CSC 261/461 – Database Systems
Lecture 18

Spring 2018

Announcement

- Quiz 8 WAS due at 2:59 pm today

- Project 2 Part 2 is due tomorrow:
  - 03/29/2018 (11:59 pm)

TYPES OF INDEXES

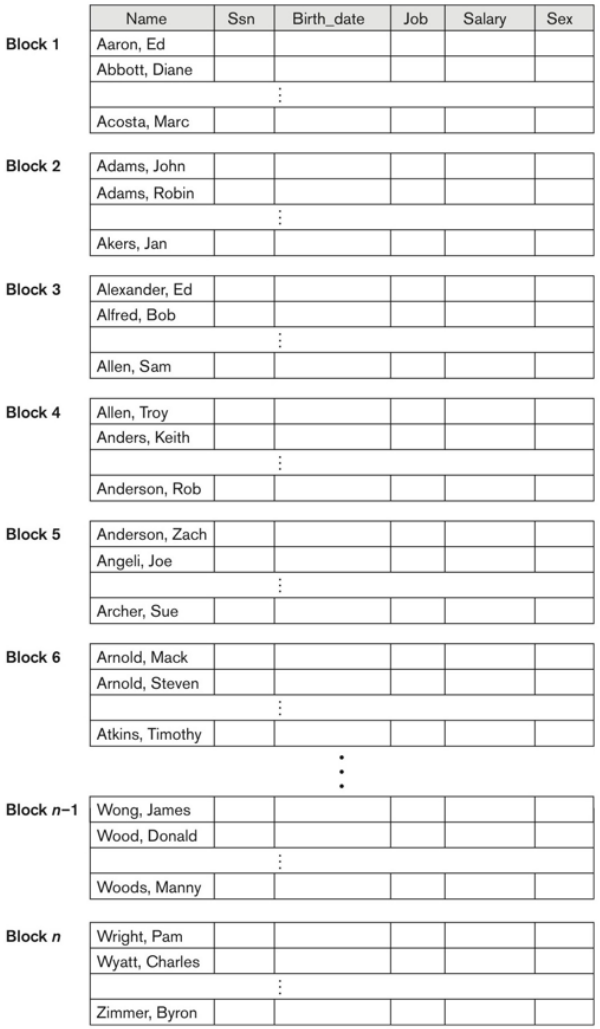
Types of Indexing

- Primary Indexes
- Clustering Indexes
- Secondary Indexes
- Multilevel Indexes
  - Dynamic Multilevel Indexes
- Hash Indexes
- Easy introduction: https://www.tutorialspoint.com/dbms/dbms_indexing.htm

Sorted Files

- Fig 16.7

Recap: No Indexing



	Name	Ssn	Birth_date	Job	Salary	Sex
Block 1	Aaron, Ed					
	Abbott, Diane					
	⋮					
	Acosta, Marc					
Block 2	Adams, John					
	Adams, Robin					
	⋮					
	Akers, Jan					
Block 3	Alexander, Ed					
	Alfred, Bob					
	⋮					
	Allen, Sam					
Block 4	Allen, Troy					
	Anders, Keith					
	⋮					
	Anderson, Rob					
Block 5	Anderson, Zach					
	Angeli, Joe					
	⋮					
	Archer, Sue					
Block 6	Arnold, Mack					
	Arnold, Steven					
	⋮					
	Atkins, Timothy					
	⋮					
Block *n*−1	Wong, James					
	Wood, Donald					
	⋮					
	Woods, Manny					
Block *n*	Wright, Pam					
	Wyatt, Charles					
	⋮					
	Zimmer, Byron					

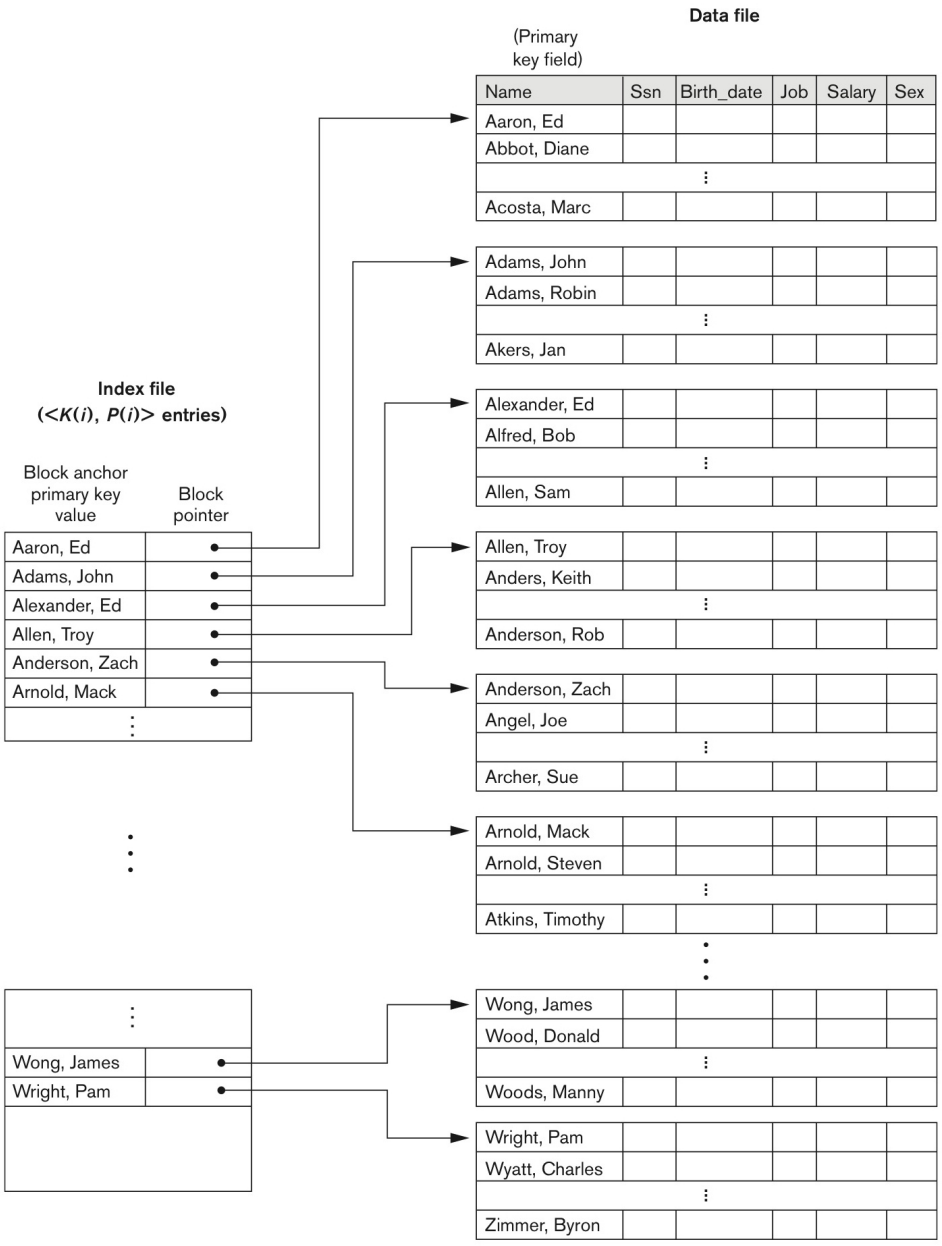
Sorted Files (zoomed)

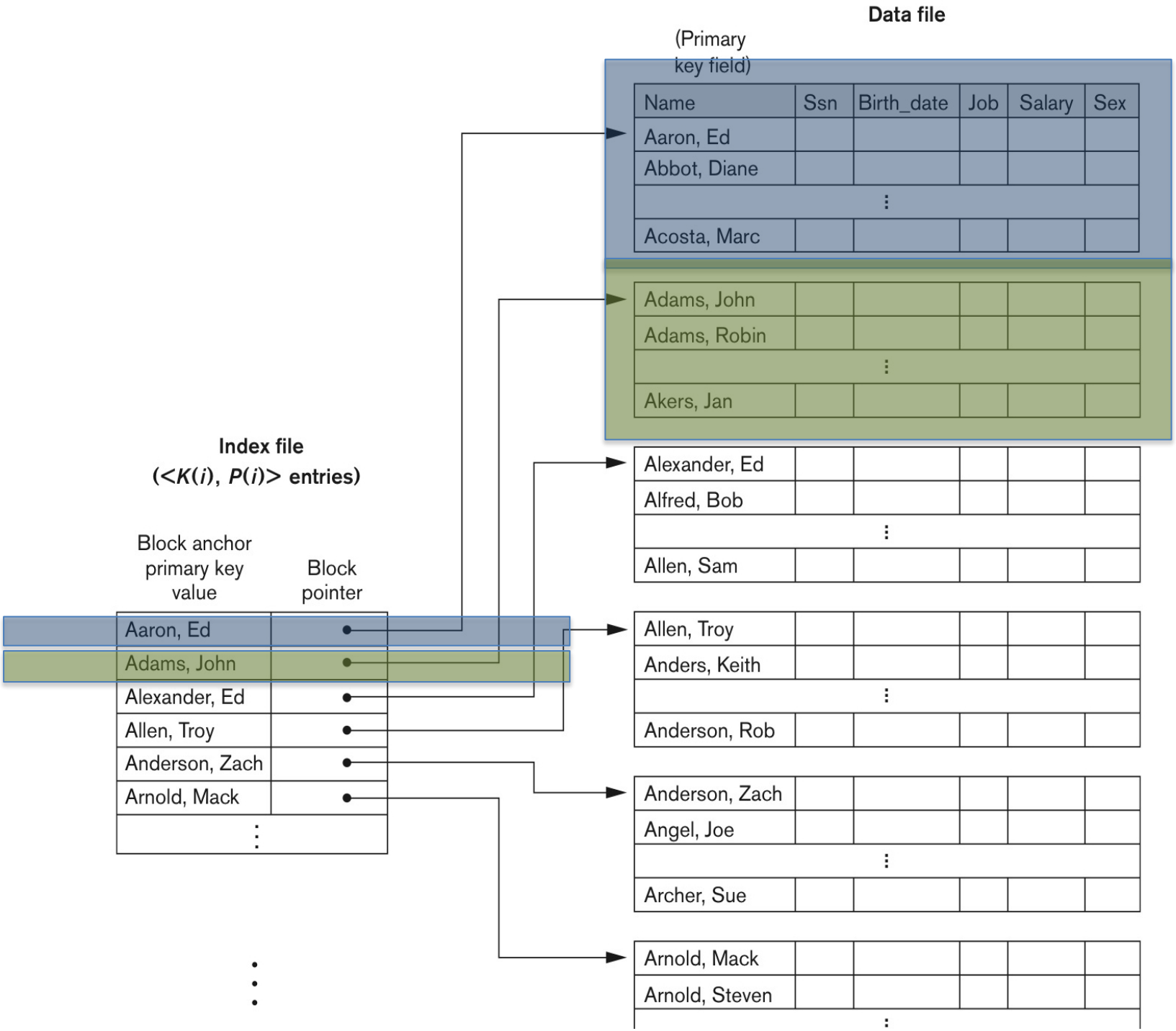
- Fig 16.7

Recap: No Indexing

	Name	Ssn	Birth_date	Job	Salary	Sex
Block 1	Aaron, Ed					
	Abbott, Diane					
	⋮					
	Acosta, Marc					
Block 2	Adams, John					
	Adams, Robin					
	⋮					
	Akers, Jan					

Primary Indexes: Index for Sorted (Ordered) Files




Data file
(Primary key field)
Name
Ssn
Birth_date
Job
Salary
Sex
Aaron, Ed
Abbot, Diane
Acosta, Marc
Adams, John
Adams, Robin
Akers, Jan
Index file
(<K(i), P(i)> entries)
Block anchor primary key value
Block pointer
Aaron, Ed
Adams, John
Alexander, Ed
Allen, Troy
Anderson, Zach
Arnold, Mack
Alexander, Ed
Alfred, Bob
Allen, Sam
Allen, Troy
Anders, Keith
Anderson, Rob
Anderson, Zach
Angel, Joe
Archer, Sue
Arnold, Mack
Arnold, Steven

Example 1

- Suppose that we have an ordered file with r = 30,000 records stored on a disk with block size B = 1024 bytes.
- File records are of fixed size and are unspanned, with record length R = 100 bytes. Also, suppose that the ordering key field of the file is V = 9 bytes long, a block pointer is P = 6 bytes long, and we have constructed a primary index for the file.

- Find out:

1. Cost of searching for a record using Binary Search on the data file
2. Cost of searching for a record using the index

Example 1 (Part 1)

- Suppose that we have an ordered file with r = 30,000 records stored on a disk with block size B = 1024 bytes.
- File records are of fixed size and are unspanned, with record length R = 100 bytes. Also, suppose that the ordering key field of the file is V = 9 bytes long, a block pointer is P = 6 bytes long, and we have constructed a primary index for the file.

Find out the cost of searching for a record using Binary Search on the data file

Answer

- Blocking factor:
- bfr = $\lfloor (B/R) \rfloor = \lfloor (1024/100) \rfloor = 10$ records per block.

- The number of blocks needed for the file is
  - b = $\lceil (r/bfr) \rceil = \lceil (30000/10) \rceil = 3000$ blocks.

- A binary search on the data file would need approximately
- $\lceil \log_2 b \rceil = \lceil (\log_2 3000) \rceil = 12$ block accesses.

Example 1 (Part 2)

- Suppose that we have an ordered file with r = 30,000 records stored on a disk with block size B = 1024 bytes.
- File records are of fixed size and are unspanned, with record length R = 100 bytes. Also, suppose that the ordering key field of the file is V = 9 bytes long, a block pointer is P = 6 bytes long, and we have constructed a primary index for the file.

- Find out the cost of searching for a record using the index

Answer

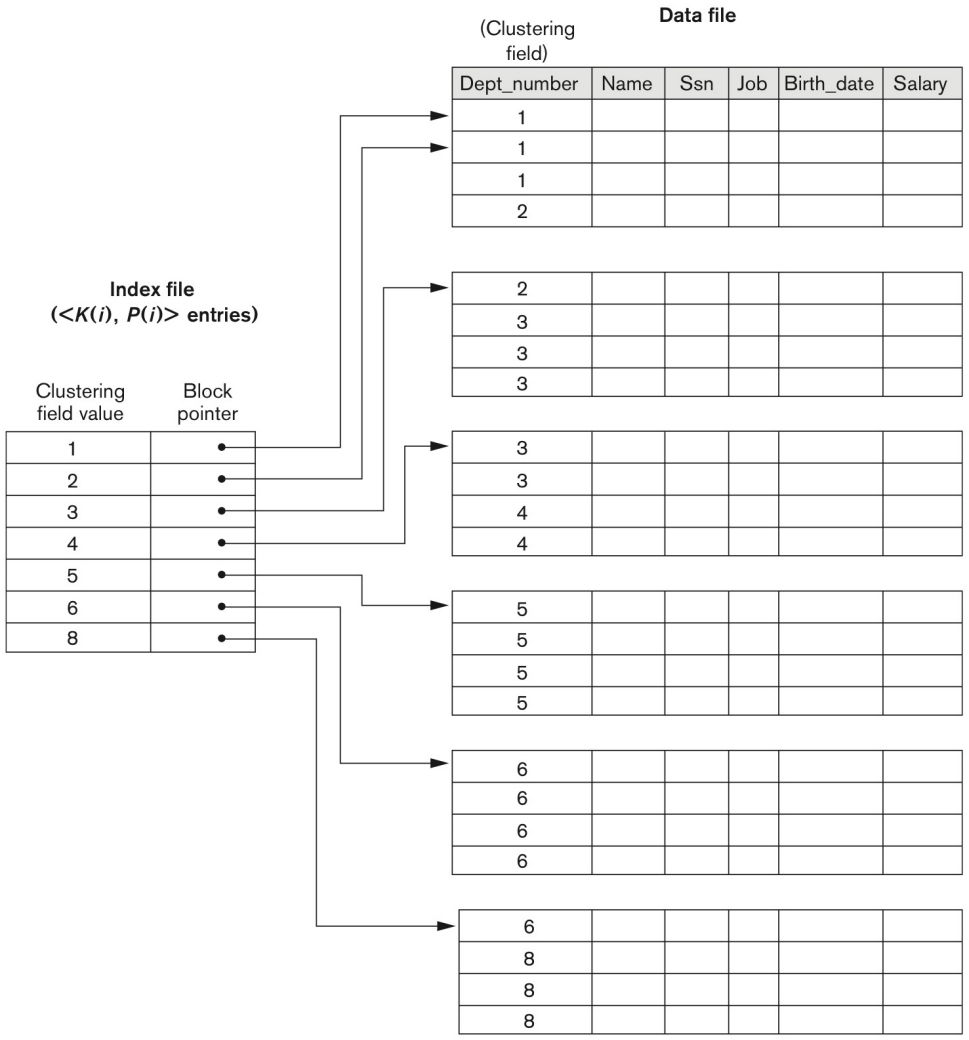
- The size of each index entry :
  - Ri= (9 + 6) = 15 bytes,

  So the blocking factor for the index is $bfr_i = \lfloor (B/Ri) \rfloor = \lfloor (1024/15) \rfloor = 68$ entries per block.

- The total number of index entries is equal to the number of blocks in the data file, which is 3000.
- The number of index blocks is hence $b_i = \lceil (ri/bfri) \rceil = \lceil (3000/68) \rceil = 45$ blocks.
- To perform a binary search on the index file would need $\lceil (\log_2 bi) \rceil = \lceil (\log_2 45) \rceil = 6$ block accesses.

- To search for a record using the index, we need one additional block access to the data file for a total of 6 + 1 = 7 block accesses—an improvement over binary search on the data file, which required 12 disk block accesses.

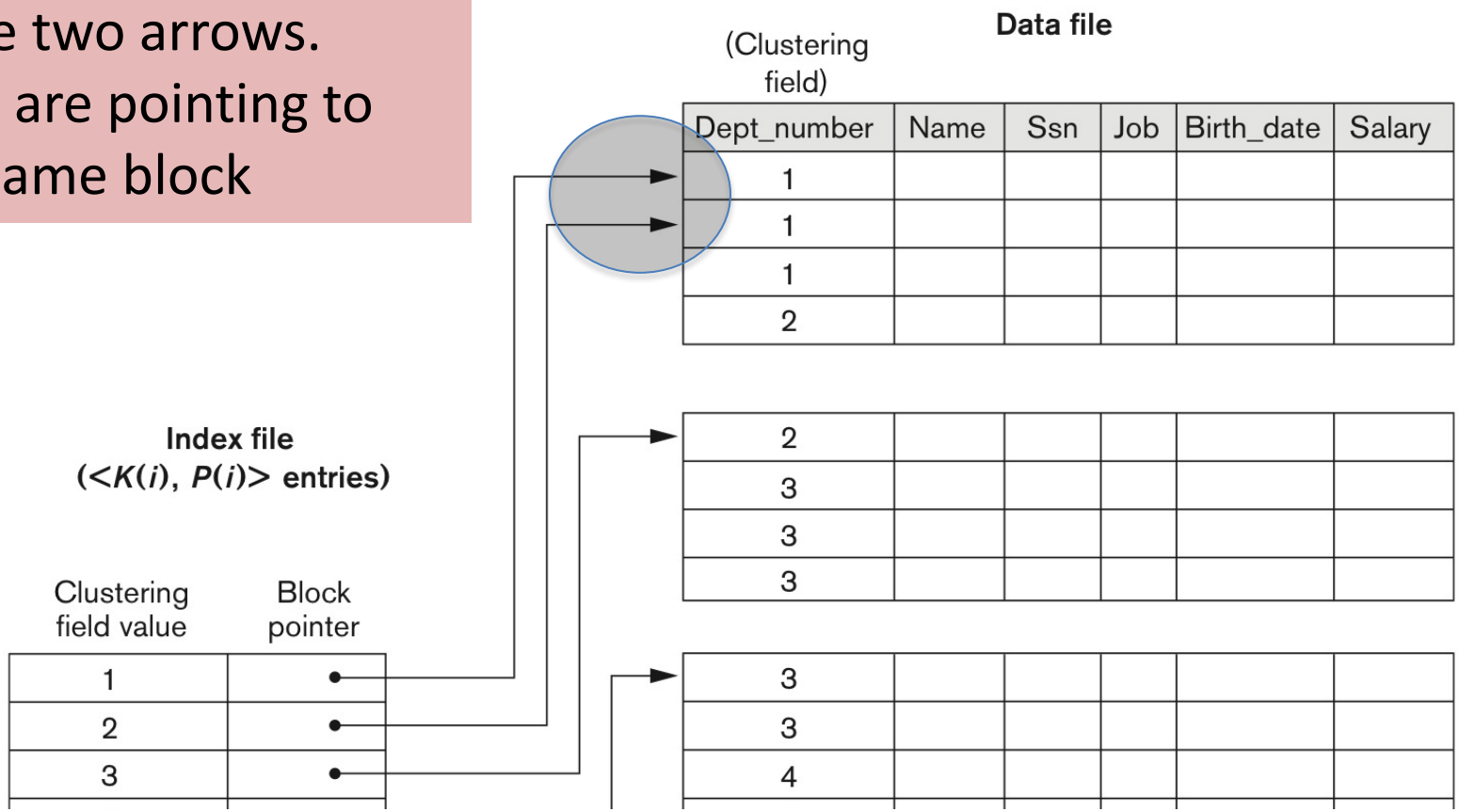
Clustering Indexes (Index for Sorted (on non-key) Files)



Clustering Indexes (Index for Sorted (on non-key) Files)

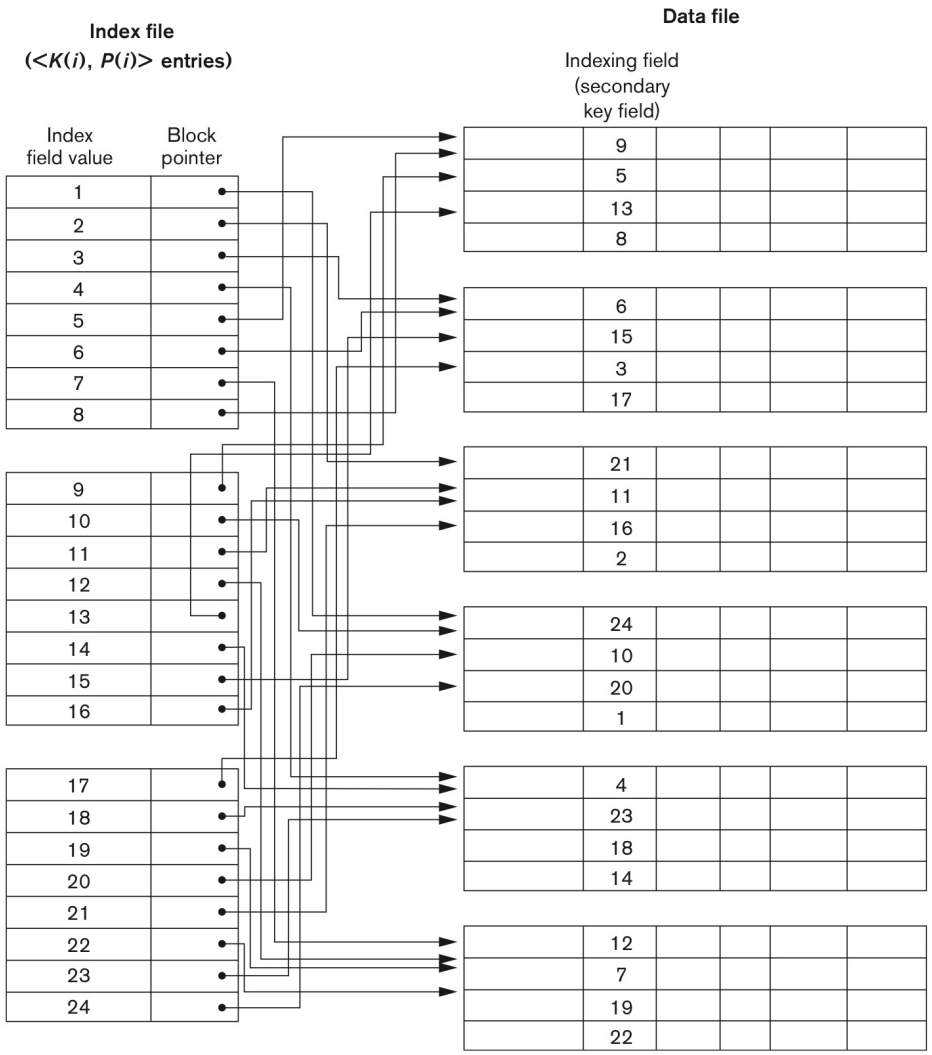
Points to the first block that contains the clustering field

Don't get confused by these two arrows. They are pointing to the same block



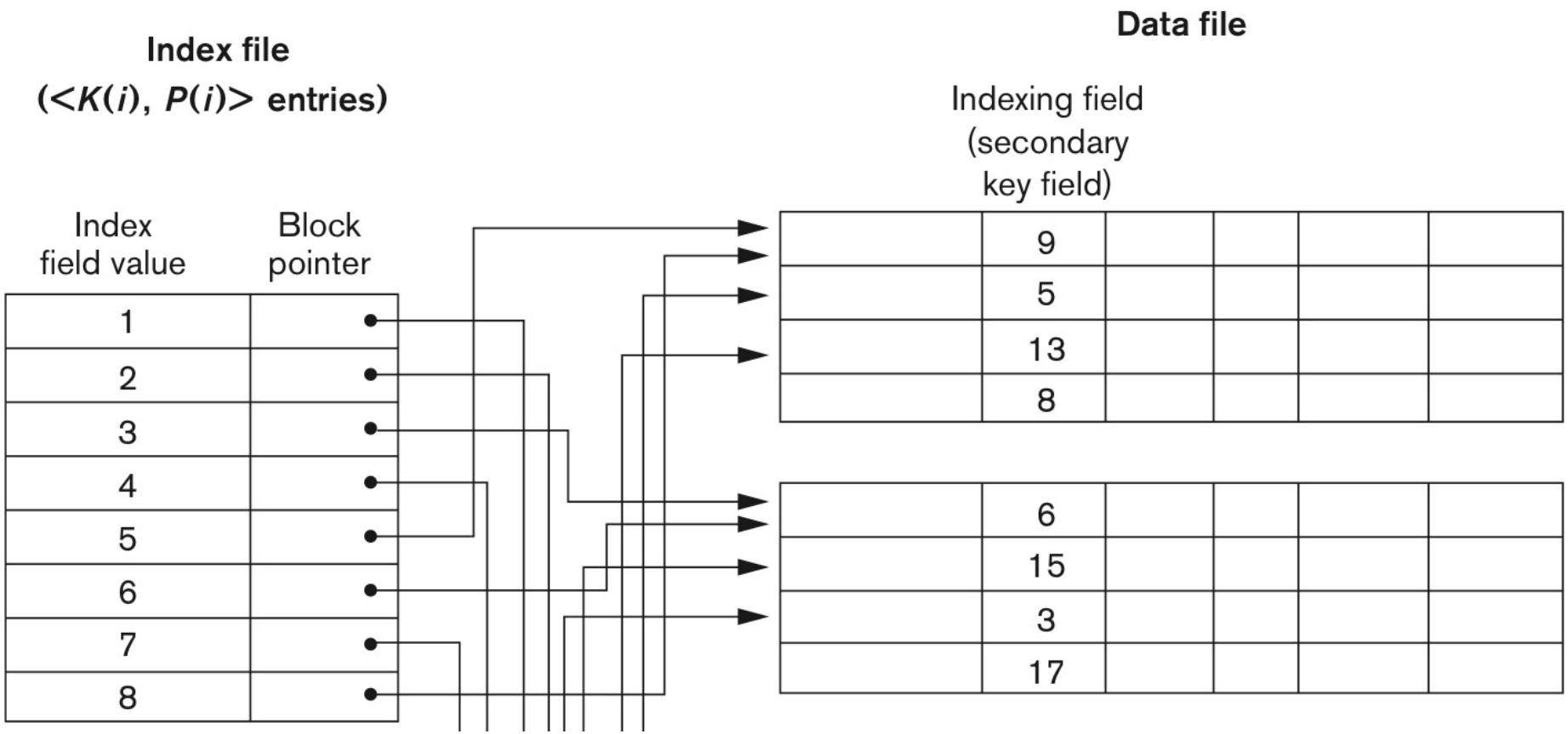
Secondary Indexes (on a key field)

- Secondary means of accessing a data file
- File records could be ordered, unordered, or hashed



Note: The data file is a heap file, i.e., not sorted

Secondary Indexes (on a key field)

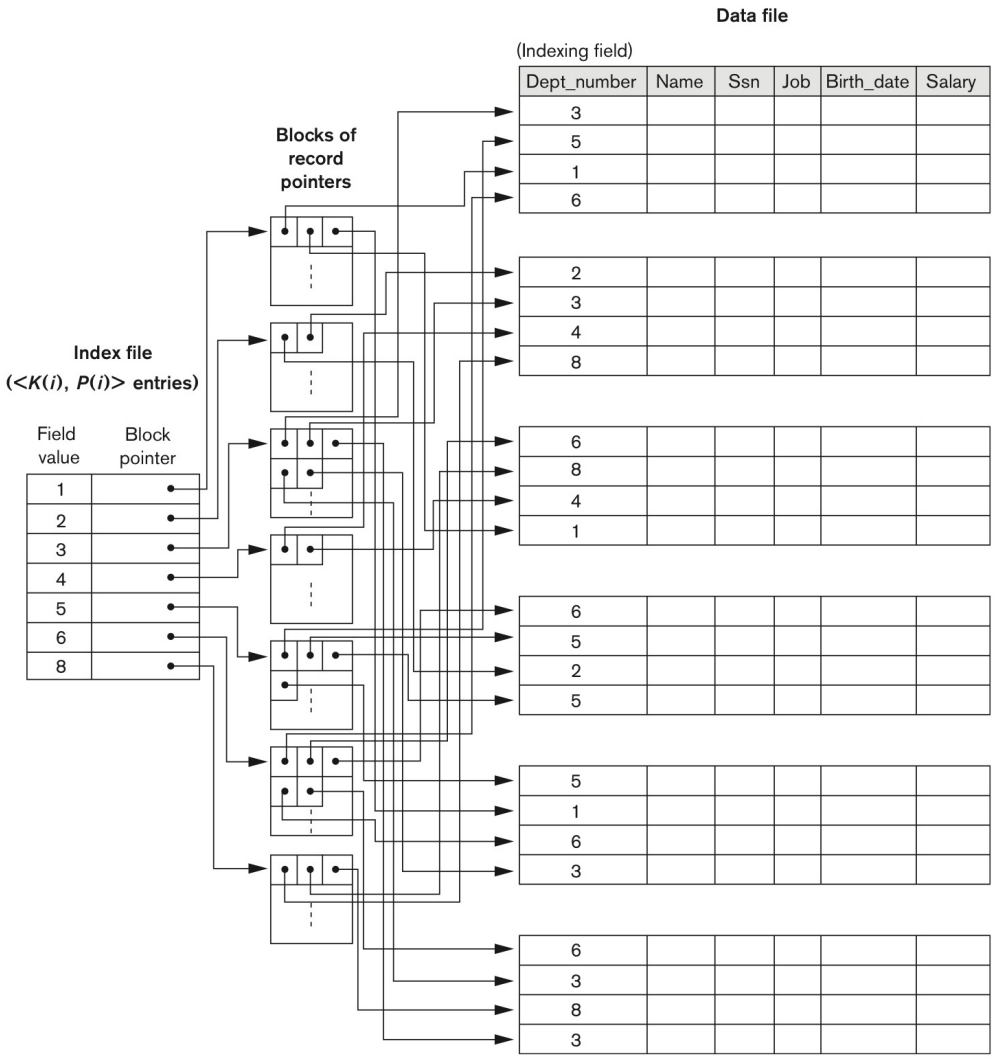


Note: The data file may be a heap file, i.e., not sorted

Secondary Indexes (on a non-key field)

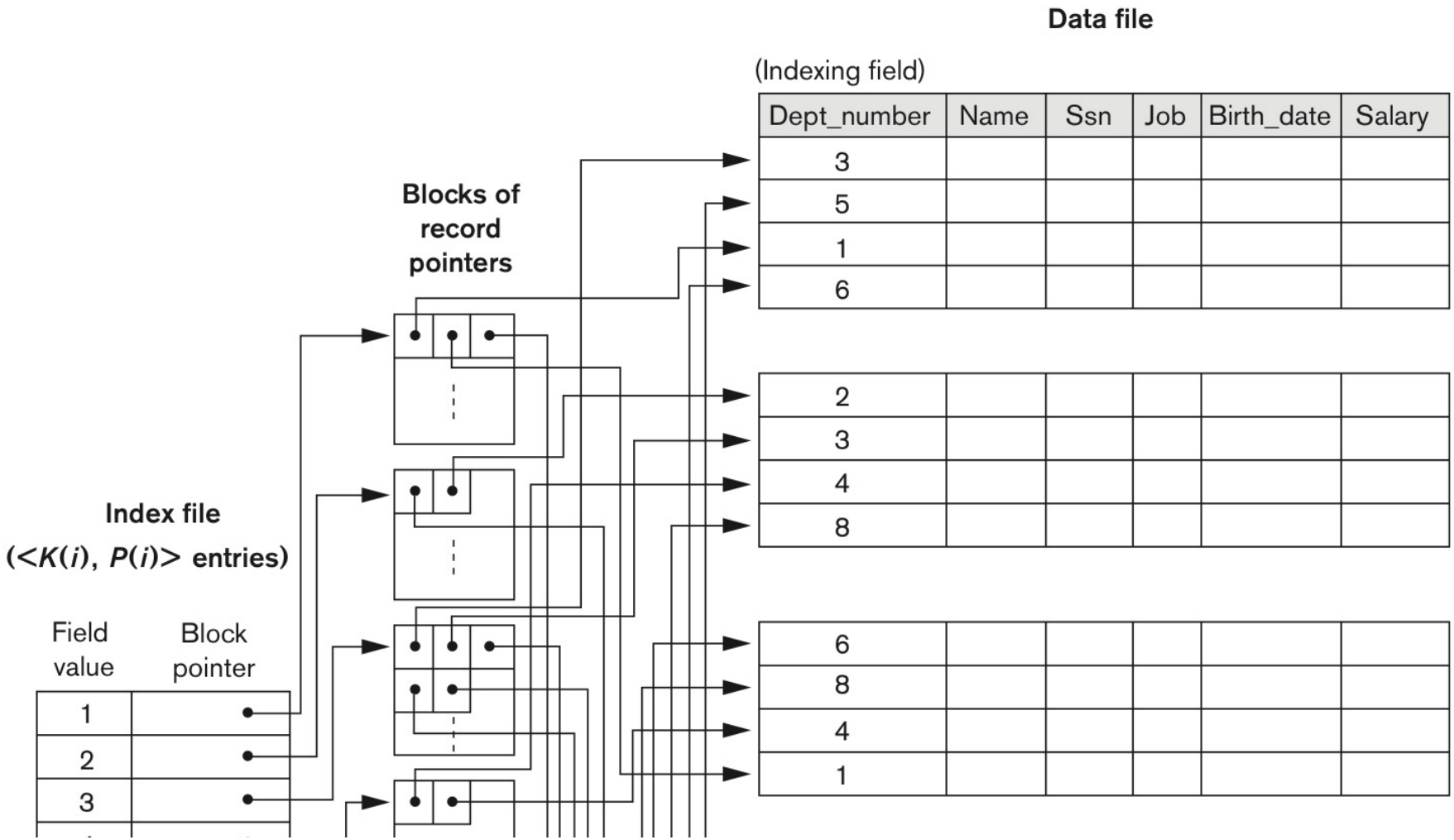
Extra level of indirection

- Provides logical ordering
  - Though records are not physically ordered



Secondary Indexes (on a non-key field)

Extra level of indirection



High-level Categories of Index Types

- Multilevel Indexes
  - Very good for range queries, sorted data
  - Some old databases only implemented B-Trees
  - *We will mostly look at a variant called **B+ Trees***

- Hash Tables
  - Very good for searching

Real difference between structures: costs of ops *determines which index you pick and why*

MULTILEVEL INDEXES

What you will learn about in this section

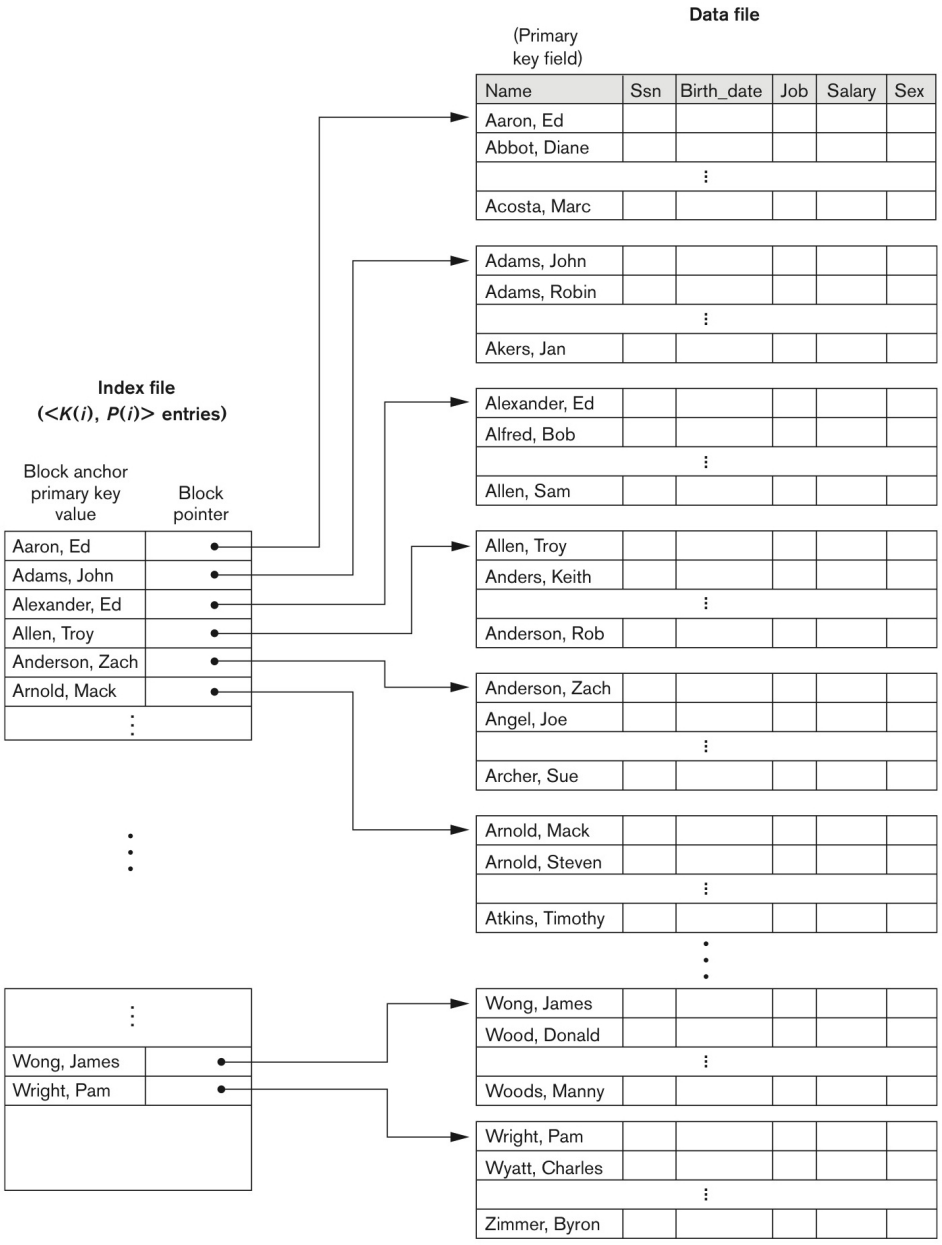
1. ISAM

2. B+ Tree

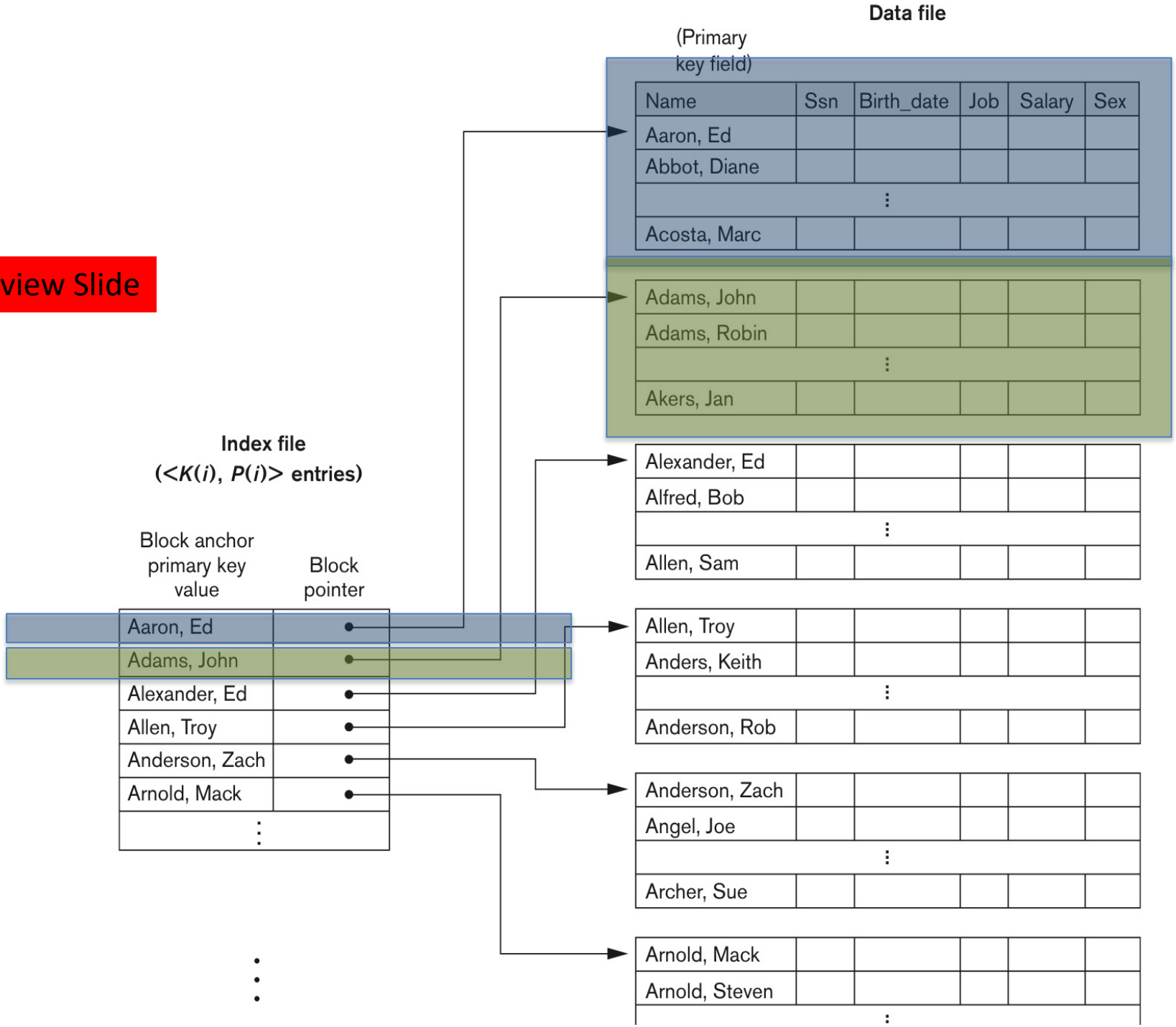
1. ISAM

Primary Indexes: Index for Sorted (Ordered) Files

Review Slide

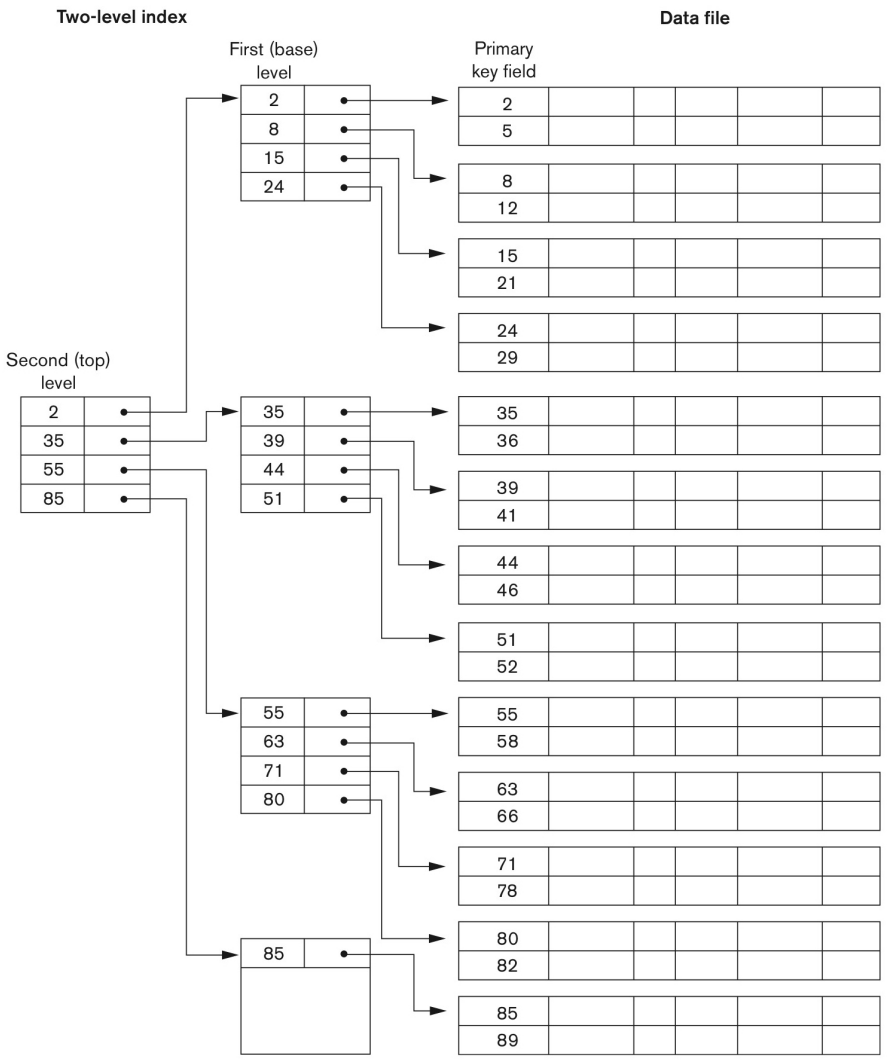


Review Slide

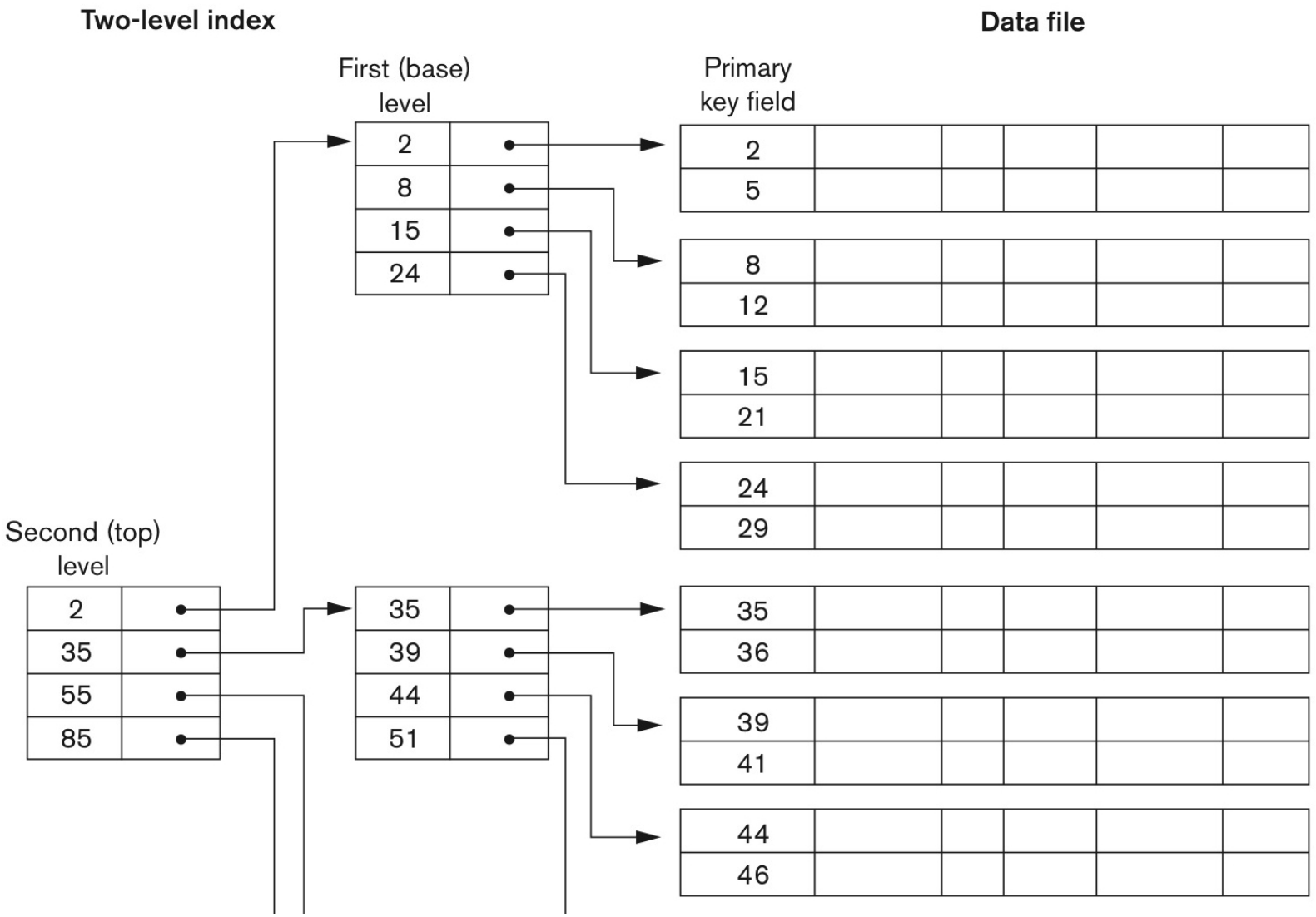


ISAM

- Indexed Sequential Access Method
  - For an index with b_i blocks
    - Earlier: $\log_2 b_i$ block access
    - Now: $\log_{fo} b_i$ block access
    - $(fo = fanout)$



ISAM


Two-level index
Data file
First (base) level
Primary key field
Second (top) level
2
8
15
24
2
5
8
12
15
21
24
29
2
35
55
85
35
39
44
51
35
36
39
41
44
46

1. B+ TREES

What you will learn about in this section

1. B+ Trees: Basics
2. B+ Trees: Design & Cost
3. Clustered Indexes

B+ Trees

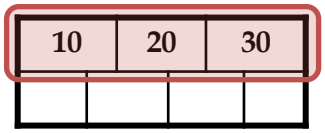
- Search trees
  - B does not mean binary!
- Idea in B Trees:
  - make 1 node = 1 physical page
  - Balanced, height adjusted tree (not the B either)
- Idea in B+ Trees:
  - Make leaves into a linked list (for range queries)

Ed McCreight answered a question on B-tree's name in 2013:

> Bayer and I were in a lunchtime where we get to think [of] a name. And ... B is, you know ... We were working for Boeing at the time, we couldn't use the name without talking to lawyers. So, there is a B. [The B-tree] has to do with balance, another B. Bayer was the senior author, who [was] several years older than I am and had many more publications than I did. So there is another B. And so, at the lunch table we never did resolve whether there was one of those that made more sense than the rest. What really lives to say is: the more you think about what the B in B-trees means, the better you understand B-trees."[5]

https://en.wikipedia.org/wiki/B-tree

B+ Tree Basics

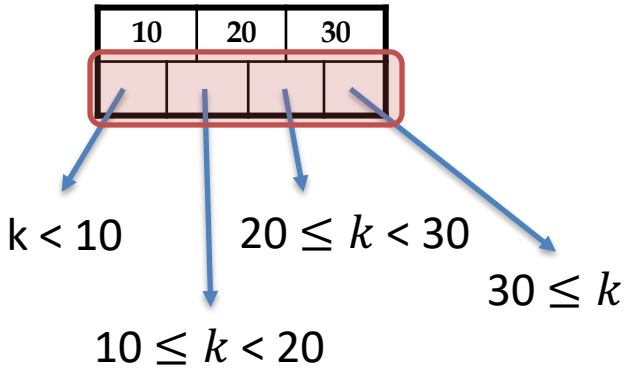


Parameter d = degree
The minimum number of key an interior node can have

Each *non-leaf ("interior")* ***node*** has ≥ d and ≤ 2d ***keys****

except for root node, which can have between **1 and 2d keys*

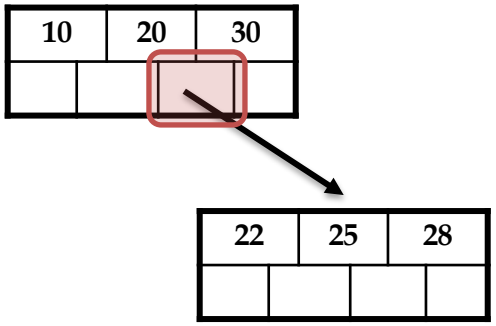
B+ Tree Basics



The *n* keys in a node define *n+1* ranges

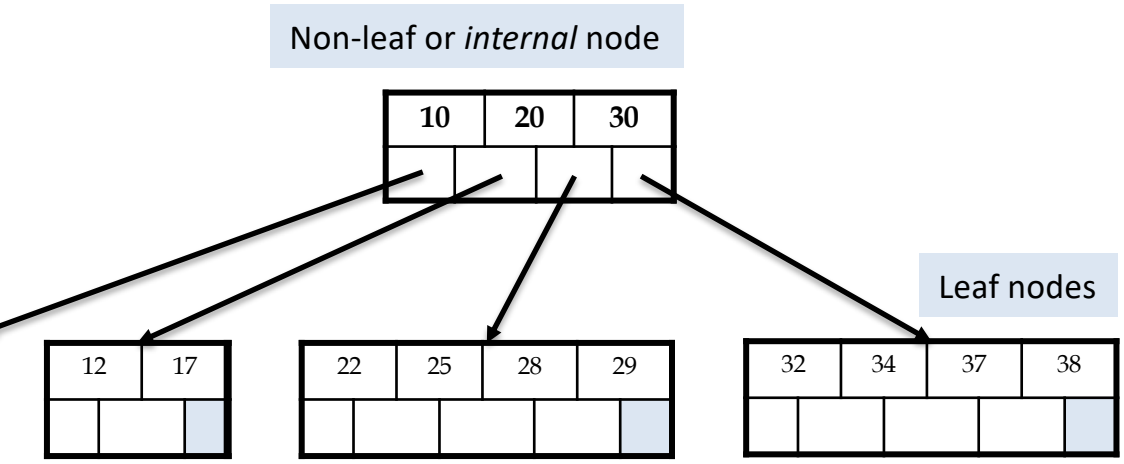
B+ Tree Basics

Non-leaf or *internal* node



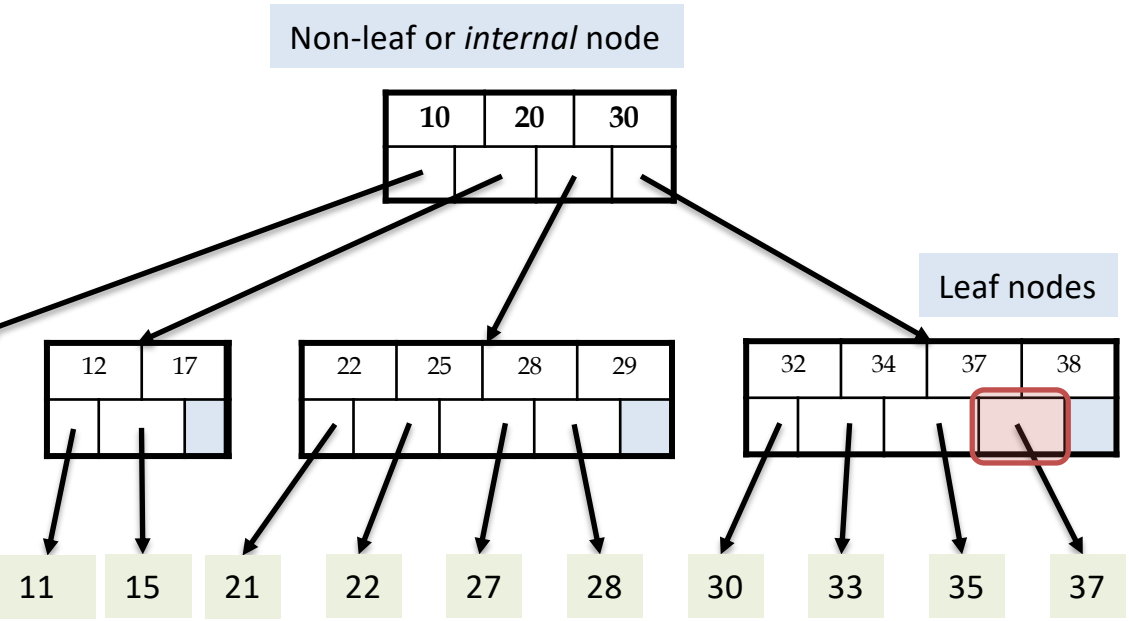
For each range, in a *non-leaf* node, there is a **pointer** to another node with keys in that range

B+ Tree Basics



Leaf nodes also have between *d* and *2d* keys, and are different in that:

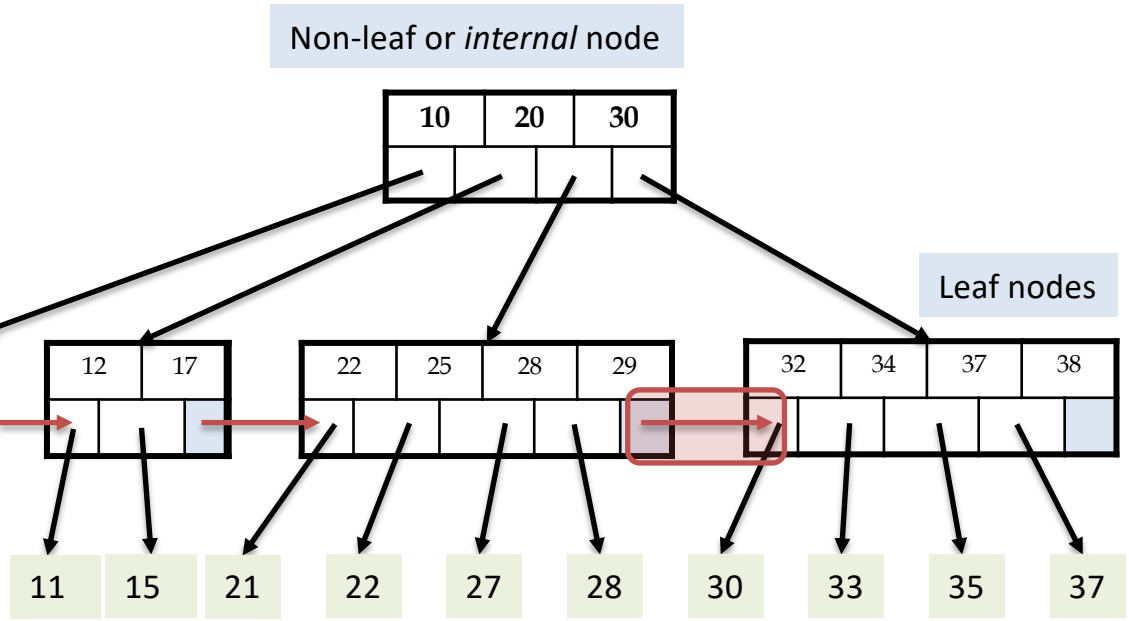
B+ Tree Basics



Leaf nodes also have between *d* and *2d* keys, and are different in that:

Their key slots contain pointers to data records

B+ Tree Basics

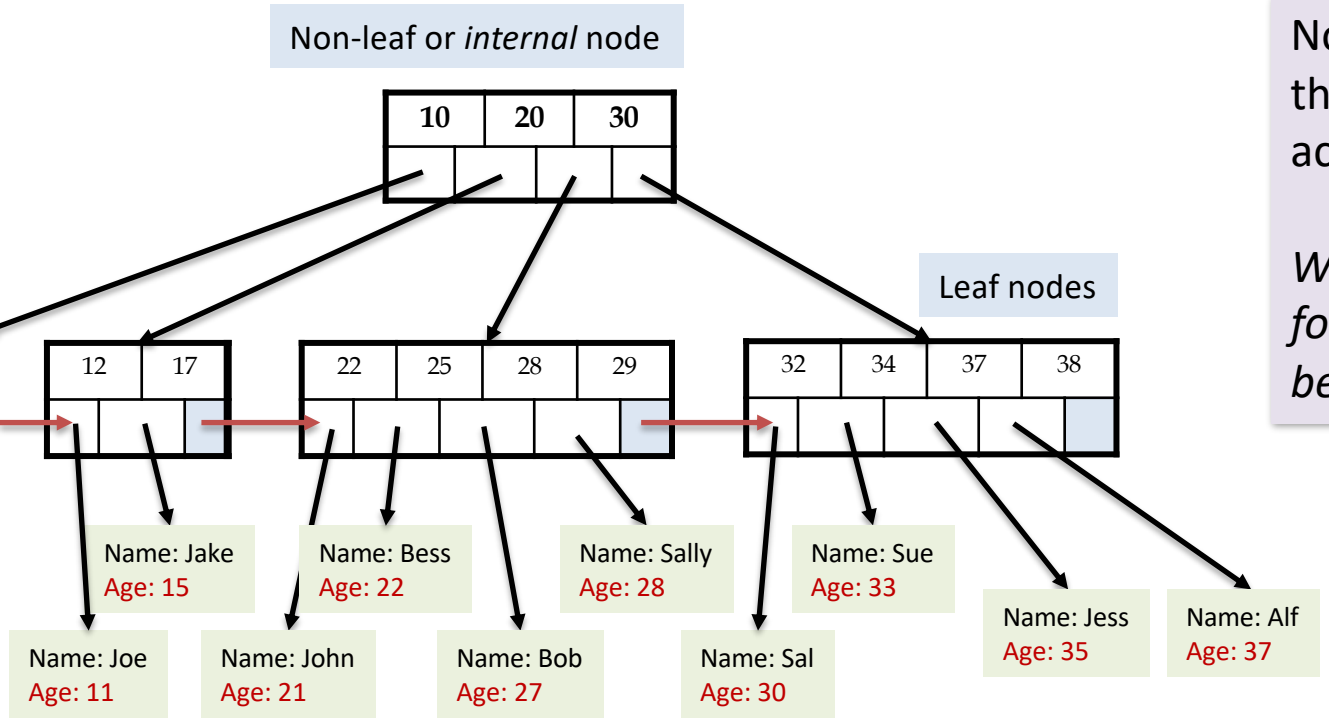


Leaf nodes also have between *d* and *2d* keys, and are different in that:

Their key slots contain pointers to data records

They contain a pointer to the next leaf node as well, ***for faster sequential traversal***

B+ Tree Basics



Note that the pointers at the leaf level will be to the actual data records (rows).

We might truncate these for simpler display (as before)…

Some finer points of B+ Trees

Searching a B+ Tree

- For exact key values:
  - Start at the root
  - Proceed down, to the leaf
- For range queries:
  - As above
  - *Then sequential traversal*

```
SELECT name
FROM   people
WHERE  age = 25
```

```
SELECT name
FROM   people
WHERE  20 <= age
  AND  age <= 30
```

B+ Tree Exact Search Animation

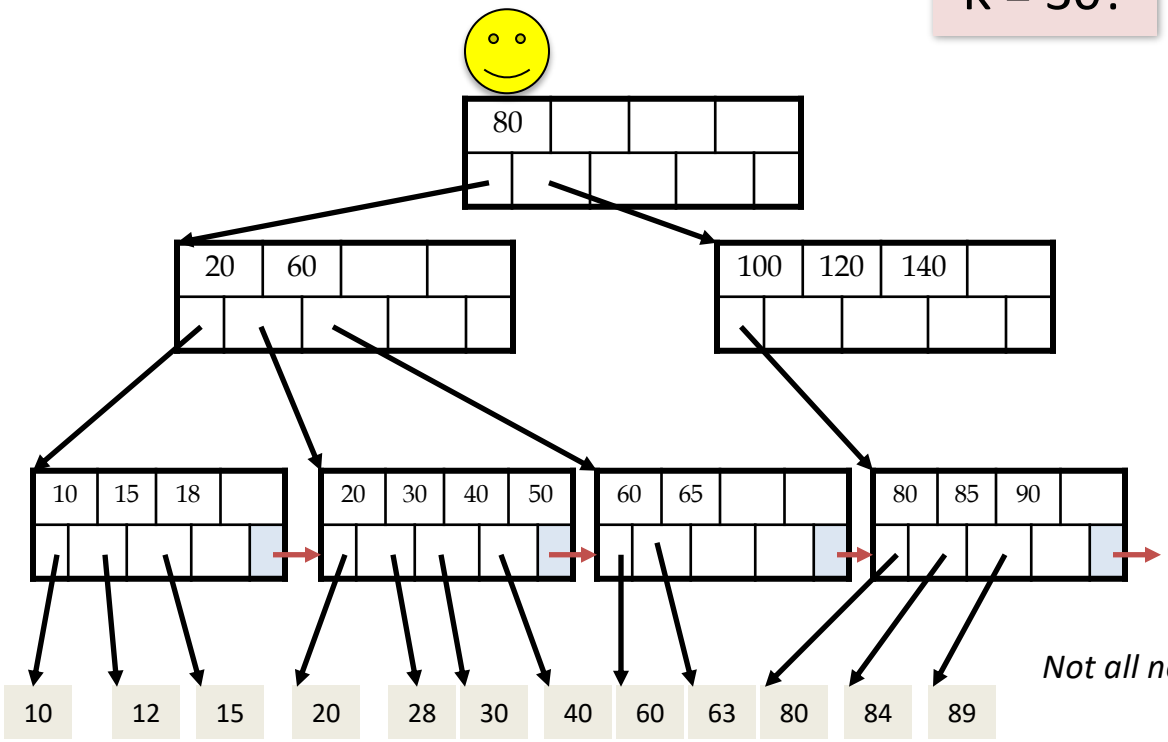
K = 30?

30 < 80

30 in [20,60)

30 in [30,40)

To the data!



Not all nodes pictured

B+ Tree Range Search Animation

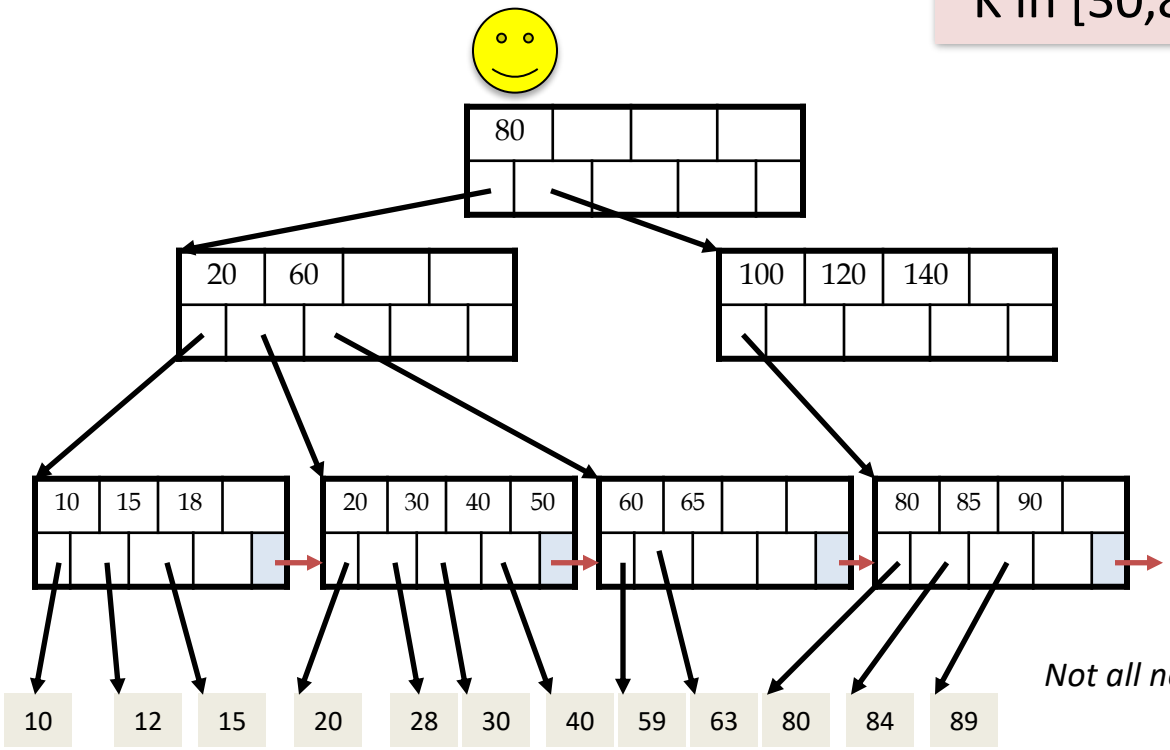
K in [30,85]?

30 < 80

30 in [20,60)

30 in [30,40)

To the data!



Not all nodes pictured

B+ Tree Design

- How large is ***d***?

- Example:
  - Key size = 4 bytes
  - Pointer size = 8 bytes
  - Block size = 4096 bytes

- We want each *node* to fit on a single *block/page*
  - 2d x 4 + (2d+1) x 8 <= 4096 → ***d <= 170***

B+ Tree: High Fanout = Smaller & Lower IO

- As compared to e.g. binary search trees, B+ Trees have **high *fanout*** (***between d+1 and 2d+1***)

- This means that the **depth of the tree is small** → getting to any element requires very few IO operations!
  - Also can often store most or all of the B+ Tree in main memory!

The **fanout** is defined as the number of pointers to child nodes coming out of a node

Note that fanout is dynamic- we'll often assume it's constant just to come up with approximate eqns!

Simple Cost Model for Search

- Let:
  - ***f*** = fanout, which is **in [d+1, 2d+1]** ***(we'll assume it's constant for our cost model…)***
  - ***N*** = the total number of *pages* we need to index
  - ***F*** = fill-factor (usually ~= 2/3)

- Our B+ Tree needs to have room to index ***N / F*** pages!
  - We have the fill factor in order to leave some open slots for faster insertions

- What height (*h*) does our B+ Tree need to be?
  - h=1 → Just the root node- room to index f pages
  - h=2 → f leaf nodes- room to index f^2 pages
  - h=3 → f^2 leaf nodes- room to index f^3 pages
  - …
  - h → f^{h-1} leaf nodes- room to index f^h pages!

→ We need a B+ Tree of height h = $\left\lceil \log_f \frac{N}{F} \right\rceil$

Fast Insertions & Self-Balancing

- **Same cost as exact search**
- ***Self-balancing:*** B+ Tree remains **balanced** (with respect to height) even after insert

B+ Trees also (relatively) fast for single insertions!
However, can become bottleneck if many insertions (if fill-factor slack is used up…)

Example

- Calculate the order p (order is same as fan-out) of a B+-tree.
- Suppose that the search key field is V = 9 bytes long, the block size is B = 512 bytes, a record pointer is Pr = 7 bytes, and a block pointer is P = 6 bytes.

Answer

- An internal node of the B+-tree can have up to p tree pointers and p – 1 search field values; these must fit into a single block.

- Hence, we have:

$(p * P) + ((p – 1) * V) \leq B$

$(P * 6) + ((P – 1) * 9) \leq 512$

$(15 * p) \leq 521$

So: p = 34

Order of the leaf nodes

The leaf nodes of the B+-tree will have the same number of values and pointers, except that the pointers are data pointers and a next pointer. Hence, the

order pleaf for the leaf nodes can be calculated as follows:

$(pleaf^* (Pr + V)) + P \leq B$

$(pleaf^* (7 + 9)) + 6 \leq 512$

$(16 * pleaf) \leq 506$

It follows that each leaf node can hold up to pleaf= 31 key value/data pointer combinations,

assuming that the data pointers are record pointers.

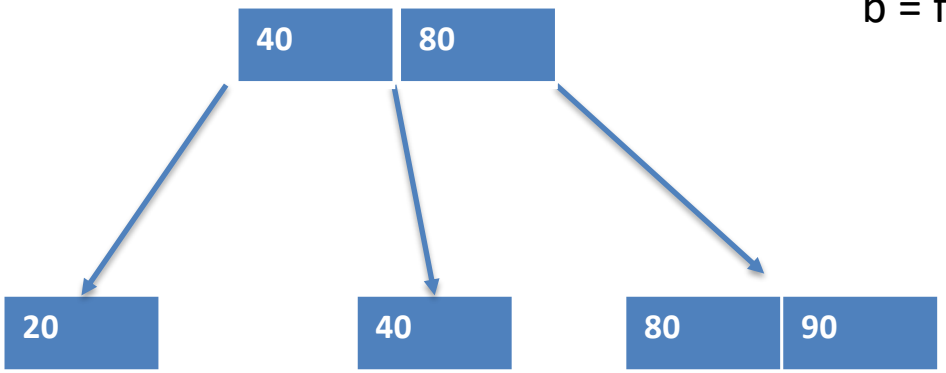
Insertion

- Perform a search to determine what bucket the new record should go into.
- If the bucket is not full (at most (b-1) entries after the insertion), add the record.
- Otherwise, split the bucket.
  - Allocate new leaf and move half the bucket's elements to the new bucket.
  - Insert the new leaf's smallest key and address into the parent.
  - If the parent is full, split it too.
    - Add the middle key to the parent node.
  - Repeat until a parent is found that need not split.
- If the root splits, create a new root which has one key and two pointers. (That is, the value that gets pushed to the new root gets removed from the original node)

- Note: B-trees grow at the root and not at the leave

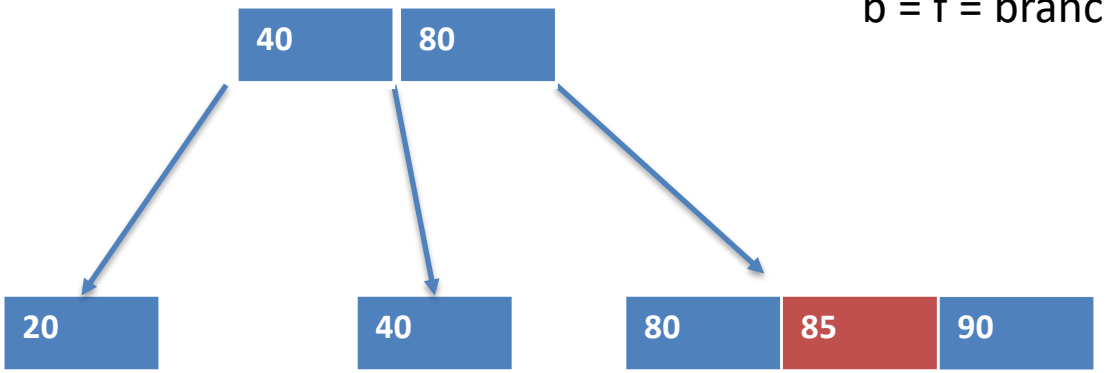
b = f = branching factor/ fan-out

Insertion (Insert 85)



b = f = branching factor/ fan-out = 3

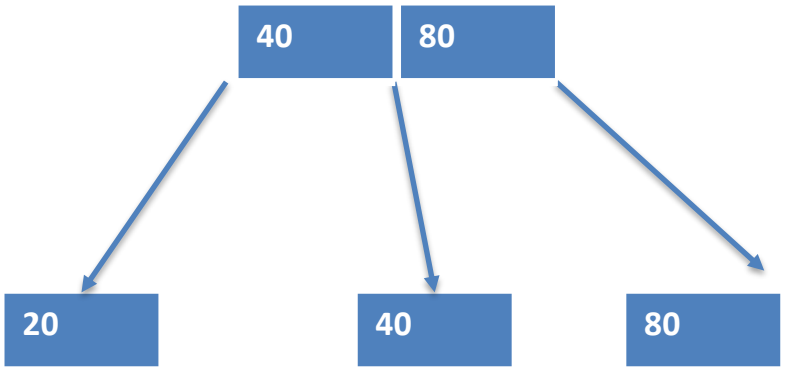
Insertion (Insert 85)



b = f = branching factor/ fan-out = 3

This is what we would like. But the maximum number of keys in any node is (3-1) = 2
So, split.

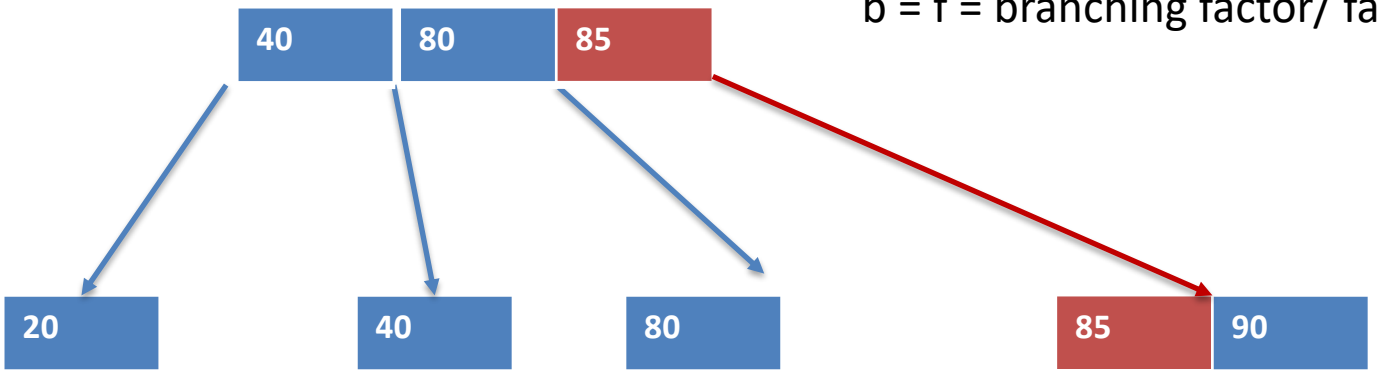
Insertion (Insert 85)



b = f = branching factor/ fan-out = 3

Allocate new leaf and move half the bucket's elements to the new bucket.

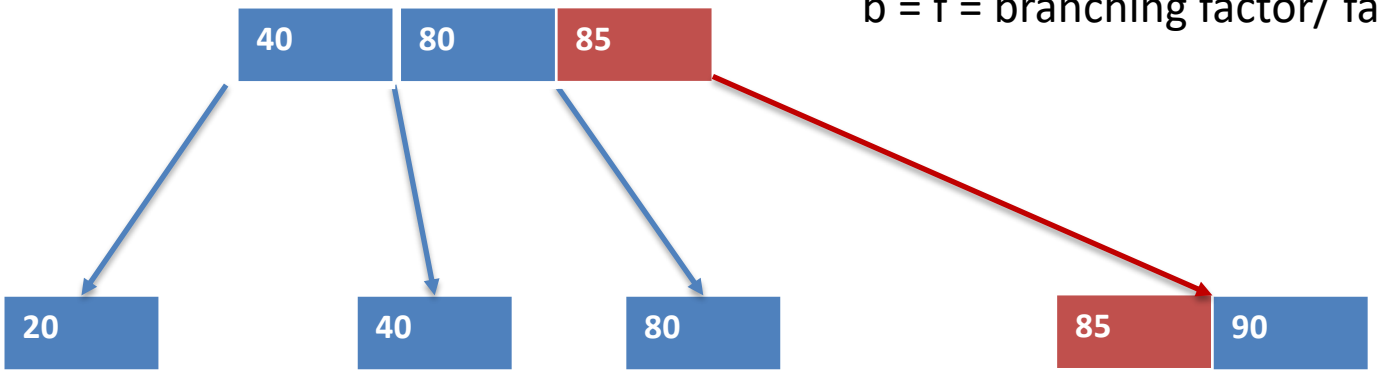
Insertion (Insert 85)



b = f = branching factor/ fan-out = 3

Insert the new leaf's smallest key and address into the parent.

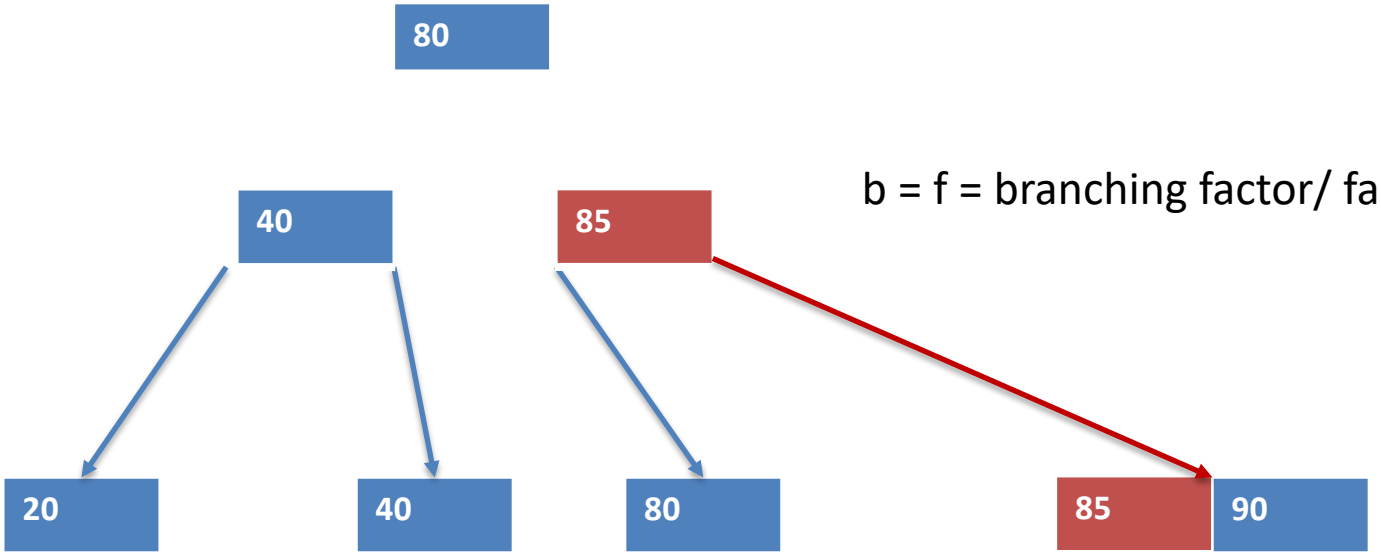
Insertion (Insert 85)



b = f = branching factor/ fan-out = 3

This is not allowed, as the parent is full. Need to split

Insertion (Insert 85)

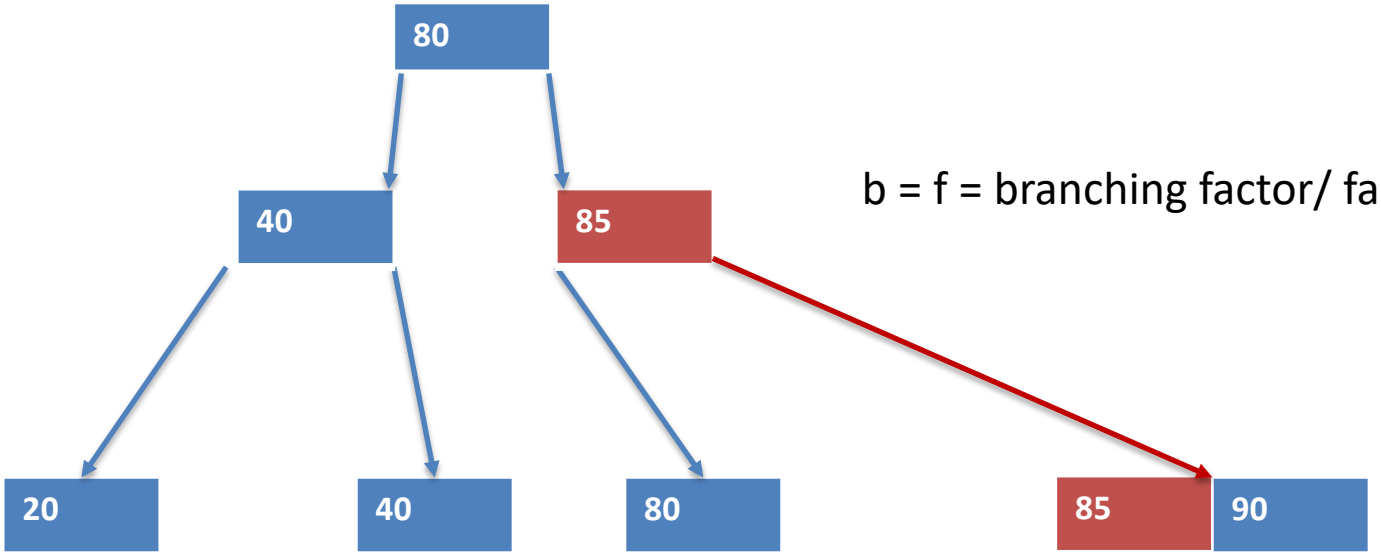


b = f = branching factor/ fan-out = 3

If the parent is full, split it too. Add the middle key to the parent node.

Repeat until a parent is found that needs no spliting

Insertion (Insert 85)



b = f = branching factor/ fan-out = 3

If the root splits, create a new root which has one key and two pointers.

(That is, the value that gets pushed to the new root gets removed from the original node)

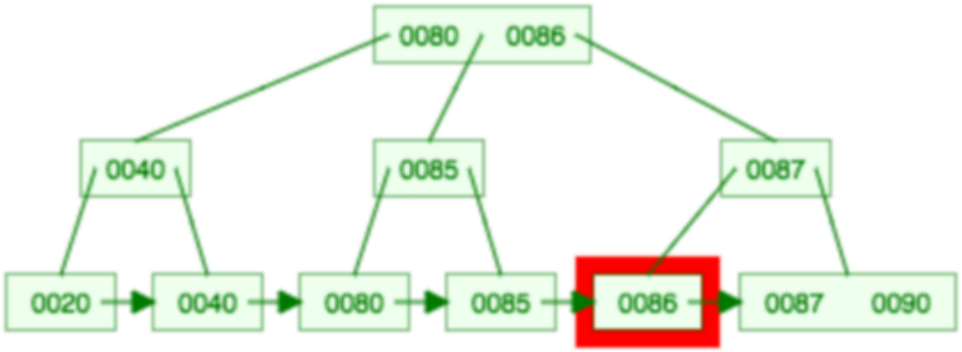
Deletion

- Start at root, find leaf *L* where entry belongs.
- Remove the entry.
  - If *L* is at least half-full, done!
  - If *L* has fewer entries than it should,
    - If sibling (adjacent node with same parent as *L*) is more than half-full, re-distribute, borrowing an entry from it.
    - Otherwise, sibling is exactly half-full, so we can merge *L* and sibling.
- If merge occurred, must delete entry (pointing to *L* or sibling) from parent of *L*.
- Merge could propagate to root, decreasing height.

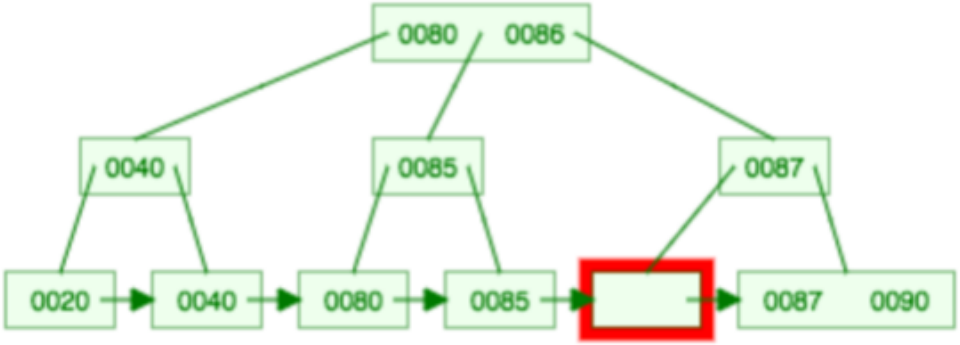
Not required for quiz or exam

- https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html
- The degree in this visualization is actually fan-out f or branching factor b.

- Find 86



- Delete it



- Stealing from right sibling (redistribute).
- Modify the parent node

- Finally,

Acknowledgement

- Some of the slides in this presentation are taken from the slides provided by the authors.
- Many of these slides are taken from cs145 course offered by Stanford University.