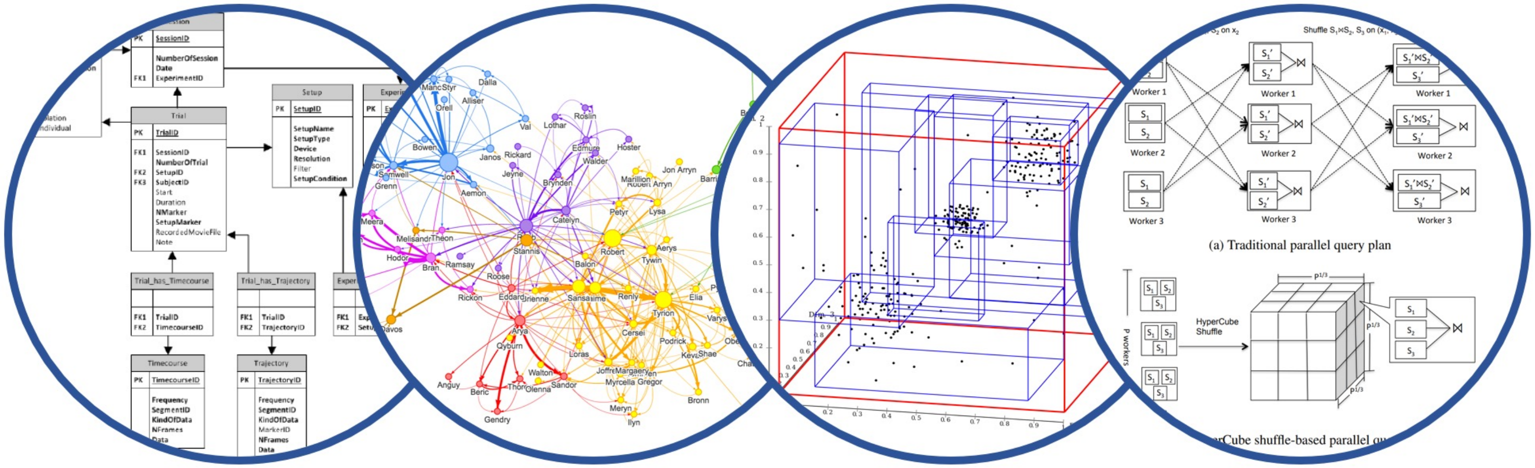


Database System Internals

Indexing

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

Announcements

- SimpleDB Partner Sign-up
  - https://forms.gle/yPac3AvyRKNF915LA
  - Last chance to submit by noon today

- Lab 1 Part 2: due next Wednesday
  - Random partner option should see a response email later today
  - Advice for working as team:
    - Make sure to schedule time to plan and/or pair code
    - Trying to strictly divide portions of the code with little communication often doesn't go well
    - Be up-front about how much time you will commit and when

Recap: Heap File

A sequence of pages (implementation in SimpleDB)

Data page	Data page	Data page	Data page	Data page	Data page	Data page	Data page

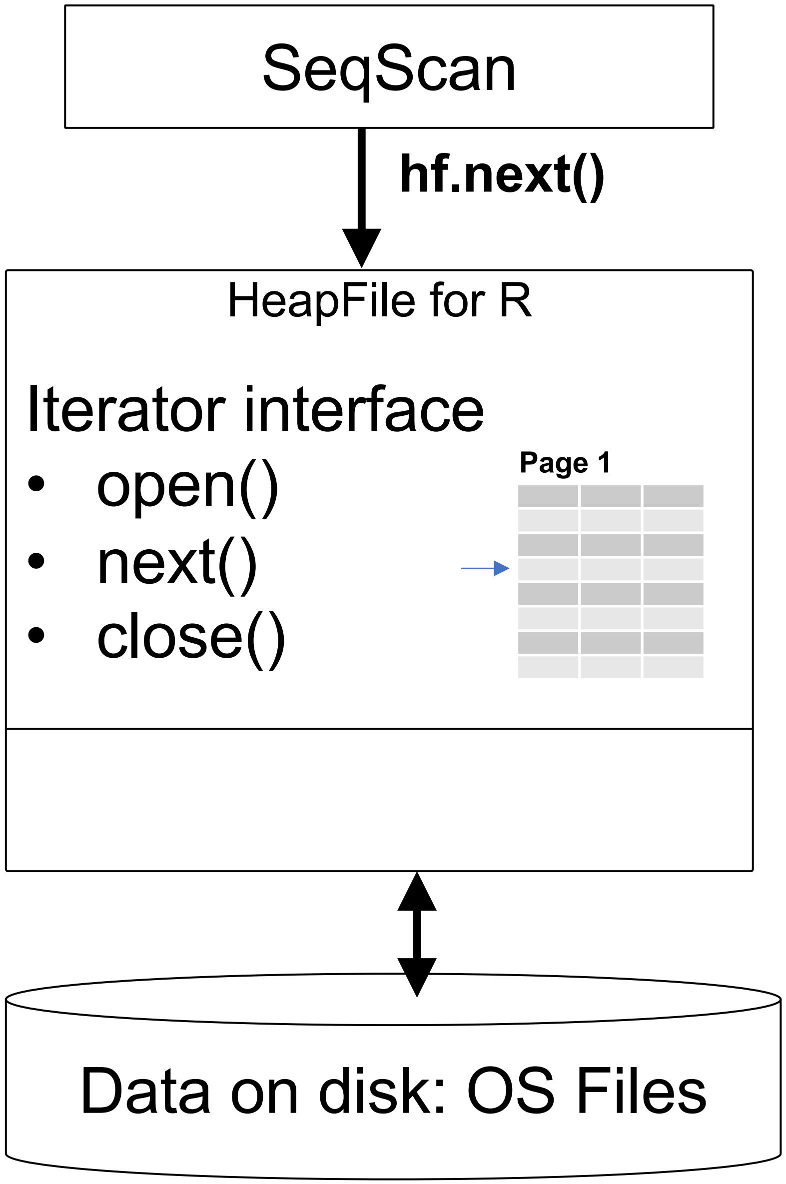
Some pages have space and other pages are full

Add pages at the end when need more space

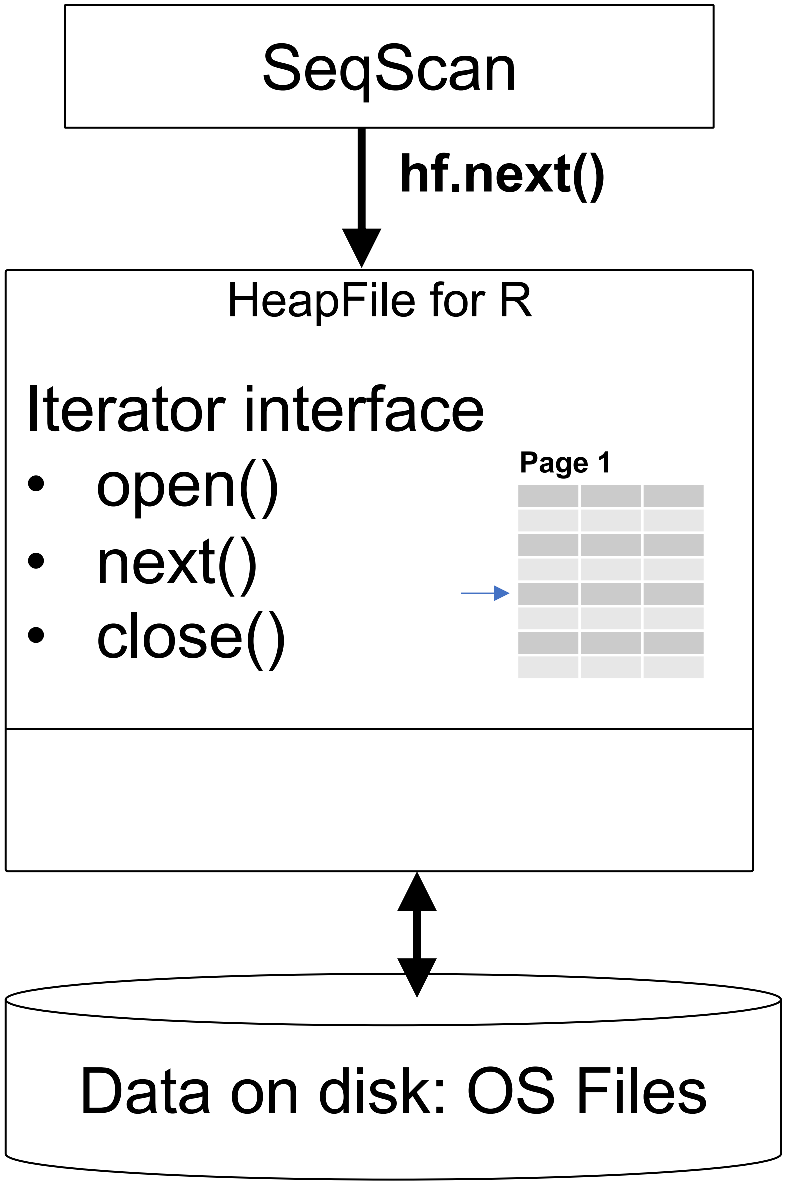
Works well for small files

But finding free space requires scanning the file…

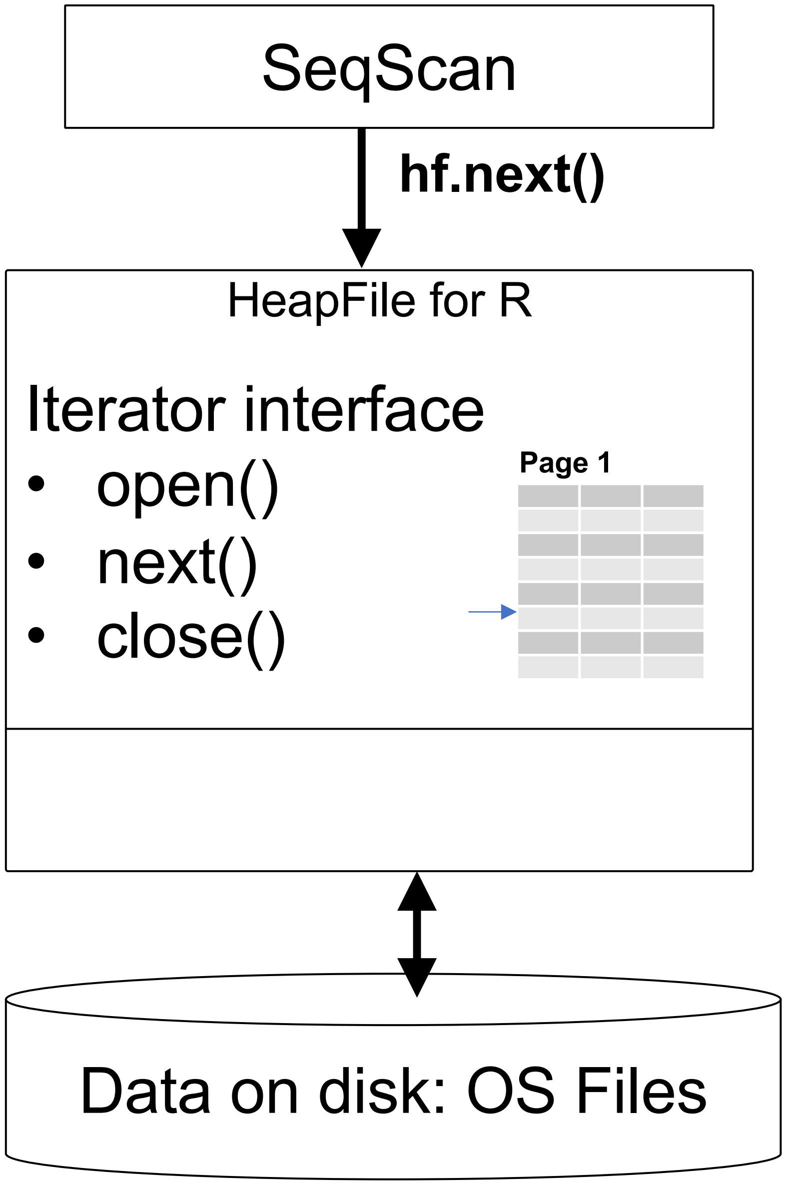
Query Execution In SimpleDB



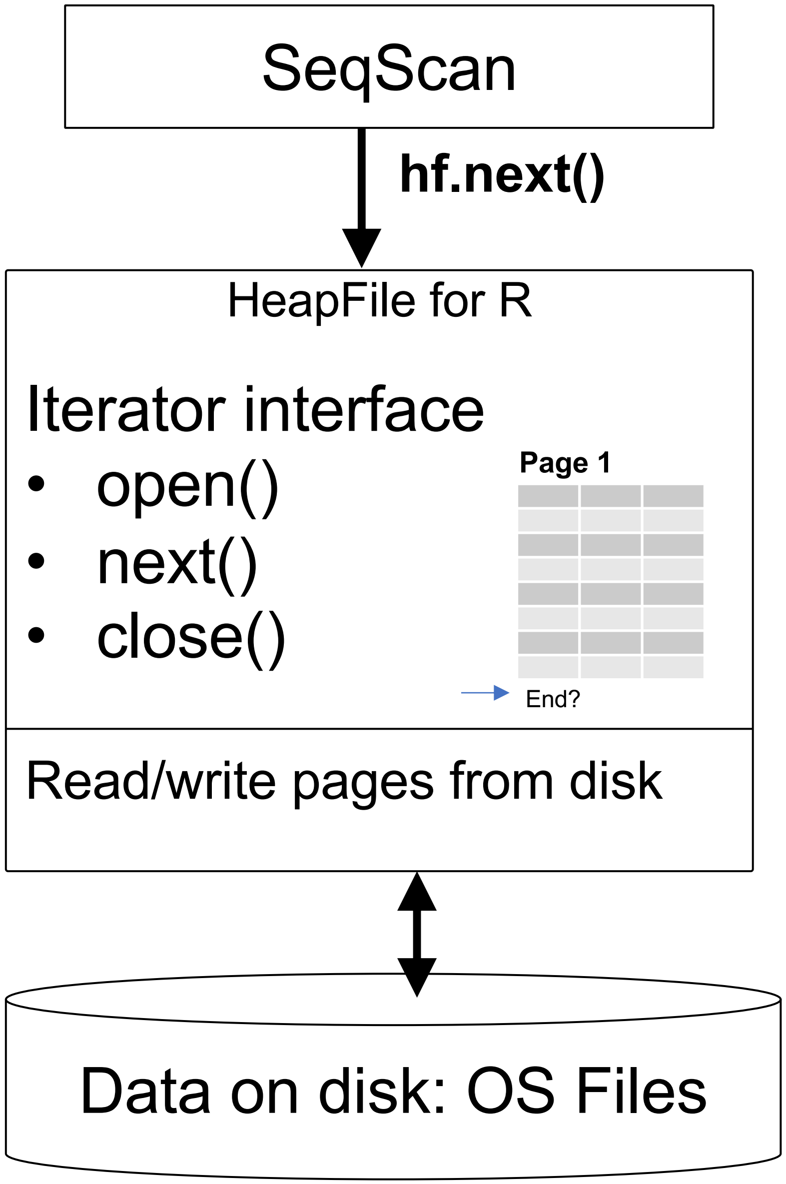
Query Execution In SimpleDB



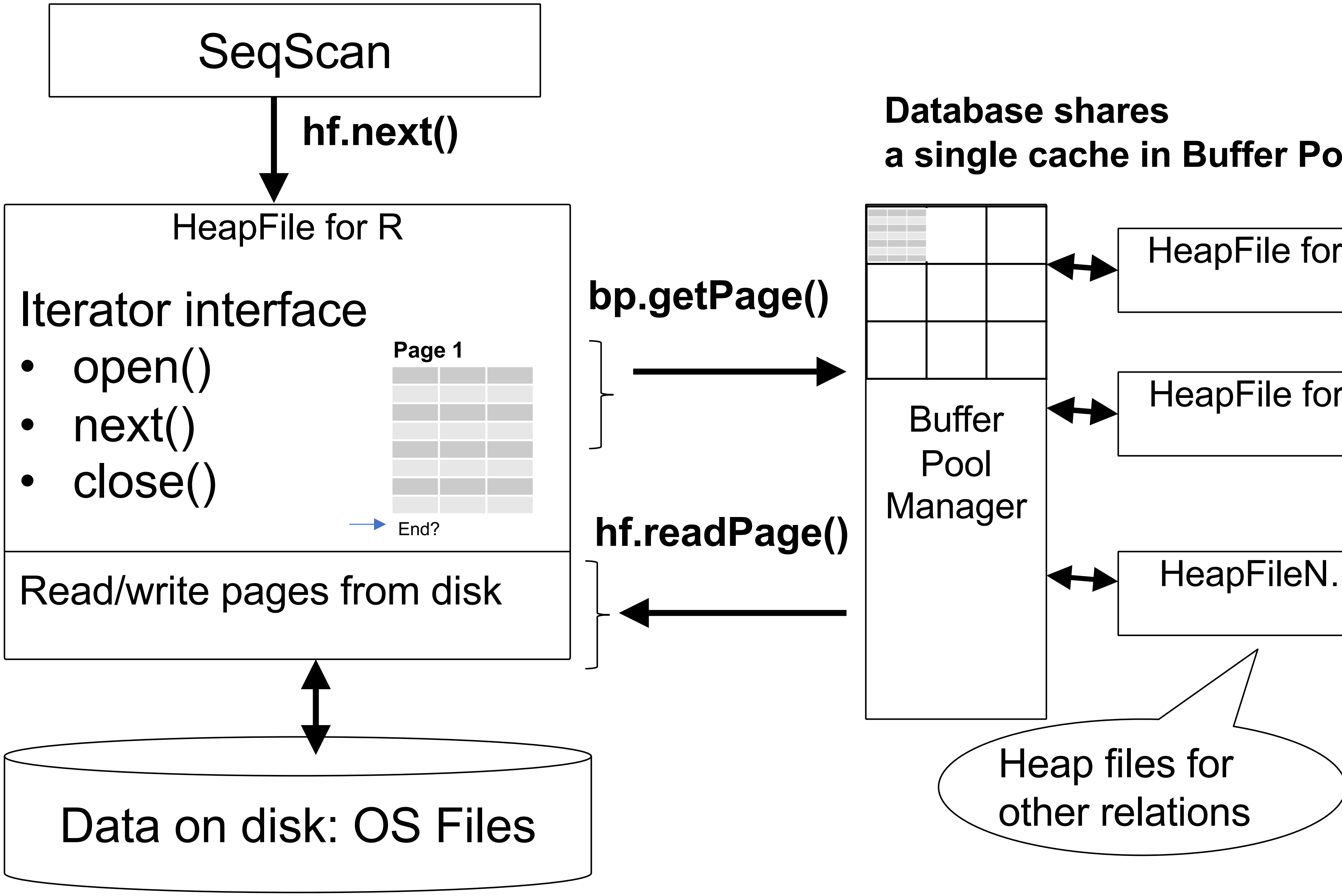
Query Execution In SimpleDB



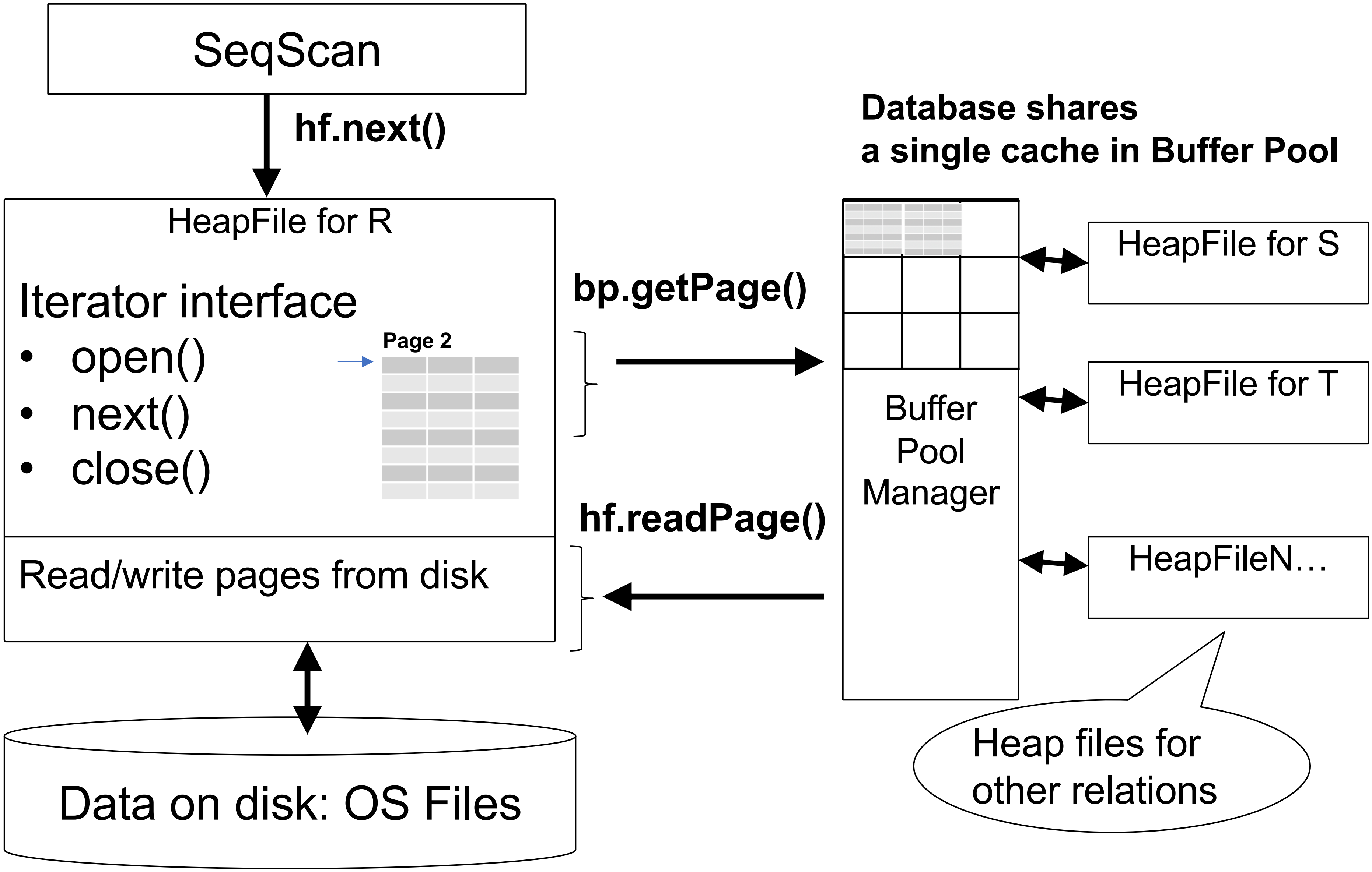
Query Execution In SimpleDB



Query Execution In SimpleDB



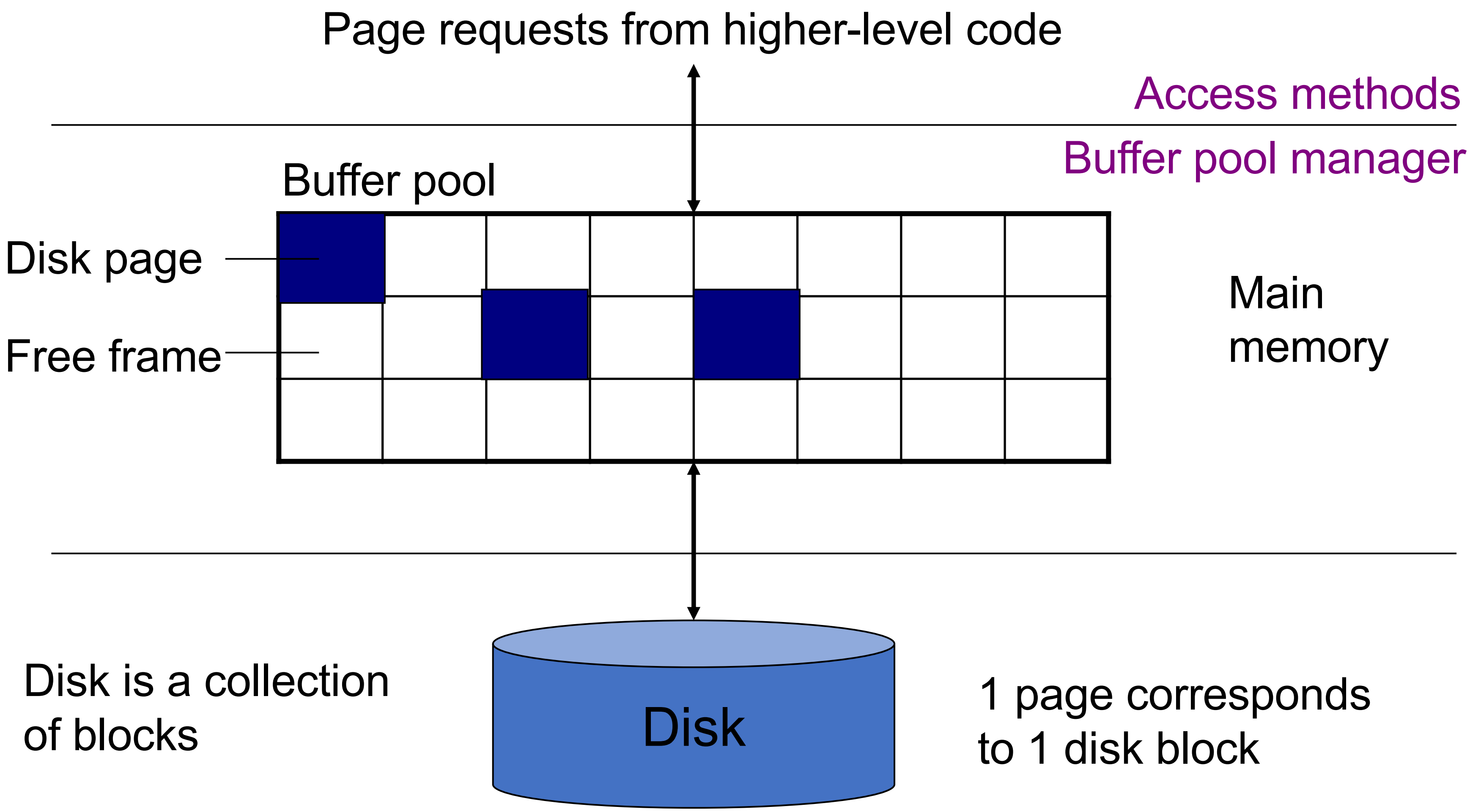
Query Execution In SimpleDB



Buffer Manager

- Brings pages in from memory and caches them
- Eviction policies
  - Random page (ok for SimpleDB)
  - Least-recently used (LRU)
  - The "clock" algorithm
- Keeps track of which **pages are dirty**
  - A dirty page has changes not reflected on disk
  - Implementation: Each page includes a dirty bit

Buffer Manager



Pushing Updates to Disk

- When inserting a tuple, HeapFile inserts it on a page but does not write the page to disk
- When deleting a tuple, HeapFile deletes tuple from a page but does not write the page to disk
- The buffer manager worries when to write pages to disk (and when to read them from disk)
- When need to add new page to file, HeapFile adds page to file on disk and then reads it through buffer manager

Basic Access Method: Heap File

API

- **Create** or **destroy** a file
- **Insert** a record
- **Delete** a record with a given rid (rid)
  - rid: unique tuple identifier
- **Get** a record with a given rid
  - Not necessary for sequential scan operator
  - But used with indexes
- **Scan** all records in the file

Basic Access Method: Heap File

API

- **Create** or **destroy** a file
- **Insert** a record
- **Delete** a record with a given rid (rid)
  - rid: unique tuple identifier (more later)
- **Get** a record with a given rid
  - Not necessary for sequential scan operator
  - But used with indexes
- **Scan** all records in the file

Next: how to scan only *some* records

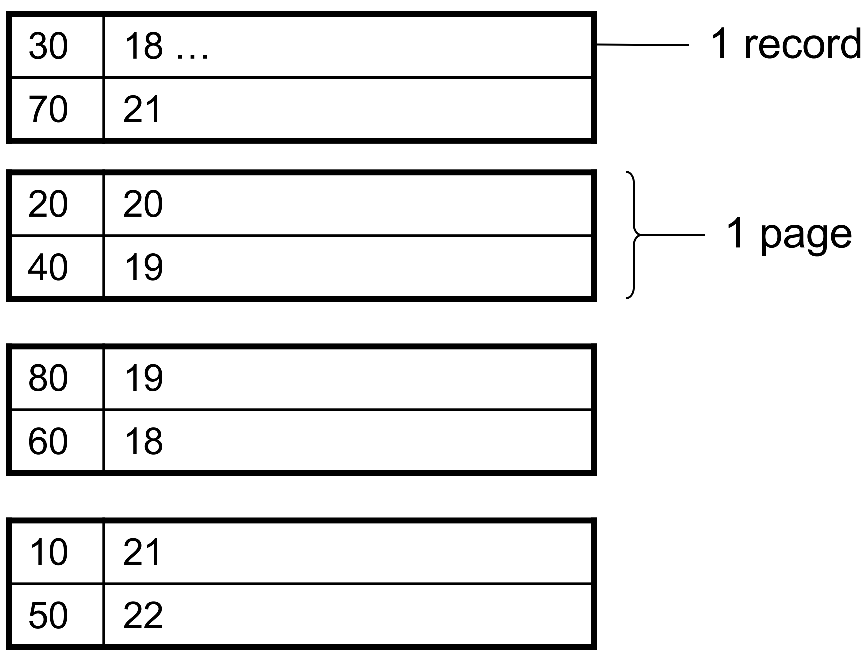
Access by Attribute Value

- Scan all Suppliers where city='Seattle'
- Scan all Students with GPA > 3.5
- Scan all Students with SID = 12345 // just one

Searching in a Heap File

File is not sorted on any attribute

`Student(sid: int, age: int, …)`



Heap File Search Example

- 10,000 students
- 10 student records per page
- Total number of pages: 1,000 pages

- Find student where sid =12345

- Find all students where age > 20

- Can we do better?

Heap File Search Example

- 10,000 students
- 10 student records per page
- Total number of pages: 1,000 pages

- Find student where sid =12345
  - Must read on average 500 pages

- Find all students where age > 20
  - Must read all 1,000 pages

- Can we do better?

Sorted File (a.k.a. Sequential File)

File sorted on an attribute, usually on primary key

`Student(sid: int, age: int, …)`

10	21 …
20	20

30	18
40	19

50	22
60	18

70	21
80	19

Sequential File Example

- Total number of pages: 1,000 pages

- Find student where sid=12345
  - How many pages do we need to read?

Sequential File Example

- Total number of pages: 1,000 pages

- Find student where sid=12345
  - How many pages do we need to read?
  - Binary search: read $\log_2(1{,}000) \approx 10$ pages

Limitations of Sorted Files

We want to support these kinds of queries:

- Find student where sid=12345
- Find students where age > 20
- Insert a new student

What are the limitations of using a sorted file?

Creating Indexes in SQL

```
CREATE  TABLE  Student(sid int, age int, gpa real, …);
```

```
select *
from Student
where sid=12345
```

Creating Indexes in SQL

```
CREATE  TABLE  Student(sid int, age int, gpa real, ...);
```

```
CREATE  INDEX s_sid ON Student(sid)
```

```
select *
from Student
where sid=12345
```

Creating Indexes in SQL

```
CREATE  TABLE  Student(sid int, age int, gpa real, ...);
```

```
CREATE  INDEX s_sid ON Student(sid)
```

```
select *
from Student
where sid=12345
```

```
CREATE  INDEX s_age ON Student(age)
```

Creating Indexes in SQL

```
CREATE  TABLE  Student(sid int, age int, gpa real, ...);
```

```
CREATE  INDEX s_sid ON Student(sid)
```

```
select *
from Student
where sid=12345
```

```
CREATE  INDEX s_age ON Student(age)
```

```
select *
from Student
where age > 25
```

Outline

- Index structures
- Hash-based indexes

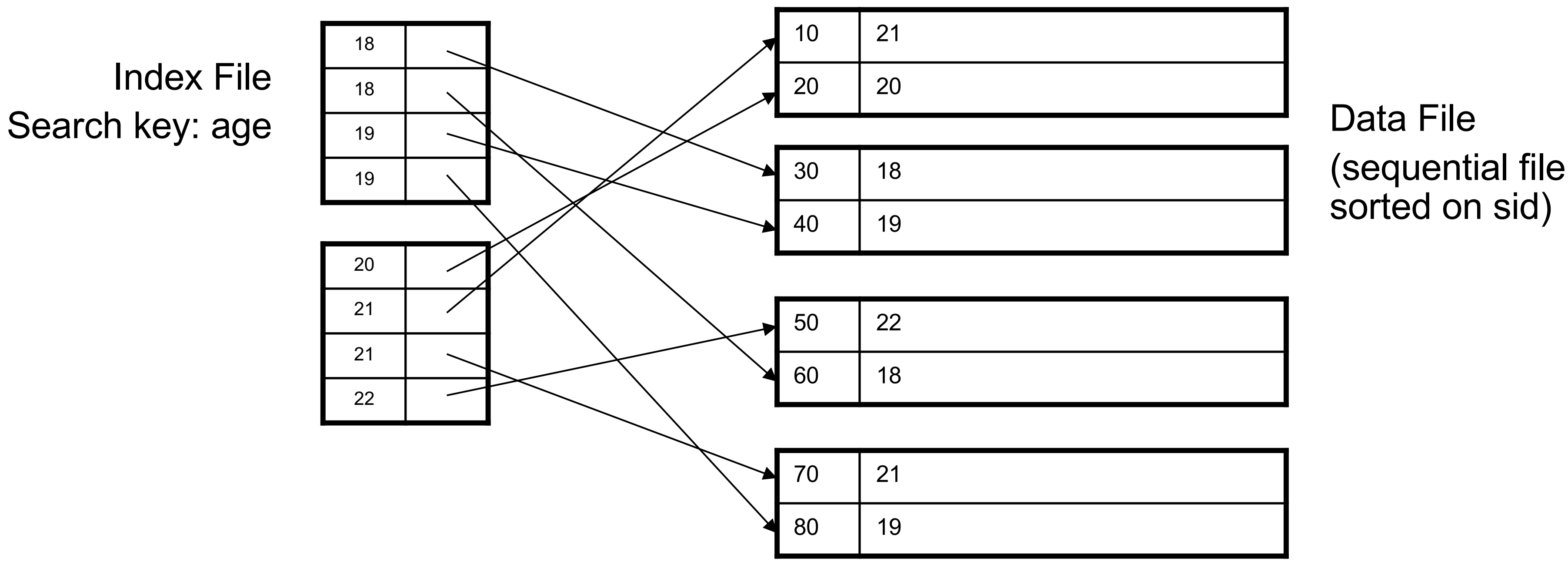
Today

- B+ trees

Next time

Indexes

- **Index:** separate file with fast access by "key" value
- Contains pairs of the form (key, RID)
- **Indexes are access methods!** Same API as Heap Files



Indexes

- **Search key** = can attribute or set of attributes
  - not the same as the primary key; not a key

- **Index** = collection of data entries

- **Data entry** for key k can be:
  - (k, RID)
  - (k, list-of-RIDs)
  - Record with key k; "clustered" or "primary" index

Different Types of Files

Imagine one relation, say Student

- The Student file can be:
  - Heap file (tuples stored without any order)
  - Sequential file (tuples sorted on some attribute(s))
  - Clustered (primary) index file (relation+index)

- There can be several unclustered (secondary) index files that store (key,rid) pairs

How Indexes Help

We want to support these kinds of queries
Assume Student = a heap file

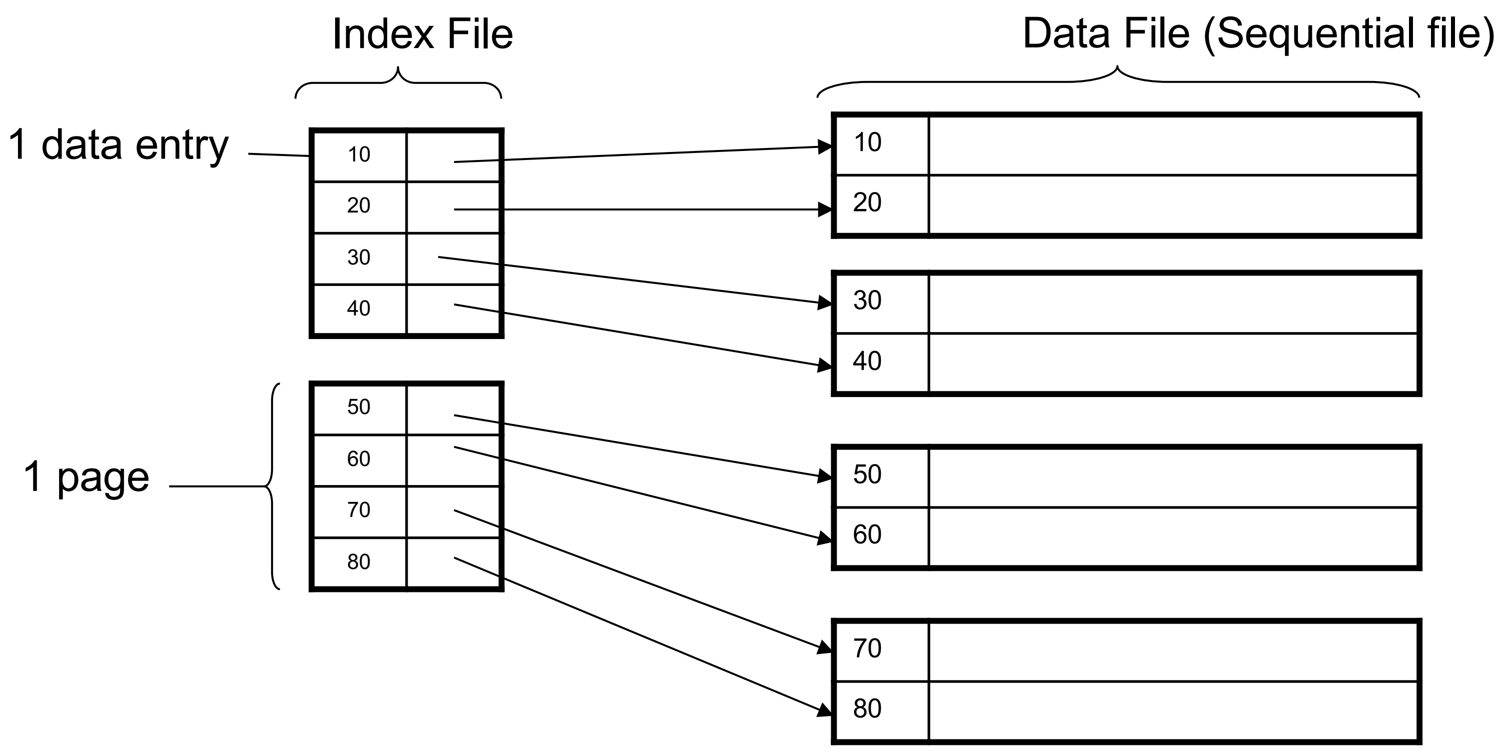
- Find student where sid=12345
  - Use an index on Student(sid)

- Find students where age > 20
  - Use an index on Student(age)

- Insert a new student
  - Insert in the Student heap file – easy
  - Insert in indexes Student(sid), Student(age) – will discuss

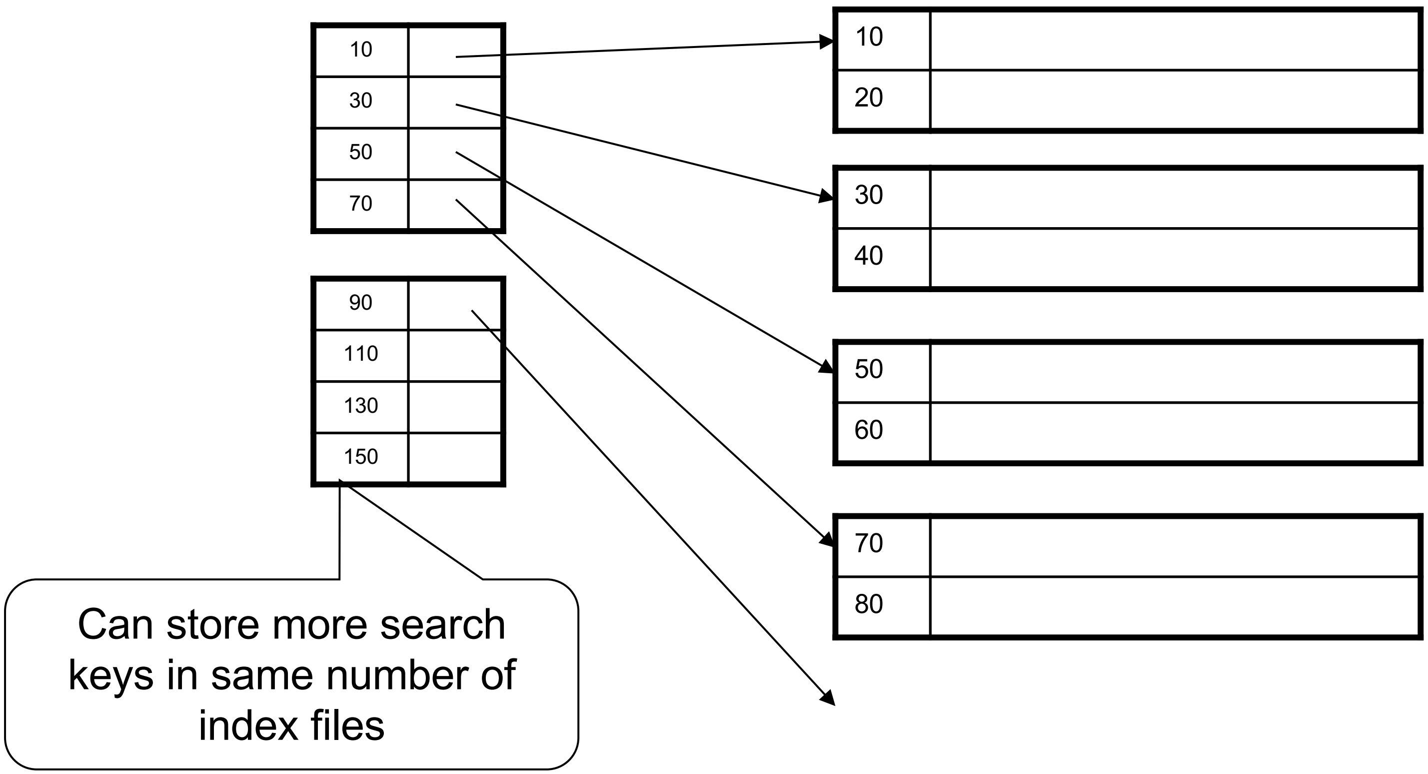
Clustered Index (aka Primary Index)

- Records in data file have same order as in index
- ***Dense*** index: sequence of (key,rid) pairs



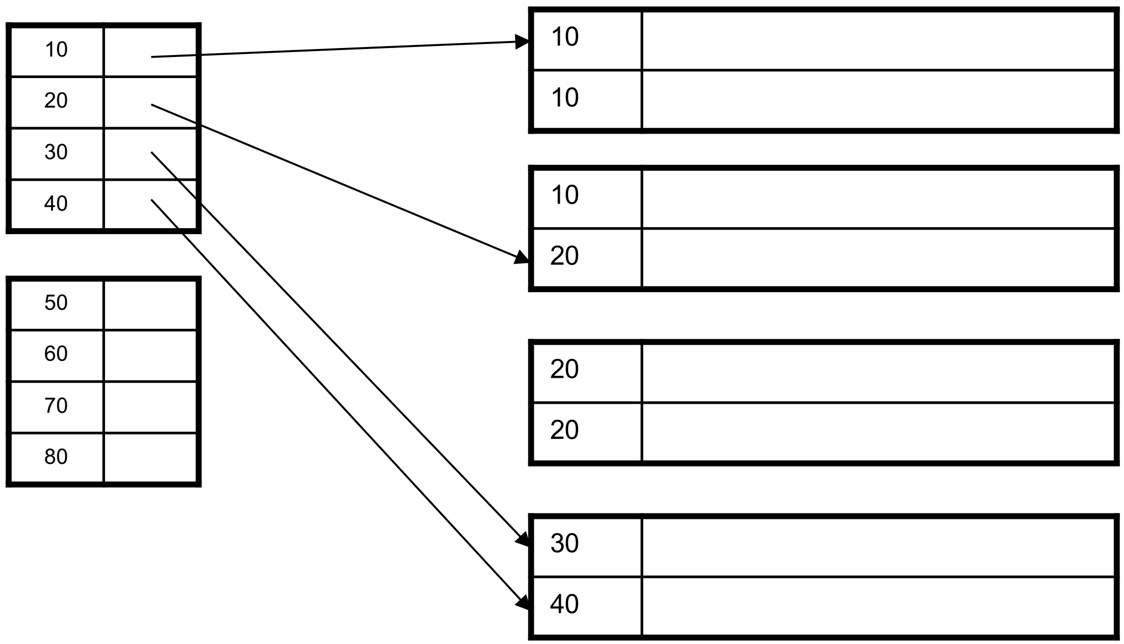
Clustered Index (aka Primary Index)

- Records in data file have same order as in index
- *Sparse* index: store a subset of (key,rid) pairs



Clustered Index with Duplicate Keys

- Dense index:

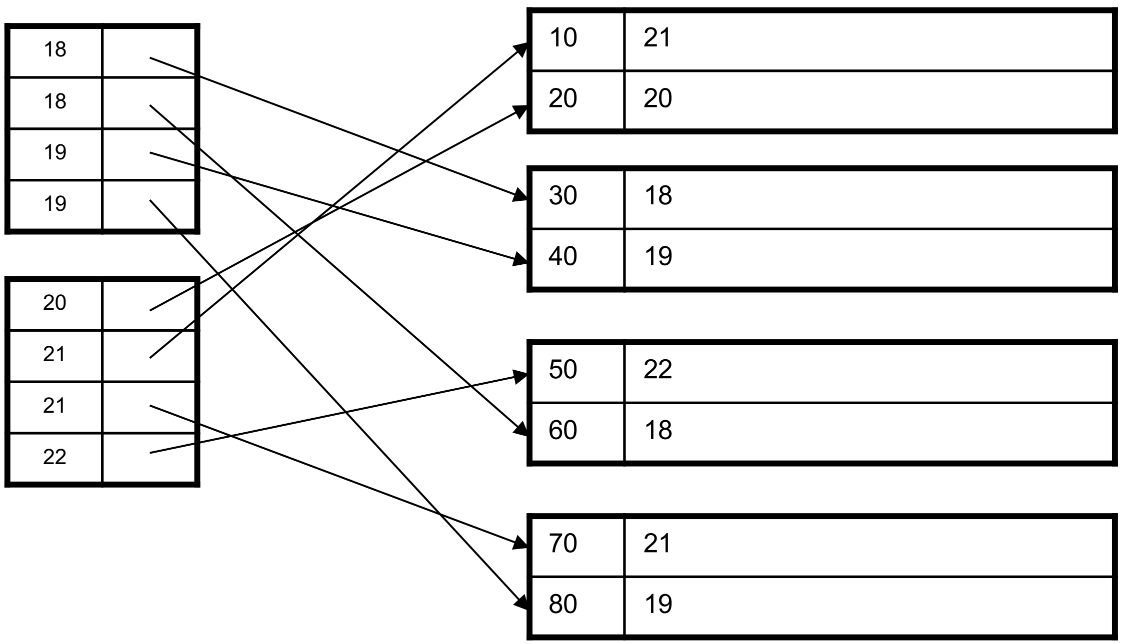


Clustered Index: Back to Example

- Assume entire index fits in main memory
- Find student where sid=12345
  - Index (dense or sparse) points directly to the page
  - Read only 1 page from disk
- Find all students where age > 20
  - Add a second index…

Secondary Indexes

- Do not determine placement of records in data files
- Always dense (why ?)



The Confusing Terminology of Indexes…

- Clustered index:
  - Means: keys close in the index are also close in the data
  - Can co-exist with the data file (quite common)
  - Can have only one clustered index
  - Sometimes called "primary index"

The Confusing Terminology of Indexes...

- Clustered index:
  - Means: keys close in the index are also close in the data
  - Can co-exist with the data file (quite common)
  - Can have only one clustered index
  - Sometimes called "primary index"
- Unclustered index:
  - Means: order in the index and order in the data differ
  - Always a separate file
  - Can have as many unclustered indexes as we want
  - Sometimes called "secondary index"

Index Organization

- The index is a collection of (key, RID(s)) pairs

- Needs to support efficiently:
  - Find the entry where key=[some value]
  - Insert a new (key, RID)
  - Delete a (key, RID)

- How would you design the index data structure?

Index Organization

- The index is a collection of (key, RID(s)) pairs
- Needs to support efficiently:
  - Find the entry where key=[some value]
  - Insert a new (key, RID)
  - Delete a (key, RID)
- How would you design the index data structure?
  - Ordered file – problem here (why?)

Index Organization

- The index is a collection of (key, RID(s)) pairs

- Needs to support efficiently:
  - Find the entry where key=[some value]
  - Insert a new (key, RID)
  - Delete a (key, RID)

- How would you design the index data structure?
  - Ordered file – problem here (why?)
  - Hash table

Index Organization

- The index is a collection of (key, RID(s)) pairs
- Needs to support efficiently:
  - Find the entry where key=[some value]
  - Insert a new (key, RID)
  - Delete a (key, RID)
- How would you design the index data structure?
  - Ordered file – problem here (why?)
  - Hash table
  - B+ tree

BRIEF Review of Hash Tables

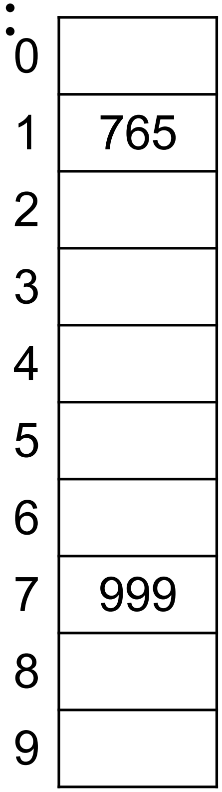
Arrays are _very_ efficient:
- Find(T[7])

0	
1	765
2	
3	
4	
5	
6	
7	999
8	
9	

BRIEF Review of Hash Tables

Arrays are *very* efficient:

- Find(T[7])
- Set T[3] := 234



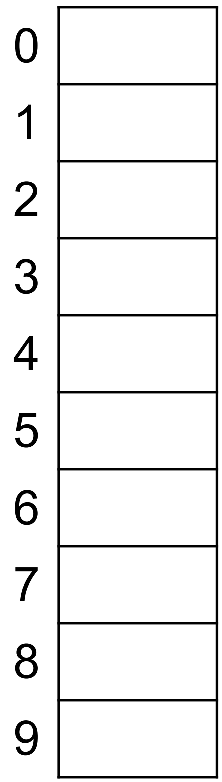
BRIEF Review of Hash Tables

Arrays are *very* efficient:
- Find(T[7])
- Set T[3] := 234

0	
1	765
2	
3	234
4	
5	
6	
7	999
8	
9	

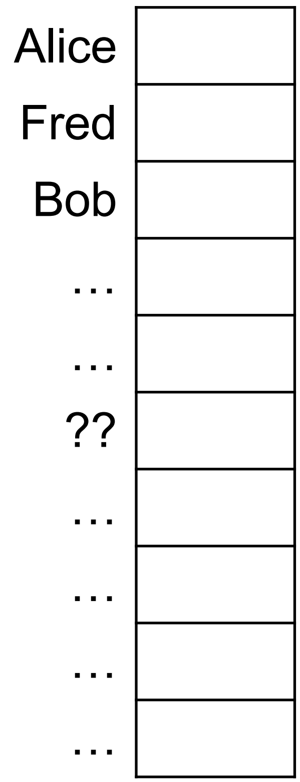
BRIEF Review of Hash Tables

Problem: the key is not 0,1,2,…9 but is a string k



BRIEF Review of Hash Tables

Problem: the key is not 0,1,2,…9 but is a string k

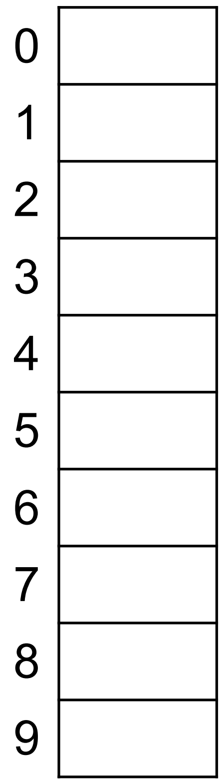


BRIEF Review of Hash Tables

Problem: the key is not 0,1,2,…9 but is a string k

A (naïve) hash function:

$$h(k) = sum(k) \bmod 10$$

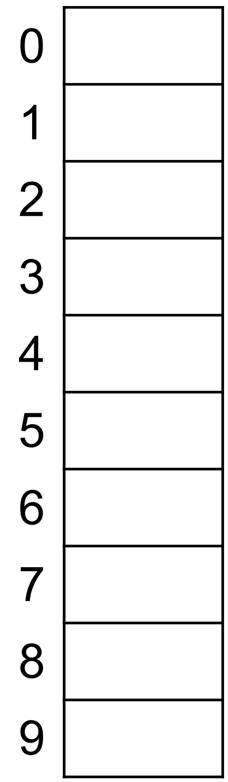


BRIEF Review of Hash Tables

Problem: the key is not 0,1,2,…9 but is a string k

A (naïve) hash function:

$$h(k) = sum(k) \bmod 10$$



Example: h("Fred") =
= (ascii("F")+ascii("r")+…) mod 10
= (70 + 114 + 101 + 100) mod 10
= 5

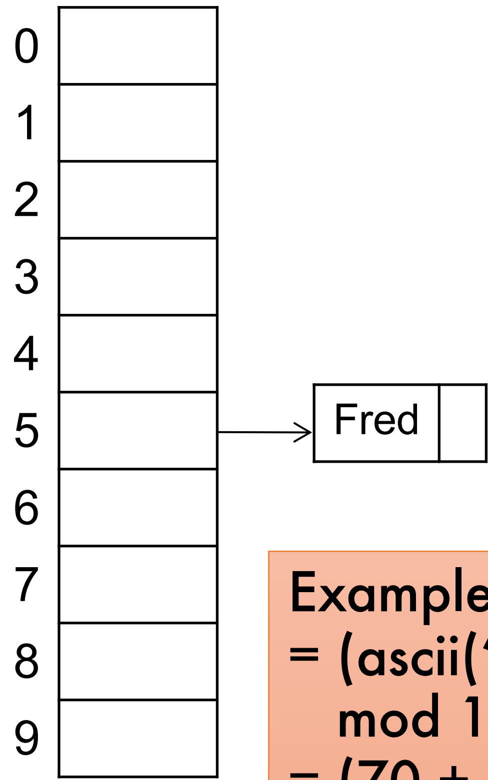
BRIEF Review of Hash Tables

Problem: the key is not 0,1,2,…9 but is a string k

A (naïve) hash function:

$$h(k) = sum(k) \bmod 10$$

Separate chaining:



Example: h(“Fred”) =
= (ascii(“F”)+ascii(“r”)+…) mod 10
= (70 + 114 + 101 + 100) mod 10
= 5

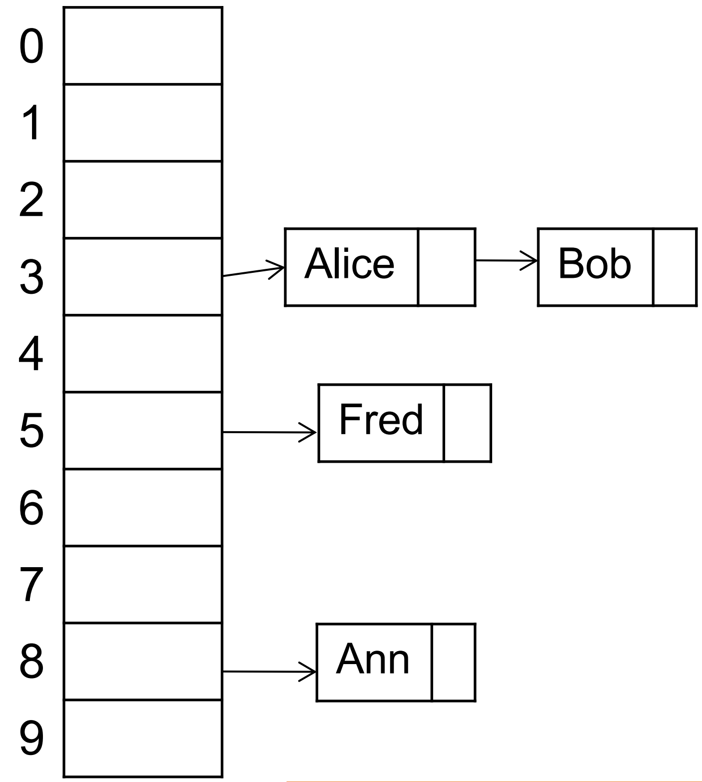
BRIEF Review of Hash Tables

Problem: the key is not 0,1,2,…9 but is a string k

A (naïve) hash function:

h(k) = sum(k) mod 10

Separate chaining:



h("Alice") = h("Bob") = 3
Called *collisions*

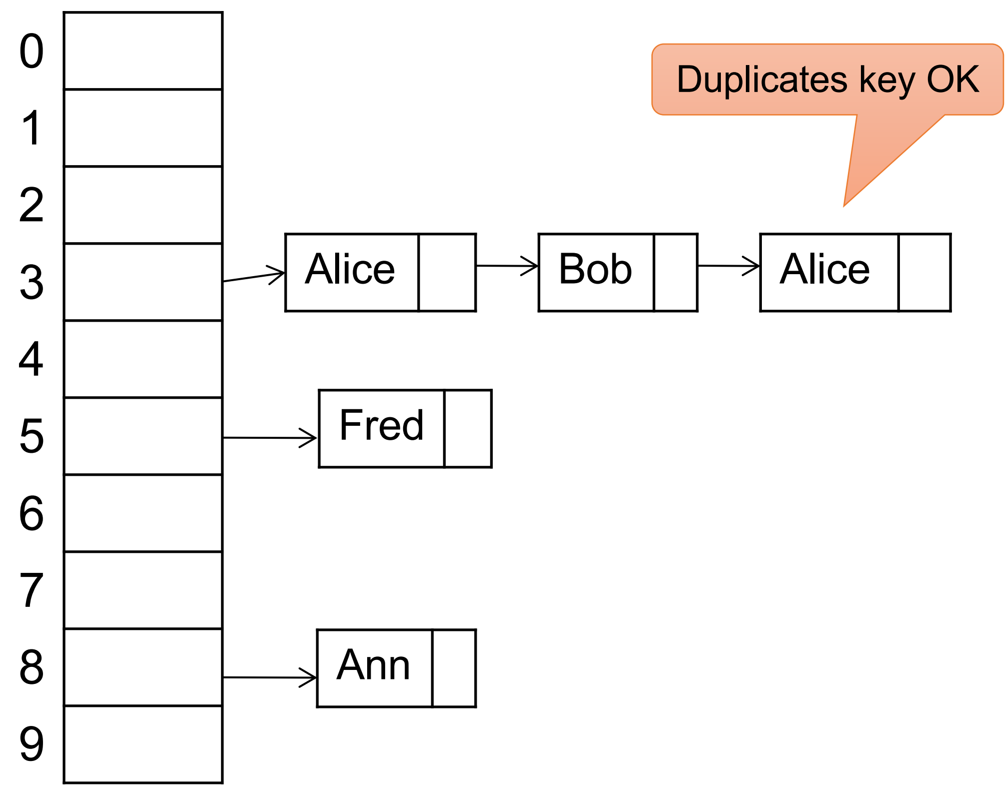
BRIEF Review of Hash Tables

Problem: the key is not 0,1,2,…9 but is a string k

A (naïve) hash function:

h(k) = sum(k) mod 10

Separate chaining:



BRIEF Review of Hash Tables

Problem: the key is not 0,1,2,…9 but is a string k

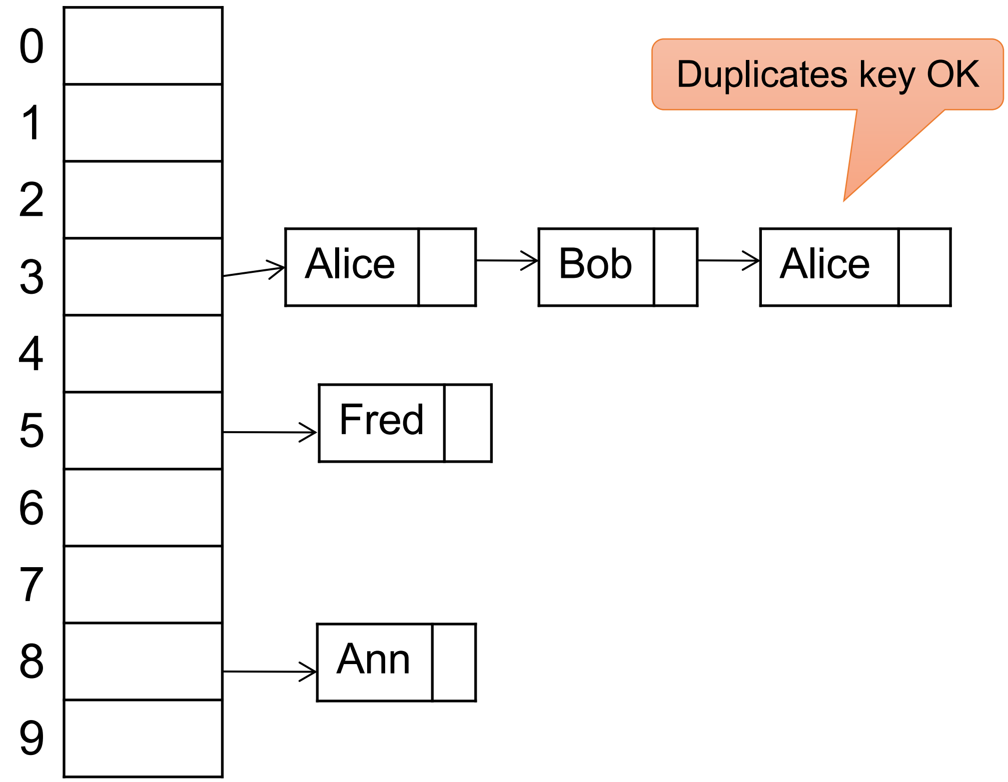
A (naïve) hash function:

h(k) = sum(k) mod 10

Operations:

find(Bob) = ??

Separate chaining:



BRIEF Review of Hash Tables

Problem: the key is not 0,1,2,…9 but is a string k

A (naïve) hash function:

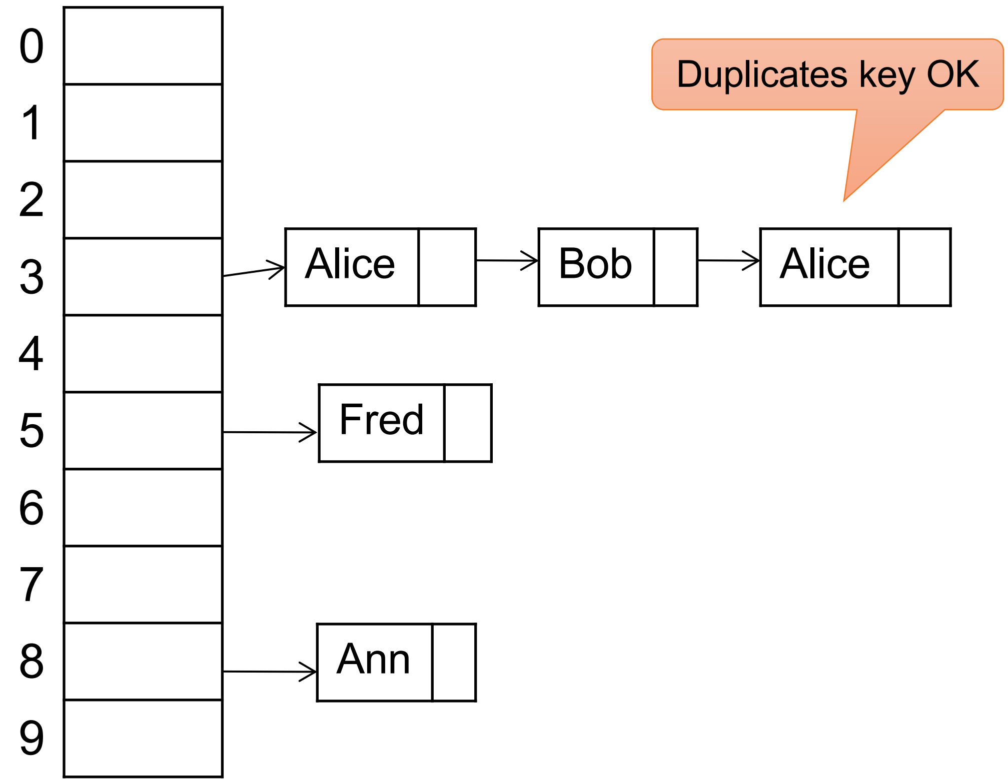
h(k) = sum(k) mod 10

Operations:

find(Bob) = ??

insert(Jon) = ??

Separate chaining:



BRIEF Review of Hash Tables

Problem: the key is not 0,1,2,…9 but is a string k

A (naïve) hash function:

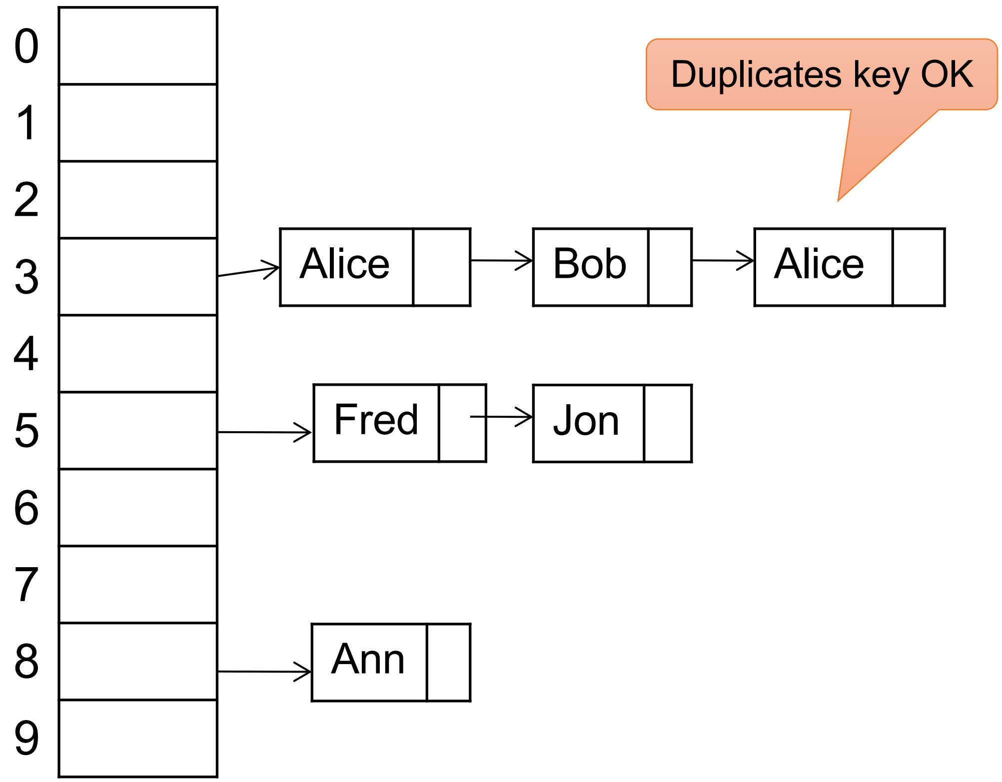
h(k) = sum(k) mod 10

Operations:

find(Bob) = ??

insert(Jon) = ??

Separate chaining:



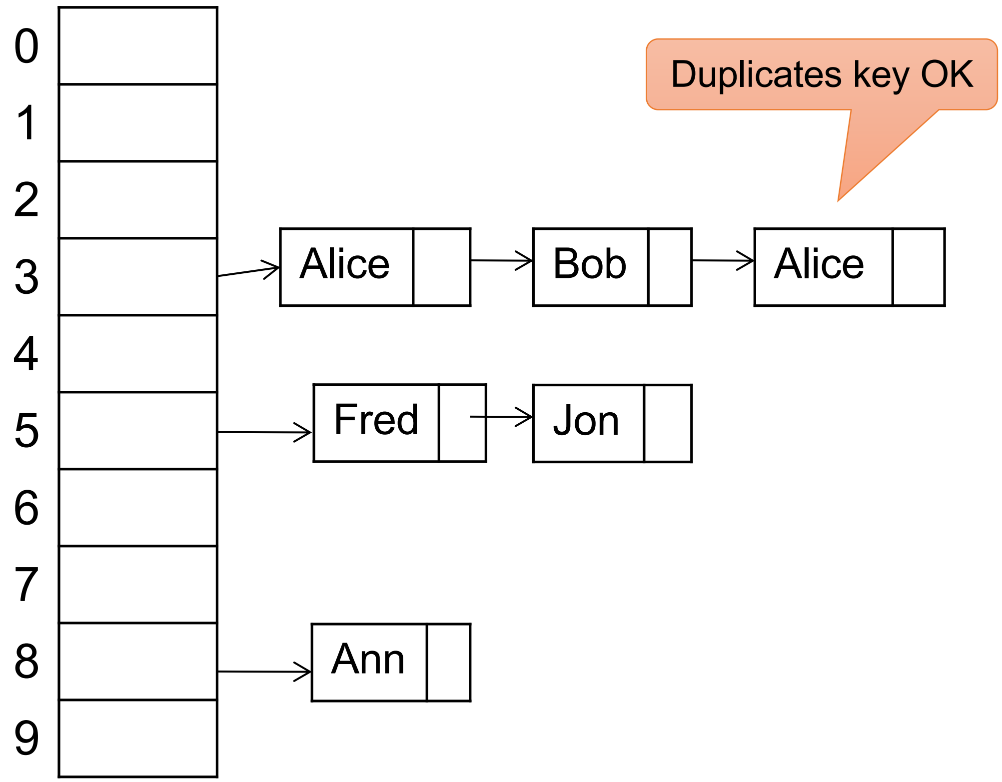
BRIEF Review of Hash Tables

Problem: the key is not 0,1,2,…9 but is a string k

A (naïve) hash function:

h(k) = sum(k) mod 10

Separate chaining:



Operations:

find(Bob) = ??

insert(Jon) = ??

delete(Ann) = ??

BRIEF Review of Hash Tables

- insert(k, v) = inserts a key k with value v
  - Duplicate k's may be OK or may not be OK

- find(k) = returns the value v associated to k,
  or the ***list*** of all values associated to k

- delete(k)

Discussion of Hash Tables

- Hash function:
  - Should distribute values uniformly
  - Never write your own! (why is x mod 10 bad?)
    Use a standard library function
  - Best: concatenate with fixed, random seed (in class)
- Hash table:
  - Size of table: large enough to avoid collisions
  - Typically: size of table ≈ size of data
  - Why not make it small? Why not make it big?
  - Problem: hash table allocated statically, at creation
  - Book describes solutions to increase size dynamically

Hash-Based Index

Good for point queries but not range queries

10	21
20	20

30	18
40	19

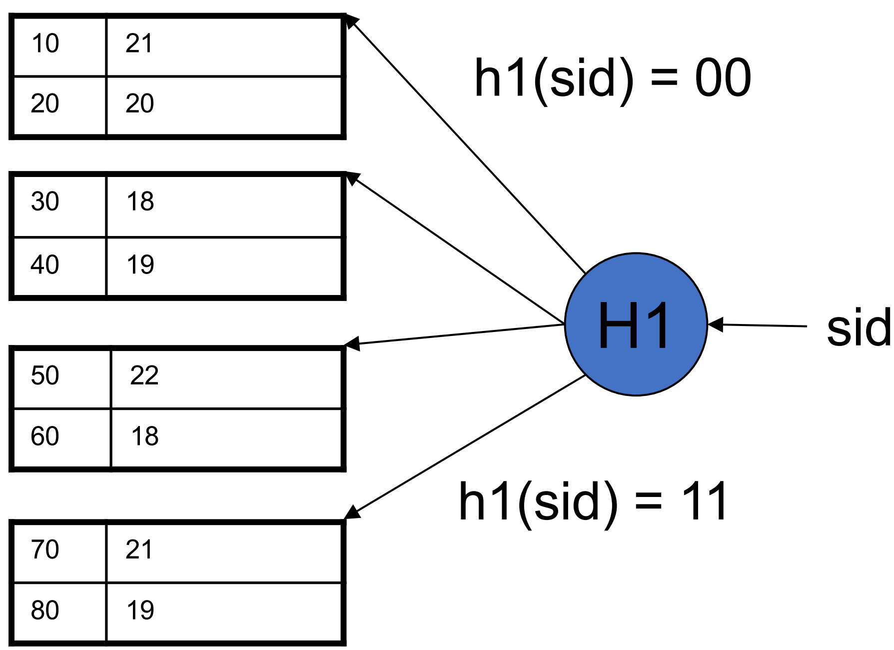
50	22
60	18

70	21
80	19

Data File

Hash-Based Index

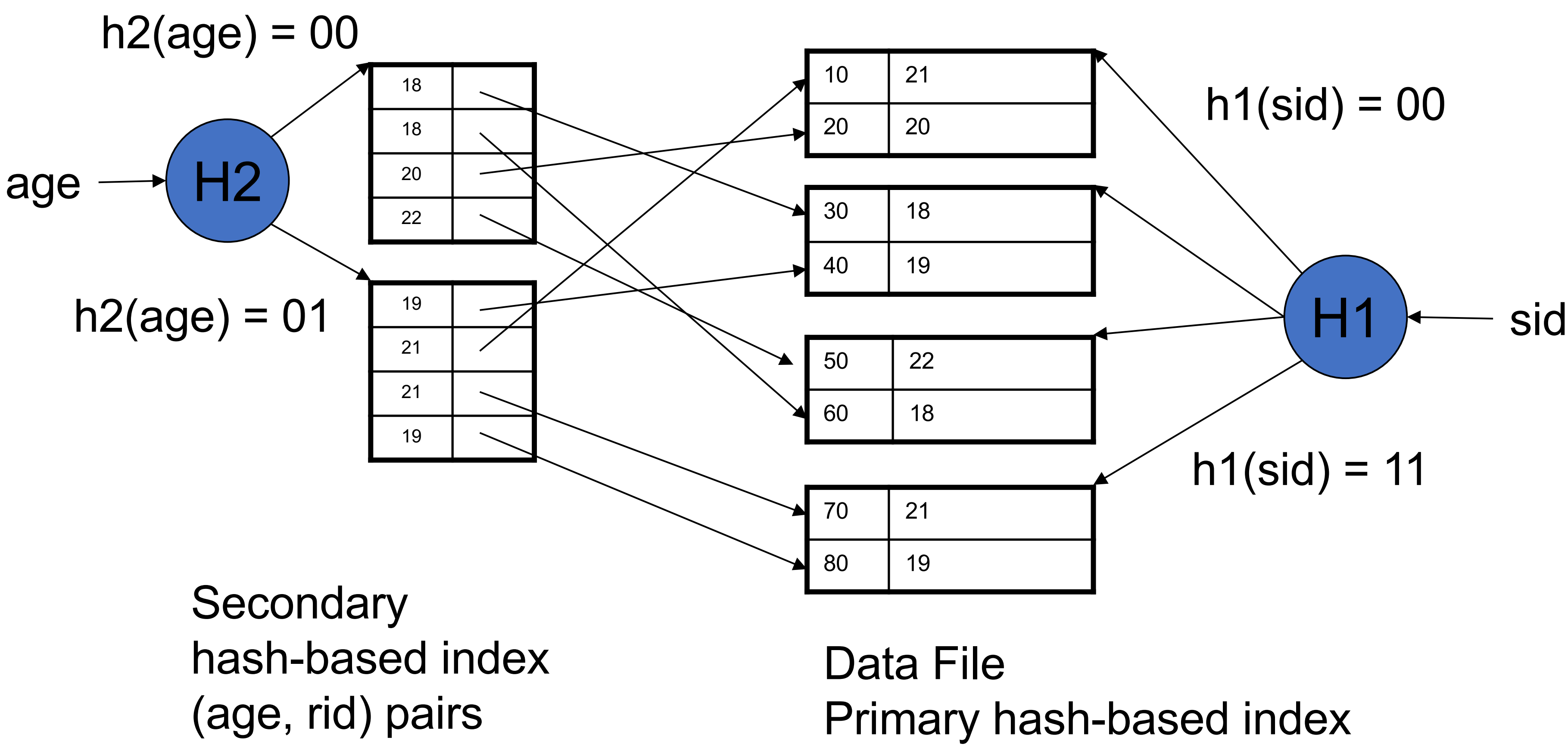
Good for point queries but not range queries



Data File
Primary hash-based index

Hash-Based Index

Good for point queries but not range queries



Making Indexes

- Why not create an index on every attribute?

B+ Trees

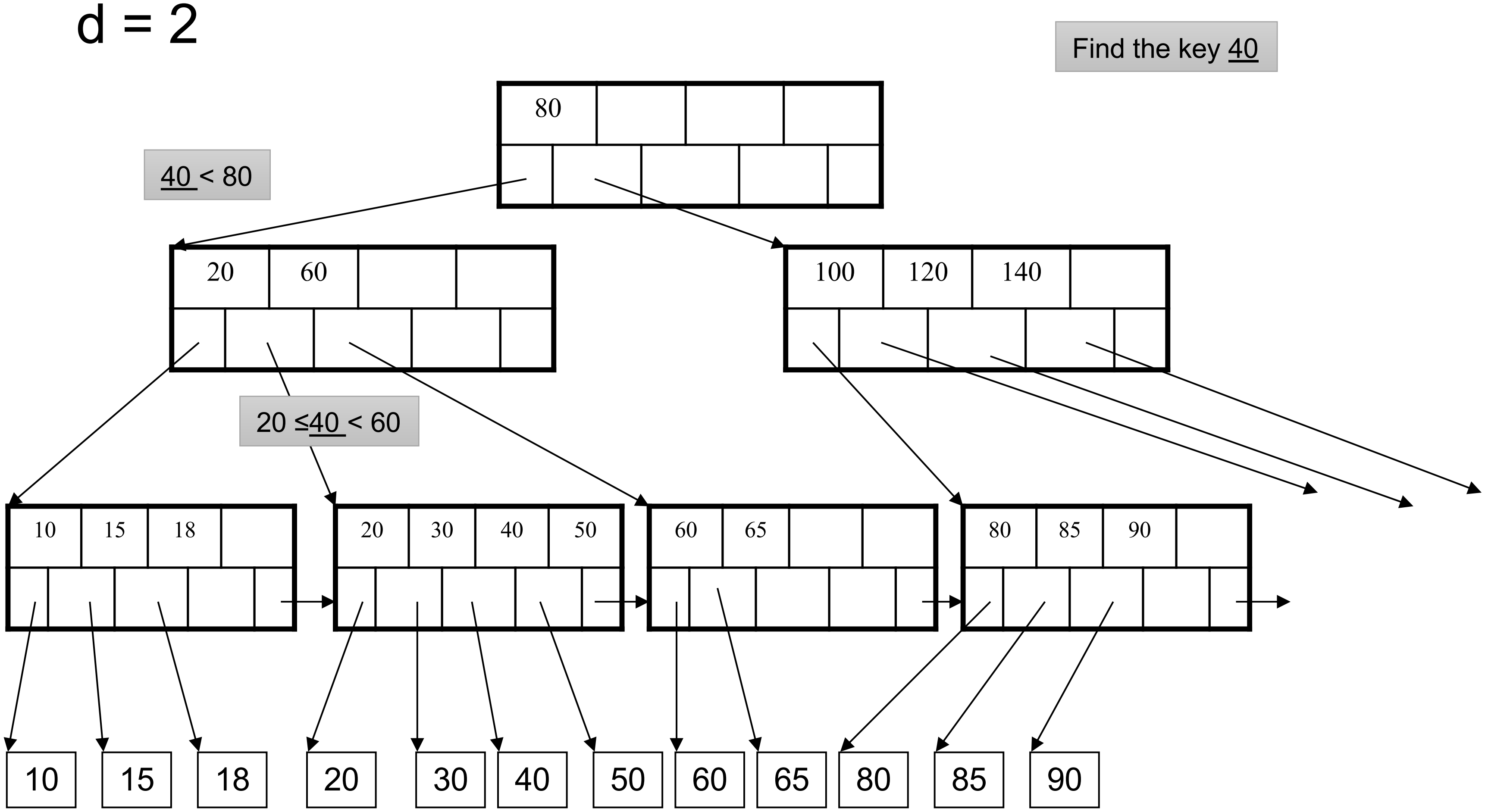
- Search trees (quick review in class)

- Idea in B Trees
  - Make 1 node = 1 page (= 1 block)
  - Maximize number of children per node

- Idea in B+ Trees
  - Keys are stored on the leaves (not internal nodes)
  - Leaves are linked in a list, for range queries

B+ Tree Example

d = 2



B+ Trees Properties

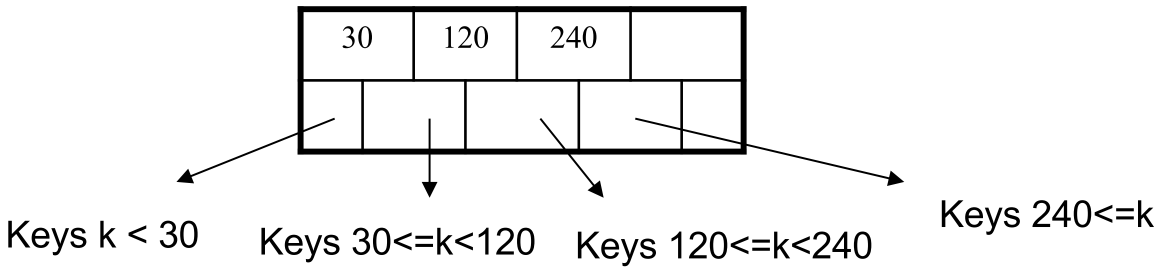
- For each node except the root, maintain 50% occupancy of keys
- Insert and delete must rebalance to maintain constraints

B+ Trees Details

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

B+ Trees Details

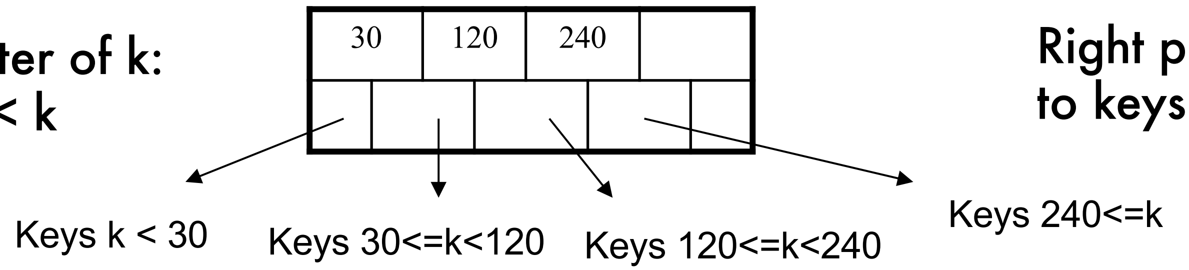
- Parameter d = the <u>degree</u>
- Each node has **d <= m <= 2d keys** (except root)
- Each node also has **m+1 pointers**



B+ Trees Details

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

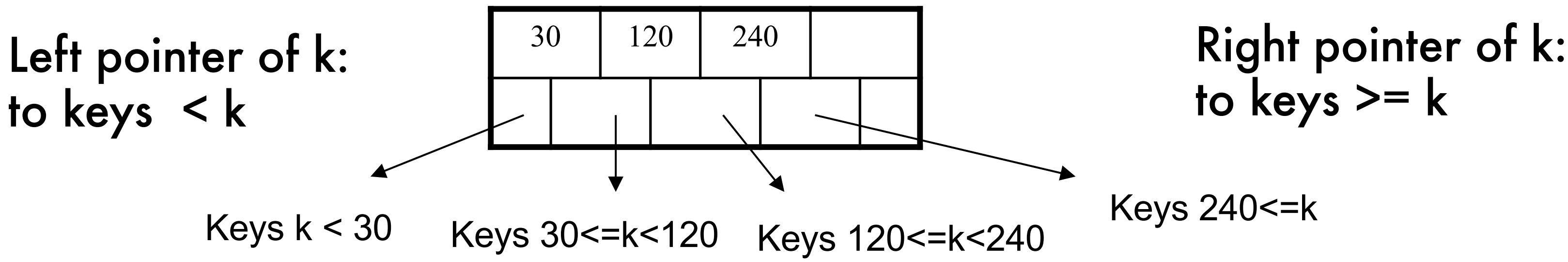
Left pointer of k:
to keys < k



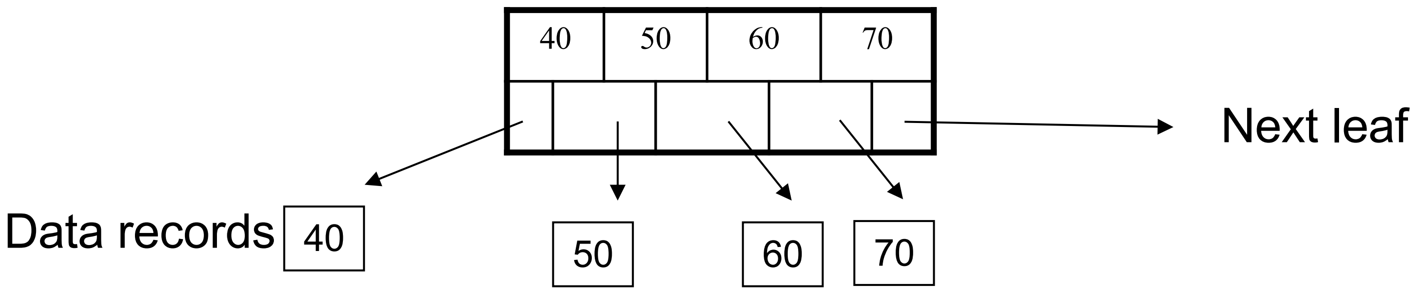
Right pointer of k:
to keys >= k

B+ Trees Details

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

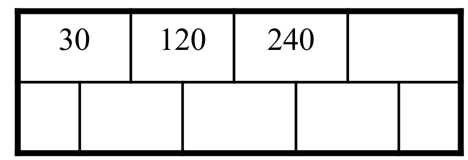


- Each leaf has **d <= m <= 2d keys:**



B+ Tree Design

- How large d ?   Make one node fit on one block
- Example:
  - Key size = 4 bytes
  - Pointer size = 8 bytes
  - Block size = 4096 bytes



(e.g. d = 2)

- 2d x 4 + (2d+1) x 8 <= 4096
- d = 170

B+ Trees in Practice

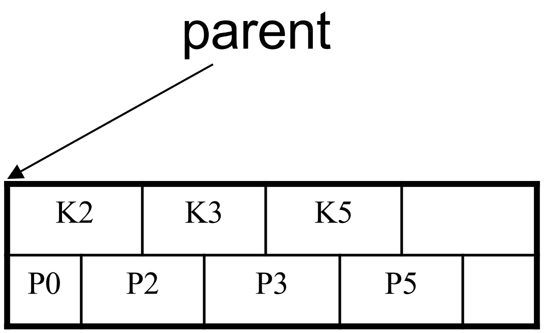
- Typical order: d=100. Typical fill-factor: 66%.
  - average node fanout (children) = 200*0.66 = 133
- Typical capacities
  - Height 4: 133^4 = 312,900,700 records
  - Height 3: 133^3 = 2,352,637 records
- Can often hold top levels in buffer pool
  - Level 1 = 1 page = 8 Kbytes
  - Level 2 = 133 pages = 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

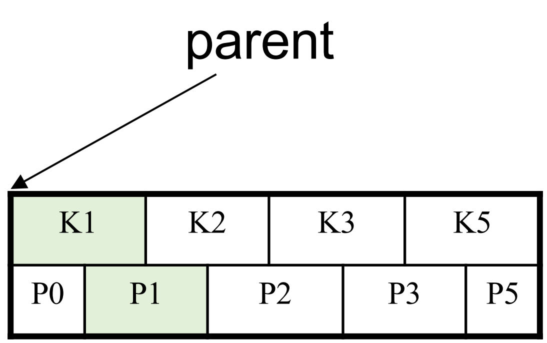


Insertion in a B+ Tree

Insert (K, P)

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

Insert k1

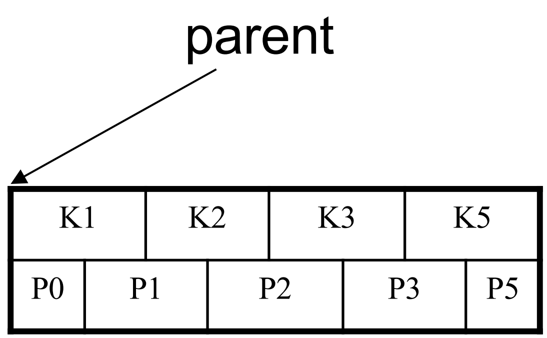


Insertion in a B+ Tree

Insert (K, P)

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

Insert k4

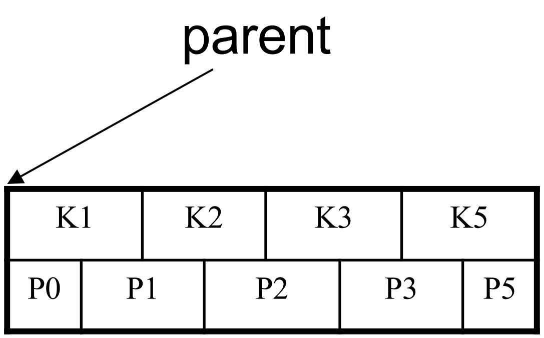


Insertion in a B+ Tree

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

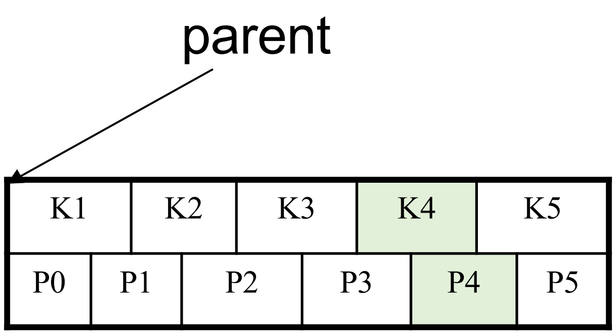


Insertion in a B+ Tree

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

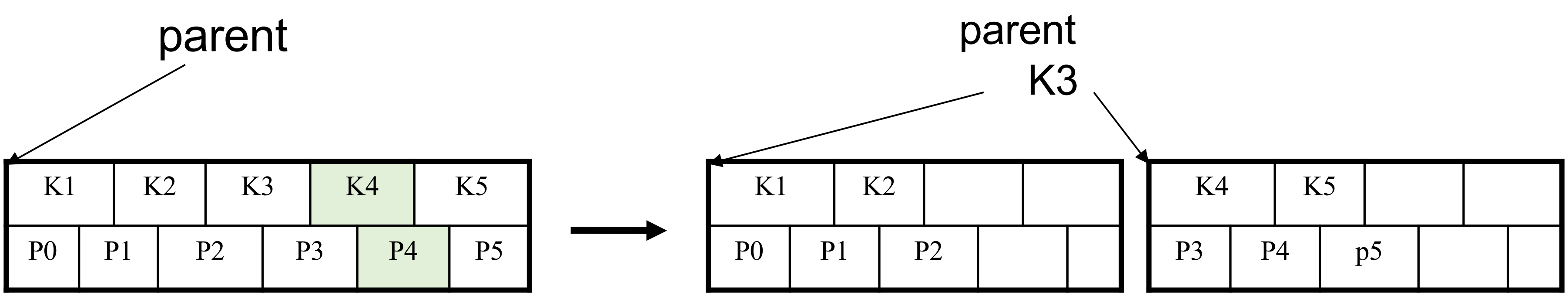


Insertion in a B+ Tree

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:

Insert k4

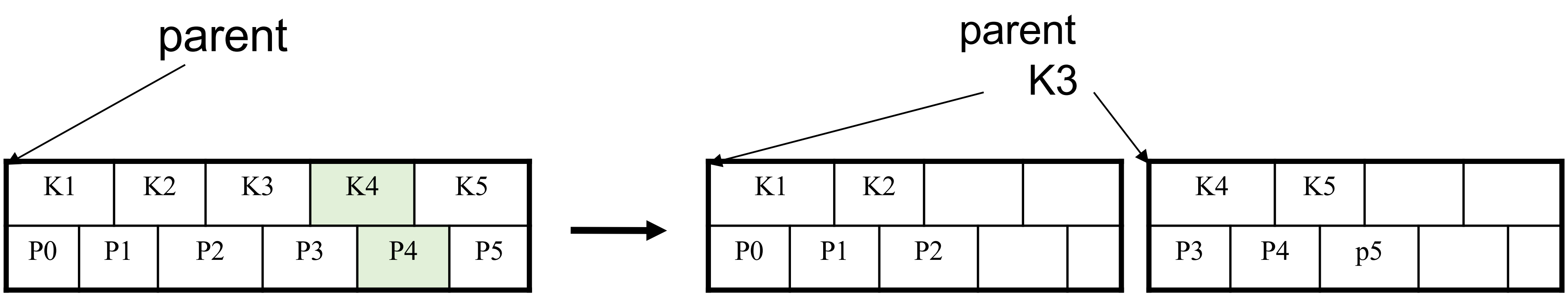


Insertion in a B+ Tree

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:

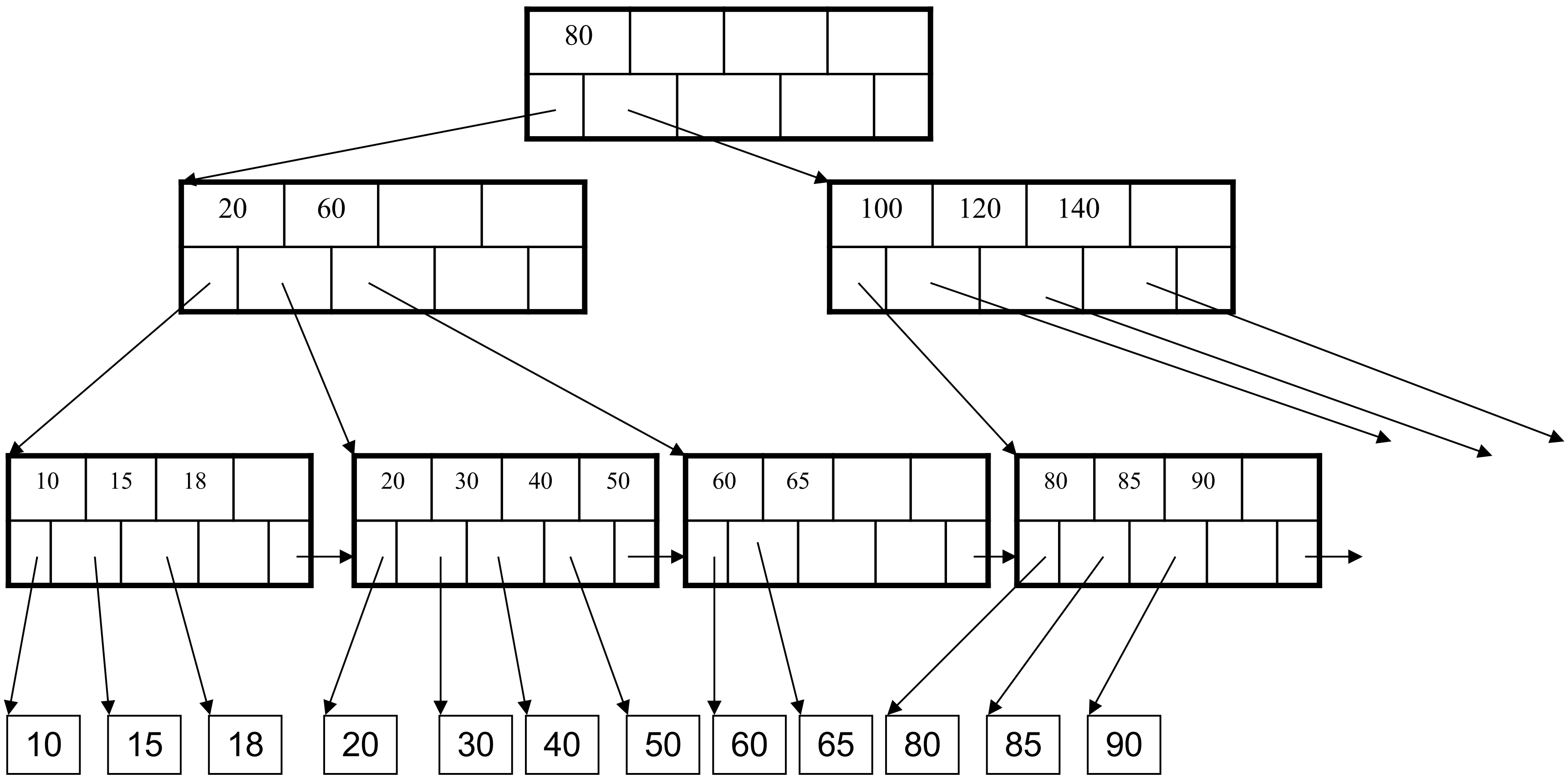
Insert k4



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

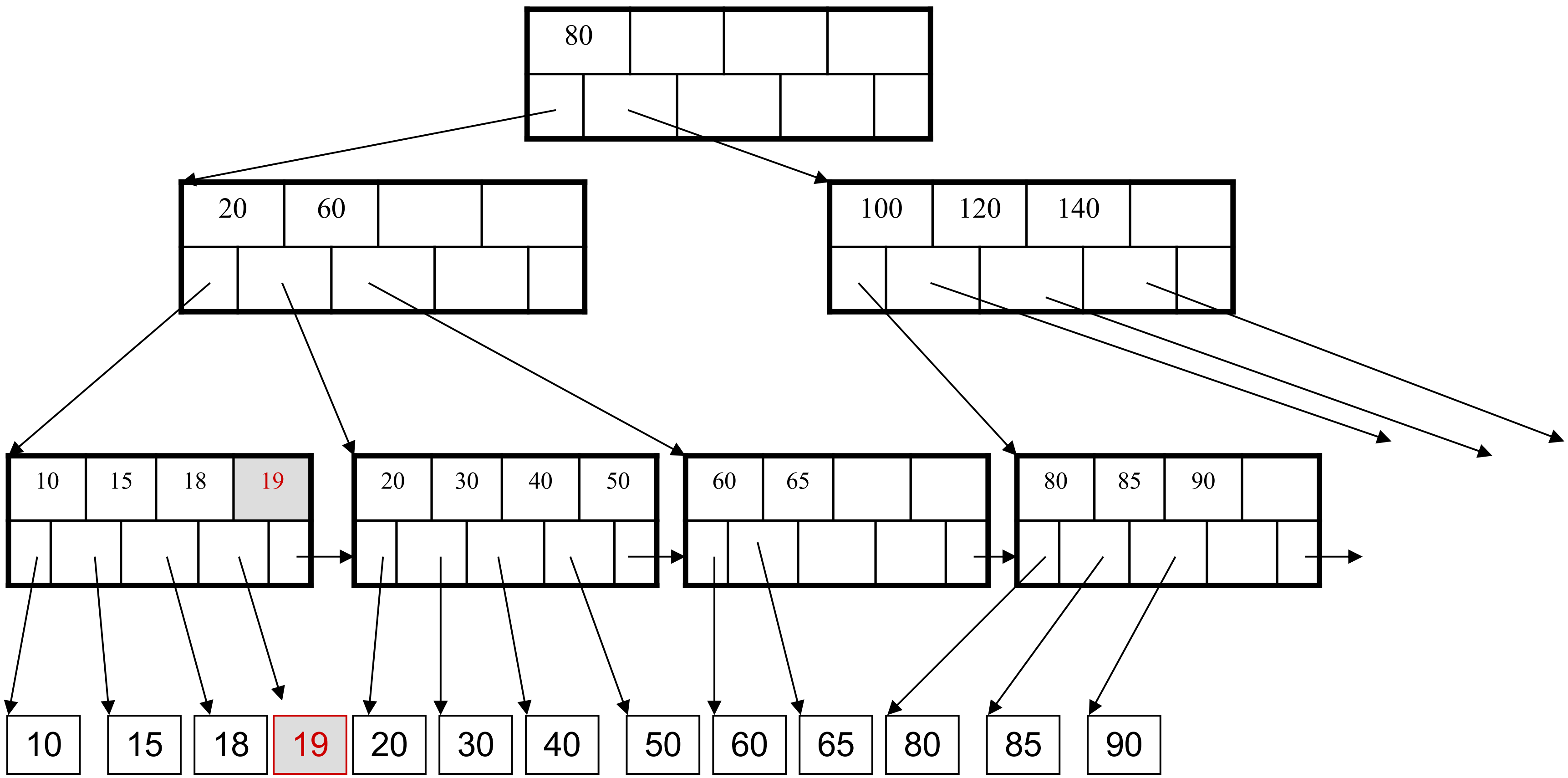
Insertion in a B+ Tree

Insert K=19



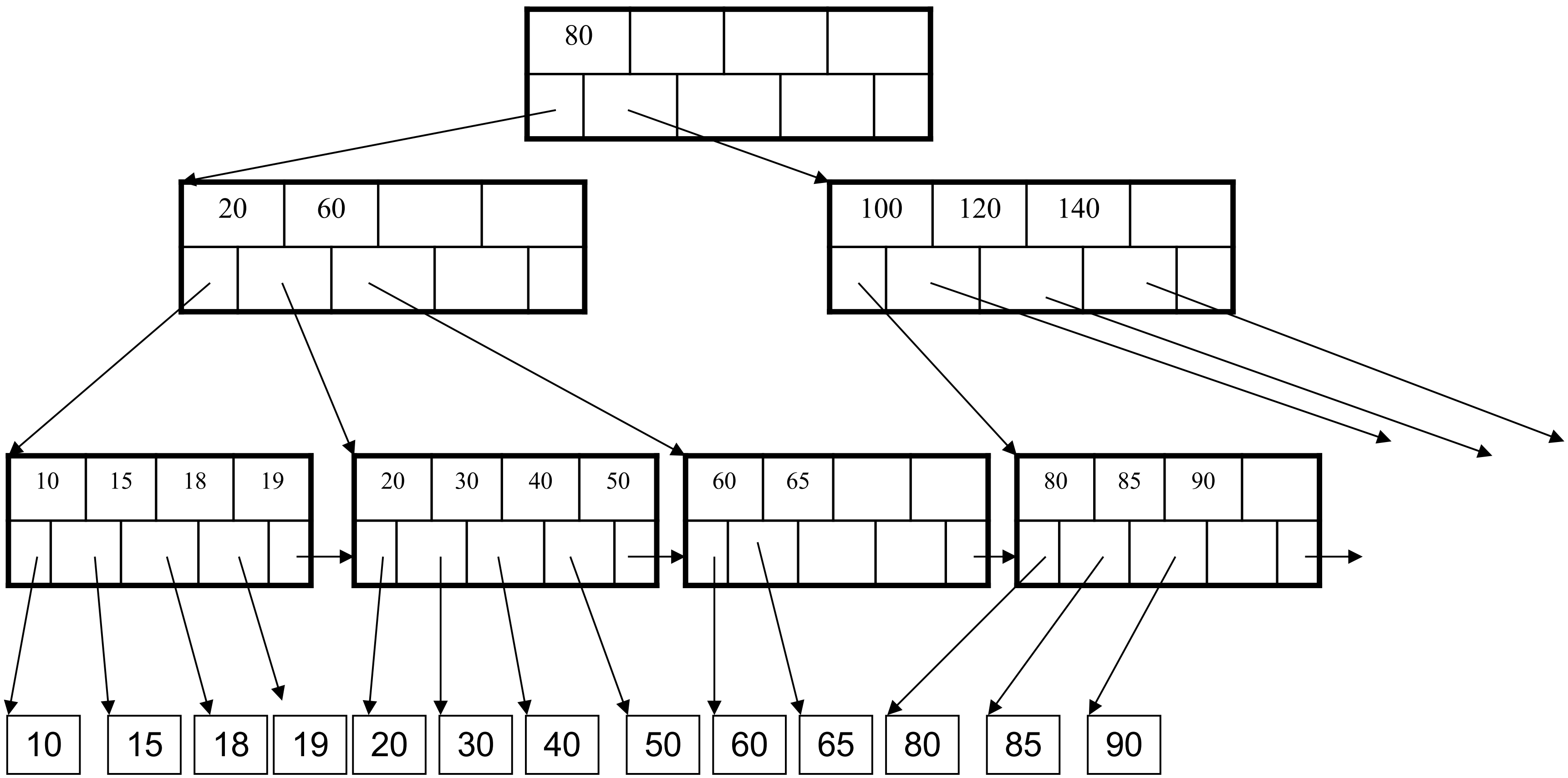
Insertion in a B+ Tree

After insertion



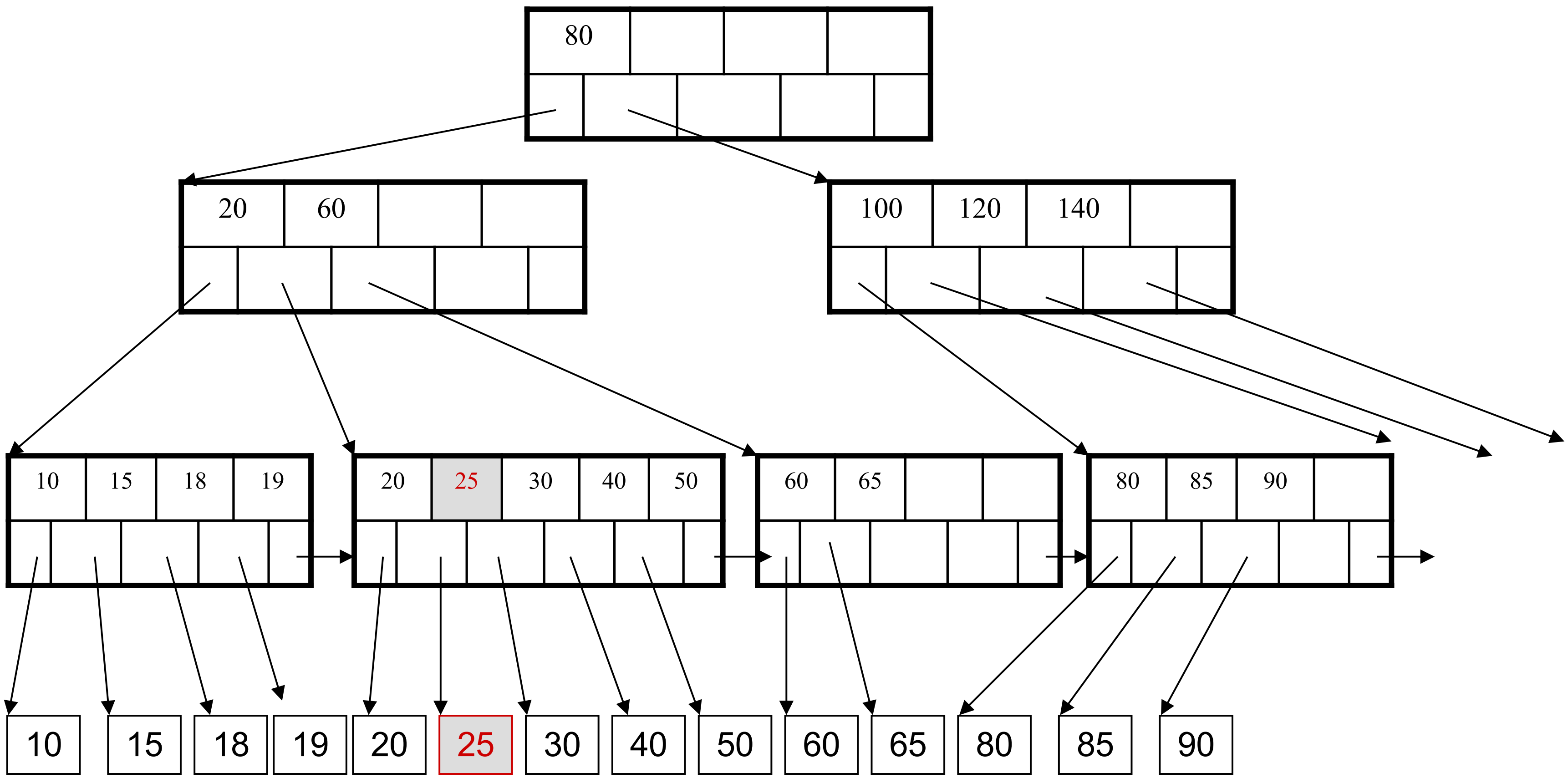
Insertion in a B+ Tree

Now insert 25



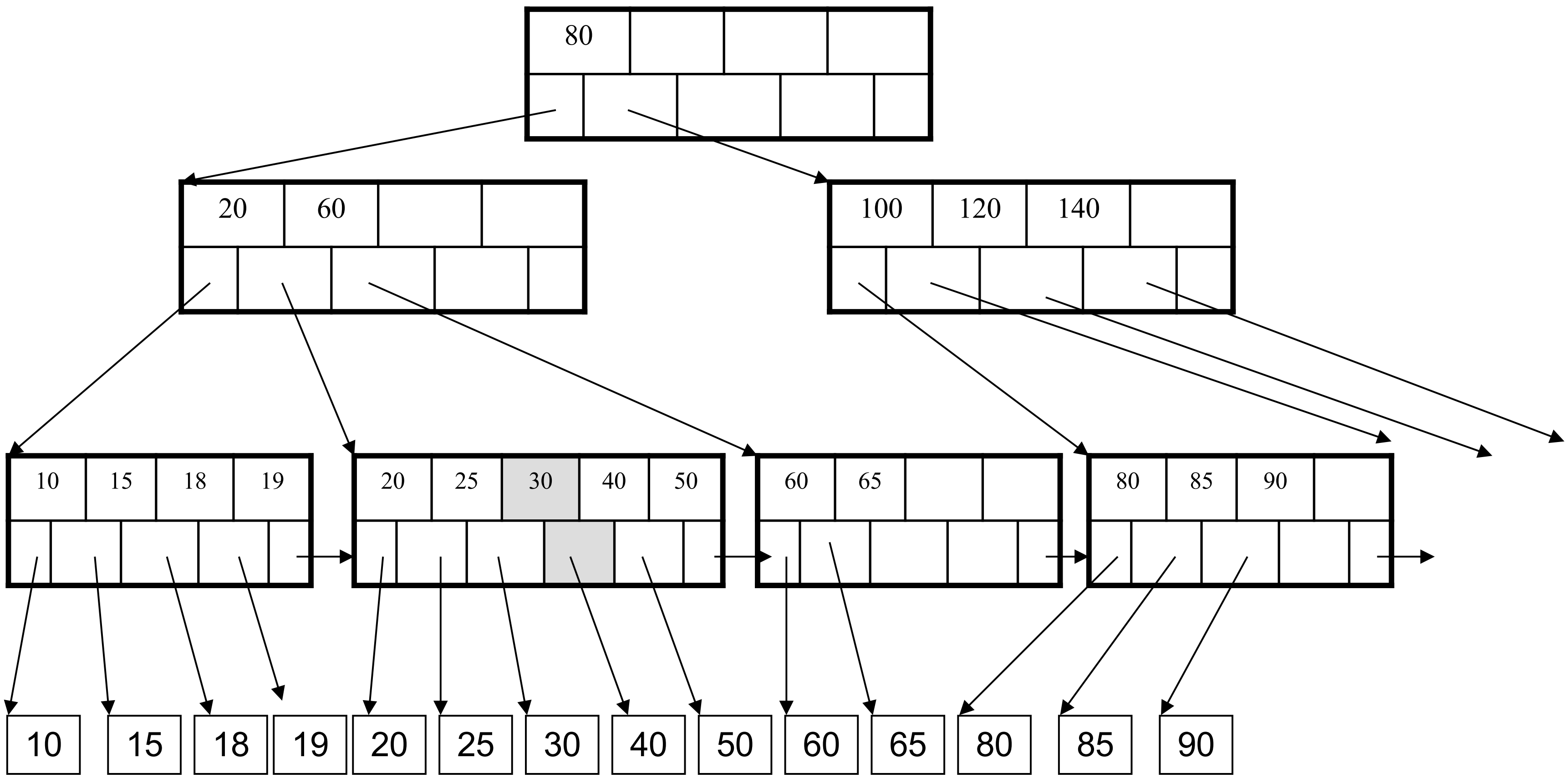
Insertion in a B+ Tree

After insertion



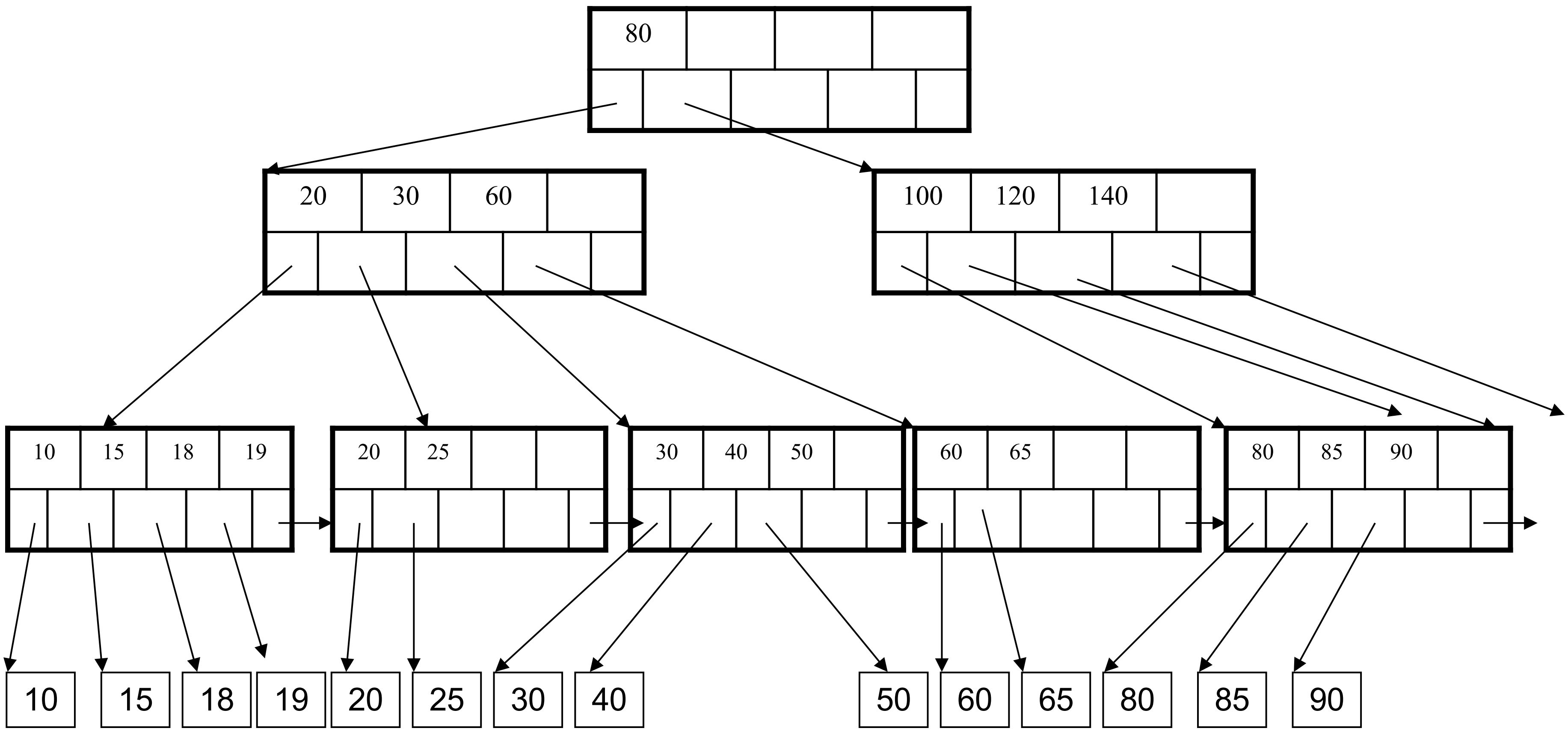
Insertion in a B+ Tree

But now have to split !



Insertion in a B+ Tree

After the split



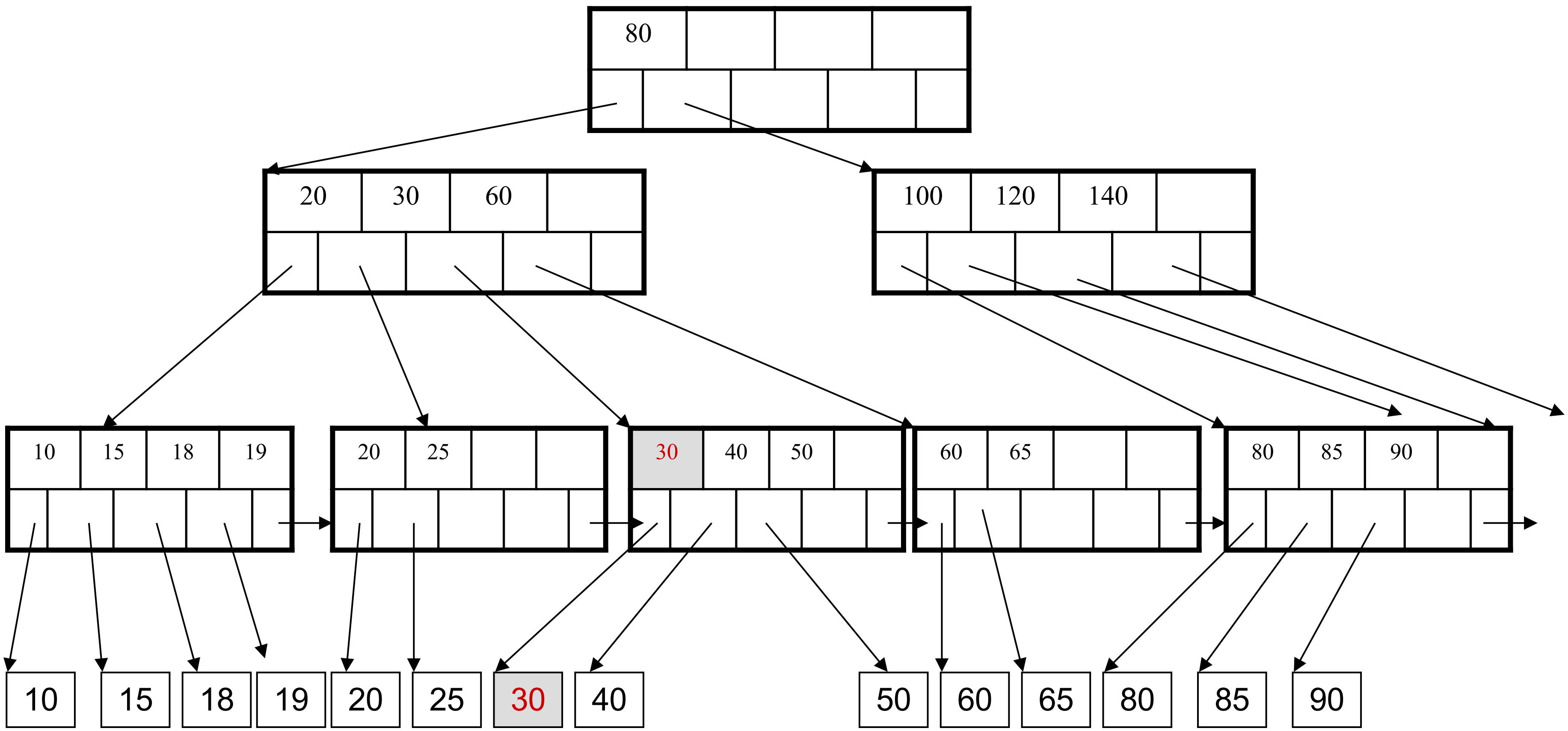
Deletion in a B+ Tree

Delete (K, P)

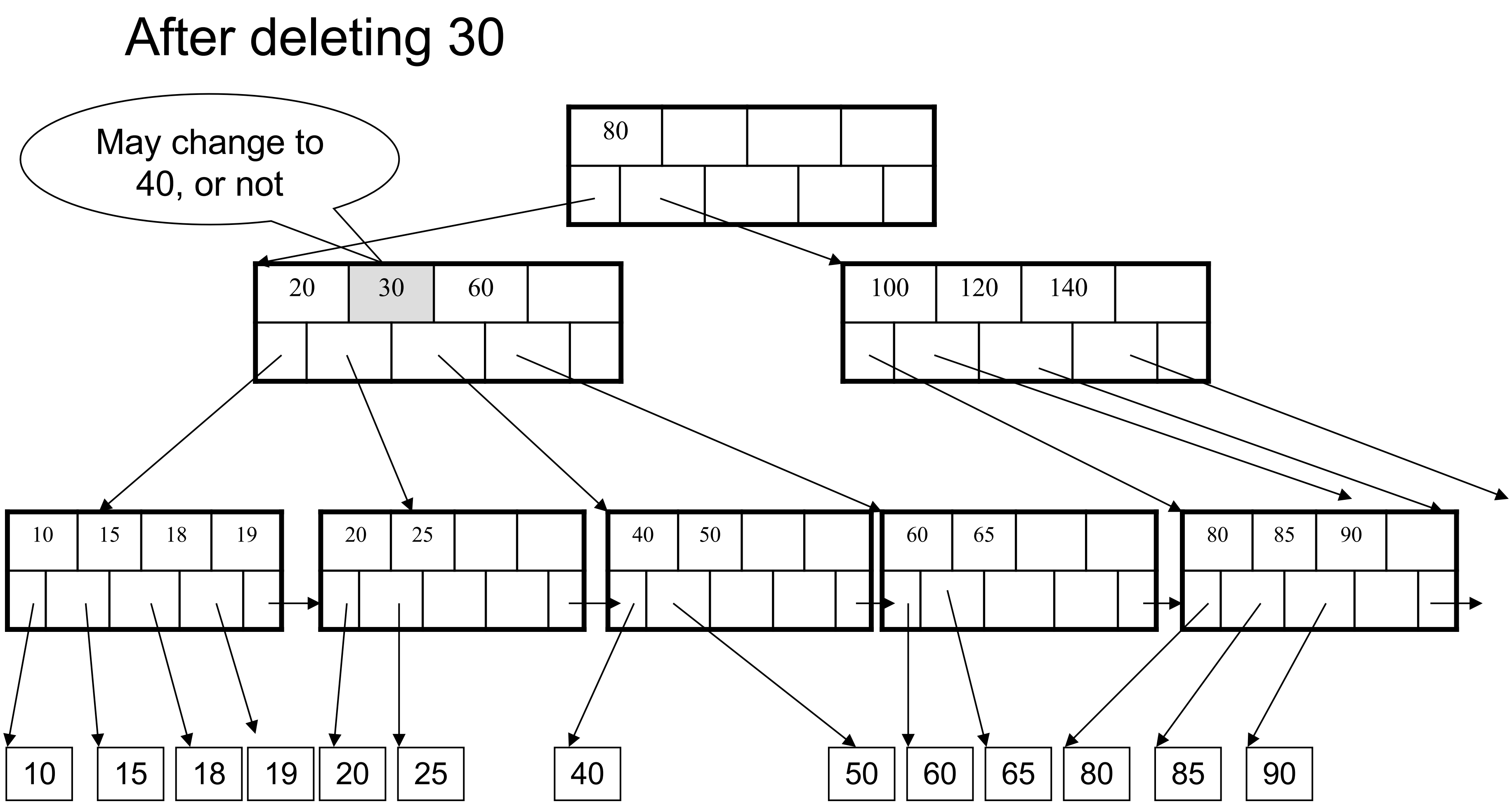
- Find leaf where K belongs, delete
- Check for capacity
- If leaf below capacity, search adjacent nodes (left first, then right) for extra tuples and rotate them to new leaf
- If adjacent nodes 50% full, merge with on adjacent node
  This removes a key/child from parent;
  repeat algorithm on parent node

Deletion from a B+ Tree

Delete 30

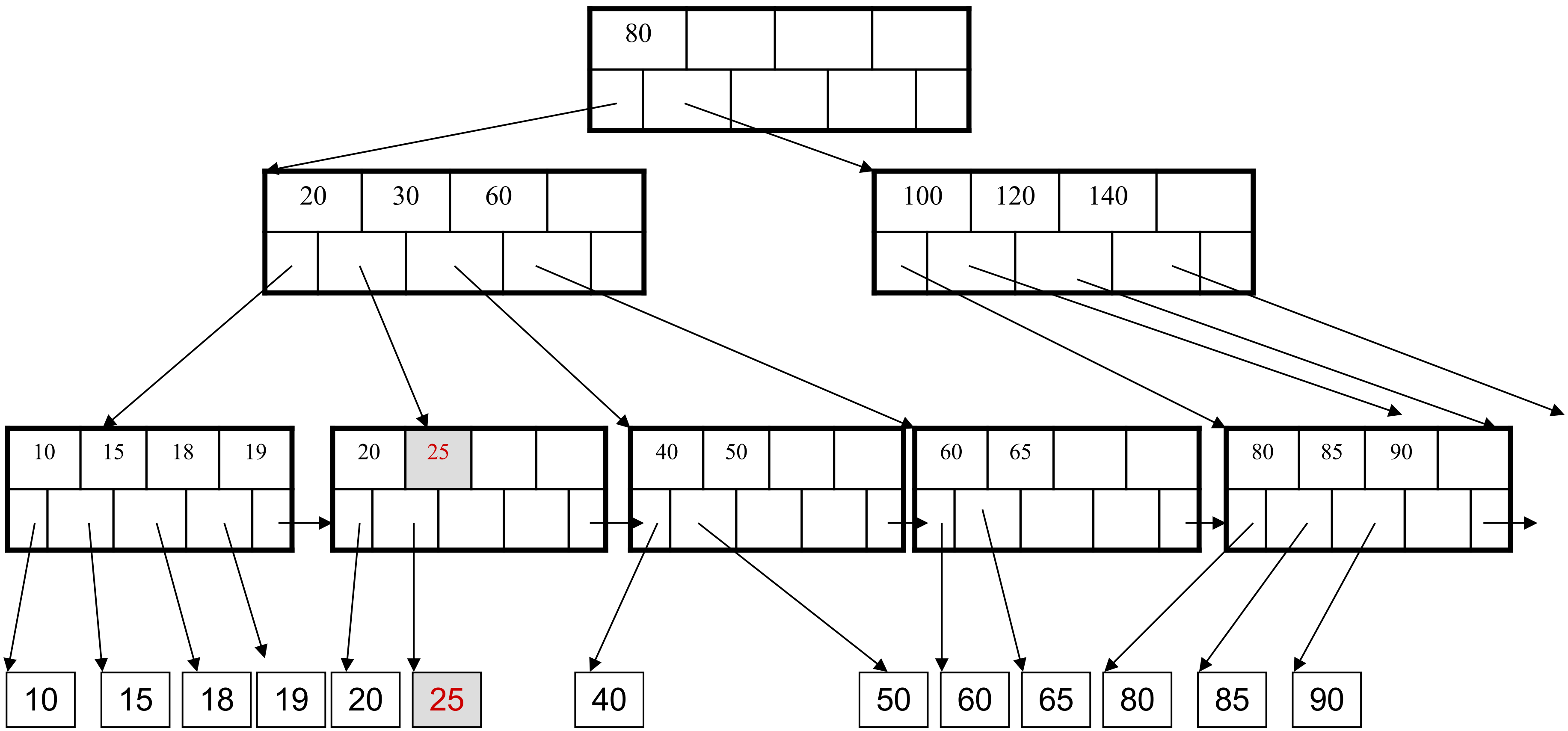


Deletion from a B+ Tree



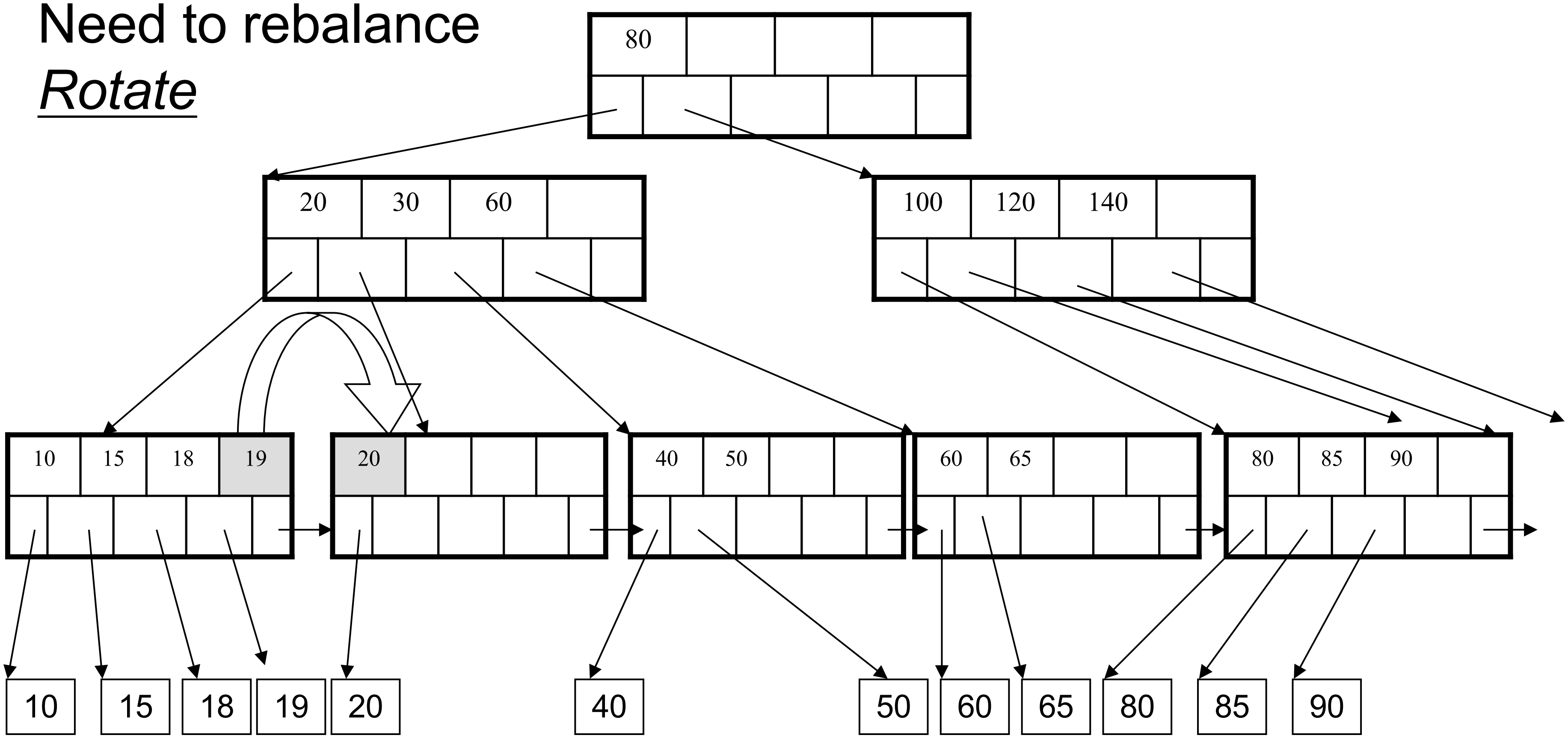
Deletion from a B+ Tree

Now delete 25



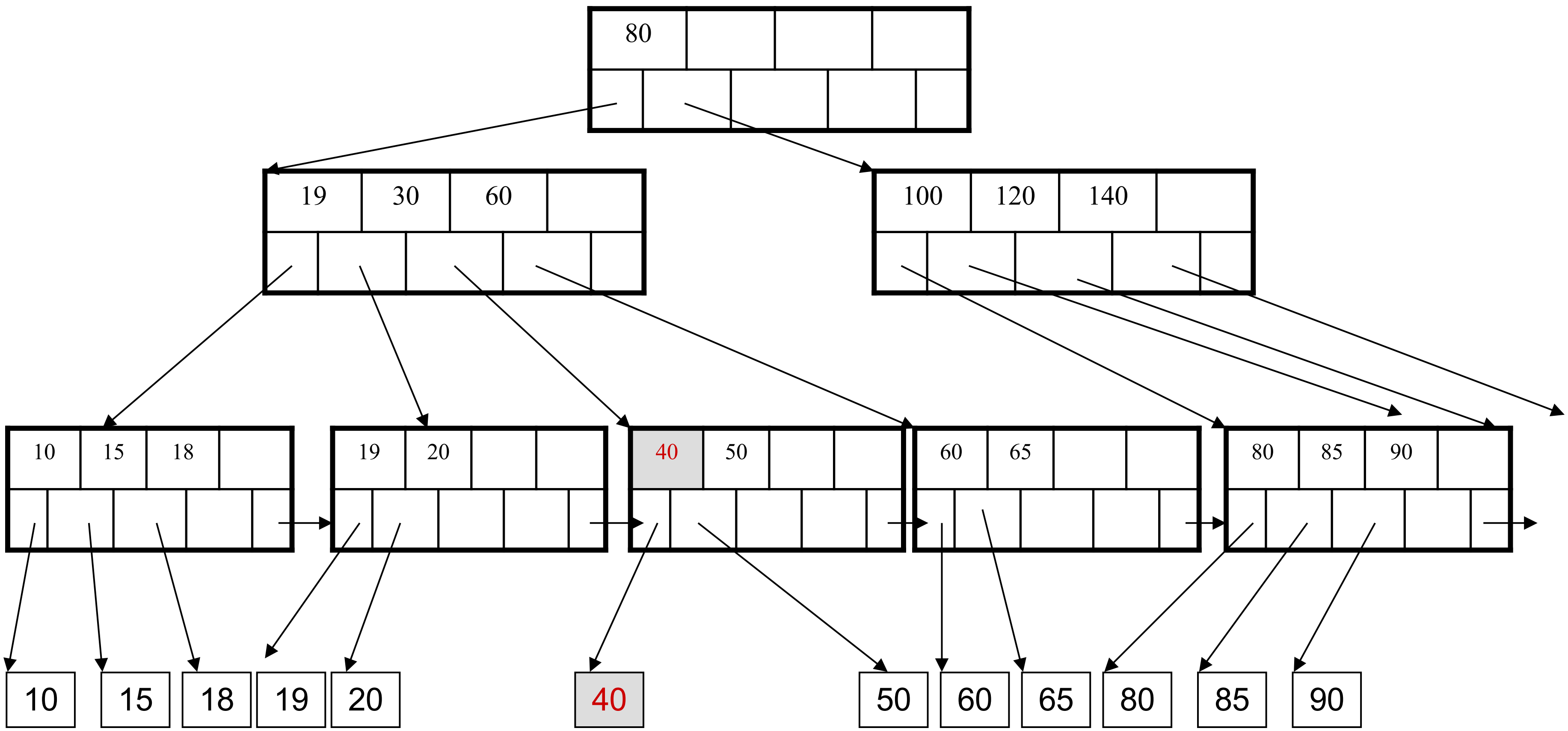
Deletion from a B+ Tree

After deleting 25
Need to rebalance
Rotate



Deletion from a B+ Tree

Now delete 40

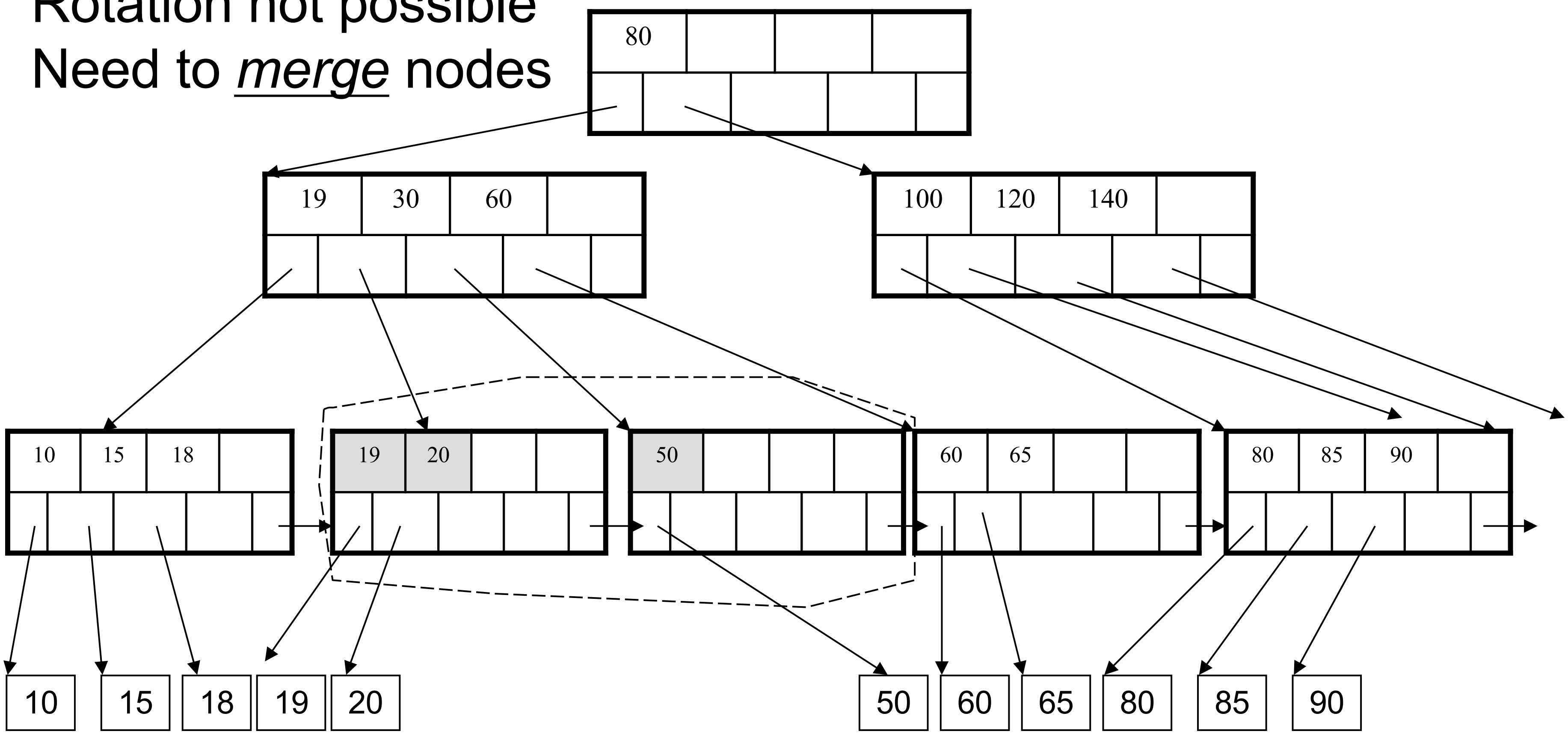


Deletion from a B+ Tree

After deleting 40

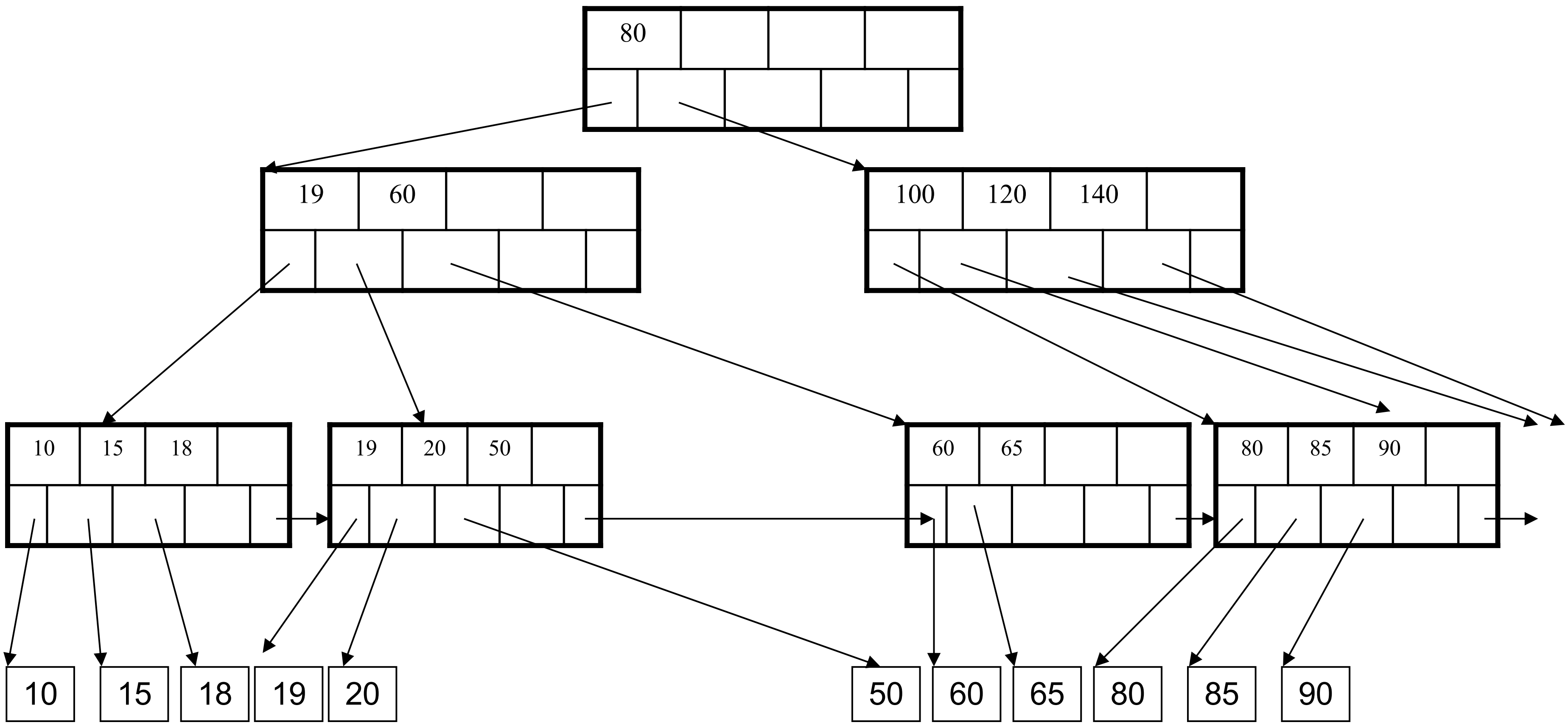
Rotation not possible

Need to *merge* nodes

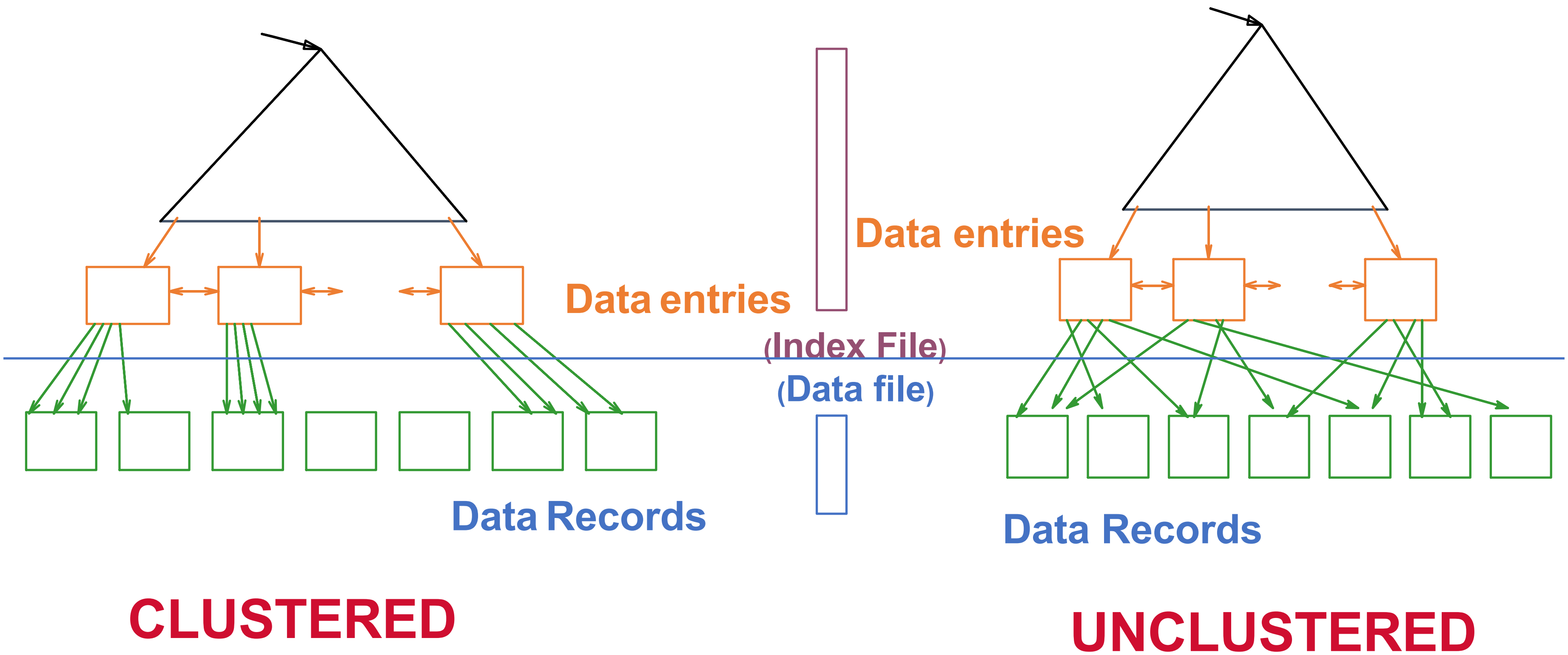


Deletion from a B+ Tree

Final tree



Clustered v.s. Unclustered B+ Trees



Note: can also store data records directly as data entries

Searching a B+ Tree

- Exact key values:
  - Start at the root
  - Proceed down, to the leaf

```
Select name
From Student
Where age = 25
```

- Range queries:
  - Find lowest bound as above
  - Then sequential traversal

```
Select name
From Student
Where 20 <= age
  and  age <= 30
```

- Less effective for multi-range
  - Can only use one B+ tree, ignore the other(s)
  - Called *access path selection*

```
Select name
From Student
Where age = 25
     and GPA = 3.5
```

Summary on B+ Trees

- Default index structure on most DBMSs
- Many improvements/optimizations
  - Prefix compression:
    "Johannes", "John", "Johnson", "Jon",…
    store only suffices, to save space
  - Allow fill capacity to decrease slightly below 50% to avoid cascading splits and merges
  - Optimizations for transactions: tree-locking protocol instead of Strict 2PL
- For multi-dimensional queries, need R-trees:
  - E.g. age = 25 and GPA > 3.5
  - R-trees are more difficult to search and rebalance