CSE544
Data Management

Lectures 6: Indexes

Announcements

- HW1 due tonight.
- HW2 will be posted soon.
- Review 3 postponed to Feb. 7

Filter Predicates

- Table R stored in a sequential file
- SELECT * FROM R WHERE zip=98195
- Need to scan entire file!
- Index: an auxiliary file for direct access

Binary Search

Sorted array:

12	24	31	37	42	48	52	59	61	66	69	75	88	92	95

Binary Search

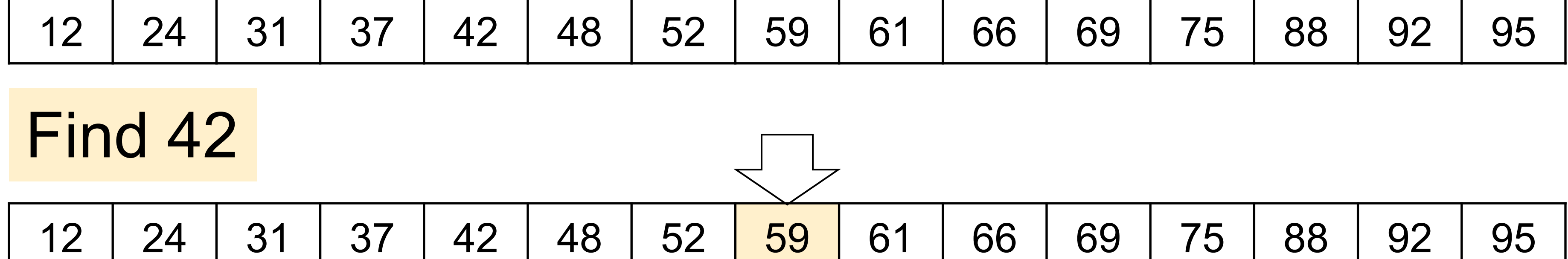
Sorted array:

12	24	31	37	42	48	52	59	61	66	69	75	88	92	95

Find 42

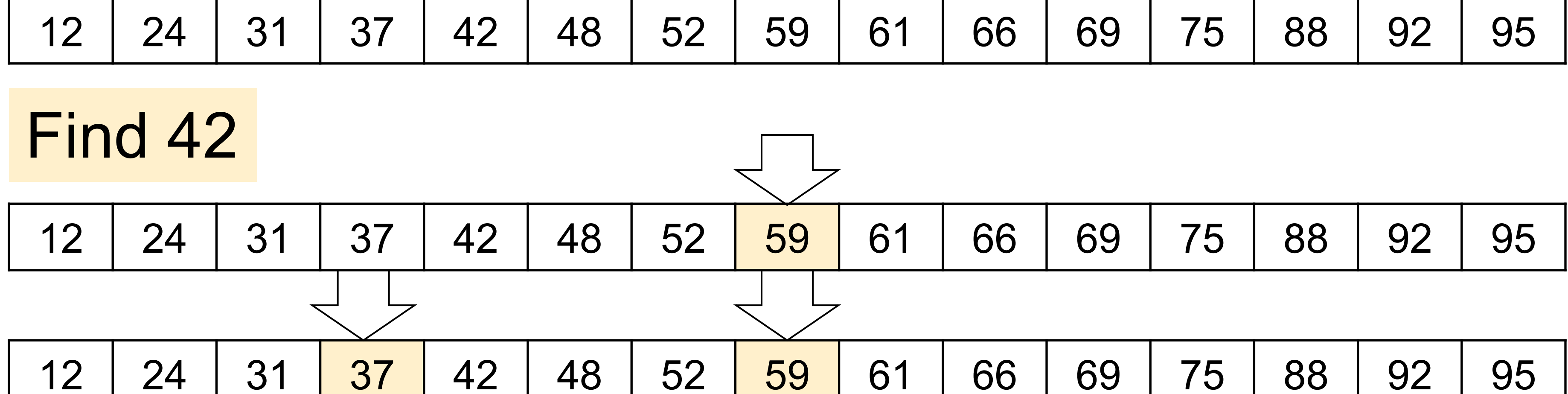
Binary Search

Sorted array:



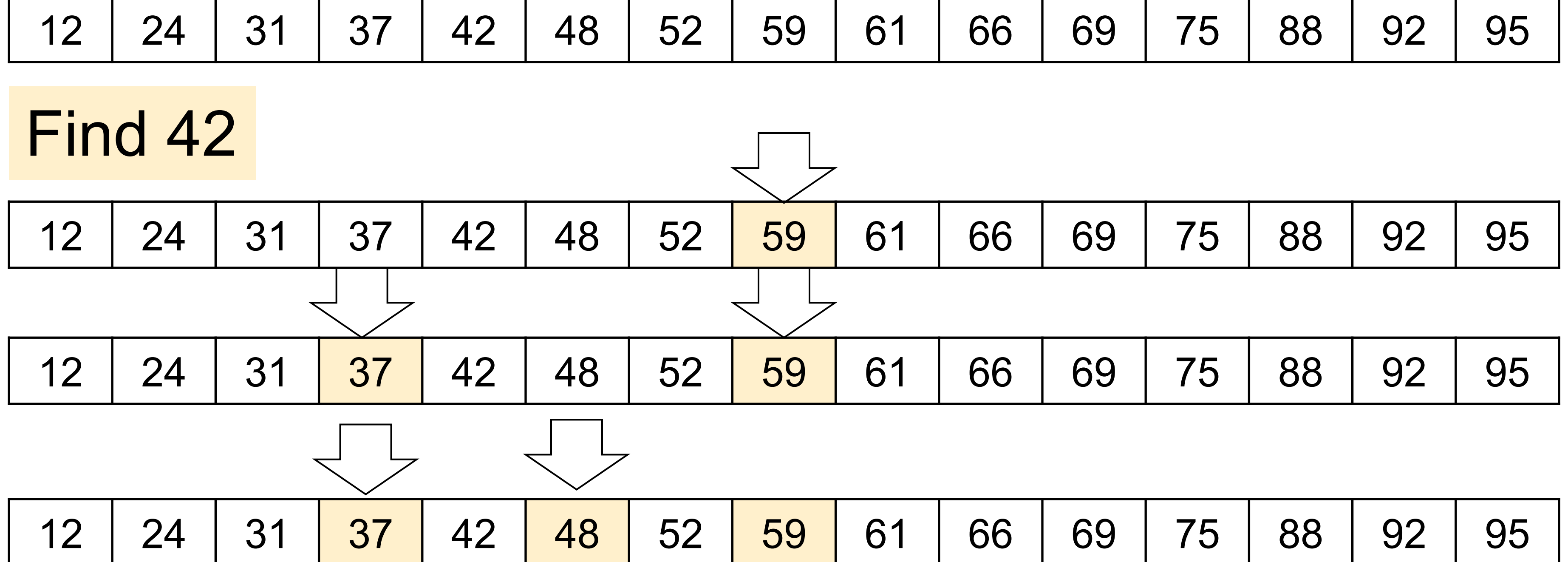
Binary Search

Sorted array:



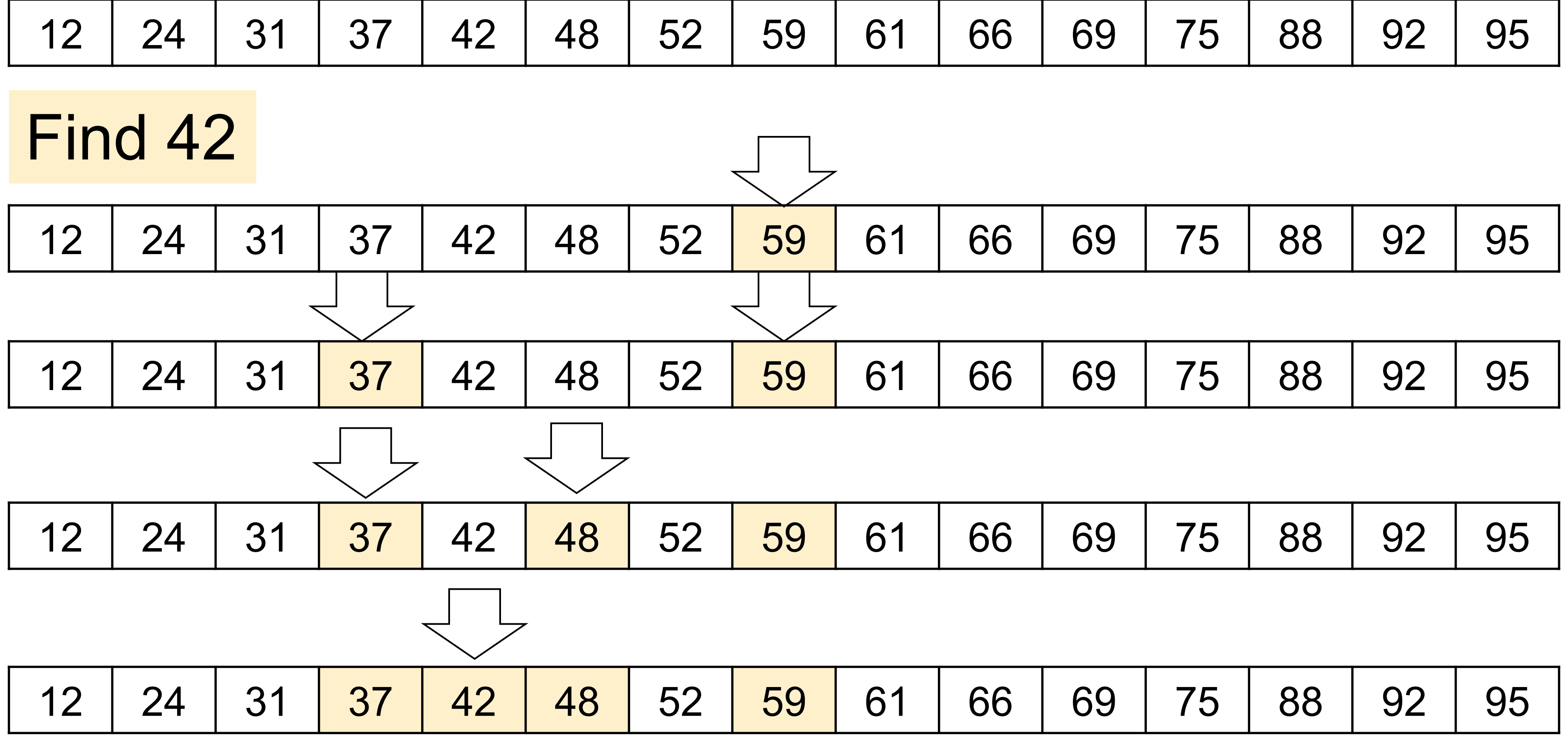
Binary Search

Sorted array:



Binary Search

Sorted array:



Binary Search

- log N comparisons -- why?
- Any algorithm needs log N -- why?

Binary Search

- log N comparisons -- why?
- Any algorithm needs log N -- why?

- Ex. N = 1,000,000,000
  - Sequential search: ≈ 500,000,000 steps
  - Binary search: 30 steps

Binary Search

- log N comparisons -- why?
- Any algorithm needs log N -- why?

- Ex. N = 1,000,000,000
  - Sequential search: ≈ 500,000,000 steps
  - Binary search: 30 steps

Sorted array + binary search = optimal!

Binary Search On Disk

Page 0			
12	24	31	37

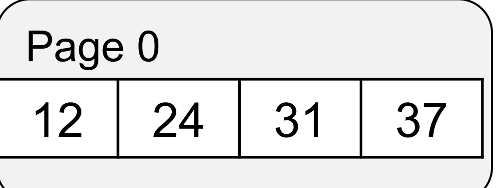
Page 1			
42	48	52	59

Page 2			
61	66	69	75

Page 3		
88	92	95

What problem do you see?

Binary Search On Disk



Page 1

42	48	52	59

Page 2

61	66	69	75

Page 3

88	92	95

What problem do you see?

Read one page, use only one key!

Solution? Wait for it!

Binary Search

Assume array is in main memory

Problem: updates with poor performance

- Insert: shift O(N) items to the right
- Delete: shift O(N) items to the left

Insert/Delete

12	24	31	37	42	48	52	59	61	66	69	75

Insert/Delete

12	24	31	37	42	48	52	59	61	66	69	75

Insert 45

Insert/Delete

12	24	31	37	42	48	52	59	61	66	69	75

Insert 45

shift

12	24	31	37	42		48	52	59	61	66	69	75

Insert/Delete

12	24	31	37	42	48	52	59	61	66	69	75

Insert 45

shift

12	24	31	37	42		48	52	59	61	66	69	75

12	24	31	37	42	45	48	52	59	61	66	69	75

Insert/Delete

12	24	31	37	42	48	52	59	61	66	69	75

Insert 45

12	24	31	37	42		48	52	59	61	66	69	75

12	24	31	37	42	45	48	52	59	61	66	69	75

Delete 31

…

Binary Search Tree

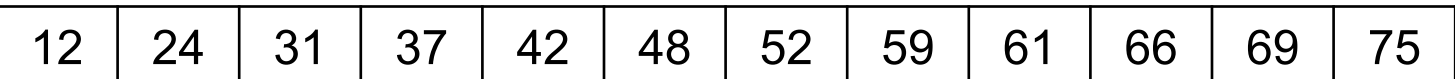
- Sorted arrays: best for main-memory search, but they cannot handle updates

- Binary search trees:
  - Keep the binary search idea
  - Allow for updates

Binary Search Tree

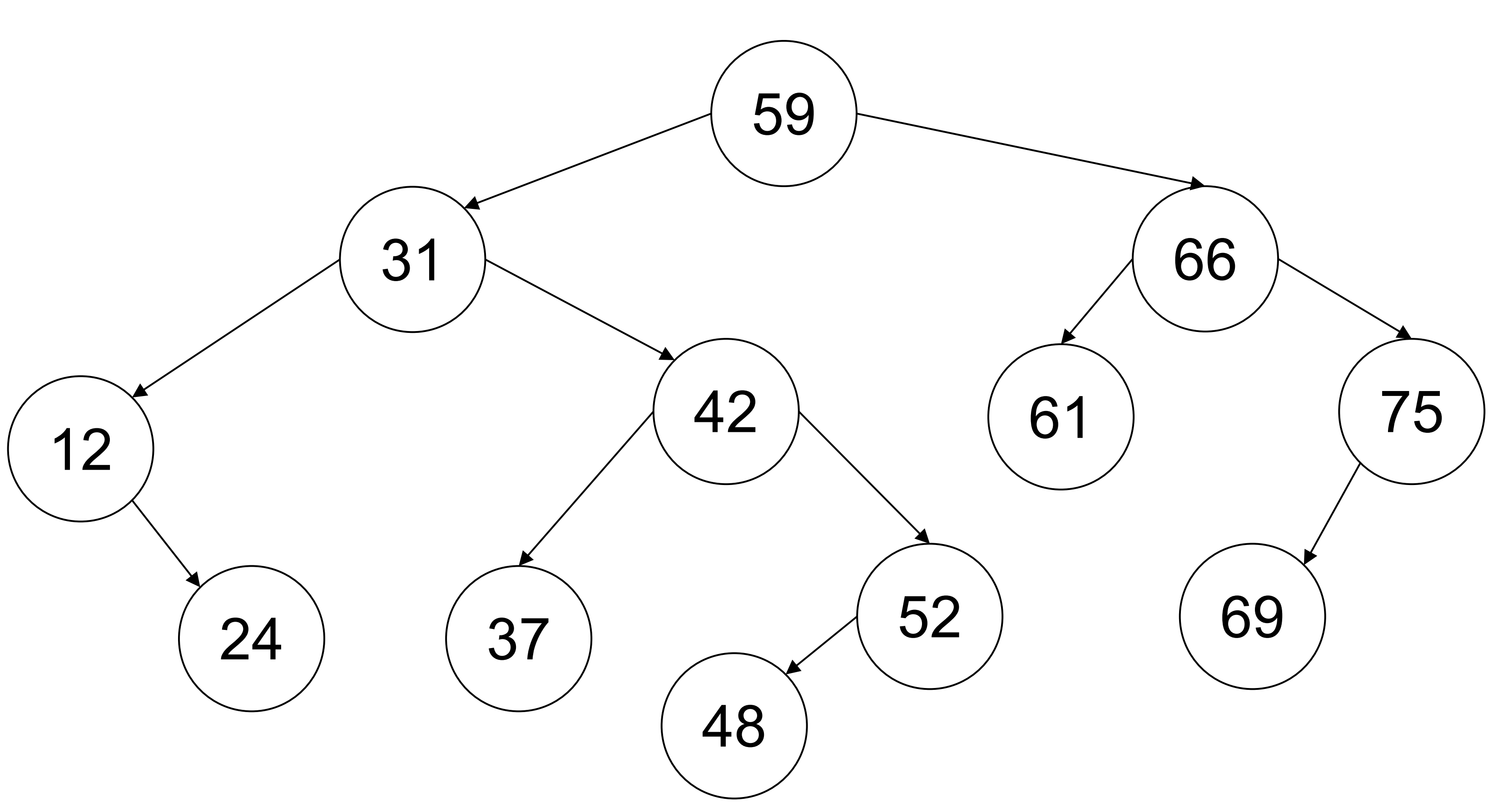
12	24	31	37	42	48	52	59	61	66	69	75

Binary Search Tree

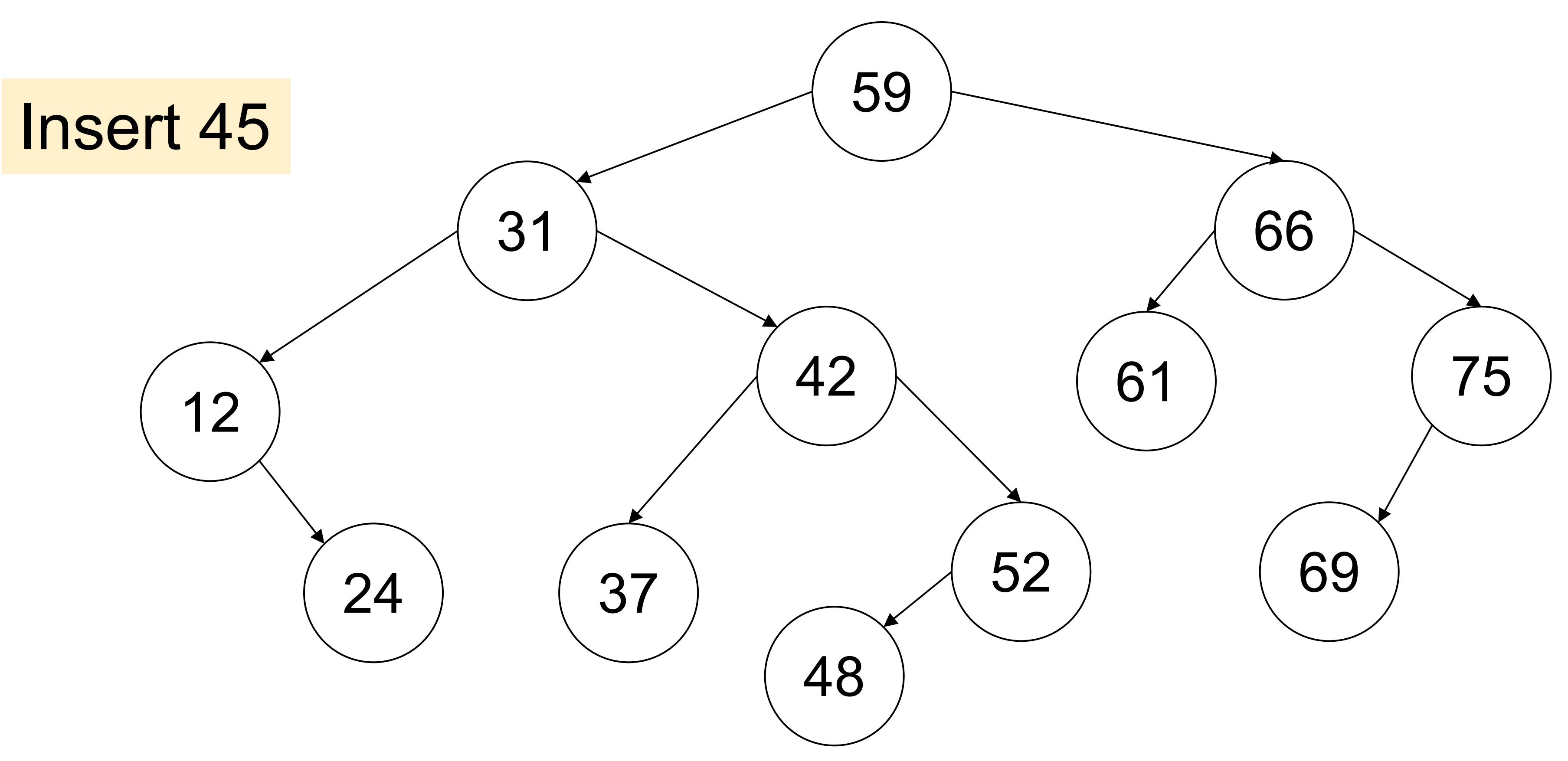


12	24	31	37	42	48	52	59	61	66	69	75

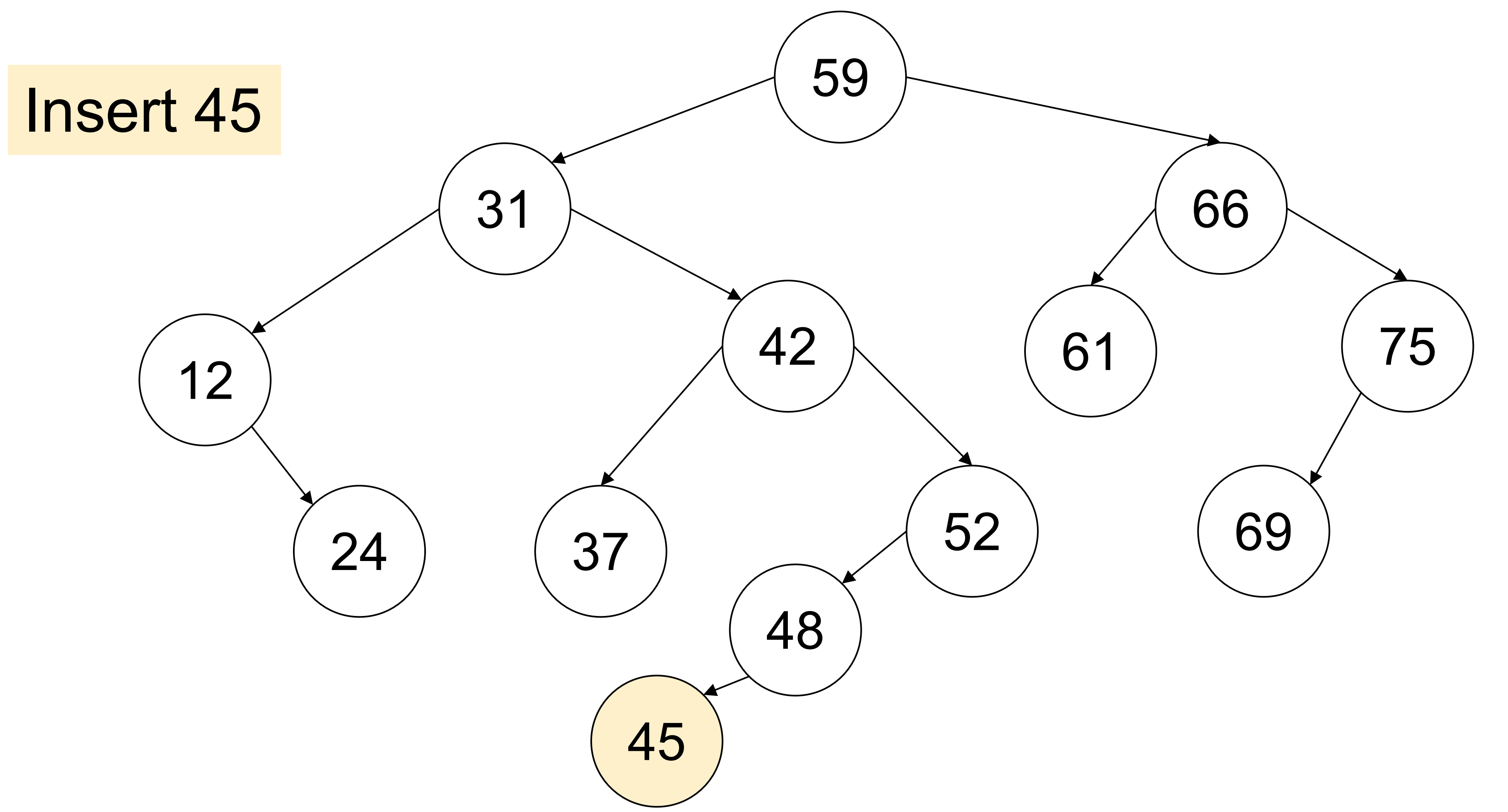
Binary Search Tree



Binary Search Tree



Binary Search Tree



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

More likely multiple blocks

- Idea in B Trees
  - Make 1 node = 1 page (= 1 block)
  - Store multiple keys and children
  - Read 1 page, use many keys

B+ Trees

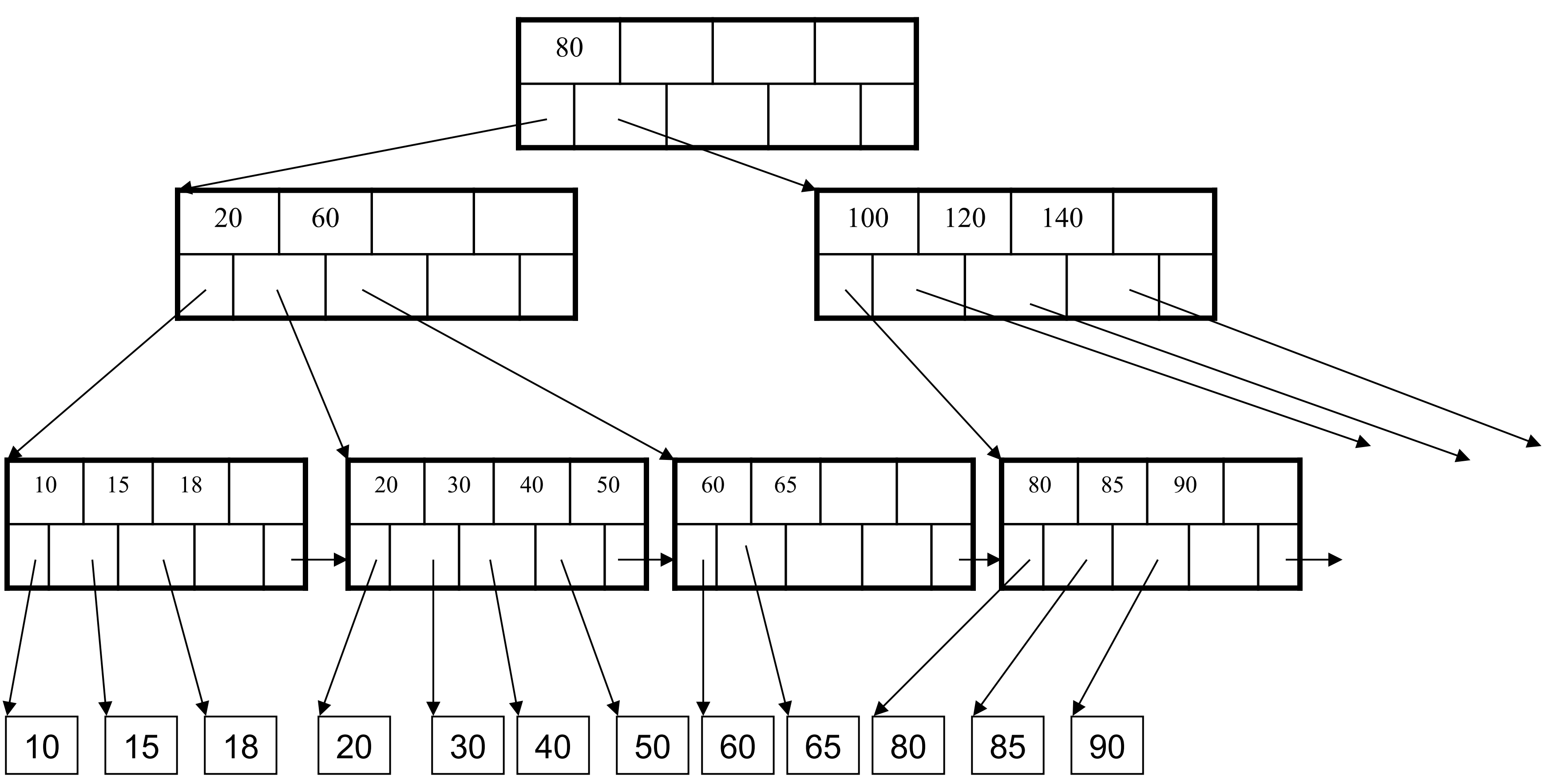
More likely multiple blocks

- 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

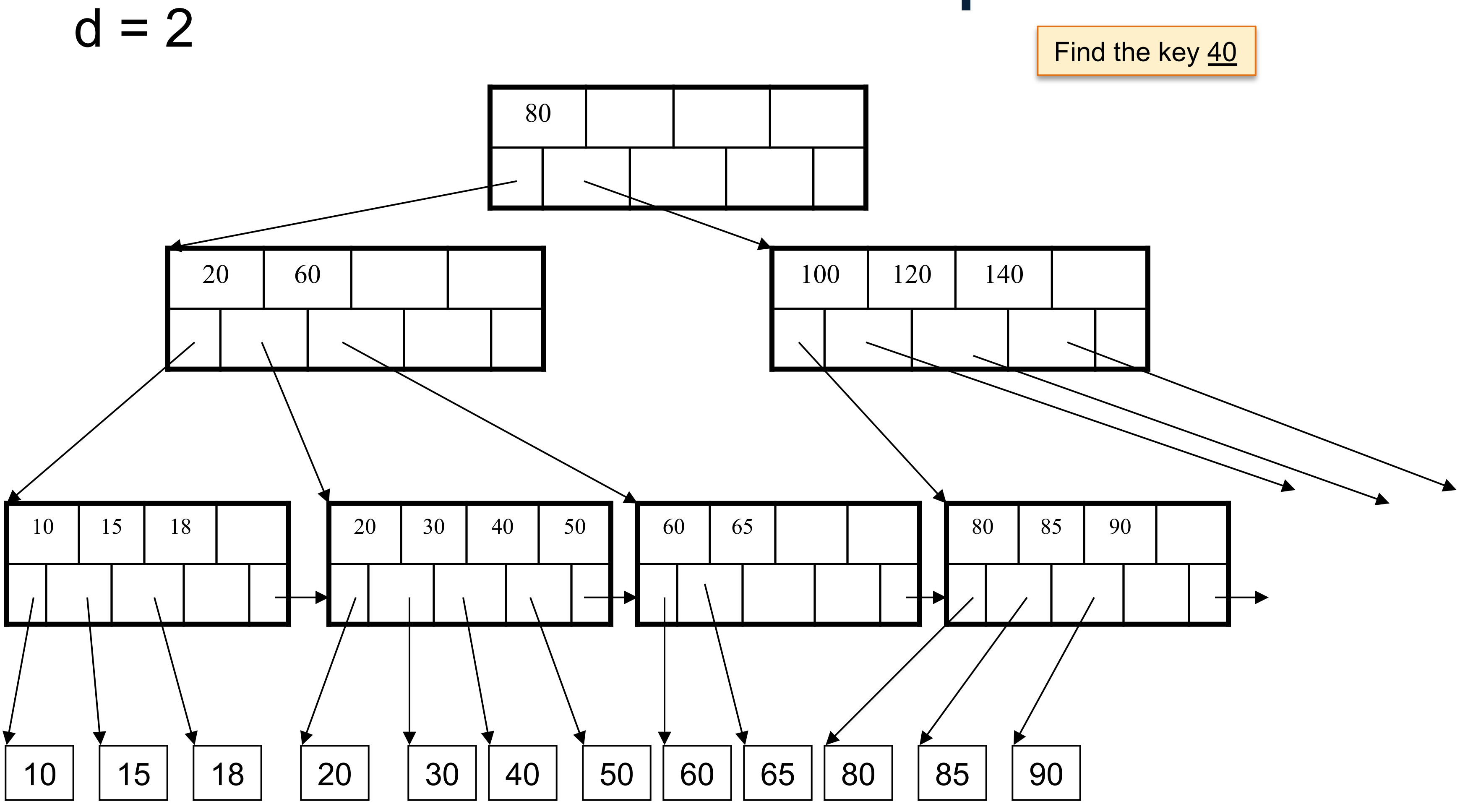
B+ Tree Example

d = 2



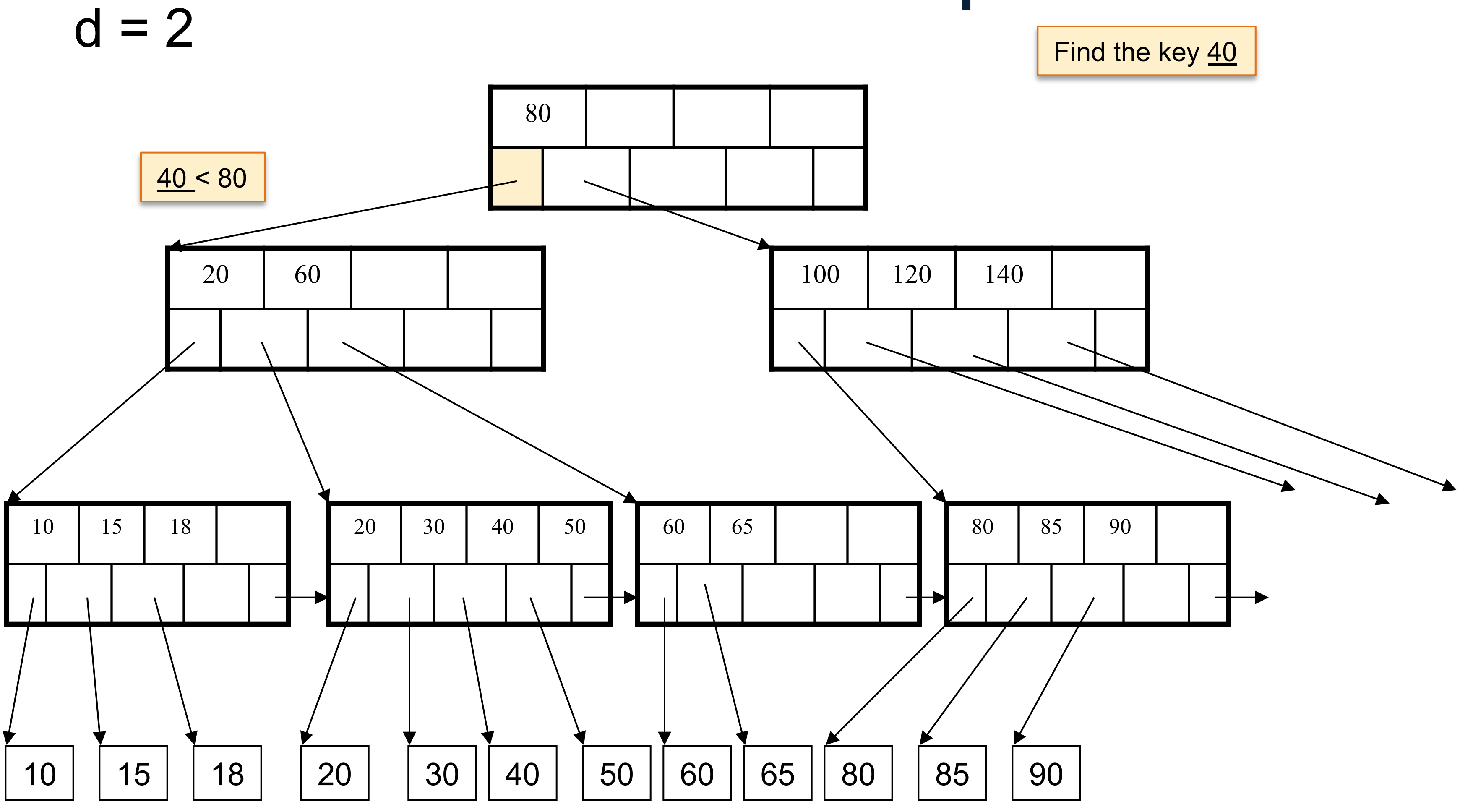
B+ Tree Example

d = 2



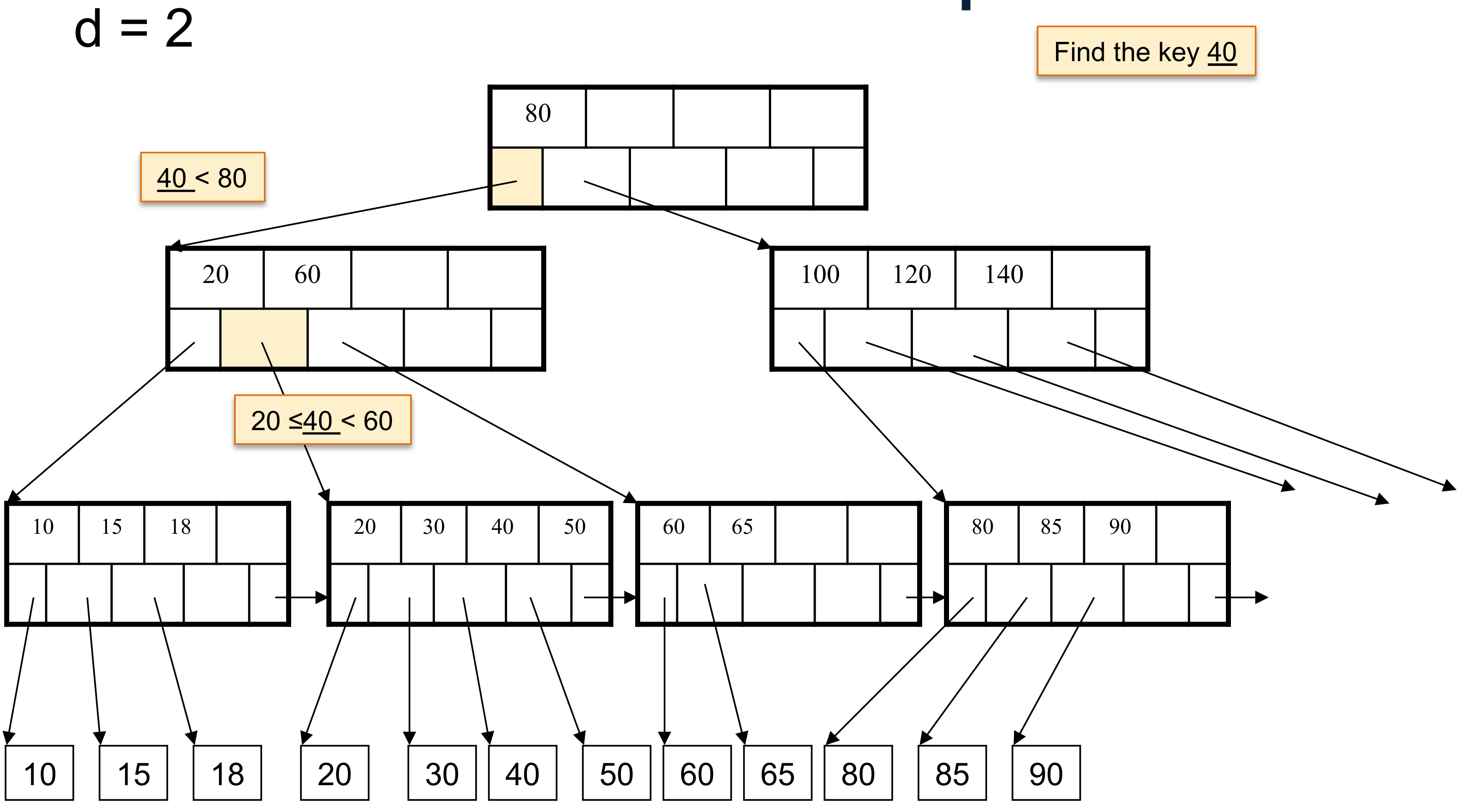
B+ Tree Example

d = 2



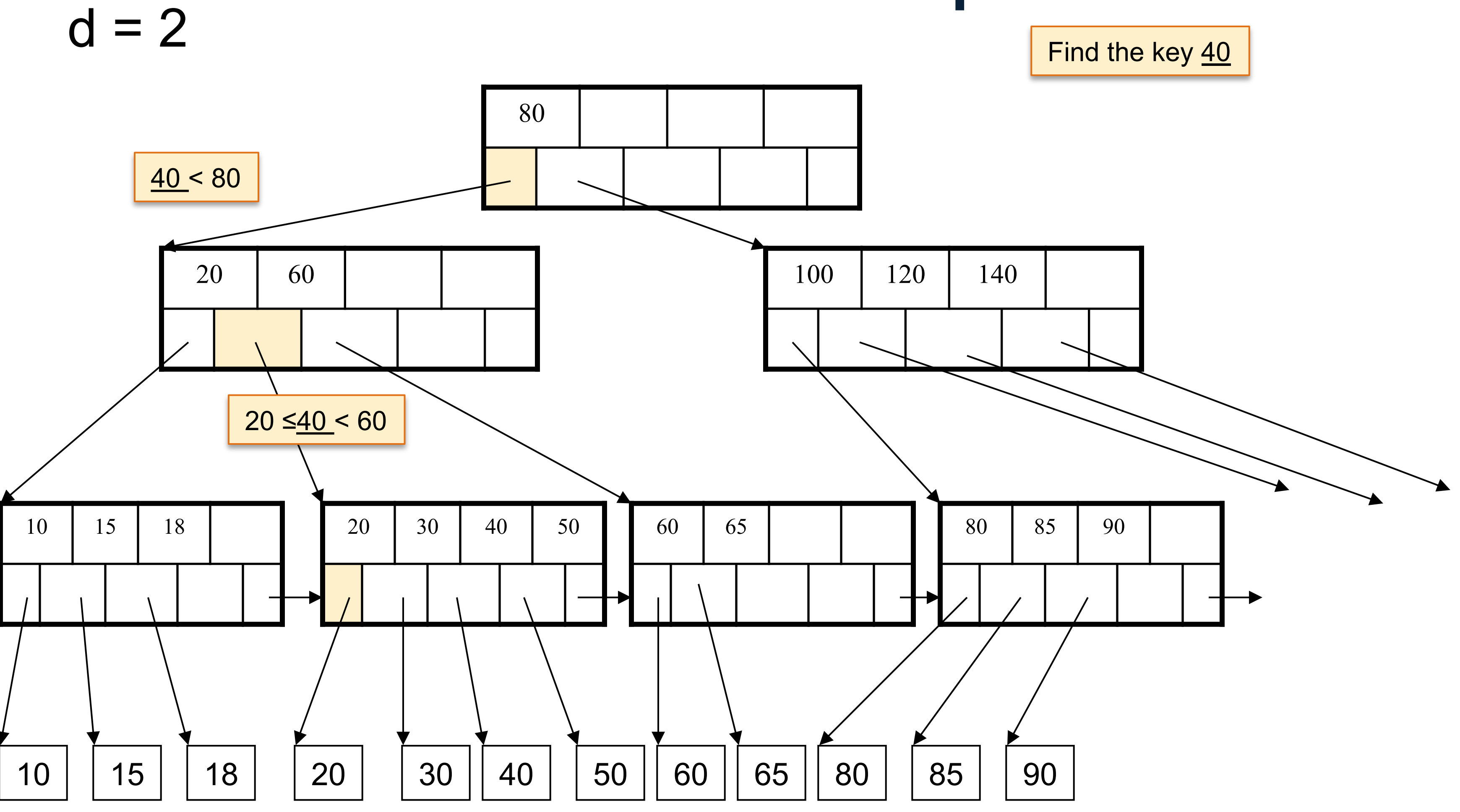
B+ Tree Example

d = 2



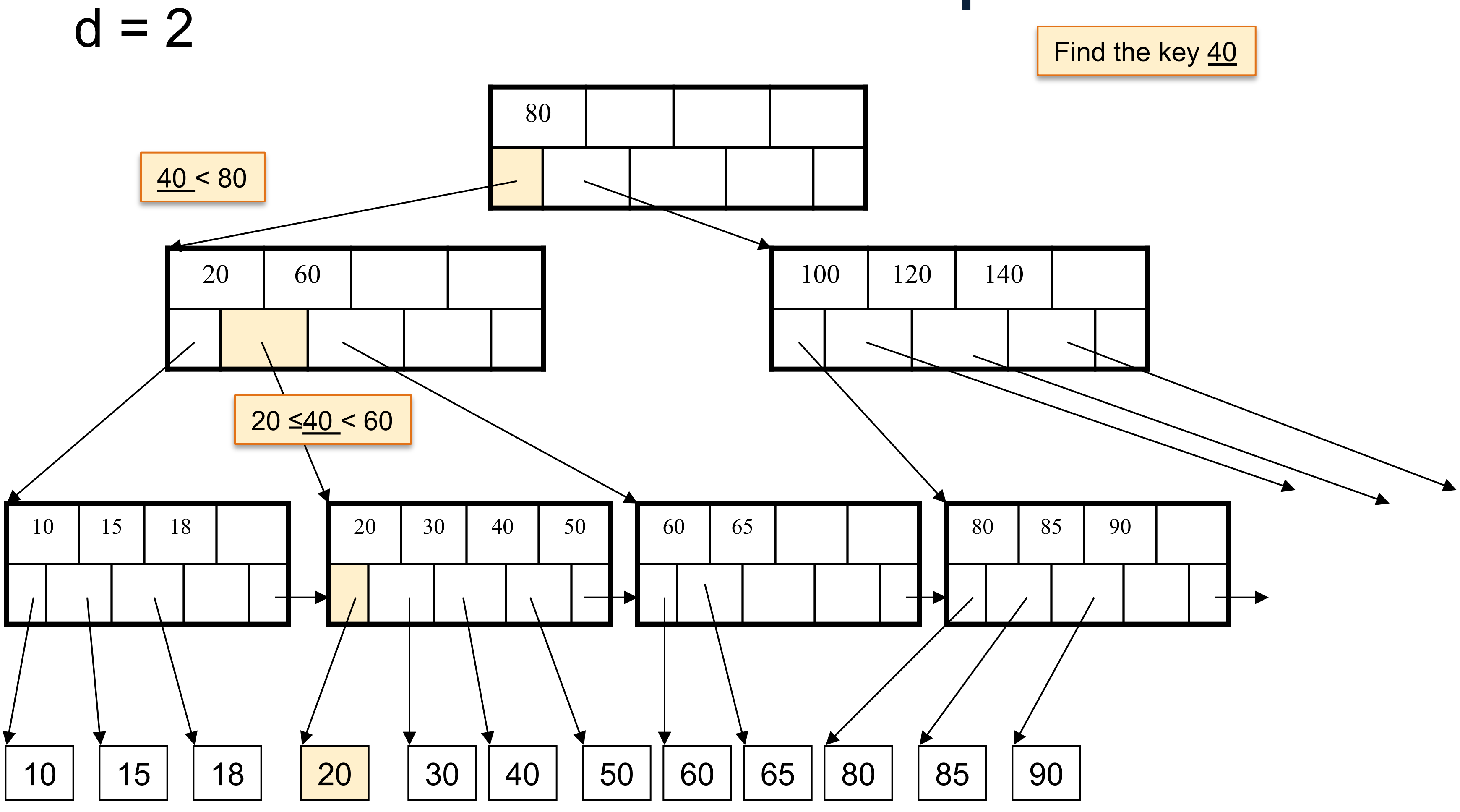
B+ Tree Example

d = 2



B+ Tree Example

d = 2



B+ Trees Properties

- For each node except the root, maintain 50% occupancy of keys

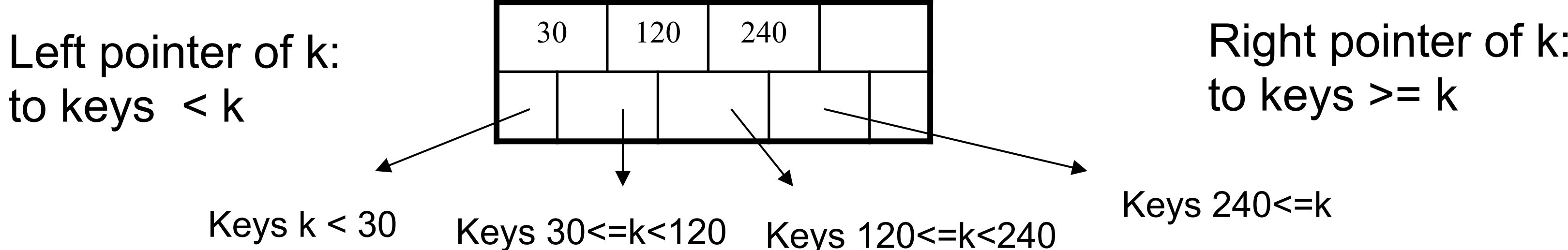
- Insert and delete must rebalance to maintain constraints

B+ Trees Details

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

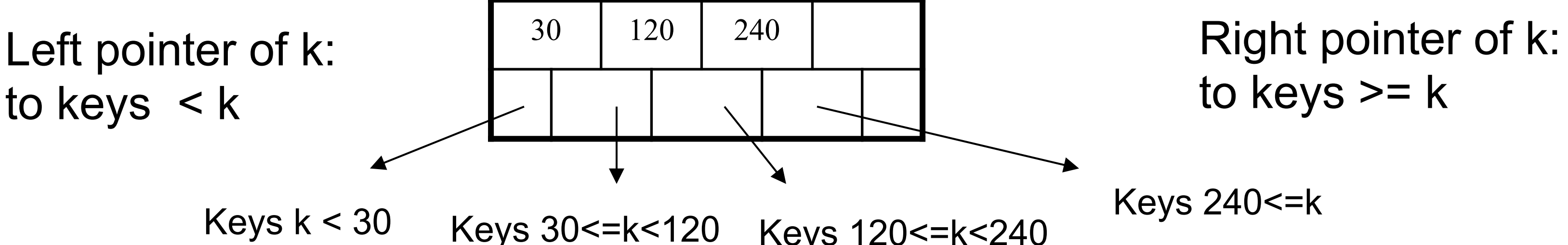
B+ Trees Details

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

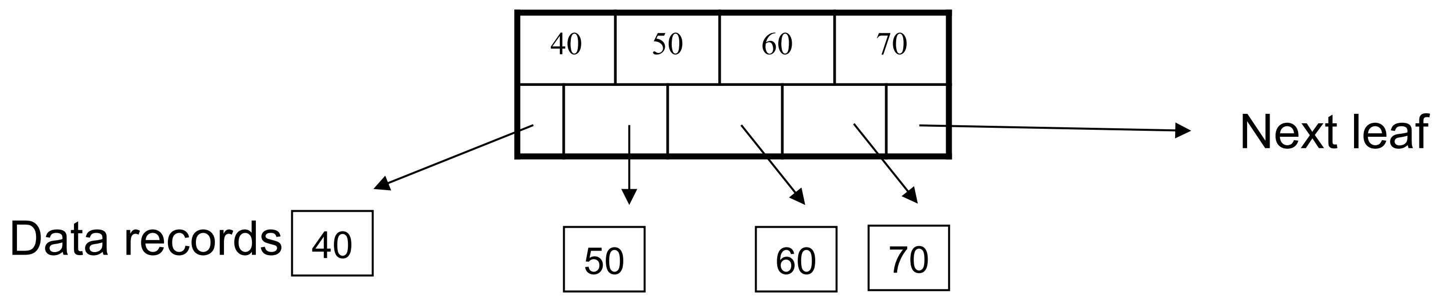


B+ Trees Details

- 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

- Every B+ tree is perfectly balanced
  - Assume each node has m children:
  - Depth=$\log_m N$

Search time

- Every B+ tree is perfectly balanced
  - Assume each node has m children:
  - Depth=$\log_m N$
- Ex. N = 1,000,000,000
  - Binary search tree: log N = 30
  - B+ tree, assume m=128: $\log_m N$ = 30/7 = 4

Search time

- Every B+ tree is perfectly balanced
  - Assume each node has m children:
  - Depth=$\log_m N$
- Ex. N = 1,000,000,000
  - Binary search tree: log N = 30
  - B+ tree, assume m=128: $\log_m N$ = 30/7 = 4
- Search time better than log N??

Search time

- Every B+ tree is perfectly balanced
  - Assume each node has m children:
  - Depth=$\log_m N$
- Ex. N = 1,000,000,000
  - Binary search tree: log N = 30
  - B+ tree, assume m=128: $\log_m N$ = 30/7 = 4
- Search time better than log N??
  - No: binary search in a node takes log m
  - Search time = $\log_m N$ * log m = log N

B+ Trees in Practice

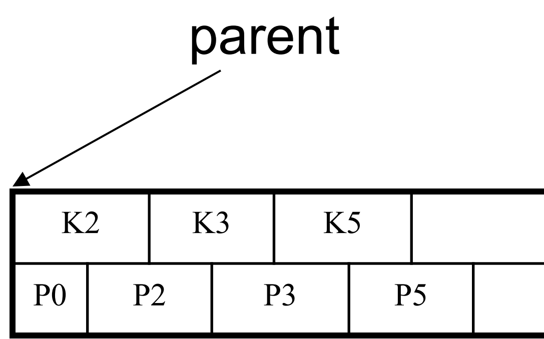
- Typical order: 100. Typical fill-factor: 67%.
  - average fanout = 133
- Typical capacities
  - Height 4: 133^4 = 312,900,700 records
  - Height 3: 133^3 = 2,352,637 records
- Can often hold top levels in buffer pool
  - Level 1 = 1 page = 8 Kbytes
  - Level 2 = 133 pages = 1 Mbyte
  - Level 3 = 17,689 pages = 133 Mbytes

Insertion in a B+ Tree

Insert (K, P)

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

Insert k1

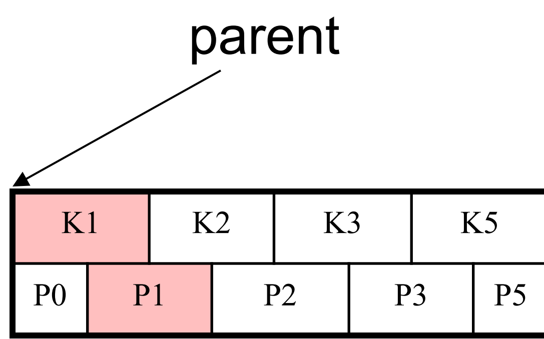


Insertion in a B+ Tree

Insert (K, P)

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

Insert k1

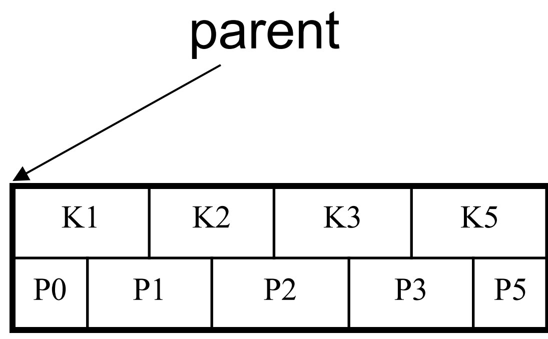


Insertion in a B+ Tree

Insert (K, P)

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

Insert k4

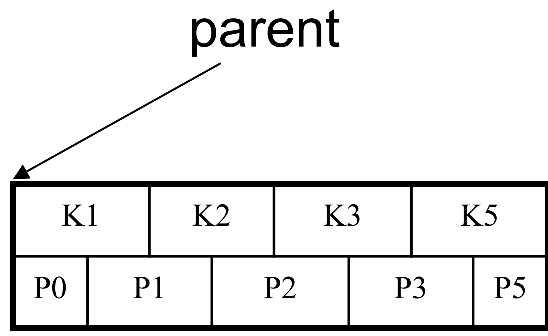


Insertion in a B+ Tree

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

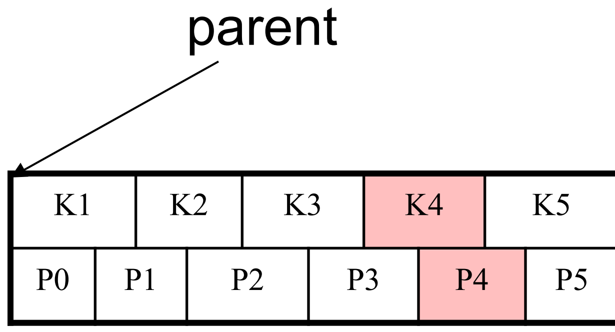


Insertion in a B+ Tree

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:

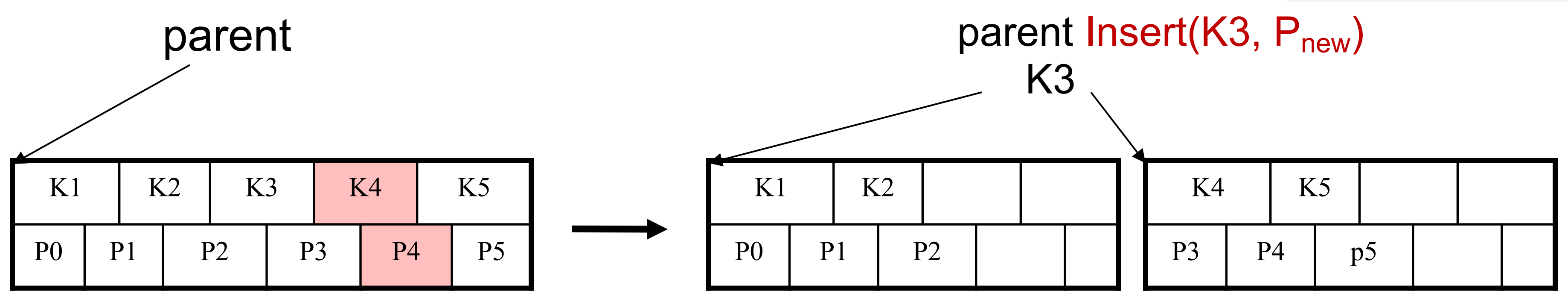
Insert k4



Insertion in a B+ Tree

Insert (K, P)

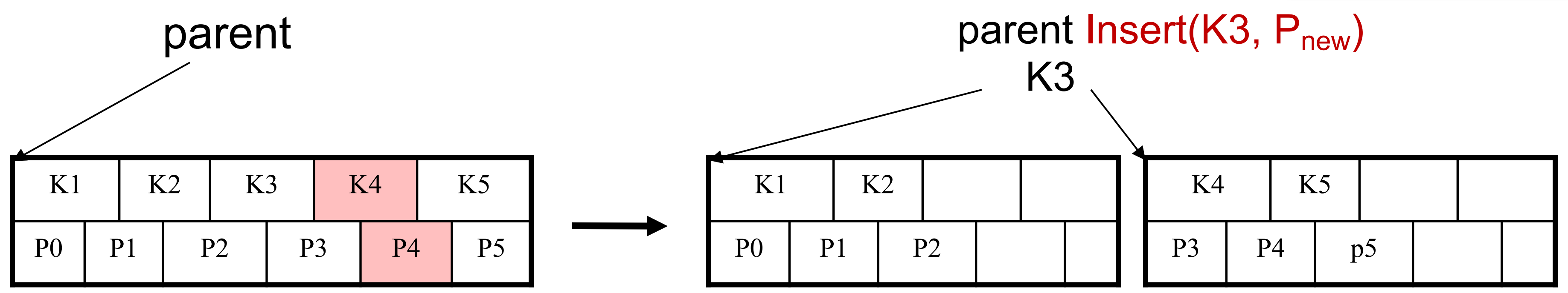
- Find leaf where K belongs, insert
- If no overflow (2d keys or less), halt
- If overflow (2d+1 keys), split node, insert in parent:



Insertion in a B+ Tree

Insert (K, P)

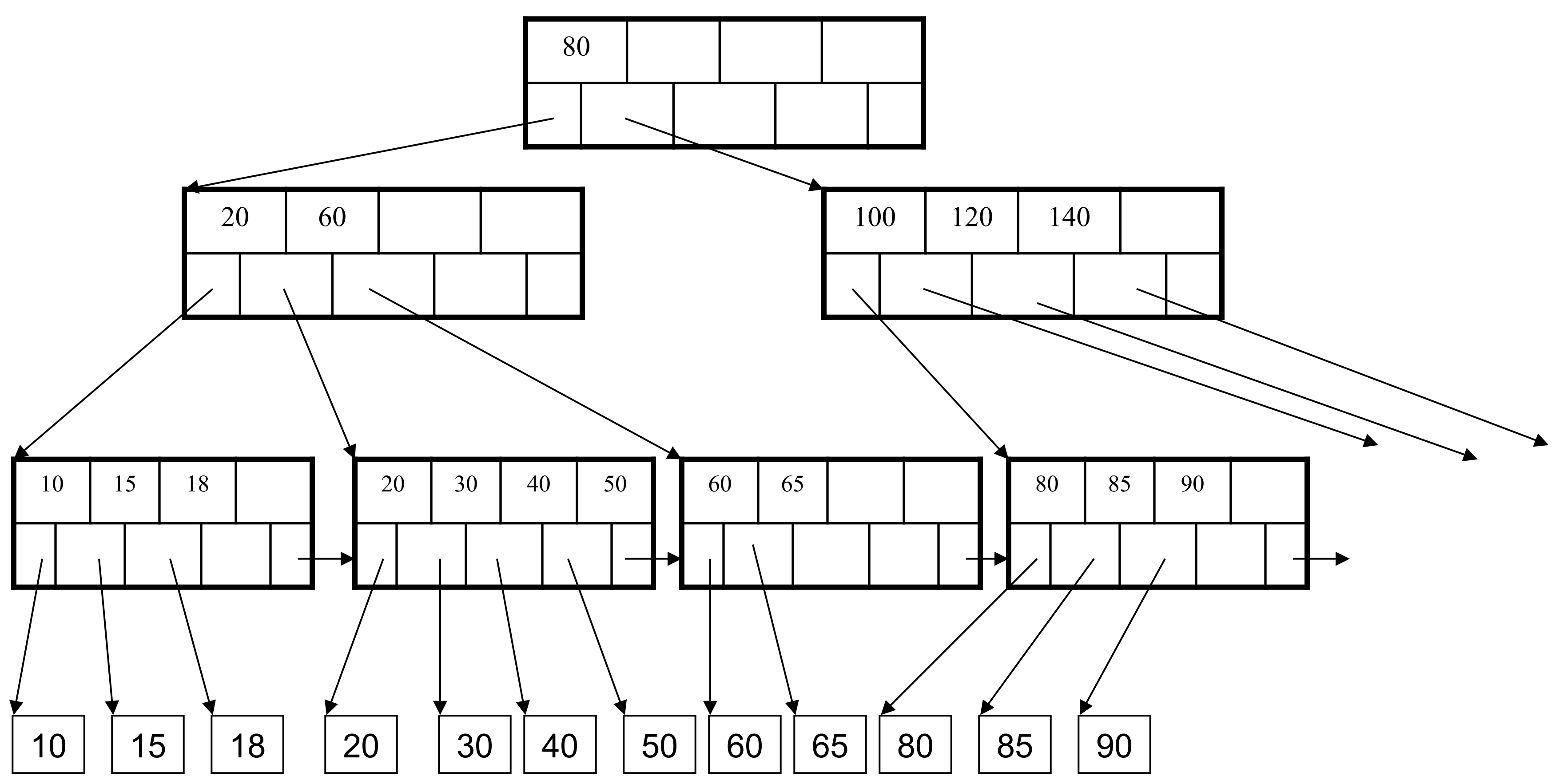
- Find leaf where K belongs, insert
- If no overflow (2d keys or less), halt
- If overflow (2d+1 keys), split node, insert in parent:



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

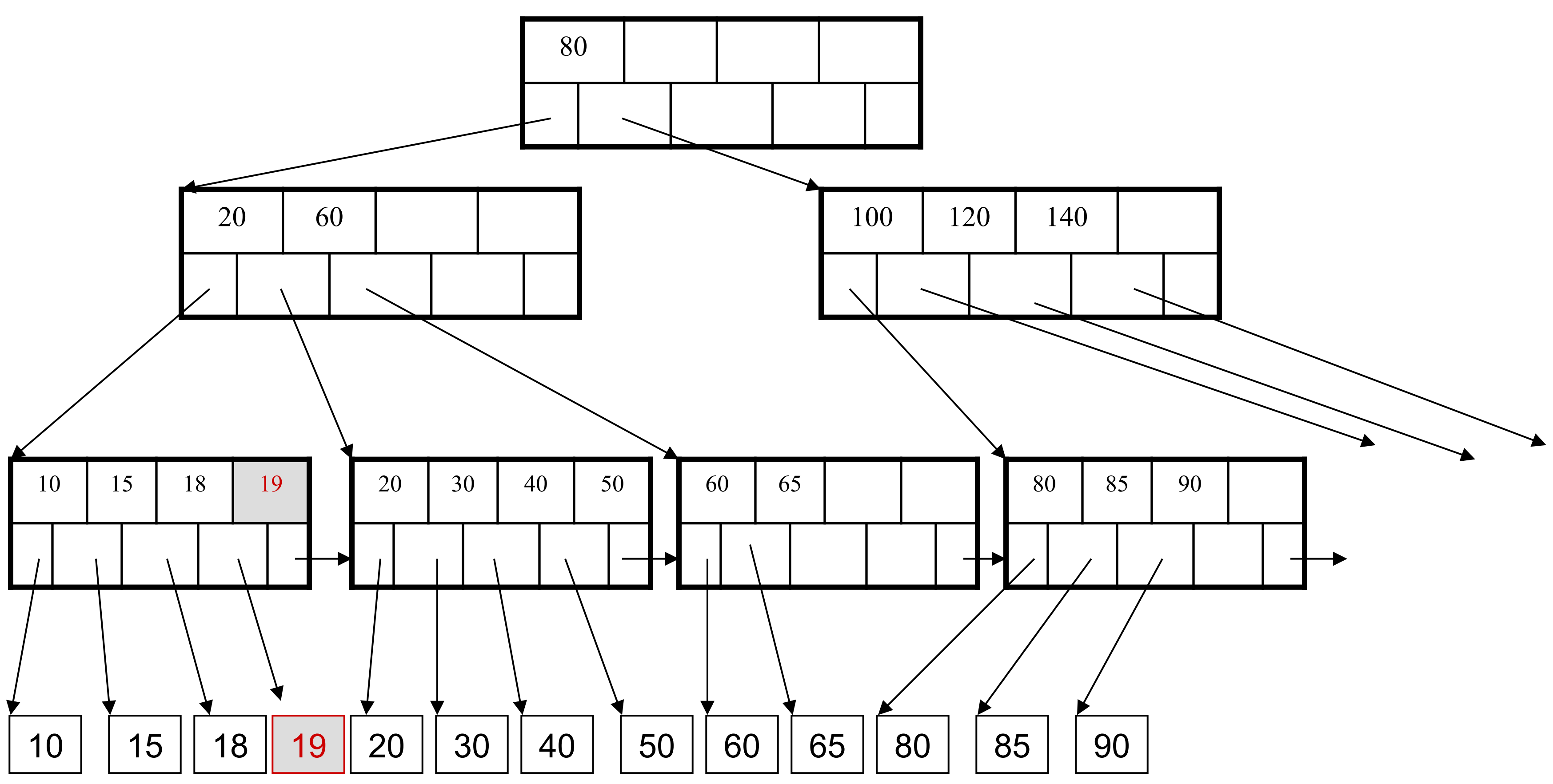
Insertion in a B+ Tree

Insert K=19



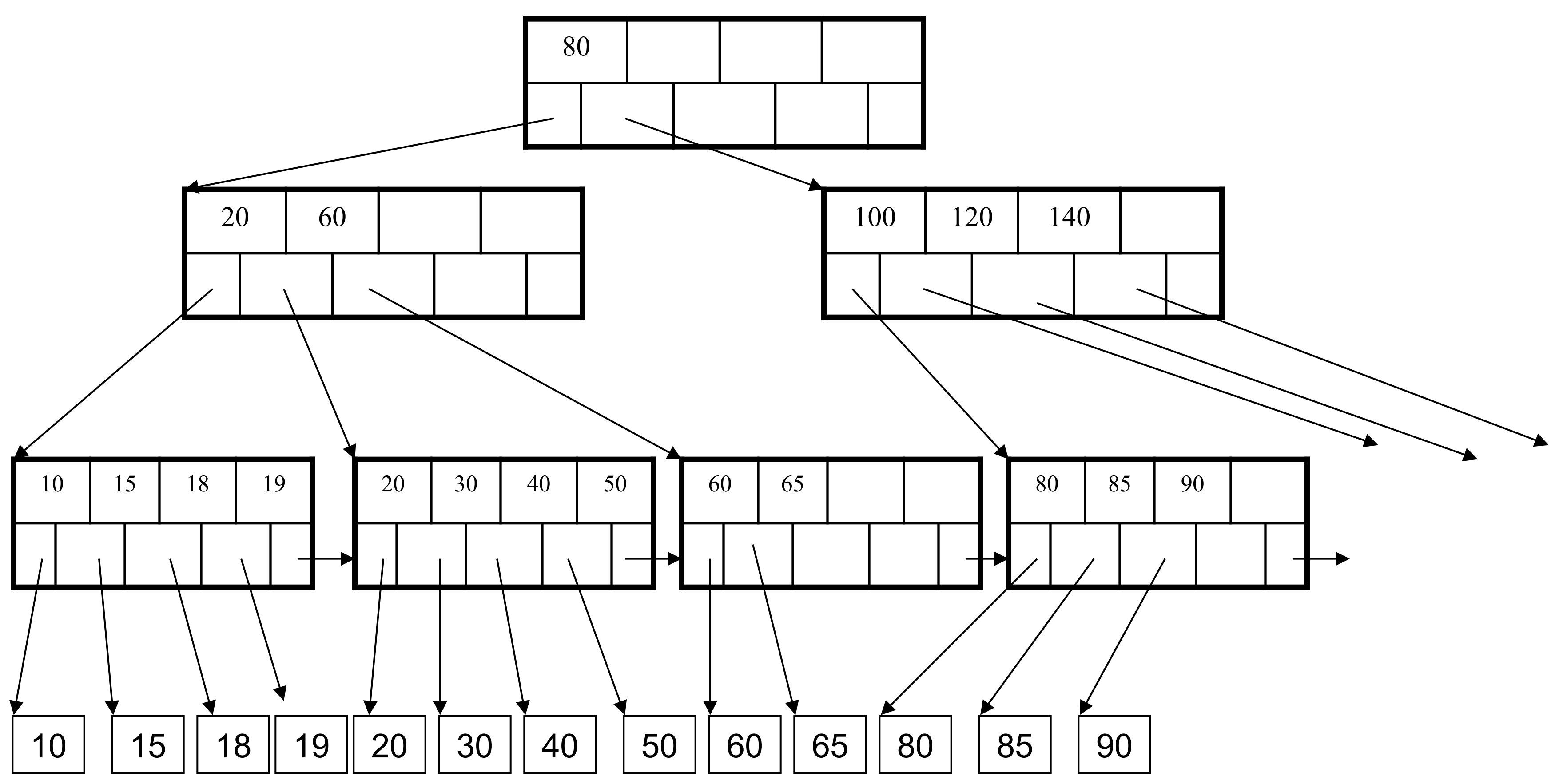
Insertion in a B+ Tree

After insertion



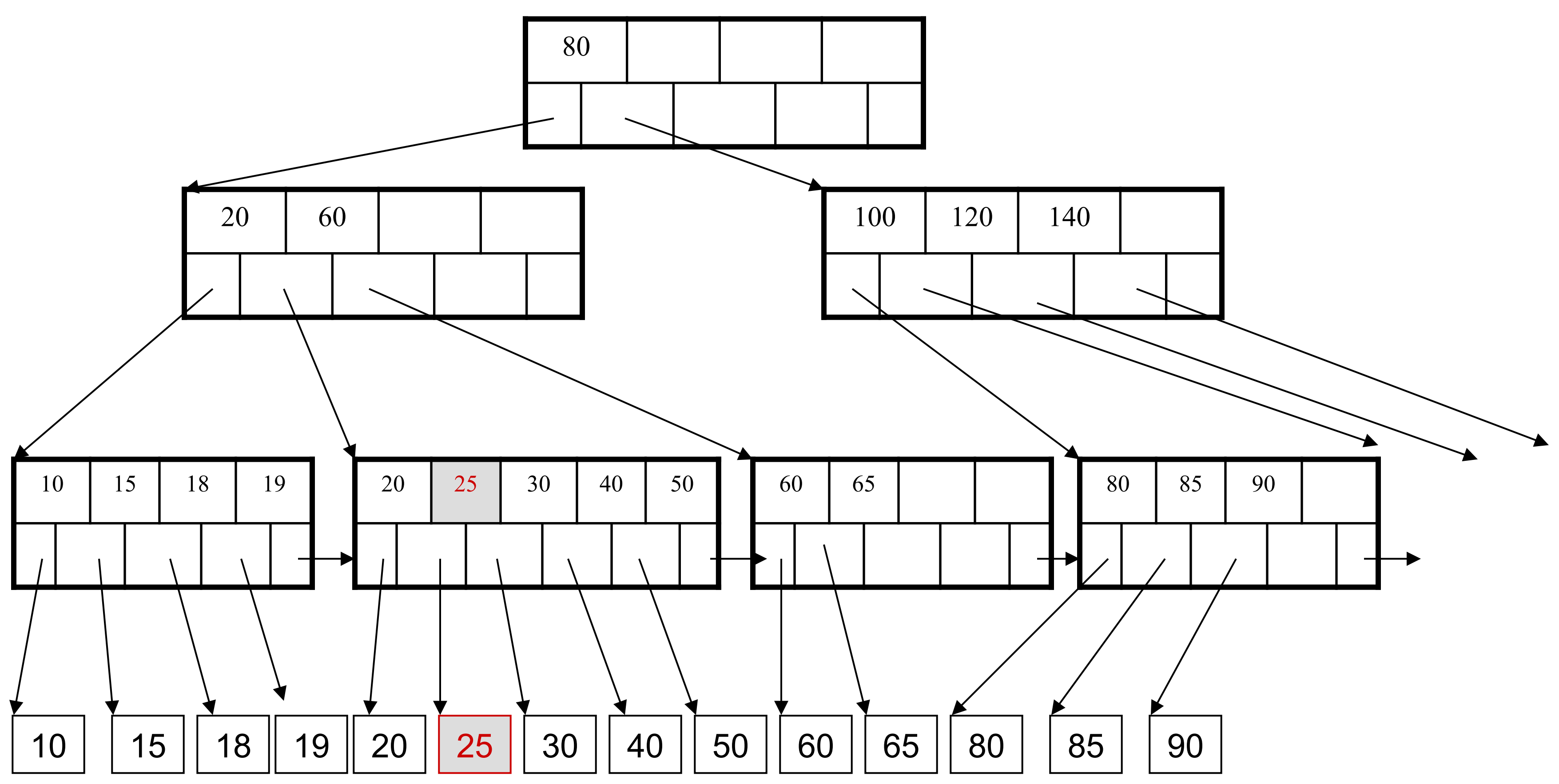
Insertion in a B+ Tree

Now insert 25



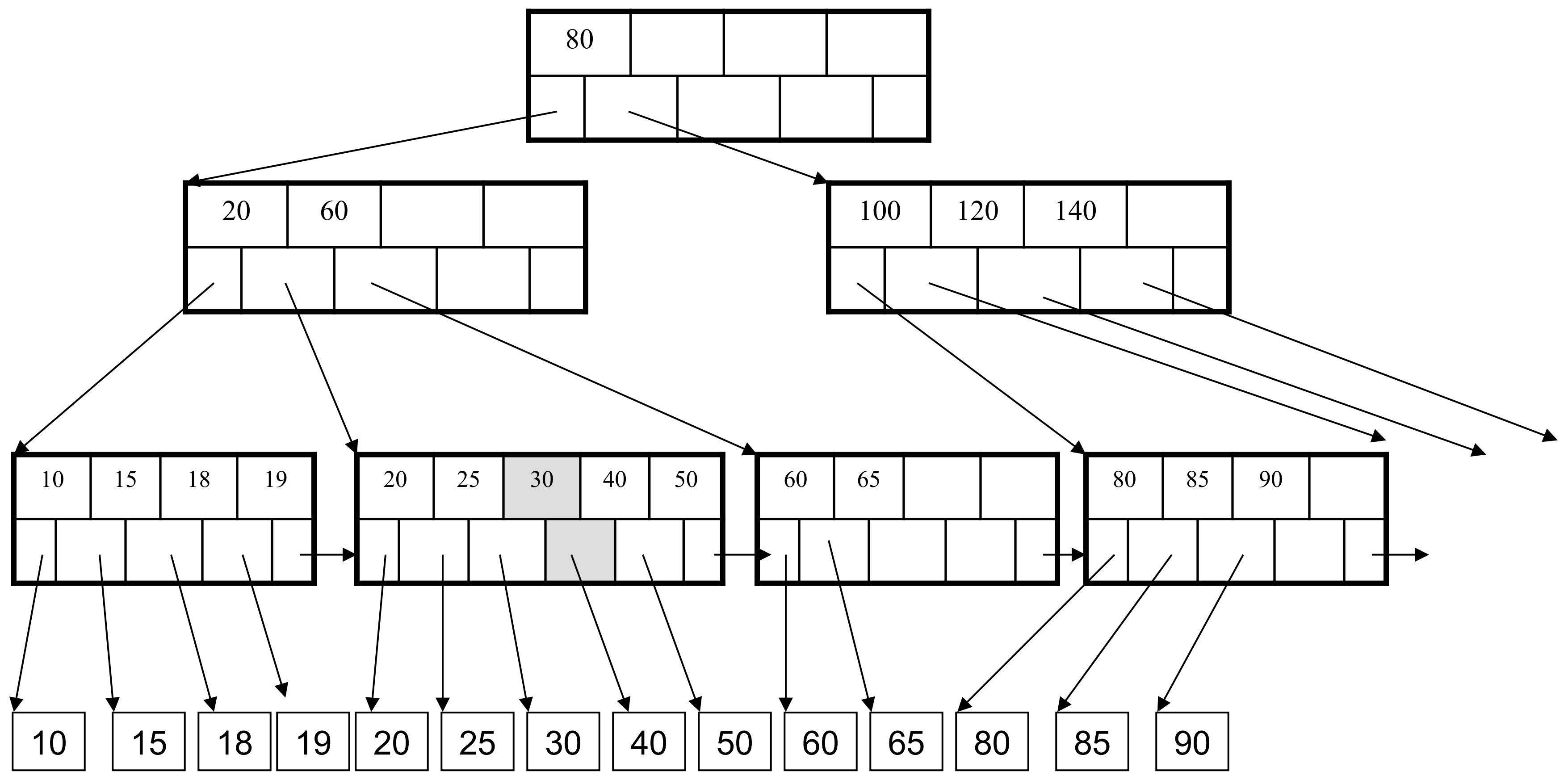
Insertion in a B+ Tree

After insertion



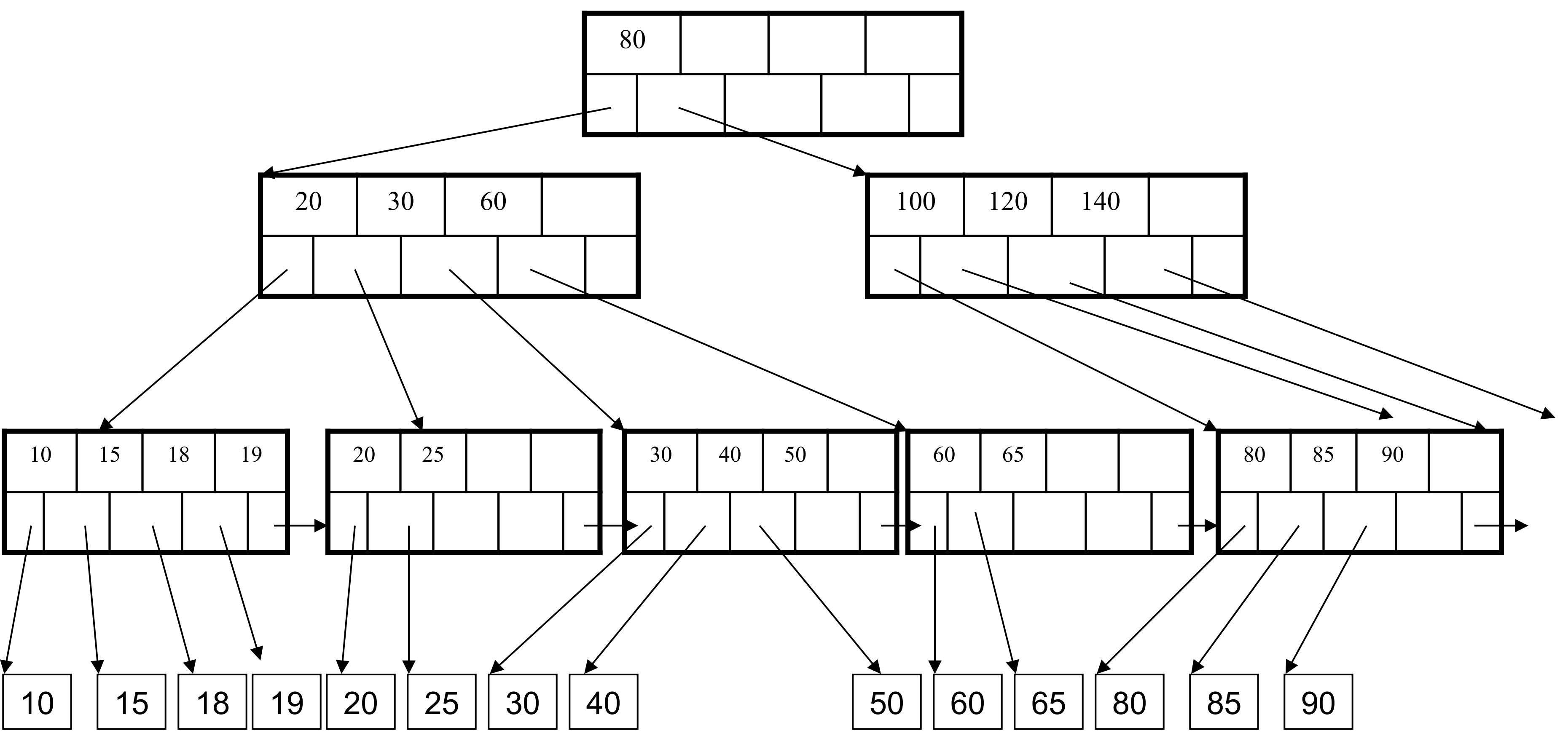
Insertion in a B+ Tree

But now have to split !



Insertion in a B+ Tree

After the split



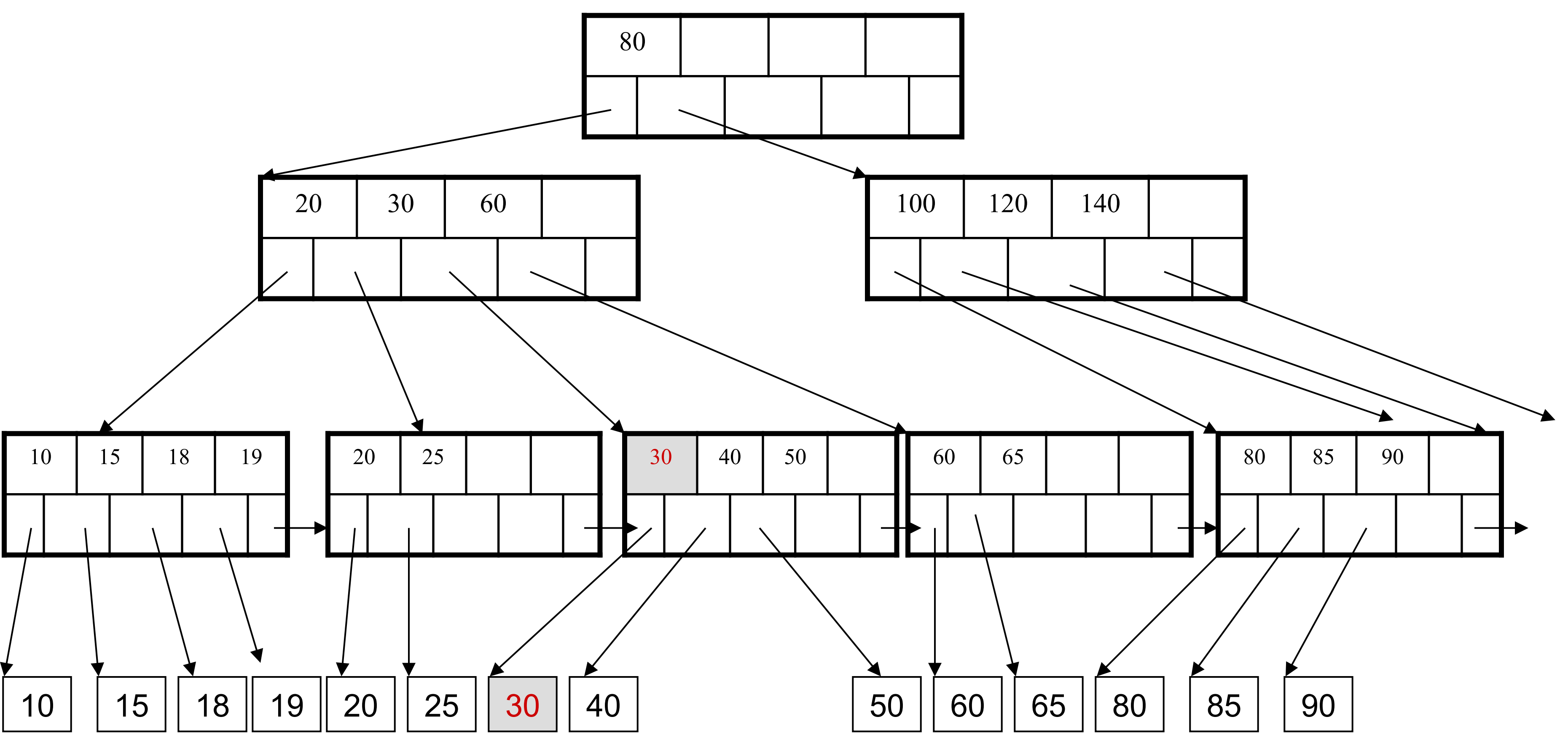
Deletion: Summary

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
  - Parent: Delete (K', P')

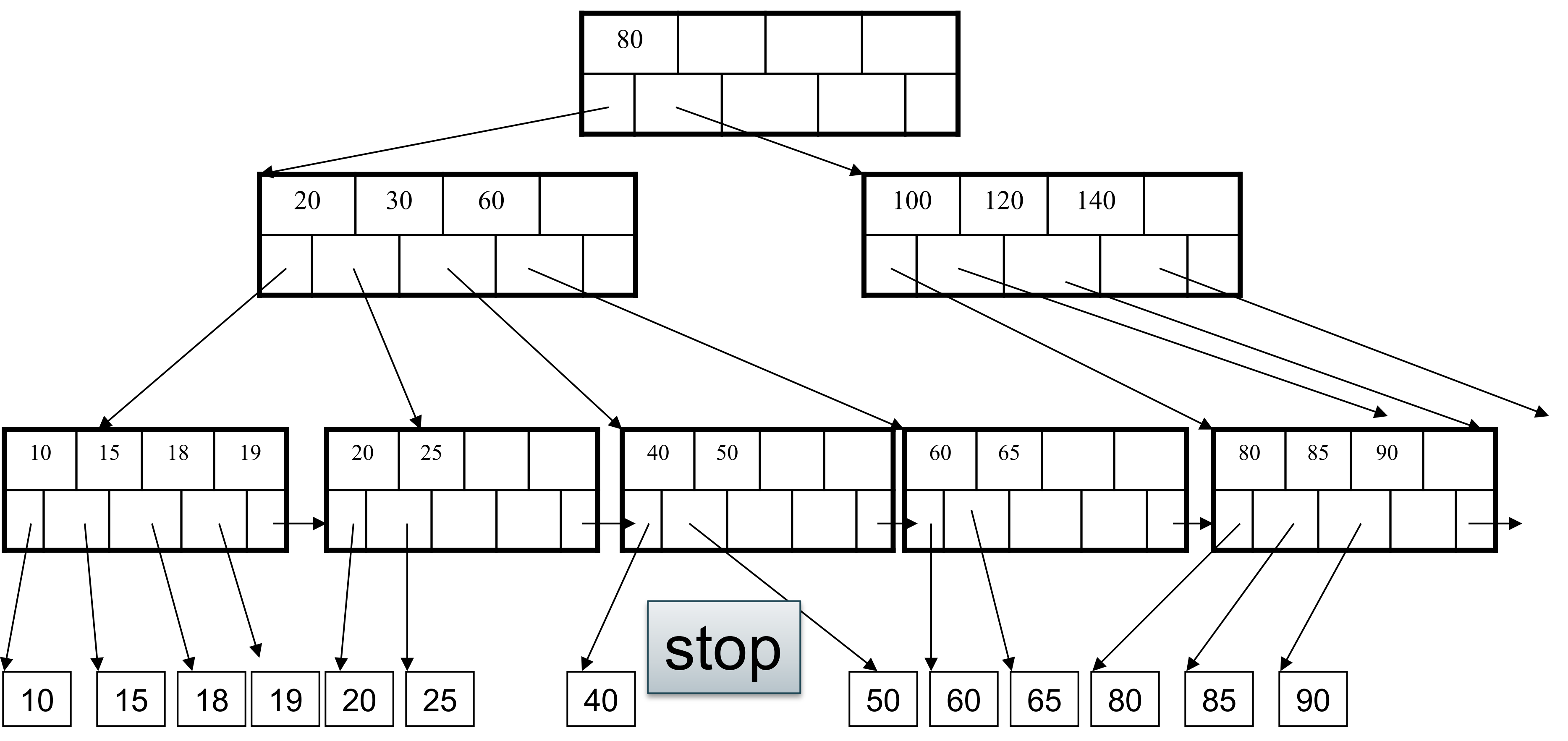
Deletion from a B+ Tree

Delete 30



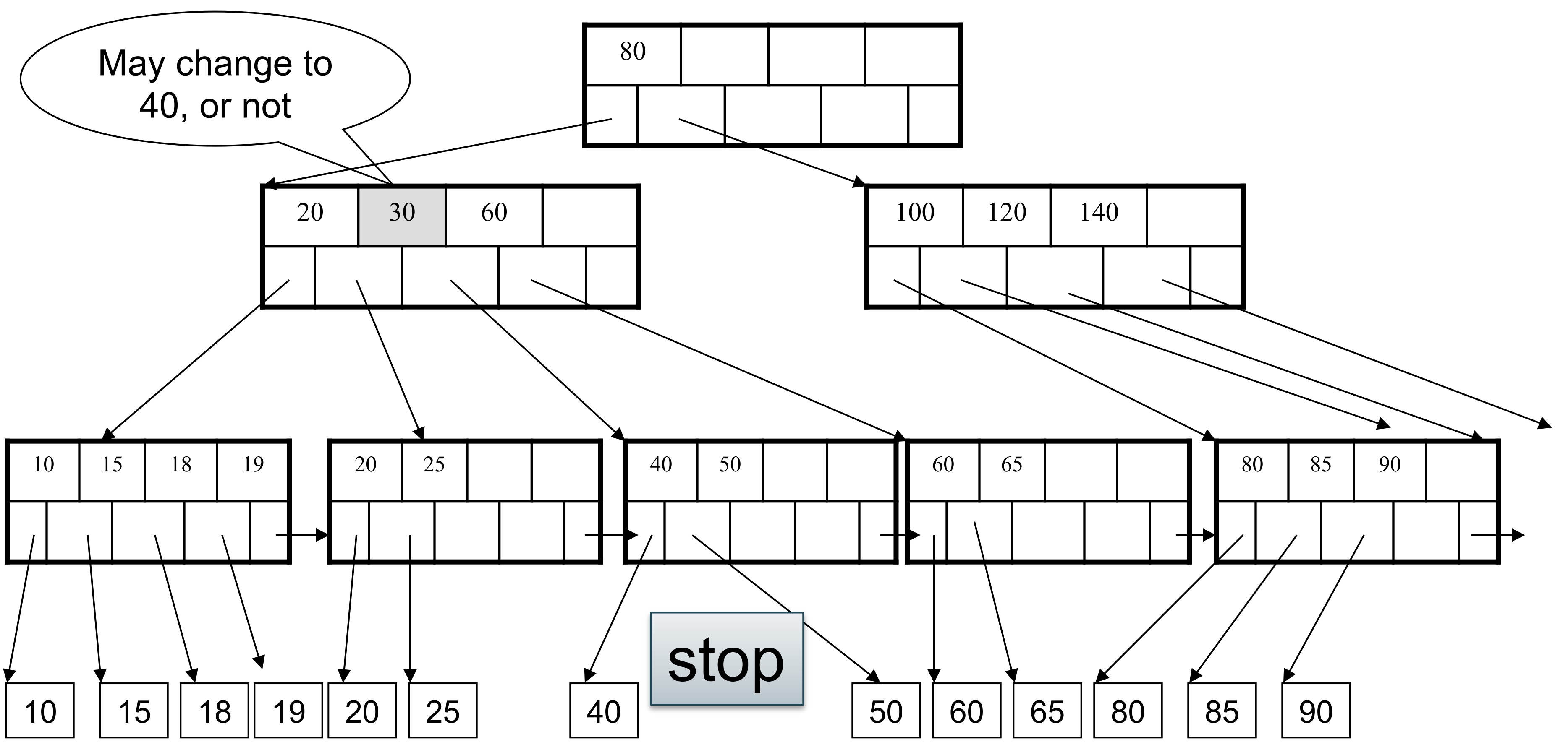
Deletion from a B+ Tree

After deleting 30



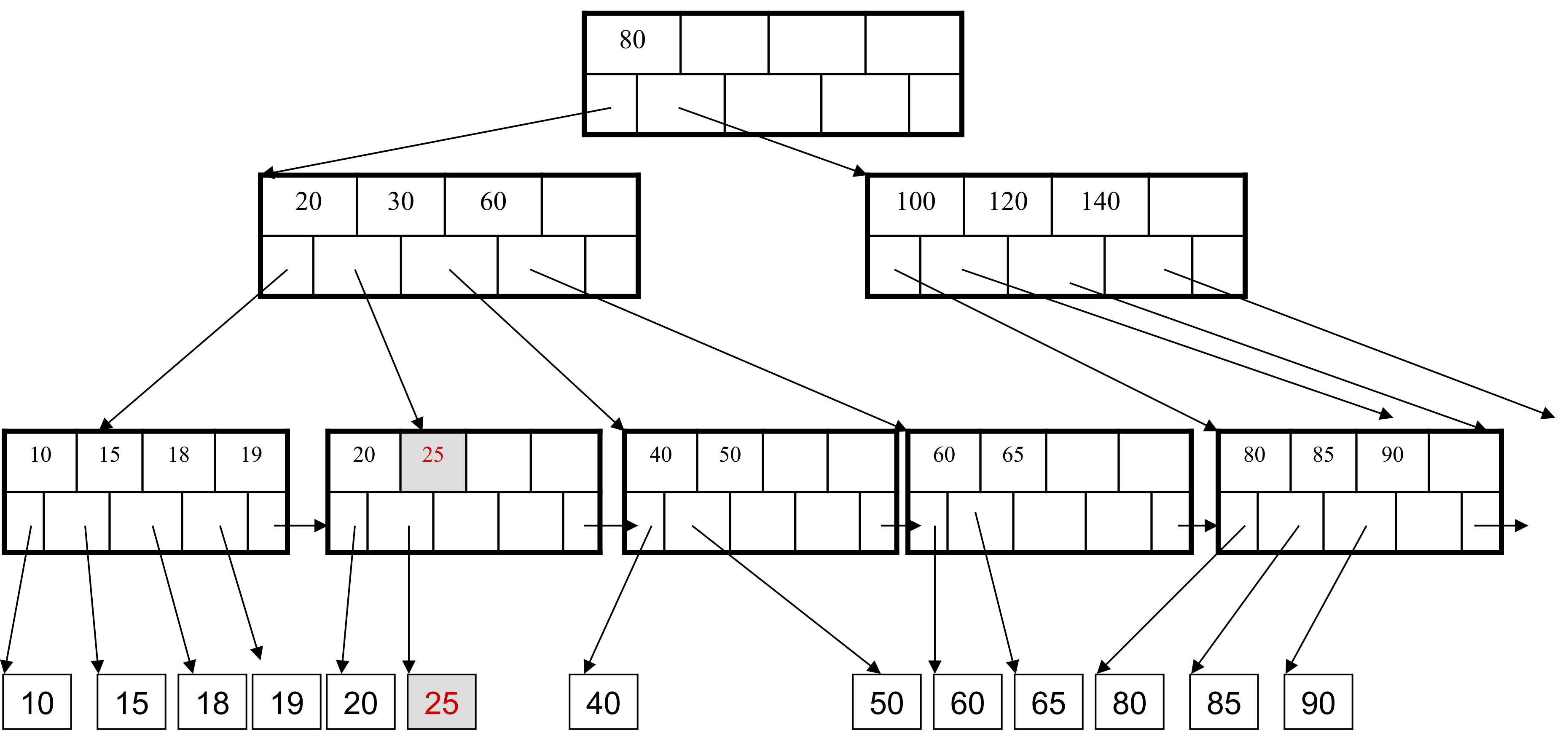
Deletion from a B+ Tree

After deleting 30



Deletion from a B+ Tree

Now delete 25

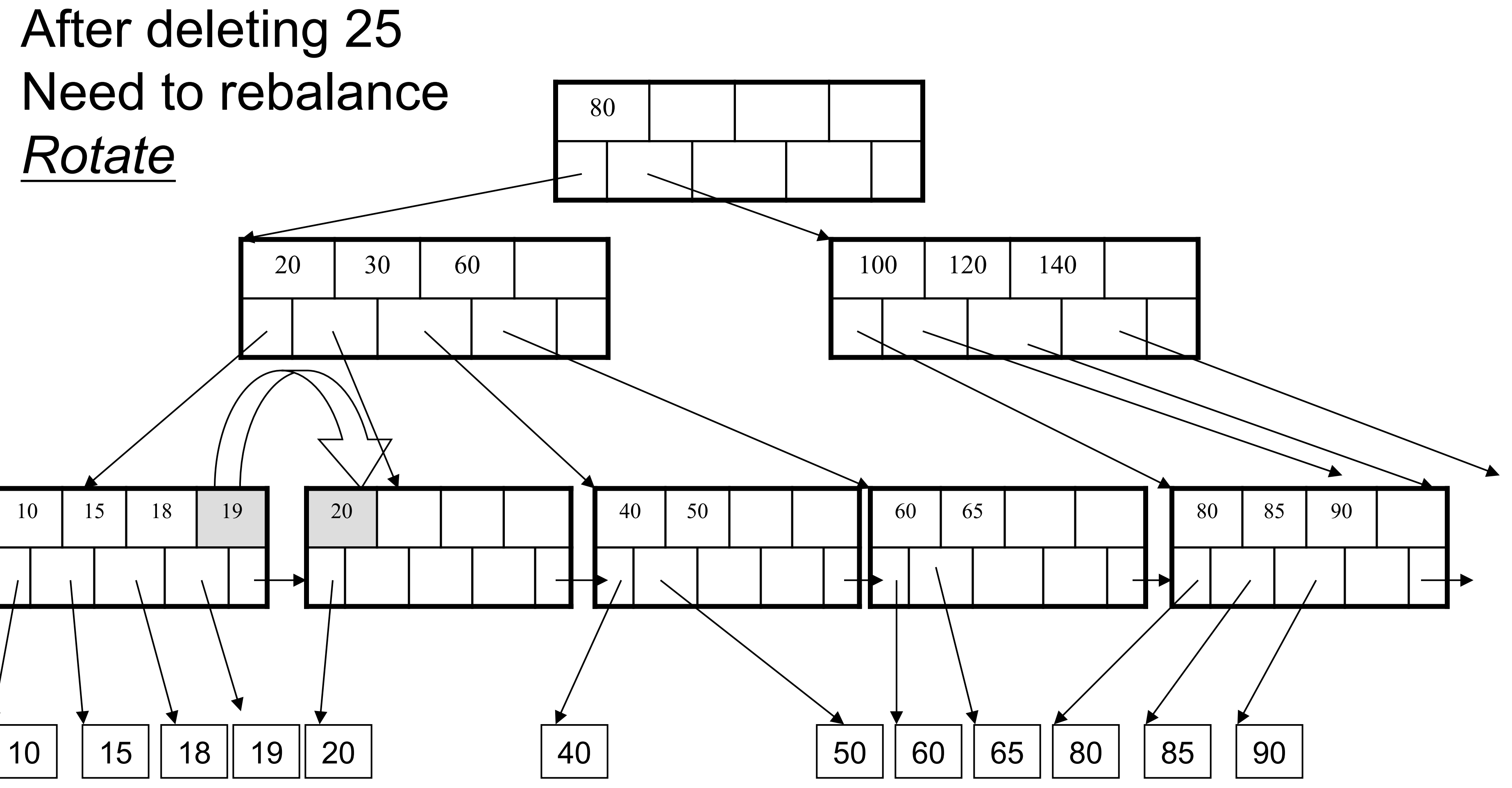


Deletion from a B+ Tree

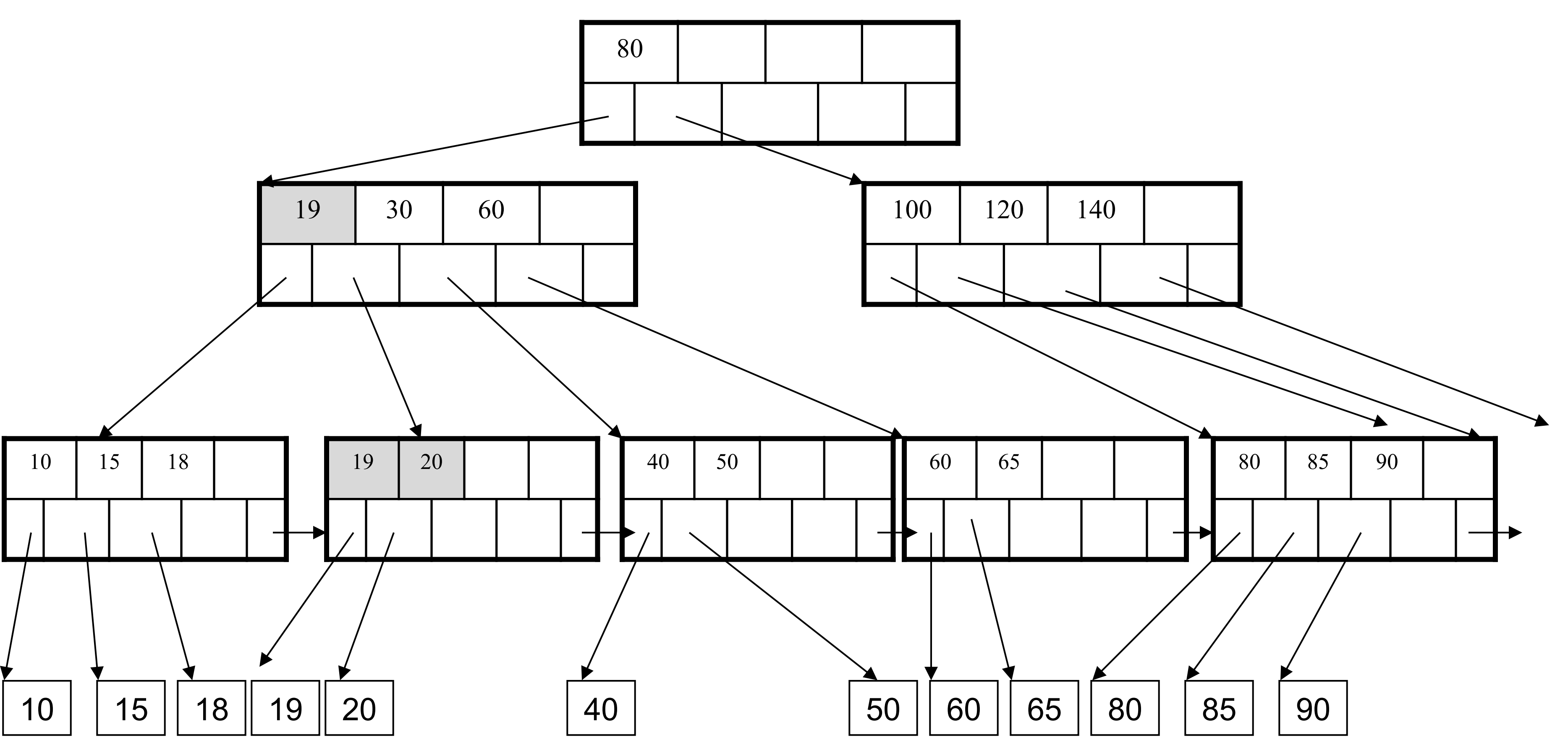
After deleting 25

Need to rebalance

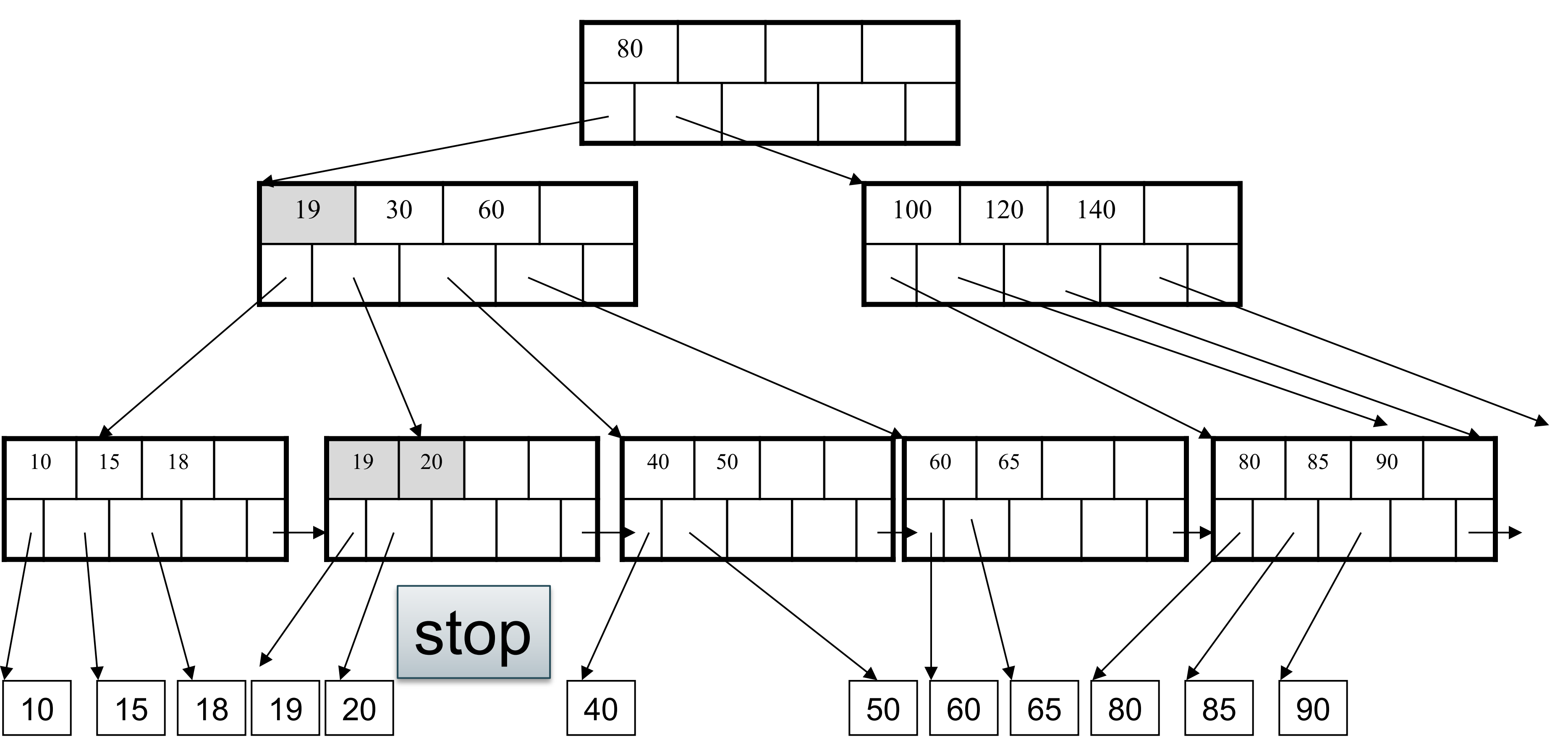
Rotate



Deletion from a B+ Tree

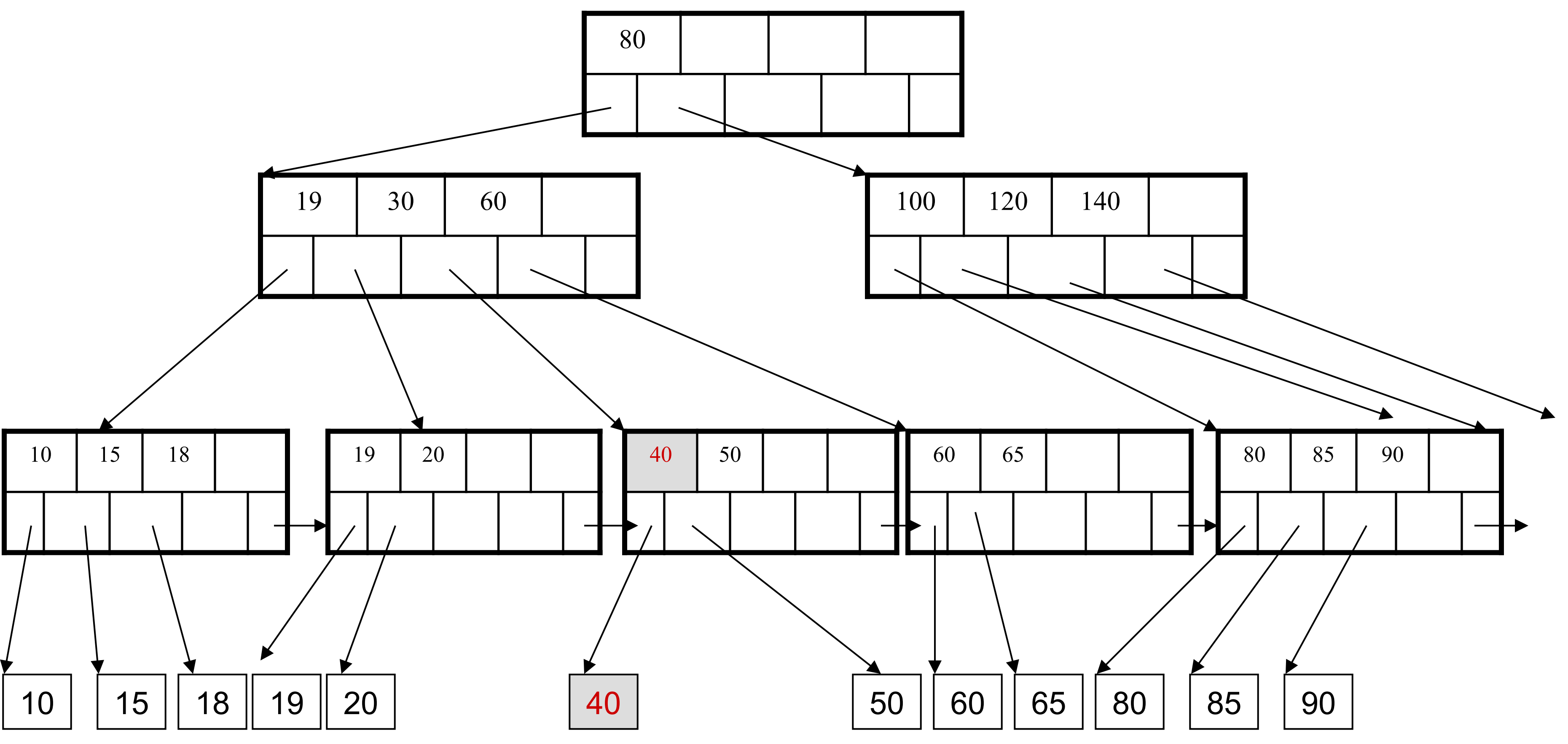


Deletion from a B+ Tree



Deletion from a B+ Tree

Now delete 40

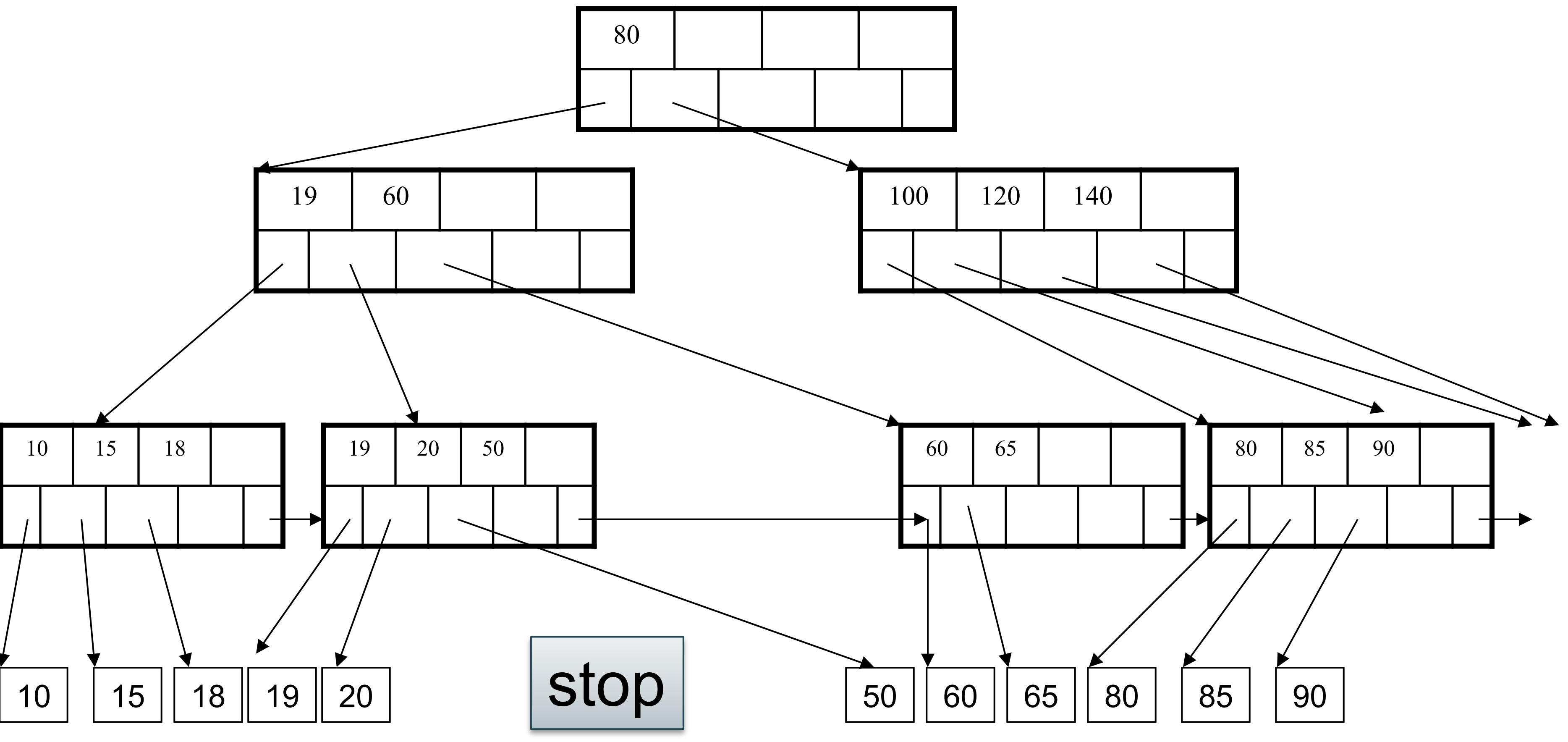


Deletion from a B+ Tree

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

Deletion from a B+ Tree

Final tree

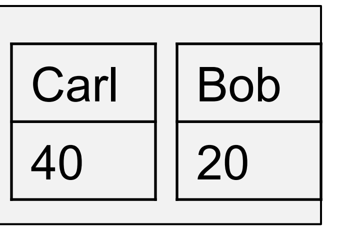


Index in SQL

```
create table Person(name text, age int)
```

Index in SQL

```
create table Person(name text, age int)
```



Alice
50
. . .

. . .

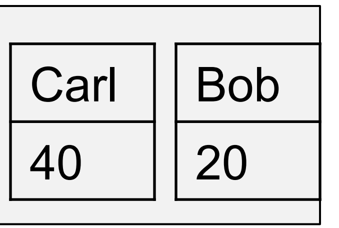
. . .

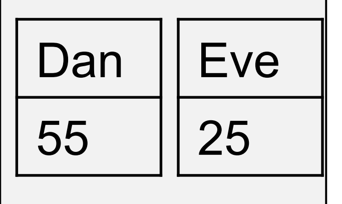
Person data file

Index in SQL

```
create table Person(name text, age int)
```

```
create index pAge on Person(age)
```





Alice
50
…

…

…

Person data file

Index in SQL

create table Person(name text, age int)

create index pAge on Person(age)

pAge index

80

20 25

40 50 55

Carl 40

Bob 20

Dan 55

Eve 25

Alice 50

. . .

. . .

. . .

Person data file

Index in SQL

```
create table Person(name text, age int)
```

```
create index pAge on Person(age)
```

```
select *
from Person
where age = 44
```

This query will execute faster after creating the index

Index in SQL

```
create table Person(name text, age int)
```

```
create index pAge on Person(age)
```

```
select *
from Person
where age = 44
```

```
select *
from Person x, Car y
where x.age = y.age
```

A join will not benefit from this index

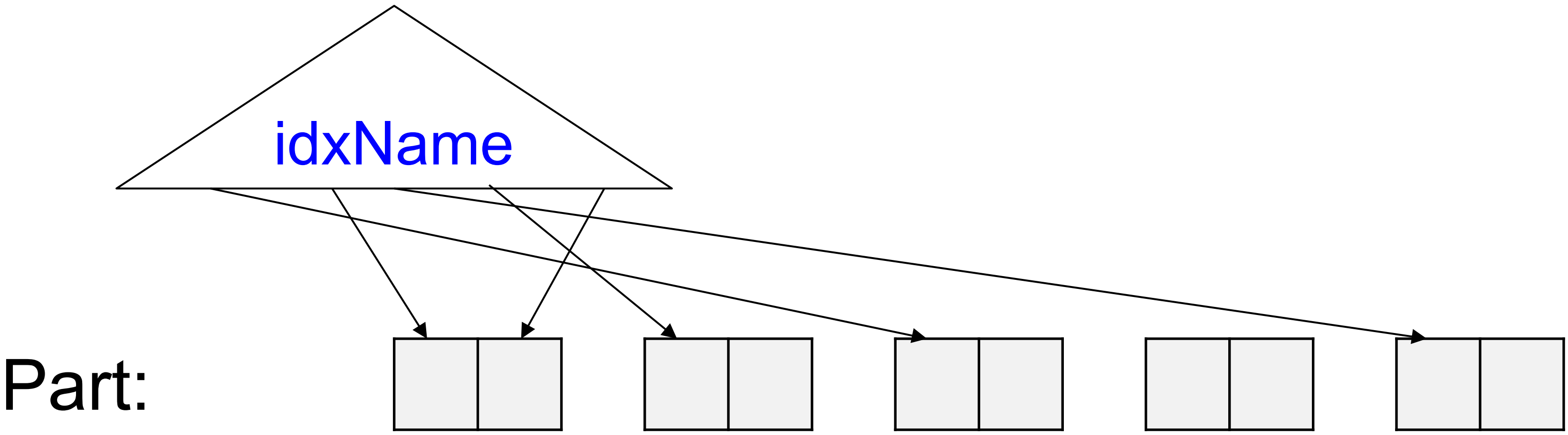
Discussion

- We can create many indexes for a table
- One query can only use one index
- Each update to the table requires updating all indices

Part(pno,pname,psize,pcolor)

Create Index

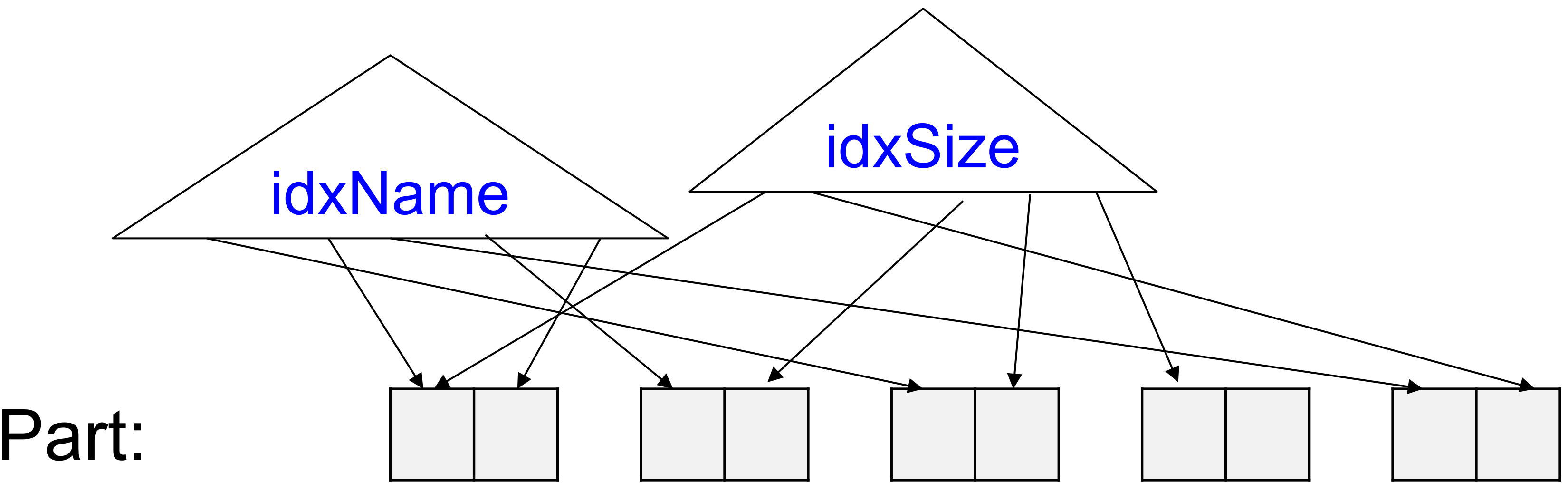
```
create index idxName on Part(pname);
```



Part(pno,pname,psize,pcolor)

Create Index

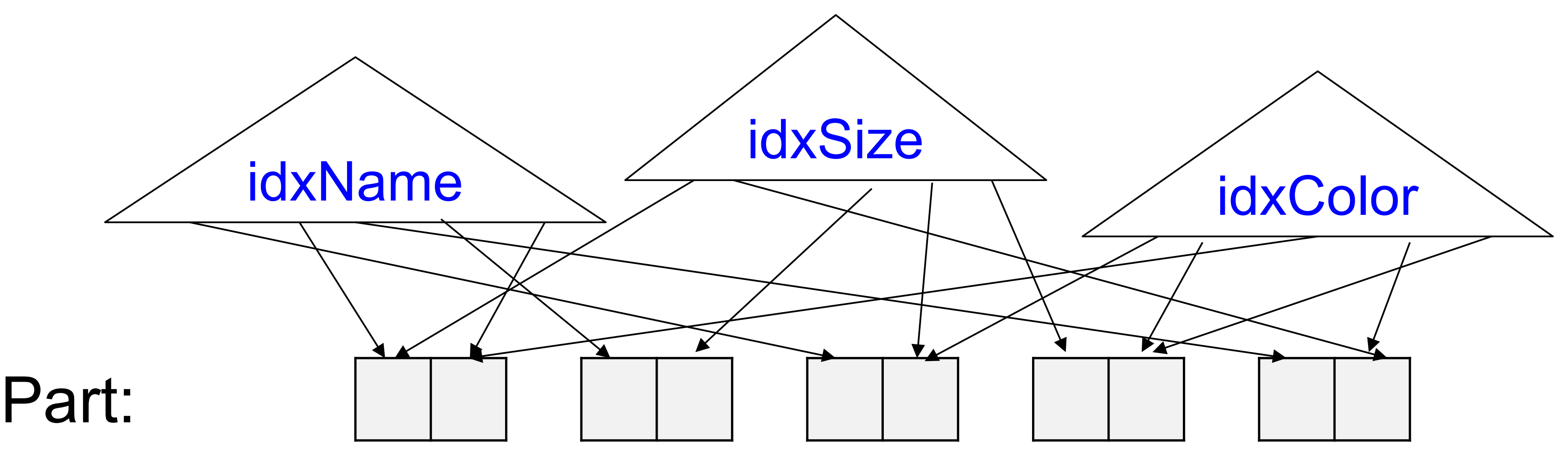
```
create index idxName on Part(pname);
create index idxSize on Part(psize);
```



Part(pno,pname,psize,pcolor)

Create Index

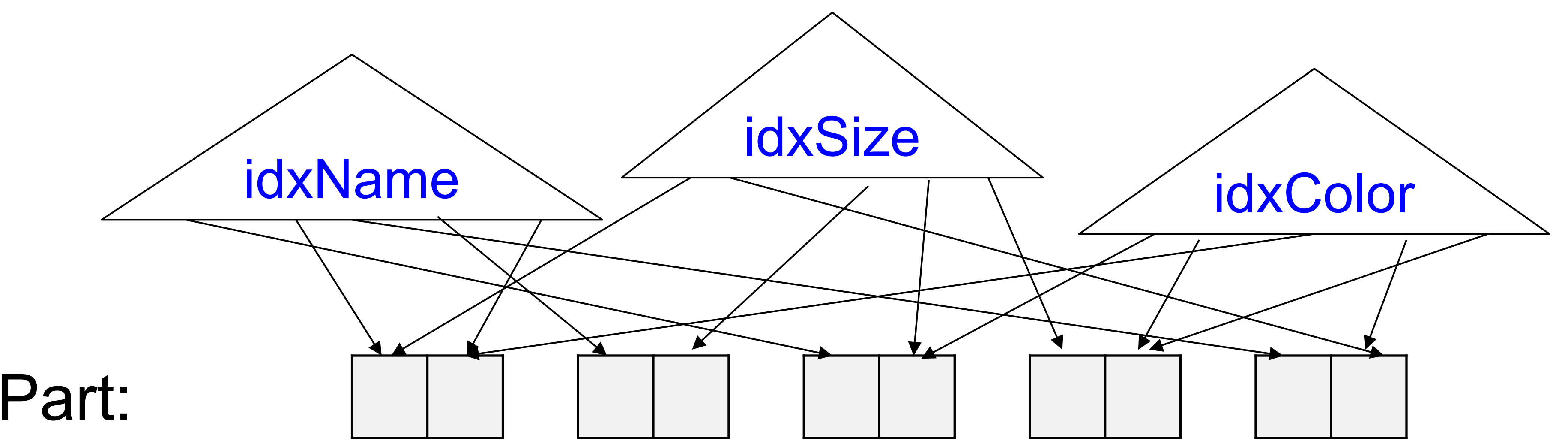
```
create index idxName on Part(pname);
create index idxSize on Part(psize);
create index idxColor on Part(pcolor);
```



Part(<u>pno</u>,pname,psize,pcolor)

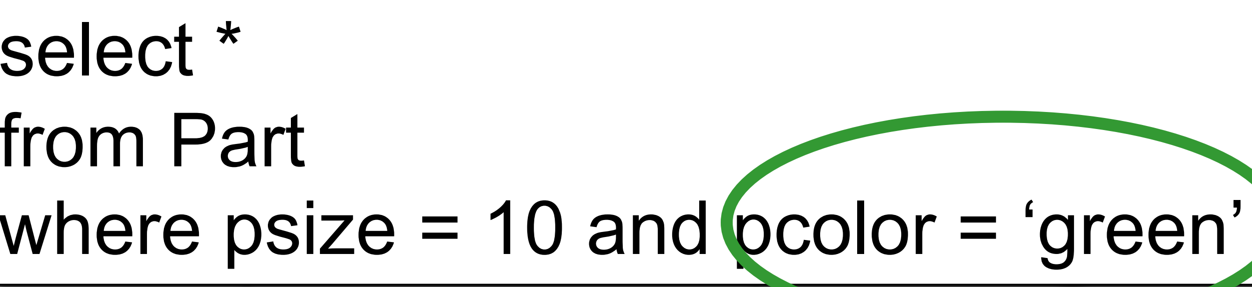
Create Index

```
select *
from Part
where psize = 10 and pcolor = 'green'
```

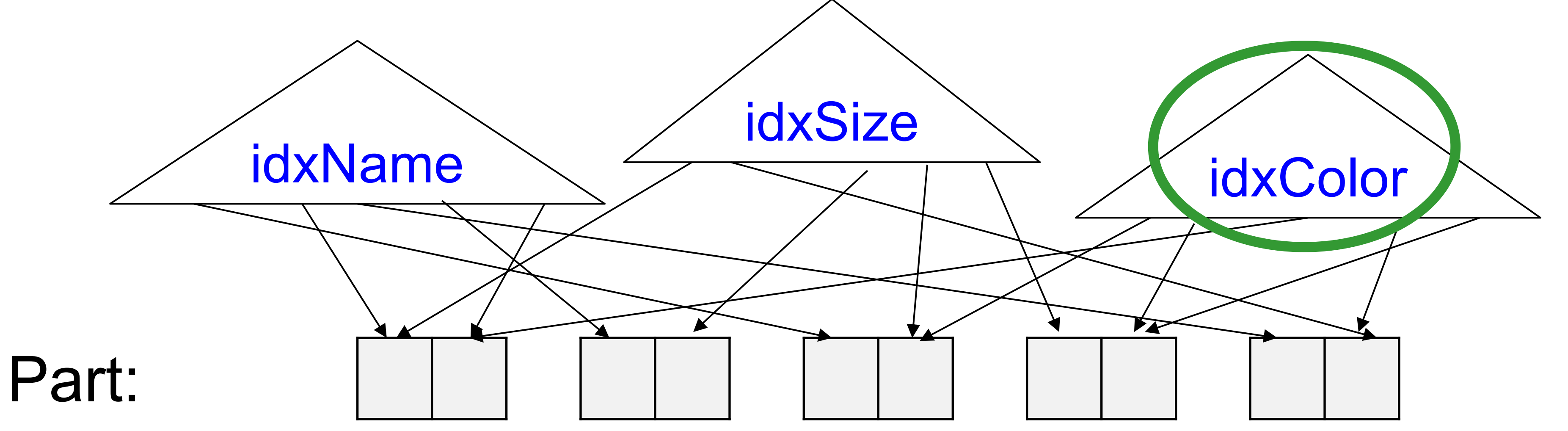


Part(pno,pname,psize,pcolor)

Create Index

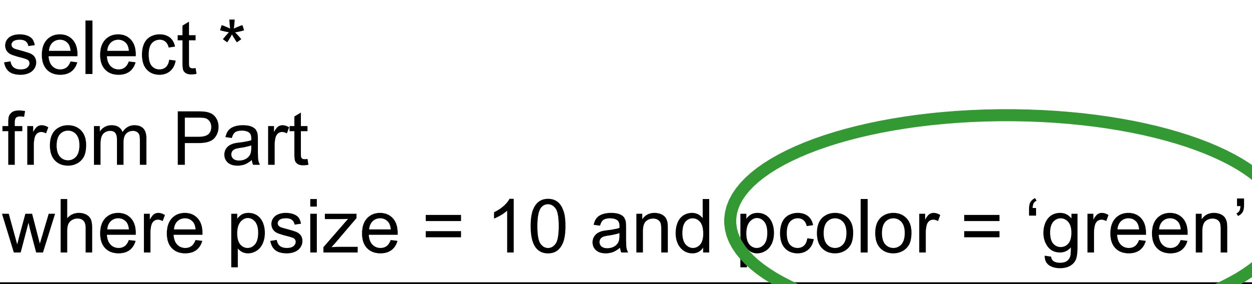


Use idxColor



Part(pno,pname,psize,pcolor)

Create Index



Use idxColor

idxName

idxSize

idxColor

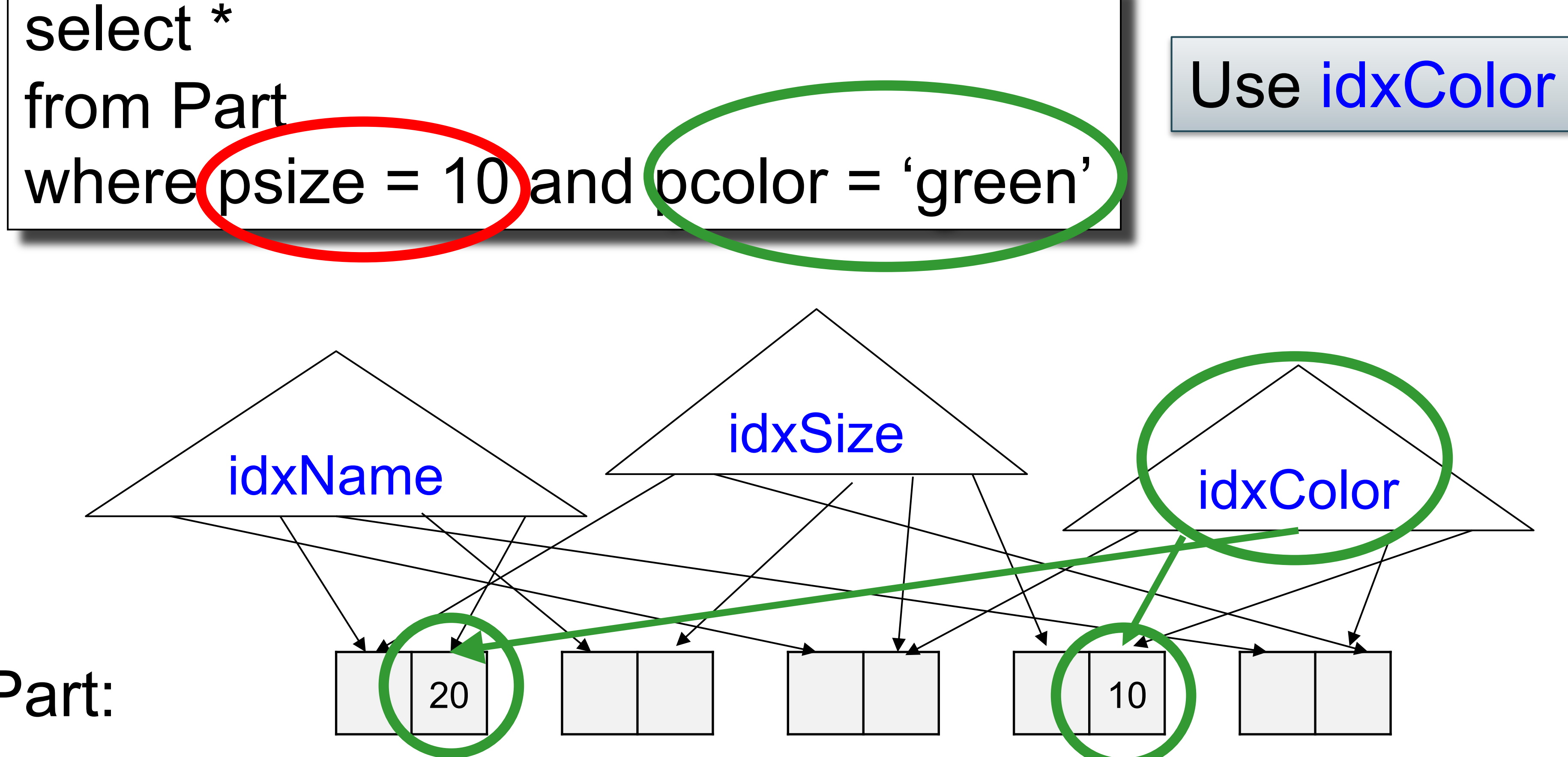
Part:

Part(pno,pname,psize,pcolor)

Create Index

```
select *
from Part
where psize = 10 and pcolor = 'green'
```

Use idxColor



Read all green parts, return those with psize=10

Part(pno,pname,psize,pcolor)

Create Index

Use idxColor

idxName

idxSize

idxColor

Part:

20

10

Read all green parts, return those with psize=10

Discussion

- In general there can be multiple indexes available to compute a where-condition:
  - Attr1=val1 and Attr2=val2 and …

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

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
- Problem is called access path selection

Multi-attribute Index

- We can create an index 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)

Part(pno,pname,psize,pcolor)

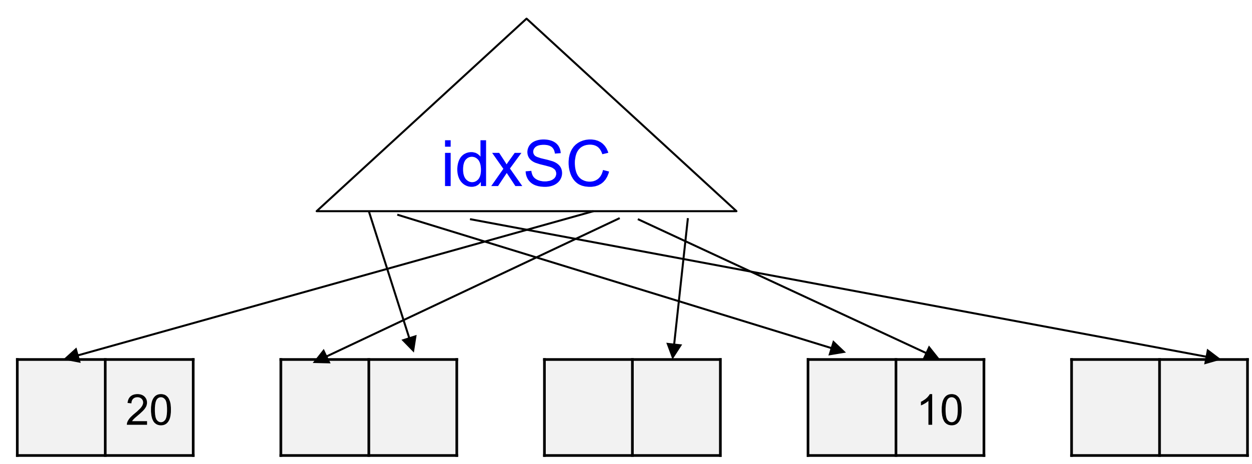
Multi-attribute Index

```
create index idxSC on Part(psize, pcolor);
```

Part(pno,pname,psize,pcolor)

Multi-attribute Index

create index idxSC on Part(psize, pcolor);

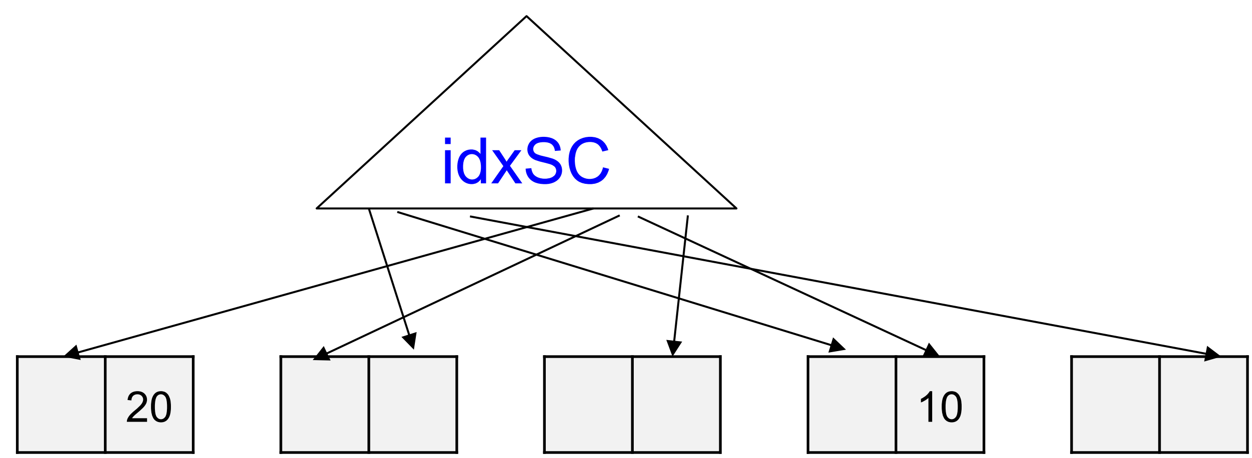


Part(pno,pname,psize,pcolor)

Multi-attribute Index

create index idxSC on Part(psize, pcolor);

select *
from Part
where psize = 10 and pcolor = 'green'

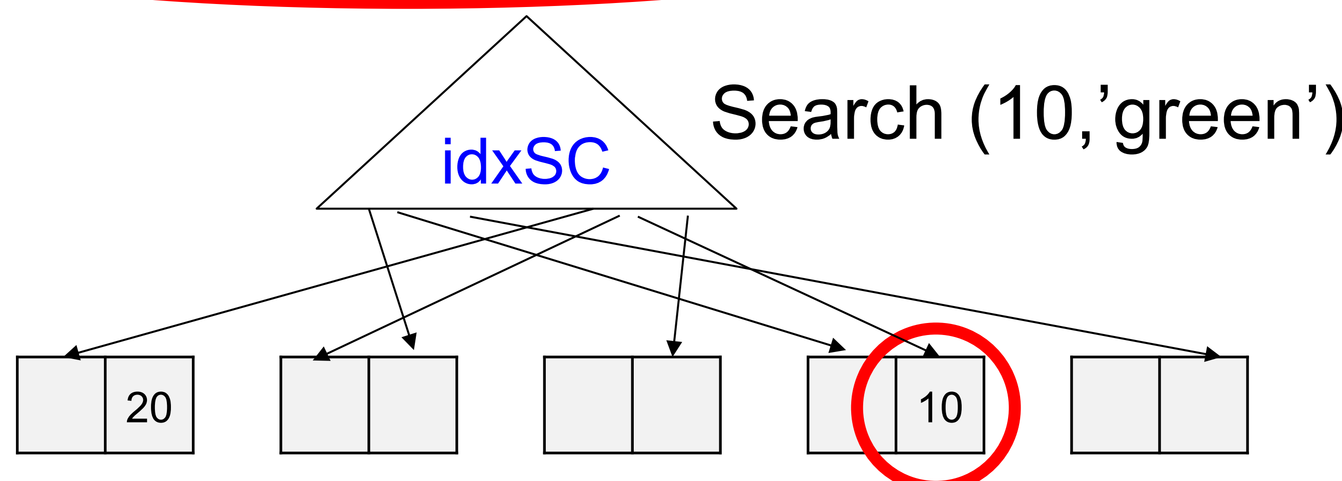


Part(pno,pname,psize,pcolor)

Multi-attribute Index

create index idxSC on Part(psize, pcolor);

select *
from Part
where psize = 10 and pcolor = 'green'

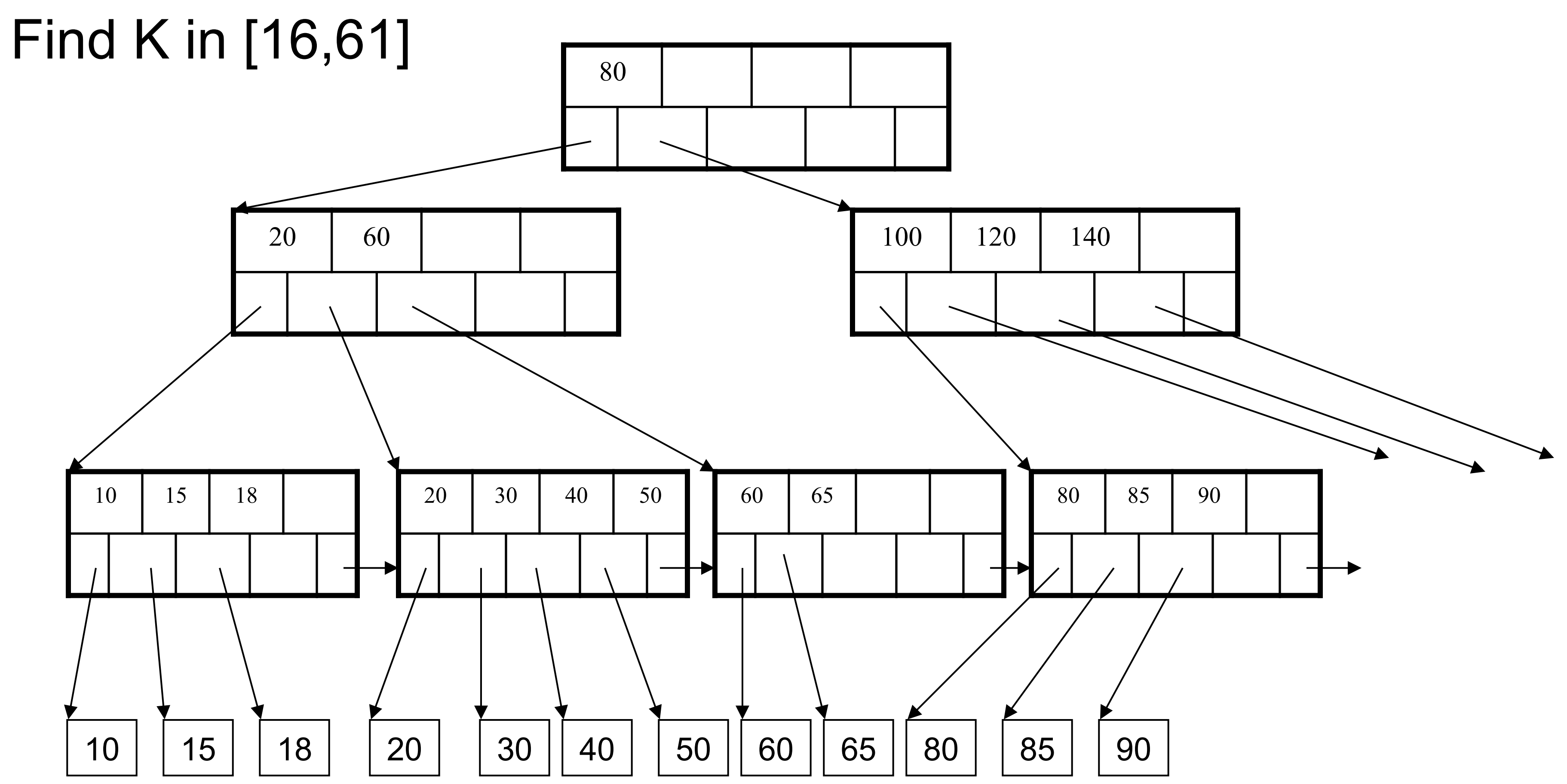


Multi-attribute Index

- A pair ordered lexicographically:
  - (Smith,Alice) < (Smith,Bob)<(Yu,Alice)
- Only 1st attribute known: range seach
  - Find (Smith, *)
- Only 2nd attribute known: impossible
  - Find (*, Alice)

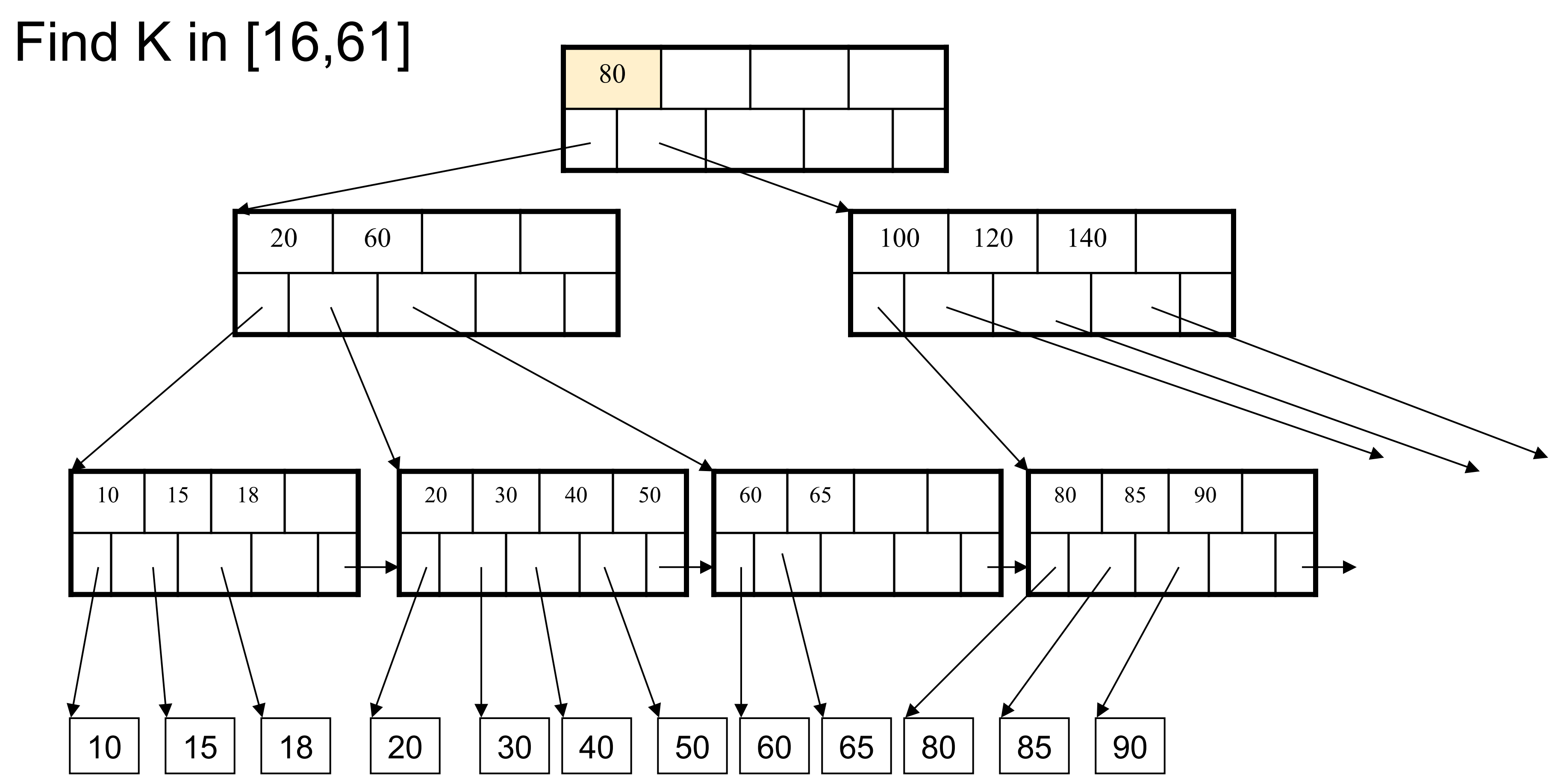
Range Search in B+ Tree

Find K in [16,61]



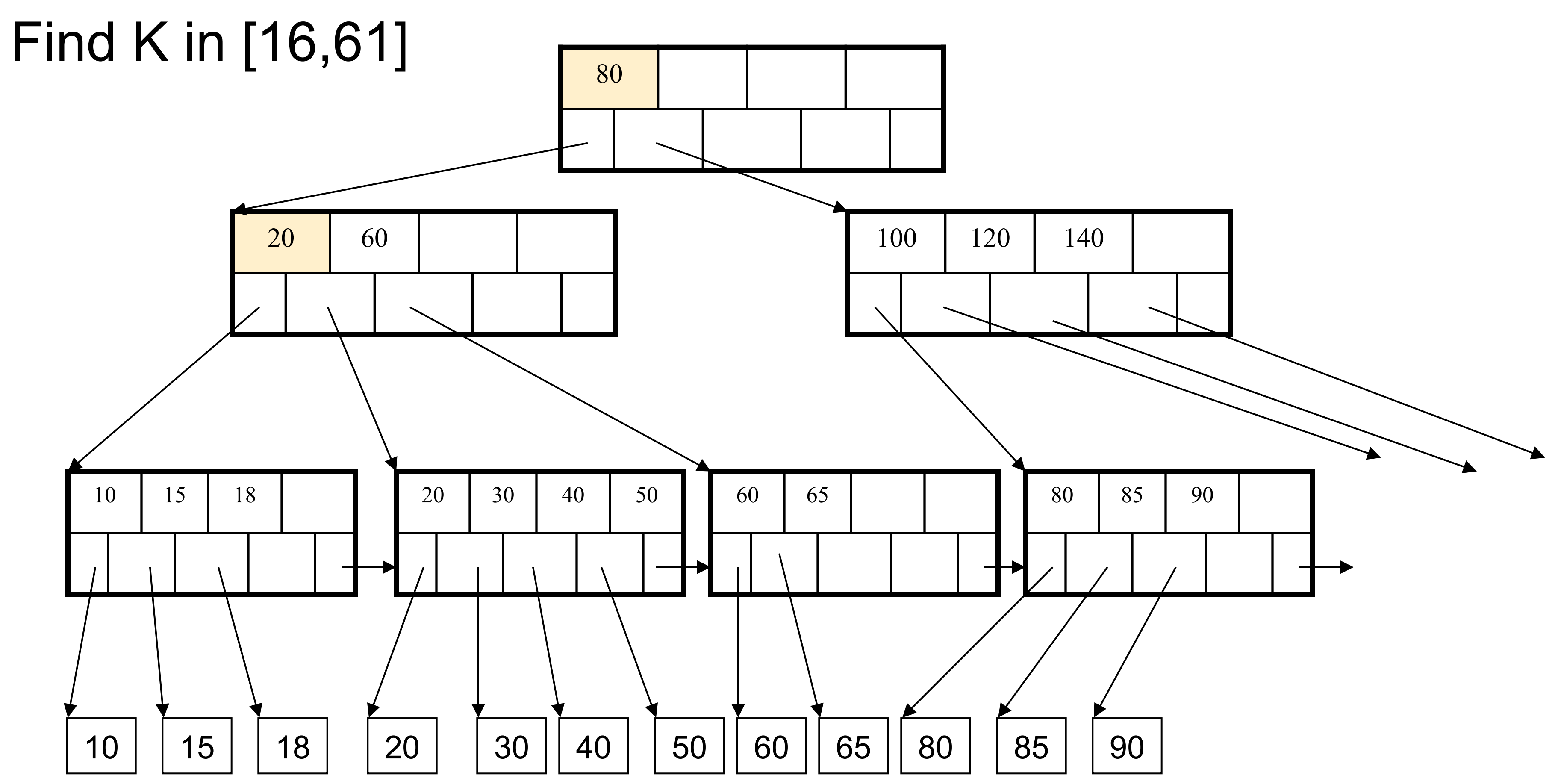
Range Search in B+ Tree

Find K in [16,61]



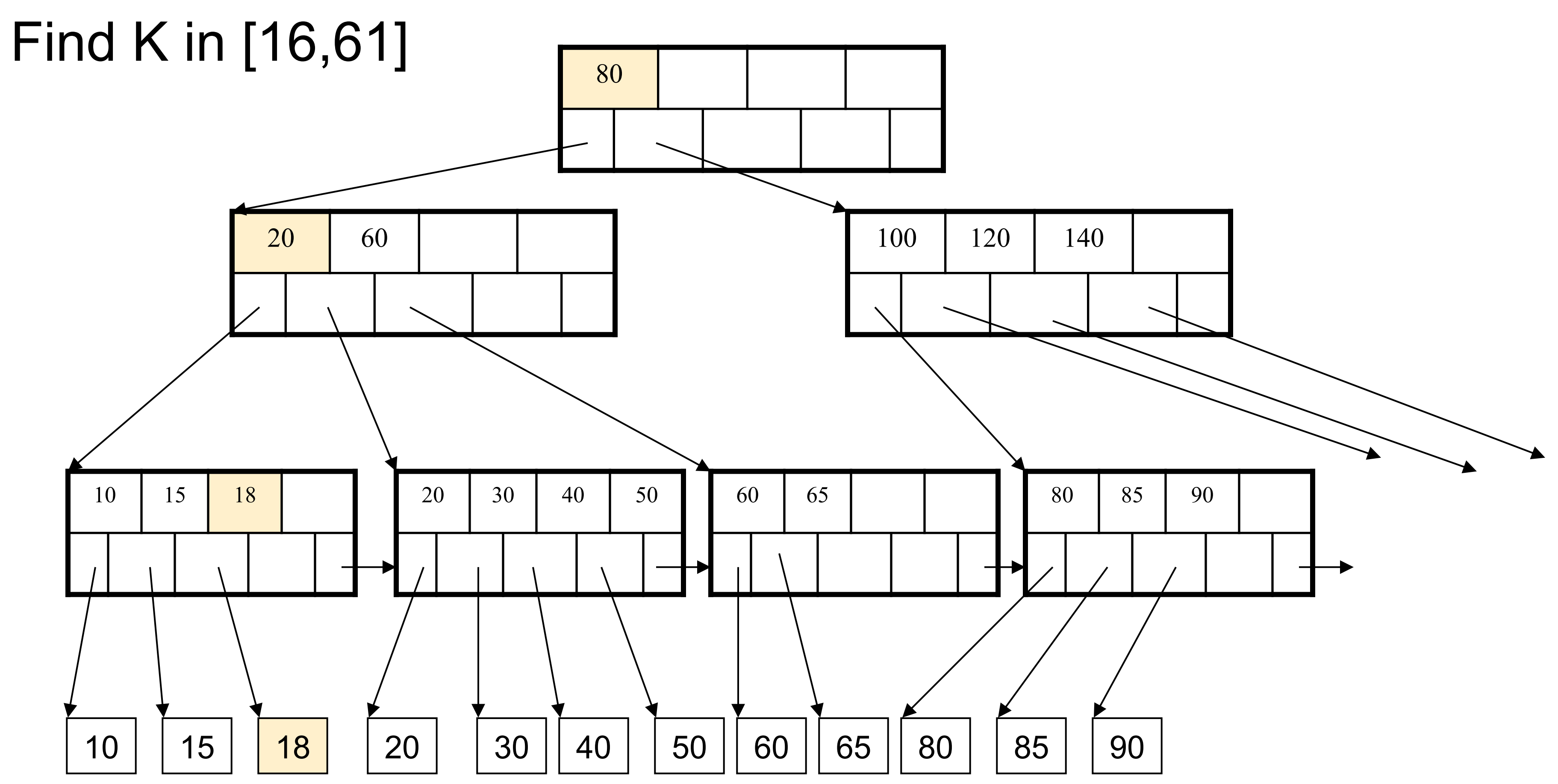
Range Search in B+ Tree

Find K in [16,61]



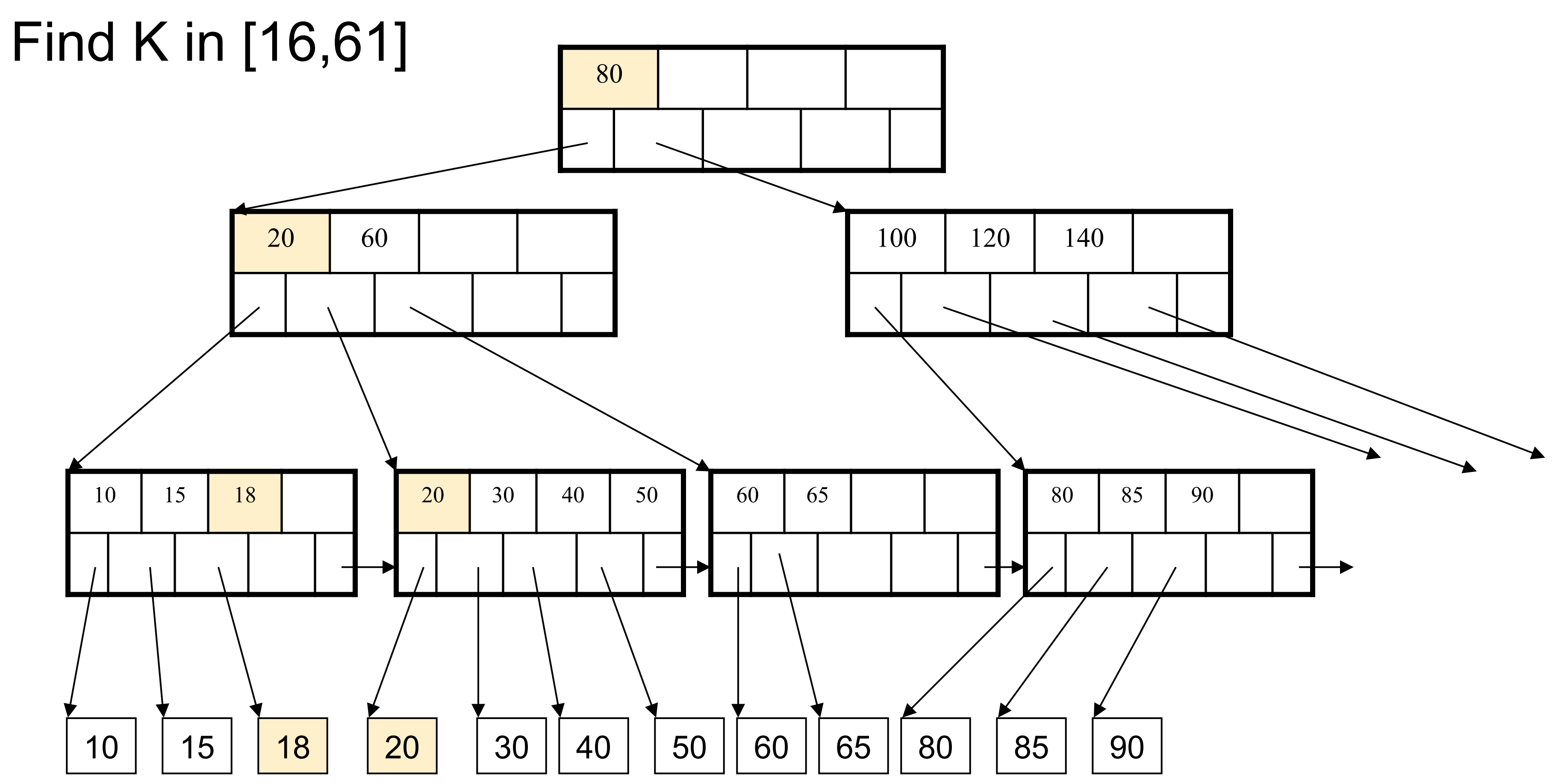
Range Search in B+ Tree

Find K in [16,61]



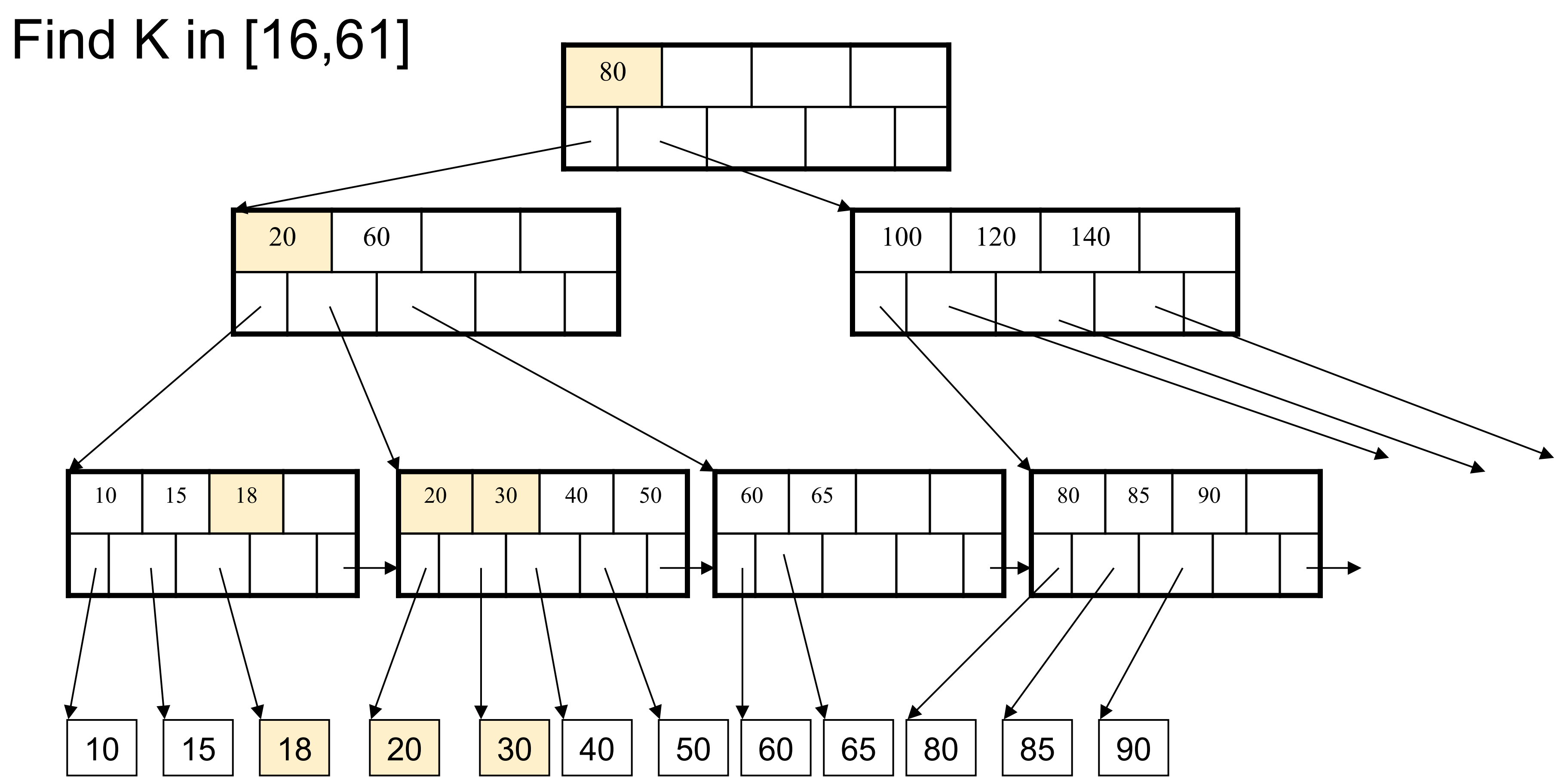
Range Search in B+ Tree

Find K in [16,61]



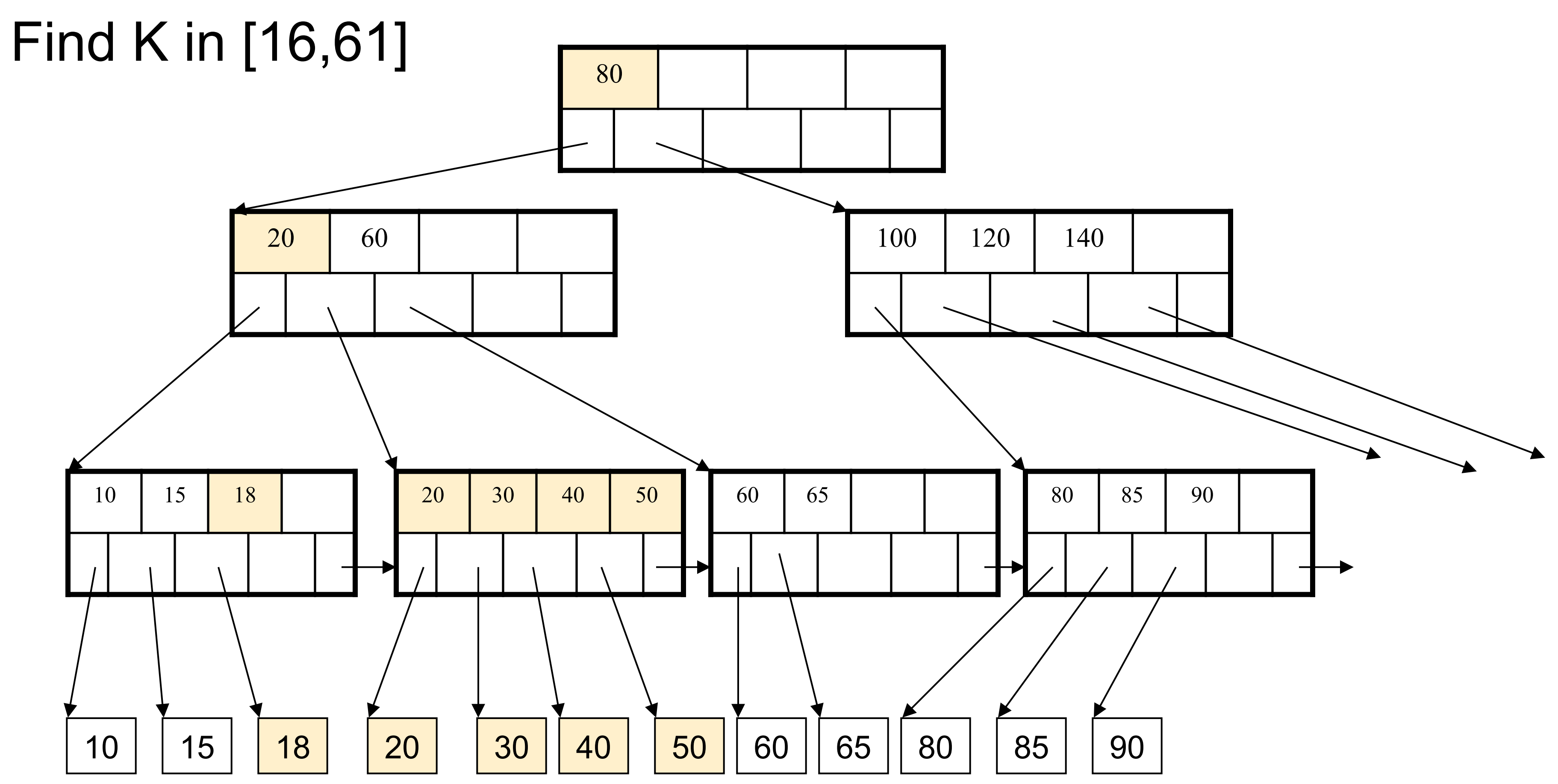
Range Search in B+ Tree

Find K in [16,61]



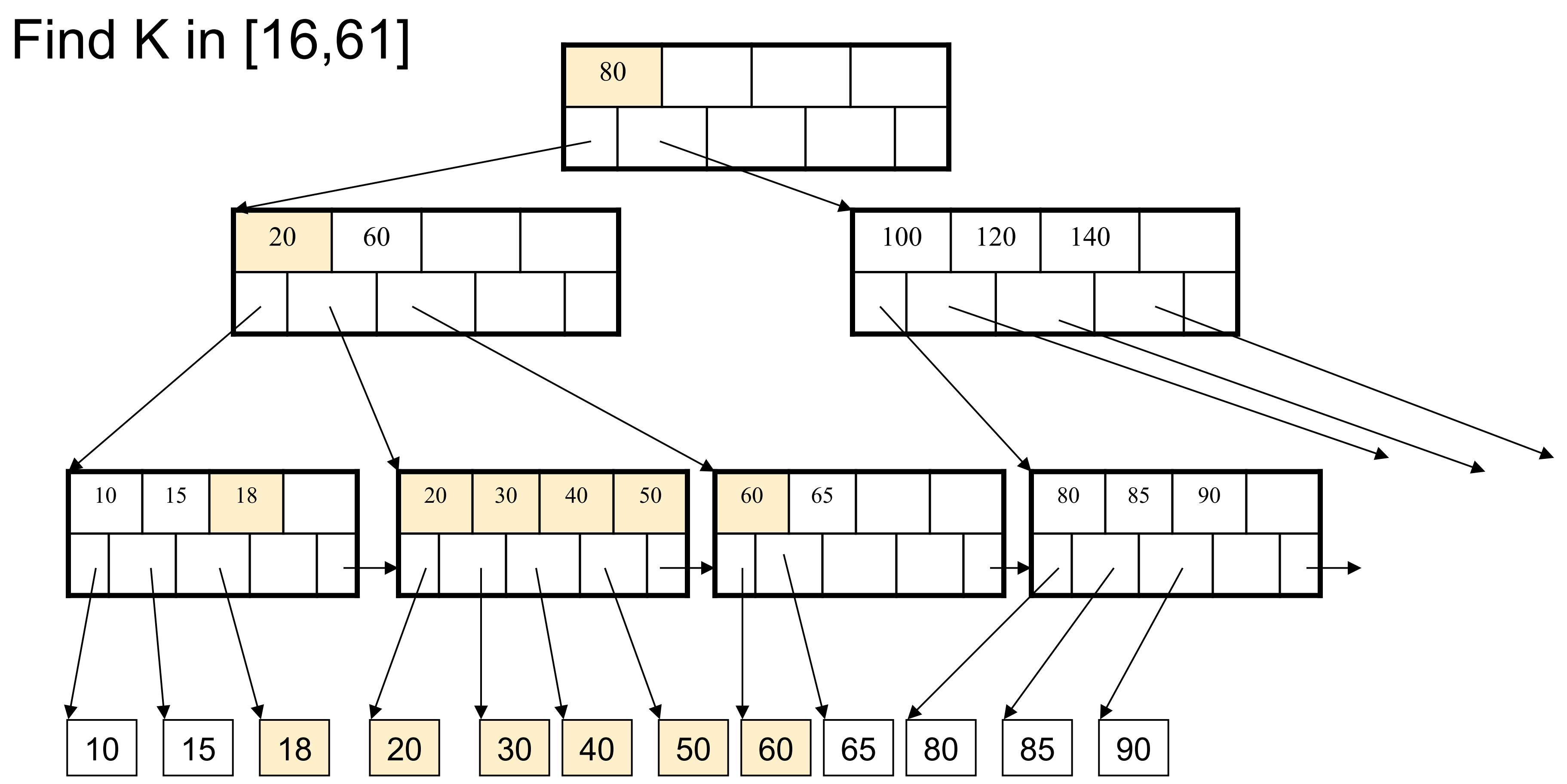
Range Search in B+ Tree

Find K in [16,61]



Range Search in B+ Tree

Find K in [16,61]



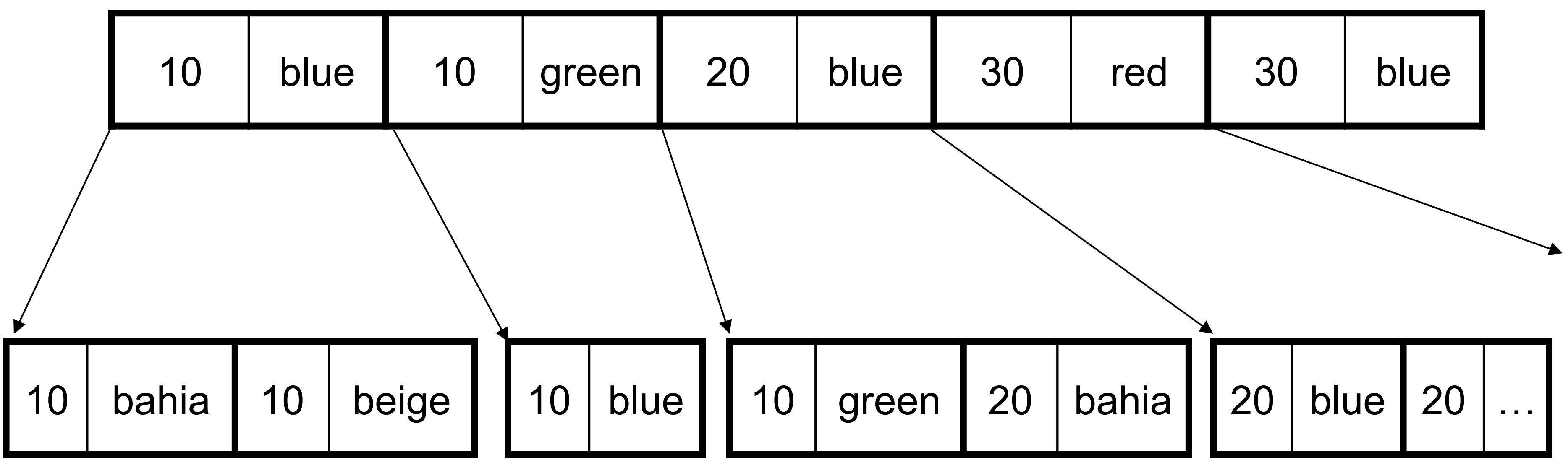
Multi-Attribute Index

- If we only know a prefix of a multi-attribute index, then we can use a range-search

- If we know only a subset that is not a prefix, then we cannot use the index

```
create index idxSC on Part(psize, pcolor);
```

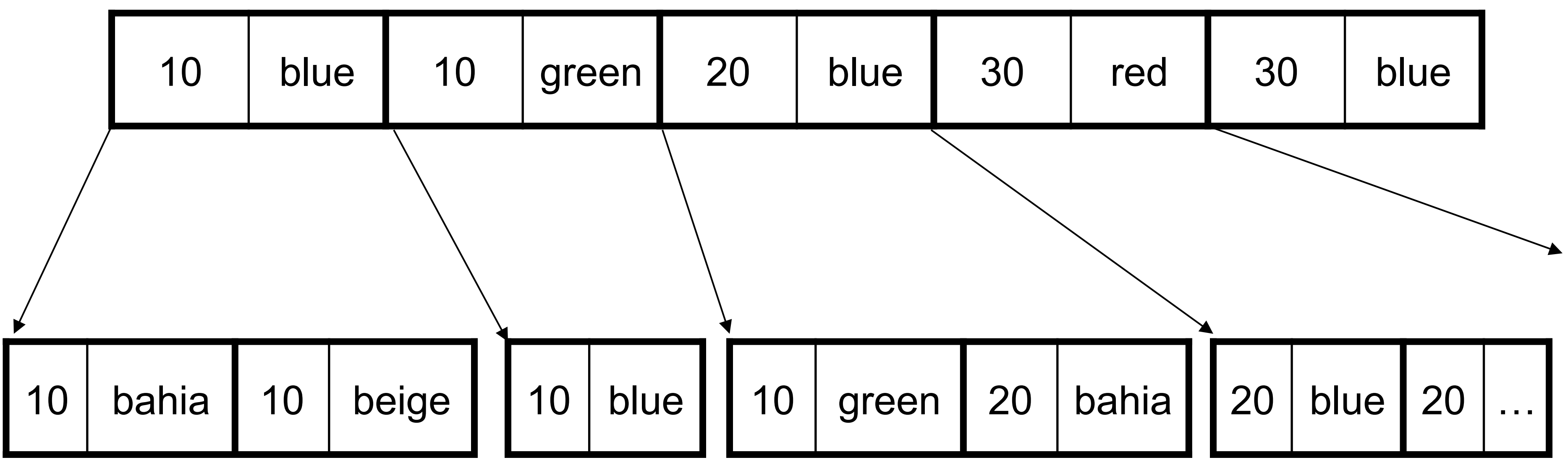
Multi-Attribute Index



```
create index idxSC on Part(psize, pcolor);
```

Multi-Attribute Index

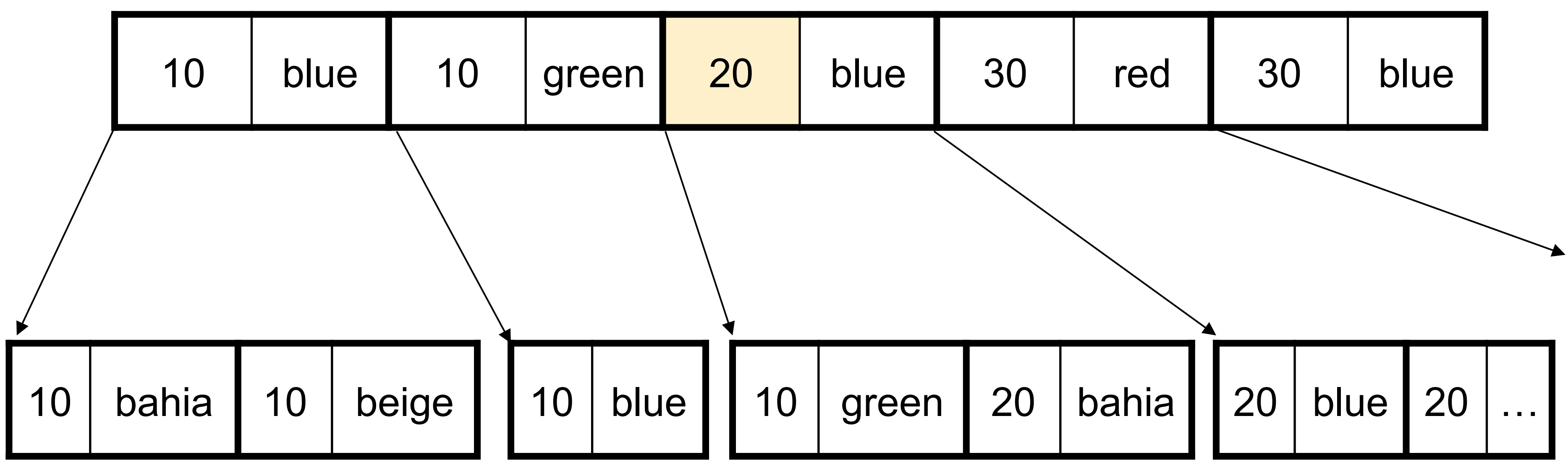
psize = 20, pcolor = *



create index idxSC on Part(psize, pcolor);

Multi-Attribute Index

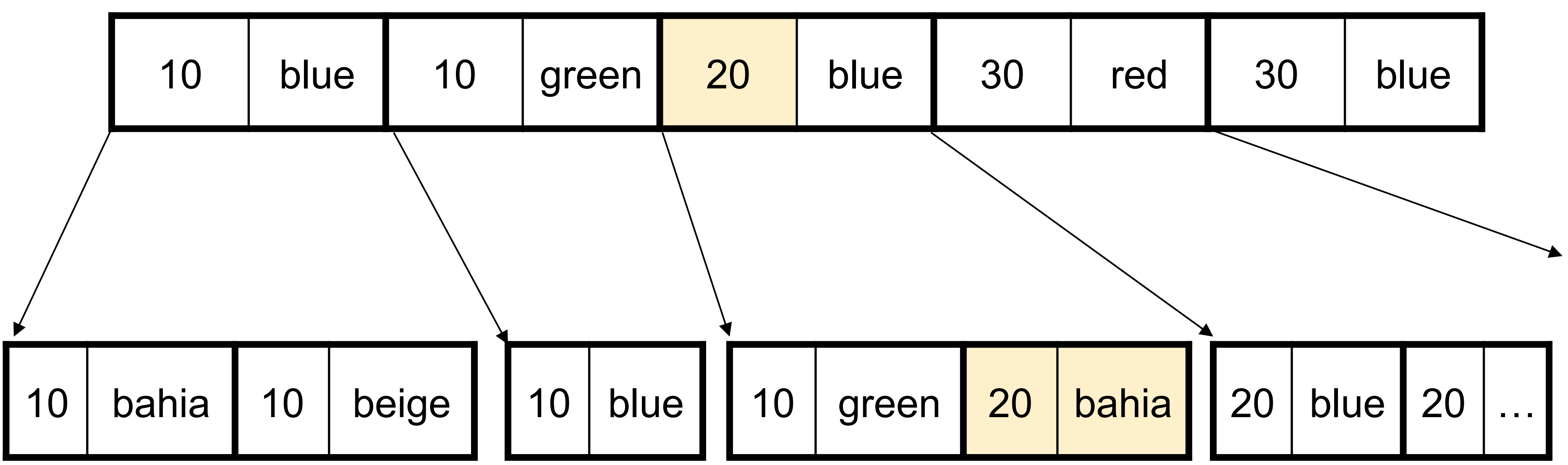
psize = 20, pcolor = *



create index idxSC on Part(psize, pcolor);

Multi-Attribute Index

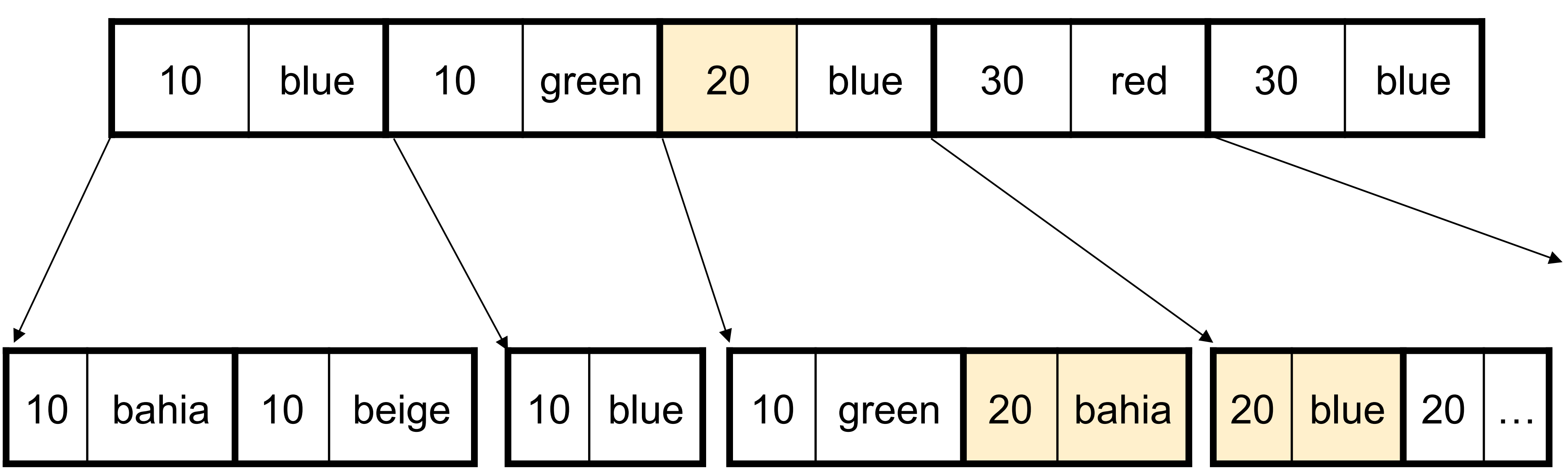
psize = 20, pcolor = *



create index idxSC on Part(psize, pcolor);

Multi-Attribute Index

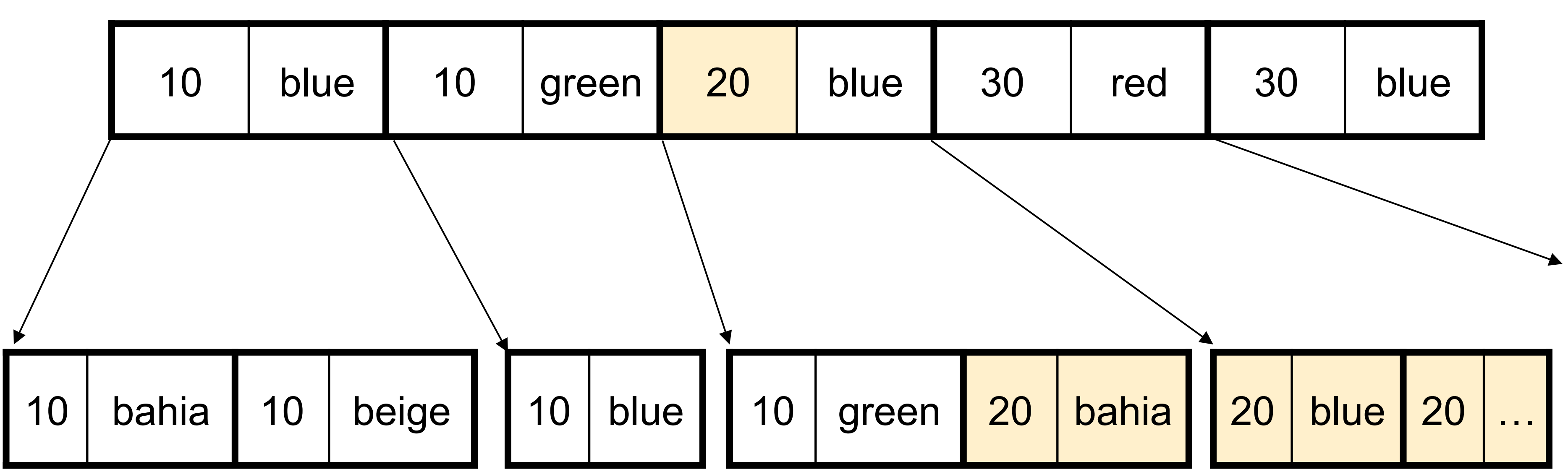
psize = 20, pcolor = *



create index idxSC on Part(psize, pcolor);

Multi-Attribute Index

psize = 20, pcolor = *



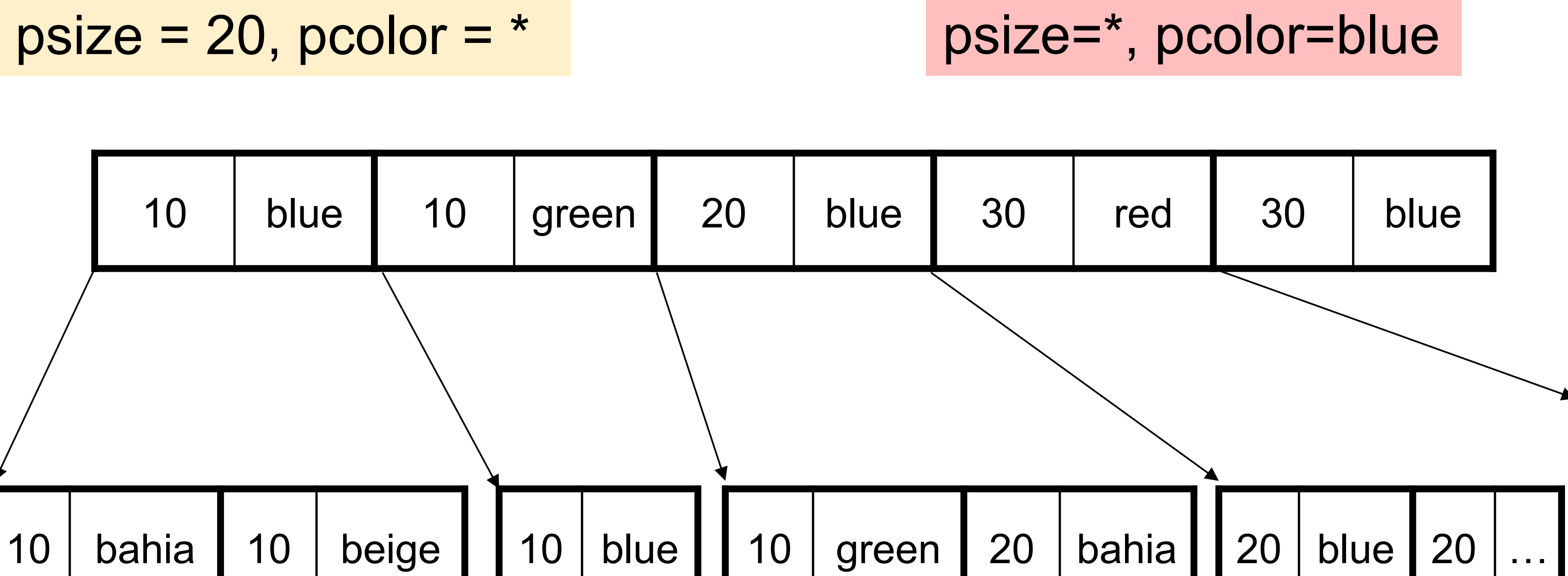
Index is useful

create index idxSC on Part(psize, pcolor);

Multi-Attribute Index

psize = 20, pcolor = *

psize=*, pcolor=blue

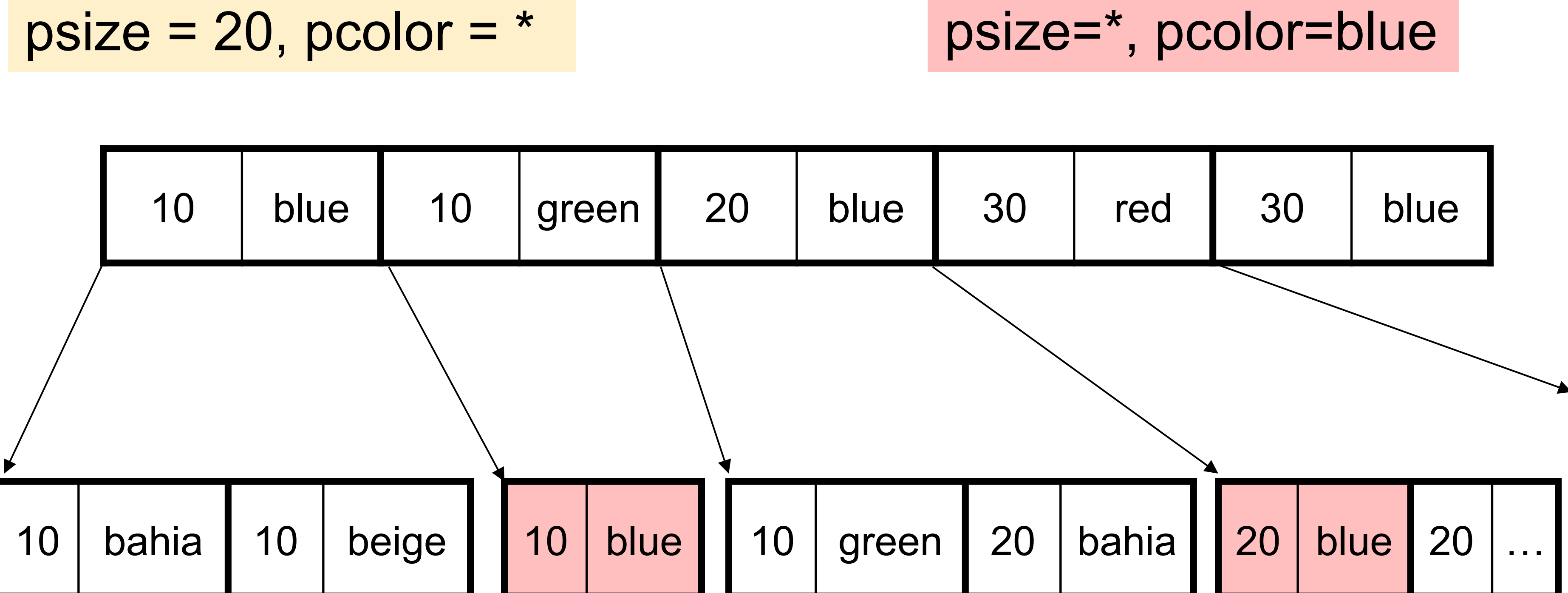


create index idxSC on Part(psize, pcolor);

Multi-Attribute Index

psize = 20, pcolor = *

psize=*, pcolor=blue



Index is not useful

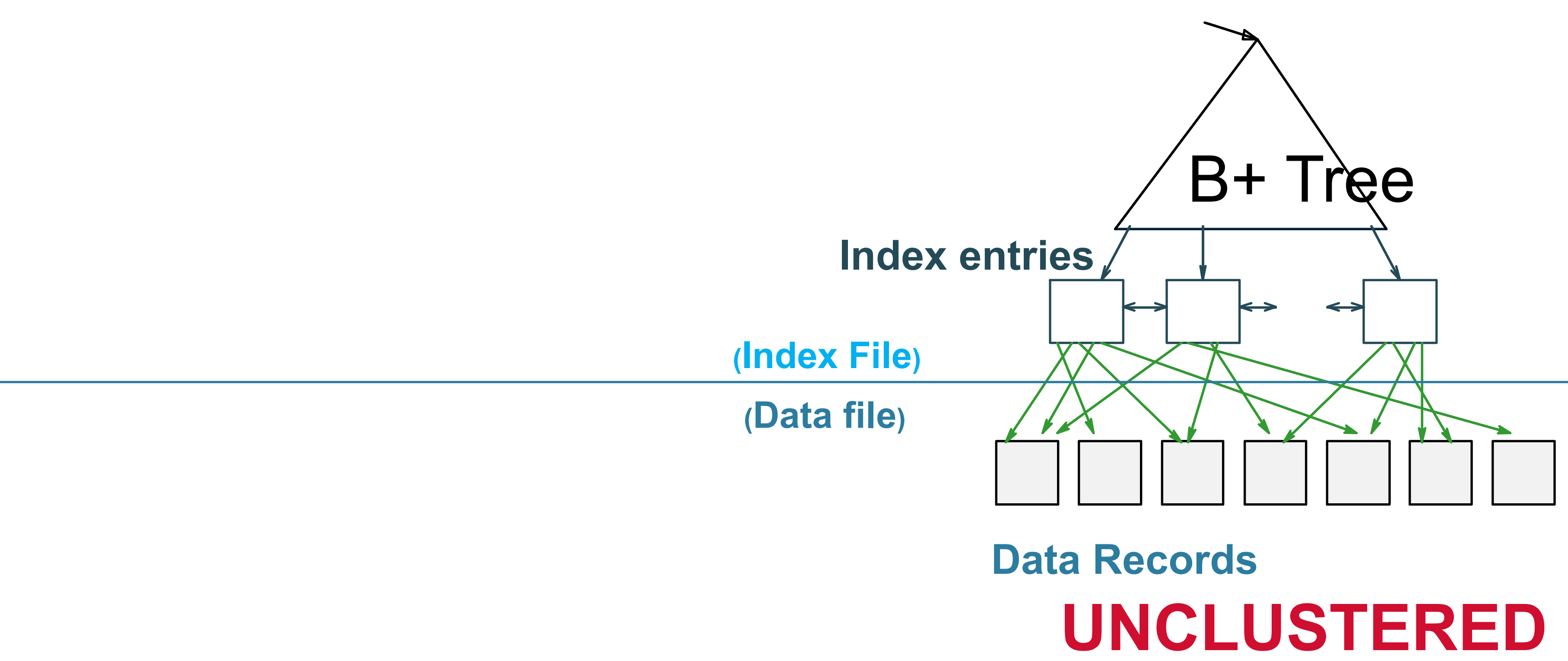
Discussion

- So far, we assumed that the leaves are pointers to records in the data file
  - Pointer = (pageID, slotNumber)
  - Range search → random reads
- Solution: clustered index
  - Range search → sequential reads

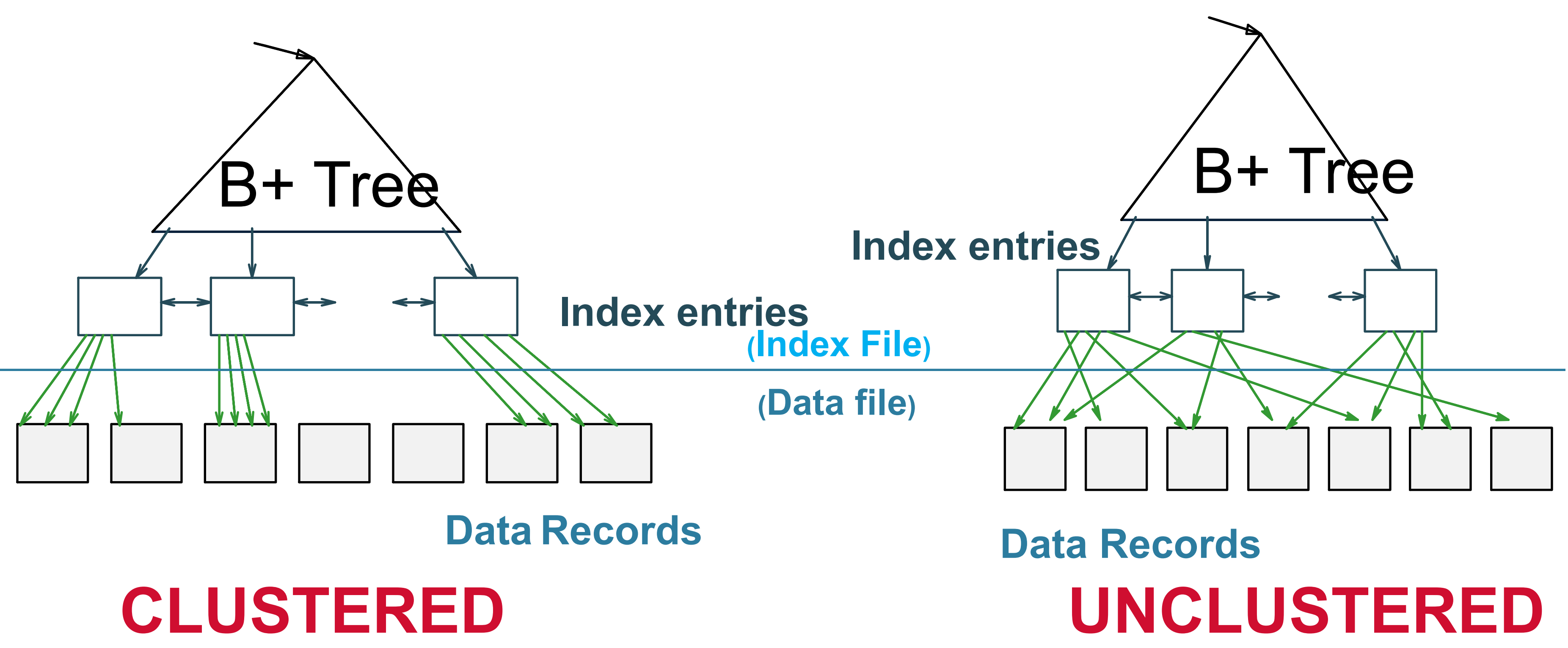
Clustered Index

- A clustered index is sorted in the same order as the data file
  - Better: leaves of the index = data file
  - At most one clustered index per table
- An unclustered index: everything else
  - Many unclustered indexes per table

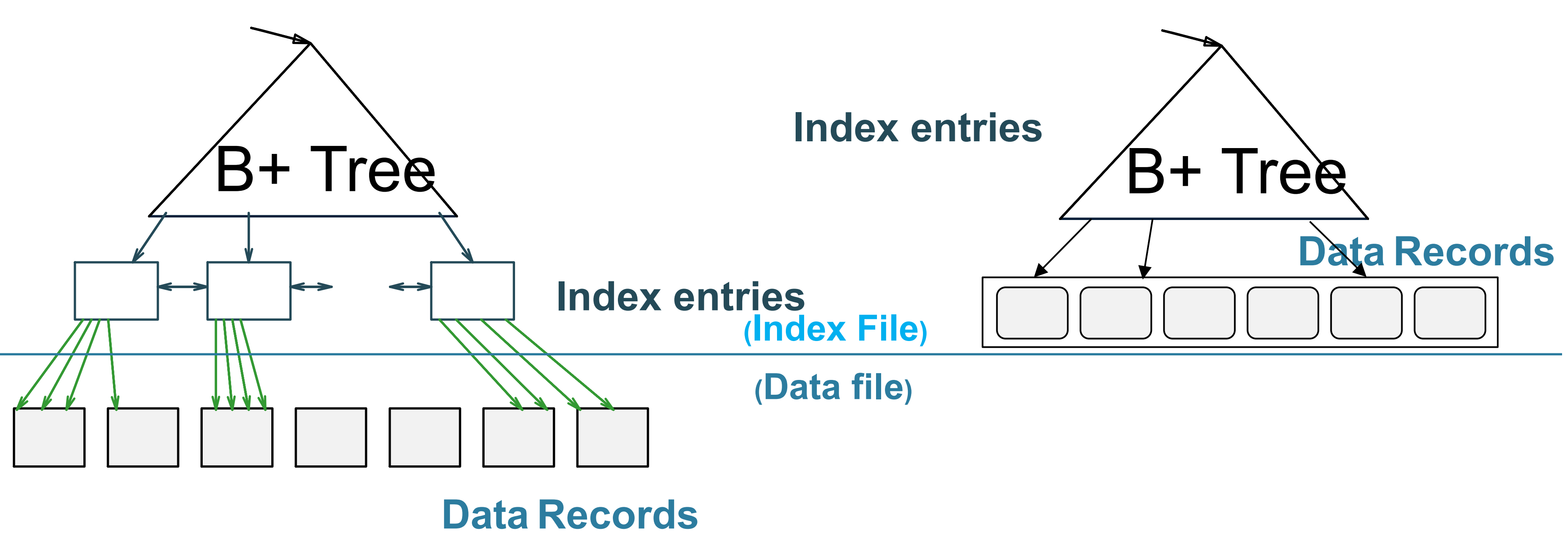
Clustered vs Unclustered



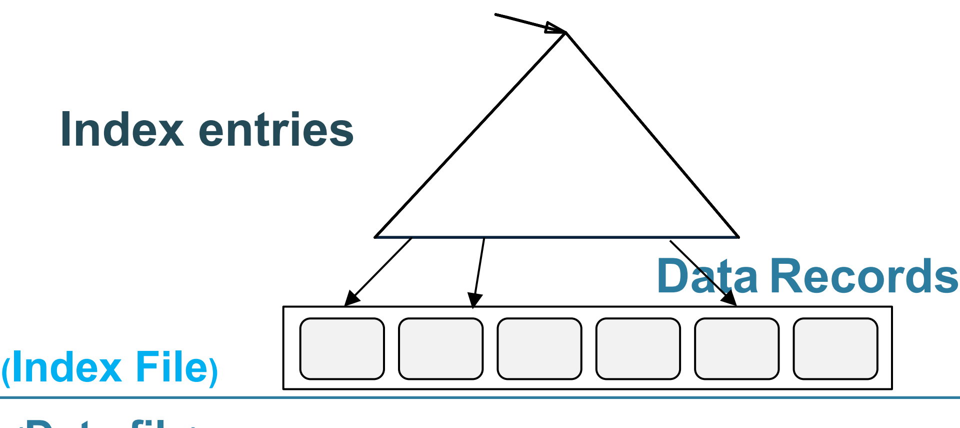
Clustered vs Unclustered



Clustered vs Unclustered

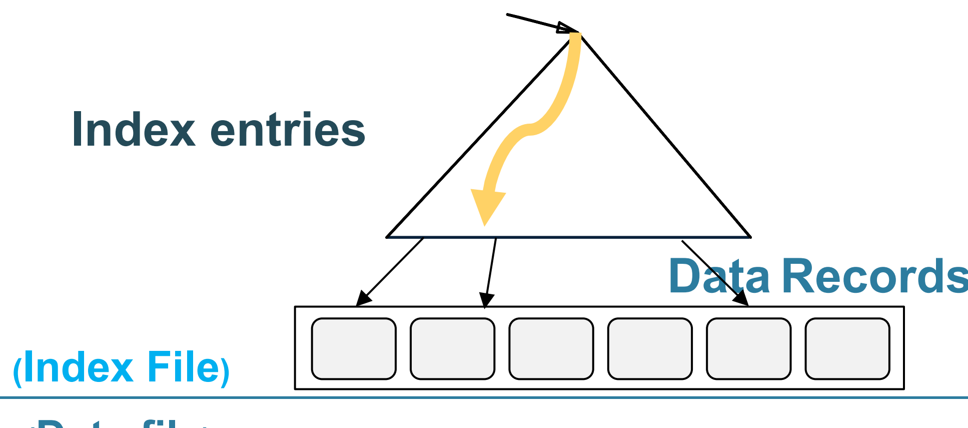


Clustered vs Unclustered



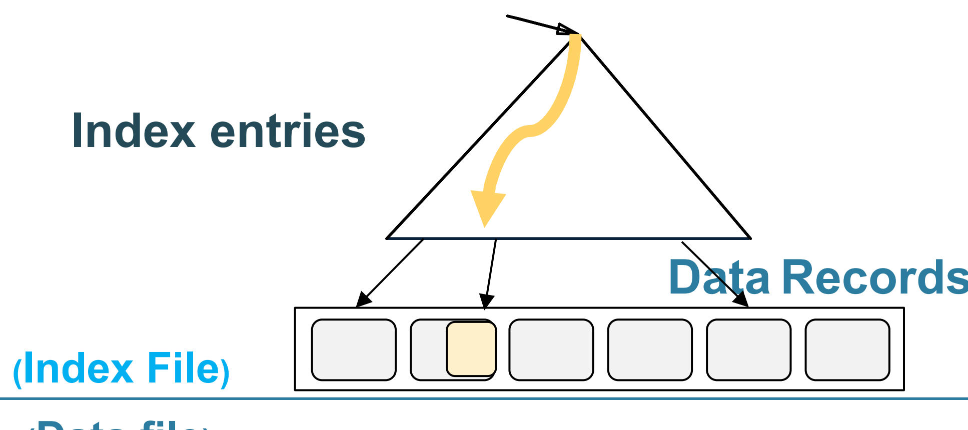
RANGE SEARCH

Clustered vs Unclustered



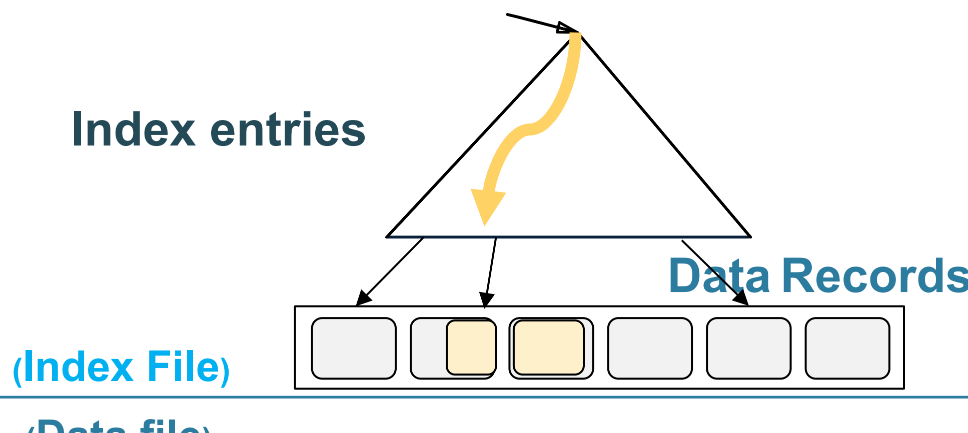
RANGE SEARCH

Clustered vs Unclustered



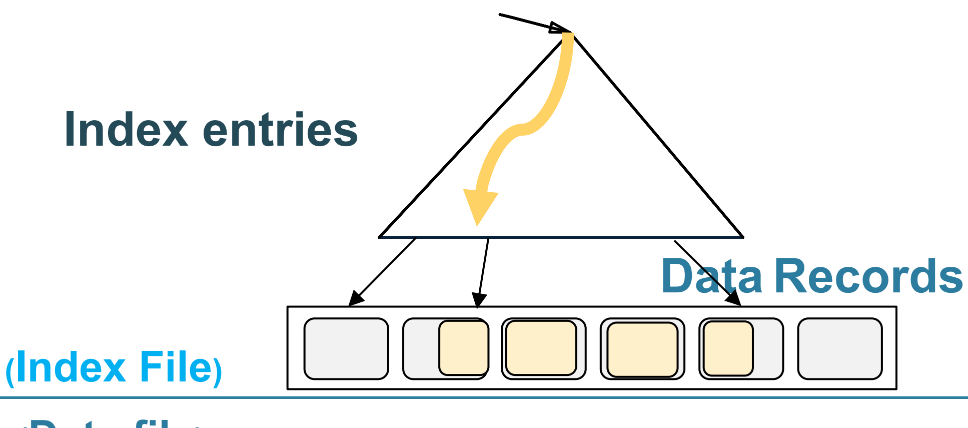
RANGE SEARCH

Clustered vs Unclustered



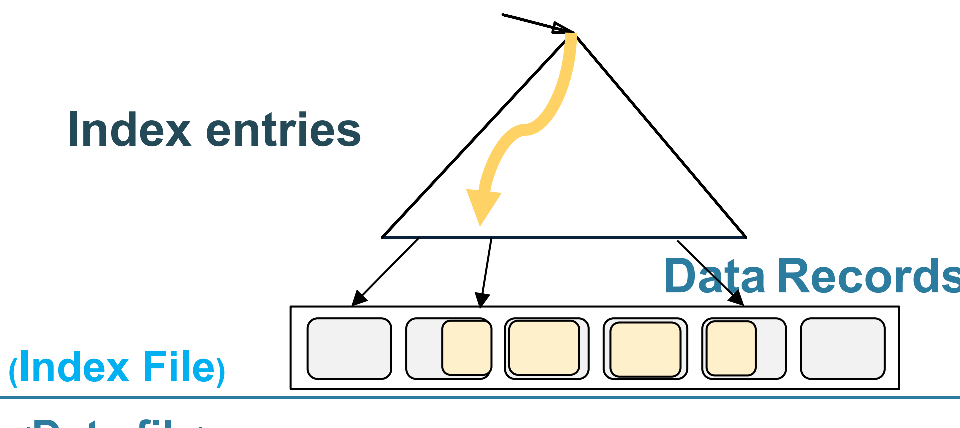
RANGE SEARCH

Clustered vs Unclustered



RANGE SEARCH

Clustered vs Unclustered



RANGE SEARCH

Sequential read

Part(pno,pname,psize,pcolor)

Clustered Index in Postgres

```
create index idxName on Part(pname);
create index idxSize on Part(psize);
create index idxColor on Part(pcolor);
```

```
cluster Part using idxName;
```

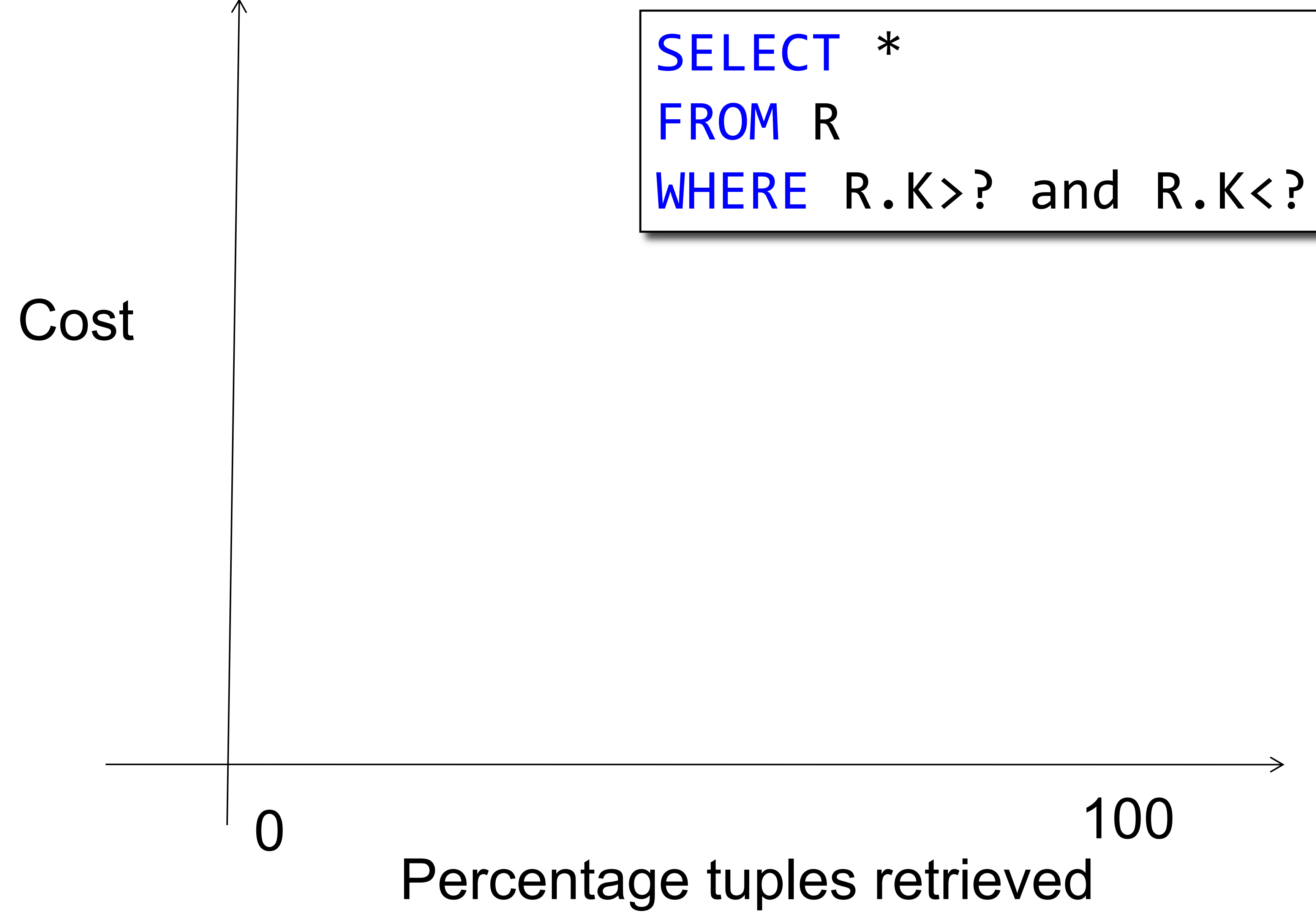
This sorts Part (takes a while) and converts it into the leaves of idxName

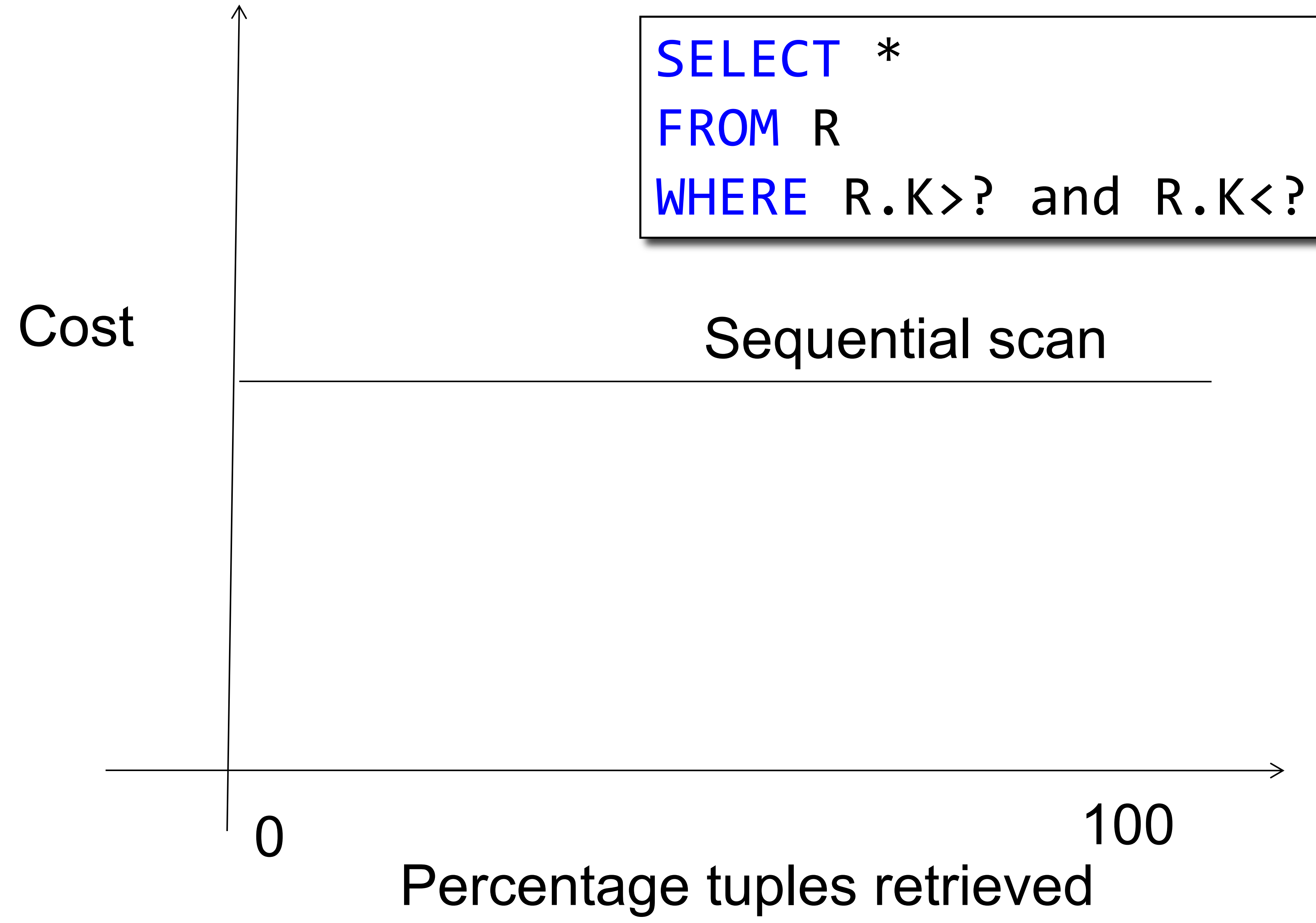
To Cluster or Not

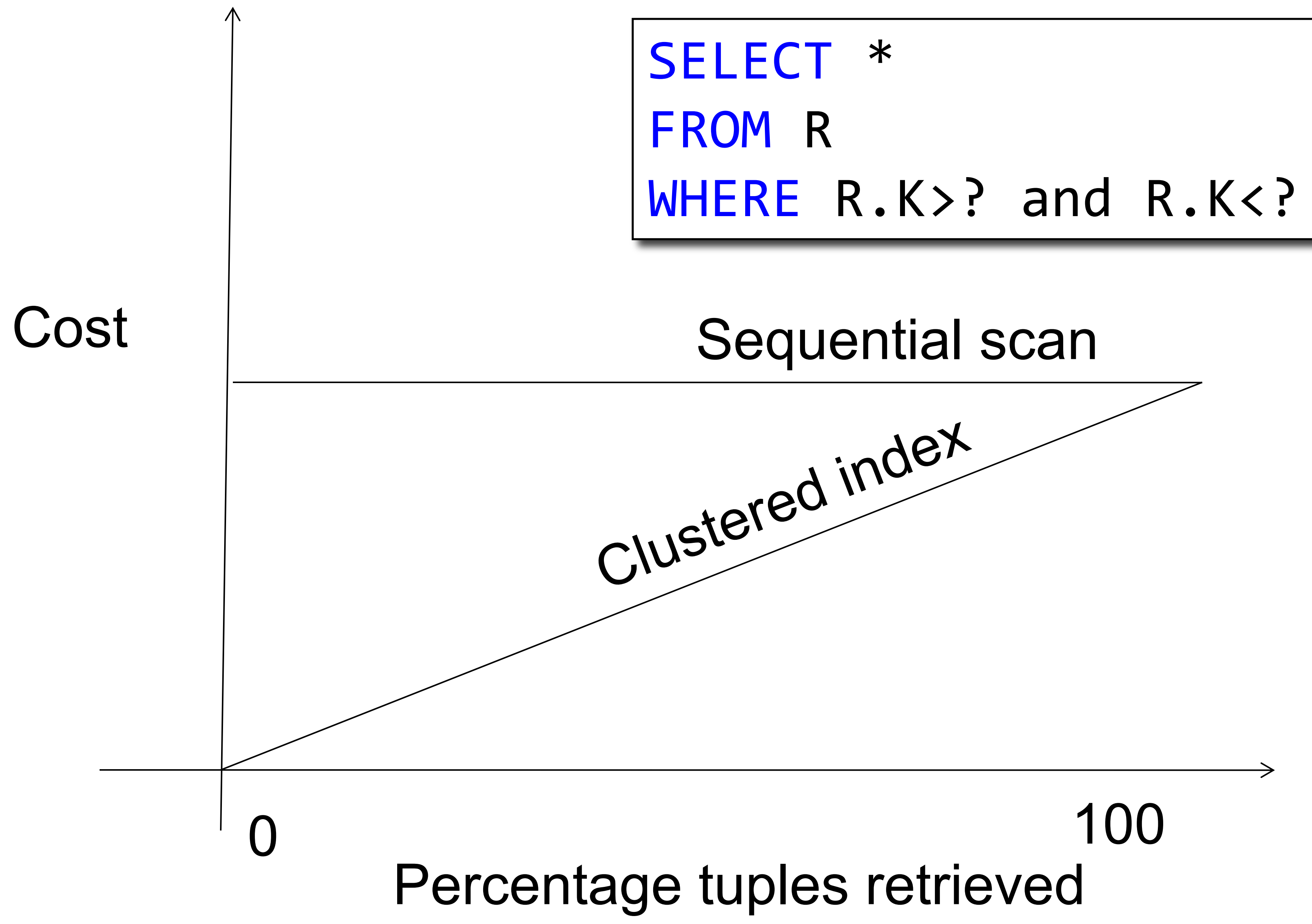
Remember:

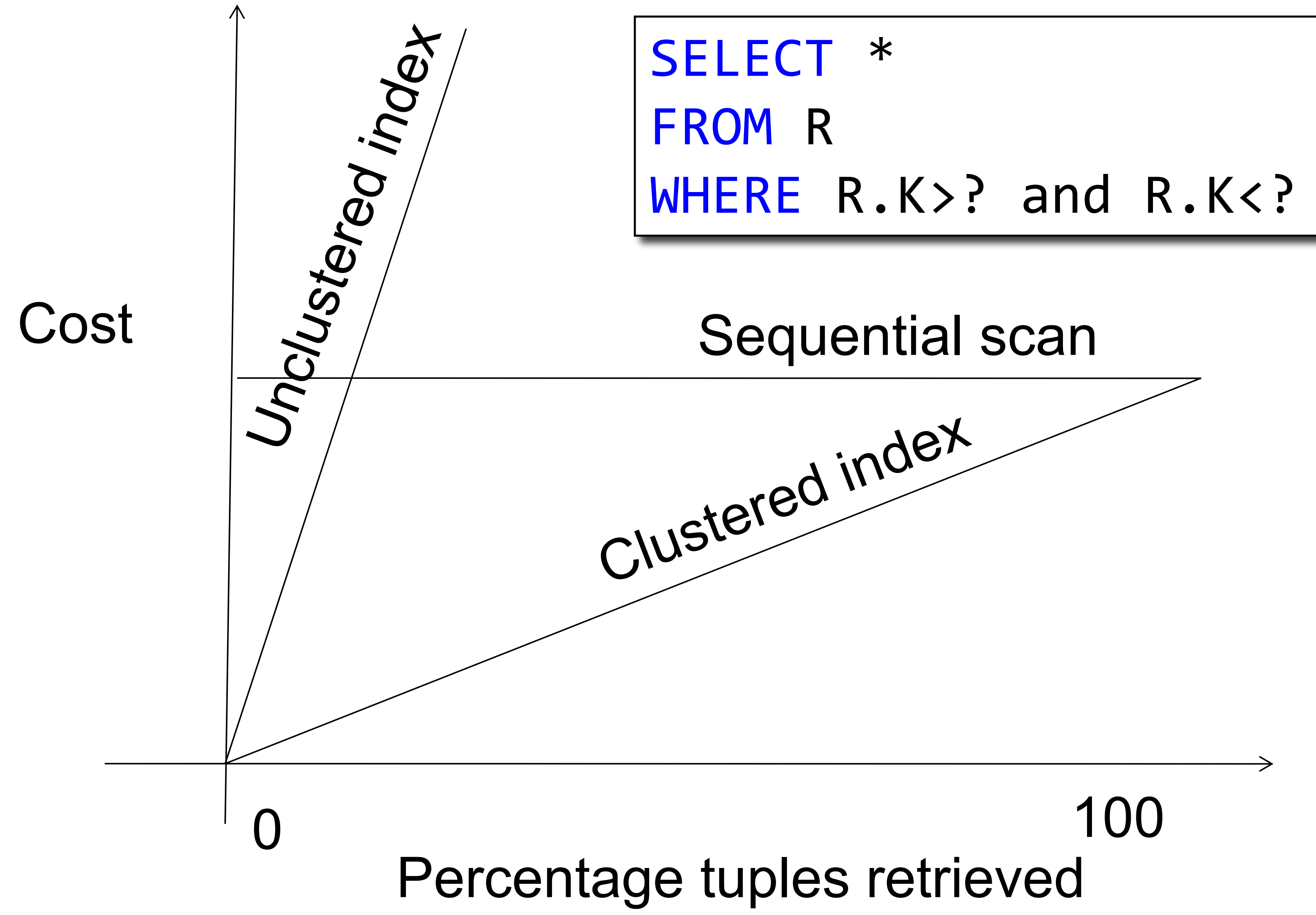
- Rule of thumb:
  Random reading 1-2% of file ≈ sequential scan entire file;

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


SELECT *
FROM R
WHERE R.K>? and R.K<?
Cost
0
100
Percentage tuples retrieved


SELECT *
FROM R
WHERE R.K>? and R.K<?
Cost
Sequential scan
0
100
Percentage tuples retrieved


SELECT *
FROM R
WHERE R.K>? and R.K<?
Cost
Sequential scan
Clustered index
0
100
Percentage tuples retrieved


SELECT *
FROM R
WHERE R.K>? and R.K<?
Unclustered index
Cost
Sequential scan
Clustered index
0
100
Percentage tuples retrieved

Final Discussion

- Indexes:
  - Speedup predicates: A=val, A<val
  - Don't speedup joins (maybe if clustered)
  - Slow down updates
- Bulk index construction:
  - More efficient than inserting one by one
  - Lesson: create the index after data import