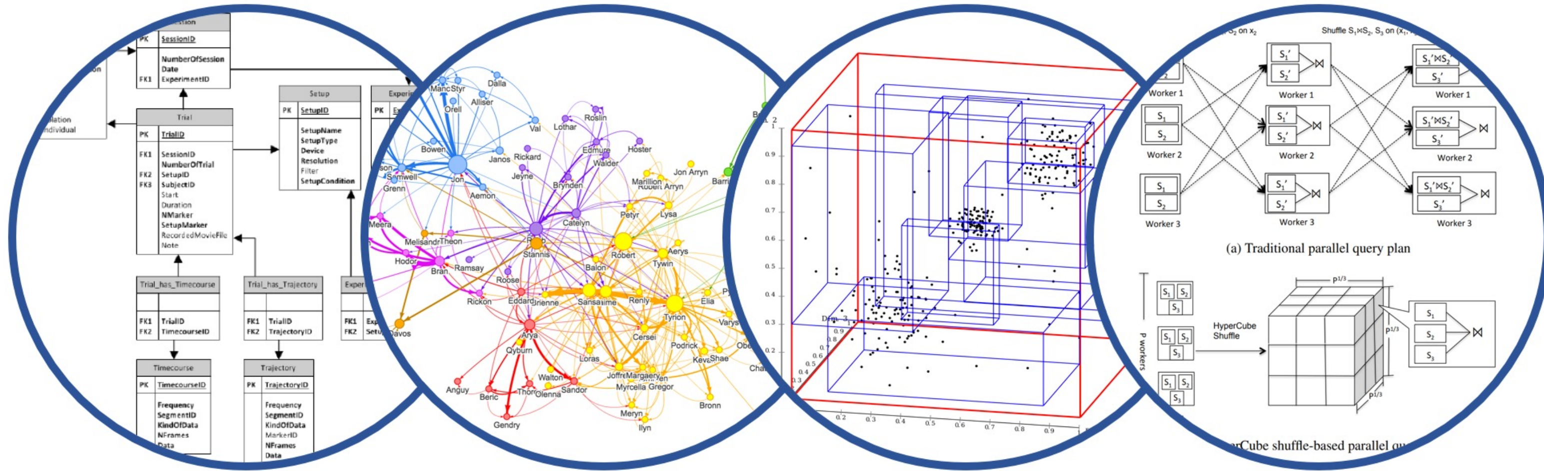


Introduction to Data Management

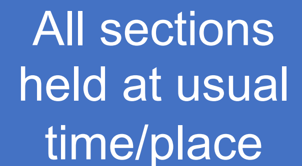
Disk Storage and Indexes

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

Announcements

- HW6:
  - Python of Java
  - Choose only one
  - Part 1 due 5/17. No late days (for quick feedback)
  - Part 2 due 5/24. Much more work than part 1

- Sections tomorrow: basically, OH for setup HW6
  - ~~Section at 12:30 in ECE 054,~~
  - ~~Section at 1:30 in SMI 105.~~
  - Come only if you need help with HW6

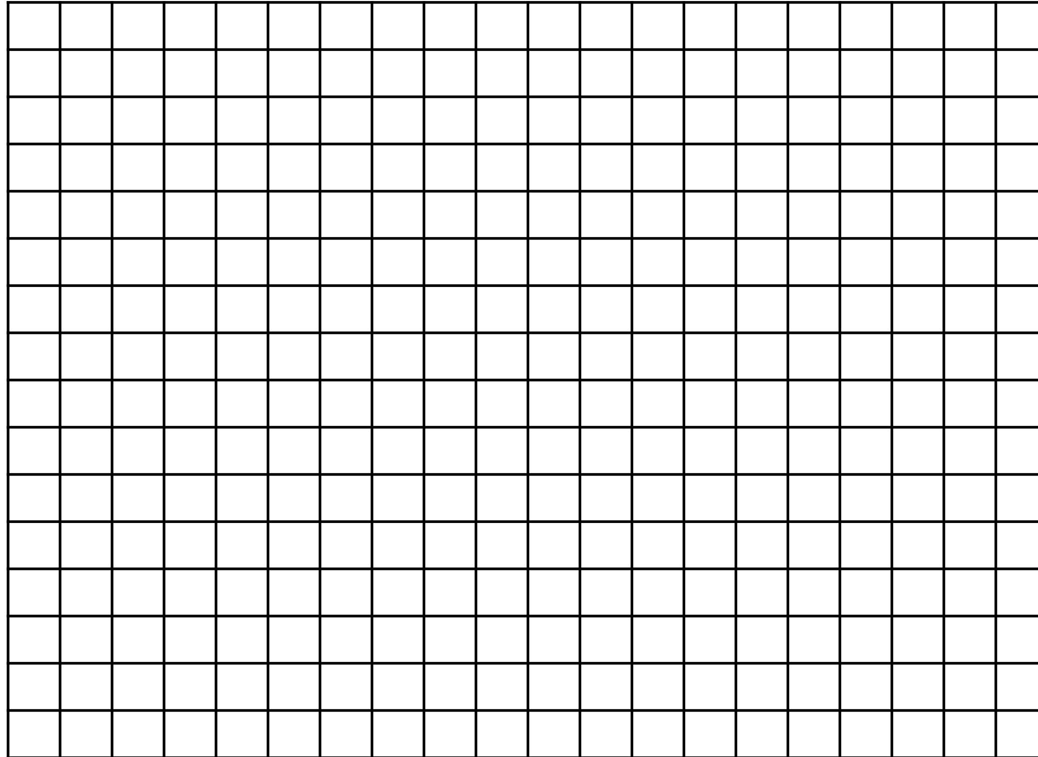


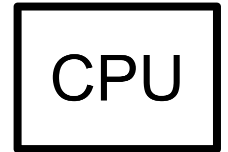
How the Main Memory Works

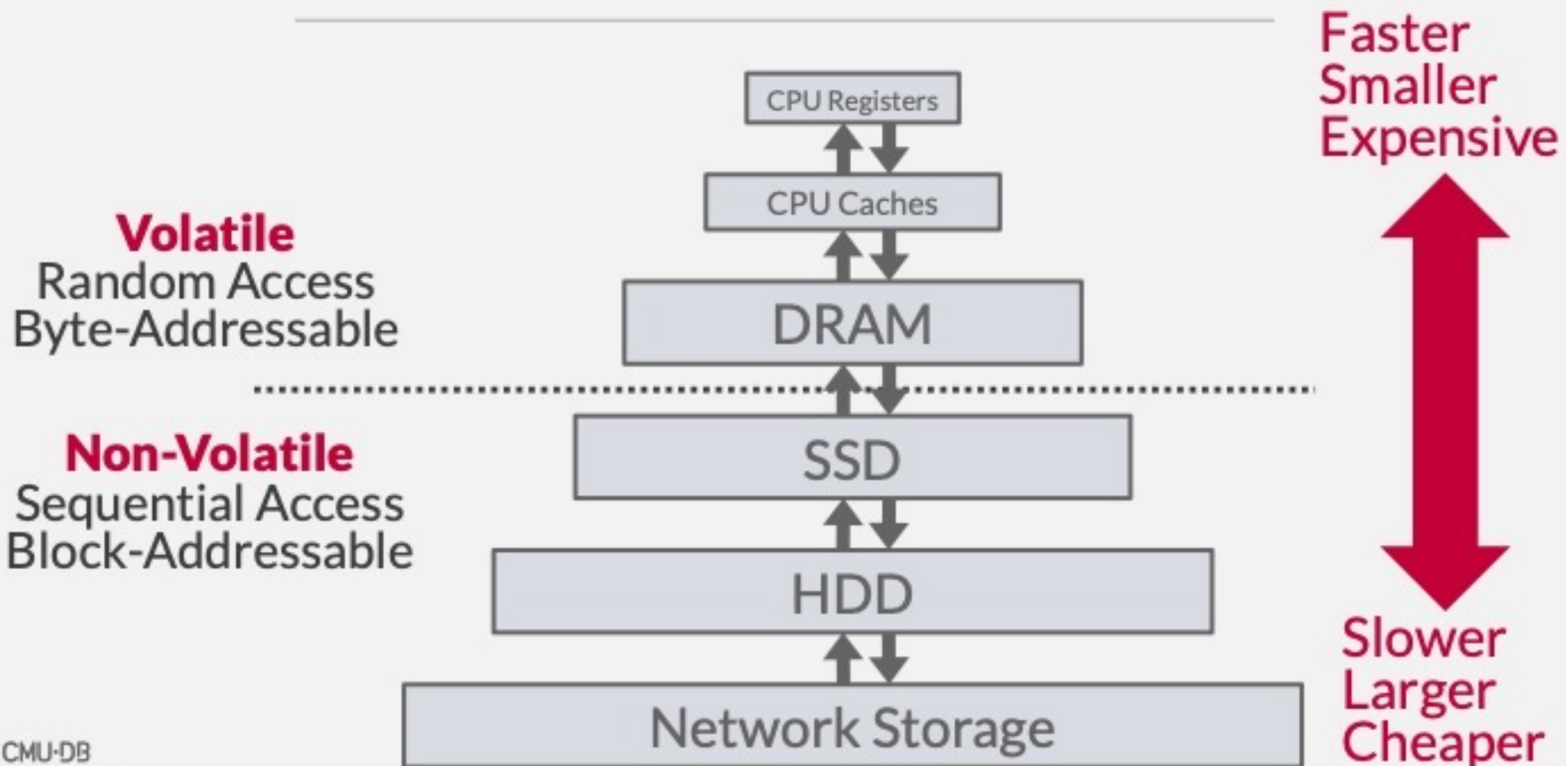
Disk

Main Memory

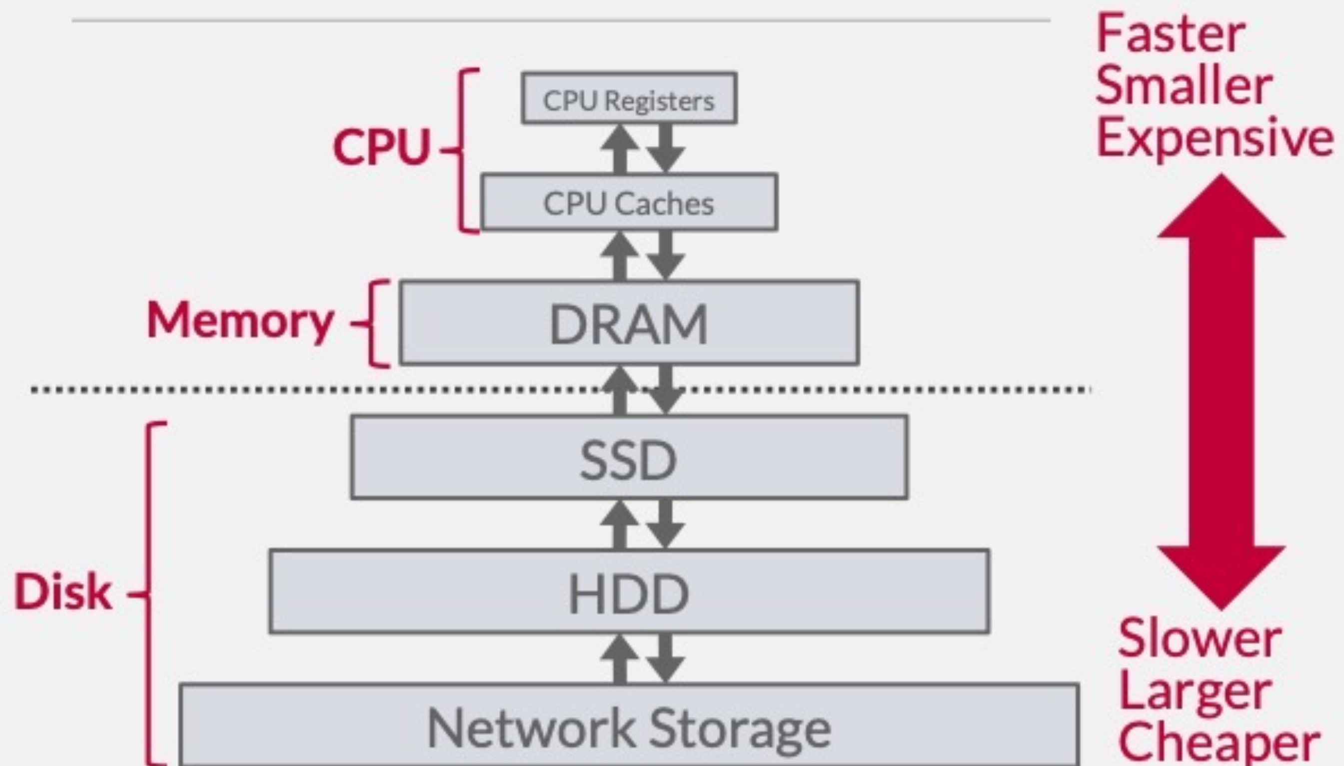
Processor







Credit: https://15445.courses.cs.cmu.edu/fall2023/



Credit: https://15445.courses.cs.cmu.edu/fall2023/

Credit: https://15445.courses.cs.cmu.edu/fall2023/

13

ACCESS TIMES

Latency Numbers Every Programmer Should Know

1 ns	L1 Cache Ref	1 sec
4 ns	L2 Cache Ref	4 sec
100 ns	DRAM	100 sec
16,000 ns	SSD	4.4 hours
2,000,000 ns	HDD	3.3 weeks
~50,000,000 ns	Network Storage	1.5 years
1,000,000,000 ns	Tape Archives	31.7 years

CMU·DB
15-445/645 (Fall 2023)

Source: Colin Scott

Credit: https://15445.courses.cs.cmu.edu/fall2023/

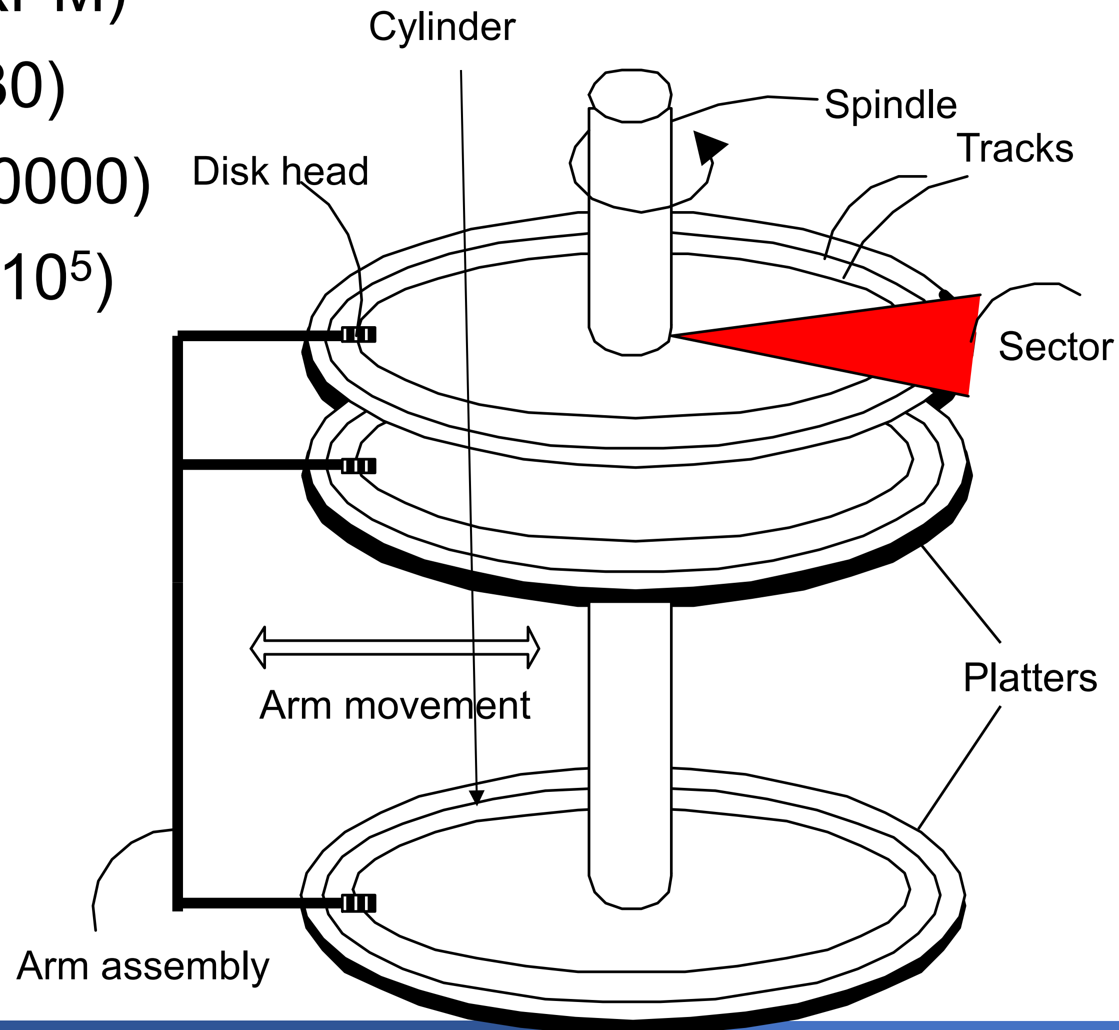
Disk

- Stores data persistenly
- Different technologies:
  - Tapes: long-term archives
  - HDD: most today's data is stored here
  - SDD: your laptop, or cache for HDD
- Unit of Read/Write operation:
  1 block = 0.5k .. 32k

Hard Drive Disk (HDD)

Mechanical characteristics:

- Rotation speed (5400RPM)
- Number of platters (1-30)
- Number of tracks (<=10000)
- Number of bytes/track(10^5)



Disk Access Characteristics

Disk latency = seek time + rotational latency

- Seek time = time for the head to reach cylinder
  - 10ms – 40ms
- Rotational latency = time for sector to rotate
  - Rotation time = 10ms

Throughput = typically 40-80MB/s

Blocks or Pages

- The unit of disk read or write is a block
- Once in main memory, we call it a page
- Block size is fixed. Typically, 4k or 8k or 16k

Storage Manager

DBMS creates one large file

- 1 file = multiple (continuous?) blocks
- 1 block = multiple records, free space

Storing Pages on Disk

Database file

Page 0	Page 1	Page 2	Page 3	Page 4

Database file

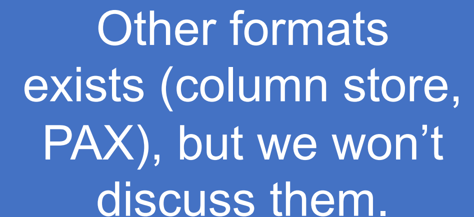
Page 5	Page 6	Page 7	Page 8	Page 9

Page Format

Row-oriented Format

A.k.a. N-ary storage model NSM

- Each page stores several records
- Each records stores its attributes



Row-Oriented Storage

Payroll

UserID	Name	Job
123	Jack	TA
345	Allison	TA
567	Magda	Prof
789	Dan	Prof

Row-Oriented Storage

Page 0

123	Jack	TA
345	Allison	TA
567	Magda	Prof
...		

Page 1

789	Dan	Prof
...		
...		

...

Payroll

UserID	Name	Job
123	Jack	TA
345	Allison	TA
567	Magda	Prof
789	Dan	Prof

Row-Oriented Storage

Physical layout

Page 0

123	Jack	TA
345	Allison	TA
567	Magda	Prof
…		

Page 1

789	Dan	Prof
…		
…		

…

Logical schema

Payroll

UserID	Name	Job
123	Jack	TA
345	Allison	TA
567	Magda	Prof
789	Dan	Prof

Row-Oriented Storage

Physical layout

Page 0

123	Jack	TA
345	Allison	TA
567	Magda	Prof
…		

Page 1

789	Dan	Prof
…		
…		

…

Logical schema

Payroll

UserID	Name	Job
123	Jack	TA
345	Allison	TA
567	Magda	Prof
789	Dan	Prof

The schema stored separately, in the *database catalog*

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

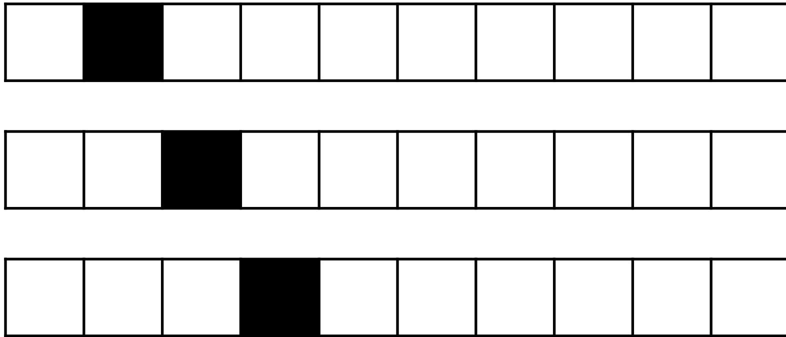
Sequential

Sequential/Random Access

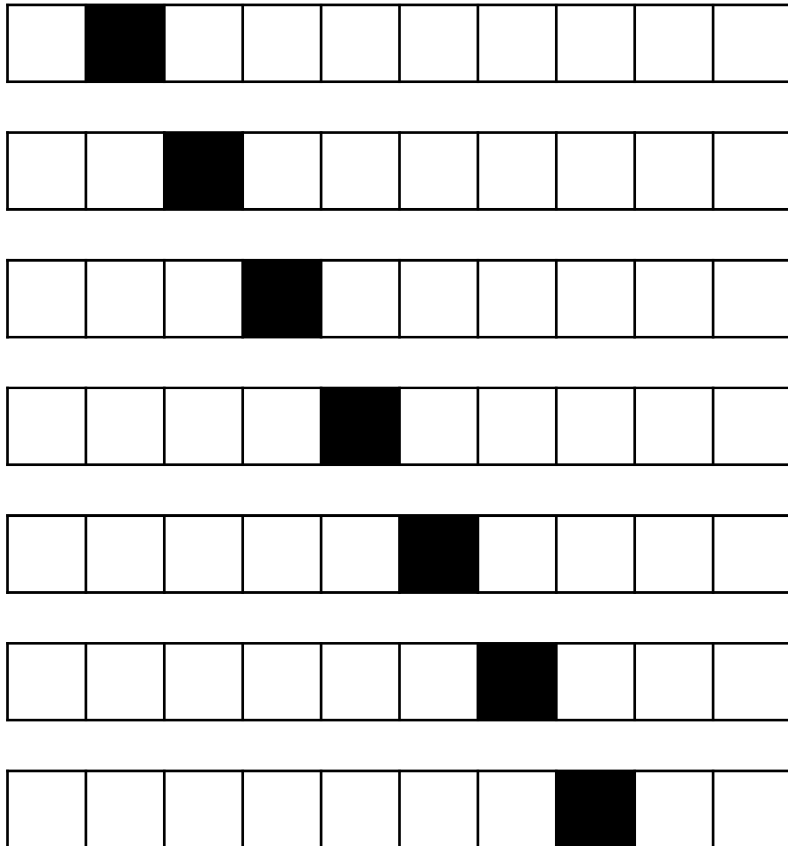
Sequential

Sequential/Random Access

Sequential

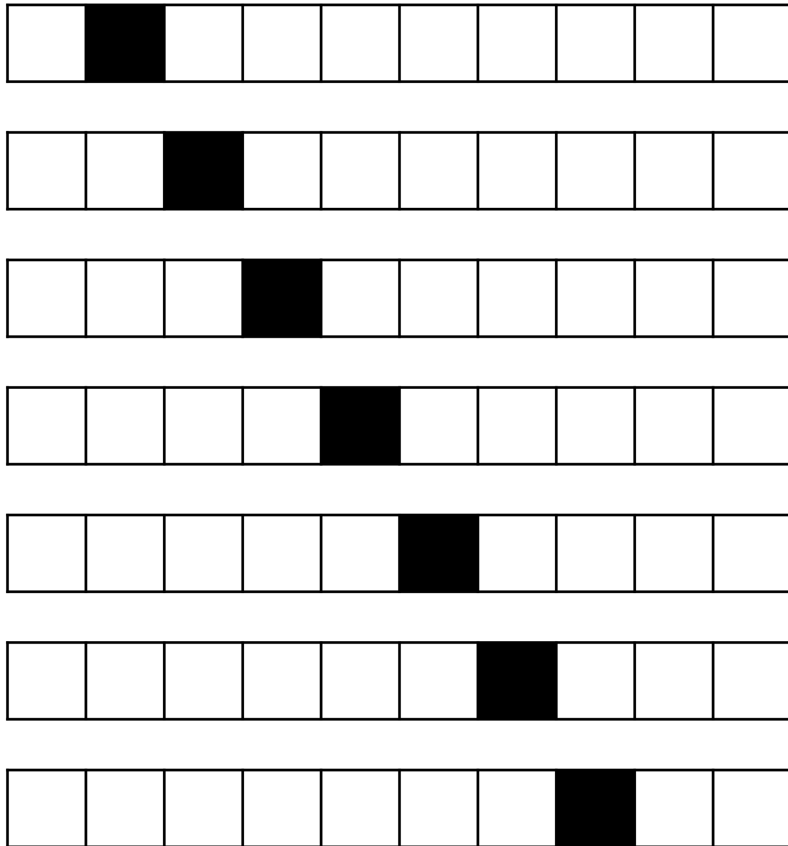


Sequential/Random Access

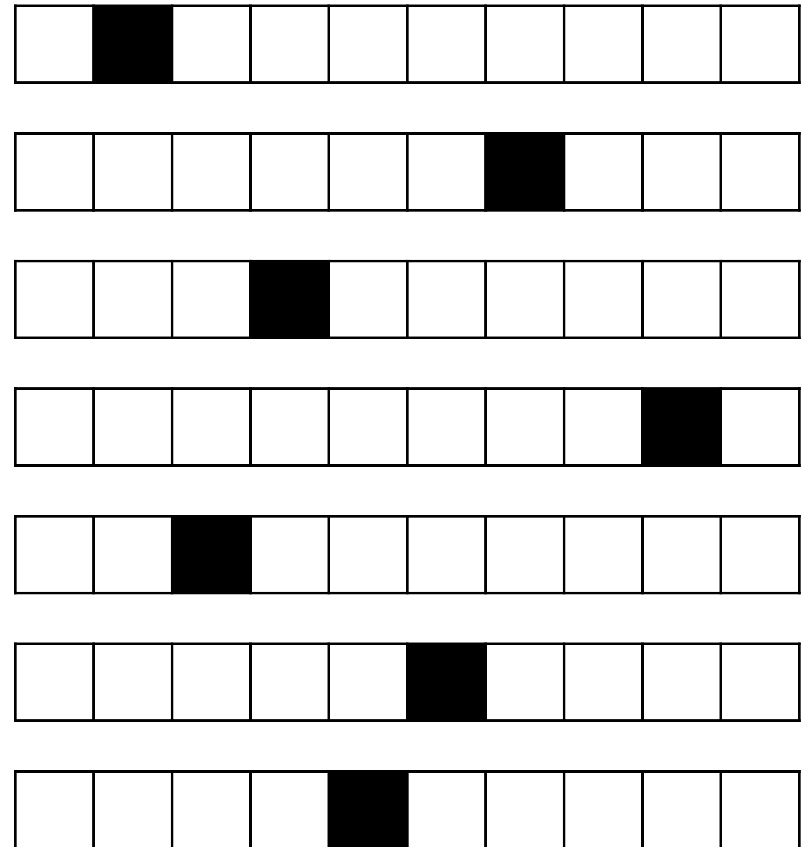


Sequential/Random Access

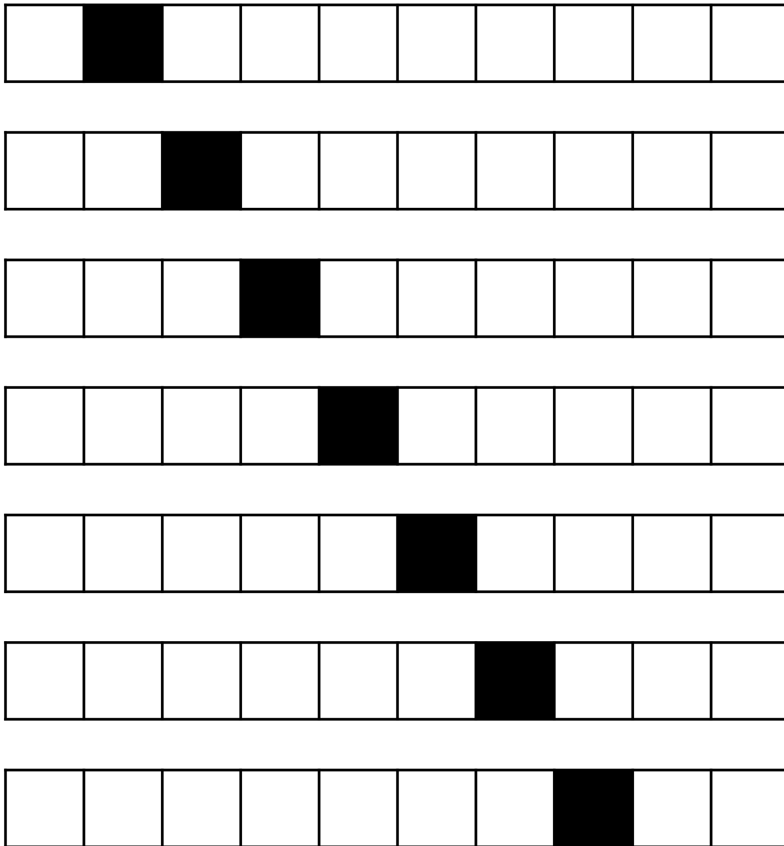
Sequential

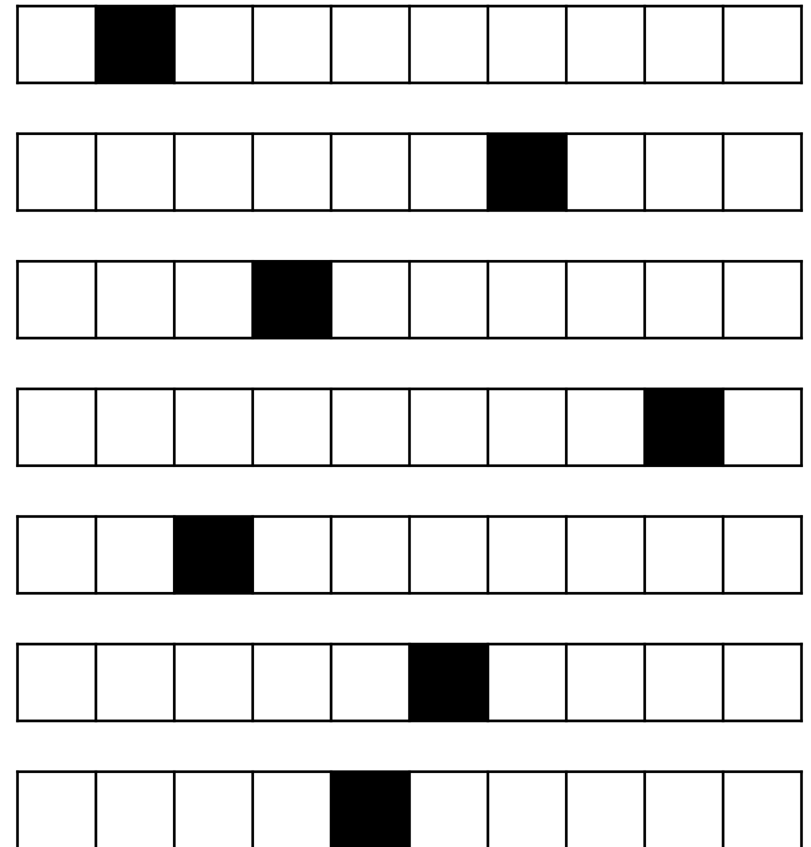


Random



Sequential/Random Access





Indexes

Indexes

- The relation is stored in a file (= set of blocks)
- An index is an auxiliary file that facilitates faster access to the data

Indexes

An index is an auxiliary file that facilitates faster access to the data

```
SELECT *
FROM Payroll
WHERE Name = 'Allison';
```

Page 0

123	Jack	TA
…		
…		

Page 1

…	…	…
…		
…		

Page 2

…	…	…
…		
…		

Page 3

…		
…	Allison	
…		

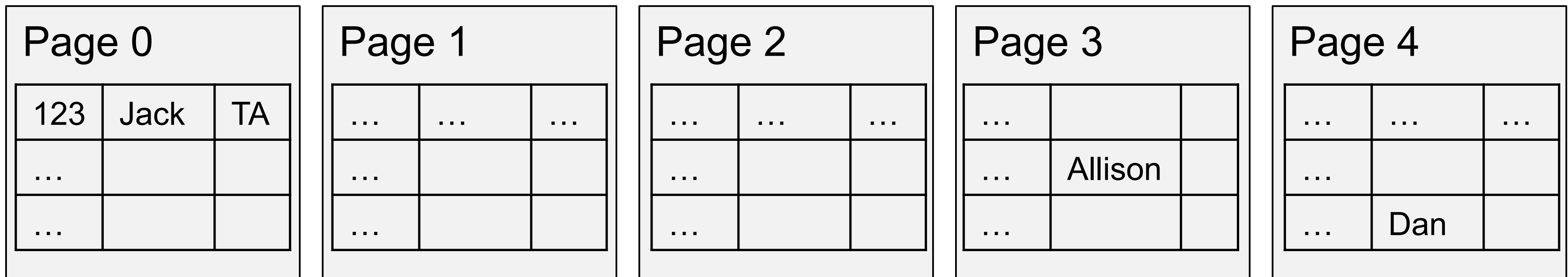
Page 4

…	…	…
…		
…	Dan	

Indexes

An index is an auxiliary file that facilitates faster access to the data

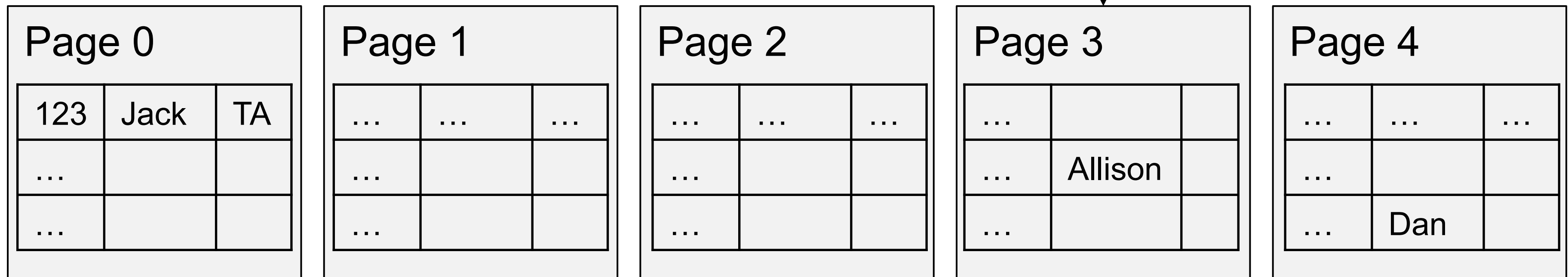
```
SELECT *
FROM Payroll
WHERE Name = 'Allison';
```



Indexes

An index is an auxiliary file that facilitates faster access to the data

```
SELECT *
FROM Payroll
WHERE Name = 'Allison';
```



Indexes

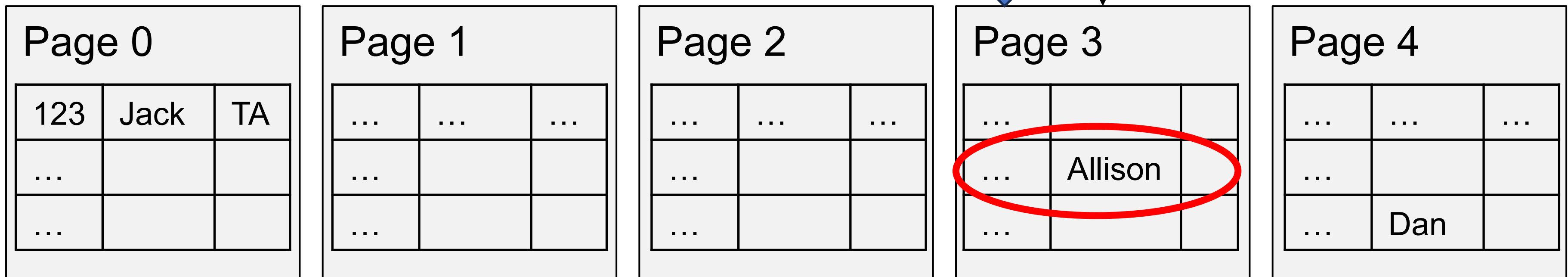
An index is an auxiliary file that facilitates faster access to the data

```
SELECT *
FROM Payroll
WHERE Name = 'Allison';
```

With an index: random access to what we need

Index on Name

Without an index: full sequential scan



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

Physical Data Independence

- SQL query is the same,
  Regardless of whether there is an index or not

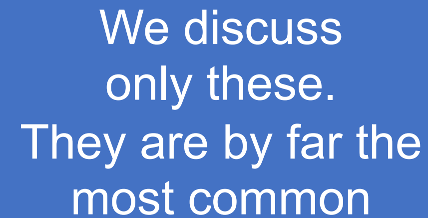
- This is called Physical Data Independence,
  and is one of the main benefits of Relational Model

- The DB administrator runs CREATE INDEX (later)

B+ Trees

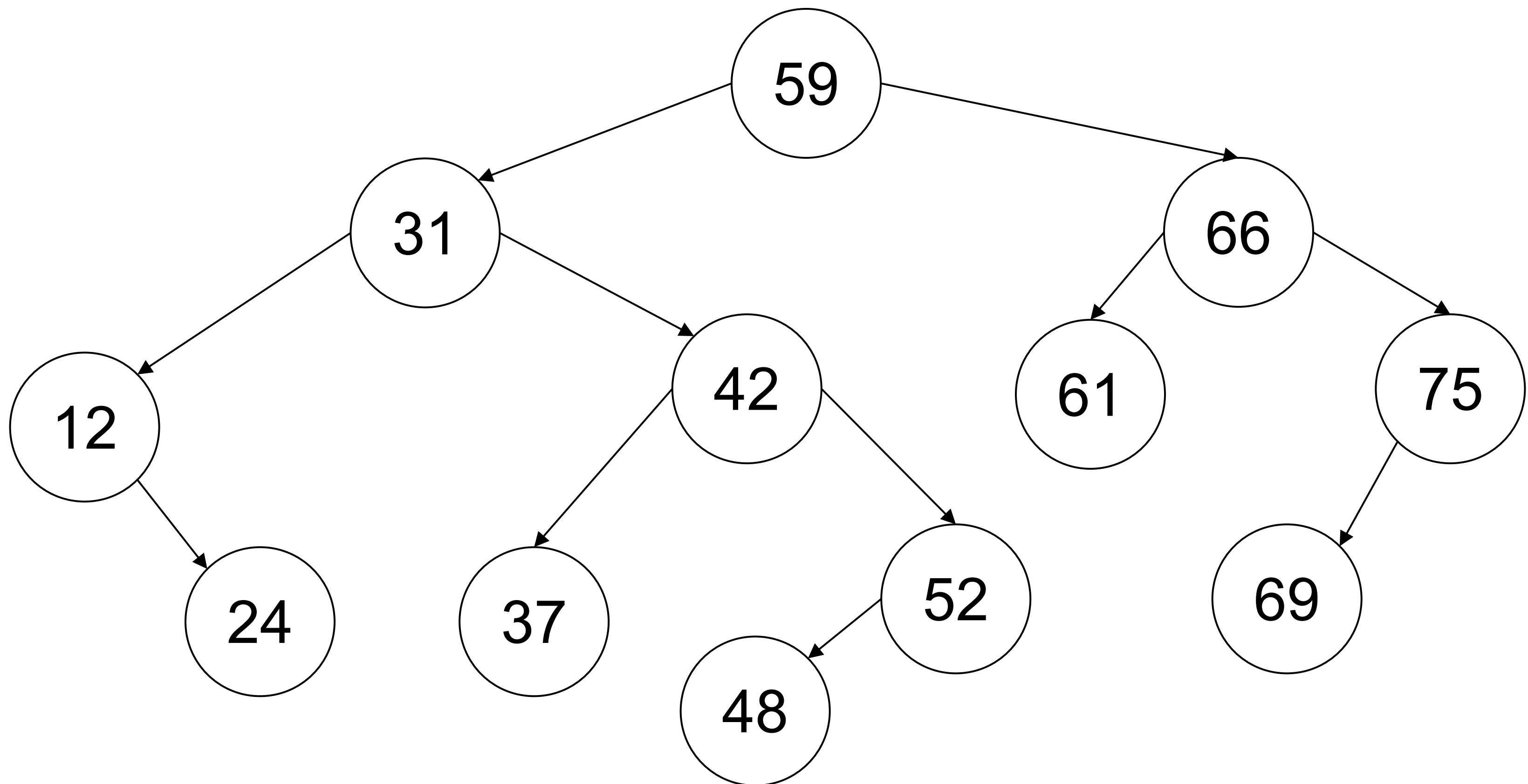
Index Types

- B+ trees



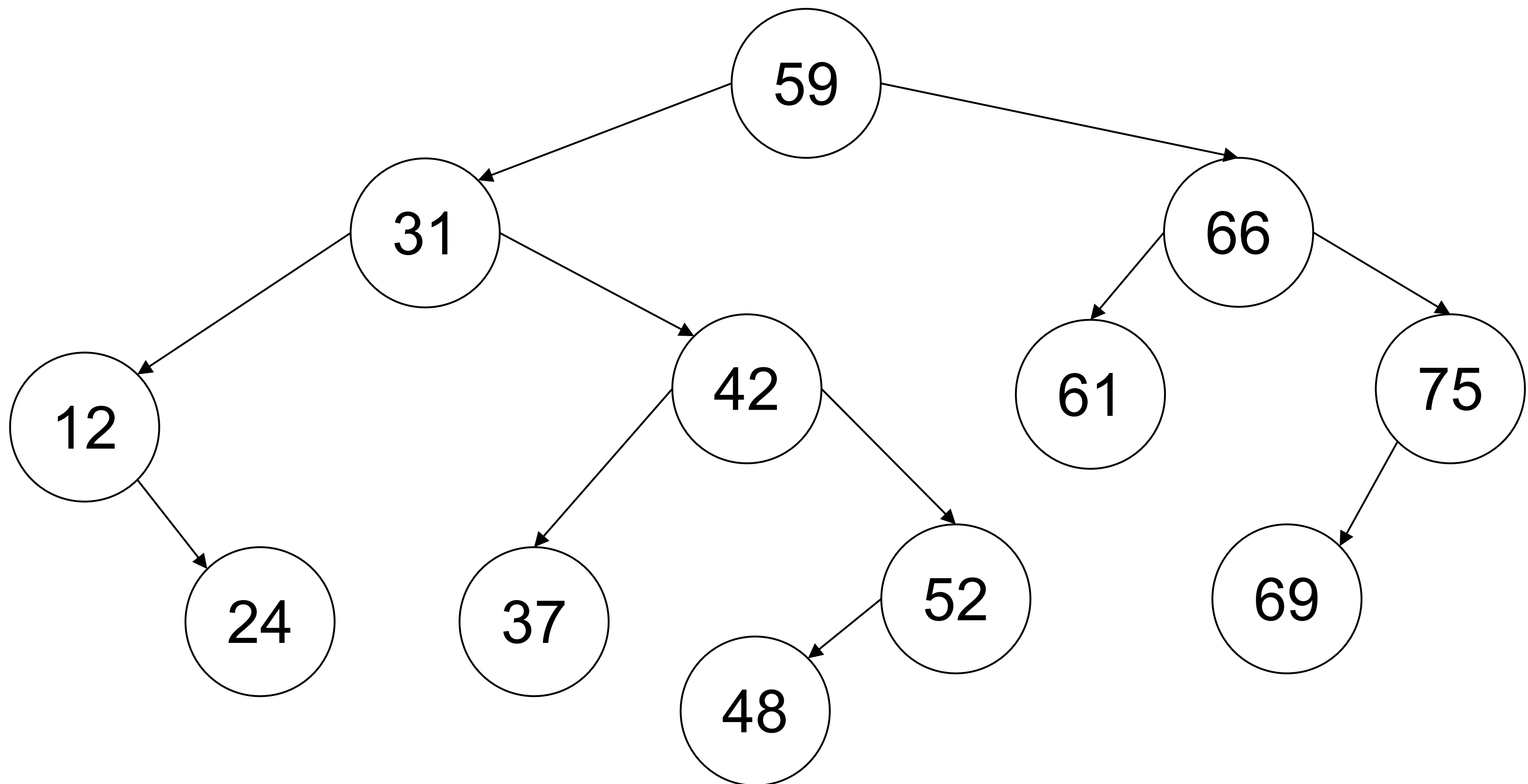
- Hash tables
- Bitmaps (for attributes with few values)
- R+ tree (for 2D data)

Review: Binary Search Tree

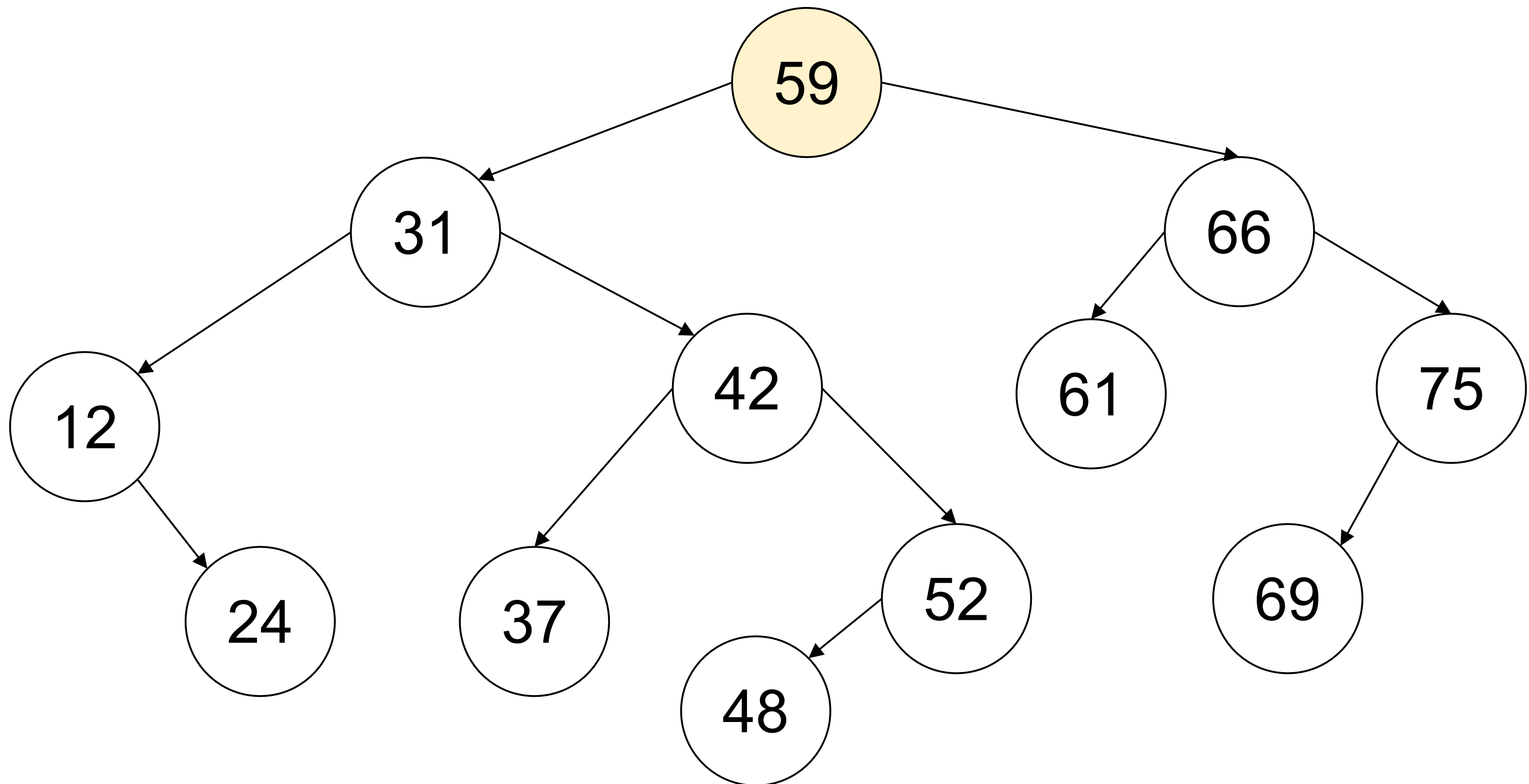


Review: Binary Search Tree

Find 48

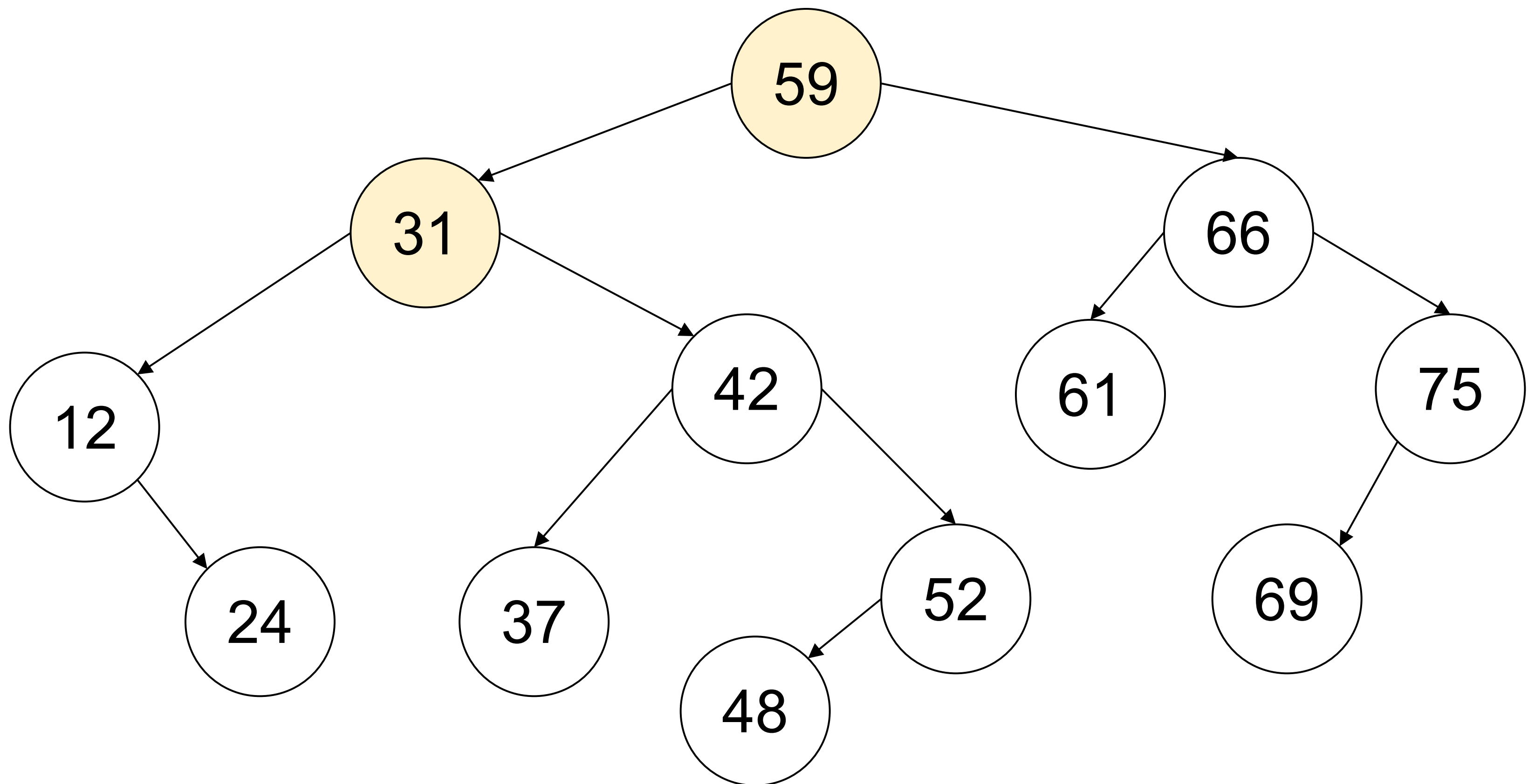


Review: Binary Search Tree



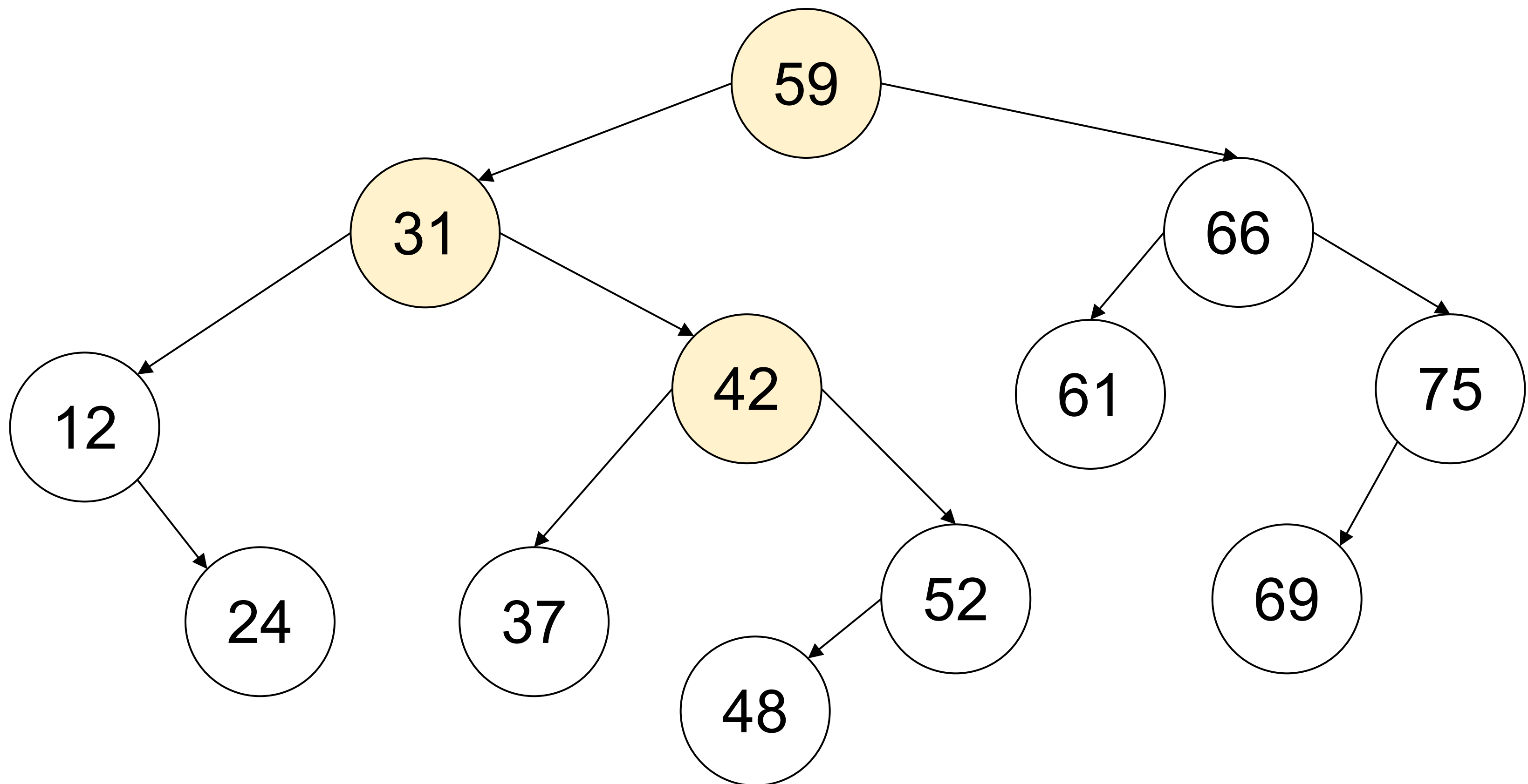
Review: Binary Search Tree

Find 48



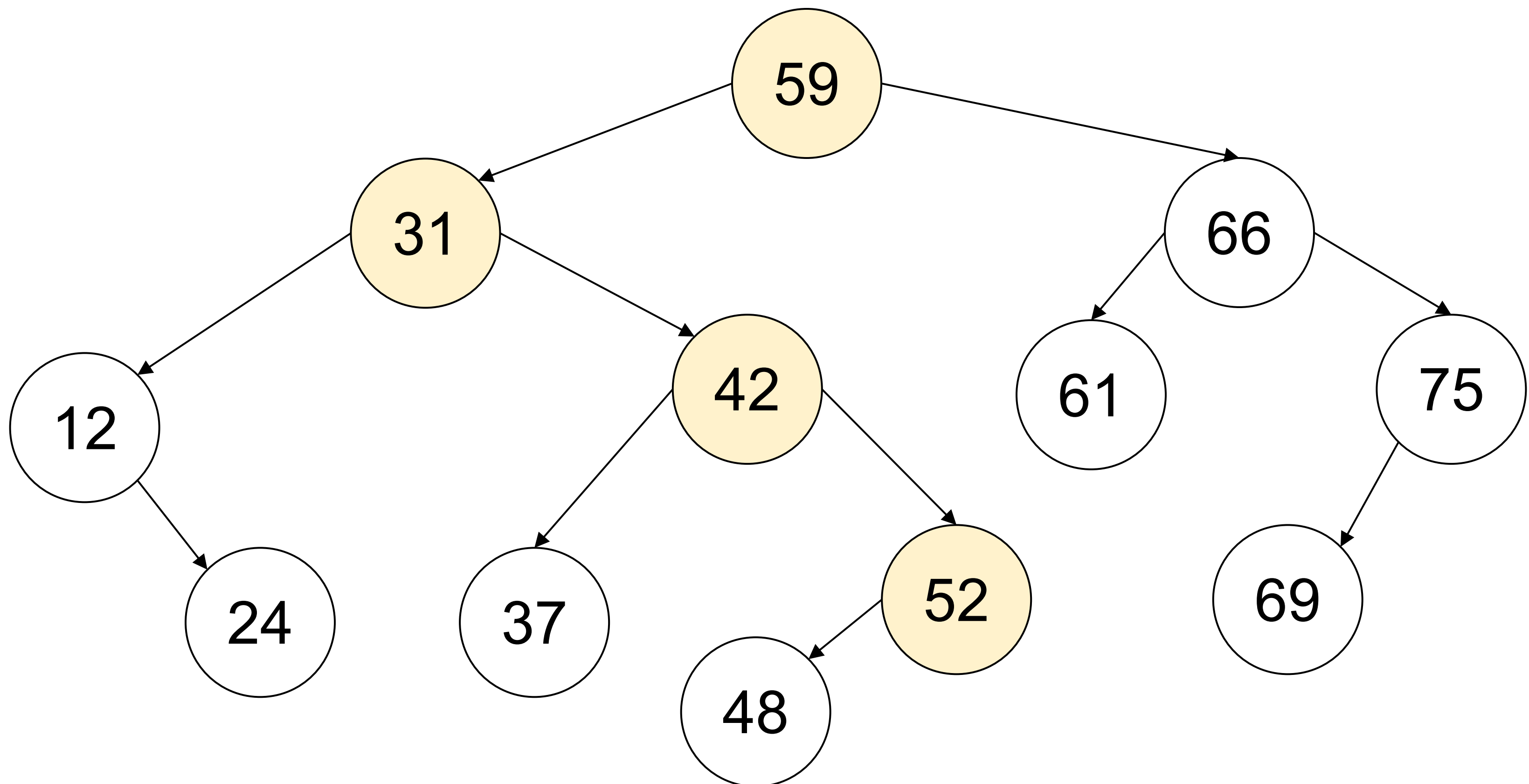
Review: Binary Search Tree

Find 48



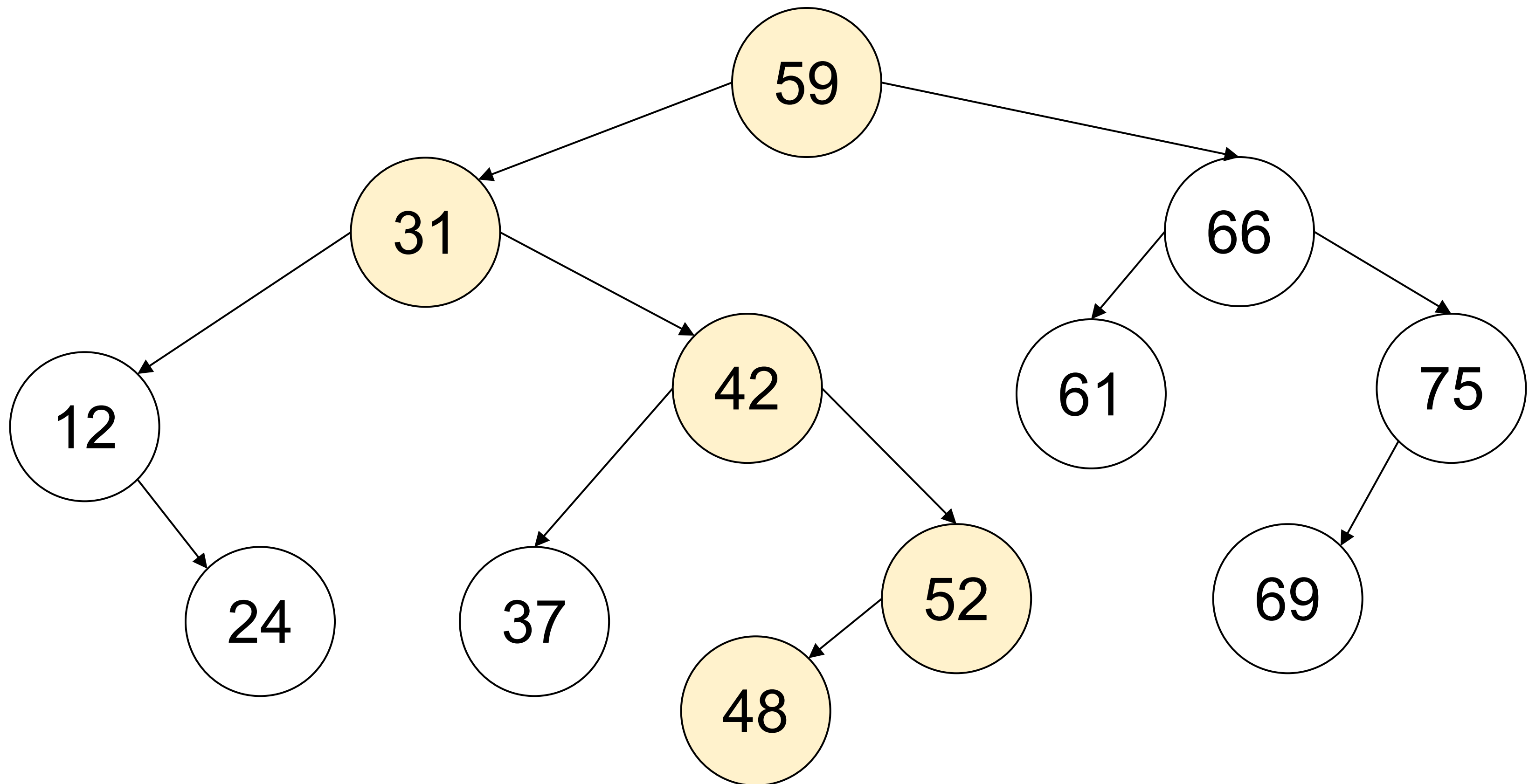
Review: Binary Search Tree

Find 48



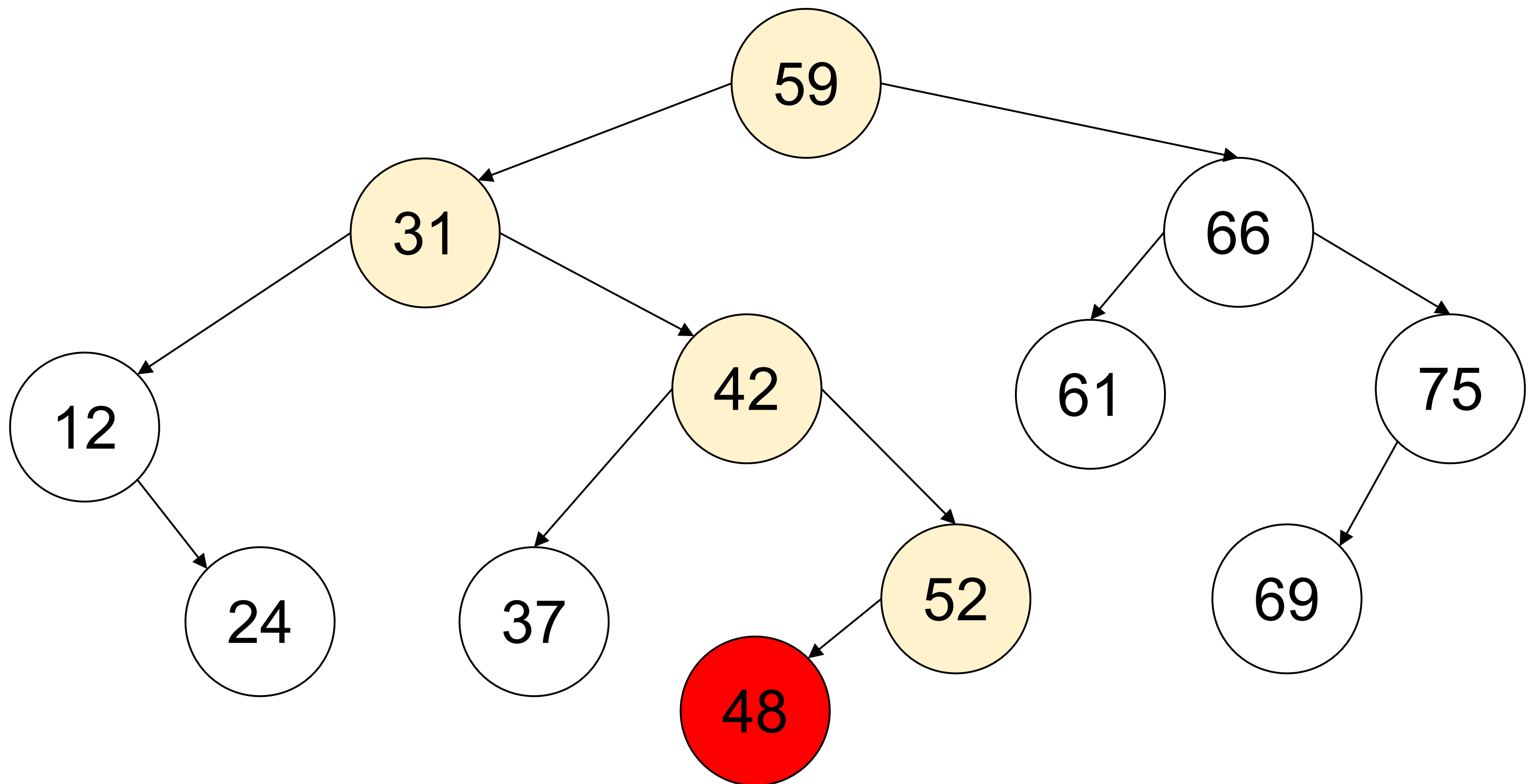
Review: Binary Search Tree

Find 48



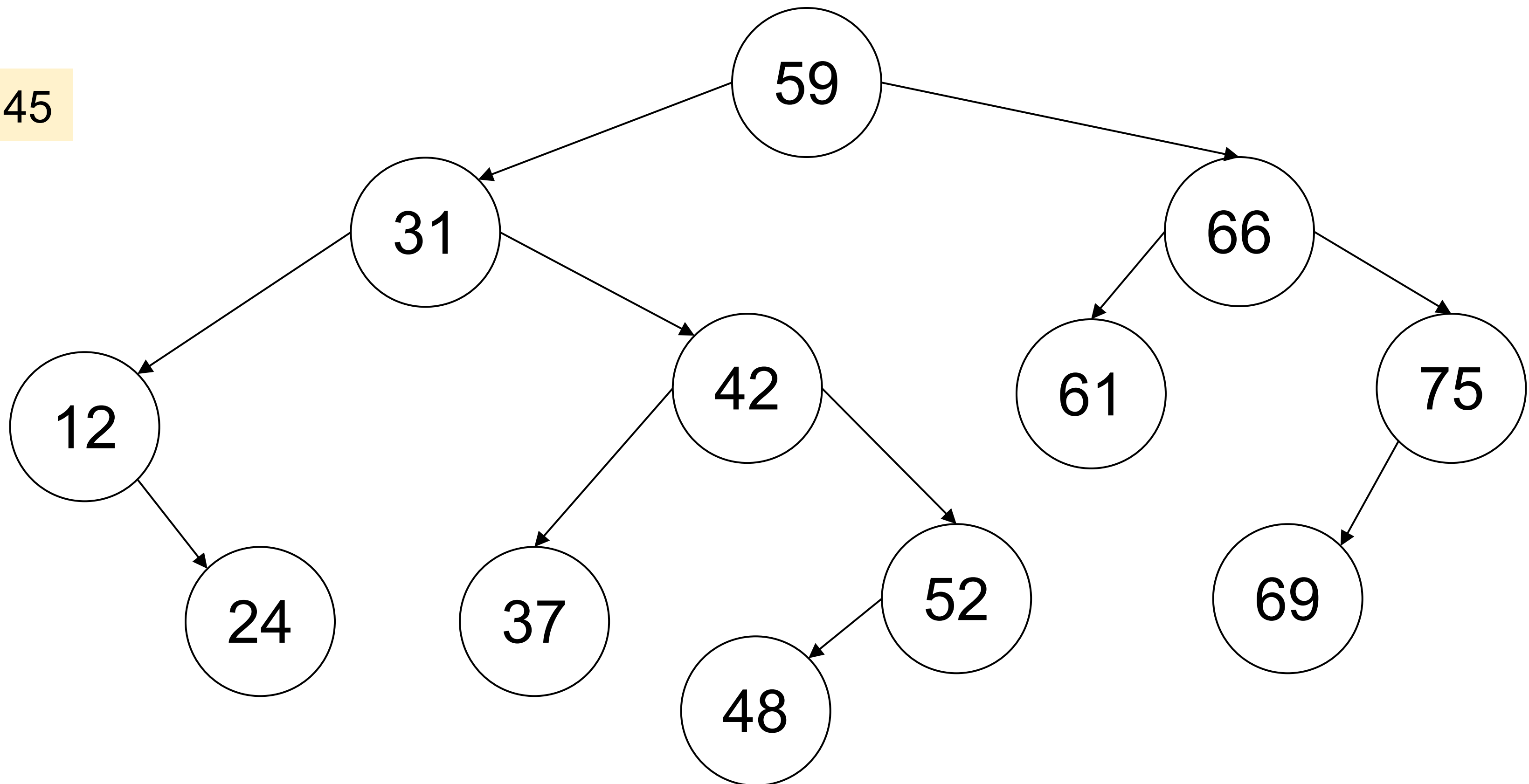
Review: Binary Search Tree

Find 48

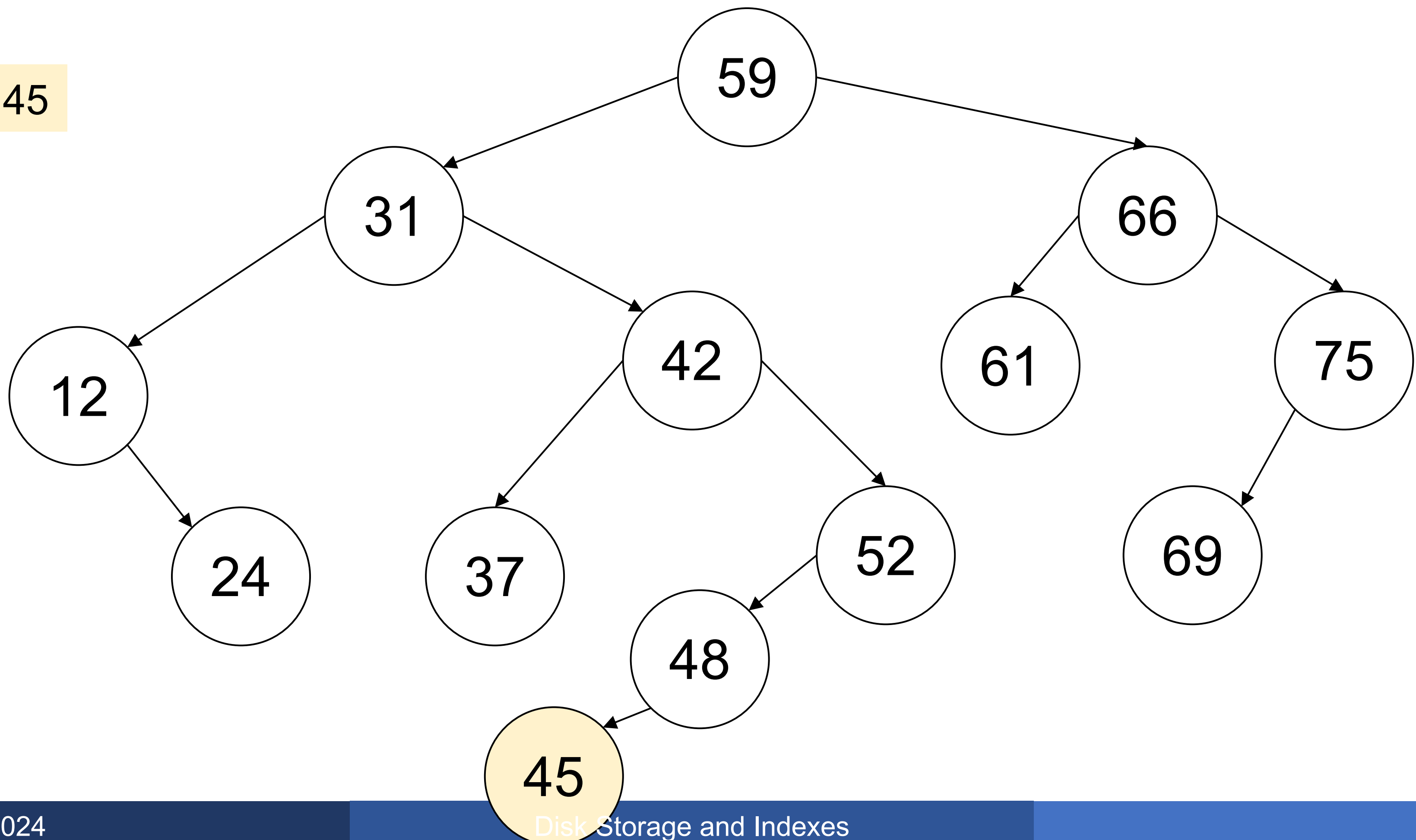


Review: Binary Search Tree

Insert 45



Review: Binary Search Tree



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)

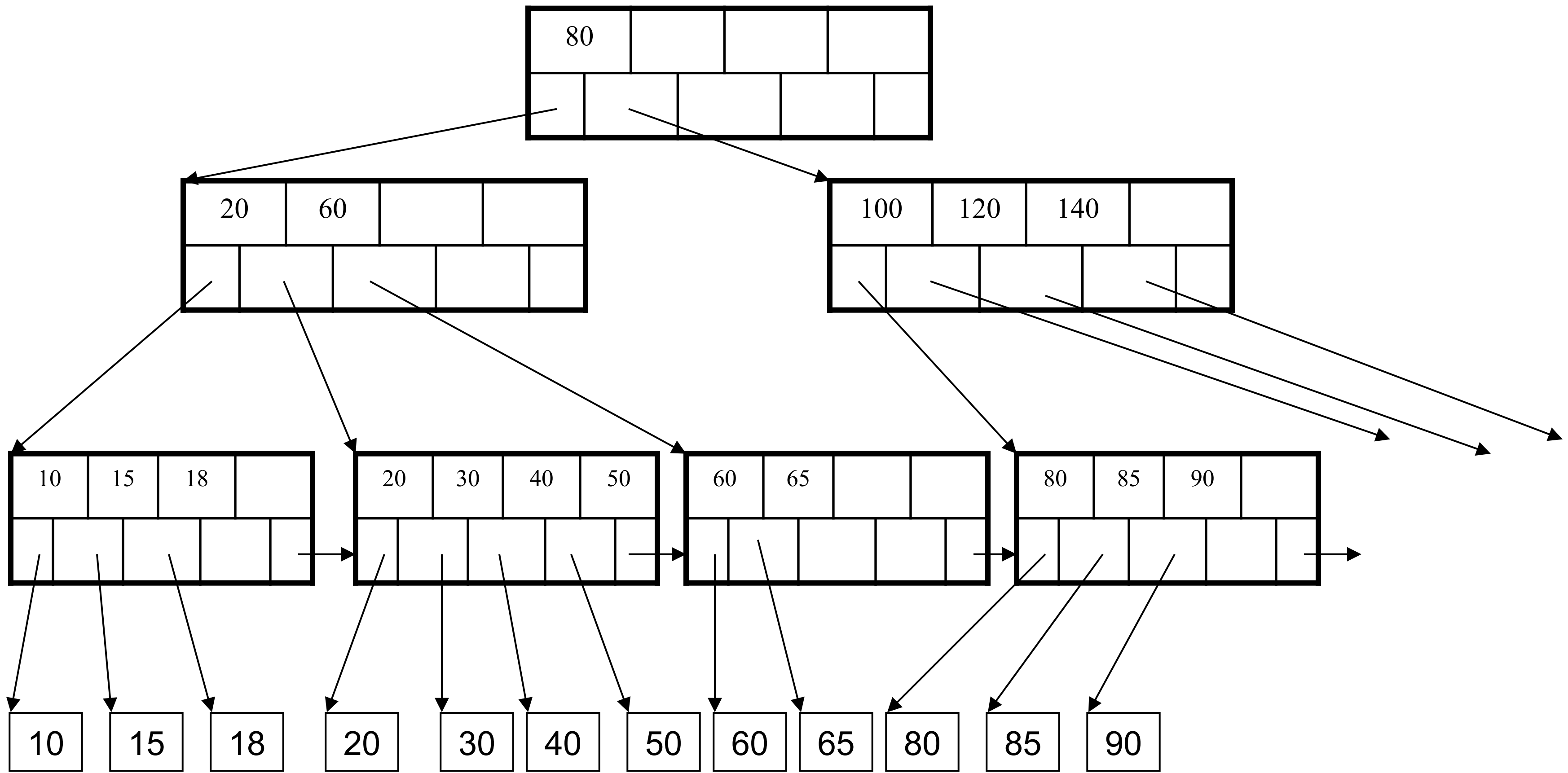
But not good for disk -- why??

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+ Tree Example

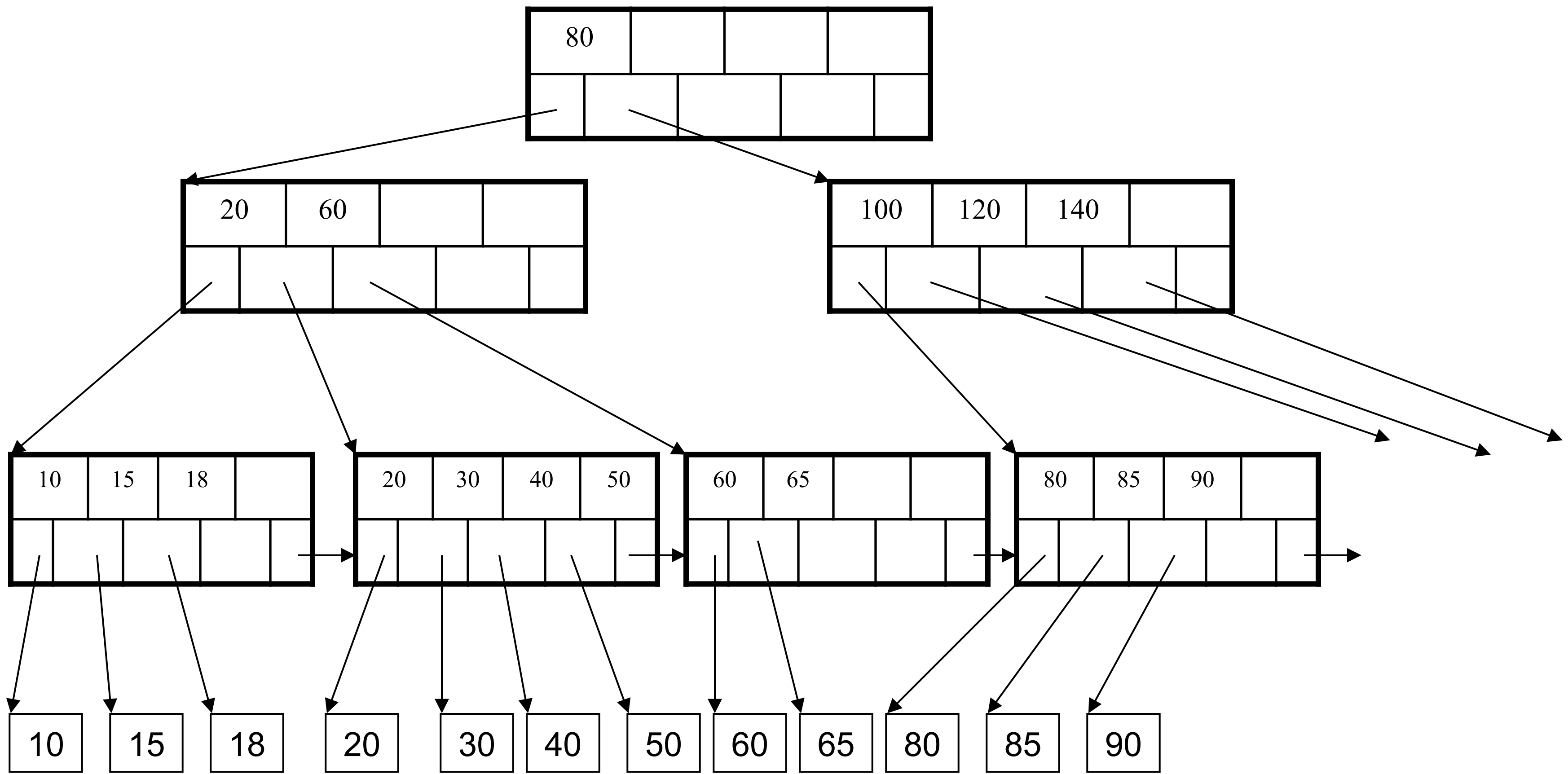
d = 2



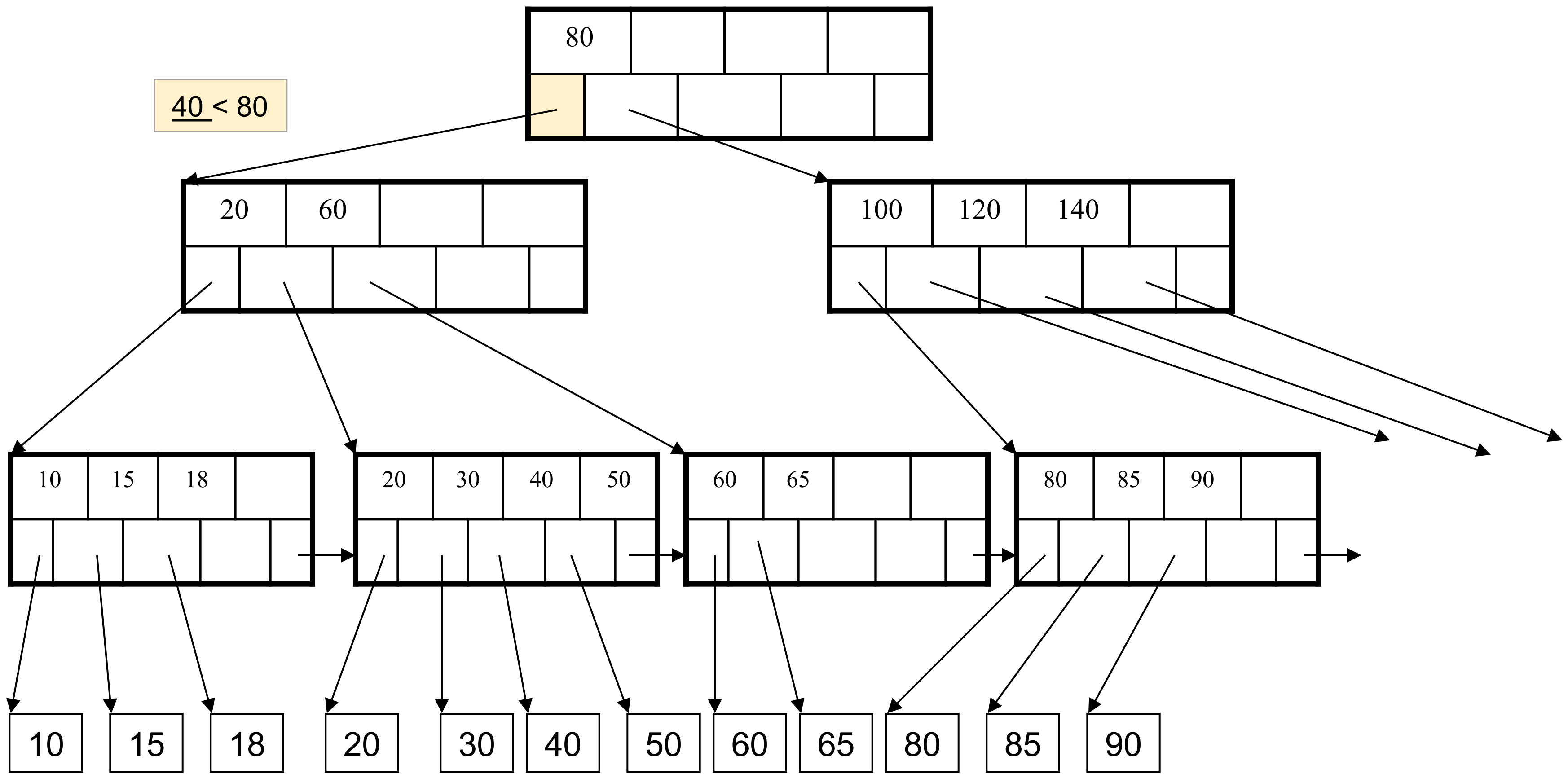
B+ Tree Example

d = 2

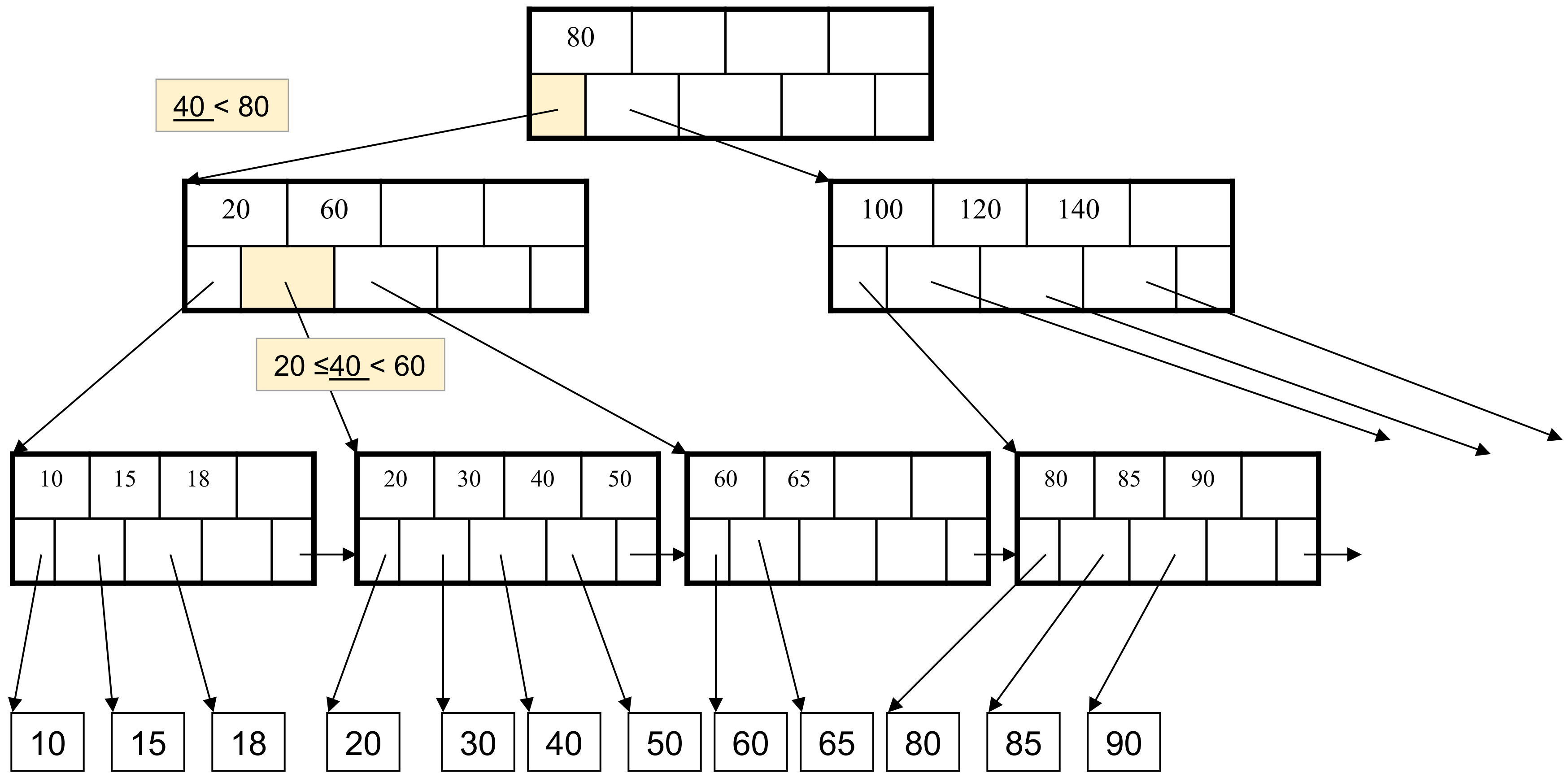
Find the key 40



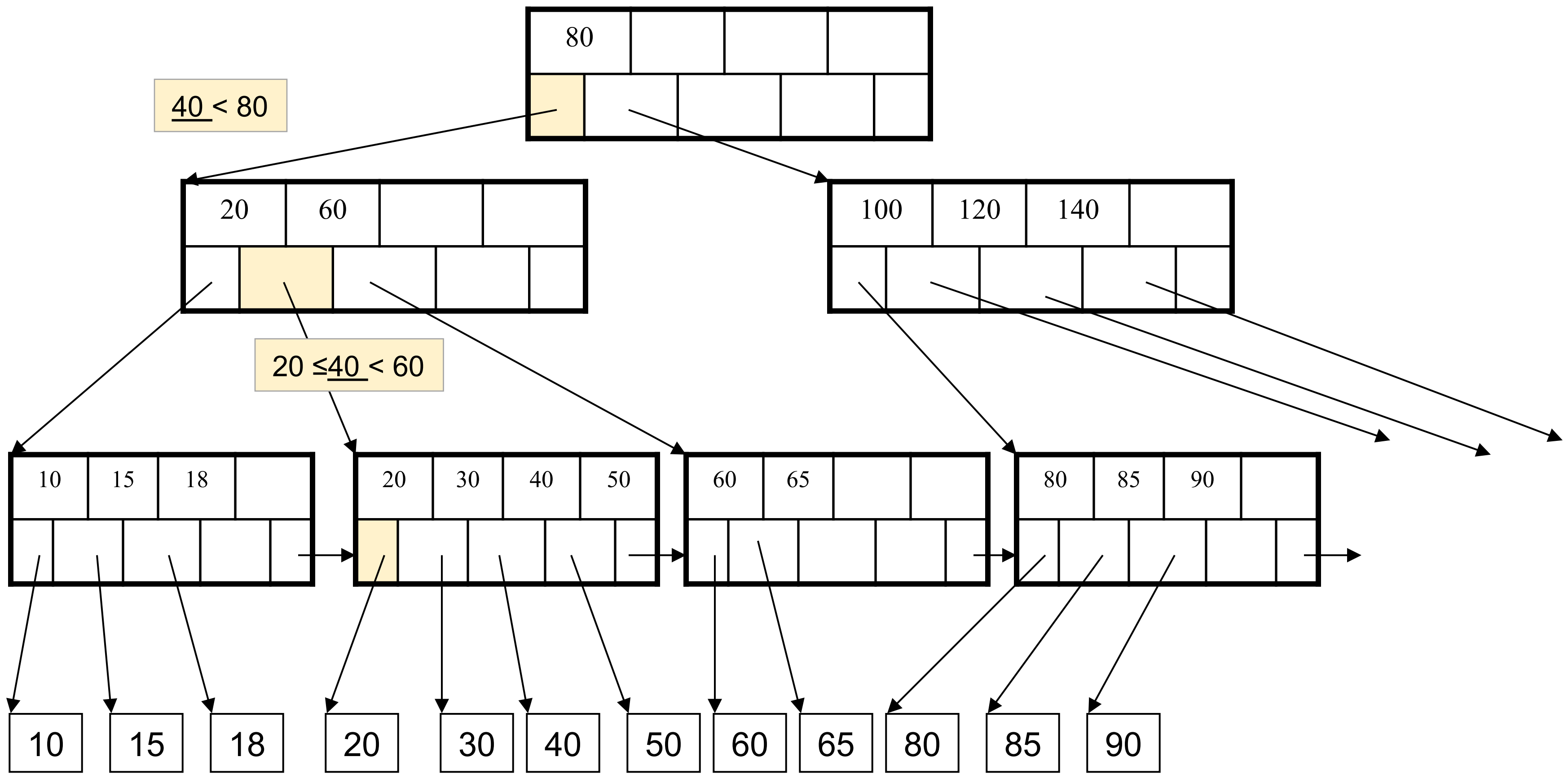
B+ Tree Example



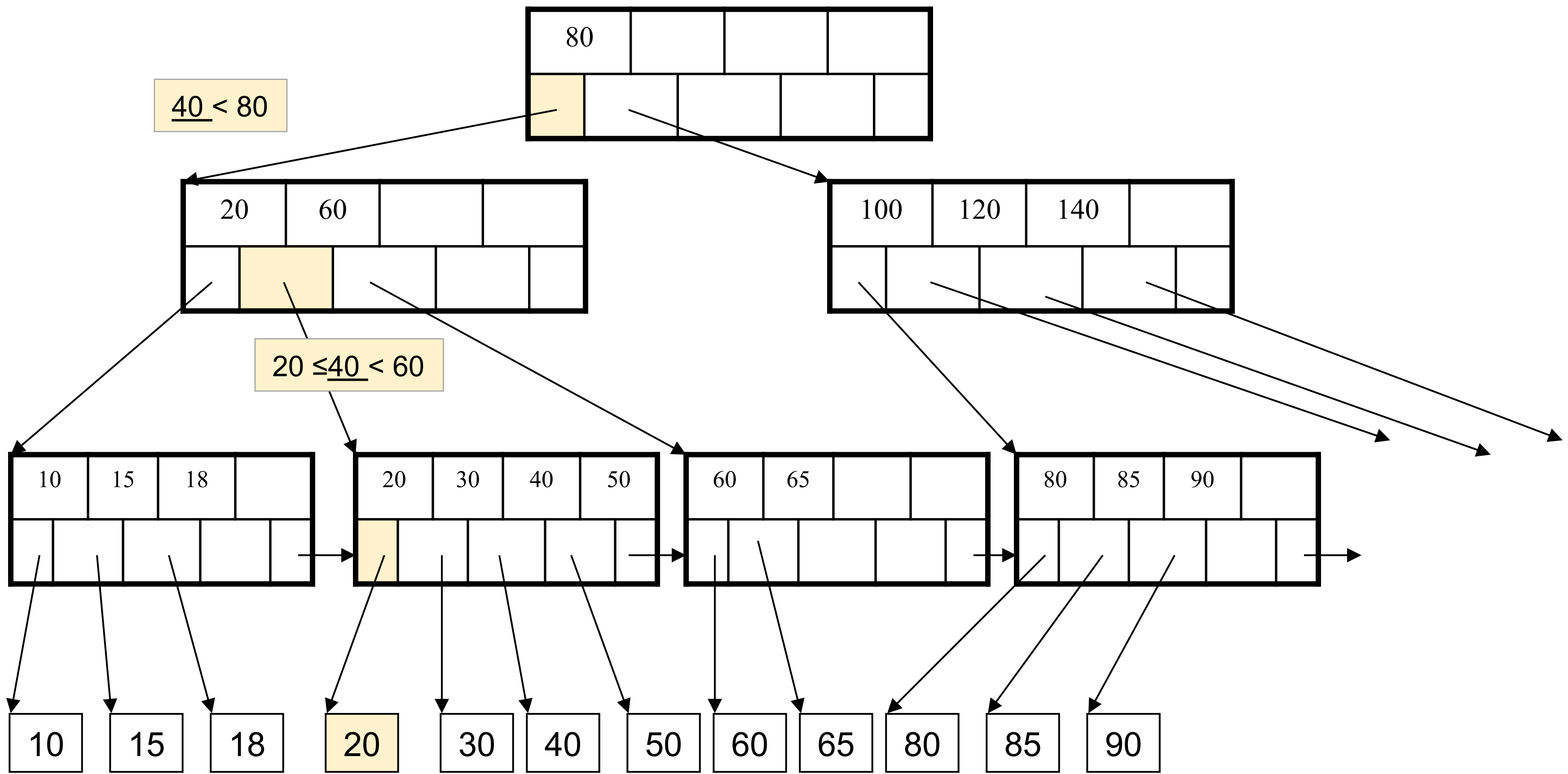
B+ Tree Example



B+ Tree Example

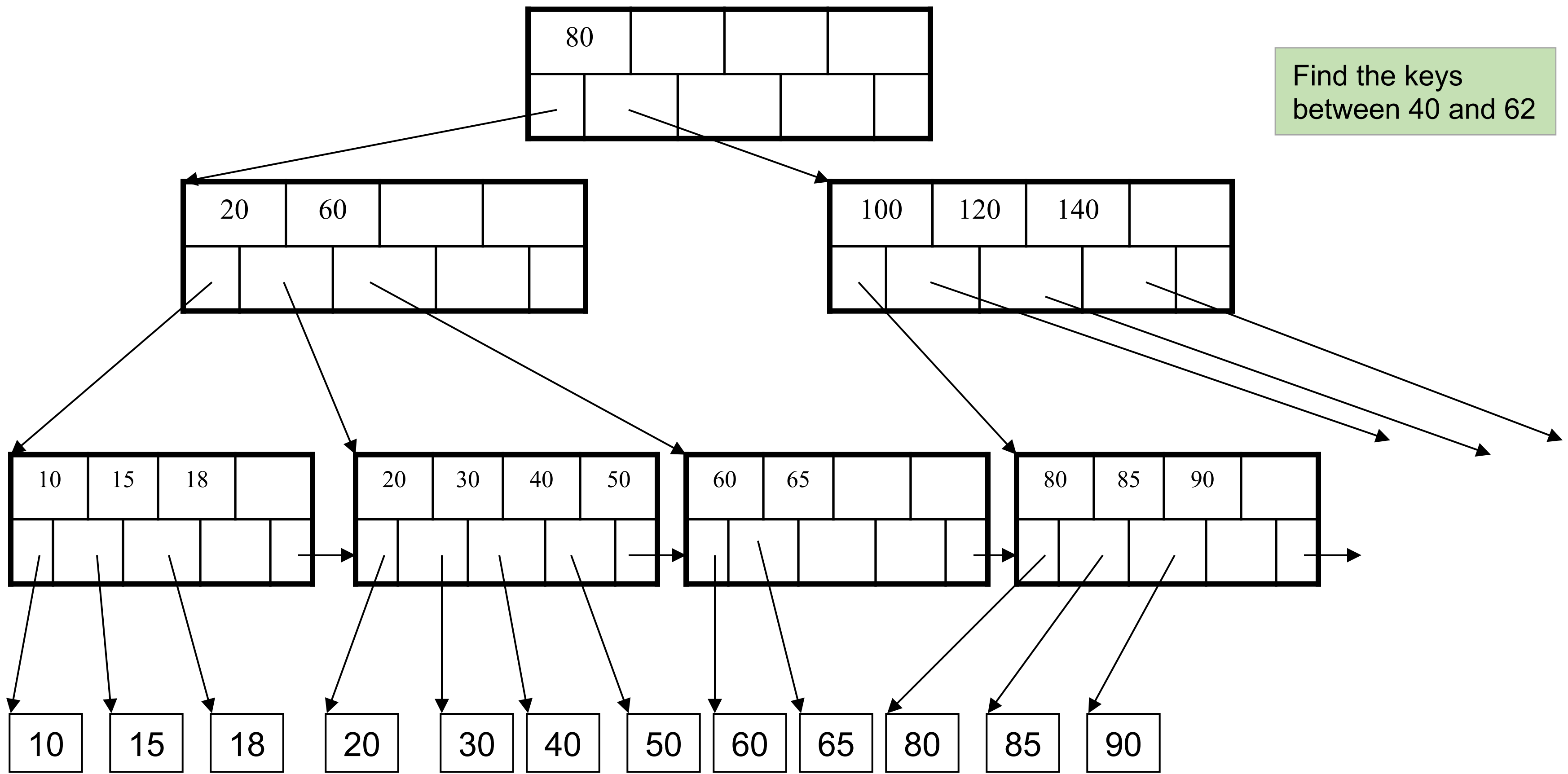


B+ Tree Example



B+ Tree Example

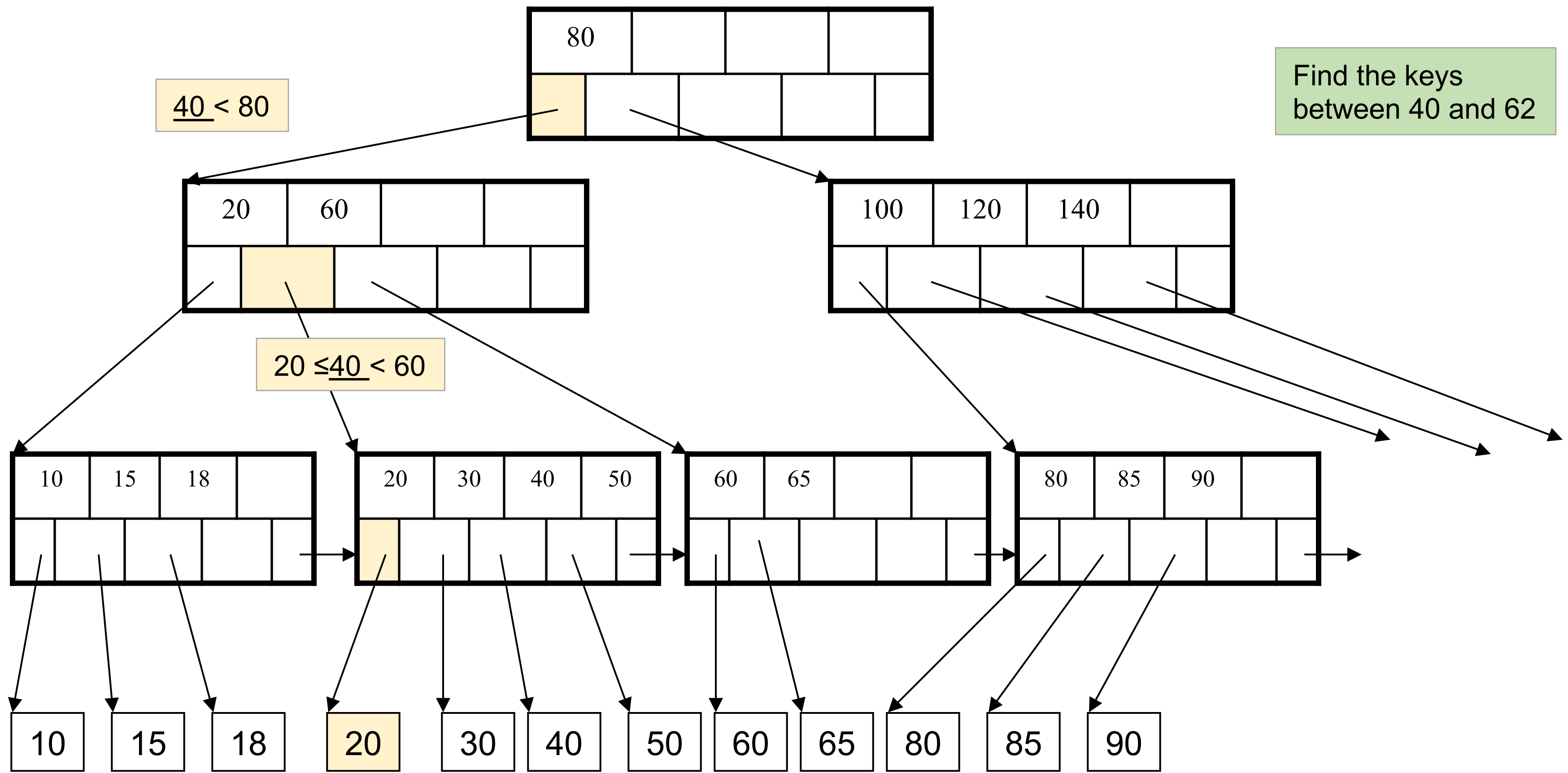
d = 2



Find the keys between 40 and 62

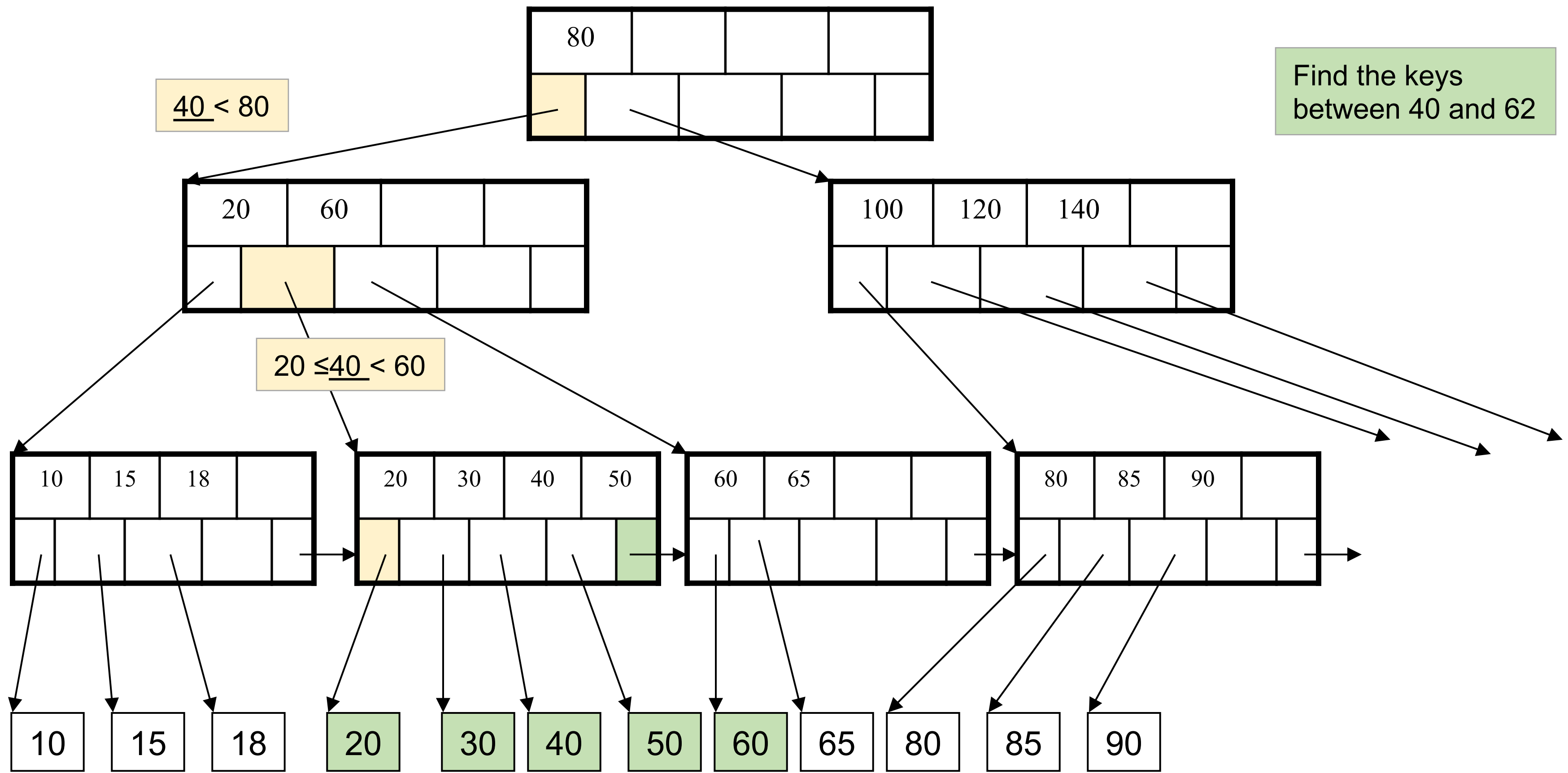
B+ Tree Example

d = 2



B+ Tree Example

d = 2



Find the keys between 40 and 62

B+ Trees Properties

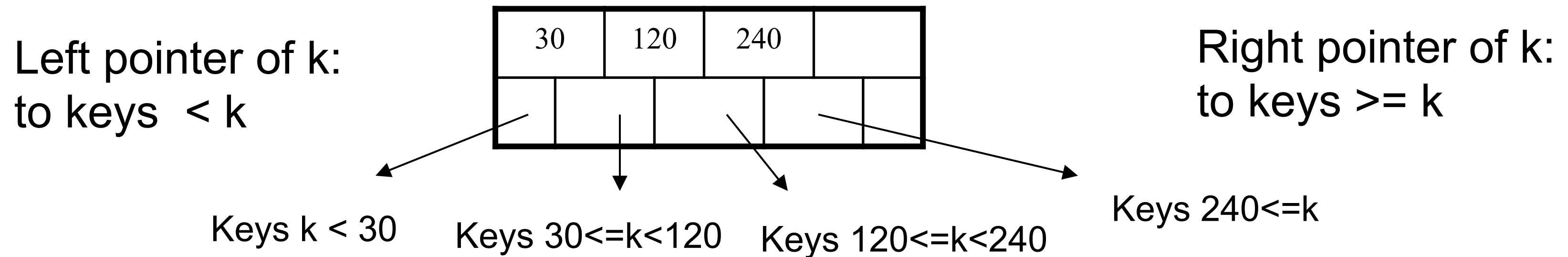
- For each node except the root, maintain 50% occupancy of keys
- Insert and delete must rebalance to maintain constraints (see the last part of this slide deck)

B+ Trees Properties

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

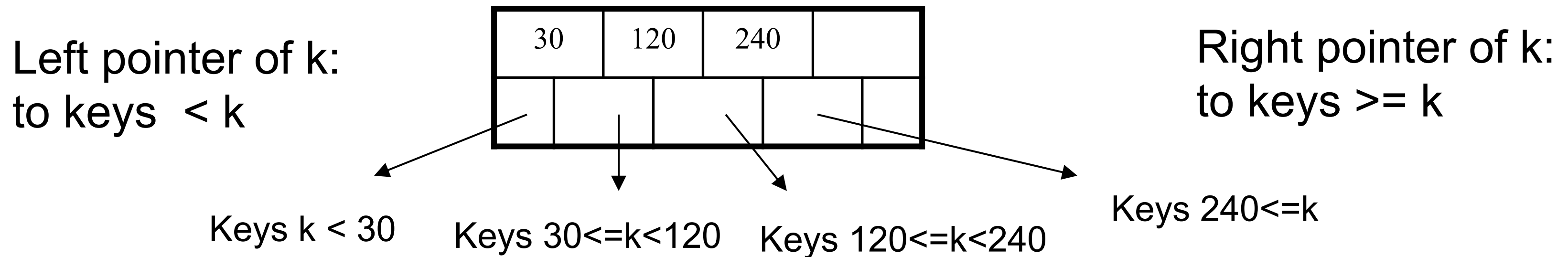
B+ Trees Properties

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

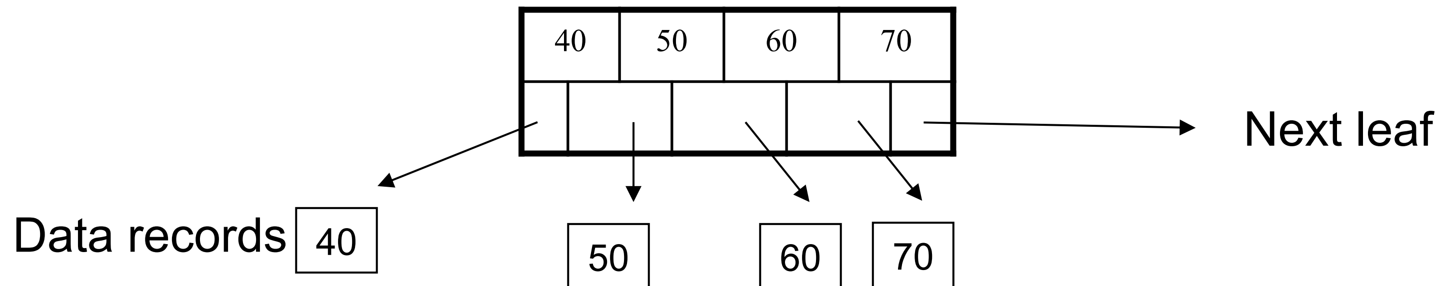


B+ Trees Properties

- 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**:



Search time

A B+ tree is perfectly balanced

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

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

- Binary search tree: log N = 30
- B+ tree, assume m=128: $\log_m N$ = 4 block reads

You can find the data with only 4 block reads!

Indexes in SQL

Index in SQL

```
CREATE TABLE Payroll(UserID int, name text, job text, salary int)
```

Index in SQL

```
CREATE TABLE Payroll(UserID int, name text, job text, salary int)
```

…		
…		

123	Jack	TA
…		
…		

…	…	…
…		
…		

…

…

Payroll data file

Index in SQL

```
CREATE TABLE Payroll(UserID int, name text, job text, salary int)
```

```
CREATE INDEX Pname on Payroll(name)
```

…		
…		

123	Jack	TA
…		
…		

…	…	…
…		
…		

…

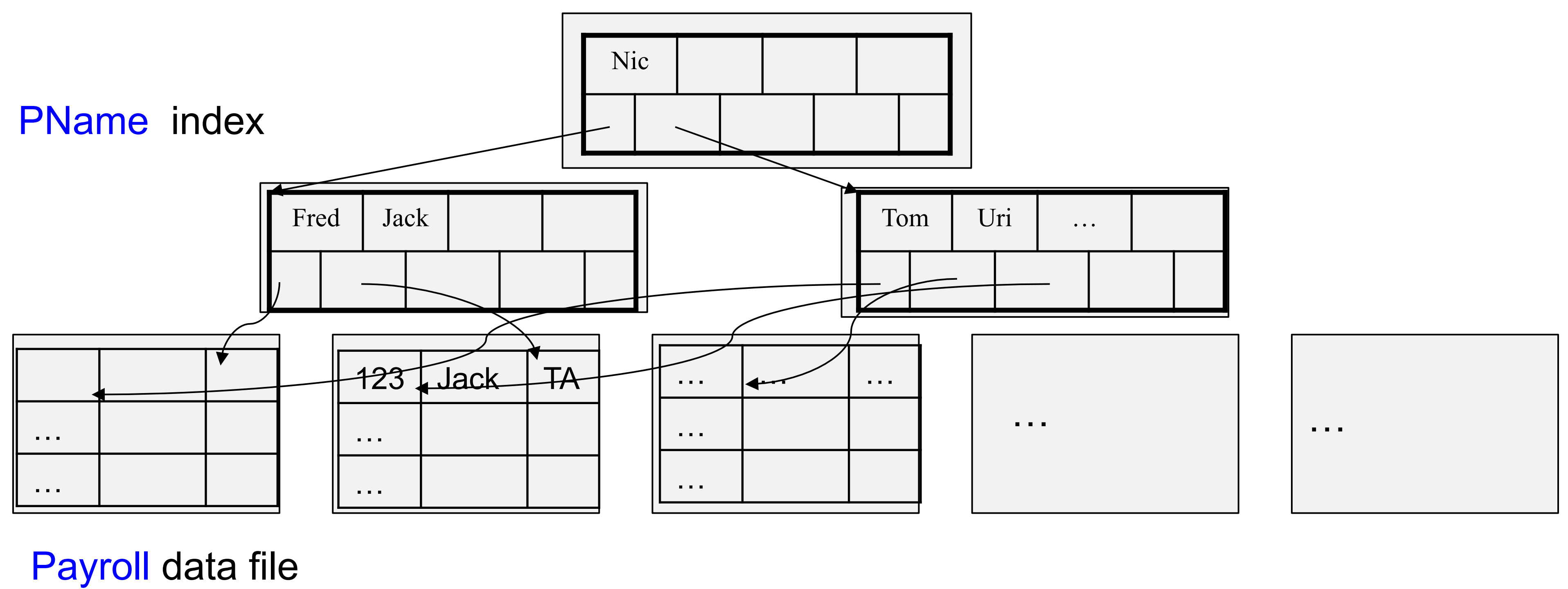
…

Payroll data file

Index in SQL

```
CREATE TABLE Payroll(UserID int, name text, job text, salary int)
```

```
CREATE INDEX Pname on Payroll(name)
```



Index in SQL

```
CREATE TABLE Payroll(UserID int, name text, job text, salary int)
```

```
CREATE INDEX Pname on Payroll(name)
```

Index in SQL

```
CREATE TABLE Payroll(UserID int, name text, job text, salary int)
```

```
CREATE INDEX Pname on Payroll(name)
```

```
SELECT *
FROM Payroll
WHERE name = 'Allison';
```

Index in SQL

```
CREATE TABLE Payroll(UserID int, name text, job text, salary int)
```

```
CREATE INDEX Pname on Payroll(name)
```

```
SELECT *
FROM Payroll
WHERE name = ‘Allison’;
```

This query will execute faster after creating the index

Index in SQL

```
CREATE TABLE Payroll(UserID int, name text, job text, salary int)
```

```
CREATE INDEX Pname on Payroll(name)
```

```
SELECT *
FROM Payroll
WHERE name = 'Allison';
```

This query will execute faster after creating the index

```
SELECT *
FROM Payroll x, Regist y
WHERE x.UserID = y.UserID;
```

The index won't help this query

Discussion

- We can create many indexes for a table
- But one query can only use one index
- Each insert/update/delete updates ALL indices

Choosing which indices to create is a difficult database administration task

Create Index

```
CREATE TABLE Payroll(UserID int, name text, job text, salary int)
```

Payroll:

Create Index

```
CREATE TABLE Payroll(UserID int, name text, job text, salary int)
```

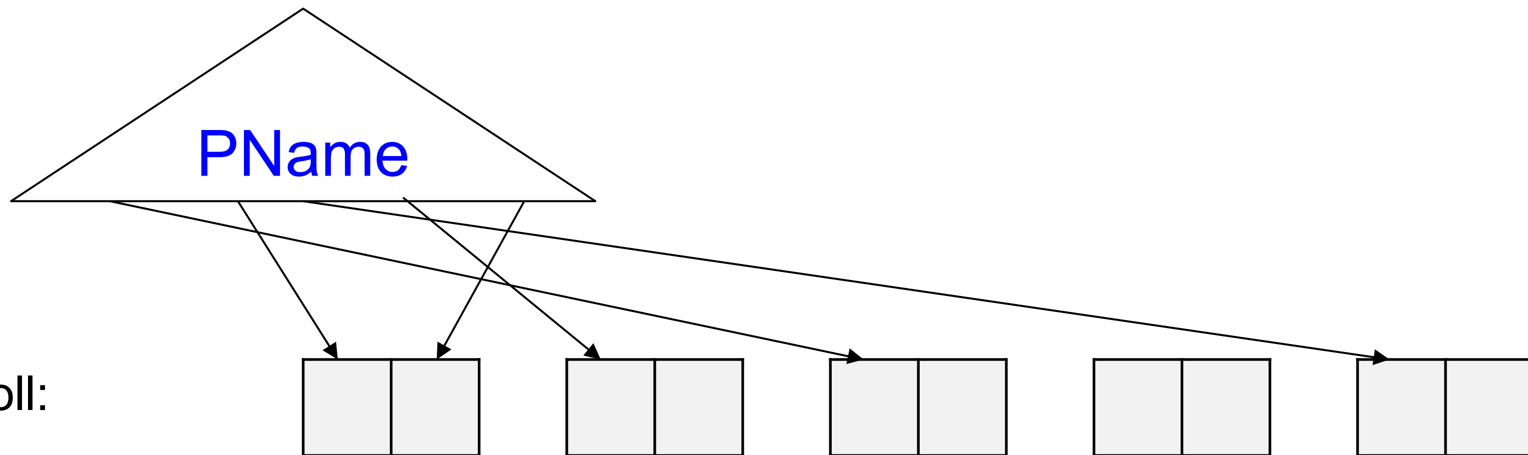
```
CREATE INDEX Pname on Payroll(name);
```

Payroll:

Create Index

```
CREATE TABLE Payroll(UserID int, name text, job text, salary int)
```

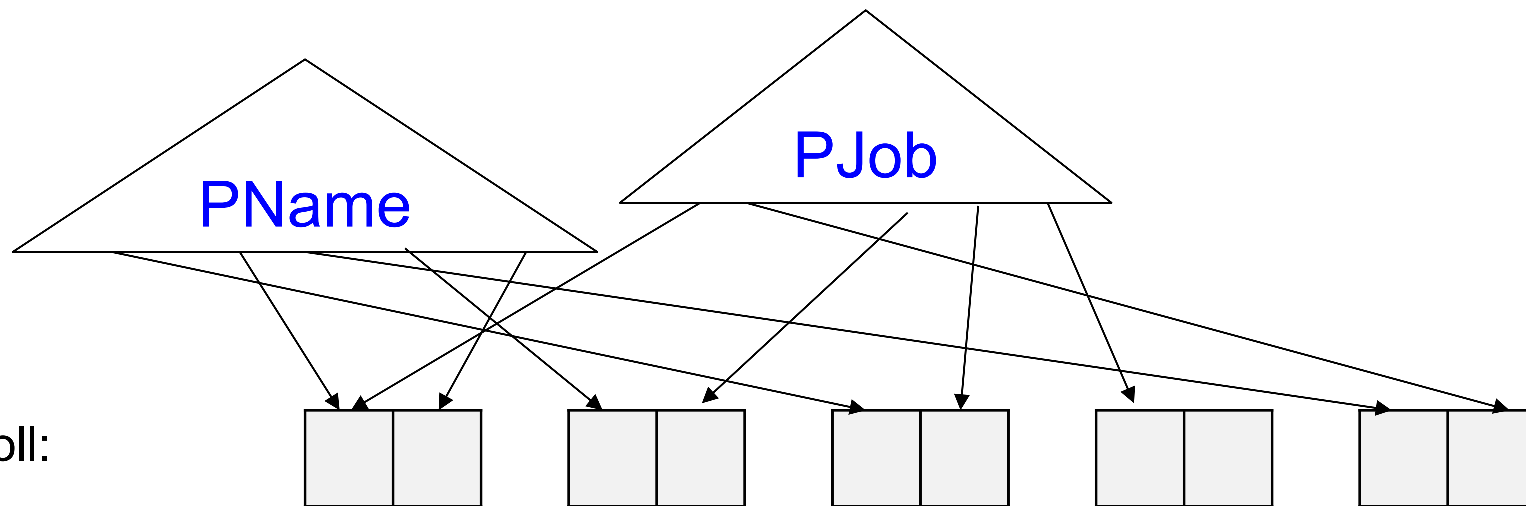
```
CREATE INDEX Pname on Payroll(name);
```



Create Index

```
CREATE TABLE Payroll(UserID int, name text, job text, salary int)
```

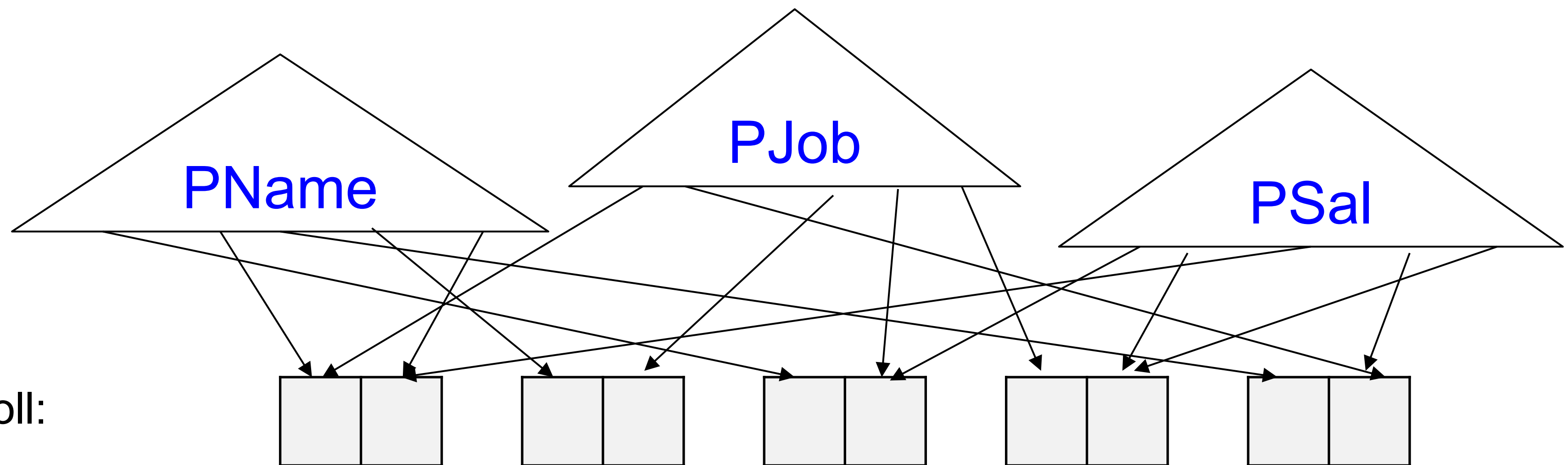
```
CREATE INDEX Pname on Payroll(name);
CREATE INDEX Pjob  on Payroll(job);
```



Create Index

```
CREATE TABLE Payroll(UserID int, name text, job text, salary int)
```

```
CREATE INDEX Pname on Payroll(name);
CREATE INDEX Pjob  on Payroll(job);
CREATE INDEX Psal  on Payroll(salary);
```

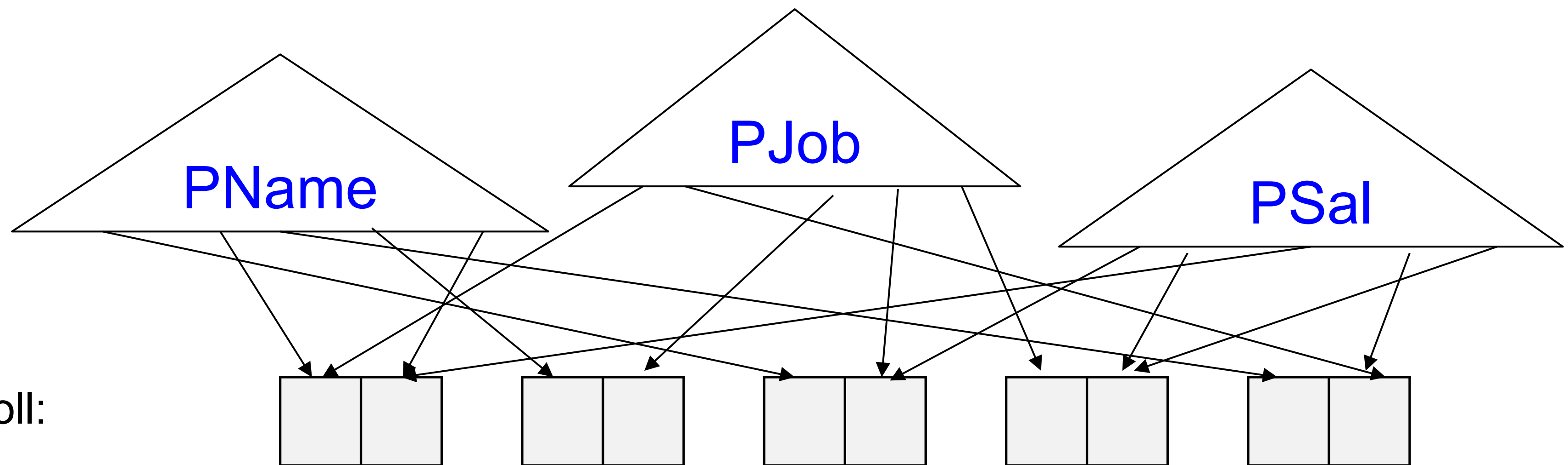


Create Index

```
CREATE TABLE Payroll(UserID int, name text, job text, salary int)
```

```
CREATE INDEX Pname on Payroll(name);
CREATE INDEX Pjob  on Payroll(job);
CREATE INDEX Psal  on Payroll(salary);
```

```
SELECT *
FROM Payroll
WHERE job='TA'
 and salary=50000;
```



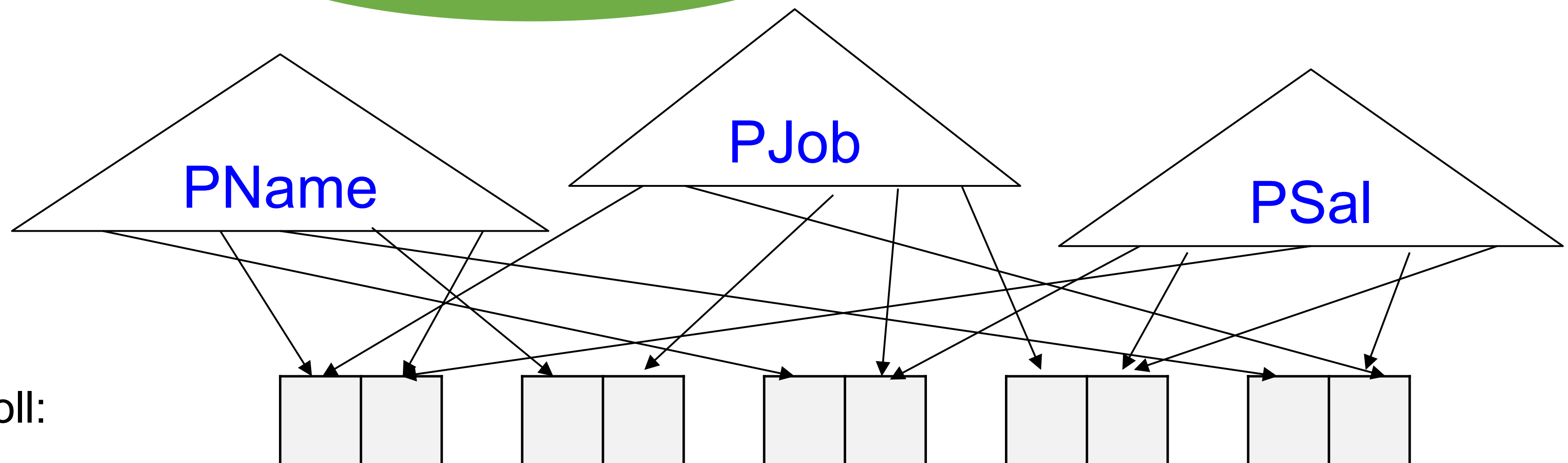
Create Index

```
CREATE TABLE Payroll(UserID int, name text, job text, salary int)
```

```
CREATE INDEX Pname on Payroll(name);
CREATE INDEX Pjob  on Payroll(job);
CREATE INDEX Psal  on Payroll(salary);
```

```
SELECT *
FROM Payroll
WHERE job='TA'
 and salary=50000;
```

Optimizer may choose index on Job



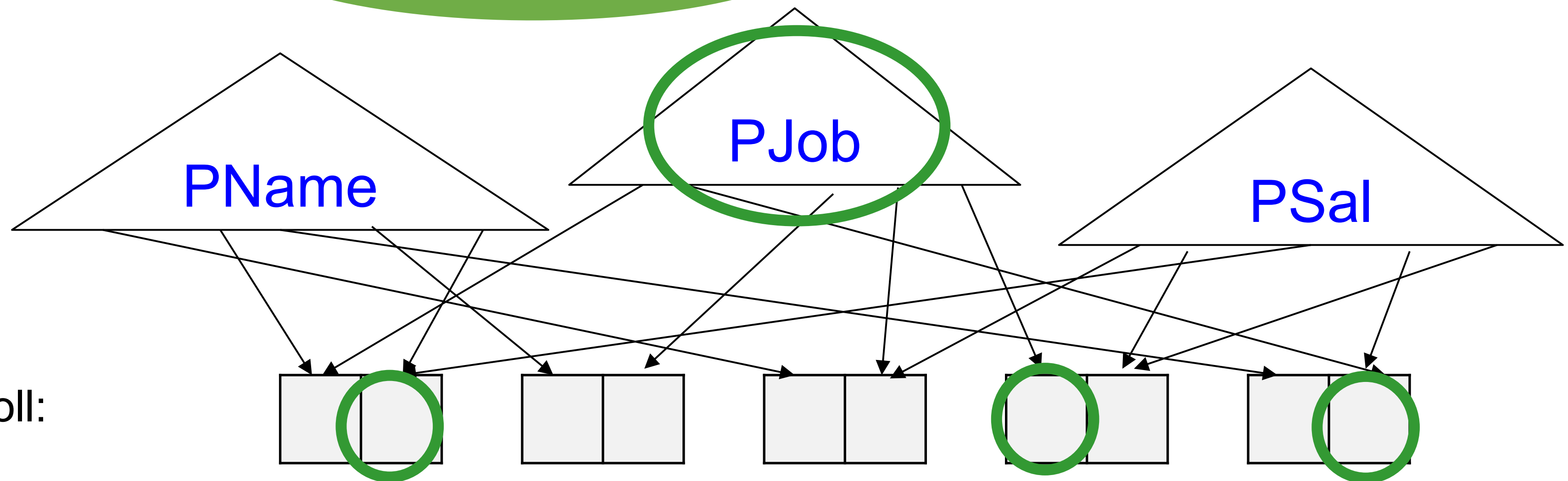
Create Index

```
CREATE TABLE Payroll(UserID int, name text, job text, salary int)
```

```
CREATE INDEX Pname on Payroll(name);
CREATE INDEX Pjob  on Payroll(job);
CREATE INDEX Psal  on Payroll(salary);
```

```
SELECT *
FROM Payroll
WHERE job='TA'
 and salary=50000;
```

Optimizer may choose index on Job

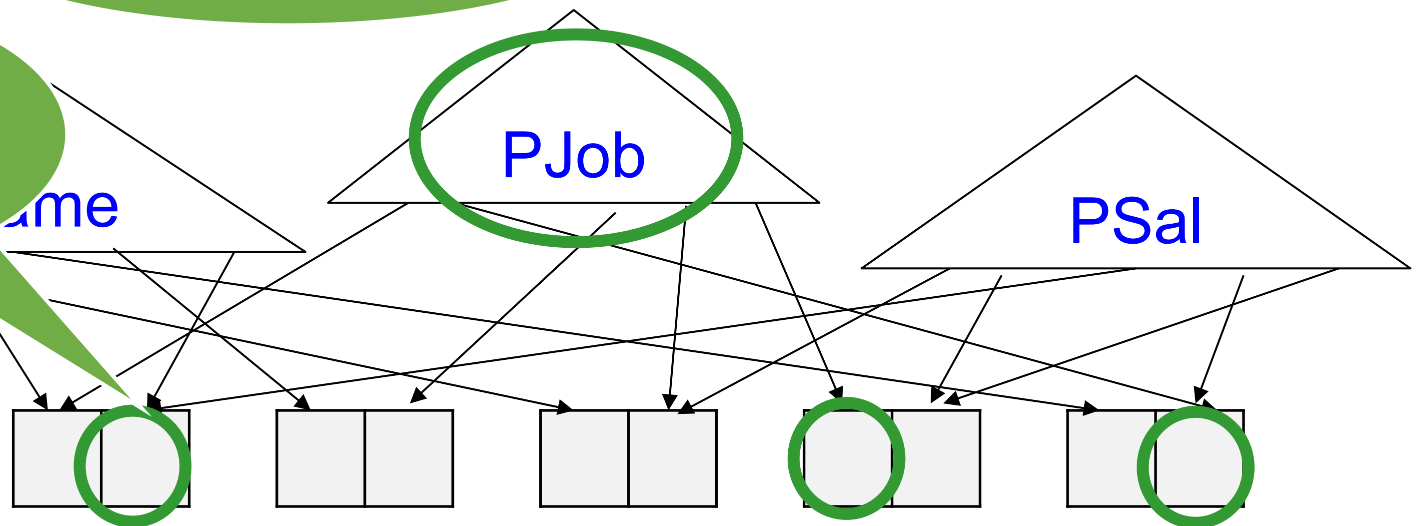


Create Index

```
CREATE TABLE Payroll(UserID int, name text, job text, salary int)
```

```
CREATE INDEX Pname on Payroll(name);
CREATE INDEX Pjob  on Payroll(job);
CREATE INDEX Psal  on Payroll(salary);
```

```
SELECT *
FROM Payroll
WHERE job='TA'
 and salary=50000;
```



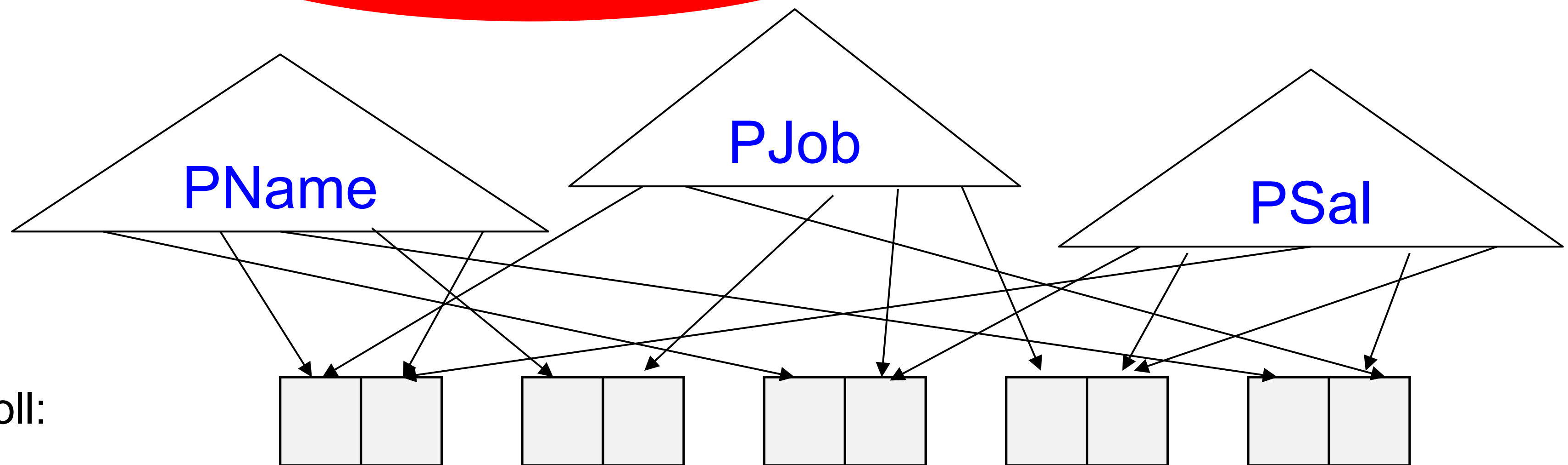
Create Index

```
CREATE TABLE Payroll(UserID int, name text, job text, salary int)
```

```
CREATE INDEX Pname on Payroll(name);
CREATE INDEX Pjob  on Payroll(job);
CREATE INDEX Psal  on Payroll(salary);
```

```
SELECT *
FROM Payroll
WHERE job='TA'
 and salary=50000;
```

…or optimizer may choose index on Salary



Create Index

```
CREATE TABLE Payroll(UserID int, name text, job text, salary int)
```

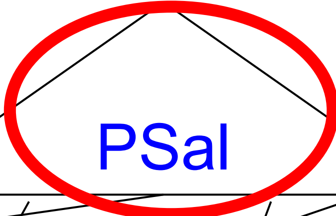
```
CREATE INDEX Pname on Payroll(name);
CREATE INDEX Pjob  on Payroll(job);
CREATE INDEX Psal  on Payroll(salary);
```

```
SELECT *
FROM Payroll
WHERE job='TA'
 and salary=50000;
```

…or optimizer may choose index on Salary

PName

PJob



P

Must check job='TA'

Discussion

- In general there can be multiple indexes available to compute a where-condition:
  - Attr1=val1 and Attr2=val2 and …
- Optimizer needs to choose one, then iterate over the answers and filter out the other conditions
- This problem is called access path selection

Multi-attribute Index

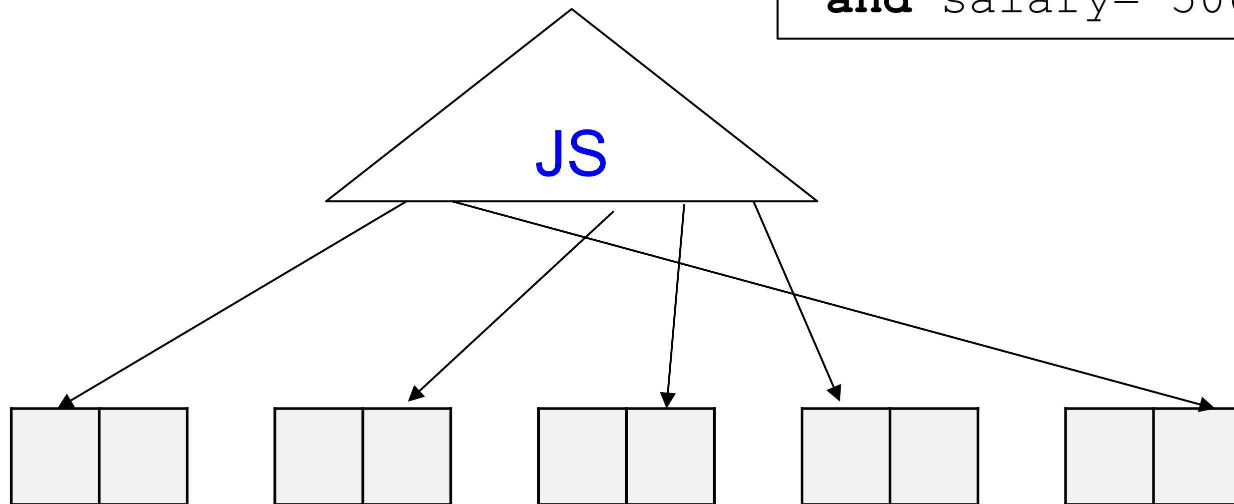
- An index can be on multiple attributes: A, B, C, …
- Values are concatenated, then sorted
- Attribute order matters:
  - Create index on R(A,B,C)
  - Create index on R(C,A,B)

Multi-attribute Index

```
CREATE TABLE Payroll(UserID int, name text, job text, salary int)
```

```
CREATE INDEX JS on Payroll(job,salary);
```

```
SELECT *
FROM Payroll
WHERE job='TA'
 and salary='50000';
```



Payroll:

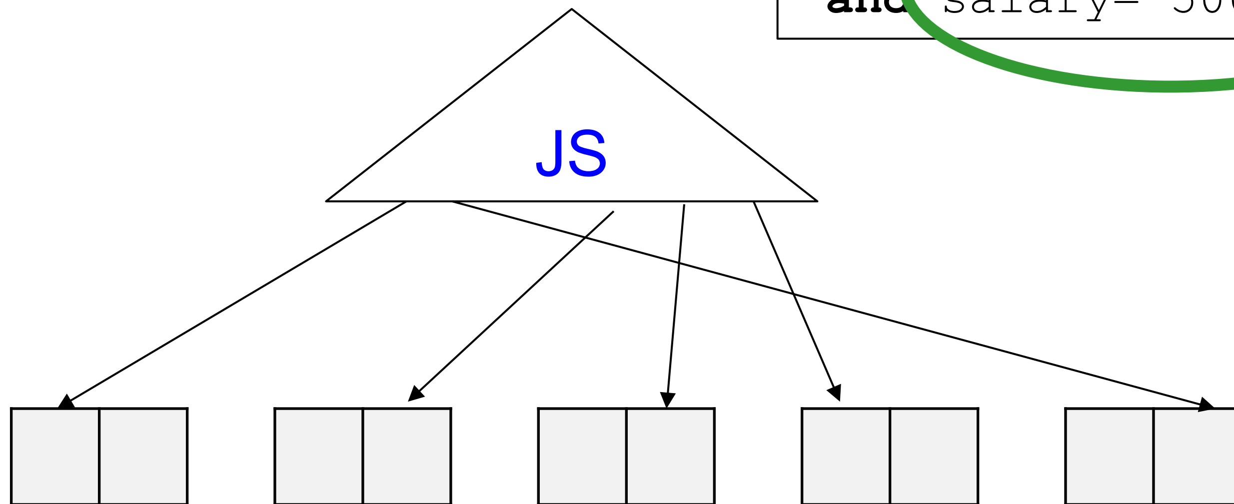
Multi-attribute Index

```
CREATE TABLE Payroll(UserID int, name text, job text, salary int)
```

```
CREATE INDEX JS on Payroll(job,salary);
```

```
SELECT *
FROM Payroll
WHERE job='TA'
 and salary='50000';
```

Payroll:



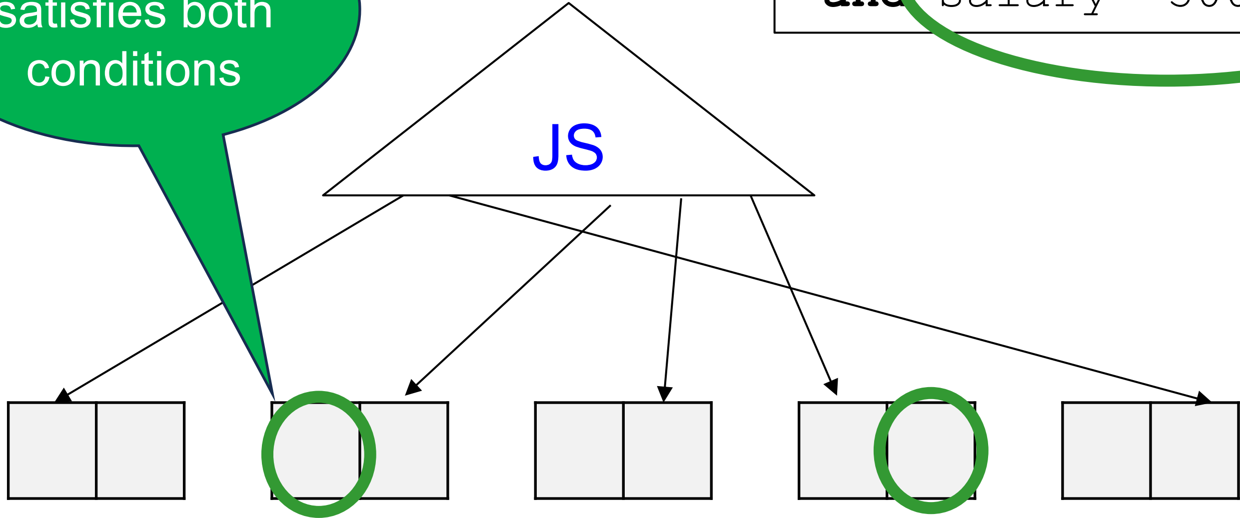
Multi-attribute Index

```
CREATE TABLE Payroll(UserID int, name text, job text, salary int)
```

```
CREATE INDEX JS on Payroll(job,salary);
```

```
SELECT *
FROM Payroll
WHERE job='TA'
 and salary='50000';
```

Payroll:



Multi-attribute Index

- A pair ordered lexicographically:
  - (‘Director’,200000) < (‘Prof’,50000)<(‘Prof’,200000)<…
- Only 1st attribute known: range seach
  - Find (‘Prof’, *)
- Only 2nd attribute known: impossible
  - Find (*, 200000)

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

parent

K2	K3	K5	

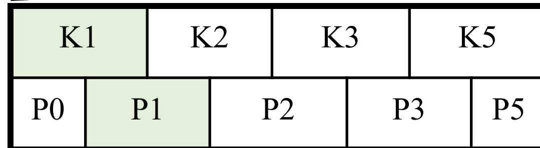
P0	P2	P3	P5	

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

parent

K1	K2	K3	K5

P0	P1	P2	P3	P5

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

parent

K1	K2	K3	K5

P0	P1	P2	P3	P5

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

parent

K1 K2 K3 K4 K5

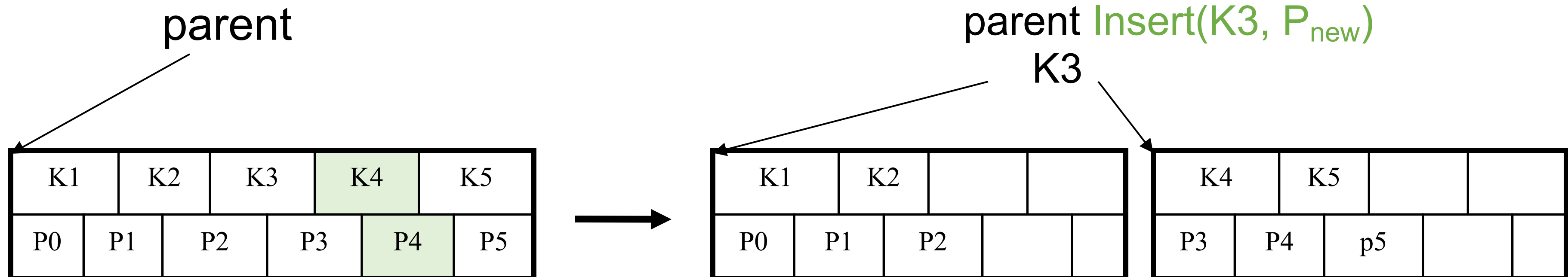
P0 P1 P2 P3 P4 P5

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

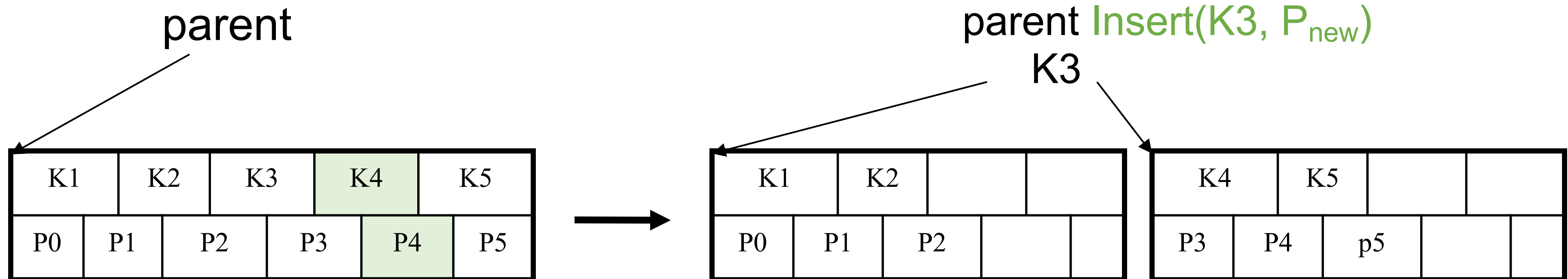


Insertion in a B+ Tree: Summary

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:

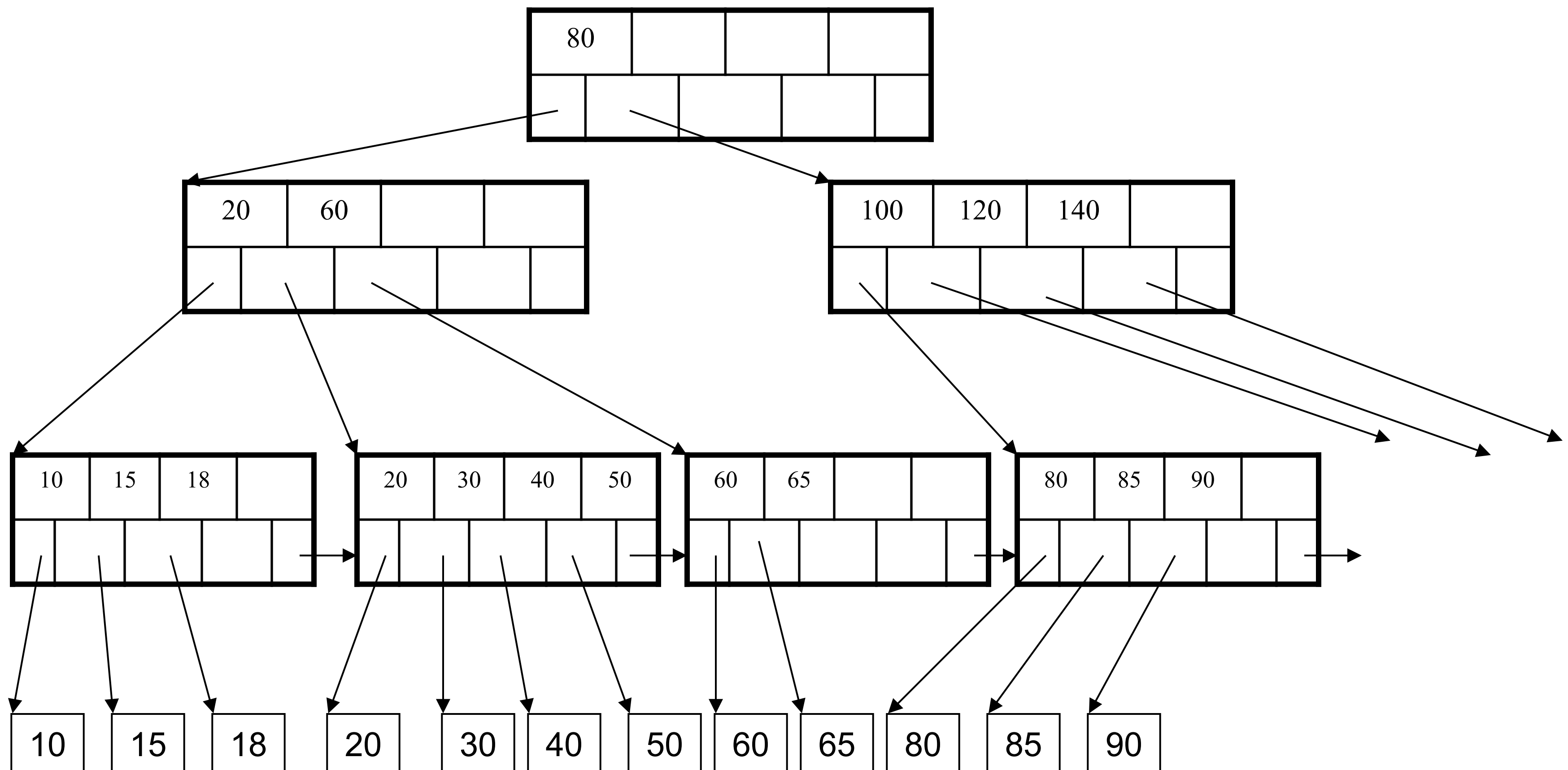
Insert k4



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

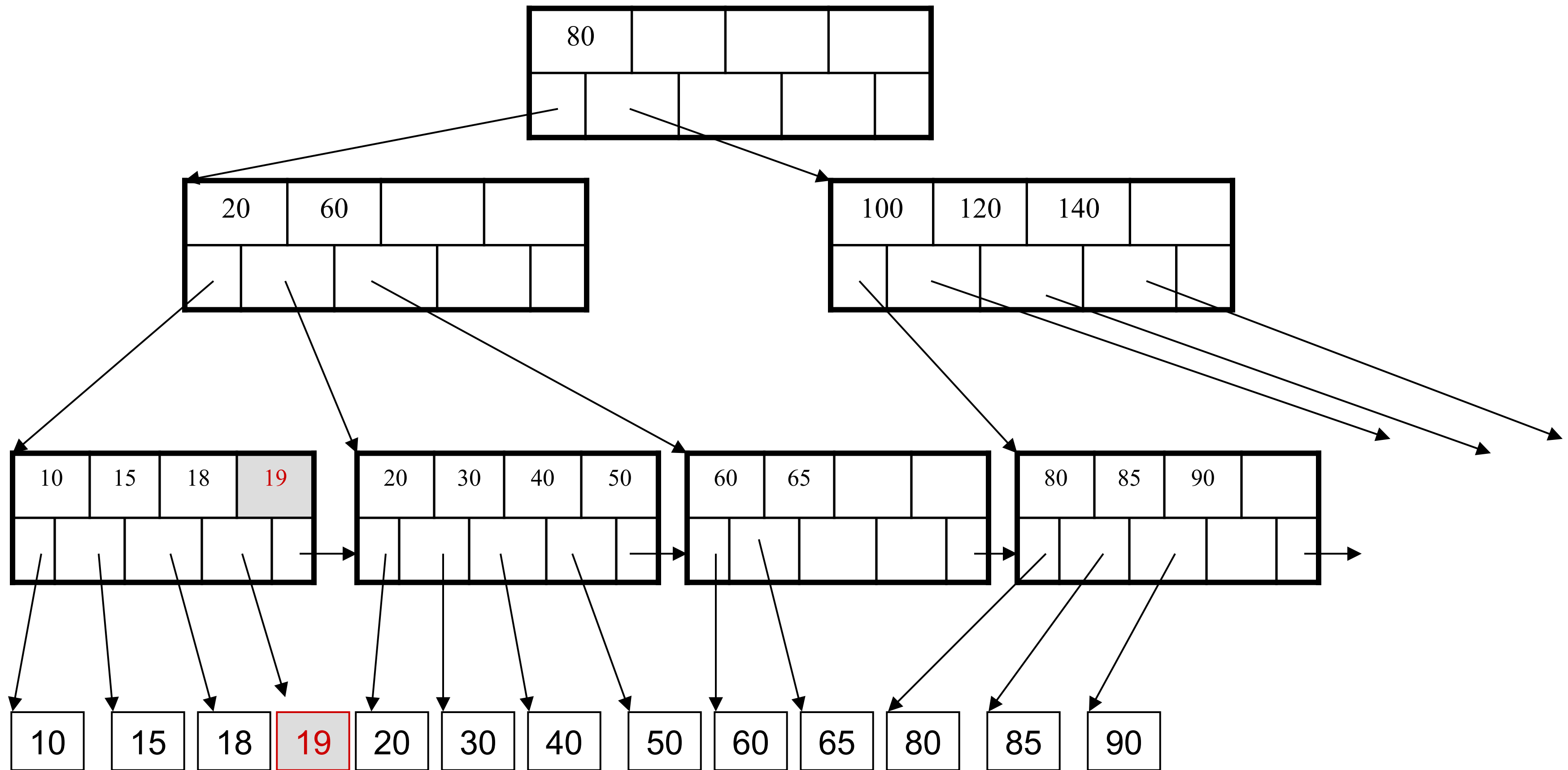
Insertion in a B+ Tree: Example

Insert K=19



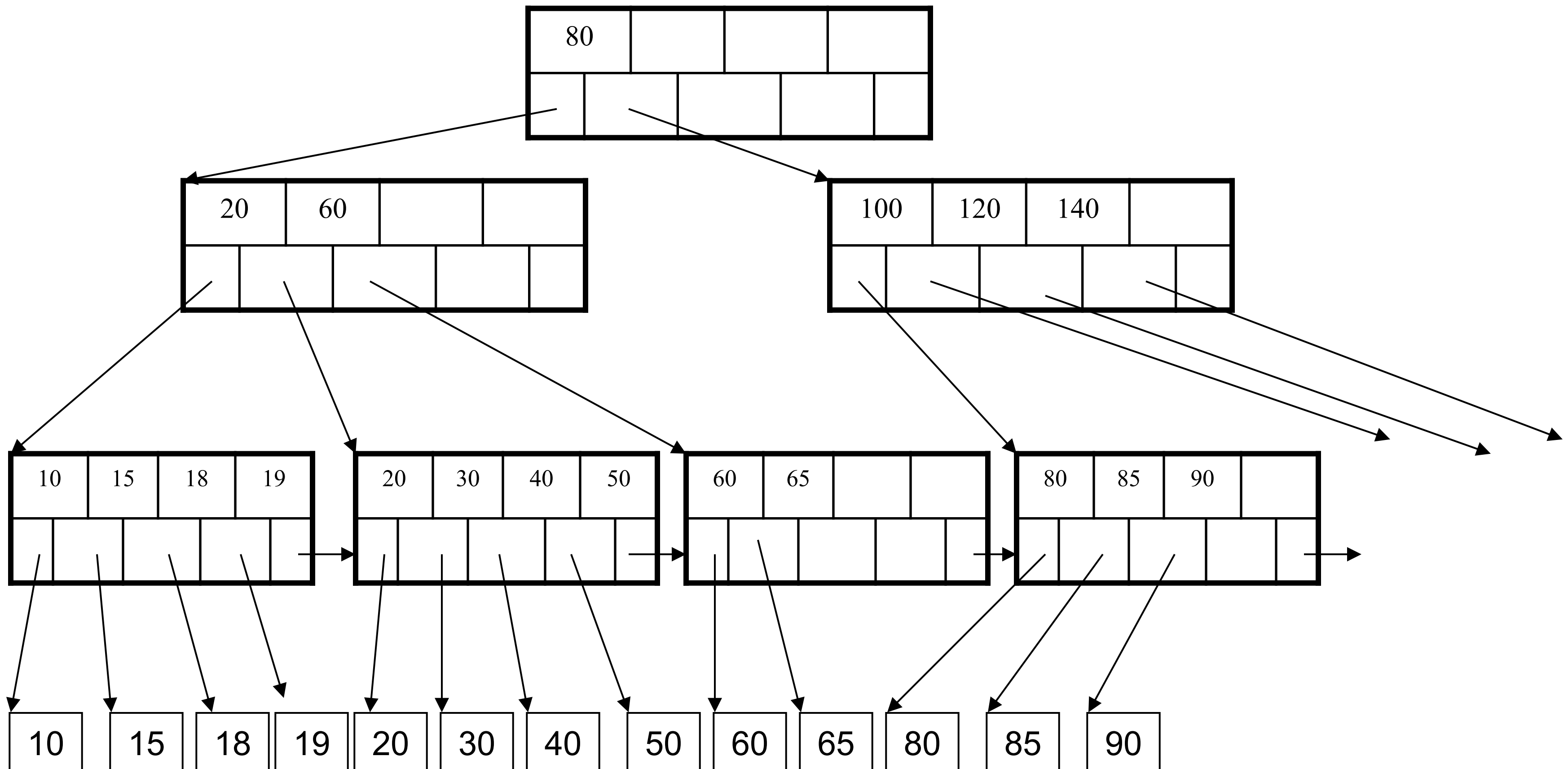
Insertion in a B+ Tree: Example

After insertion



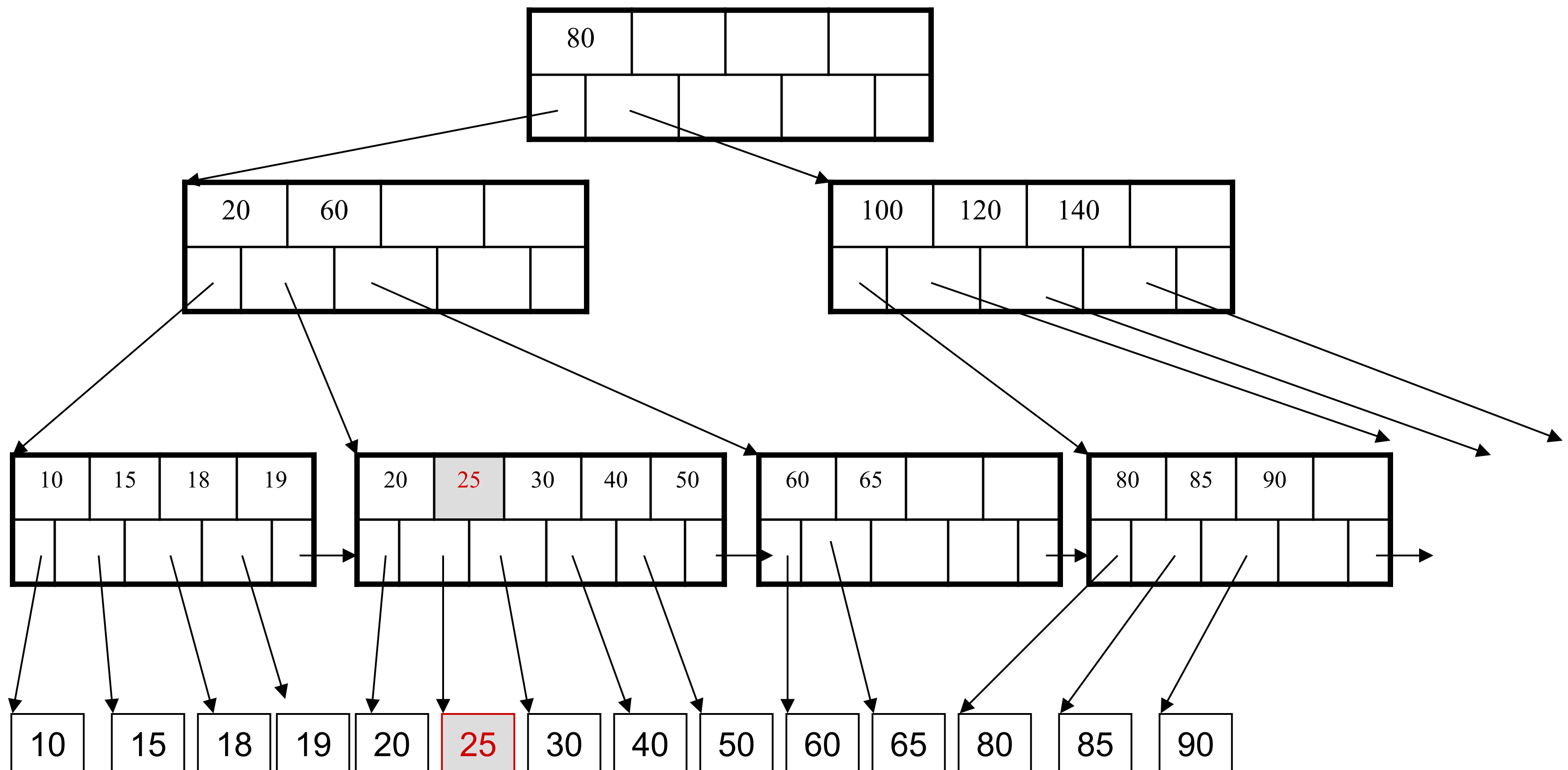
Insertion in a B+ Tree: Example

Now insert 25



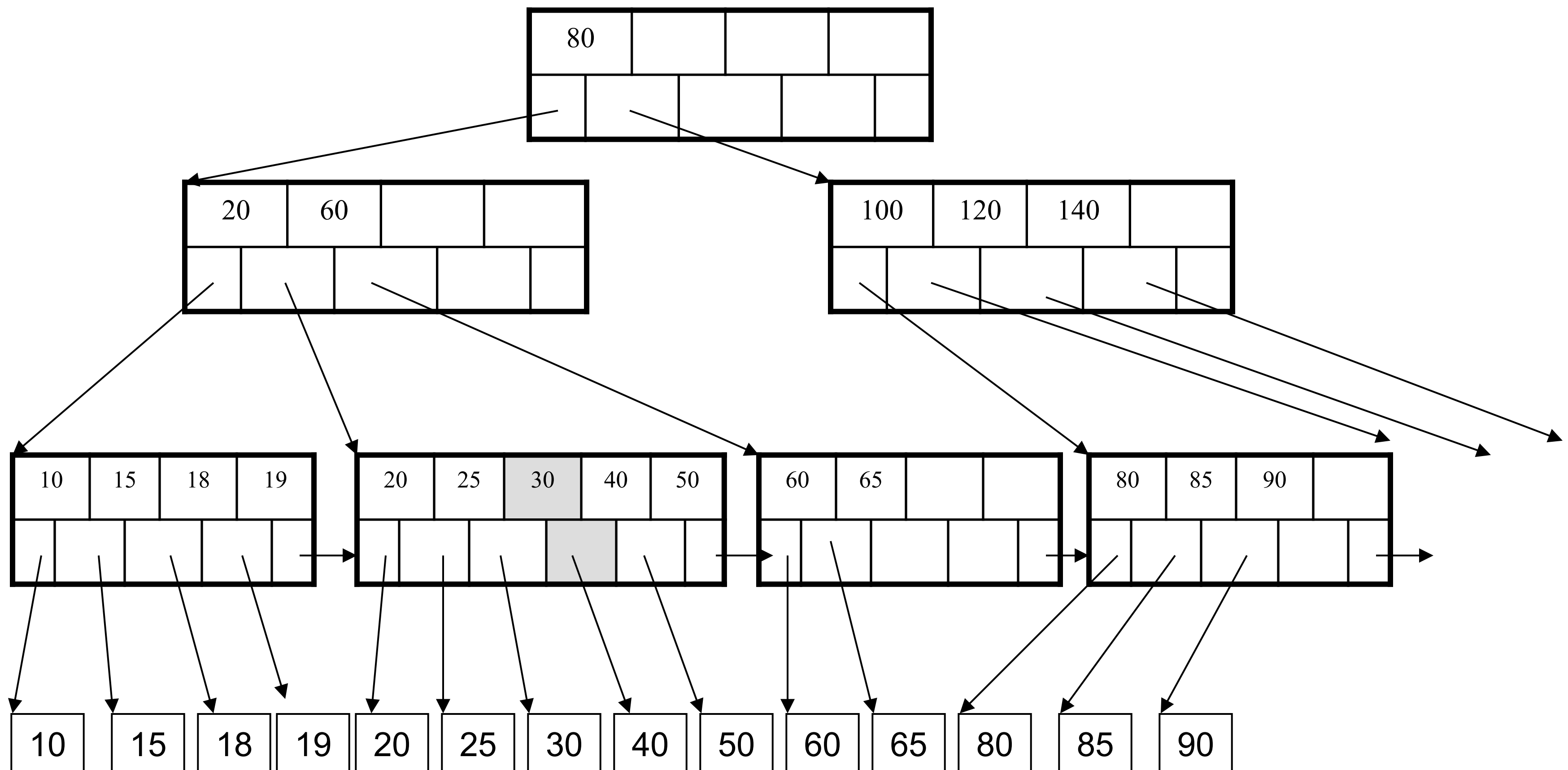
Insertion in a B+ Tree: Example

After insertion



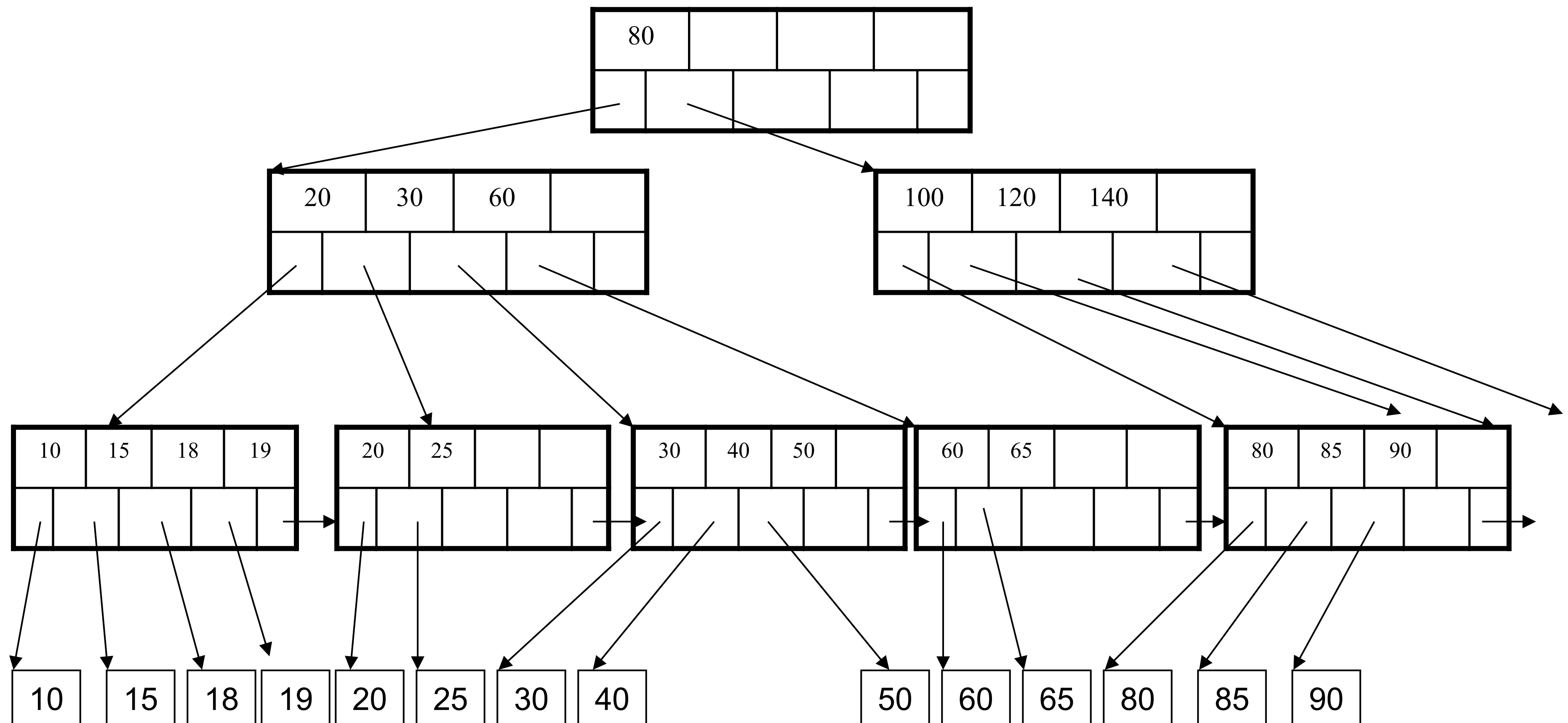
Insertion in a B+ Tree: Example

But now have to split !



Insertion in a B+ Tree: Example

After the split



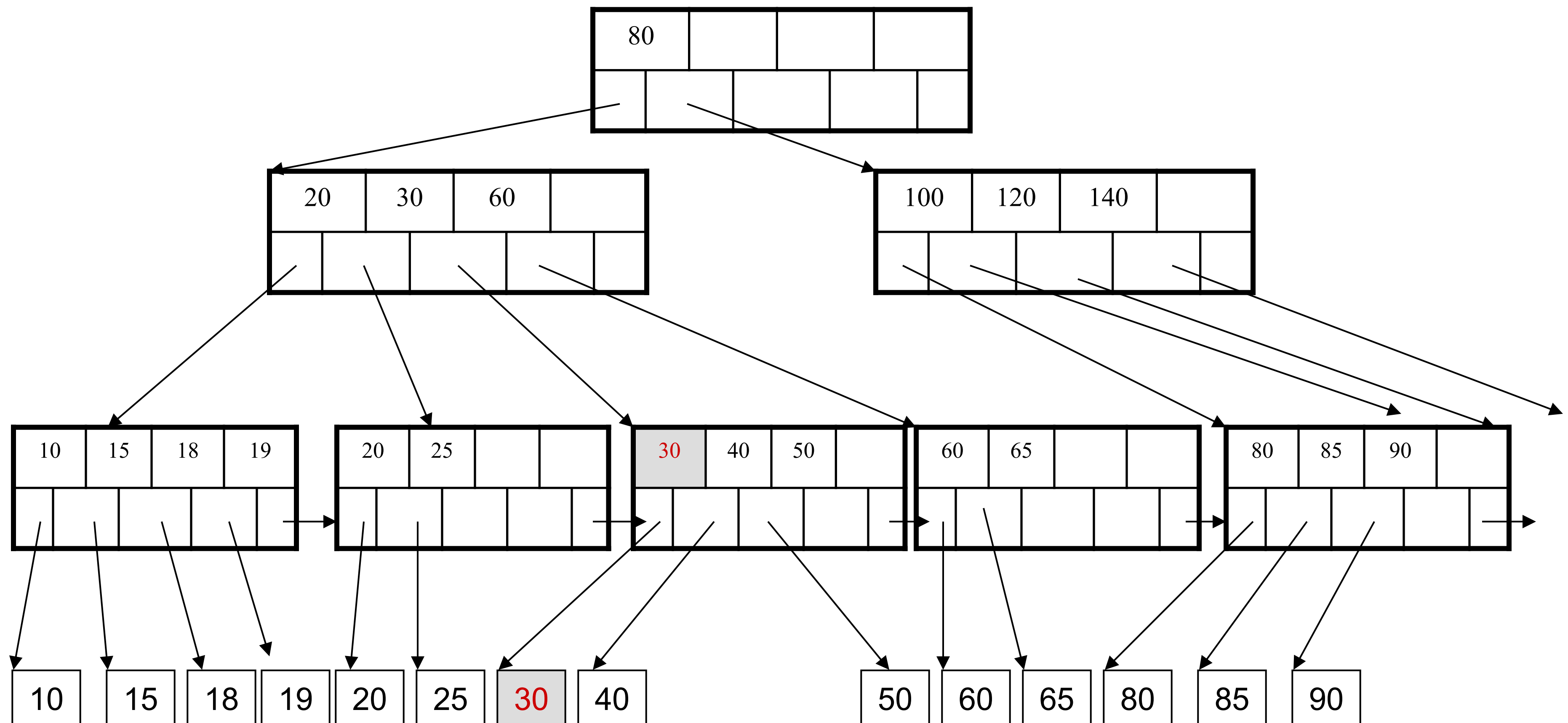
Deletion: Summary

Delete (K, P)

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

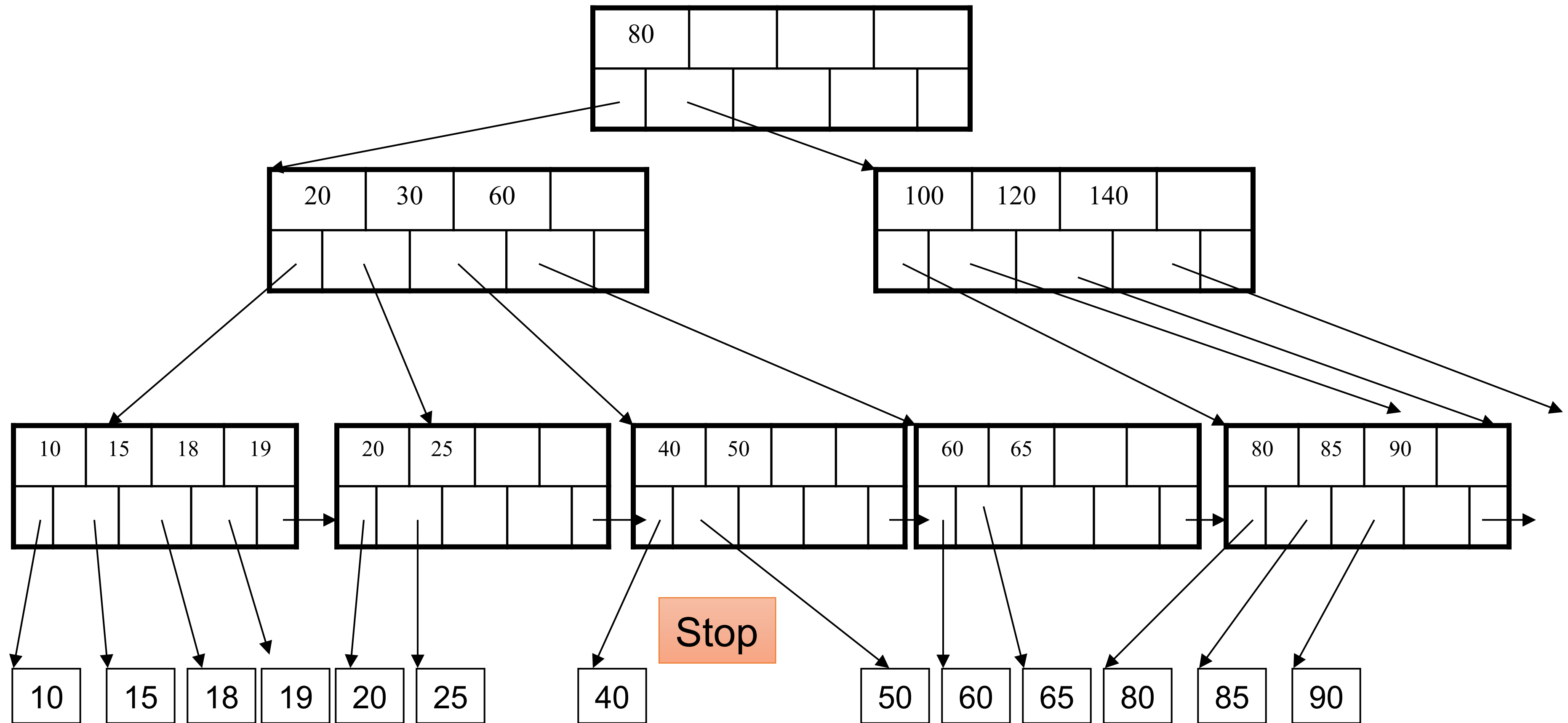
Deletion from a B+ Tree: Example

Delete 30



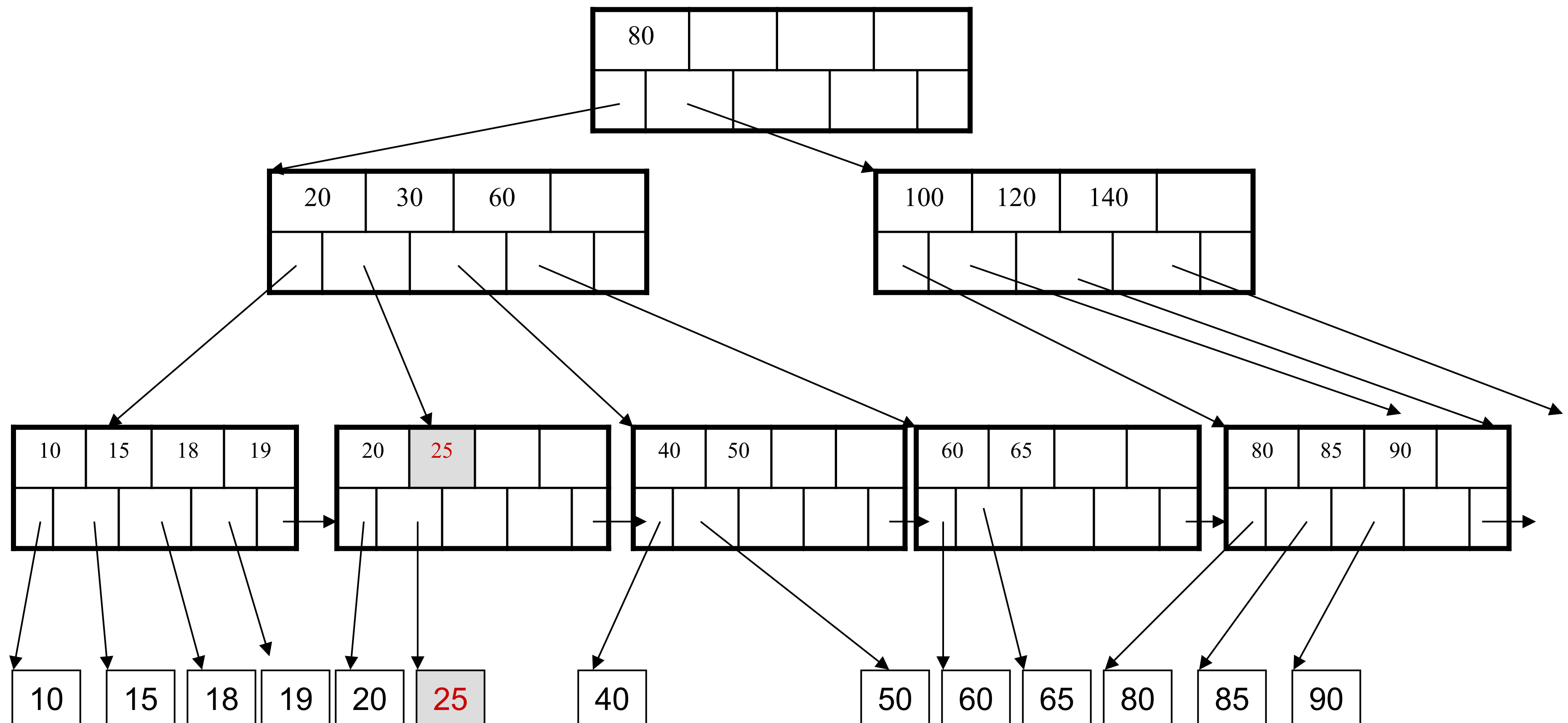
Deletion from a B+ Tree: Example

After deleting 30



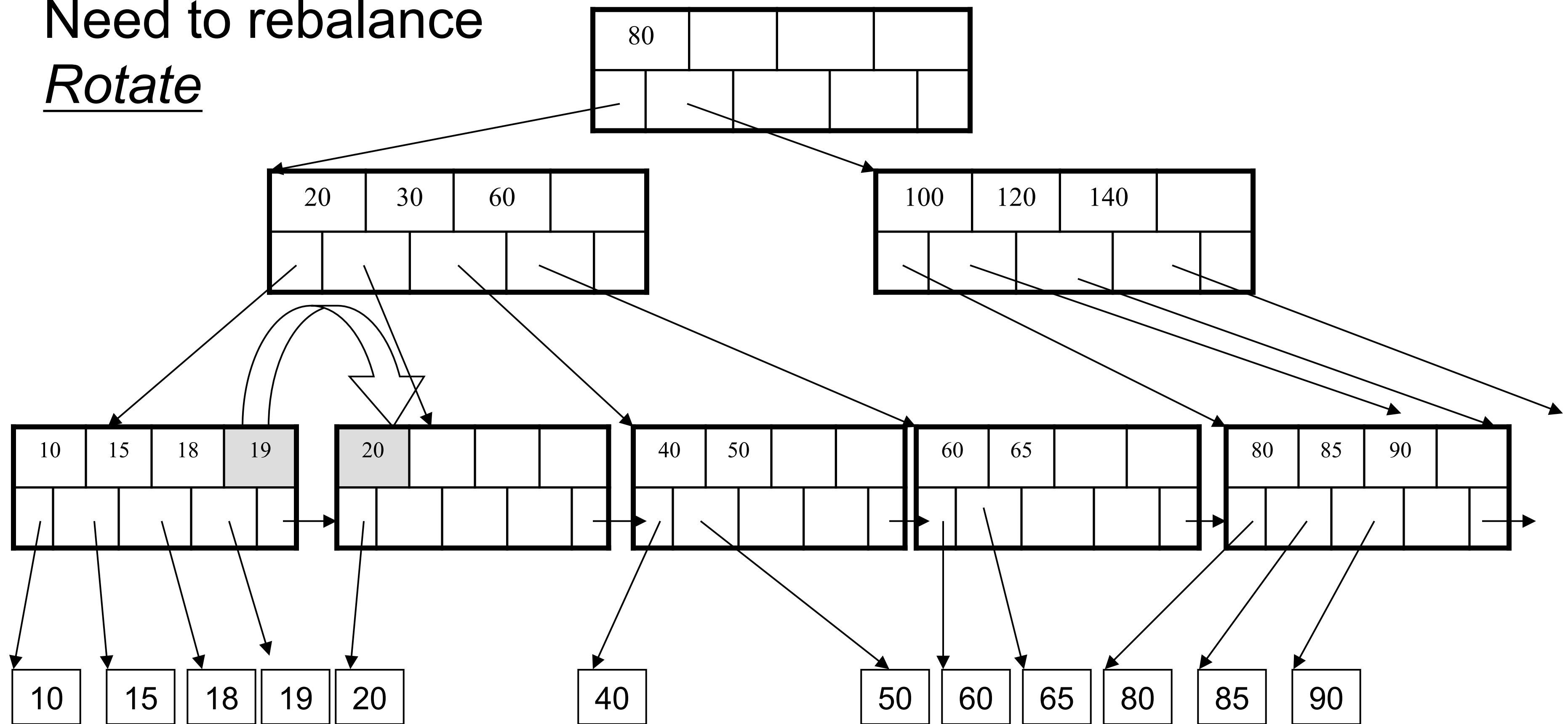
Deletion from a B+ Tree: Example

Now delete 25

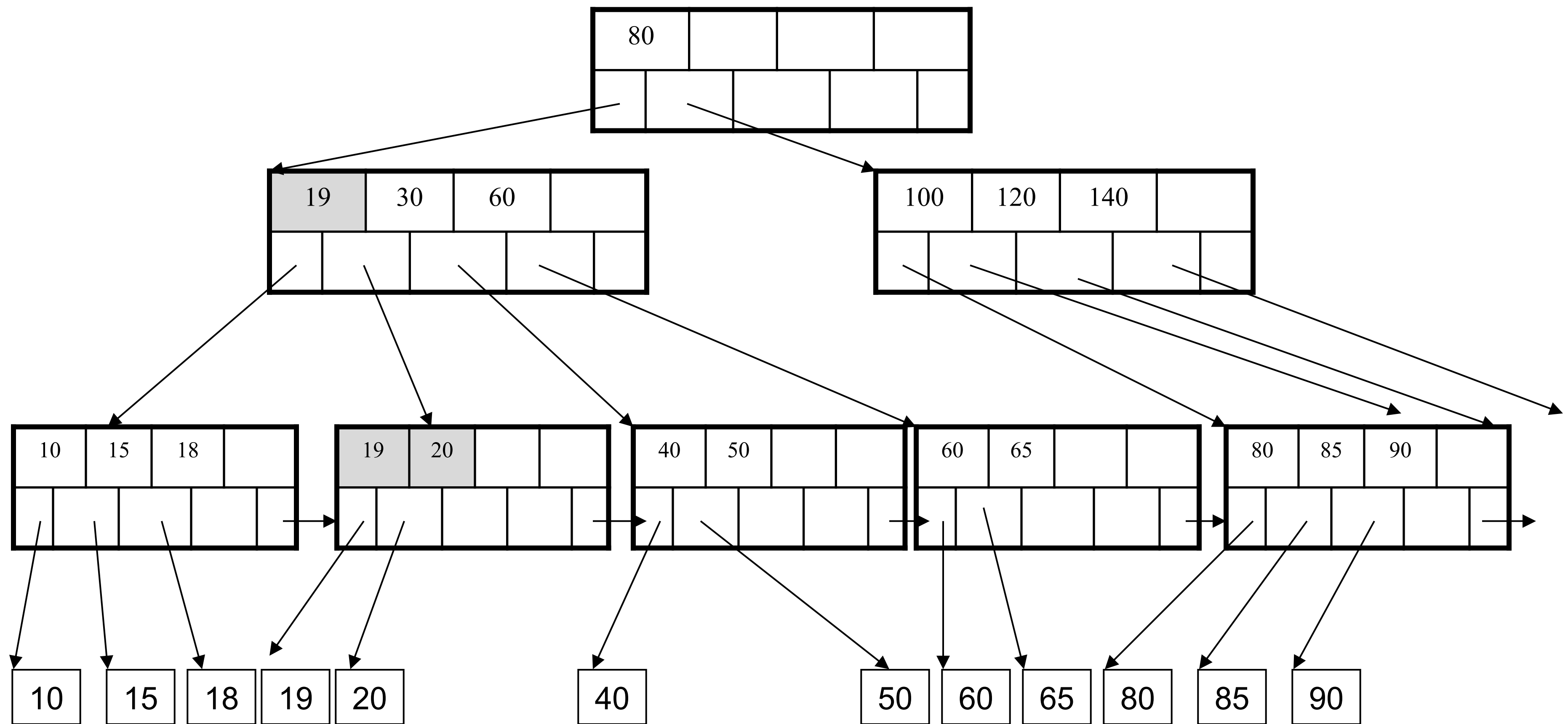


Deletion from a B+ Tree: Example

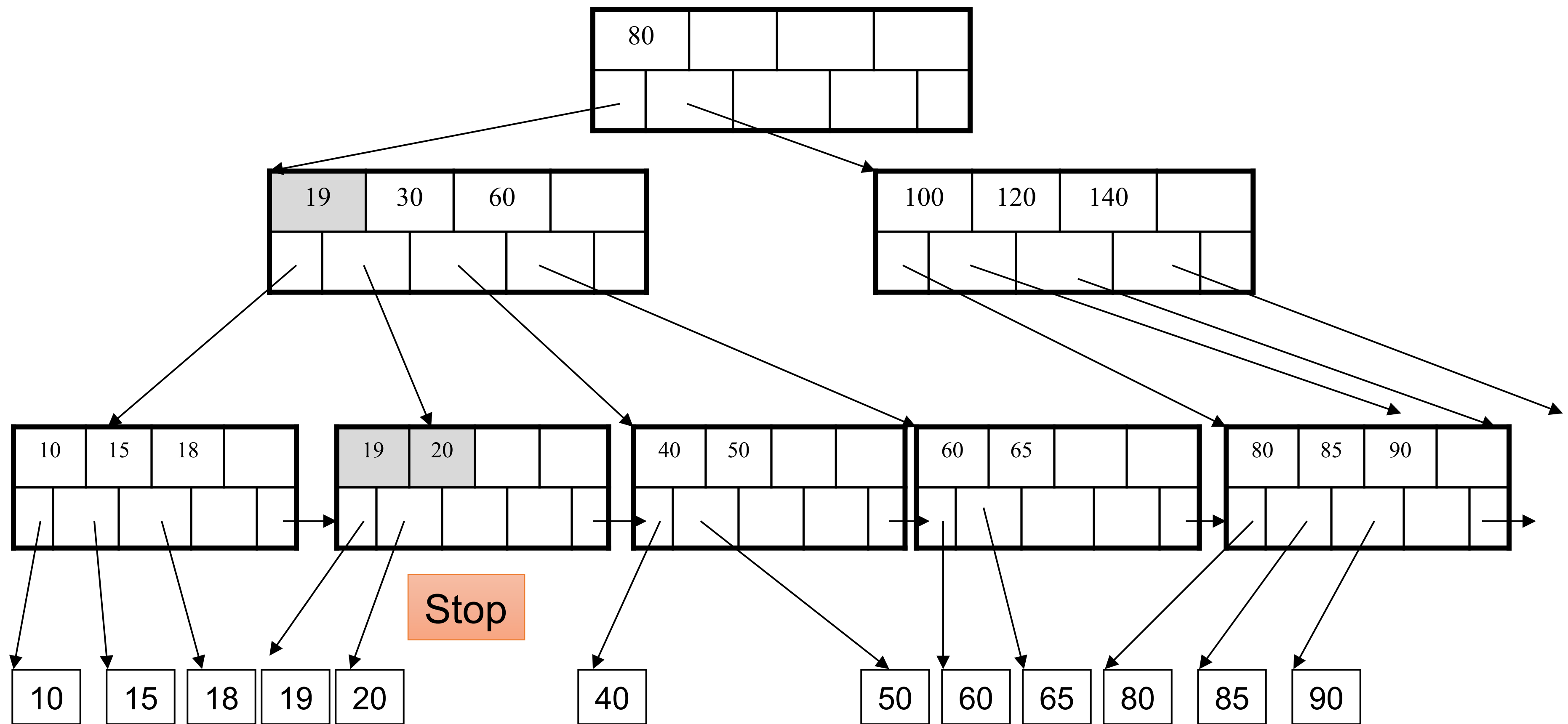
After deleting 25
Need to rebalance
Rotate



Deletion from a B+ Tree: Example

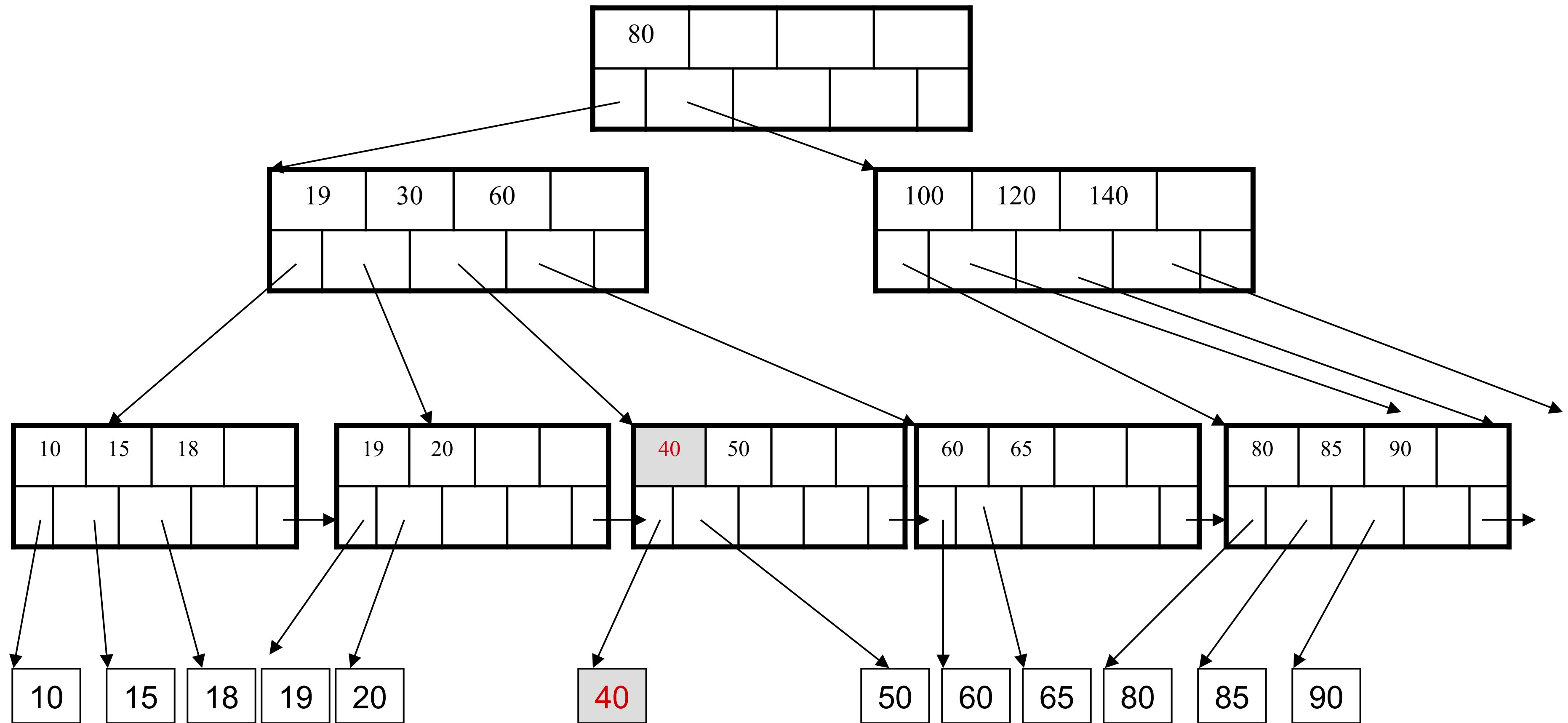


Deletion from a B+ Tree: Example



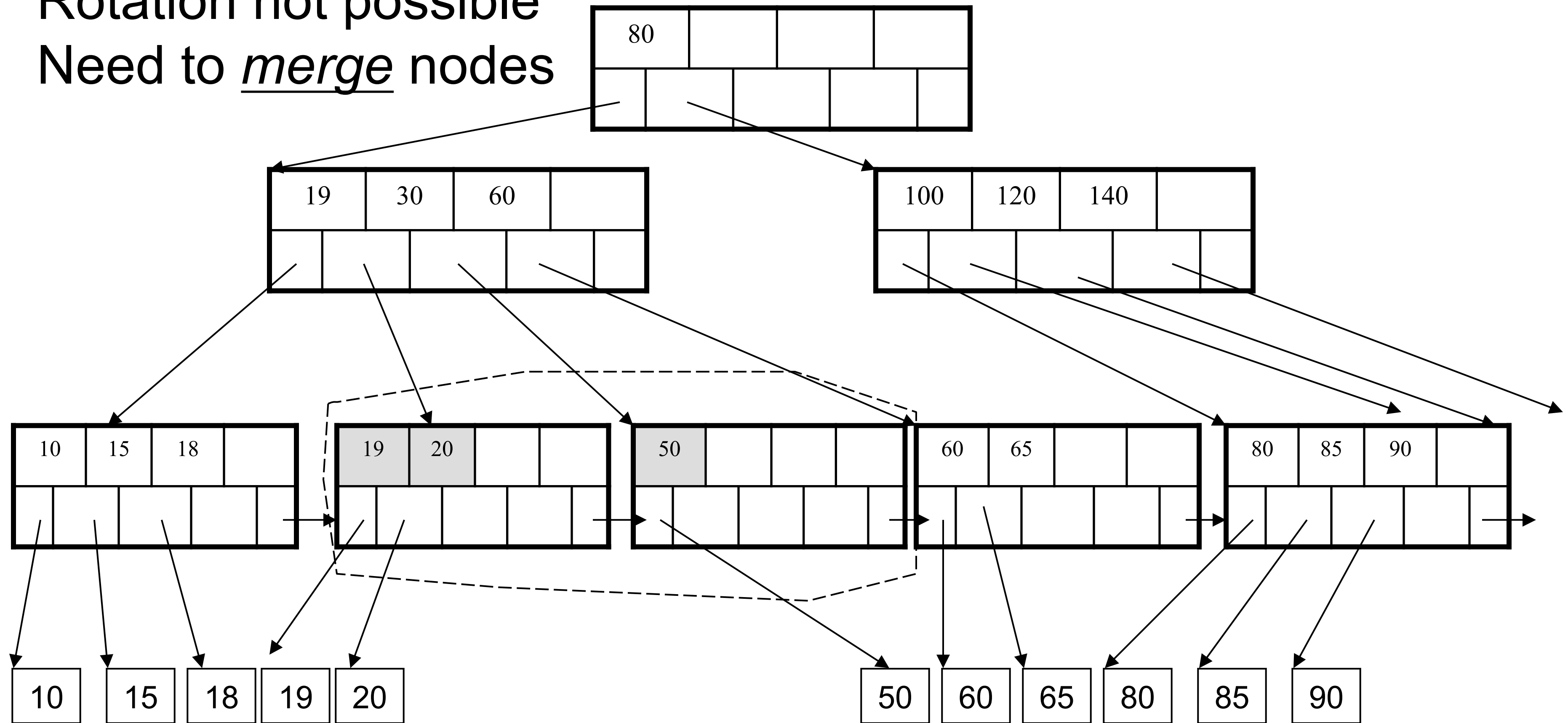
Deletion from a B+ Tree: Example

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