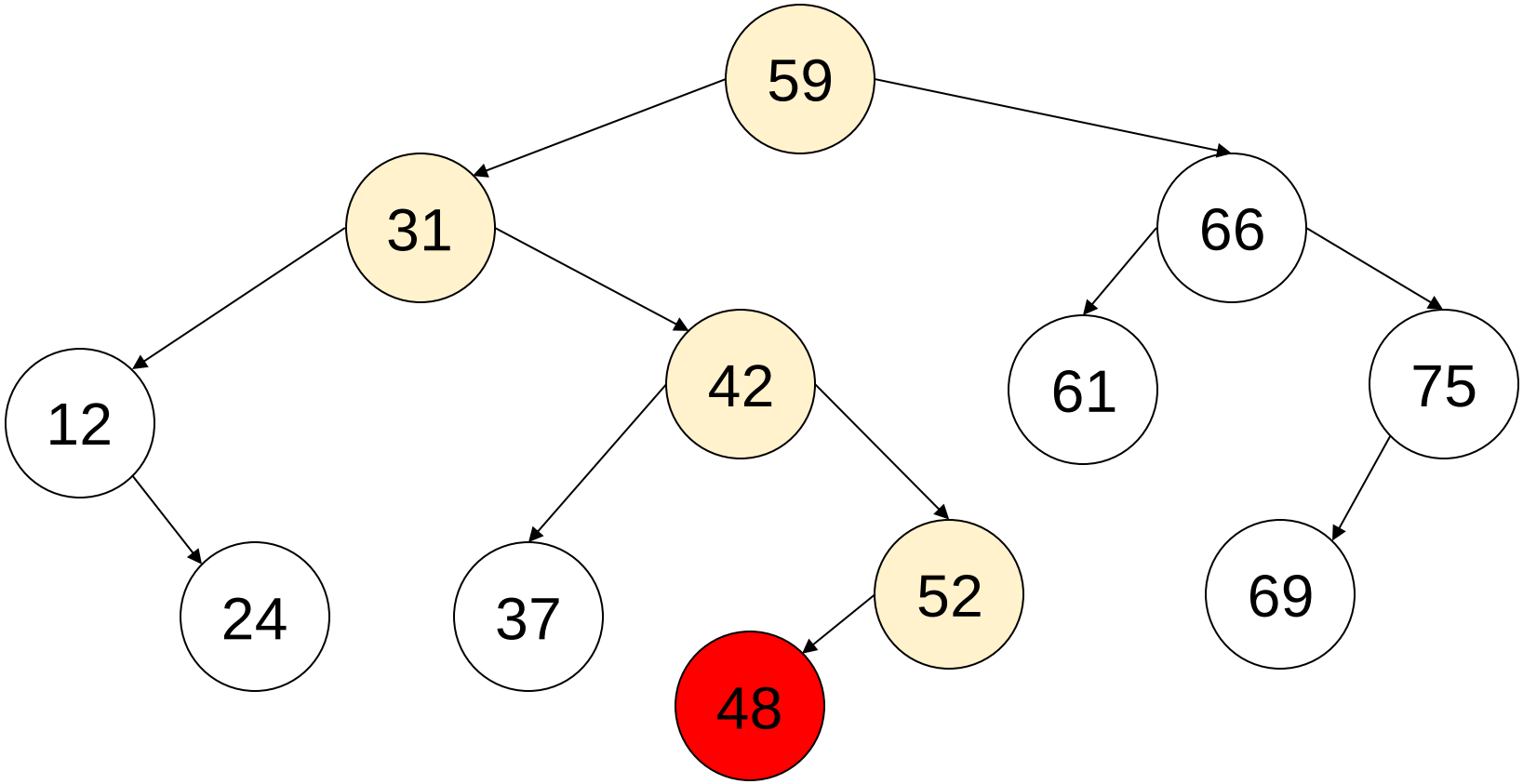
Review: Binary Search Tree

Find 48



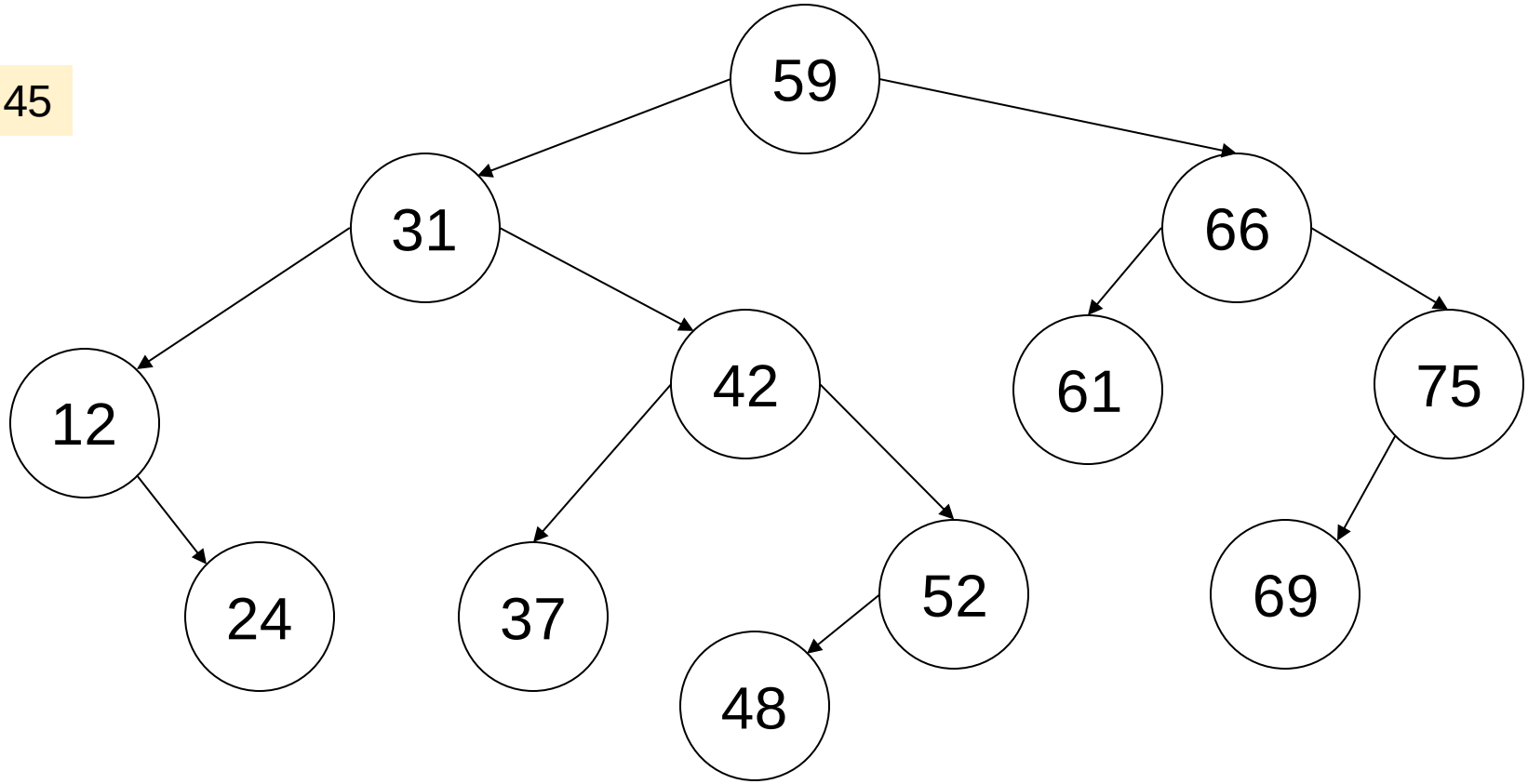
Review: Binary Search Tree

Find 48



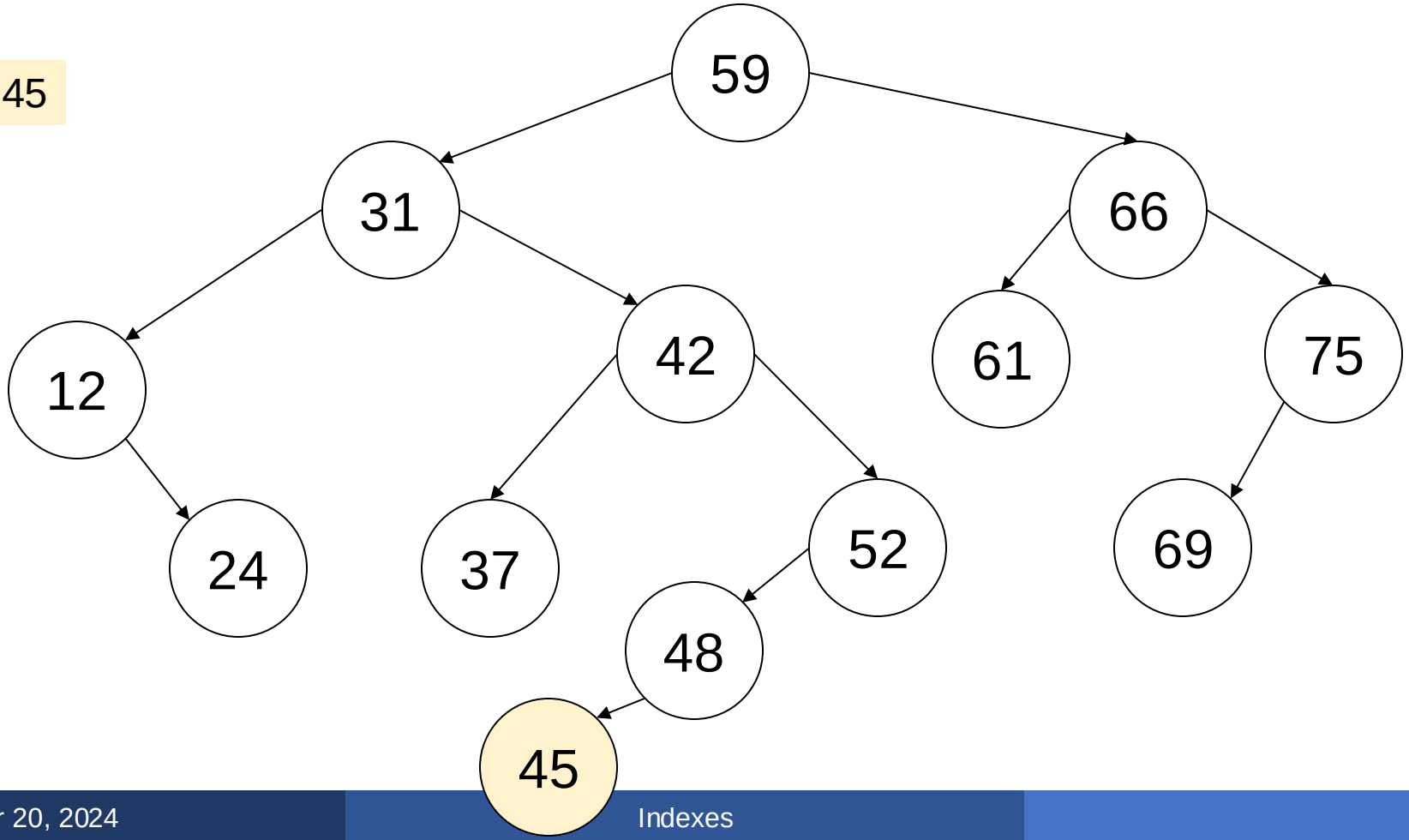
Review: Binary Search Tree

Insert 45



Review: Binary Search Tree

Insert 45



Review: Binary Search Tree

- Need to be balanced: depth=O(log N)
- Various methods to rebalance:
  - Red/black trees, splay trees, …
- Time for search/insert/delete = O(log N)

But not good for disk -- why??

B+ Trees

- Idea in B Trees
  - Make 1 node = 1 page (= 1 block)
  - Store multiple keys and children
  - Read 1 page, use many keys
- Idea in B+ Trees
  - Keys are stored only on leaves
  - Internal nodes used only for guiding
  - Leaves are linked in a list, for range queries

B+ Trees Properties

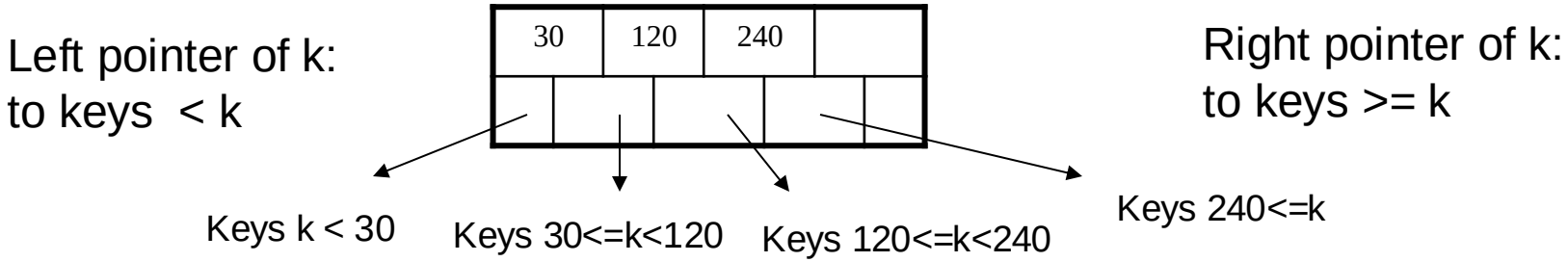
- For each node except the root, maintain 50% occupancy of keys
- Insert and delete must rebalance to maintain constraints

B+ Trees Properties

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

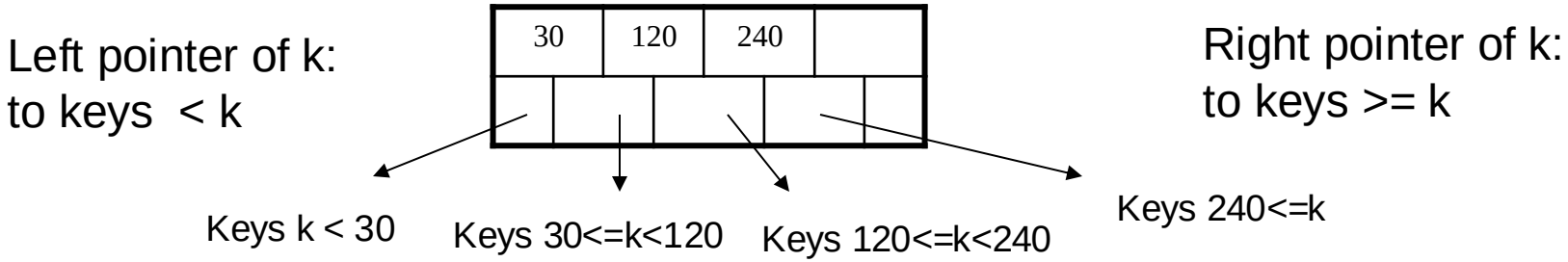
B+ Trees Properties

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

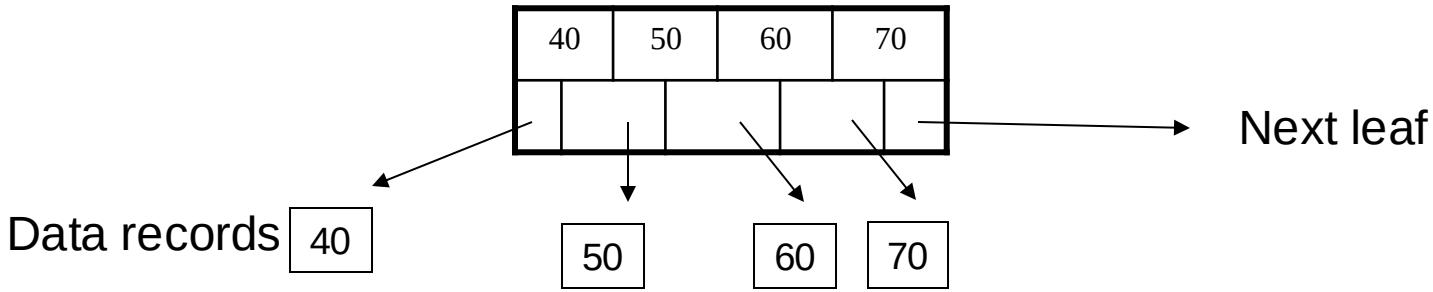


B+ Trees Properties

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

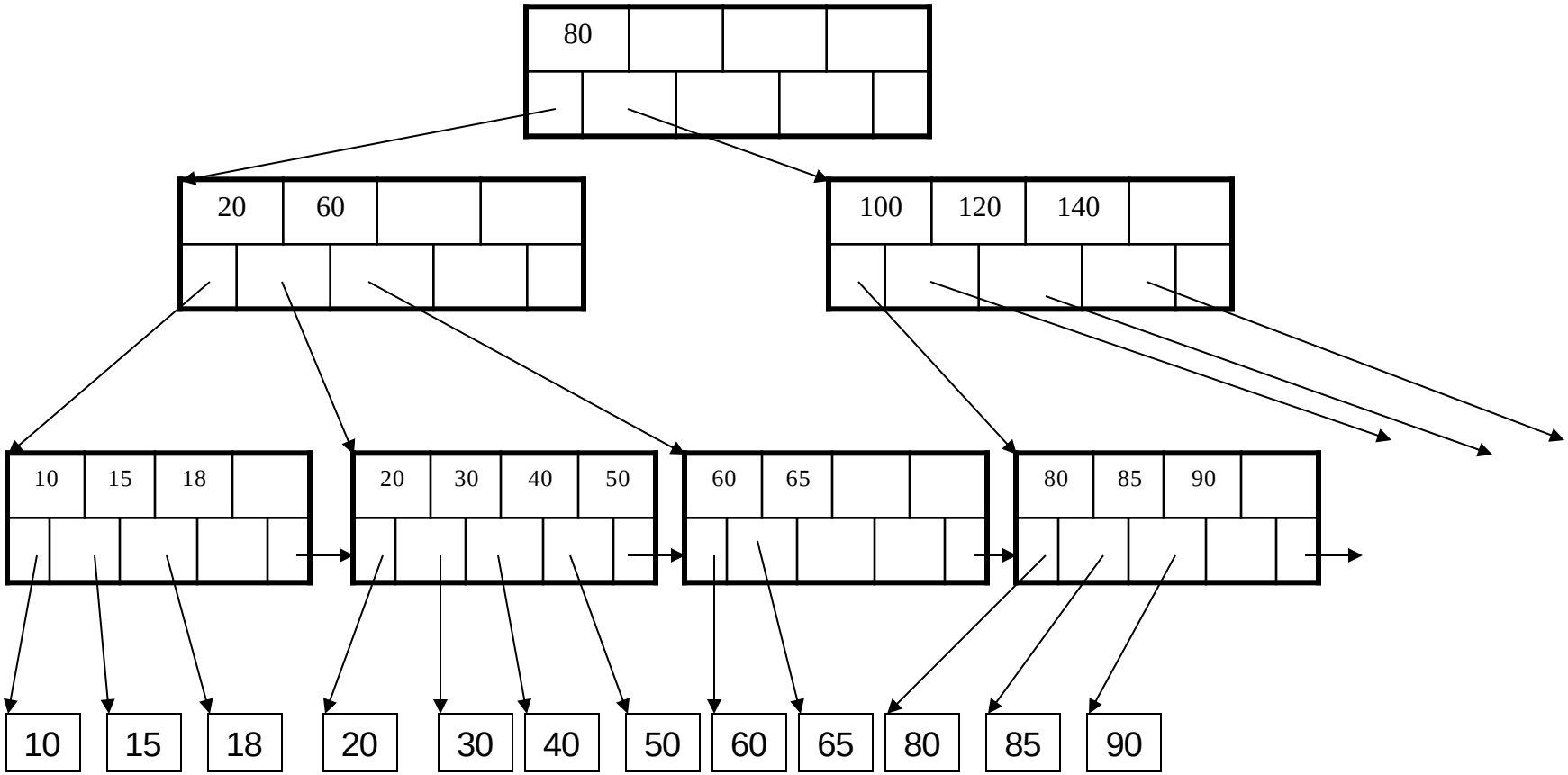


- Each leaf has **d <= m <= 2d keys**:



B+ Tree Example

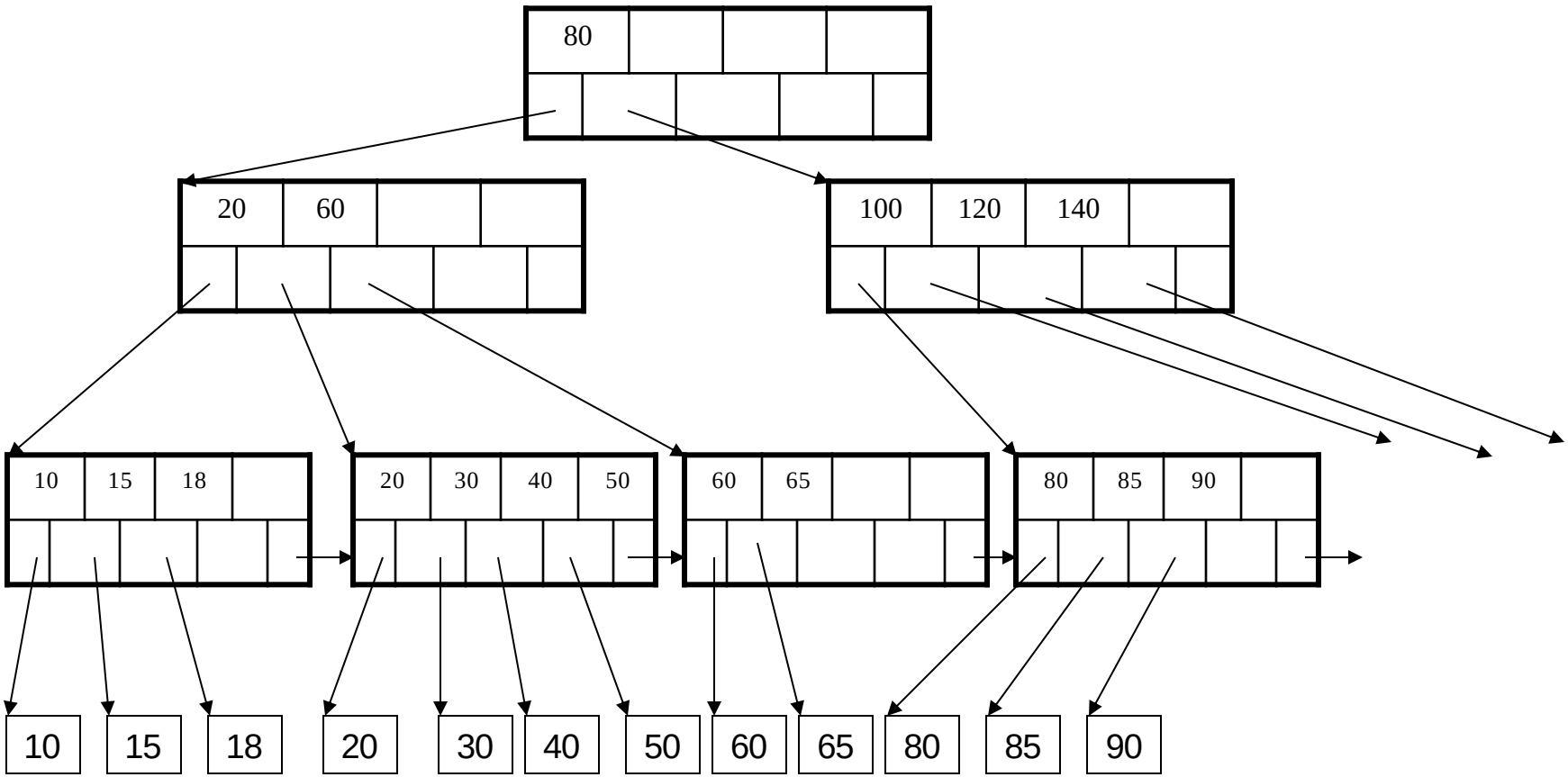
d = 2



B+ Tree Example

d = 2

Find the key 40

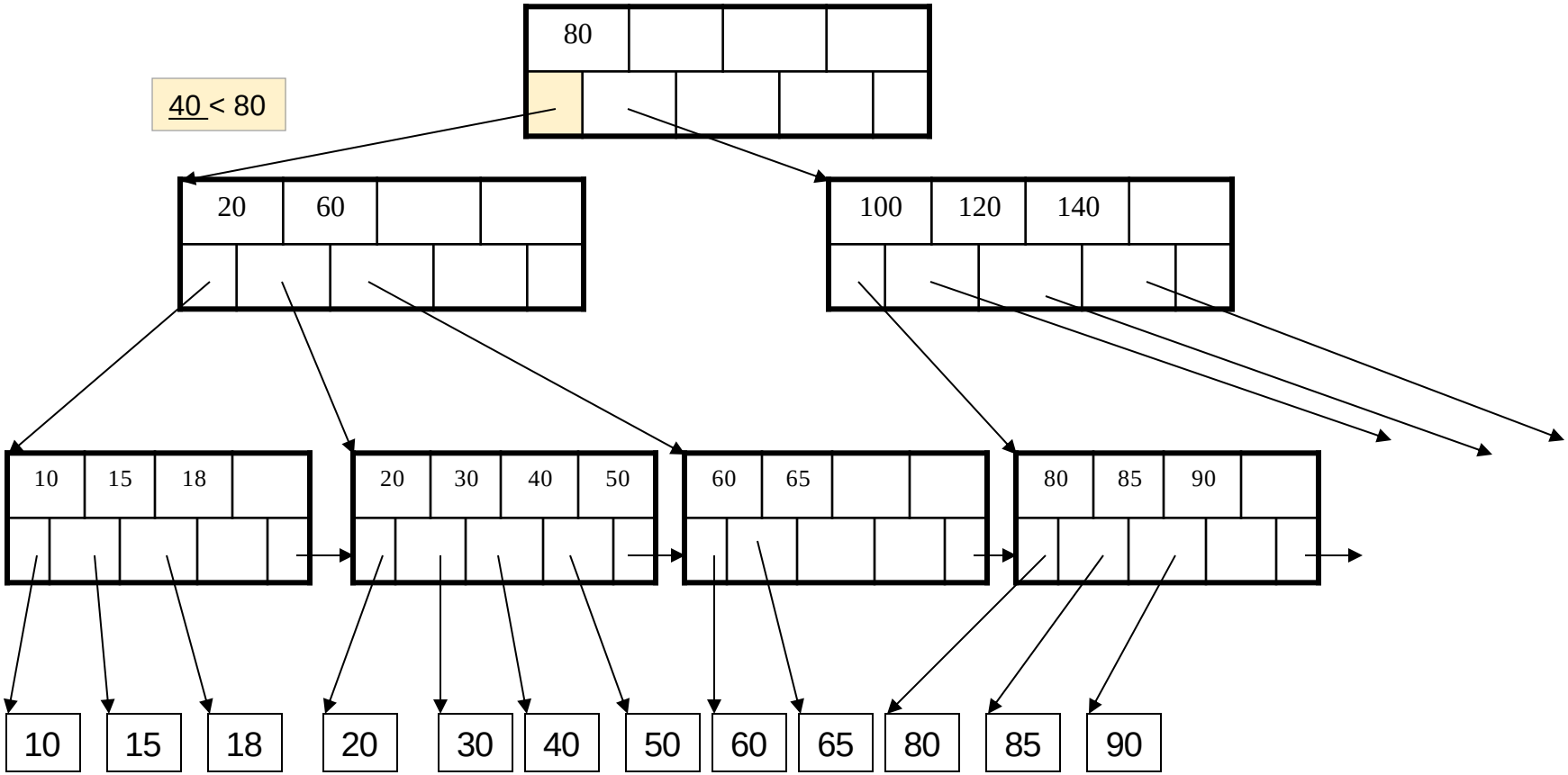


B+ Tree Example

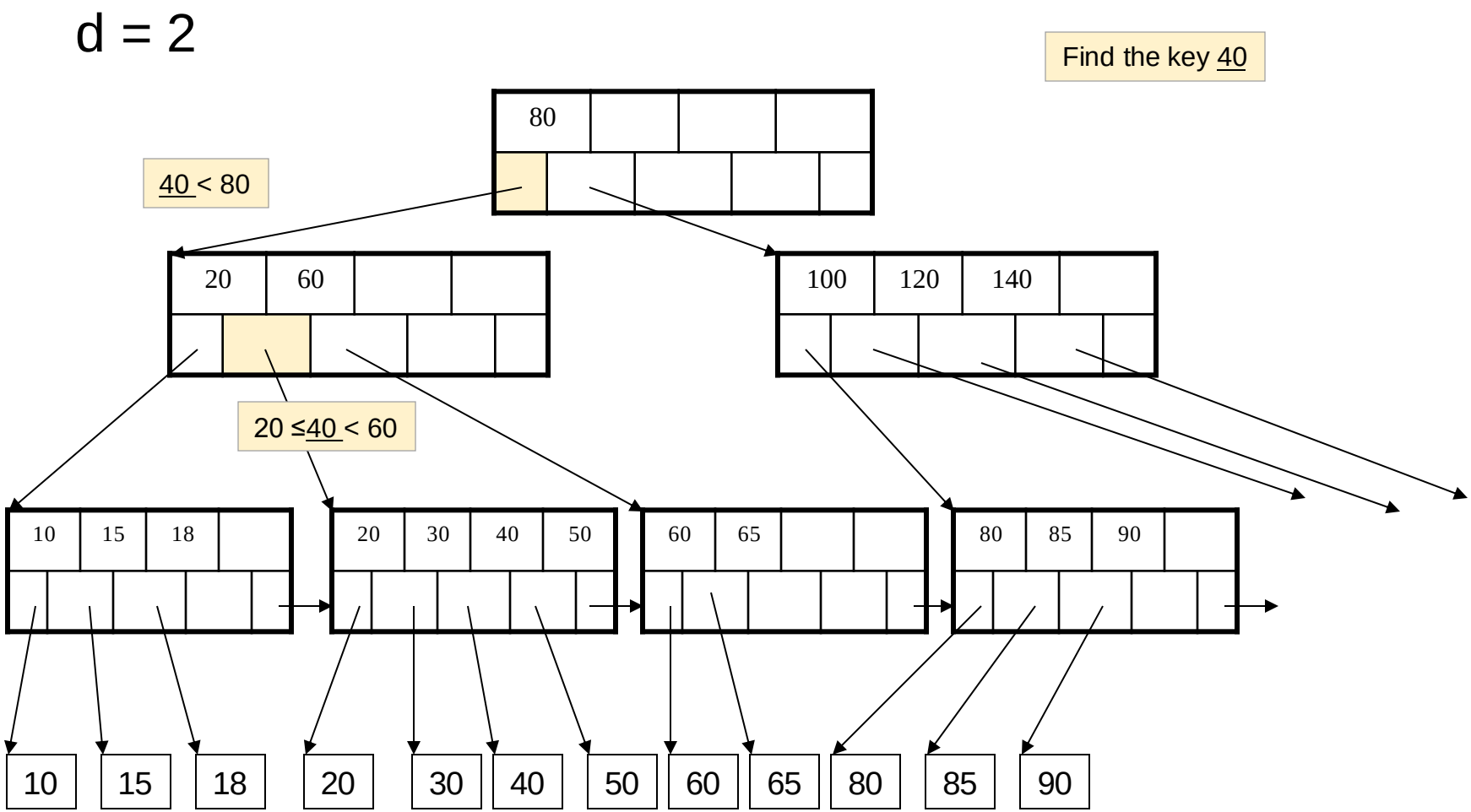
d = 2

Find the key 40

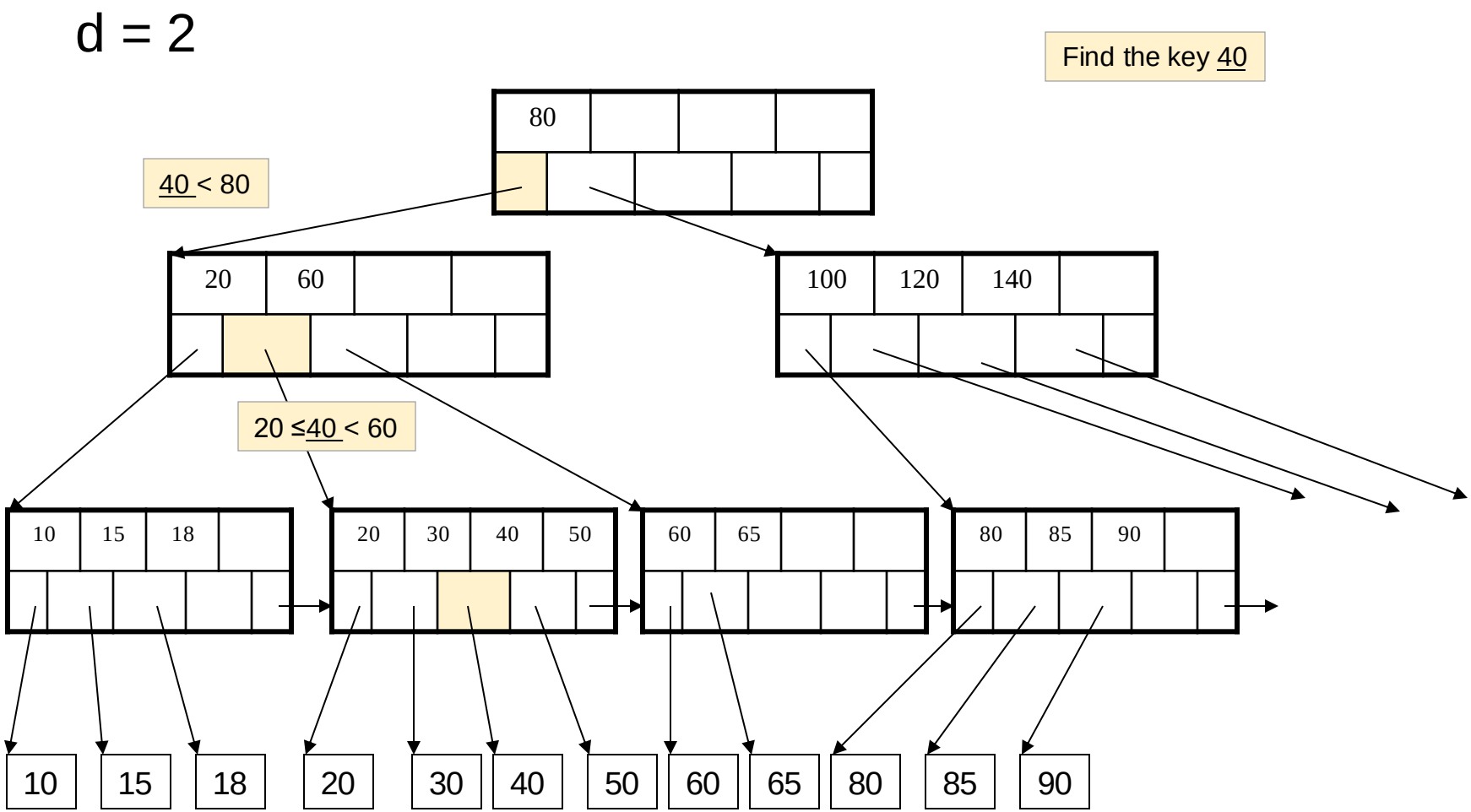
40 < 80



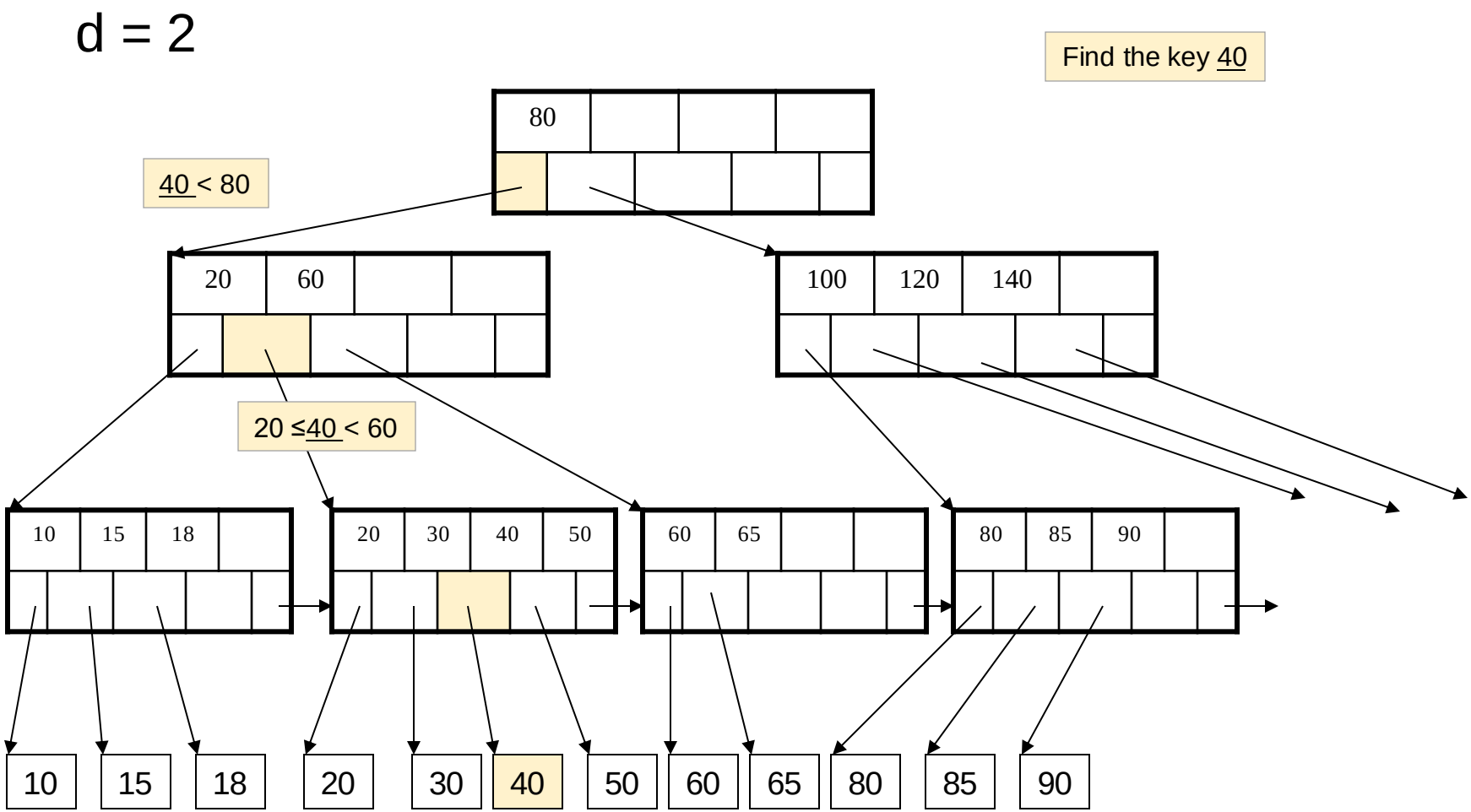
B+ Tree Example



B+ Tree Example

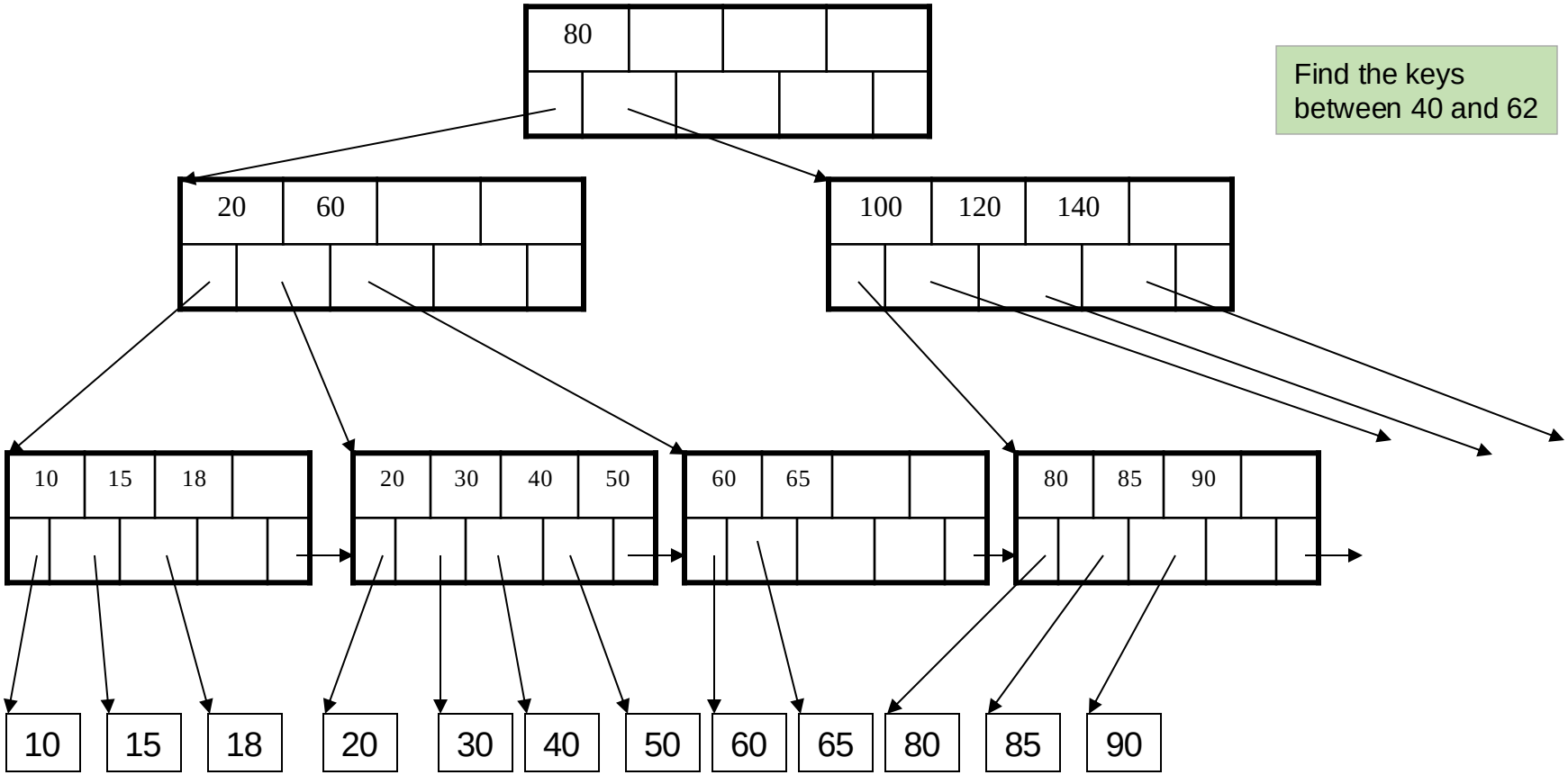


B+ Tree Example



B+ Tree Example

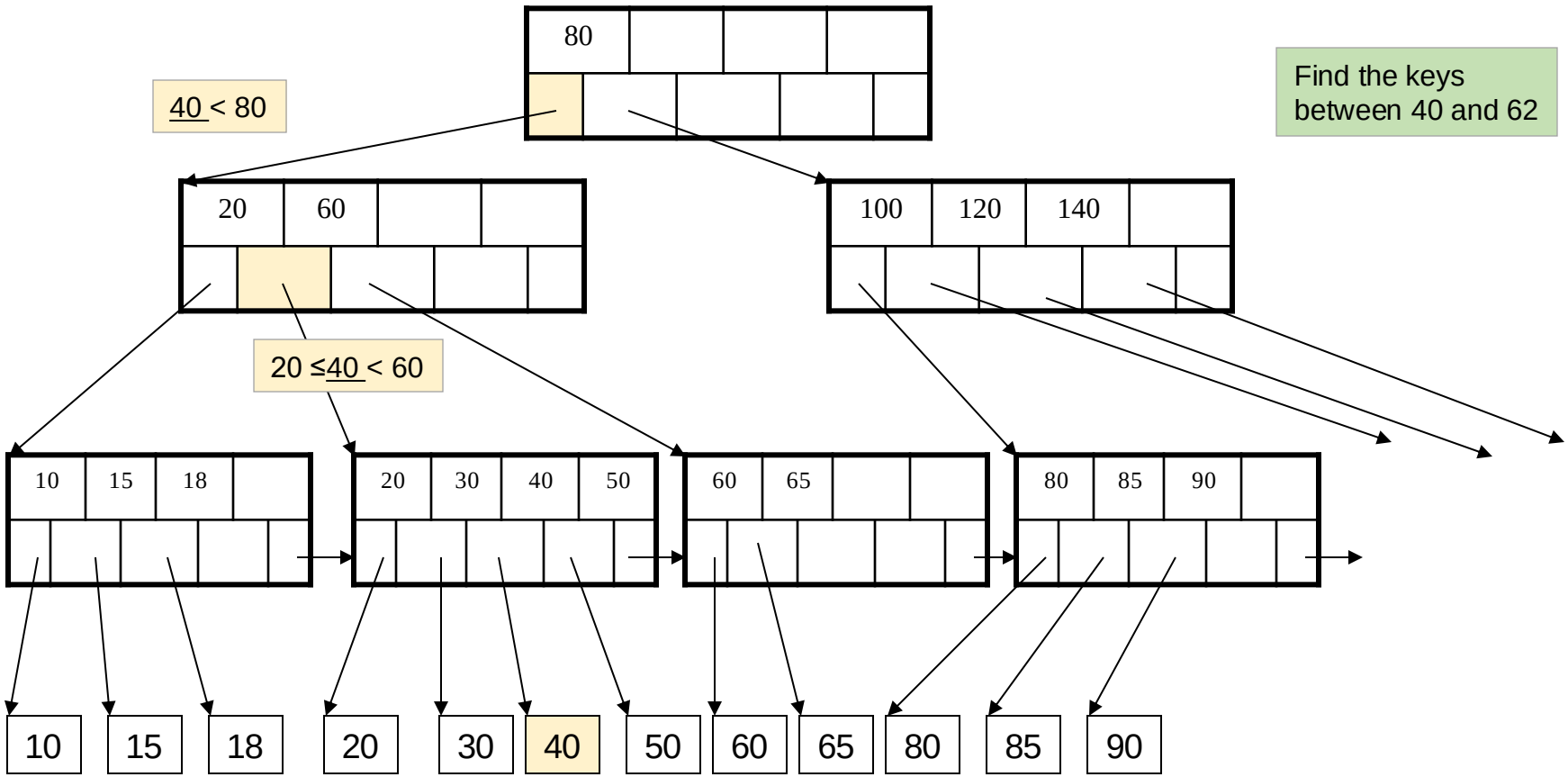
d = 2



Find the keys between 40 and 62

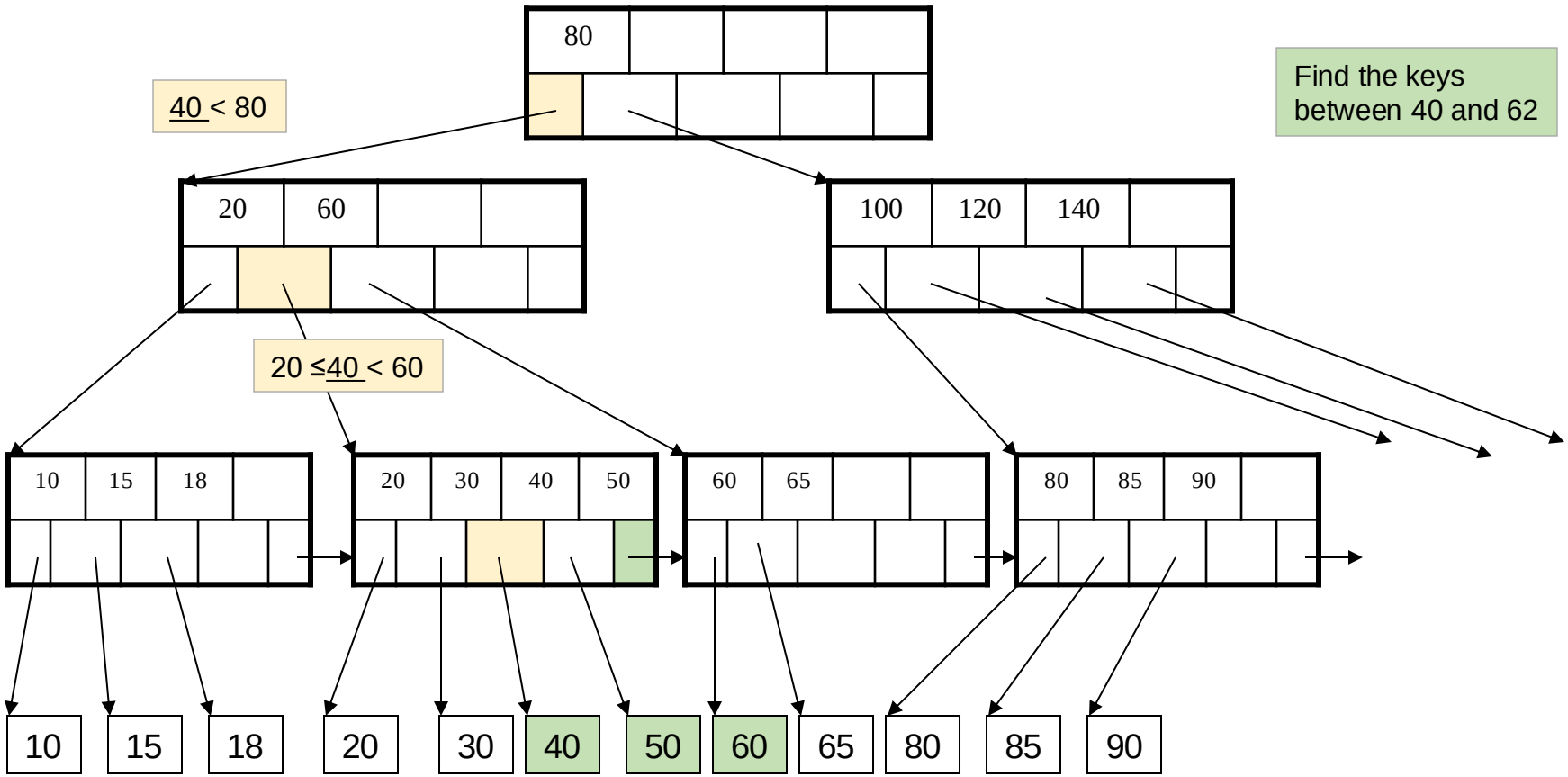
B+ Tree Example

d = 2



B+ Tree Example

d = 2



Search time

A B+ tree is perfectly balanced

- Depth=$\log_m N$ where m = avg # of children

Example: N = 1,000,000,000,000 data items

- Binary search tree: log N = 40
- B+ tree, assume m=128: $\log_m N$ = 6 block reads

You can find the data after at most 6 reads!

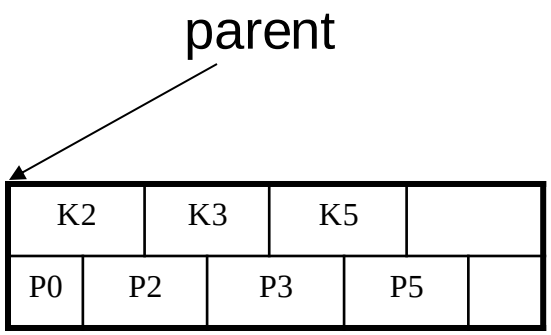
B+ Trees:
Insert and Delete

Insertion in a B+ Tree: Summary

Insert (K, P)

- Find leaf where K belongs, insert
- If no overflow (2d keys or less), halt

Insert k1

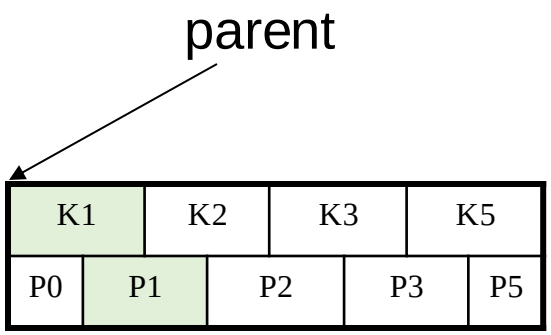


Insertion in a B+ Tree: Summary

Insert (K, P)

- Find leaf where K belongs, insert
- If no overflow (2d keys or less), halt

Insert k1

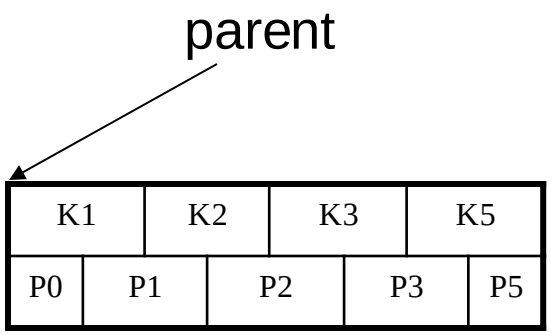


Insertion in a B+ Tree: Summary

Insert (K, P)

- Find leaf where K belongs, insert
- If no overflow (2d keys or less), halt

Insert k4

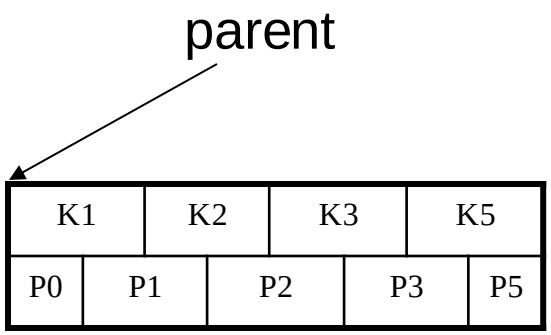


Insertion in a B+ Tree: Summary

Insert (K, P)

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

Insert k4

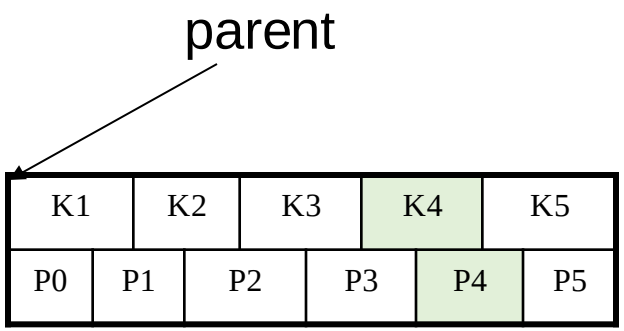


Insertion in a B+ Tree: Summary

Insert (K, P)

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

Insert k4

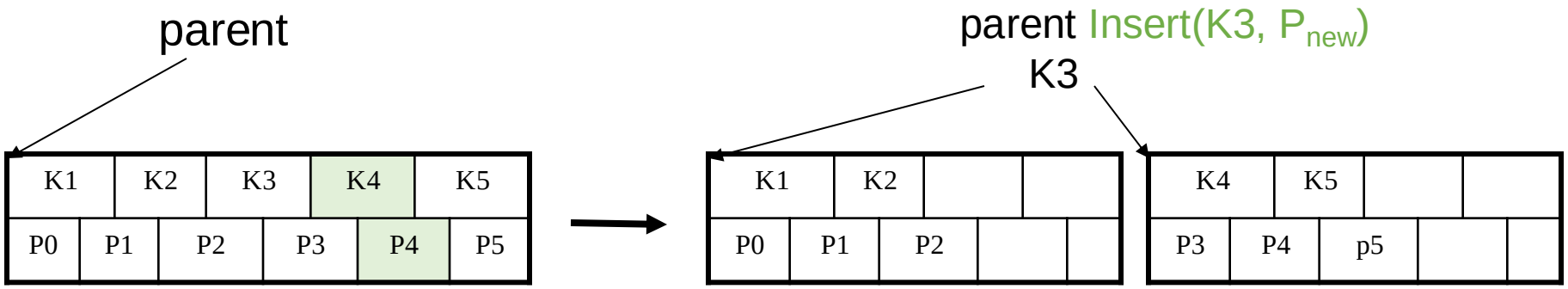


Insertion in a B+ Tree: Summary

Insert (K, P)

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

Insert k4

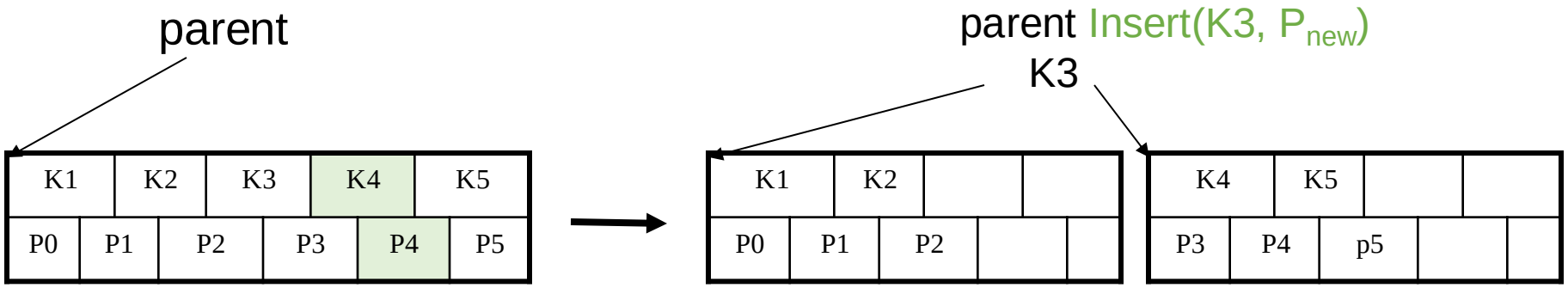


Insertion in a B+ Tree: Summary

Insert (K, P)

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

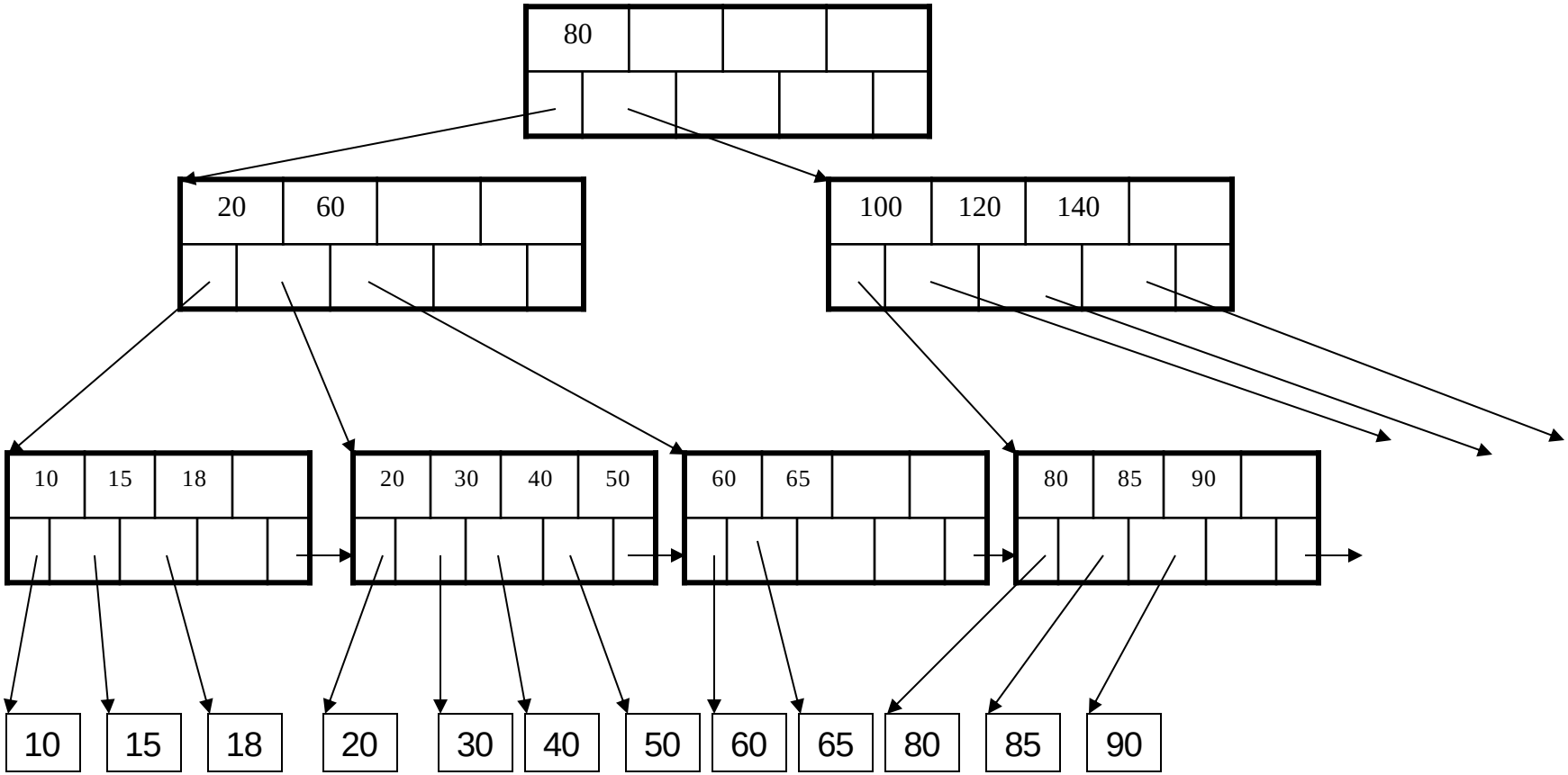
Insert k4



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

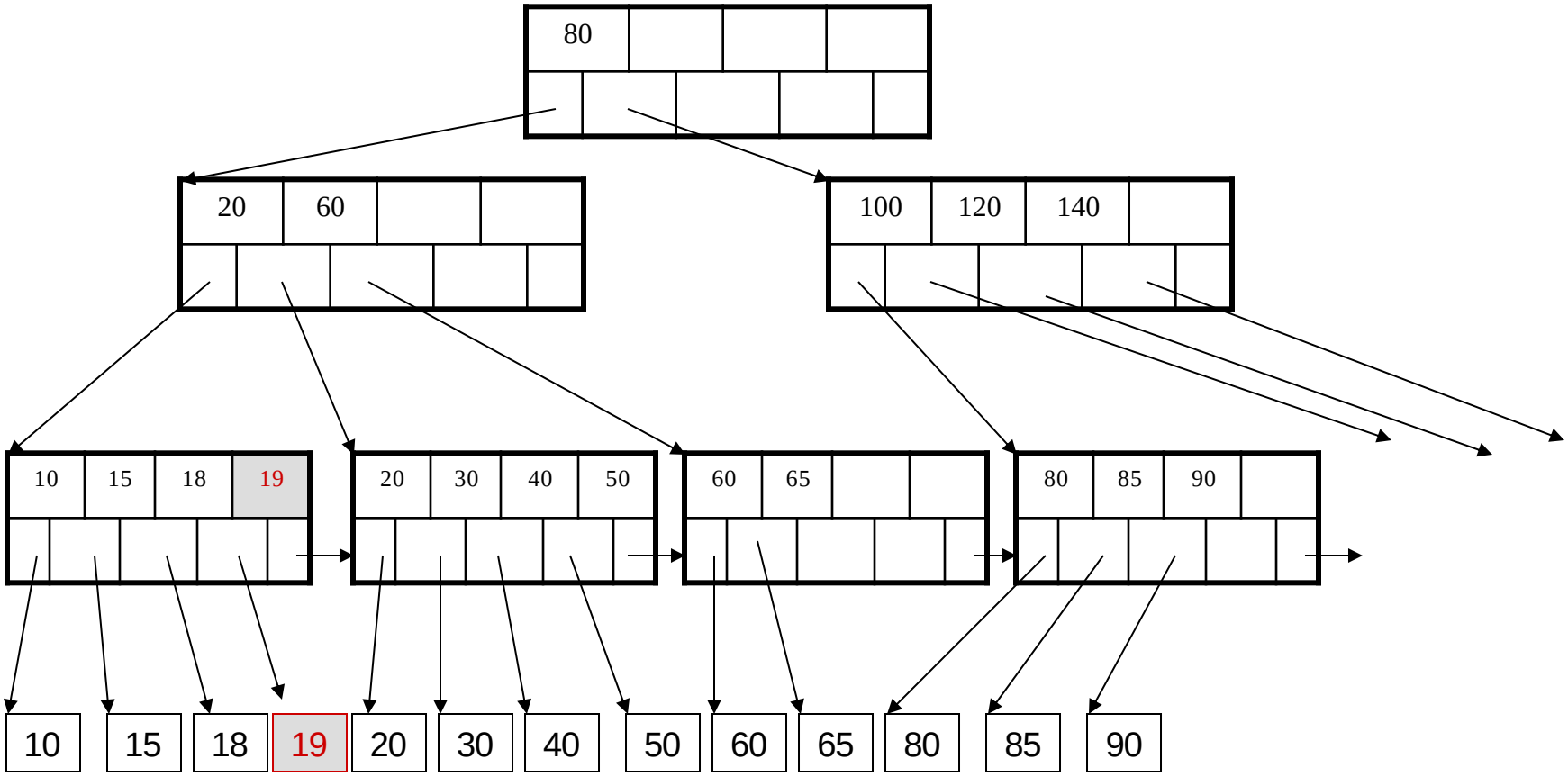
Insertion in a B+ Tree: Example

Insert K=19



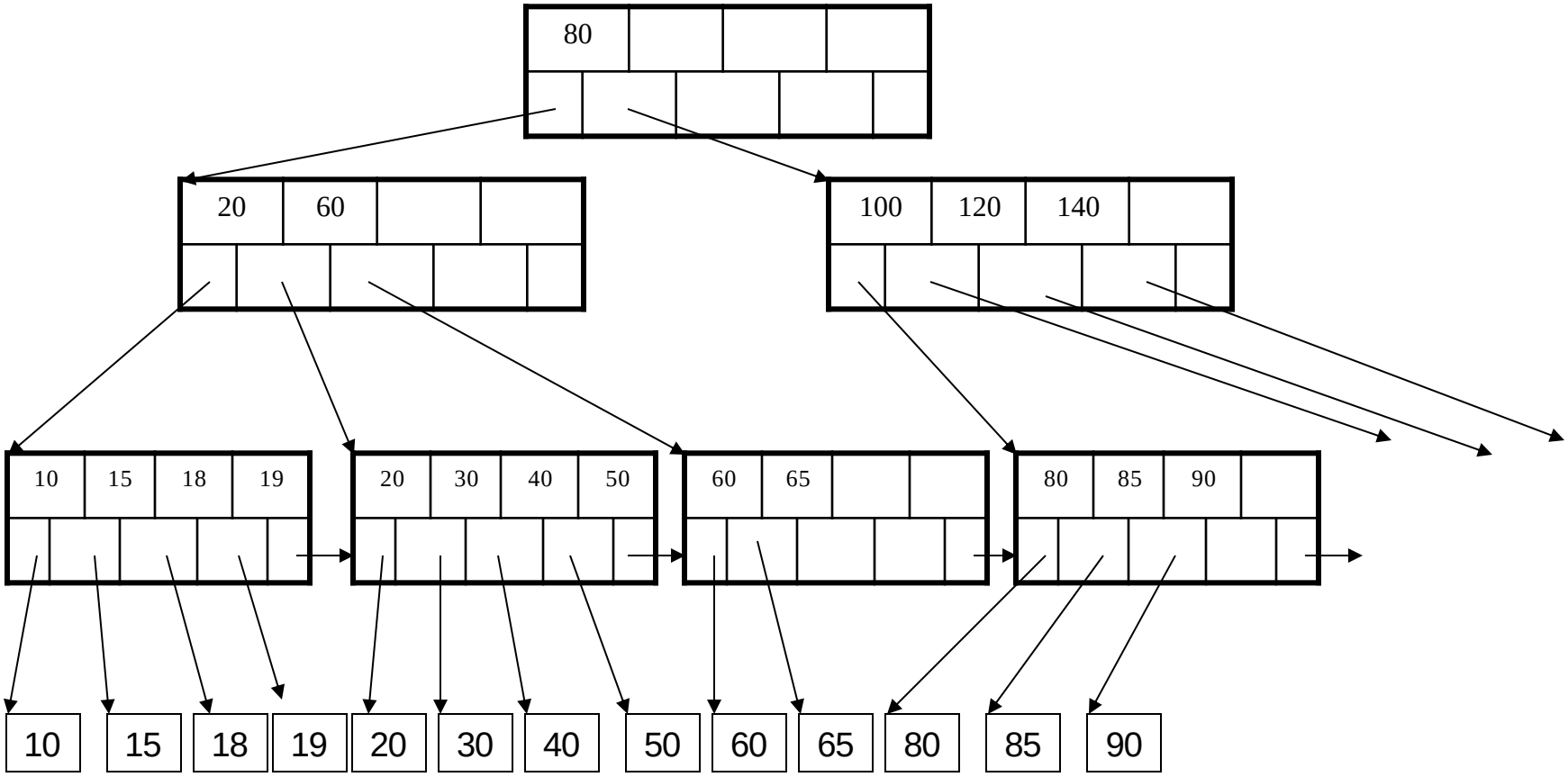
Insertion in a B+ Tree: Example

After insertion



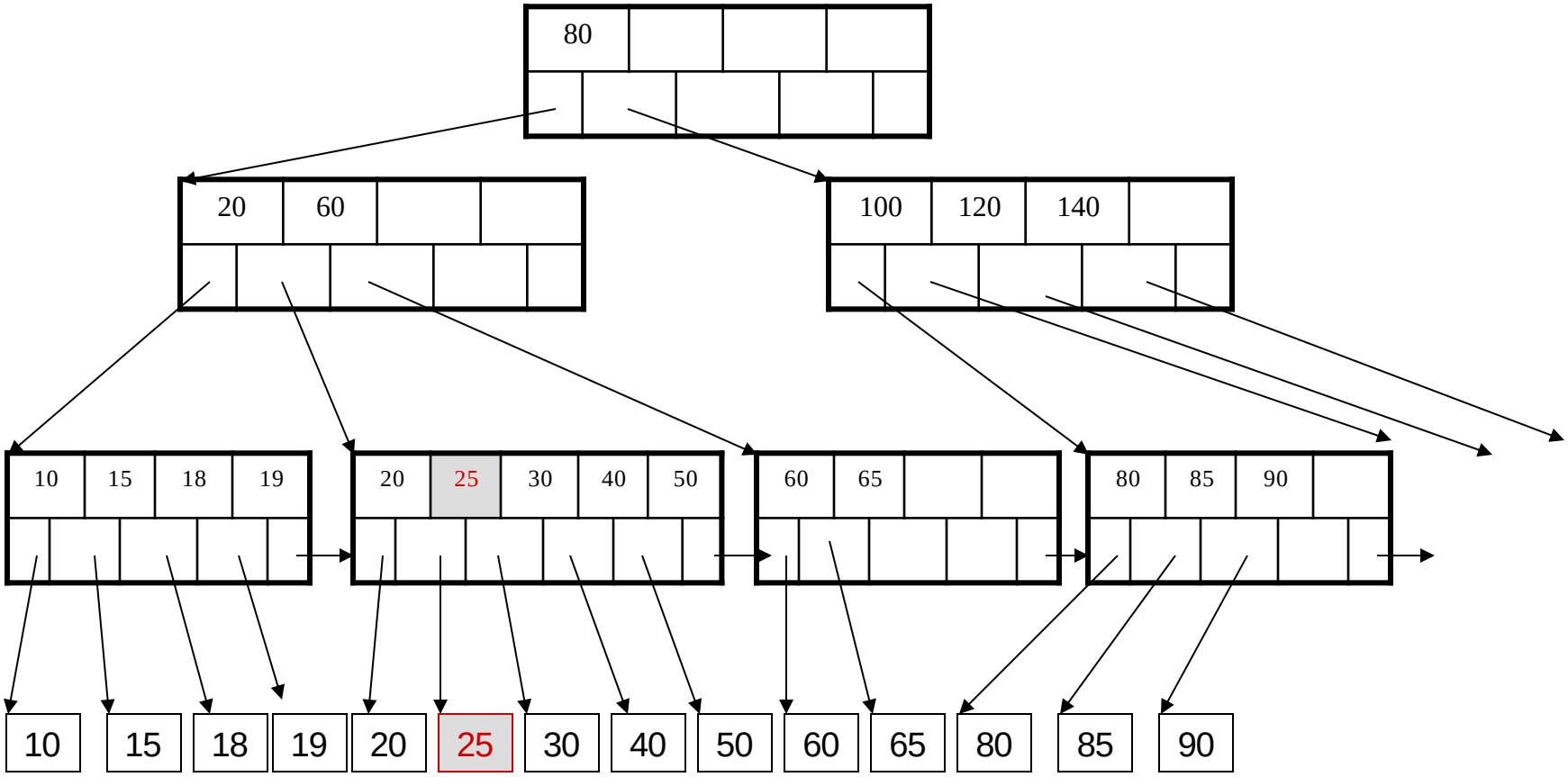
Insertion in a B+ Tree: Example

Now insert 25



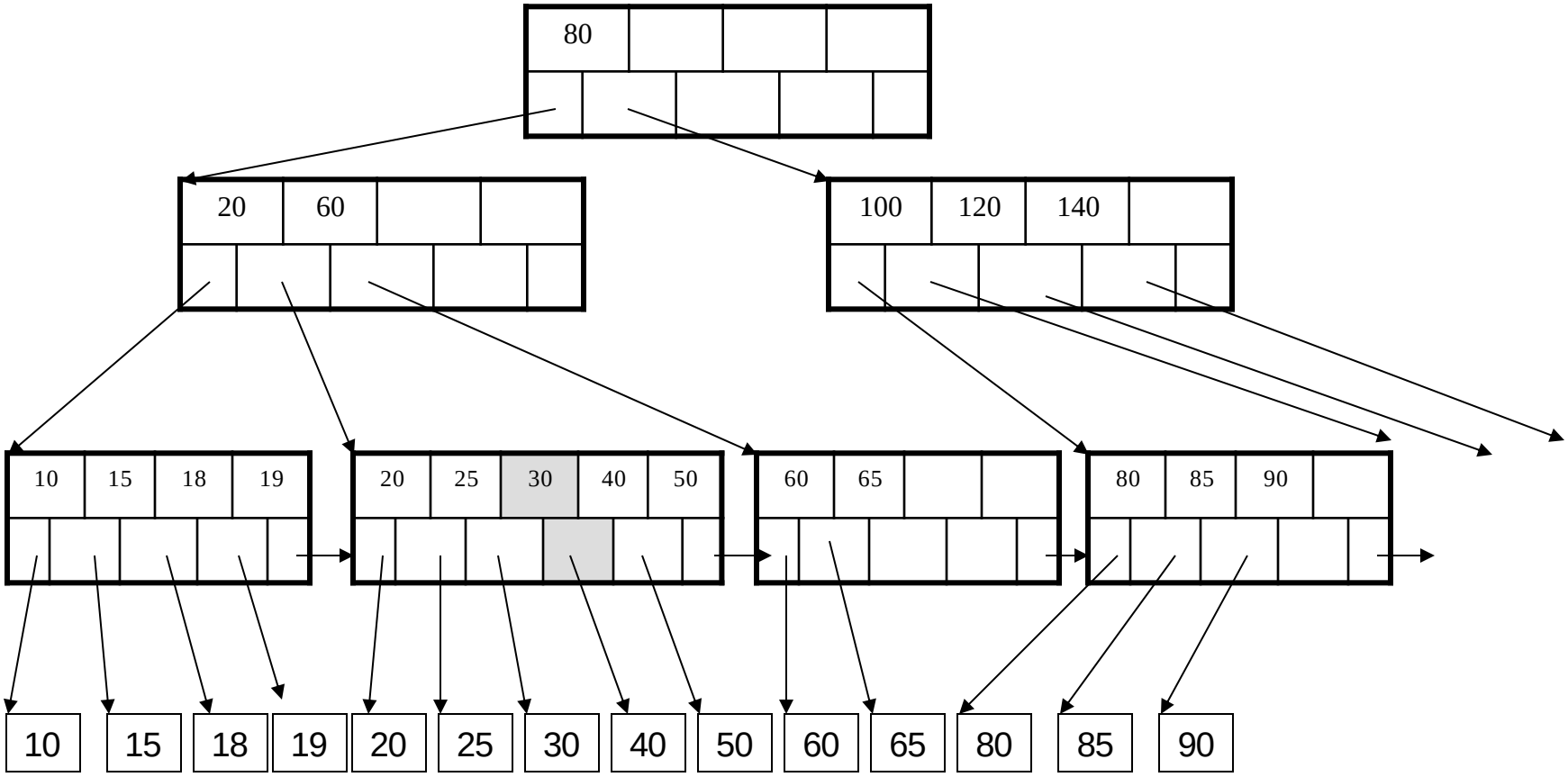
Insertion in a B+ Tree: Example

After insertion



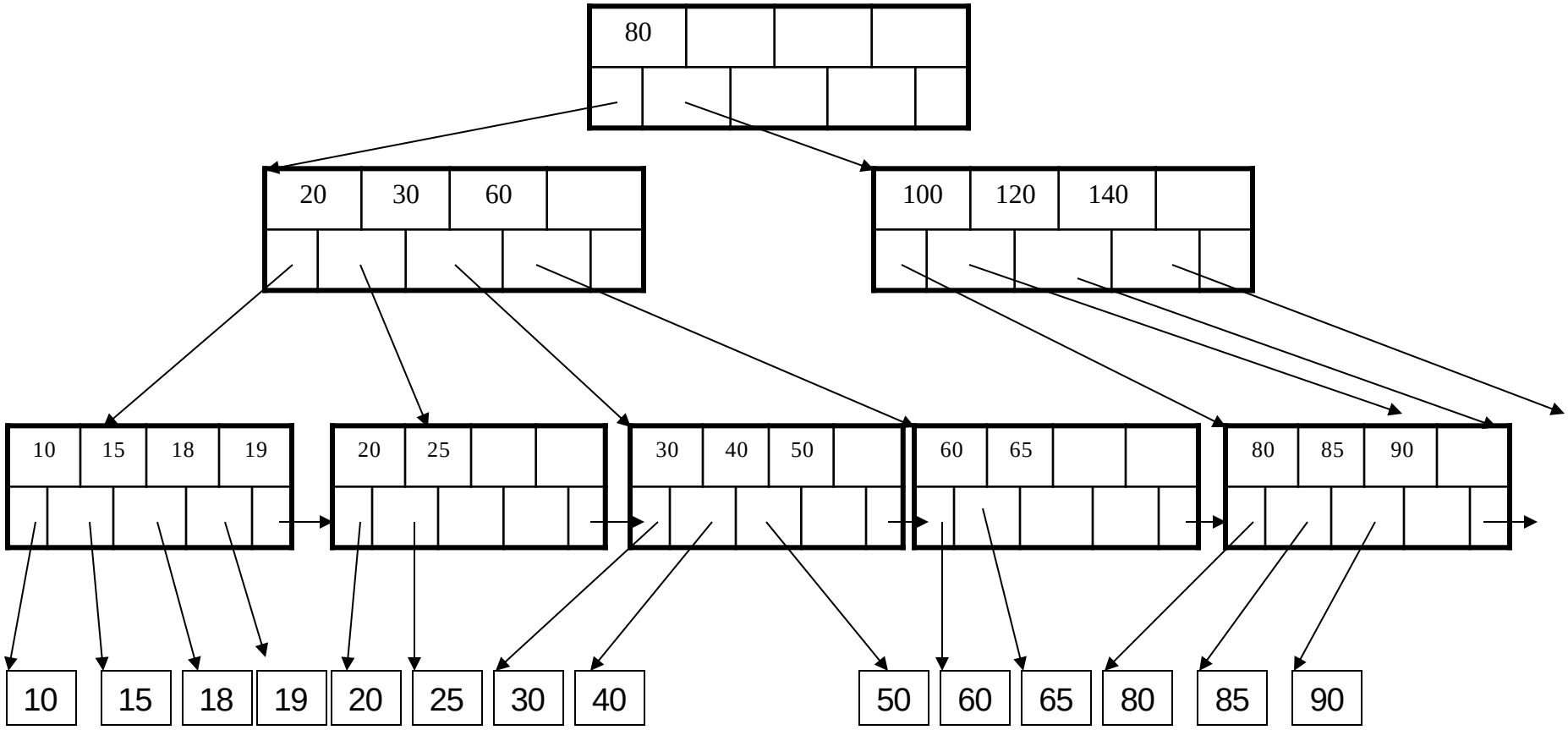
Insertion in a B+ Tree: Example

But now have to split !



Insertion in a B+ Tree: Example

After the split



Deletion: Summary

Delete (K, P)

- Delete K, P from the leaf
- If capacity ≥ min: **Stop**

Deletion: Summary

Delete (K, P)

- Delete K, P from the leaf
- If capacity ≥ min: **Stop**
- If neighbor capacity > min: rotate and **Stop**

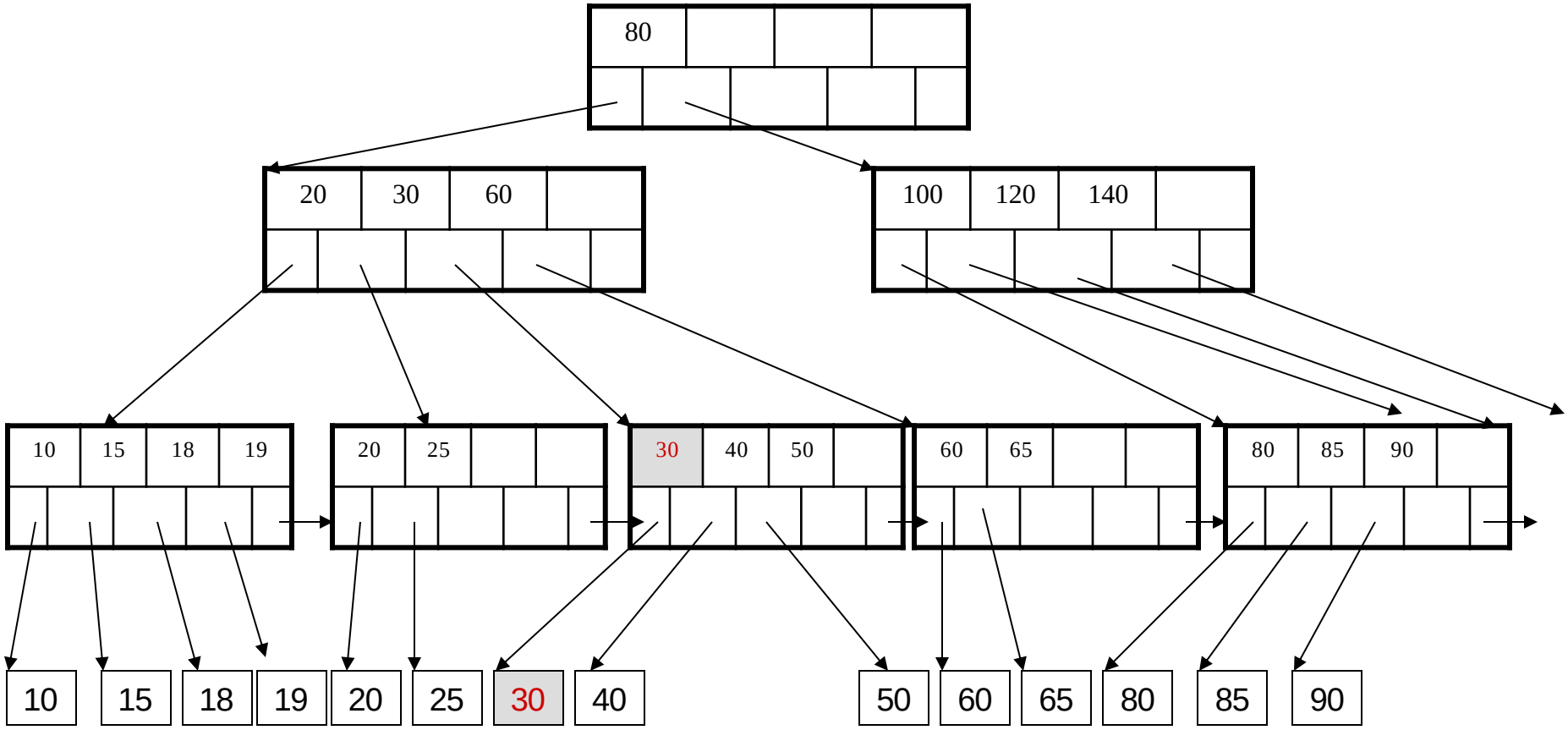
Deletion: Summary

Delete (K, P)

- Delete K, P from the leaf
- If capacity ≥ min: **Stop**
- If neighbor capacity > min: rotate and **Stop**
- Merge with a neighbor P' (choose right or left) and steal a key K' from parent
- Delete (K', P') where P'=the parent

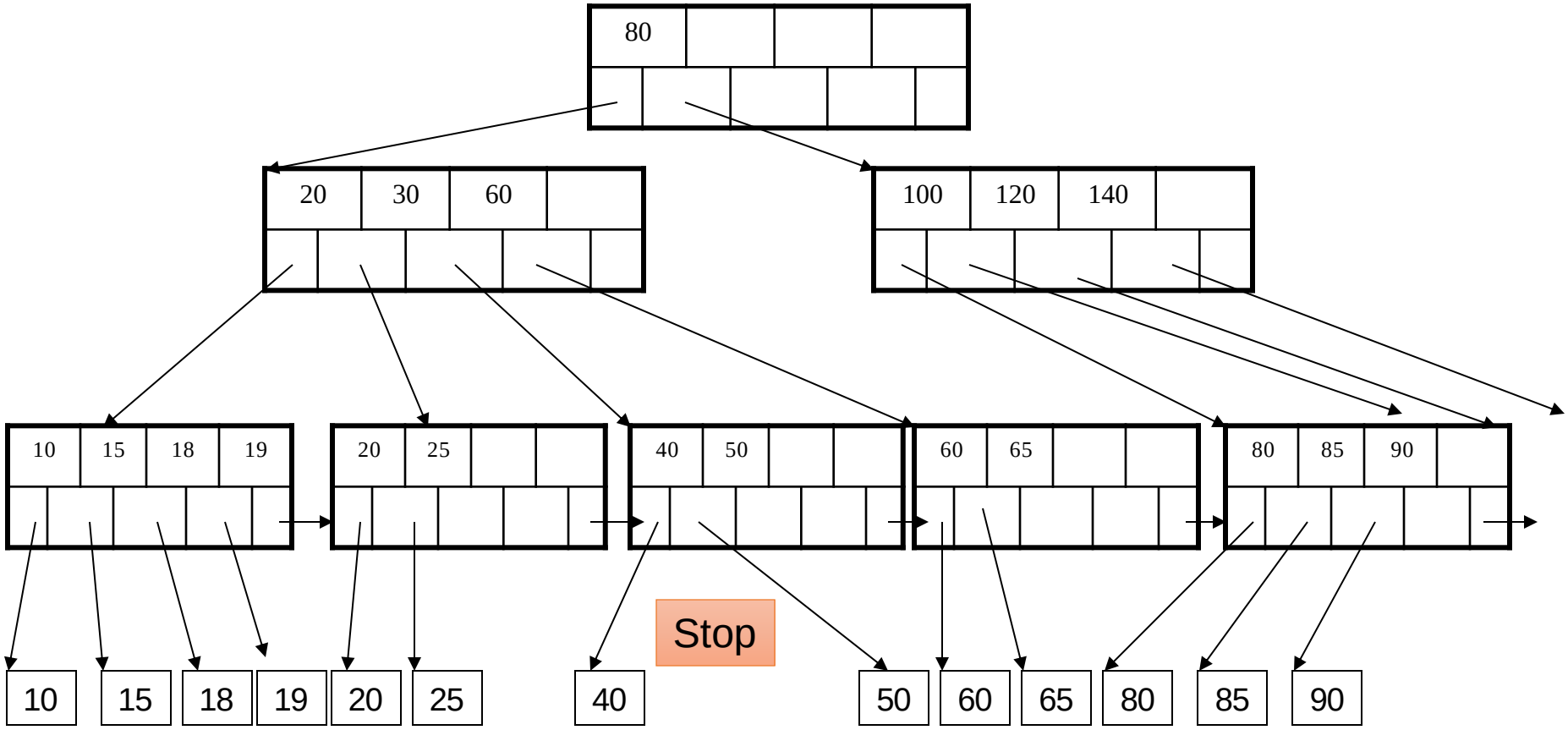
Deletion from a B+ Tree: Example

Delete 30



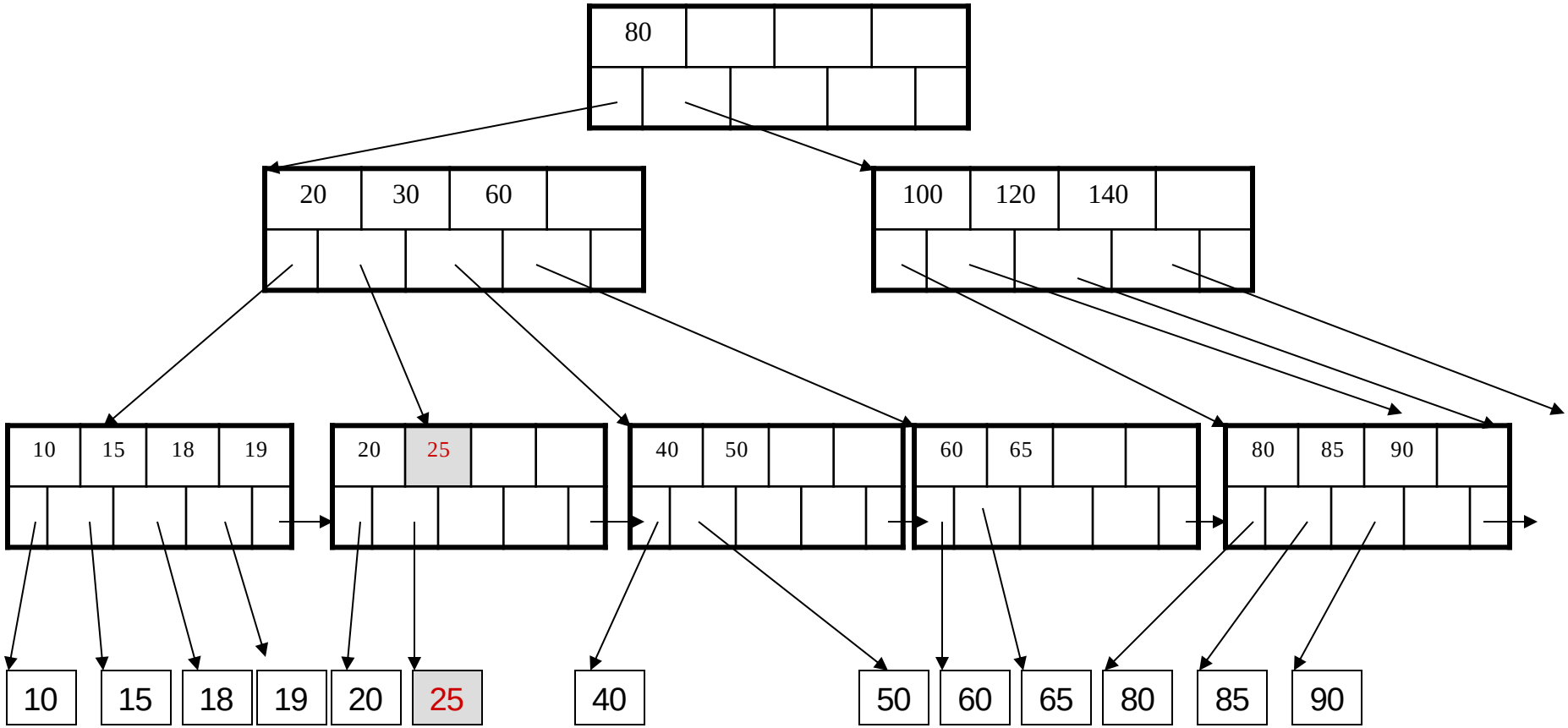
Deletion from a B+ Tree: Example

After deleting 30



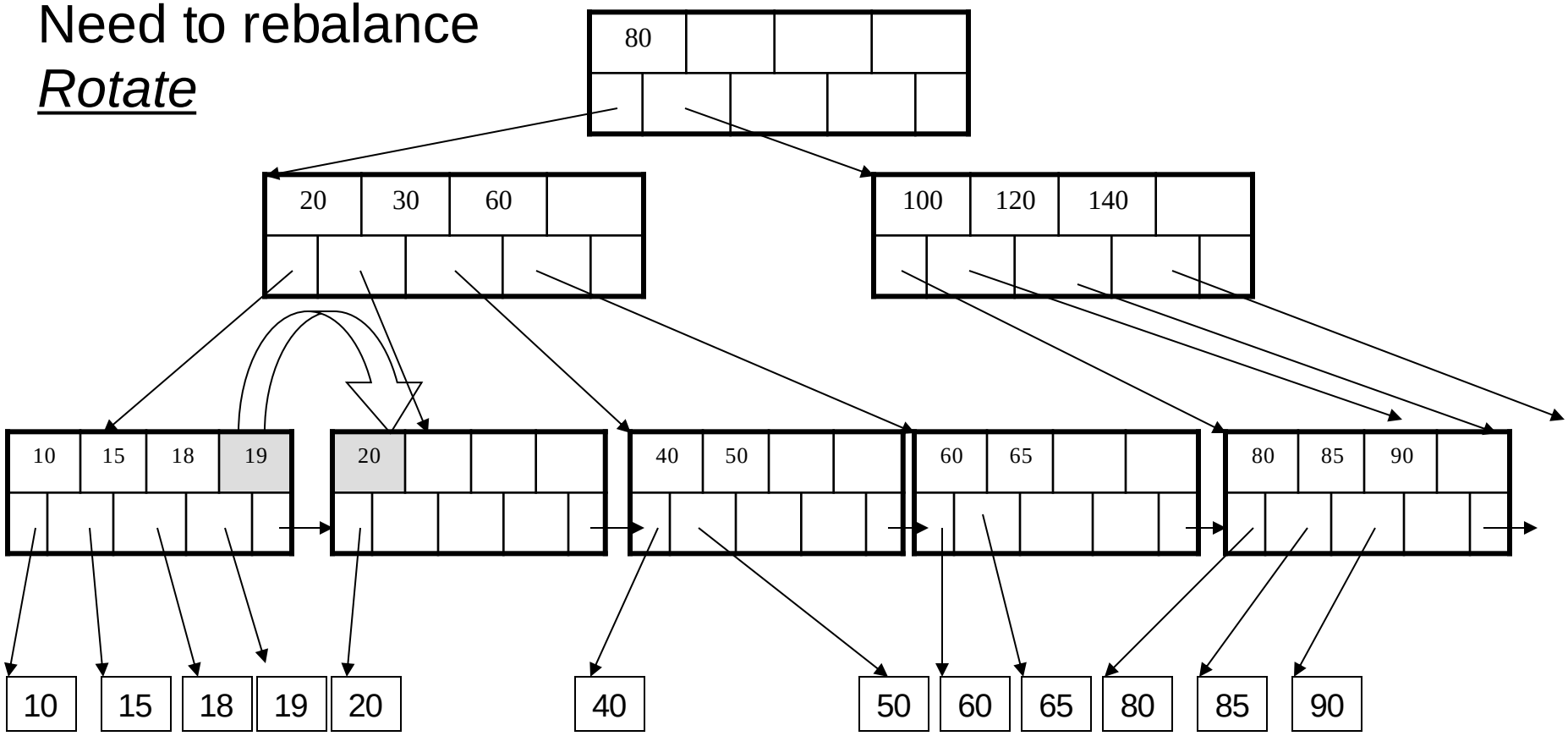
Deletion from a B+ Tree: Example

Now delete 25

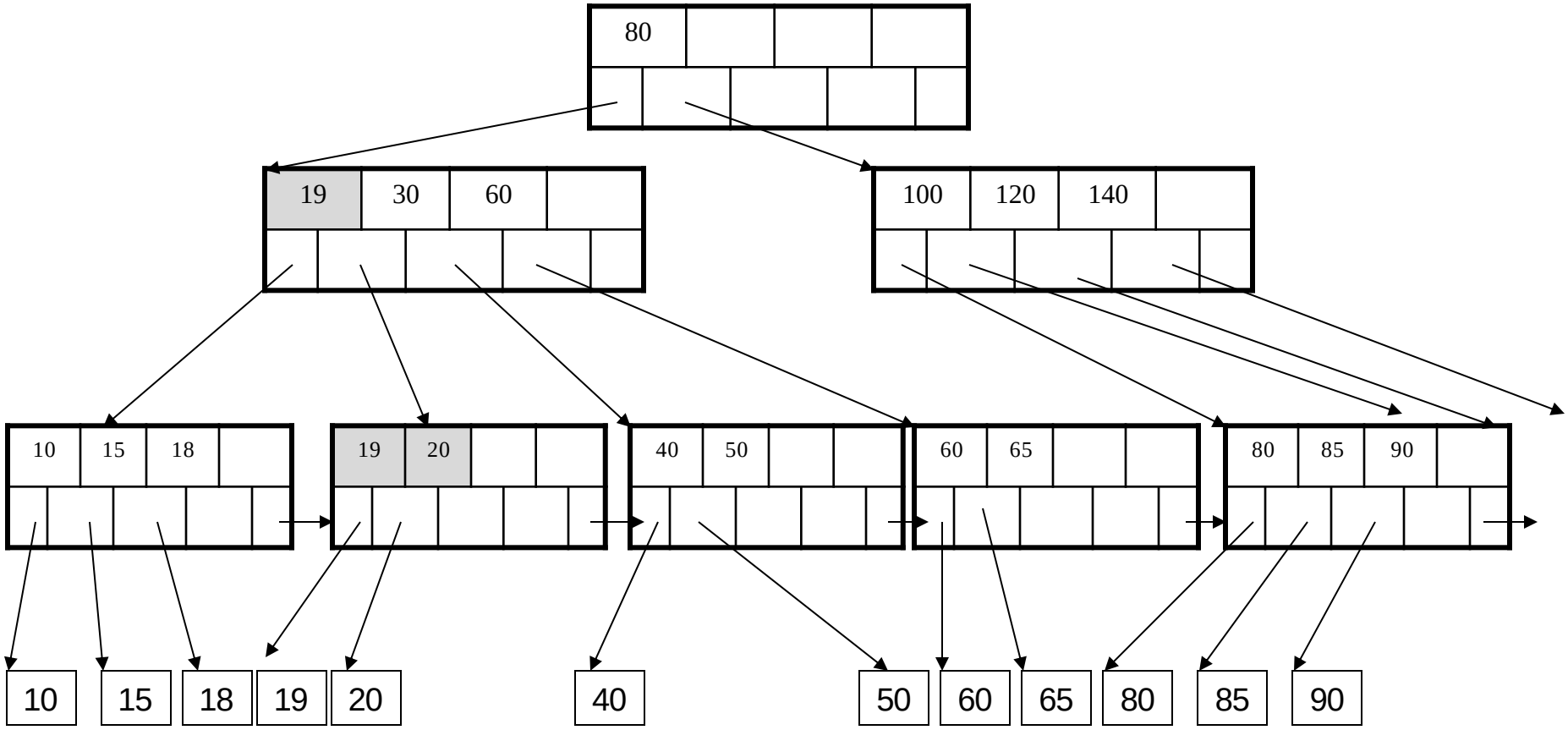


Deletion from a B+ Tree: Example

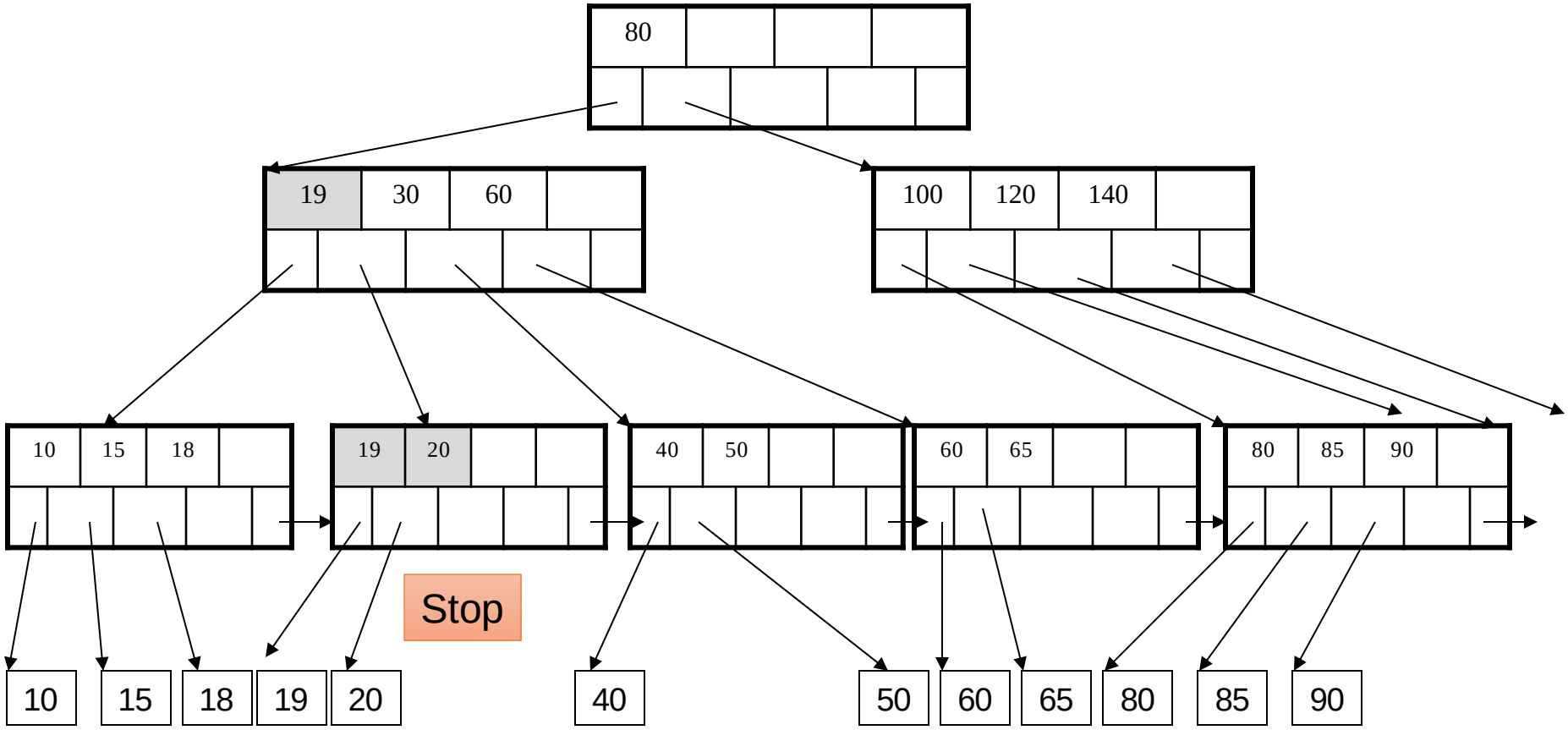
After deleting 25
Need to rebalance
Rotate



Deletion from a B+ Tree: Example

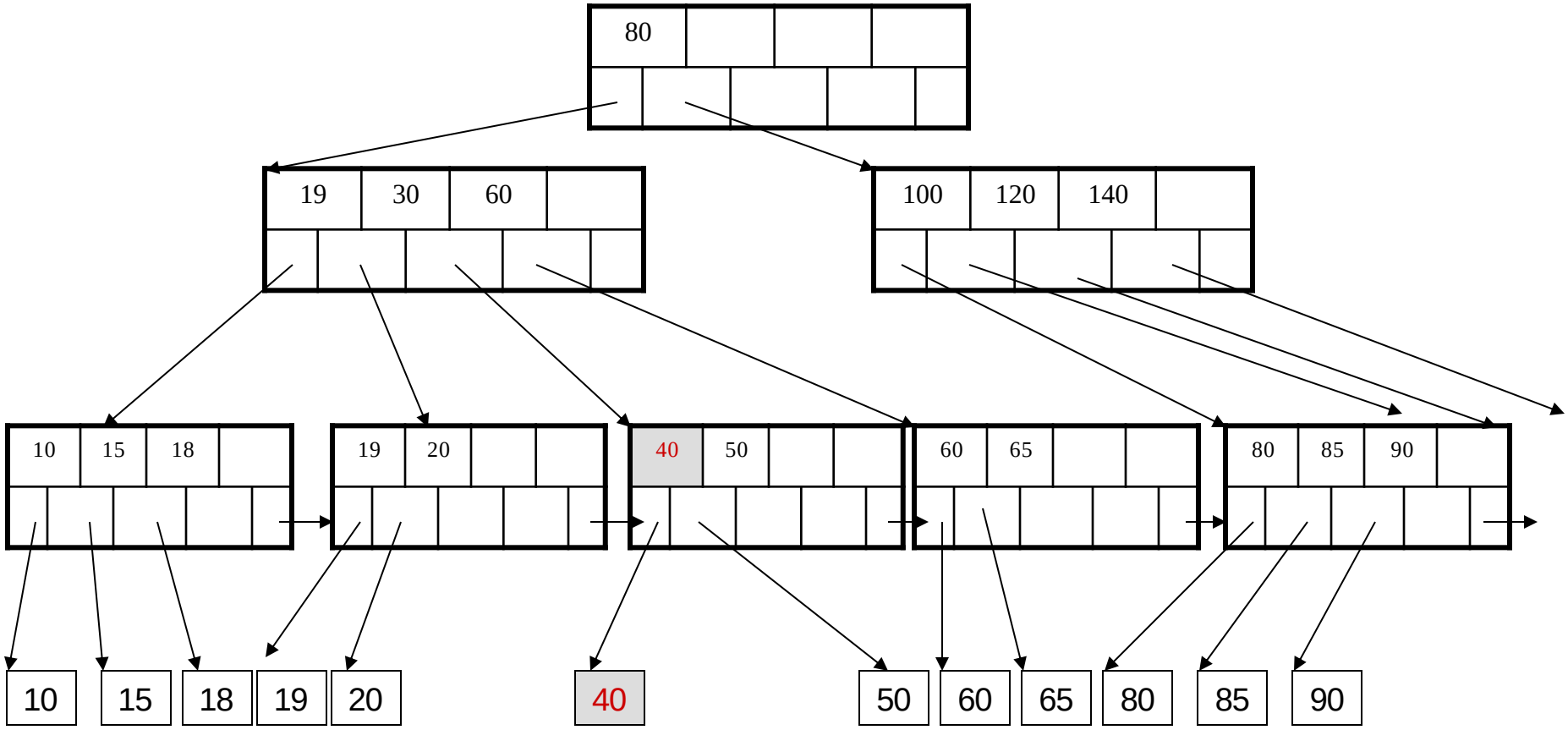


Deletion from a B+ Tree: Example



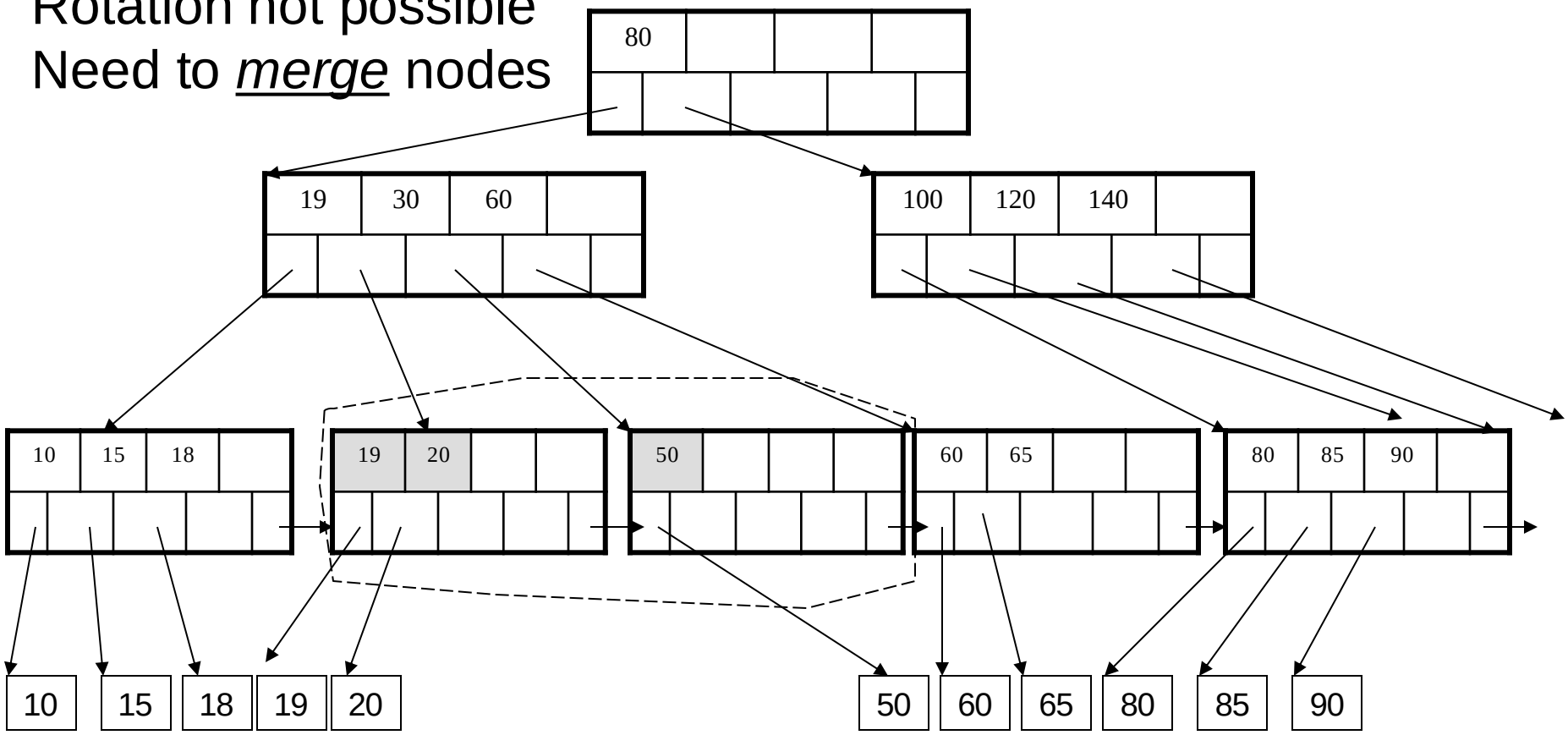
Deletion from a B+ Tree: Example

Now delete 40



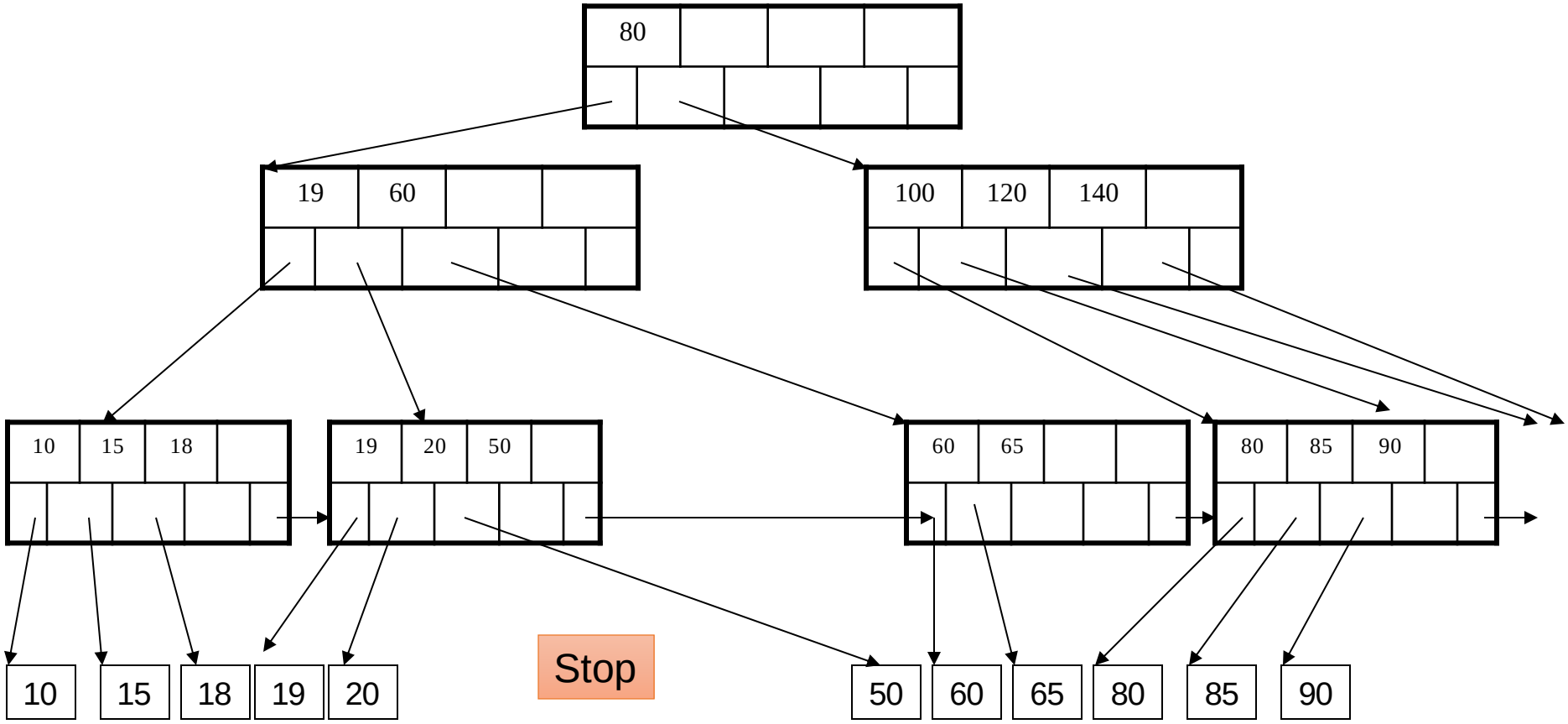
Deletion from a B+ Tree: Example

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



Deletion from a B+ Tree: Example

Final tree



Sequential v.s. Random Access

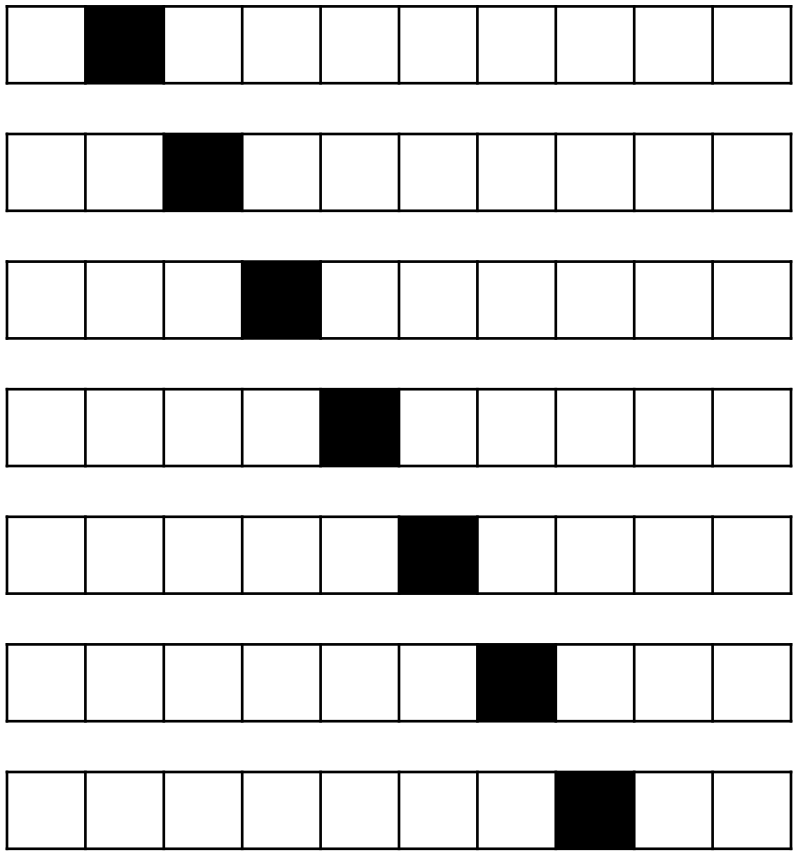
Sequential/Random Access

Recall: the disk always reads one entire block

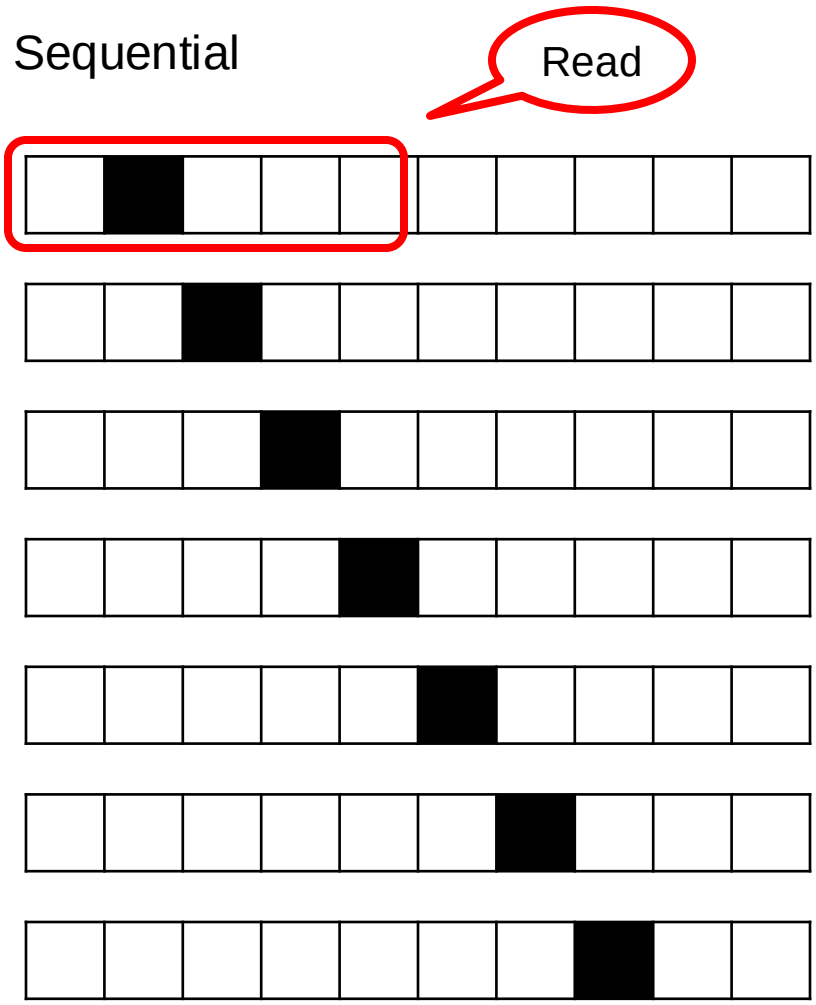
- Sequential access: we read consecutive blocks
  1001, 1002, 1003, 1004, …
- Random access: we read them in whatever order
  1523, 1003, 2999,1010, …

Sequential/Random Access

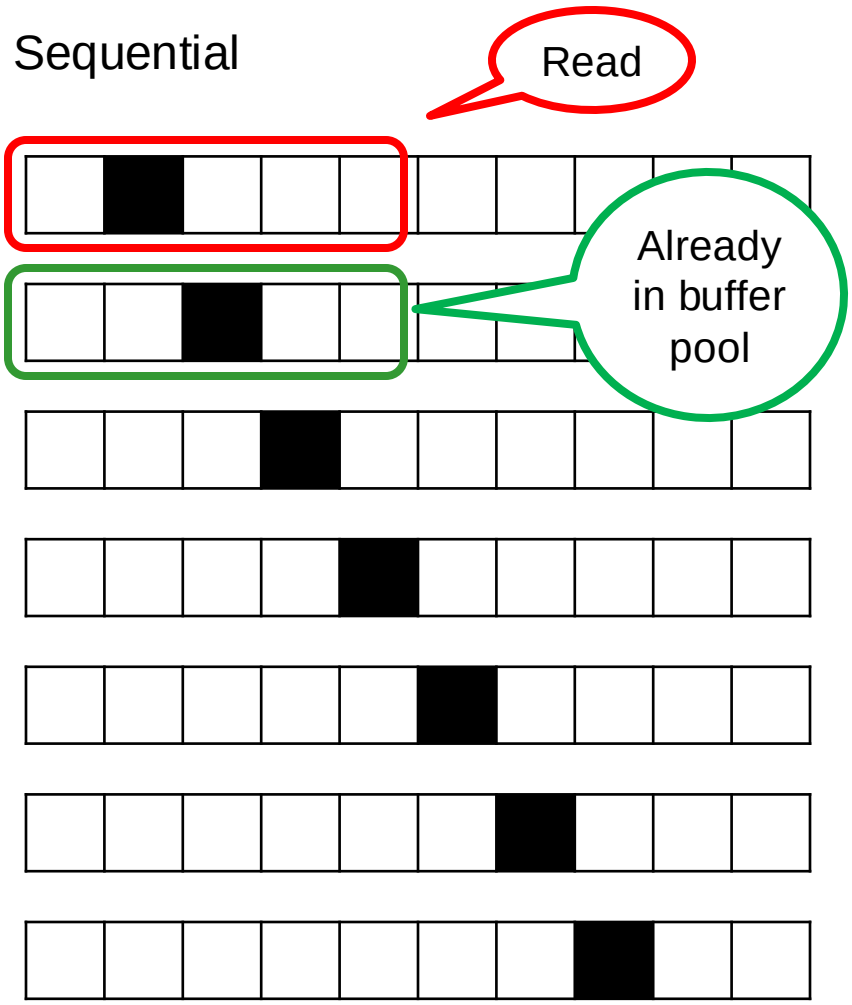
Sequential



Sequential/Random Access

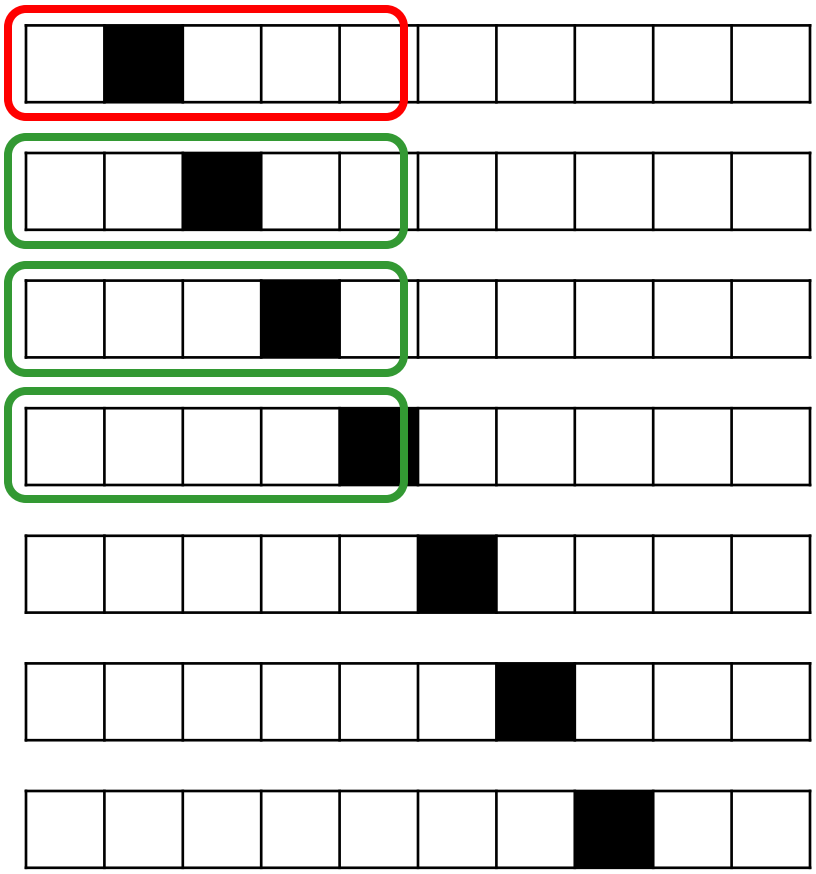


Sequential/Random Access



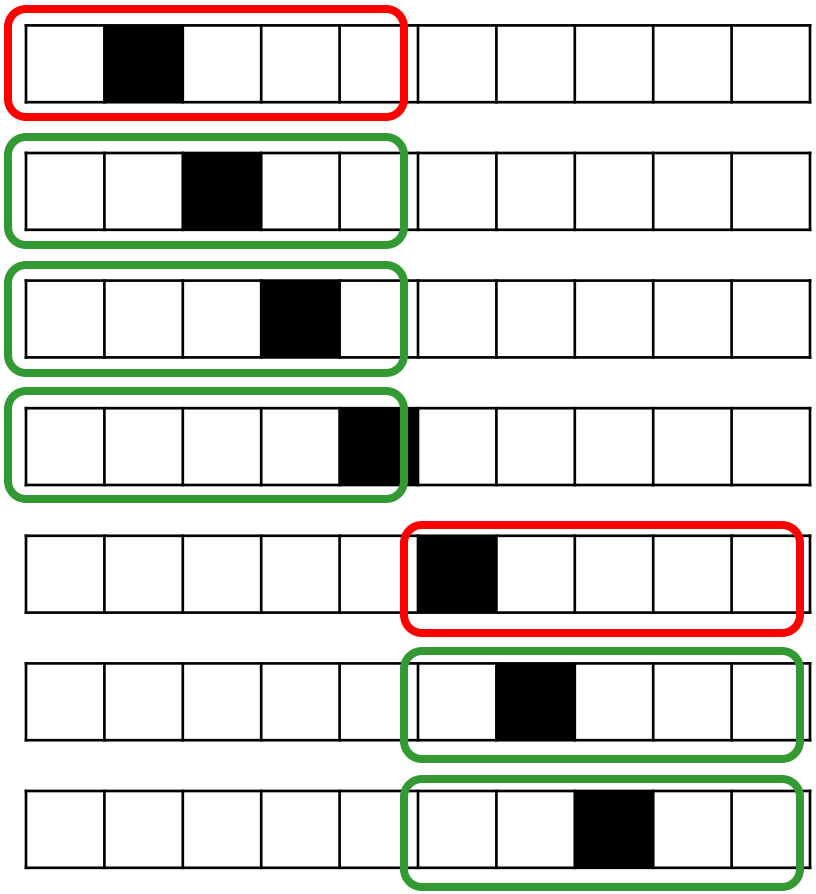
Sequential/Random Access

Sequential



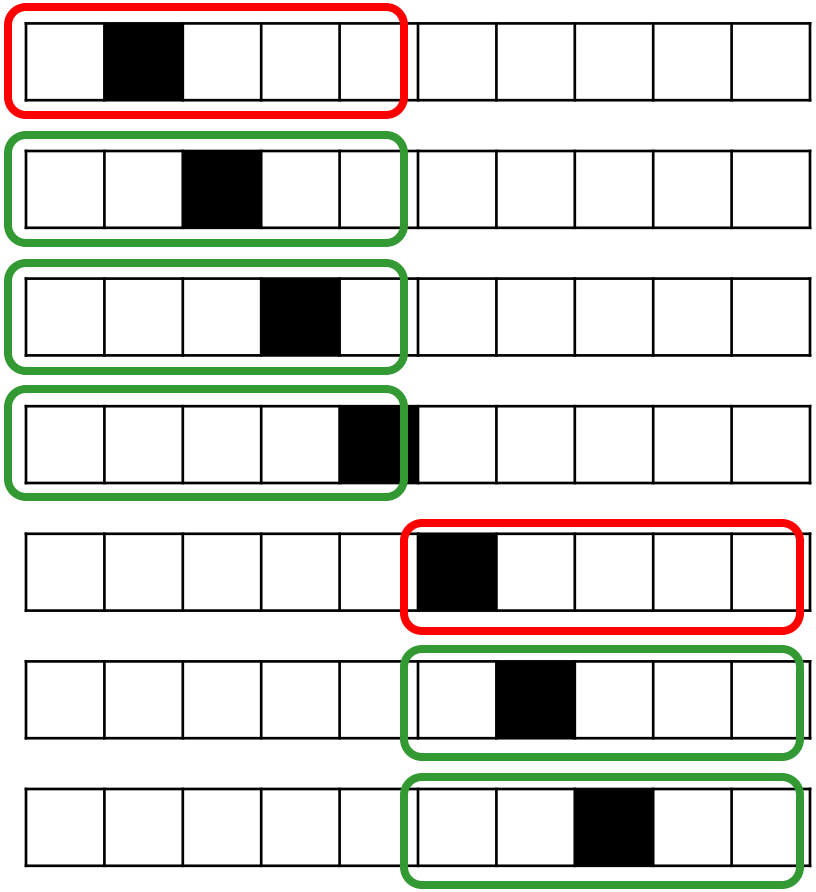
Sequential/Random Access

Sequential: 2 reads

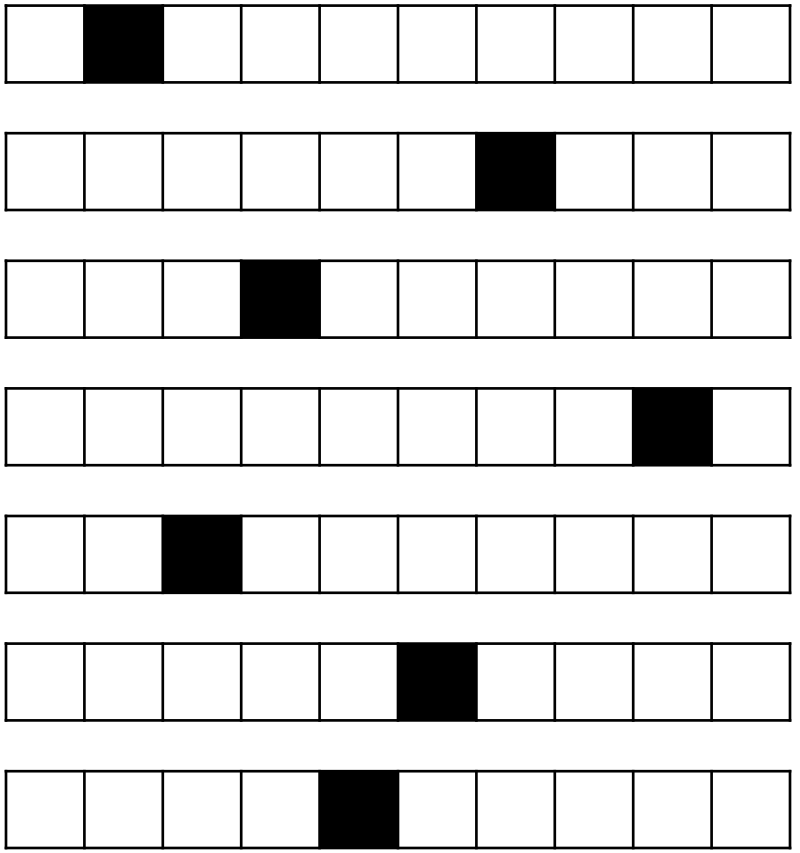


Sequential/Random Access

Sequential: 2 reads

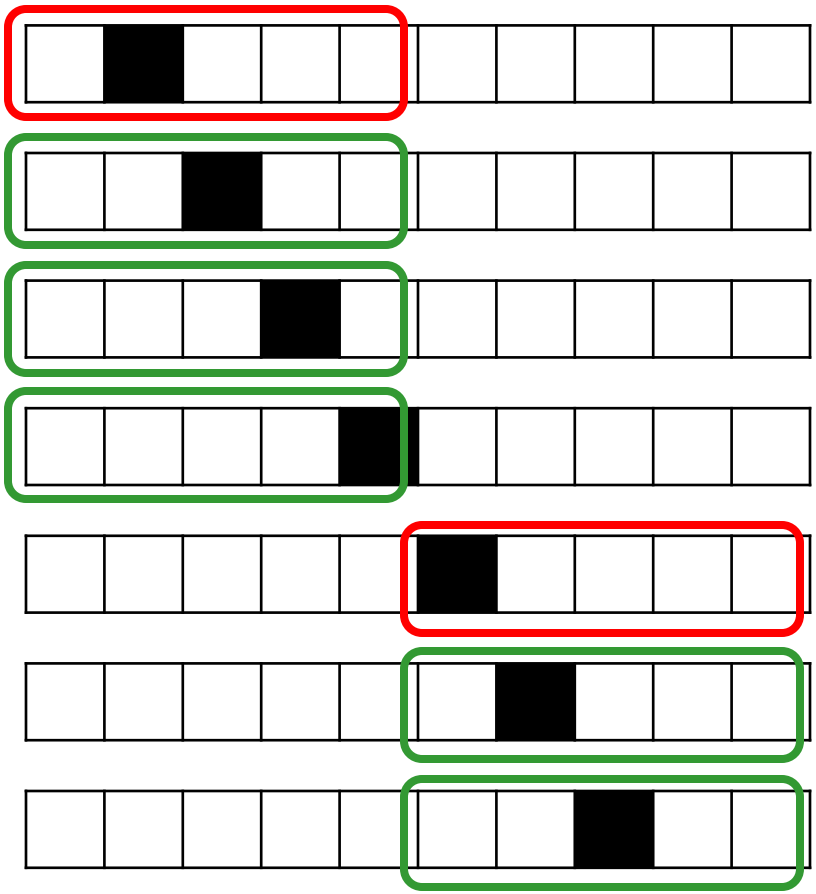


Random

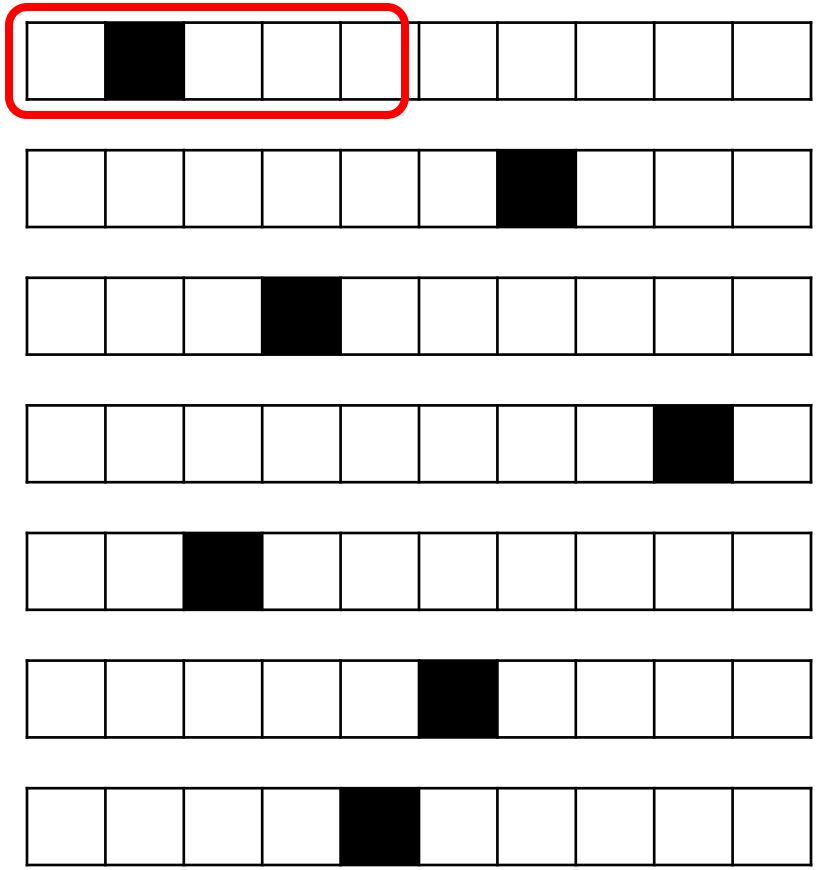


Sequential/Random Access

Sequential: 2 reads

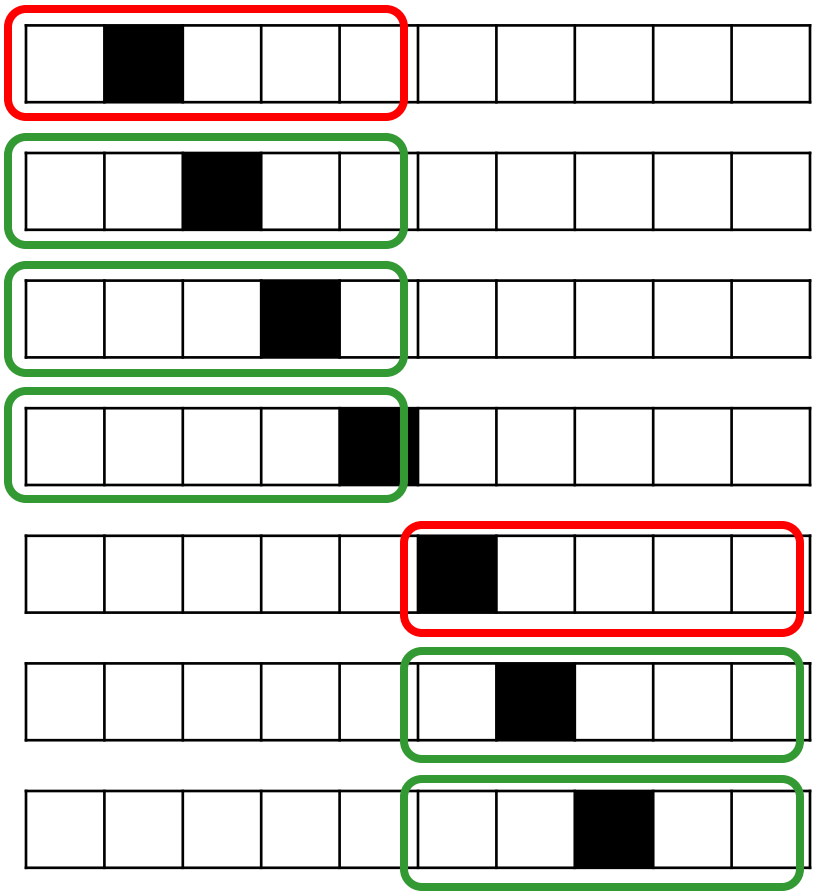


Random

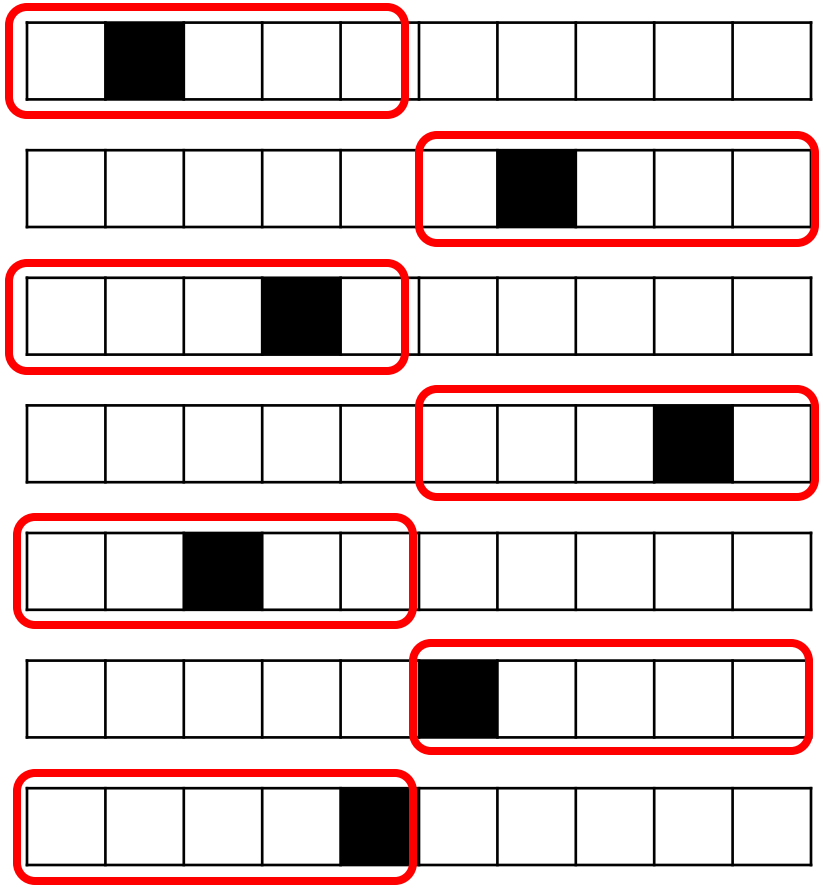


Sequential/Random Access

Sequential: 2 reads



Random: many reads



Discussion

Sequential scan:

- Needs to read all blocks
- But uses sequential access to disk

Index lookup:

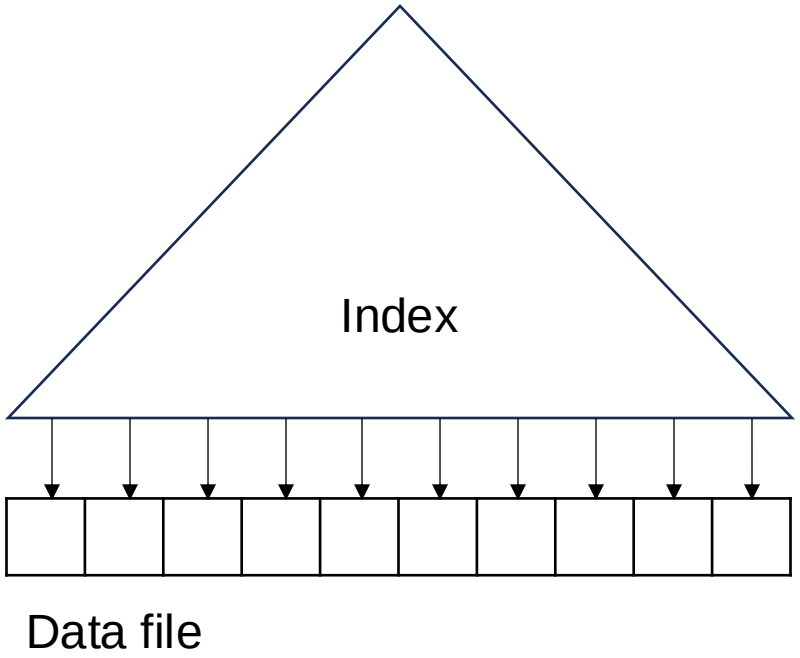
- Reads far fewer blocks (sometimes just 1)
- But it uses random access

Clustered v.s. Unclustered Indexes

- An index is clustered if the data file is sorted in by the index key
- Otherwise, it is called unclusterd

Clustered v.s. Unclustered Indexes

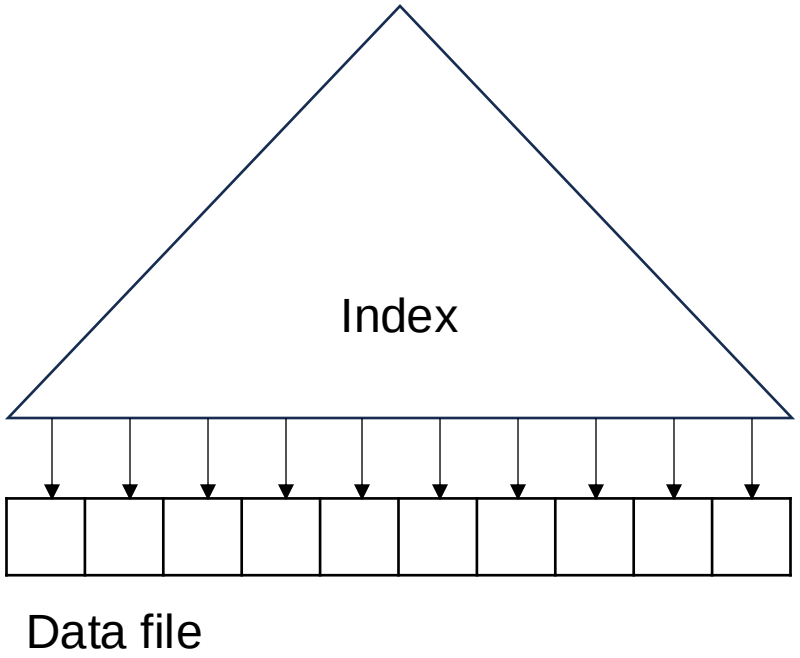
- An index is clustered if the data file is sorted in by the index key
- Otherwise, it is called unclusterd



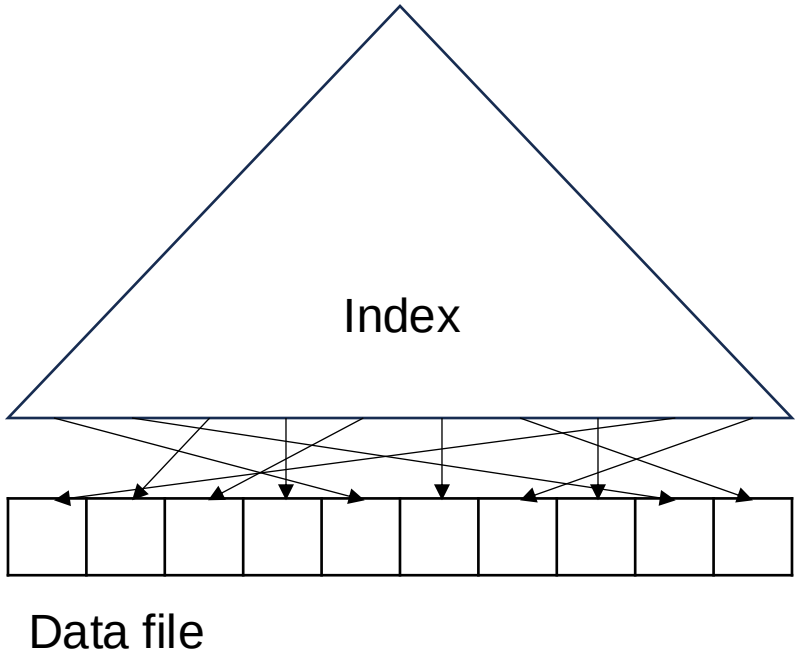
Clustered

Clustered v.s. Unclustered Indexes

- An index is clustered if the data file is sorted in by the index key
- Otherwise, it is called unclusterd



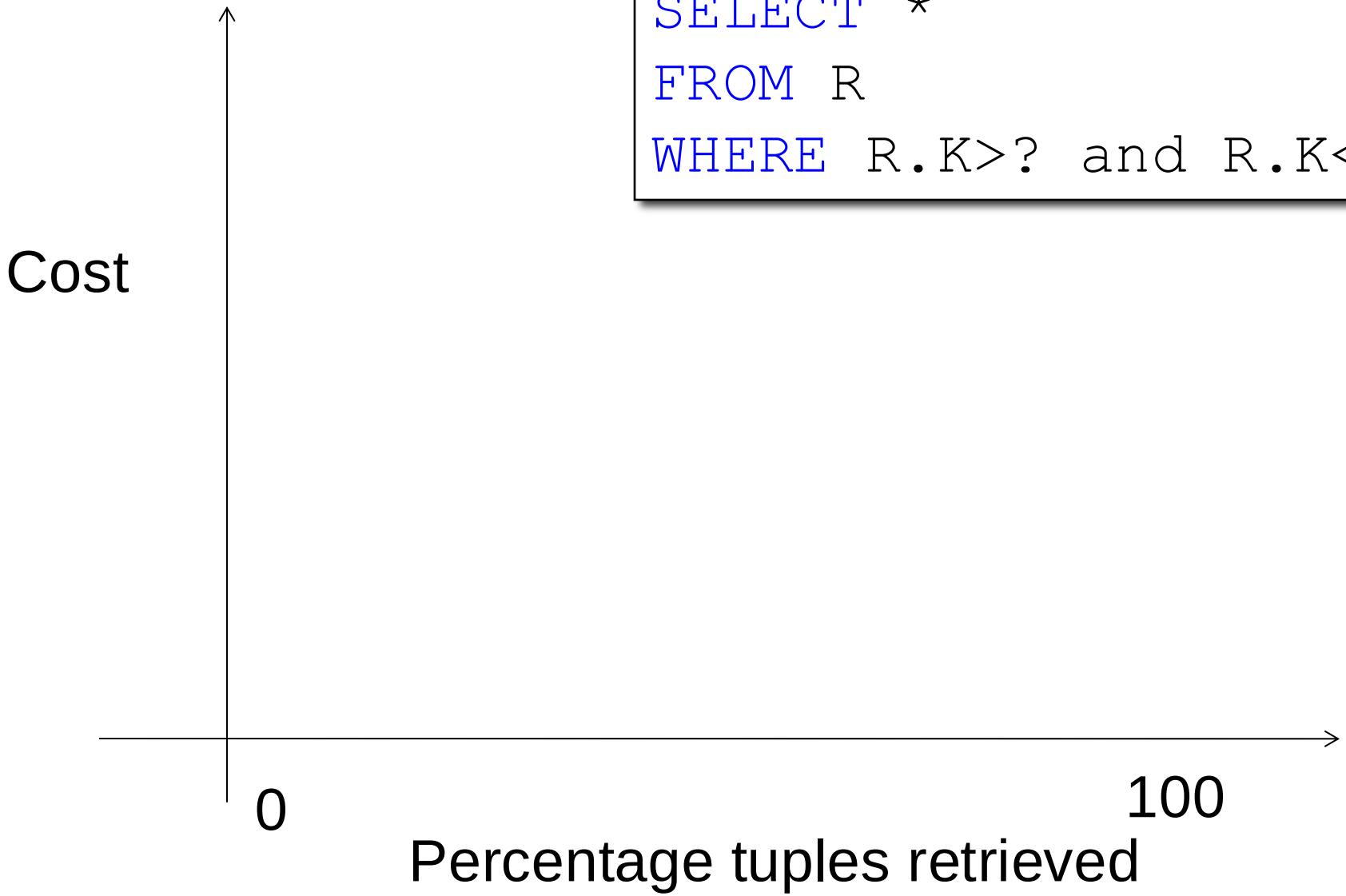
Clustered



Unclustered

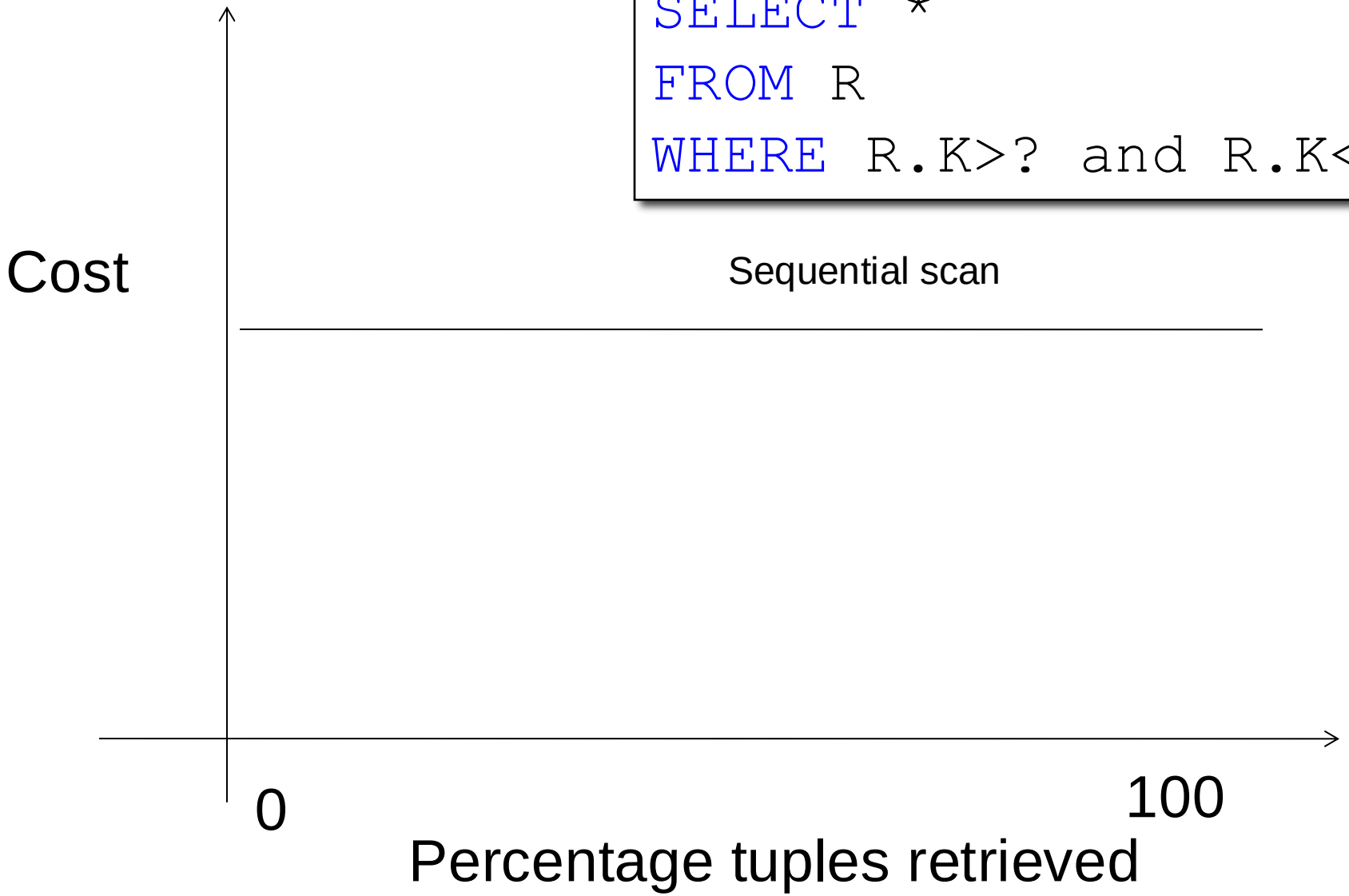
Clustered v.s. Unclustered Indexes

```
SELECT *
FROM R
WHERE R.K>? and R.K<?
```



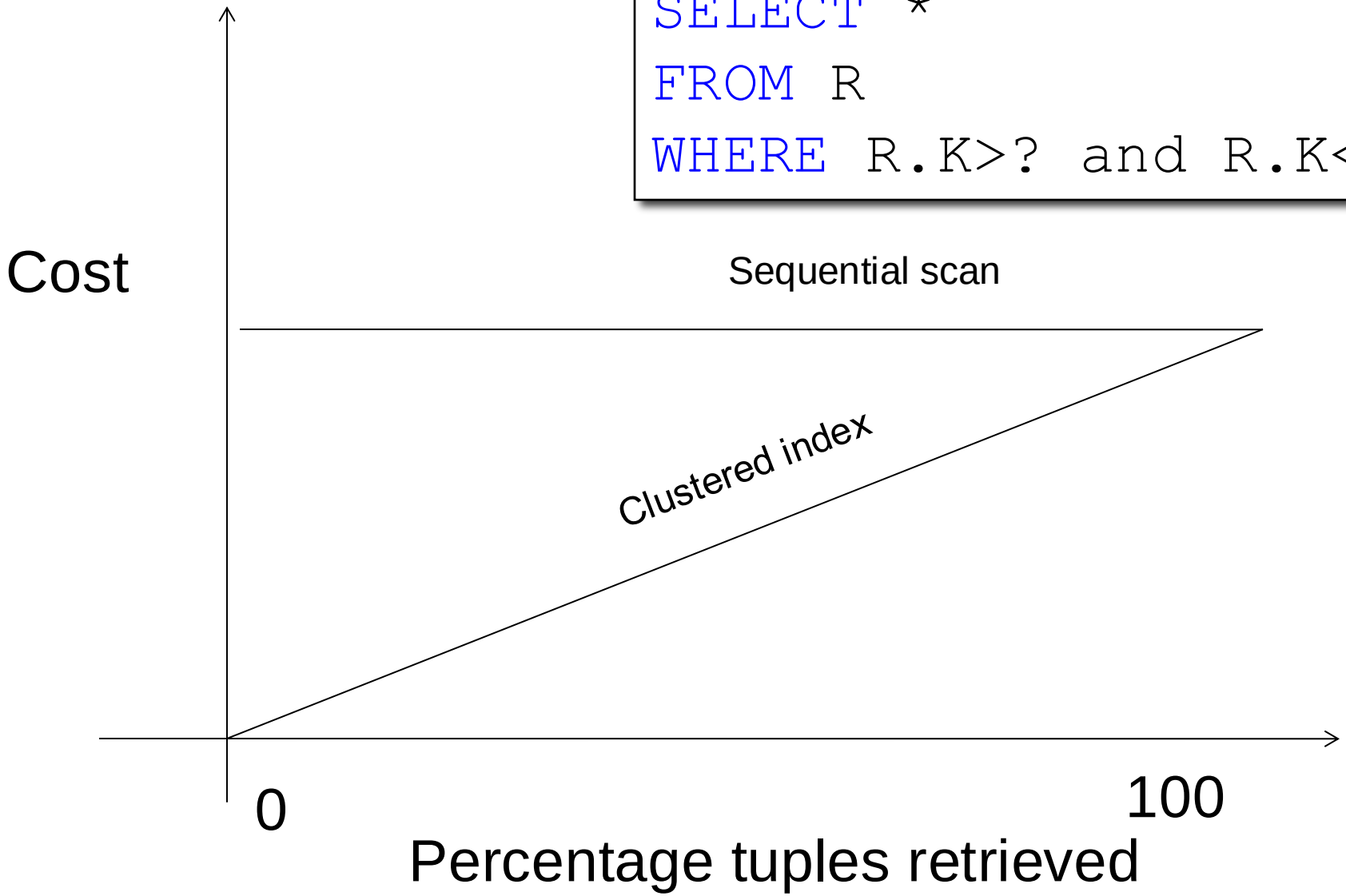
Clustered v.s. Unclustered Indexes

```
SELECT *
FROM R
WHERE R.K>? and R.K<?
```



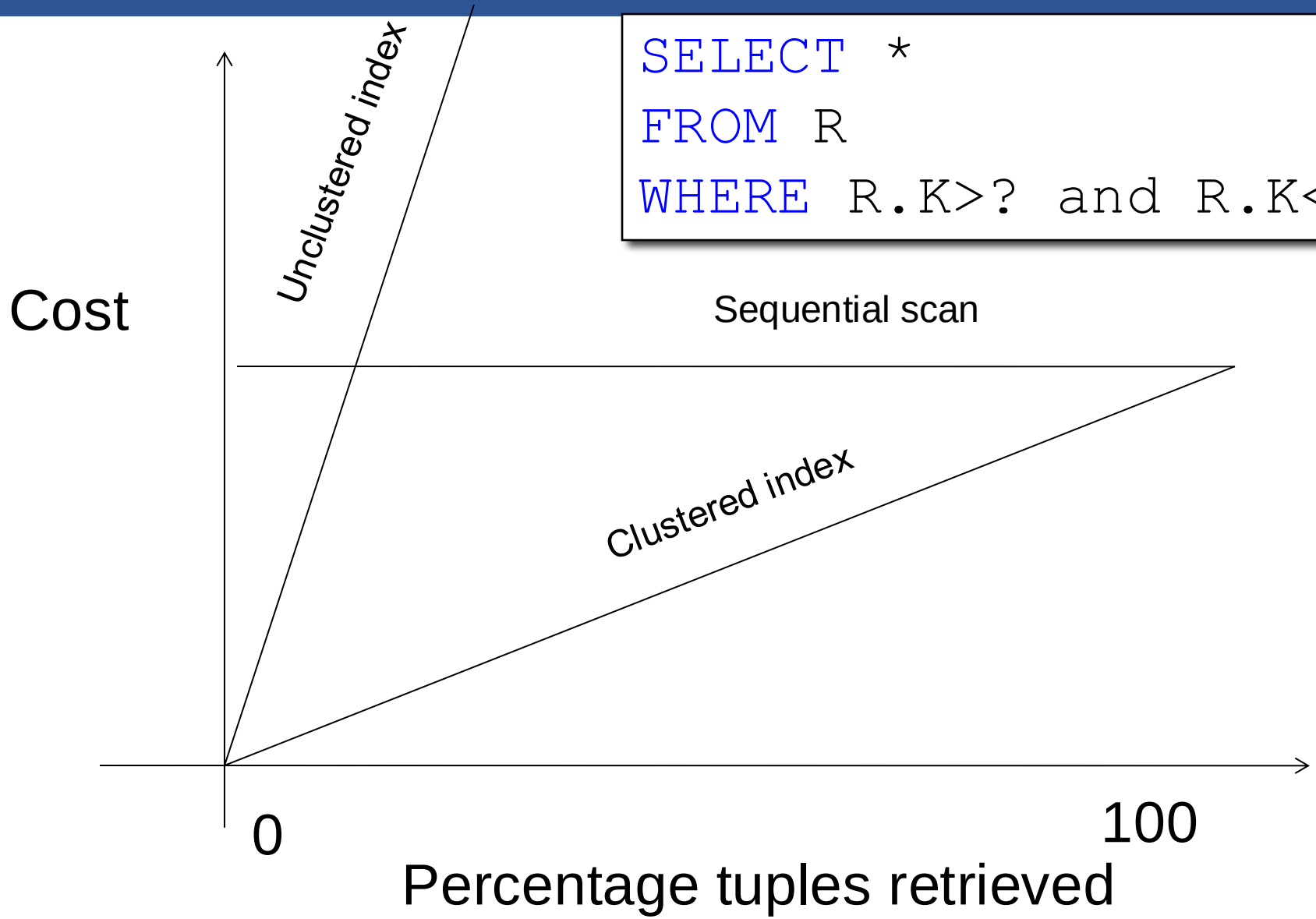
Clustered v.s. Unclustered Indexes

```
SELECT *
FROM R
WHERE R.K>? and R.K<?
```



Clustered v.s. Unclustered Indexes

```
SELECT *
FROM R
WHERE R.K>? and R.K<?
```



To Cluster or Not

- Rule of thumb:

  Random reading 1-2% of file ≈ full sequential scan

- Range queries benefit mostly from clustering because they may read more than 1-2%

Indexes in SQL

Getting Practical: Creating Indexes in SQL

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

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

Getting Practical: Creating Indexes in SQL

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

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

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

Getting Practical: Creating Indexes in SQL

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

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

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

What does this mean?

Getting Practical: Creating Indexes in SQL

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

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

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

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

Yes

What does this mean?

Getting Practical: Creating Indexes in SQL

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

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

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

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

Yes

What does this mean?

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

Yes

Getting Practical: Creating Indexes in SQL

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

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

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

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

Yes

What does this mean?

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

Yes

```
select *
from V
where M=77
```

No

Getting Practical: Creating Indexes in SQL

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

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

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

```
CREATE INDEX V3 ON V(M, N);
```

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

Yes

What does this mean?

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

Yes

```
select *
from V
where M=77
```

No

Getting Practical: Creating Indexes in SQL

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

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

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

Yes

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

What does this mean?

```
CREATE INDEX V3 ON V(M, N);
```

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

Yes

```
CREATE UNIQUE INDEX V4 ON V(N);
```

```
select *
from V
where M=77
```

No

Getting Practical: Creating Indexes in SQL

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

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

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

Yes

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

What does this mean?

```
CREATE INDEX V3 ON V(M, N);
```

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

Yes

```
CREATE UNIQUE INDEX V4 ON V(N);
```

```
select *
from V
where M=77
```

No

```
CREATE CLUSTERED INDEX V5 ON V(N);
```

Getting Practical: Creating Indexes in SQL

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

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

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

Yes

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

What does this mean?

```
CREATE INDEX V3 ON V(M, N);
```

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

Yes

```
CREATE UNIQUE INDEX V4 ON V(N);
```

```
select *
from V
where M=77
```

No

```
CREATE CLUSTERED INDEX V5 ON V(N);
```

Not supported in SQLite