Index Structures for Files

政治大學
資訊科學系
沈錳坤

Recap: Physical Database Design

- Storage organization of large amount of data
- Data stored in disk is organized as files of records
- Process of physical DB design
  - among the options, choosing particular data organizations that best suit the given application requirements
- Primary file organization: determine
  - how the records of a file are physically placed on the disk
  - How the records can be accessed
- Secondary file organization: determine
  - How to allow efficient access to the records of a file based on alternate fields than those that have been used for primary file organization

Recap: Primary File Organization

- Heap files (unordered files, pipe files, sequential files)
  - Records are placed in the inserted order
  - New records are inserted at the end of the file
  - Insertion: efficient,
  - Deletion: inefficient, lazy deletion
- Sorted files
  - ordered files, sequential files
  - Binary search on ordering key field
  - Insertion, deletion cost: master file + overflow file(transaction file)
  - Modifying cost
  - Used when primary index is constructed
- Hashing files
  - Internal hashing
  - External hashing

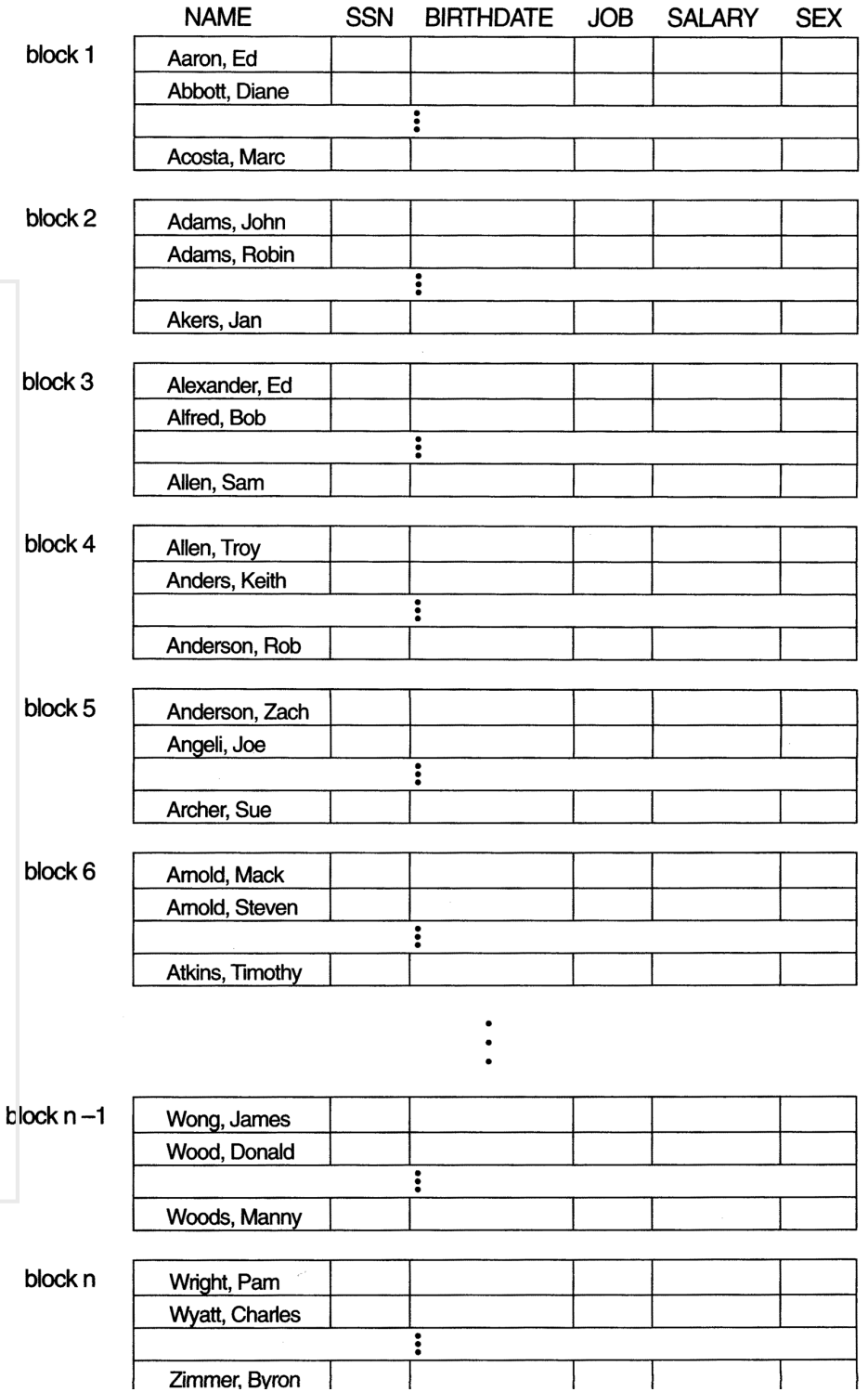
Recap: Sorted File

	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					

Recap:

Binary search on sorted file

$l = 1,\ u = \text{n}$
While $(l \leq u)$ do
    $i = (l + u)\ /2$
    read block i into buffer
    if $K <$ first record of block i
        $u = i - 1$
    else if $K >$ last record of block i
        $l = i + 1$
    else if K is in the buffer
        found
    else not found



Recap: Creating Indexes in SQL

- Statements can create and drop indexes on base relations
- One or more indexing attributes are specified for each index
- **CREATE INDEX** statement is used
- Each index is given an *index name*

I1: **CREATE INDEX** LNAME_INDEX
ON EMPLOYEE(LNAME);

Content

- Single-level ordered indexes
- Multilevel indexes
- Dynamic multilevel indexes using B-trees and B+-trees
- Multiple dimensional Indexes
- Other types of indexes
- Issues Concerning Indexes

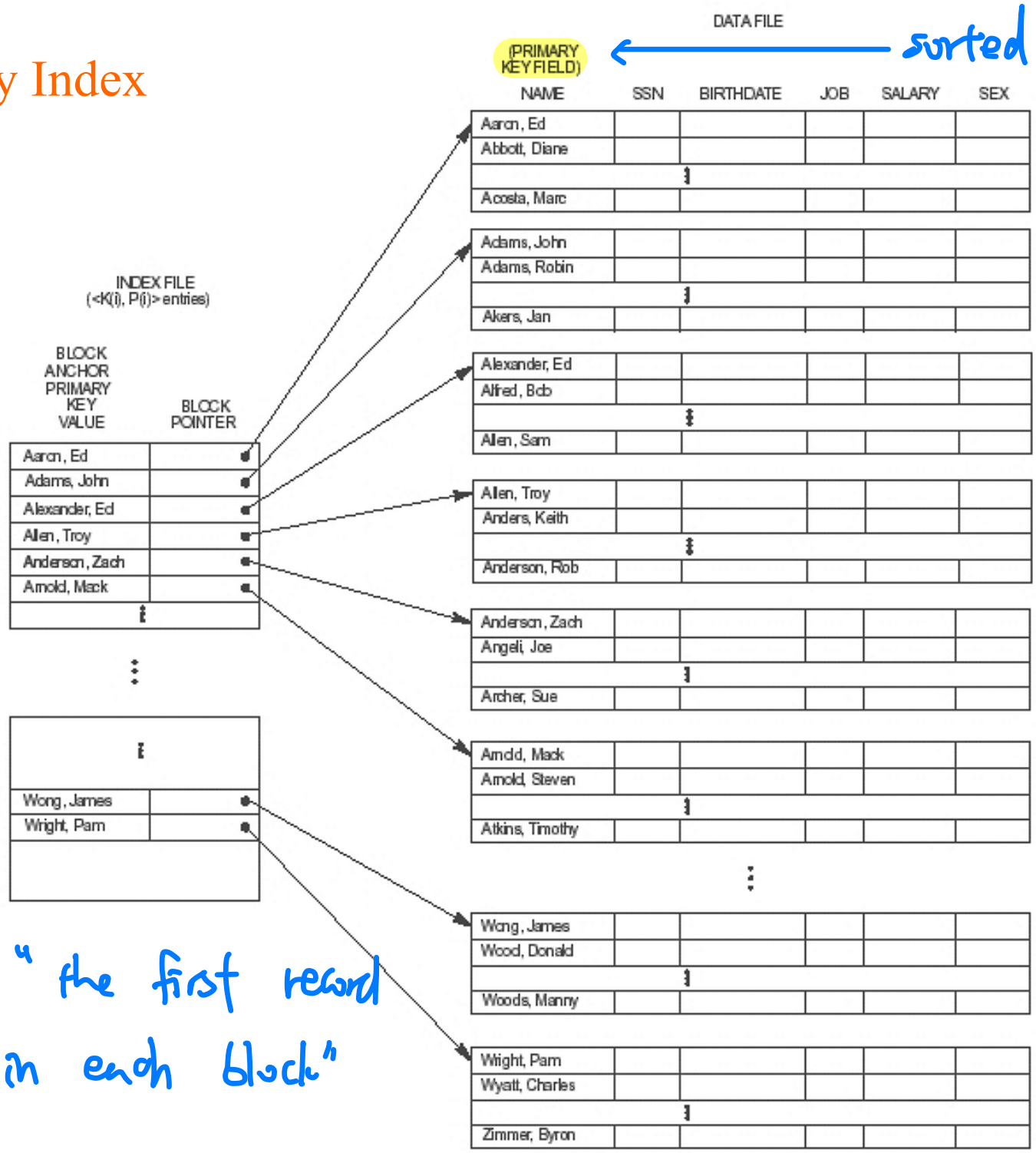
Index

- Index structure
  - To speed up searching process
  - defined on indexing fields
  - Stores each value of the index field

    along with a list of pointers
  - Pointer to all disk blocks

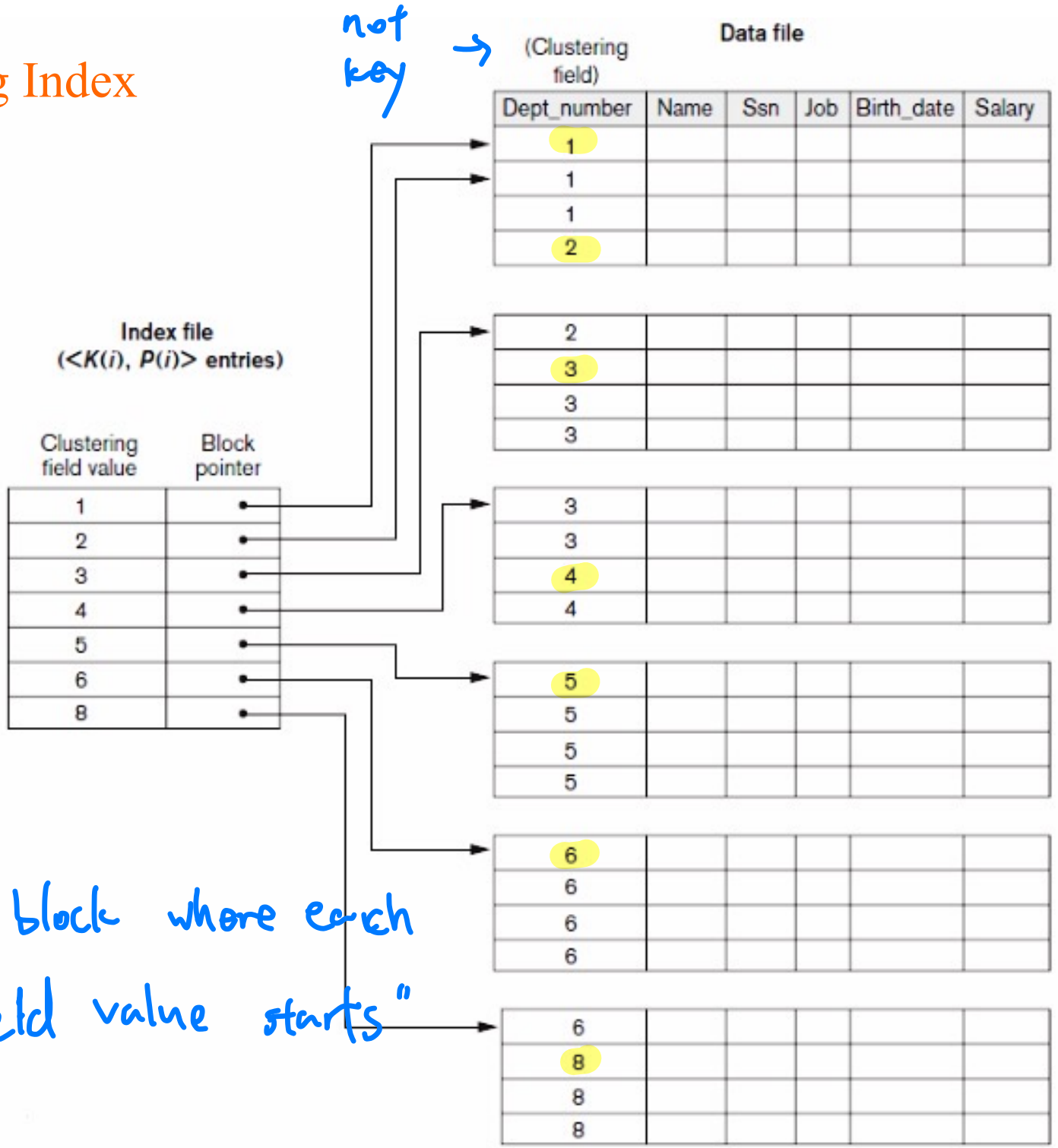
    containing records with that index field value
  - Value in index are ordered
  - Binary search on the index file which is much smaller than data file

Primary Index



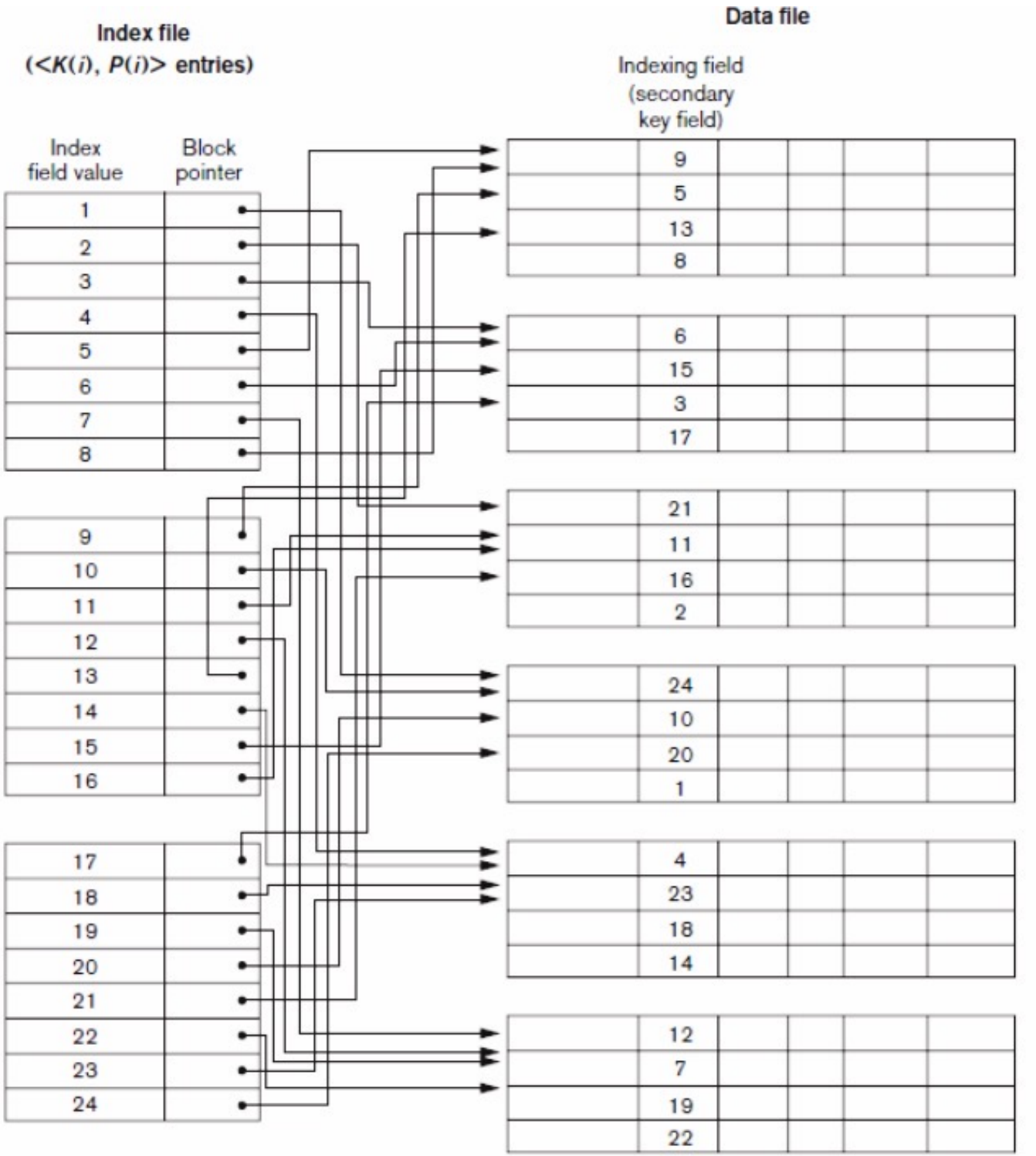
The key is "the first record key field in each block"

Clustering Index

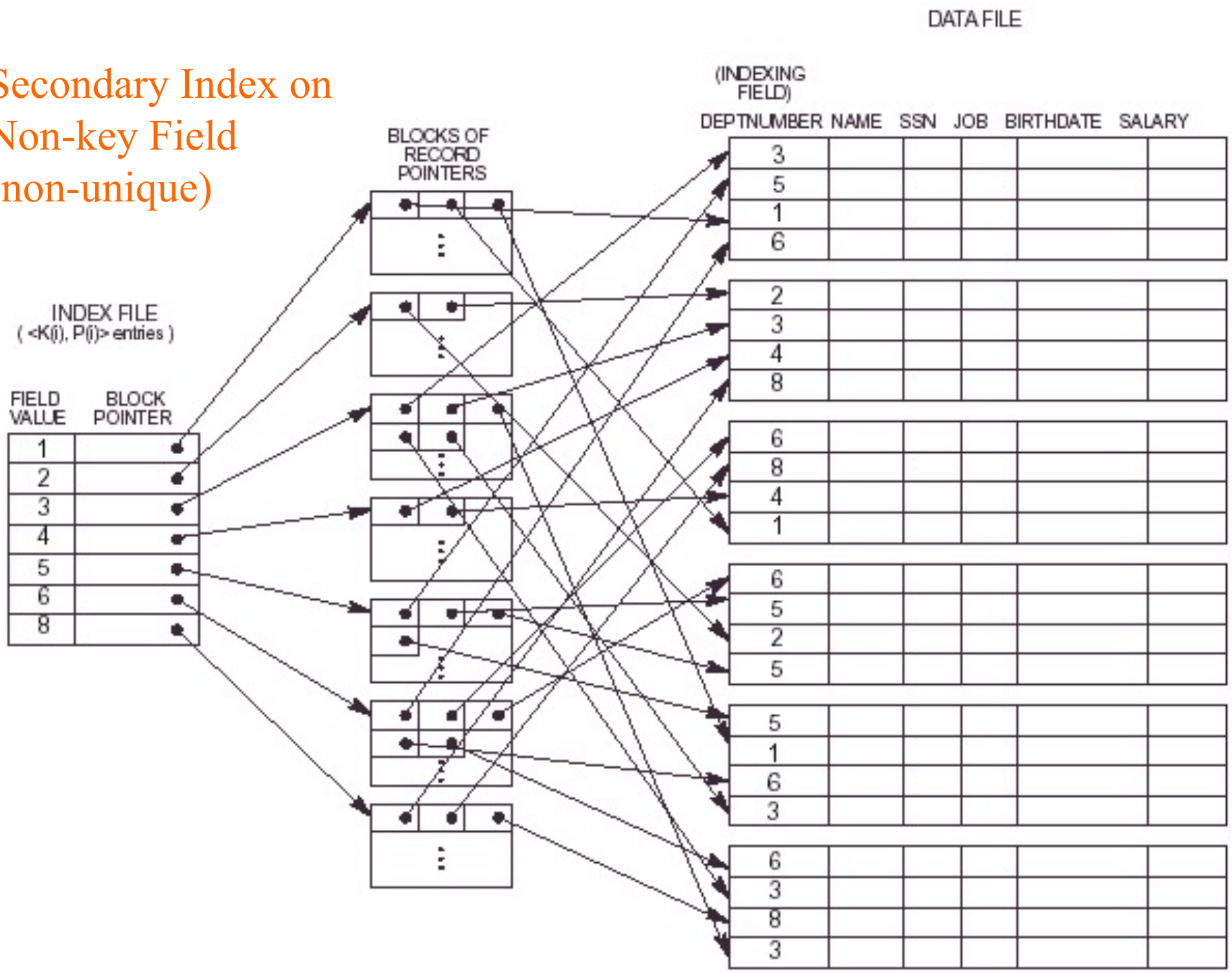


Points to "the block where each clustering field value starts"

Secondary Index on Key Field (unique)



Secondary Index on Non-key Field (non-unique)



Types of Ordered Indexes

- Primary, secondary, clustering
  - Primary index
    - Ordering field of a sorted file is key
  - Clustering index
    - Ordering field of a sorted file is non-key
  - Secondary index
    - Non-ordering field of a file

* A file can have at most one primary index or one clustering index, but not both.

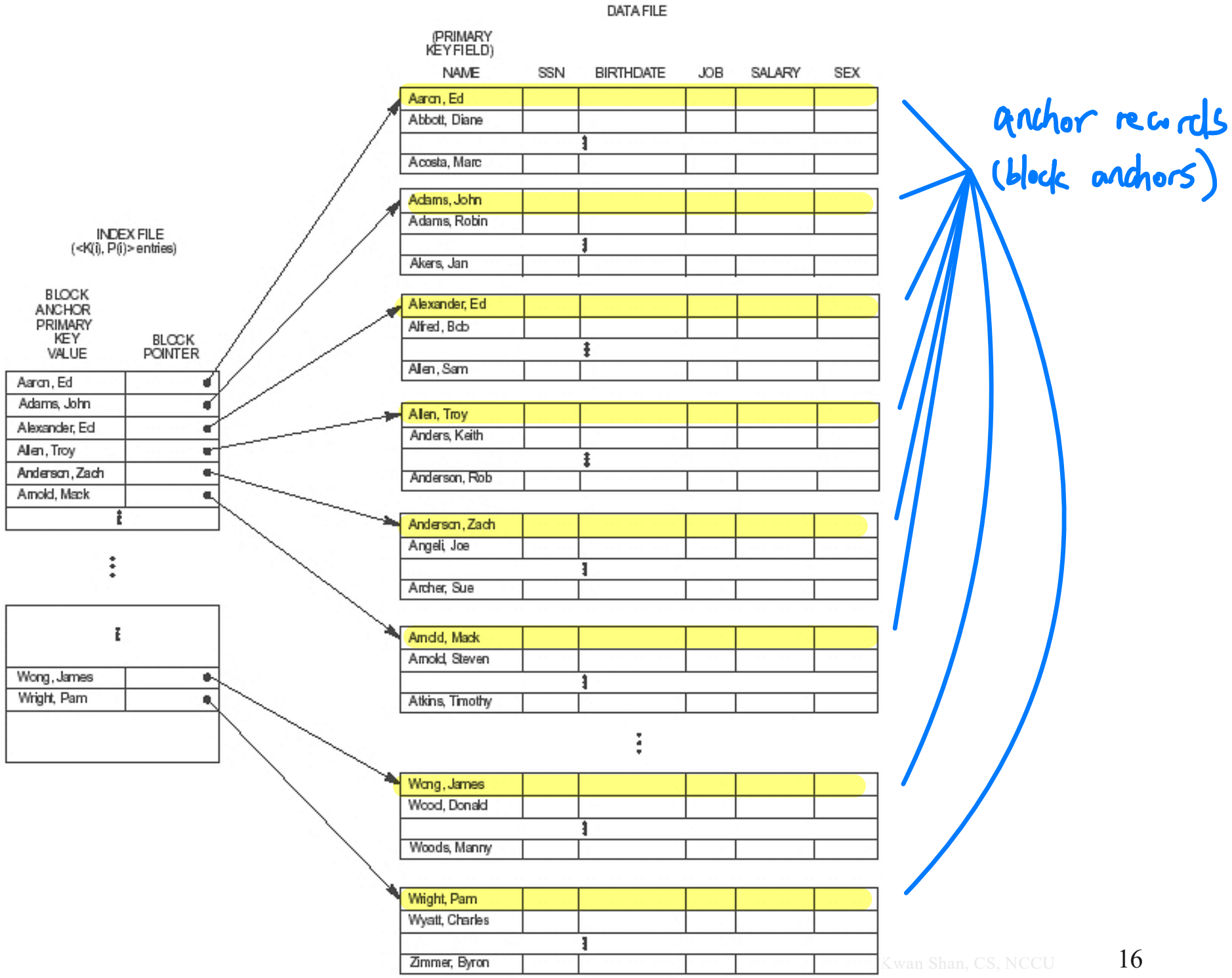
* A file can have several secondary indexes

Types of Ordered Indexes (cont.)

- Dense vs. sparse index
  - Dense index: there is an index entry for every search key value in the data file
  - Sparse index: index entries for only some of the search values

Primary Index

- Primary index
  - An ordered file whose records are of fixed length with two fields
  - There is one index entry for each block in the data file
  - Each index <ordering field, pointer to block>
  - #(entry) = #(blocks of ordered file)
  - Anchor record (block anchor) : first record in each block
  - Sparse index

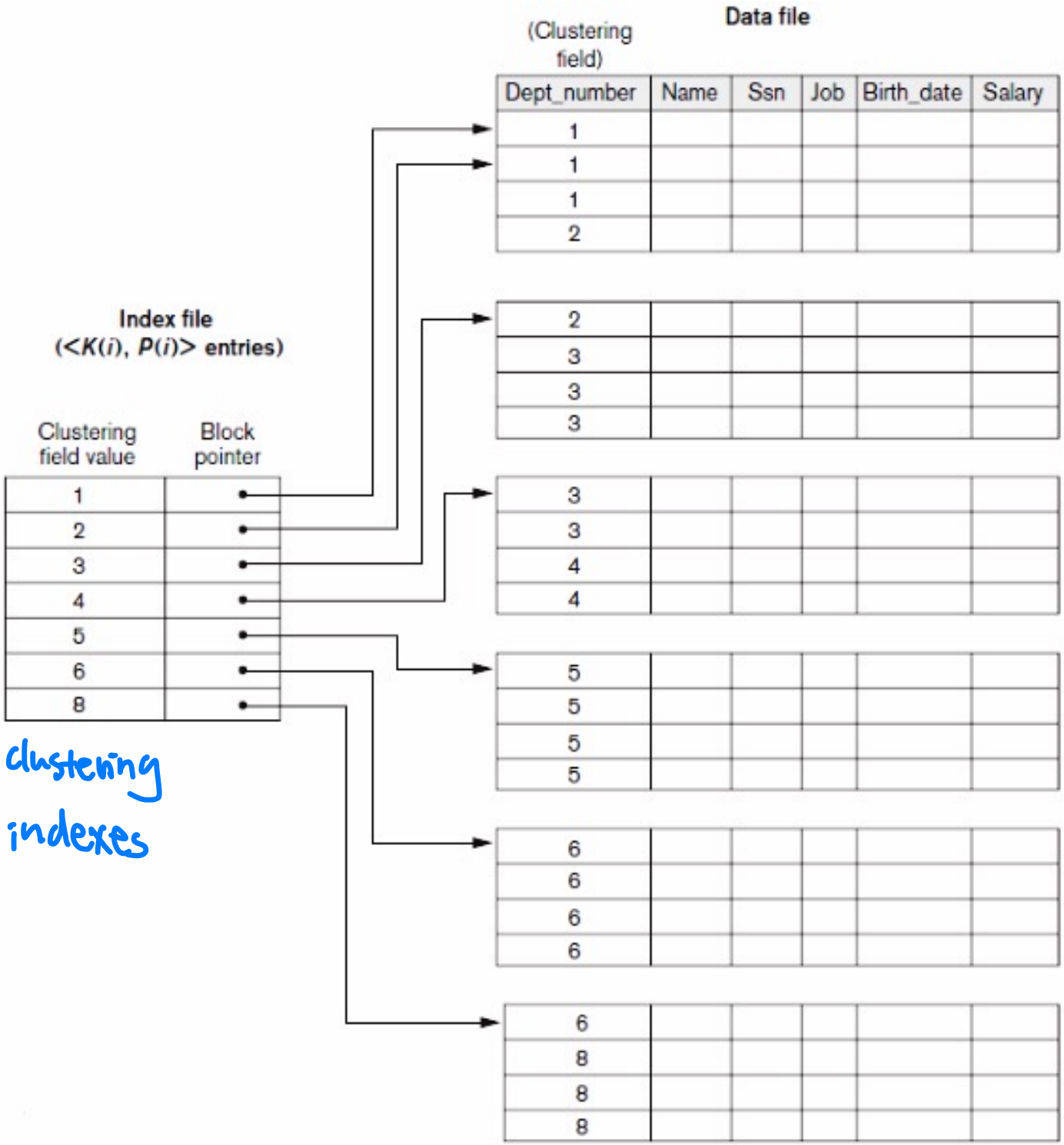

DATA FILE
(PRIMARY KEY FIELD)
NAME
SSN
BIRTHDATE
JOB
SALARY
SEX
Aaron, Ed
Abbott, Diane
Acosta, Marc
Adams, John
Adams, Robin
Akers, Jan
Alexander, Ed
Alfred, Bob
Allen, Sam
Allen, Troy
Anders, Keith
Anderson, Rob
Anderson, Zach
Angeli, Joe
Archer, Sue
Arnold, Mack
Arnold, Steven
Atkins, Timothy
Wong, James
Wood, Donald
Woods, Manny
Wright, Pam
Wyatt, Charles
Zimmer, Byron
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
Wong, James
Wright, Pam
anchor records
(block anchors)

Problem of Primary Index

- Insertion & deletion of records
  - Index structure is changed
  - Moving records will change the anchor records of some blocks
  - Solution
    - Insertion
      - Using an unordered overflow file
      - Using a linked list of overflow records for each block in the data file
    - Deletion: deletion markers (lazy deletion)

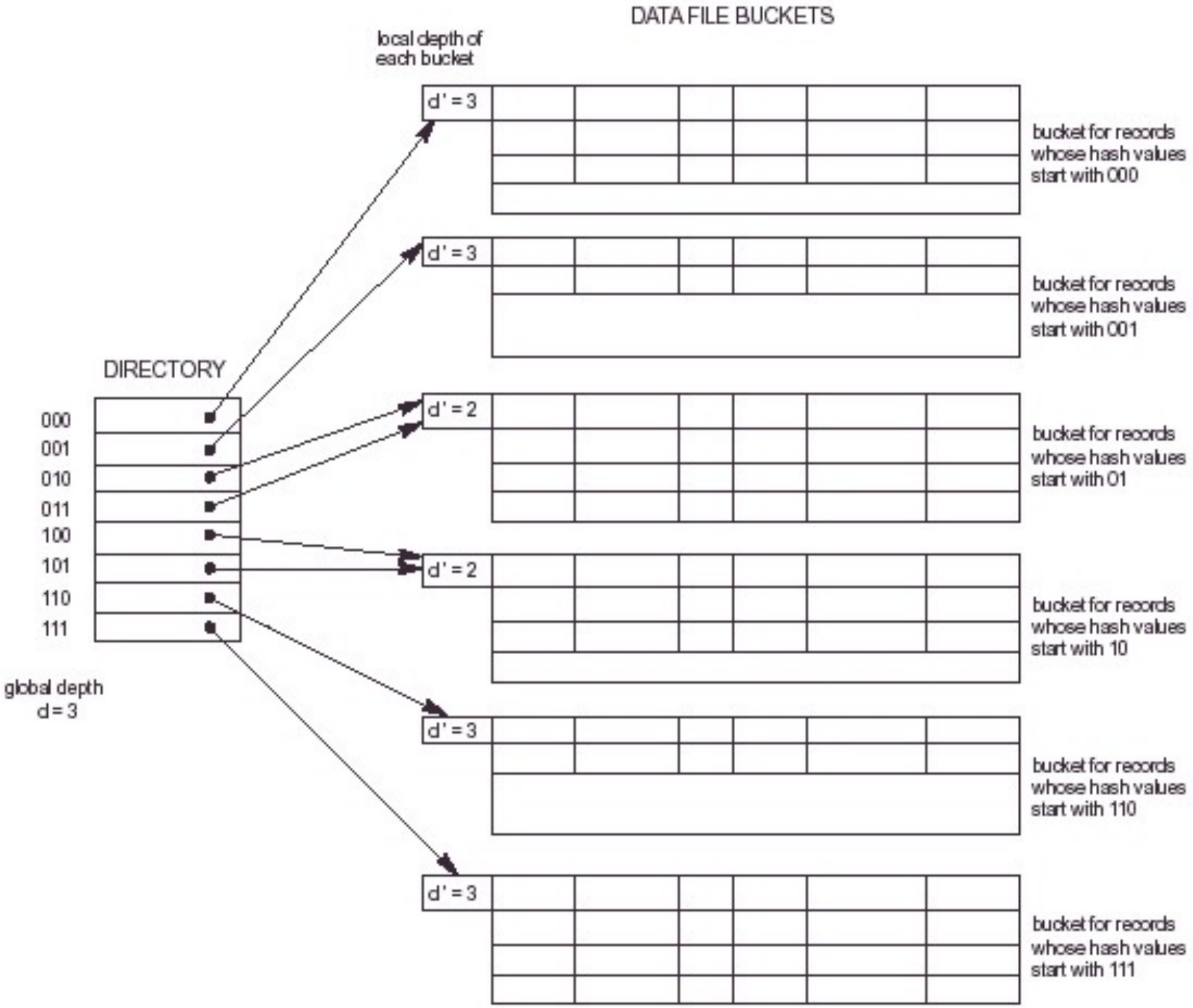
Clustering Indexes

- Clustering field
  - If a sorted file are physically ordered on a non-key field
  - Non-key field is not distinct
- Clustering index
  - Index for the clustering field
  - There is one entry for each distinct value of clustering field
  - Each entry pointer to 1st block that has a record with that value for clustering field
  - Non-dense index


Data file
(Clustering field)
Dept_number
Name
Ssn
Job
Birth_date
Salary
Index file
(<K(i), P(i)> entries)
Clustering field value
Block pointer
clustering indexes

Problems of Clustering Indexes

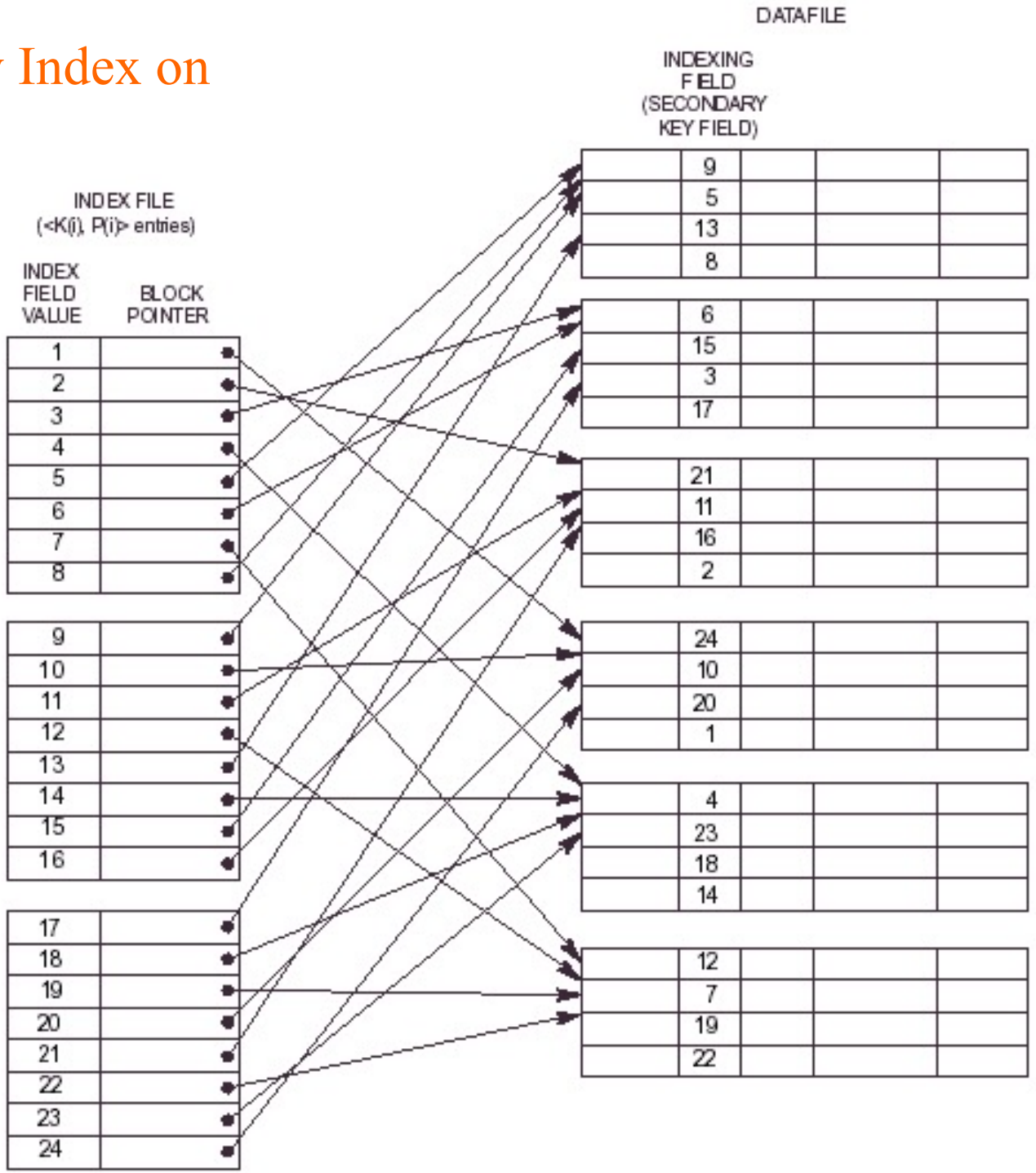
- Record insertion & deletion
  - Index structure is changed
  - Solution
    - Reserve a whole block (or a cluster of continuous blocks) for each value of the clustering field
    - Similar to the extensible hashing


DATA FILE BUCKETS
local depth of each bucket
d' = 3
bucket for records whose hash values start with 000
d' = 3
bucket for records whose hash values start with 001
DIRECTORY
000
001
010
011
100
101
110
111
d' = 2
bucket for records whose hash values start with 01
d' = 2
bucket for records whose hash values start with 10
global depth d = 3
d' = 3
bucket for records whose hash values start with 110
d' = 3
bucket for records whose hash values start with 111

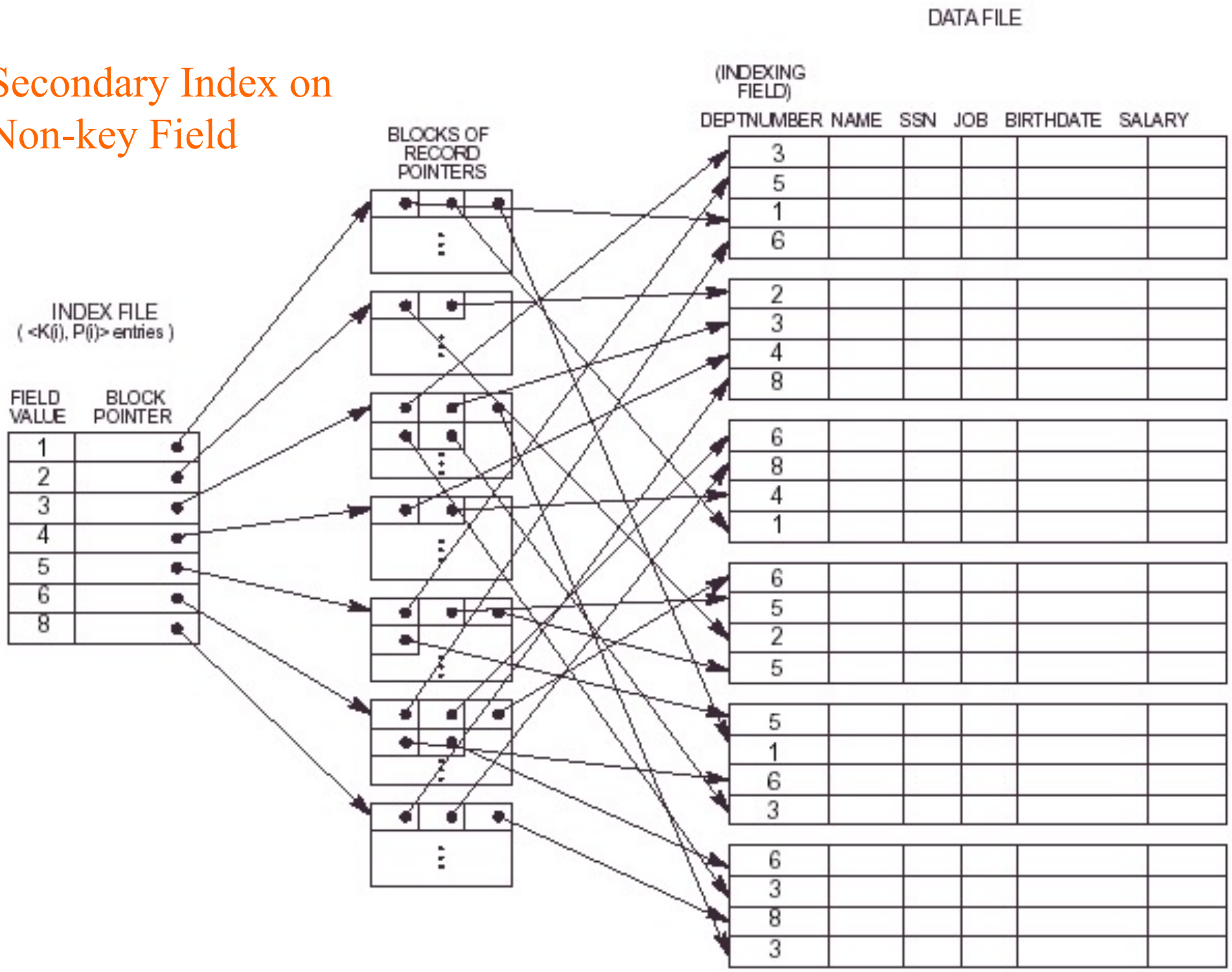
Secondary Index

- An ordered file with two fields <indexing field, pointer>
  - Indexing field: no-ordering field
  - Pointer: a block pointer or a record pointer
- Indexing field
  - Key (secondary key)
    - dense index
    - Pointer to the block which store the indexed record
  - Non-key
    - Several index entries with the value, one for each record
    - Variable length records for the index entries
    - Create an extra level of indirection to handle multiple pointers

Secondary Index on Key Field



Secondary Index on Non-key Field



Types of Indexes

	Index Field Used for Physical Ordering of the File	Index Field Not Used for Physical Ordering of the File
Indexing field is key	Primary index	Secondary index (Key)
Indexing field is nonkey	Clustering index	Secondary index (NonKey)

Properties of Indexes

Type of Index	Number of (First-Level) Index Entries	Dense or Nondense (Sparse)	Block Anchoring on the Data File
Primary	Number of blocks in data file	Nondense	Yes
Clustering	Number of distinct index field values	Nondense	Yes/no[a]
Secondary (key)	Number of records in data file	Dense	No
Secondary (nonkey)	Number of records[b] or number of distinct index field values[c]	Dense or Nondense	No

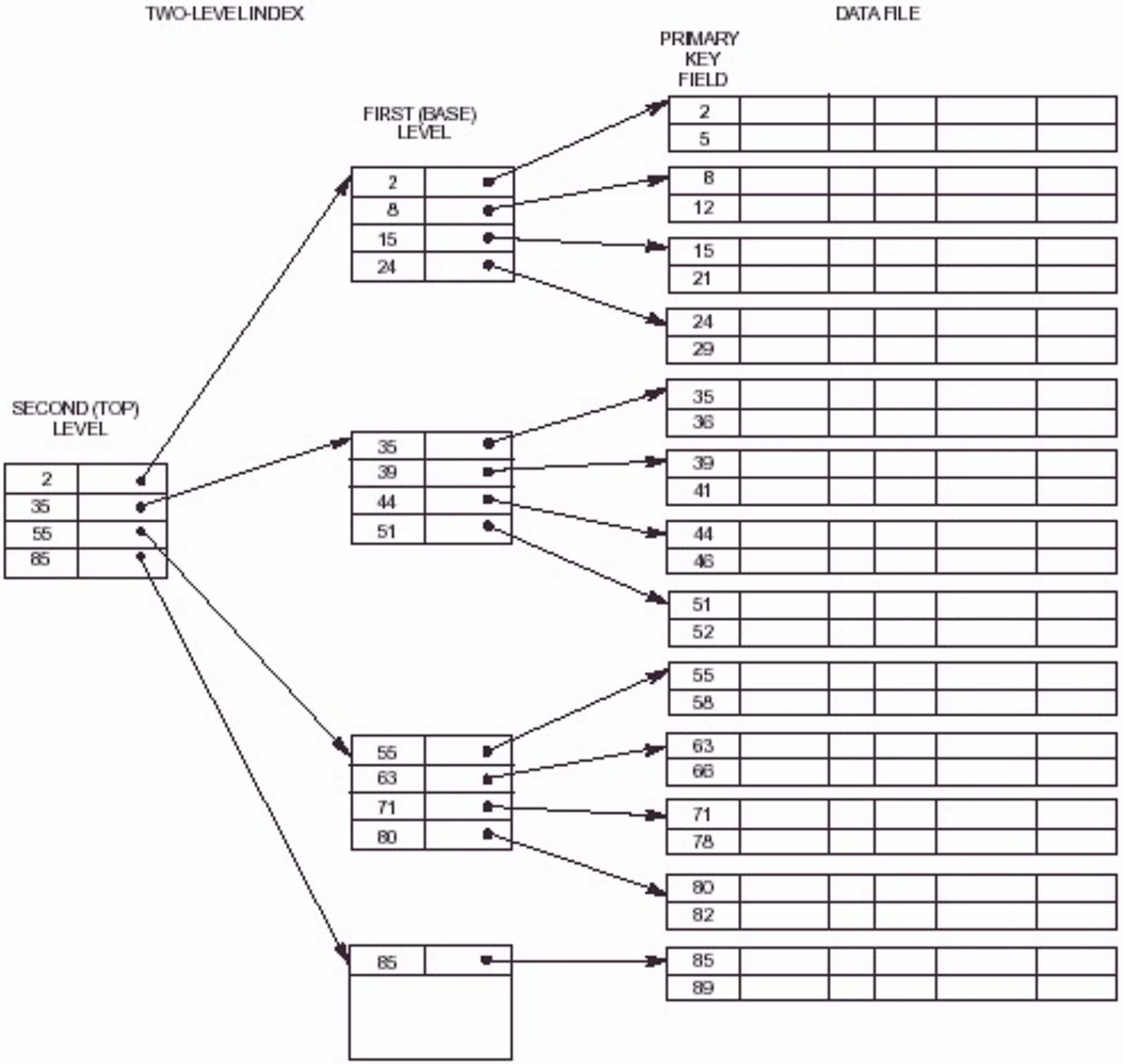
Content

- Single-level ordered indexes
- Multilevel indexes
- Dynamic multilevel indexes using B-trees and B+-trees
- Multiple dimensional Indexes
- Other types of indexes
- Issues Concerning Indexes

Multilevel Index

- ISAM (Index Sequential Access Method)
  - index sequential file on primary key
    - First level: cylinder index, anchor record
    - Second level: track index

(Not used anymore)


TWO-LEVEL INDEX
DATA FILE
PRIMARY KEY FIELD
FIRST (BASE) LEVEL
SECOND (TOP) LEVEL
2
35
55
85
2
8
15
24
35
39
44
51
55
63
71
80
85
2
5
8
12
15
21
24
29
35
36
39
41
44
46
51
52
55
58
63
66
71
78
80
82
85
89

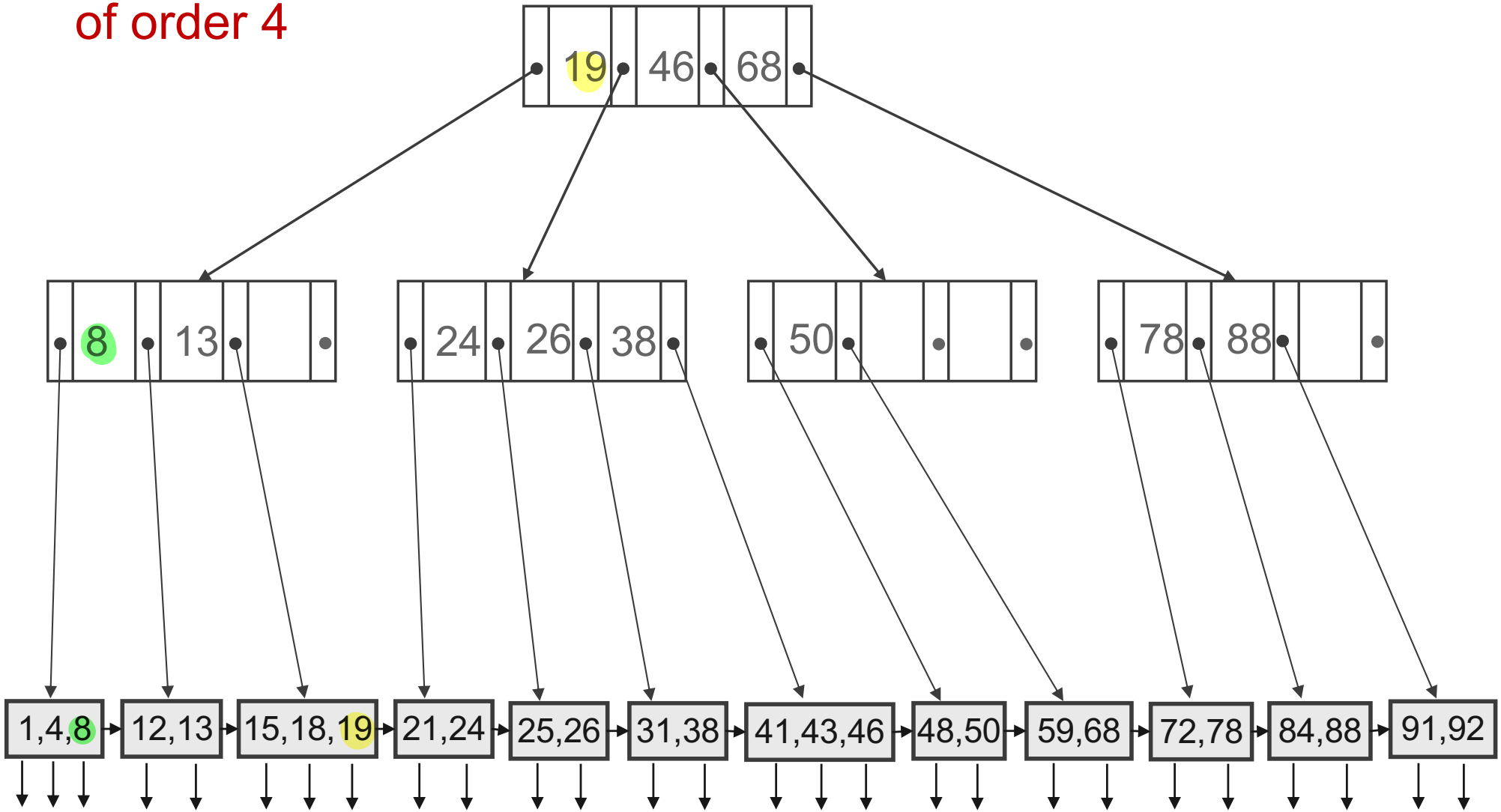
Content

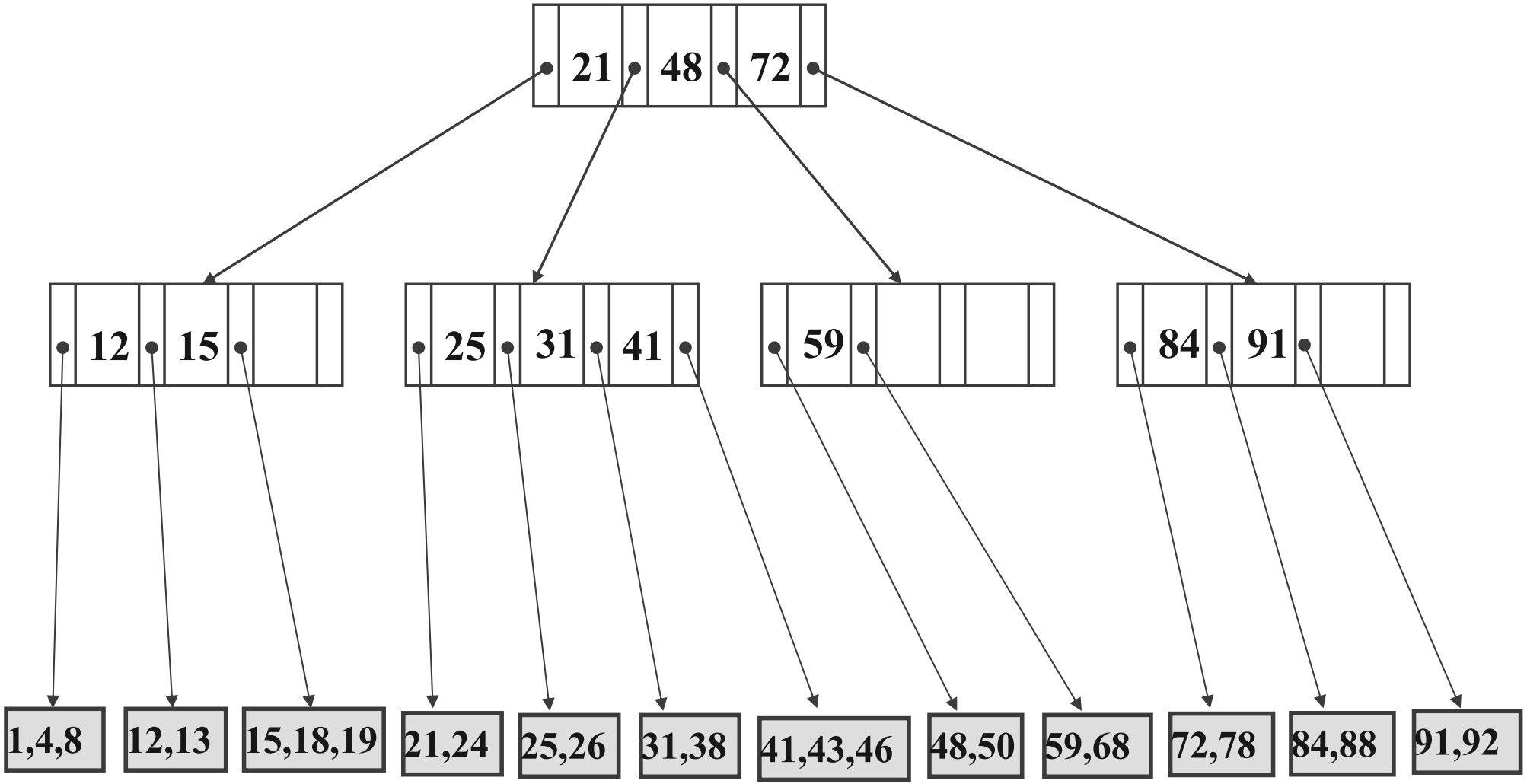
- Single-level ordered indexes
- Multilevel indexes
- Dynamic multilevel indexes using B+-trees
- Multiple dimensional Indexes
- Other types of indexes
- Issues Concerning Indexes

B+-Trees

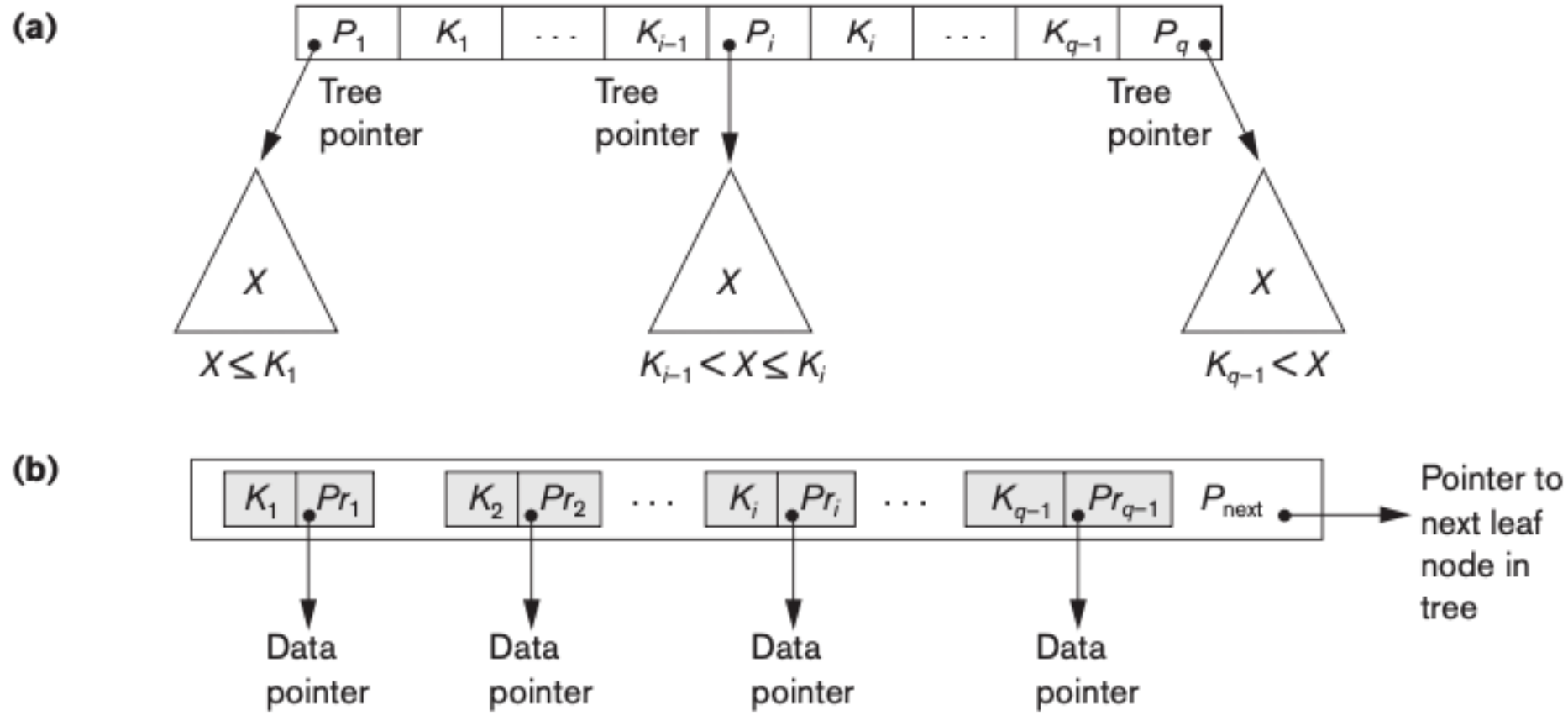
- B+-Trees of order M
  - M-way search tree
  - Root is either a leaf or has between 2 & M children
  - All nonleaf nodes have between $\lceil M/2 \rceil$ & M children
  - All leaves are at the same depth
  - Indexed key are stored in leaves
  - In each interior node, there are pointers $P_0, P_1, \ldots, P_{M-1}$ to children and values $k_1, k_2, \ldots, k_{M-1}$, representing the smallest key found in the subtrees $P_2, P_3, \ldots, P_M$ respectively.

B+-Tree of order 4




21
48
72
12
15
25
31
41
59
84
91
1,4,8
12,13
15,18,19
21,24
25,26
31,38
41,43,46
48,50
59,68
72,78
84,88
91,92

B+-Trees


(a)
P_1
K_1
. . .
K_{i-1}
P_i
K_i
. . .
K_{q-1}
P_q
Tree pointer
Tree pointer
Tree pointer
X
X
X
$X \leq K_1$
$K_{i-1} < X \leq K_i$
$K_{q-1} < X$
(b)
K_1 Pr_1
K_2 Pr_2
. . .
K_i Pr_i
. . .
K_{q-1} Pr_{q-1}
P_{next}
Pointer to next leaf node in tree
Data pointer
Data pointer
Data pointer
Data pointer

B+-Tree Find

Set N to the root

Repeat

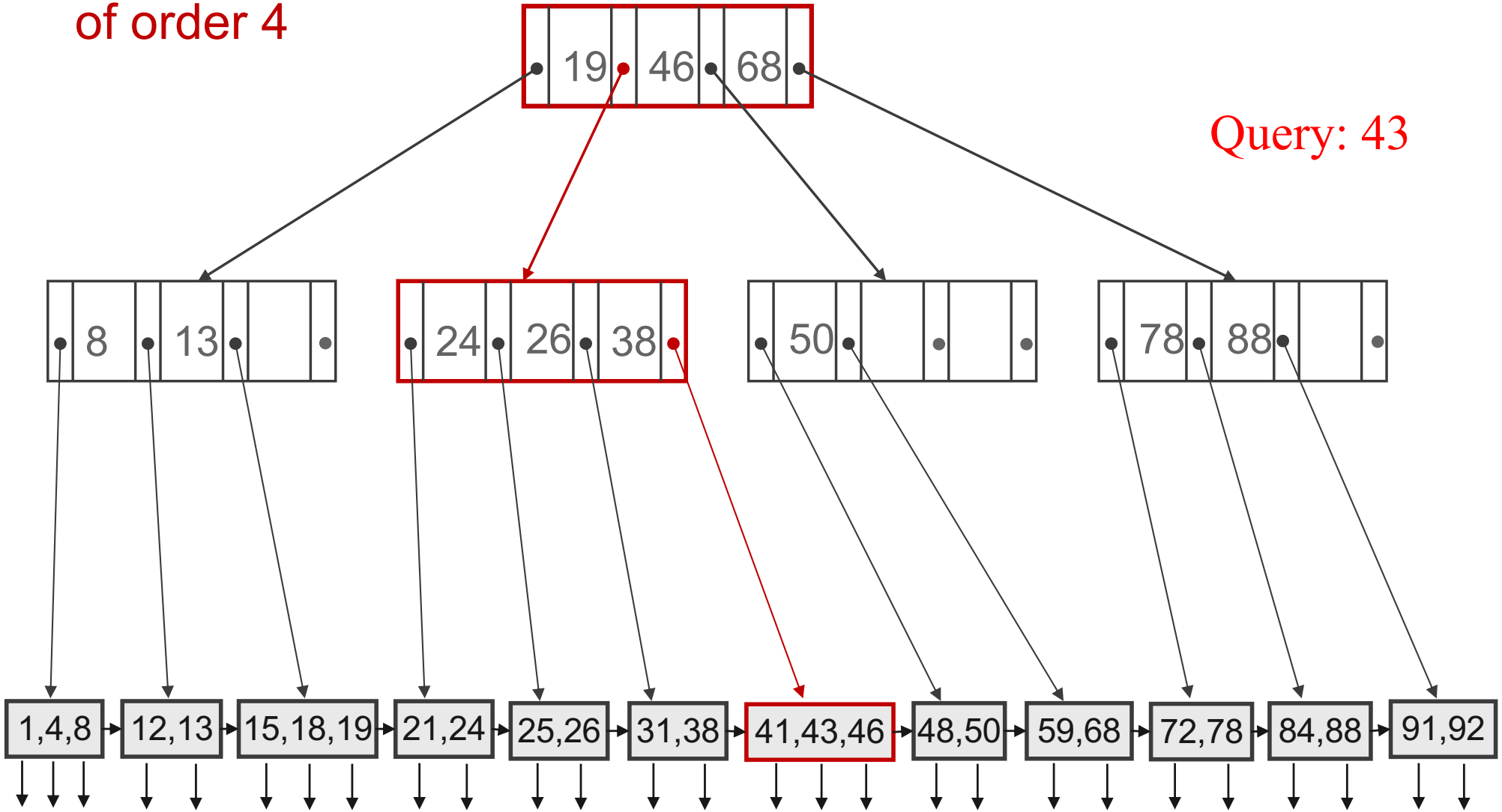
if k_i **<** **K** **<** k_{i+1}

then set N to the node pointed by P_i

Until N is a leaf node

Binary search for K in node N

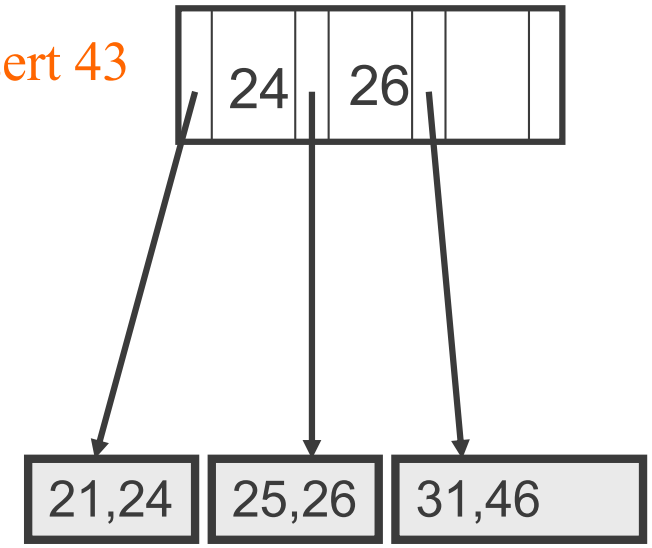
B+-Tree of order 4

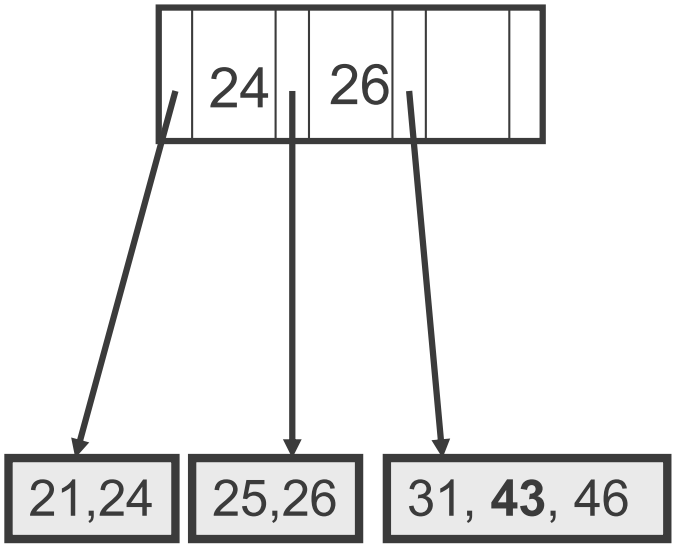


Query: 43

B+-Trees of order 4

Insert 43



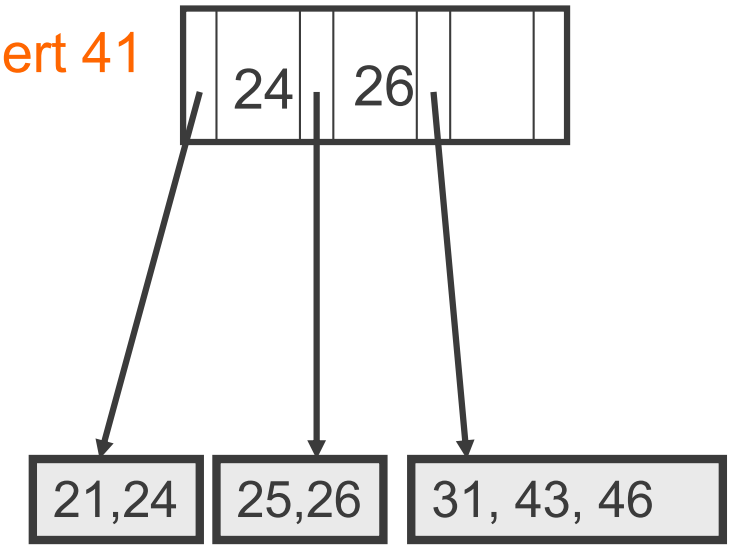


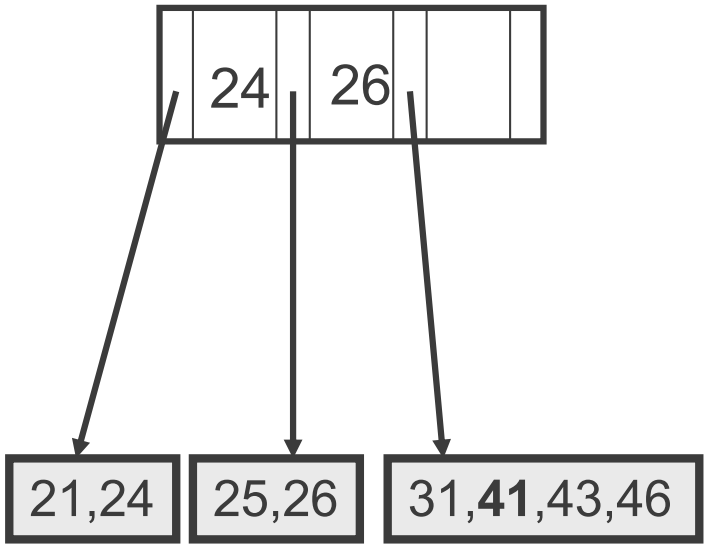
B+-Tree Insertion

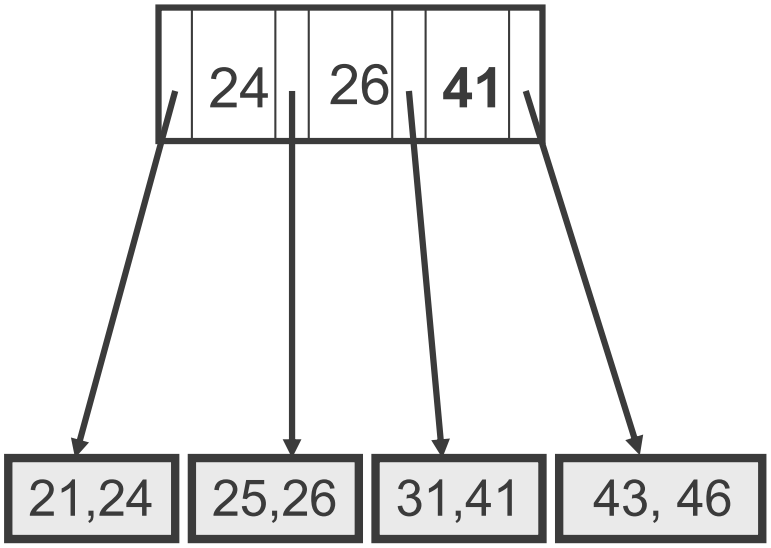
- Follow the path as performing find.
- When get to a leaf node, put into the correct place
- If leave node overflow, spilt and spilt upward if necessary

B+-Trees of order 4

Insert 41



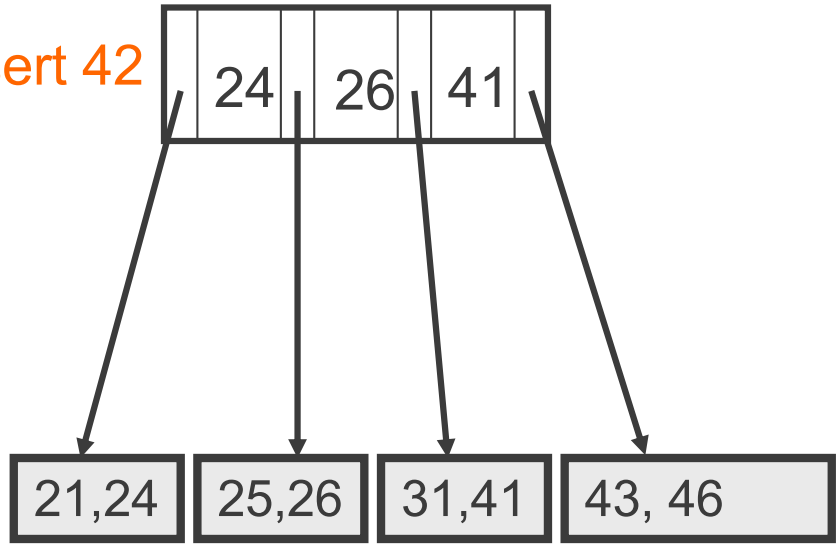


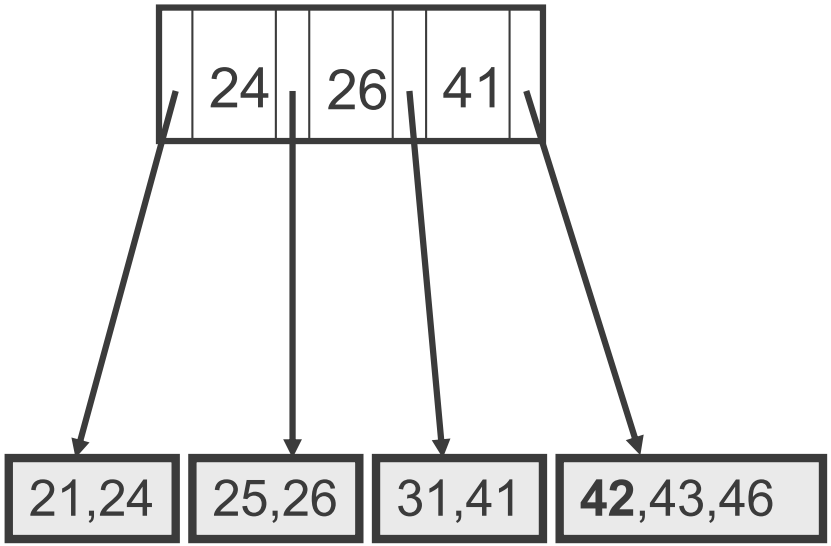


Overflow & Split & Update Upward

B+-Trees of order 4

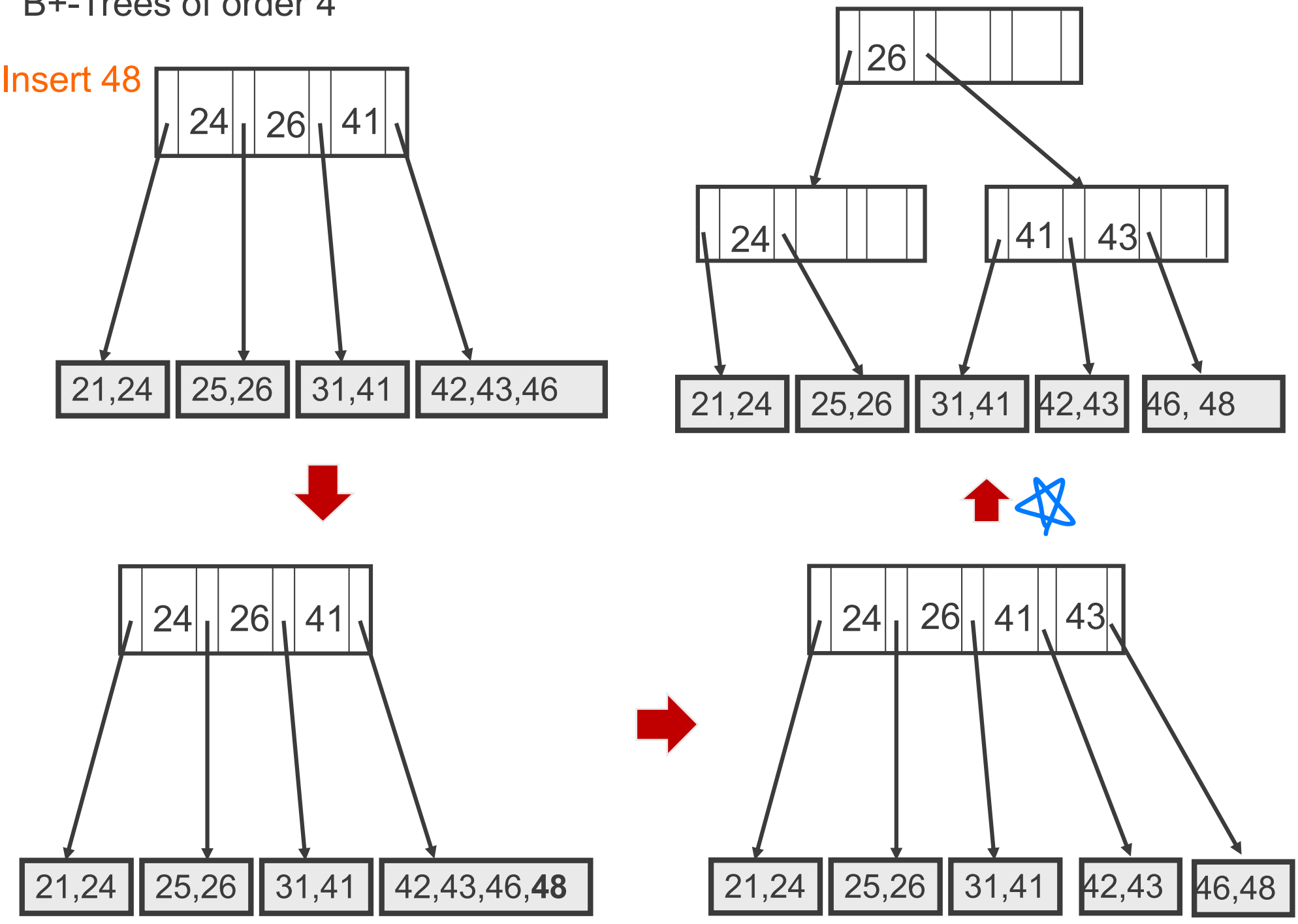
Insert 42



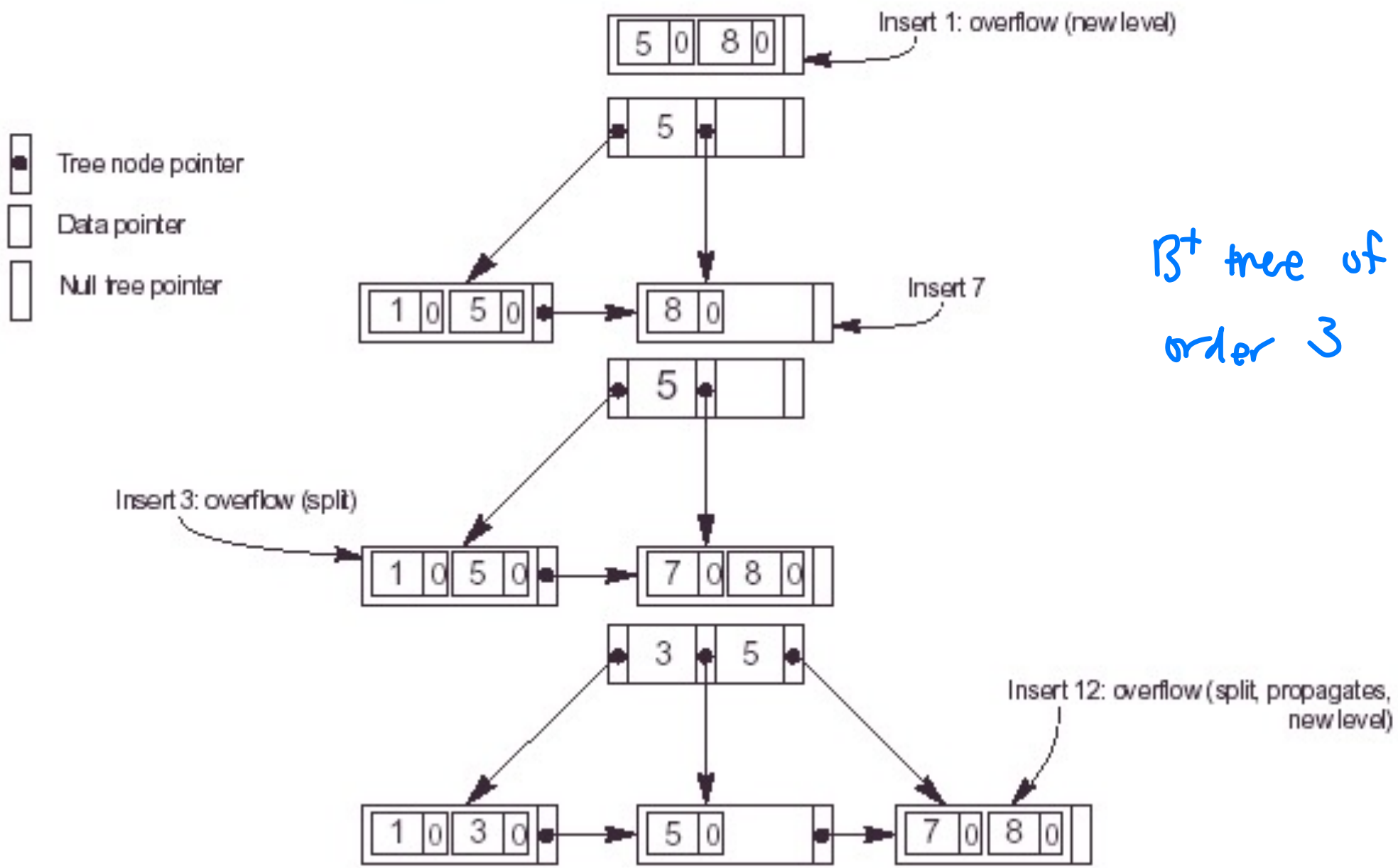


B+-Trees of order 4

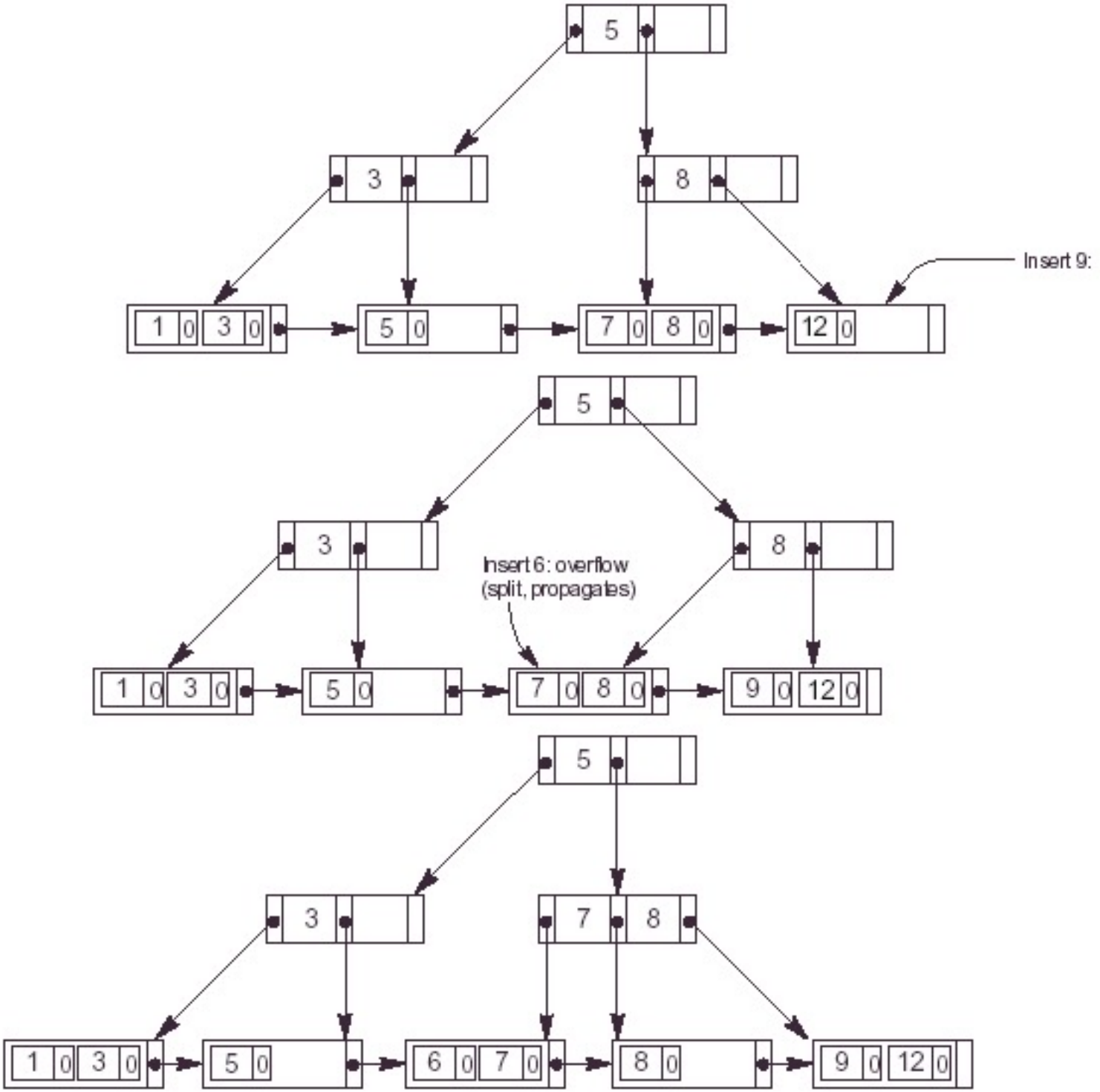
Insert 48



INSERTION SEQUENCE: 8, 5, 1, 7, 3, 12, 9, 6



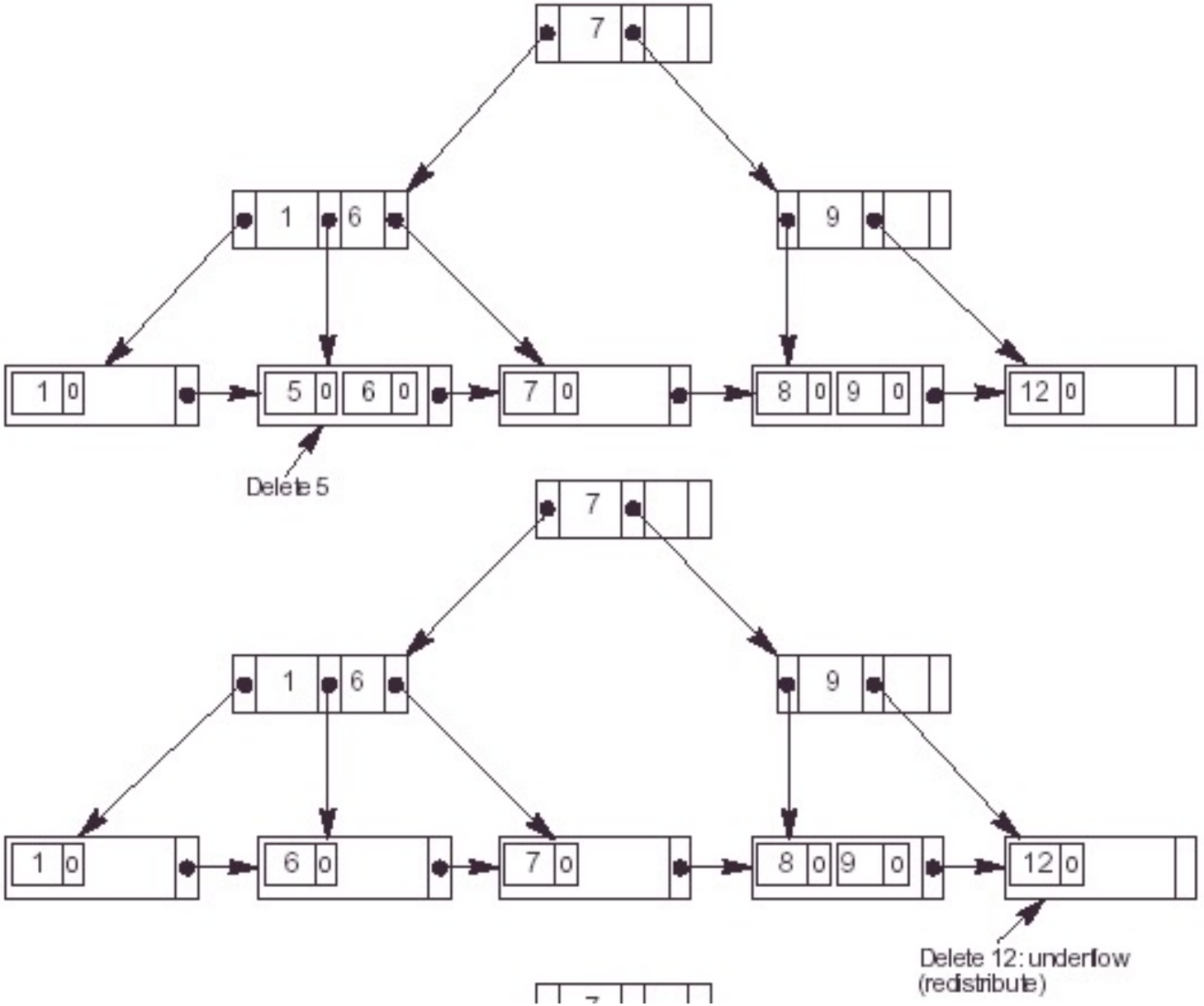
B+ tree of order 3

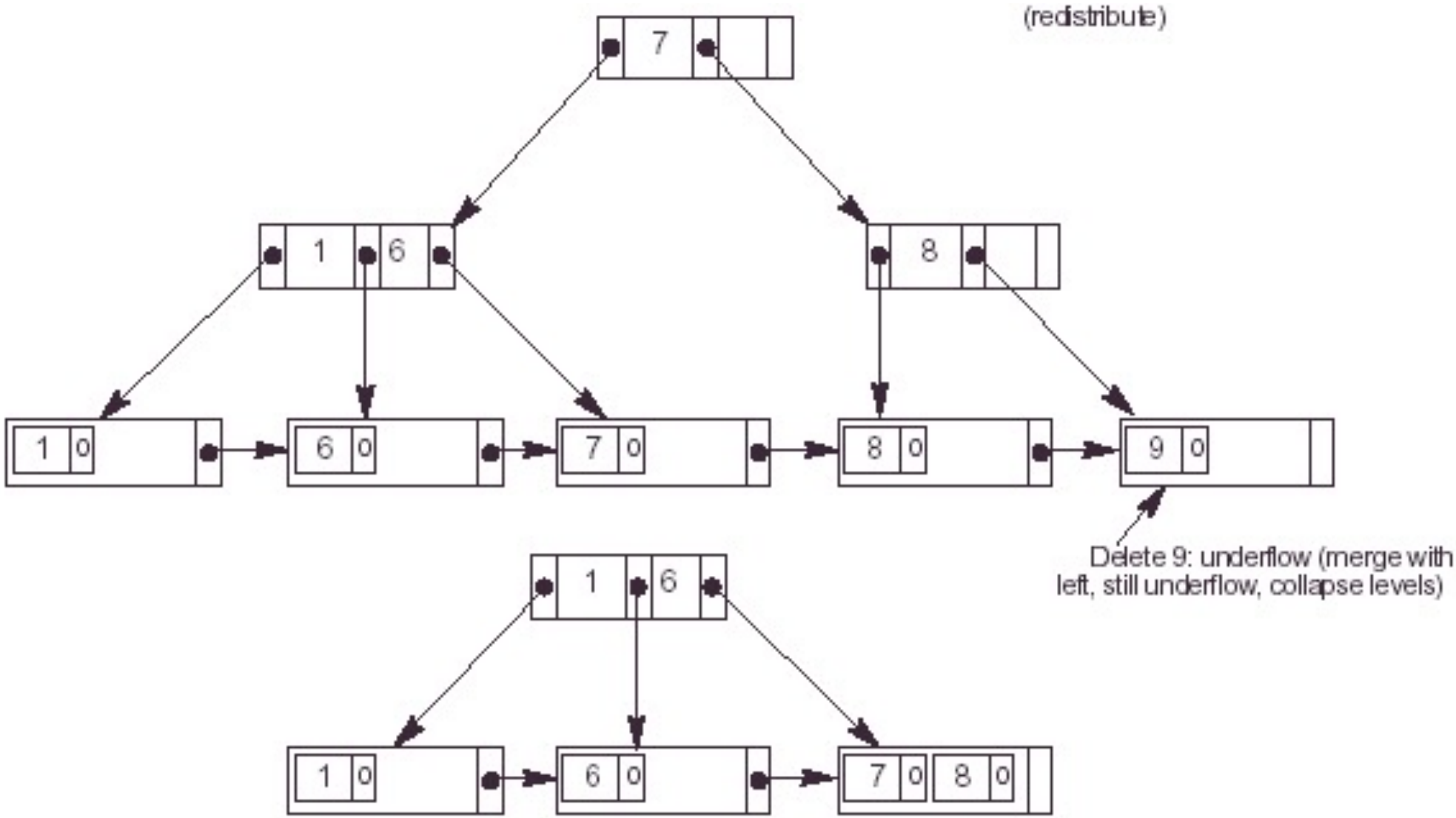

5
3
8
Insert 9:
1 0 3 0
5 0
7 0 8 0
12 0
5
3
8
Insert 6: overflow
(split, propagates)
1 0 3 0
5 0
7 0 8 0
9 0 12 0
5
3
7 8
1 0 3 0
5 0
6 0 7 0
8 0
9 0 12 0

B+-Tree Deletion

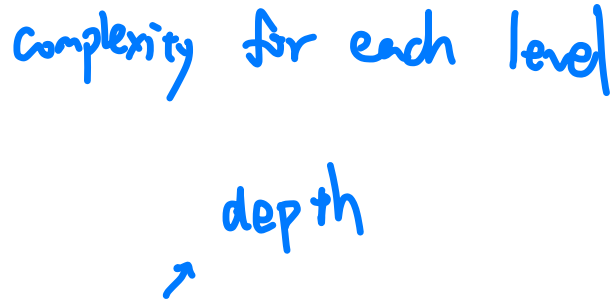
- Follow the path as performing find.
- When get to a leaf node, delete the data
- If leave node underflow, merge and merge upward if necessary.

DELETION SEQUENCE: 5, 12, 9




(redistribute)
7
1
6
8
1 0
6 0
7 0
8 0
9 0
Delete 9: underflow (merge with
left, still underflow, collapse levels)
1
6
1 0
6 0
7 0
8 0

Complexity of B+-Trees of Order M



- Depth: $\lceil \log_{\lceil M/2 \rceil} N \rceil$
- Find: $O(\log_2 \lceil M/2 \rceil * \log_{\lceil M/2 \rceil} N) = O(\log N)$
- Insertion, Deletion: $O(M \log_M N)$
- In memory, M is chosen as 3 or 4
- In disk, M is chosen that the node size fit disk page

  $32 <= M <= 256$

Content

- Single-level ordered indexes
- Multilevel indexes
- Dynamic multilevel indexes using B-trees and B+-trees
- Multiple dimensional Indexes
- Other types of indexes
- Issues Concerning Indexes

Index on Multiple Keys

- Multiple dimensional indexing
- Applications
  - Multidimensional query (partial match range query)
    - Finding the PC models where 4 < RAM < 16 (single dimensional)
    - Finding the PC models where 2.5 < CPU < 3.9 (single dimensional)
    - Finding the PC models where (4 < RAM < 16) & (2.5 < CPU < 3.9)
  - Spatial database, Geographic information system (GIS)
    - Finding the objects within a query region (region query)
      - Find the objects with coordinates (4 < X < 16) & (2.5 < Y < 3.9)
    - Finding the rectangles that intersect with the query region (intersection query)
    - Finding the rectangles that contain a given point (where-am-I query)
    - Finding the point that is nearest to the query point (nearest-neighbor query)

Multidimensional Indexes

- Ordered index on multiple attributes
- Partitioned Hashing
- Grid files
- kd-trees
- Quad trees
- R-trees

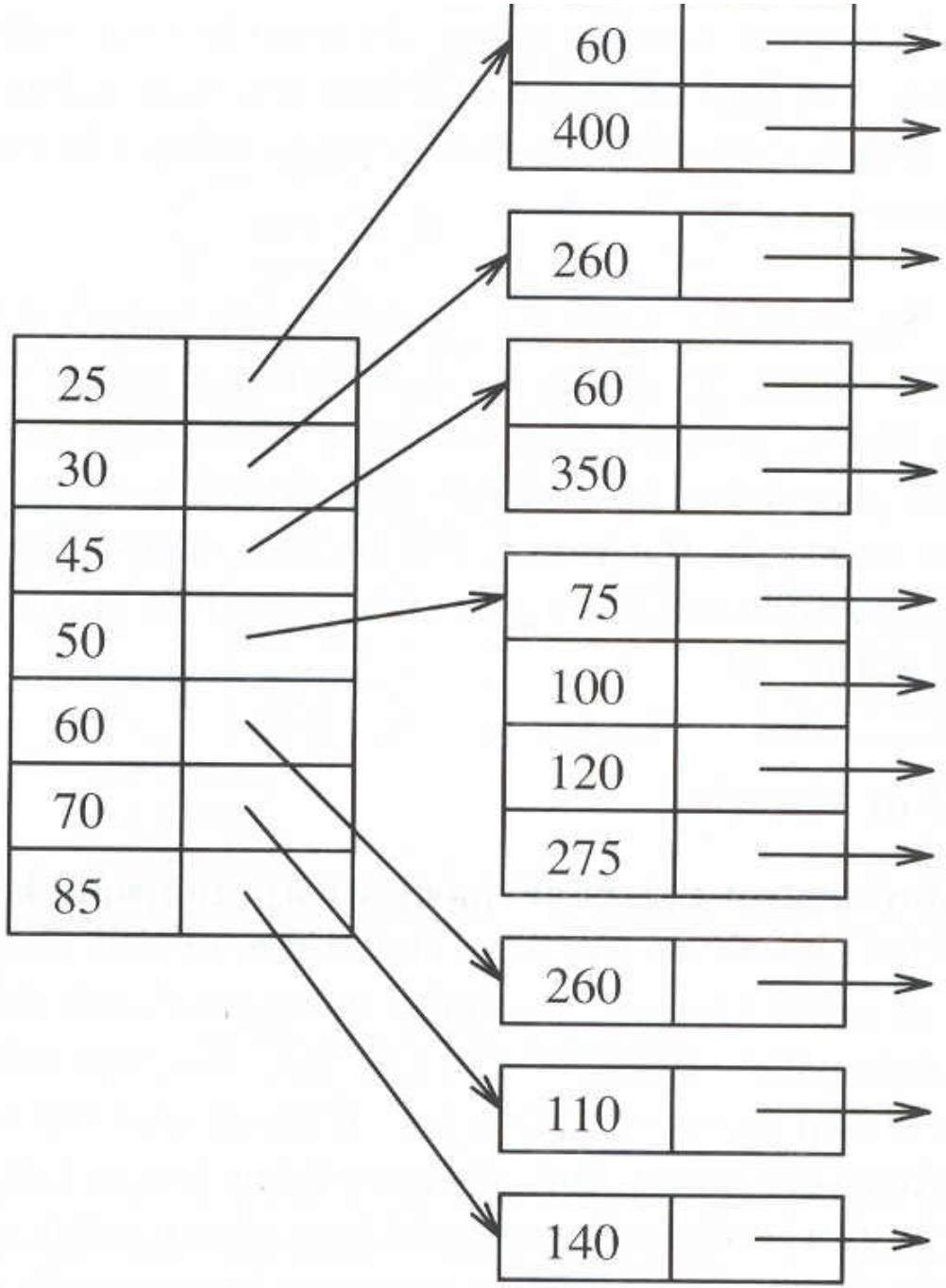


Figure 5.12: Multilevel indexes for age/salary data

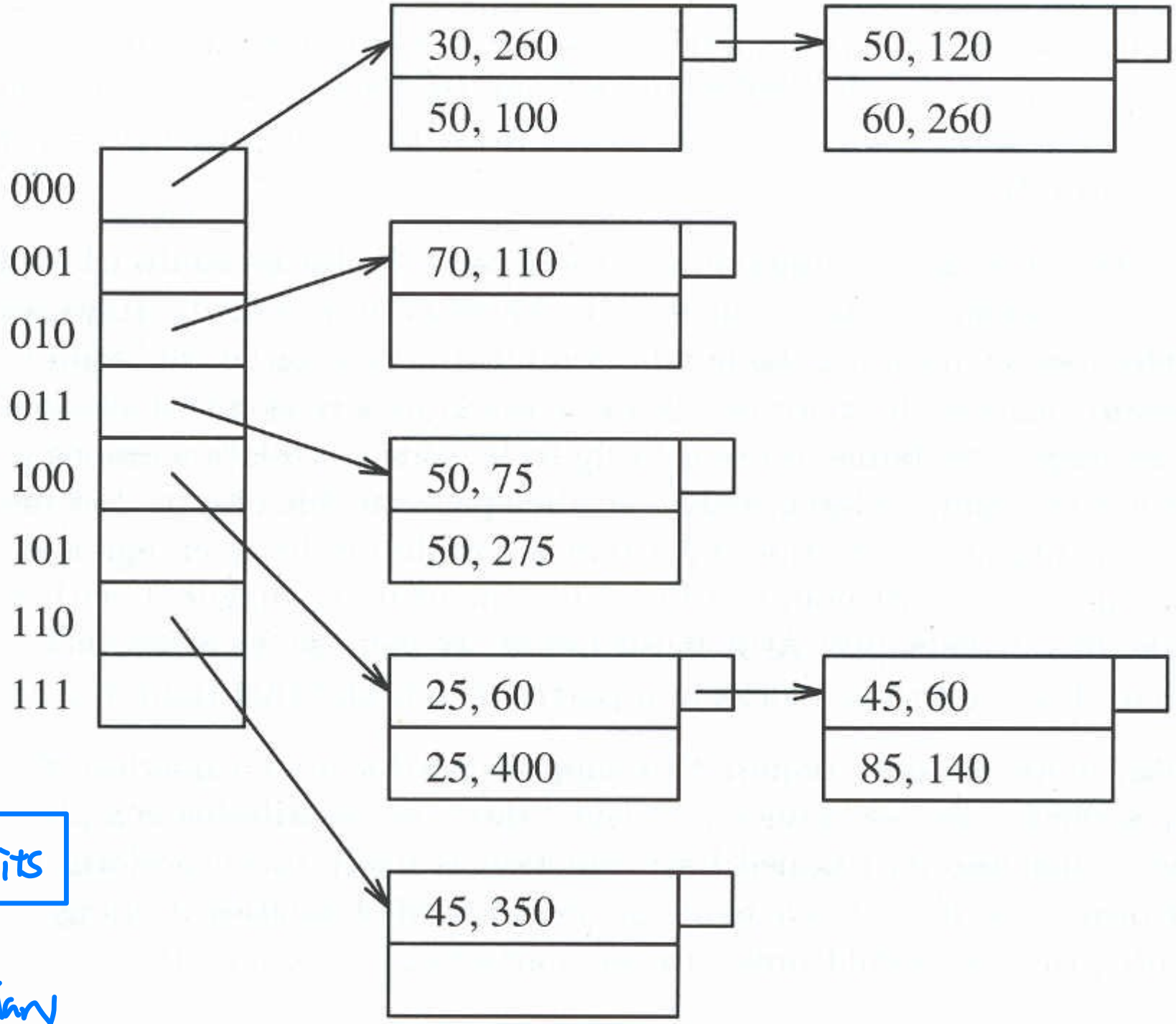
wan Shan, CS, NCCU

Ordered Index on Multiple Dimension

- An index on a composite key of *n* dimension
- e.g. index on composite key (age, salary)
  - (age=25, salary=60),
  - (age=25, salary=400),
  - (age=30, salary=260),
  - (age=45, salary=60),
  - (age=45, salary=350)
  - (age=50, salary=75),
  - (age=50, Salary=100),
  - …

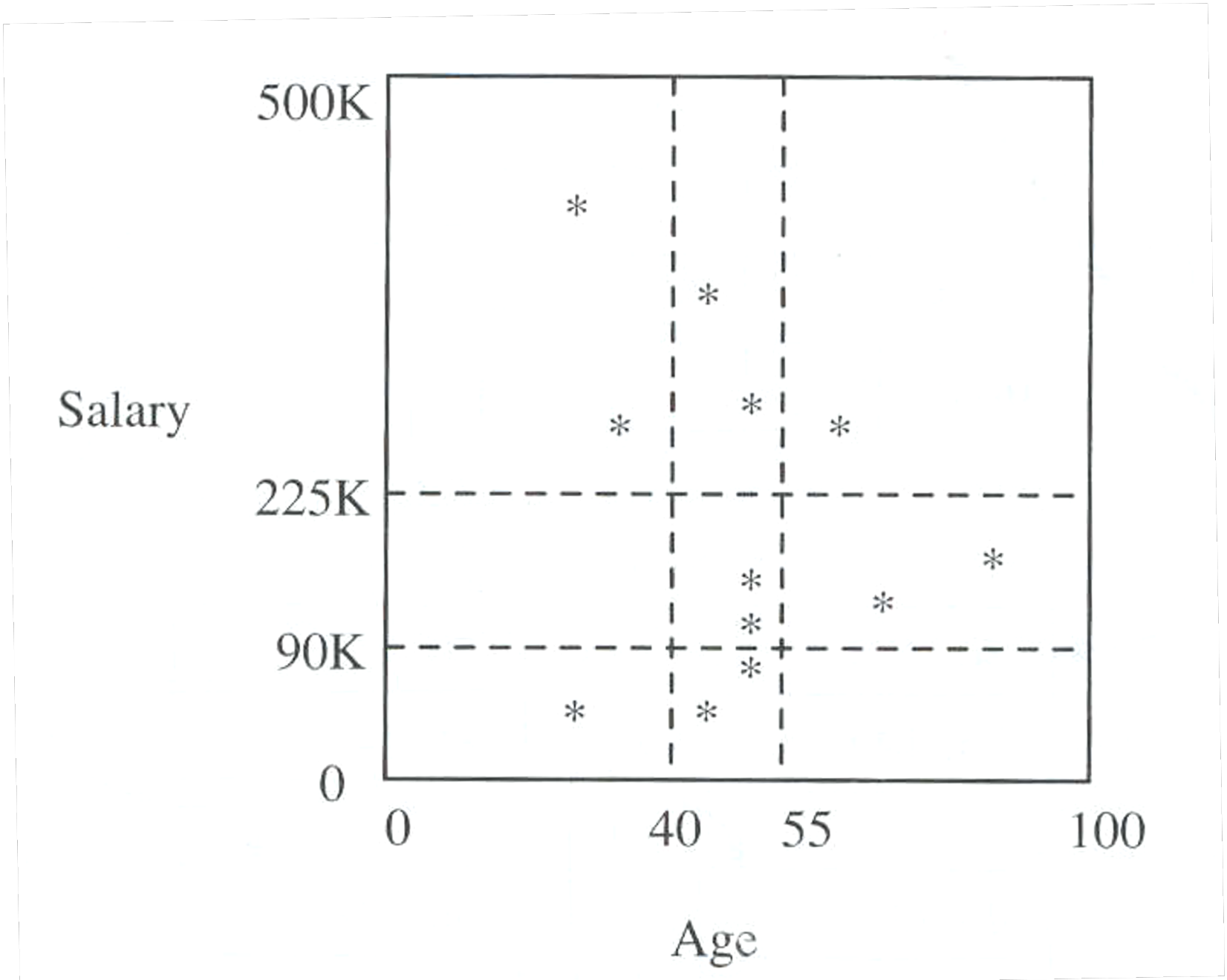
Partitioned Hash

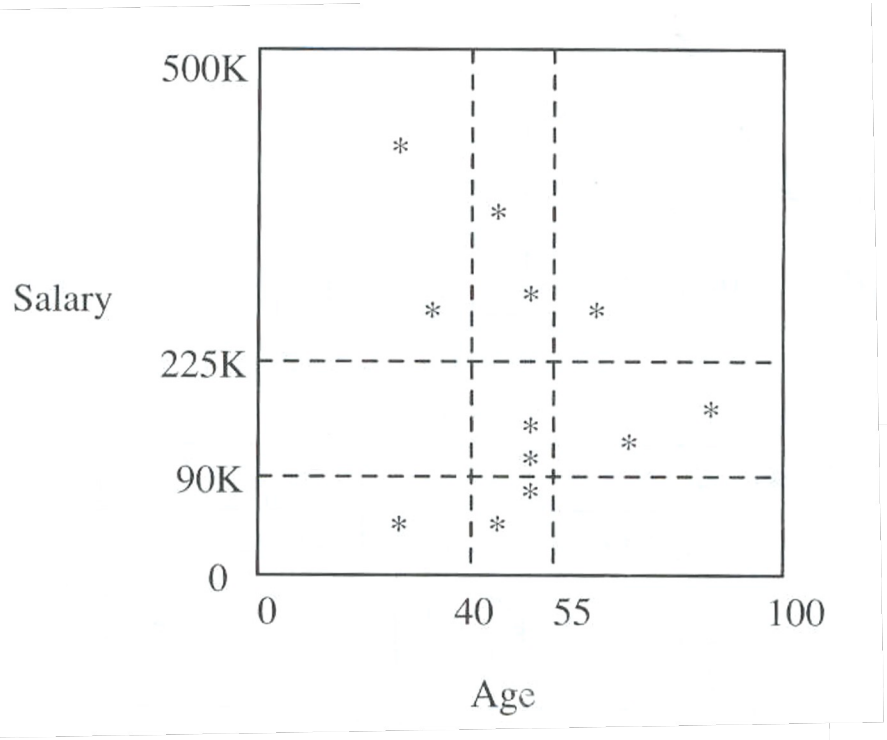
- Extension of static external hashing
- For key consisting of *n* dimension, hashing function produces result with *n* separate hash address
- Bucket address is a concatenation of these *n* address
- e.g. hashing on (age, salary) 2 dimensions
  - 1 bit hash for age (age mod 2)
  - 2 bit hash for salary (salary mod 4)

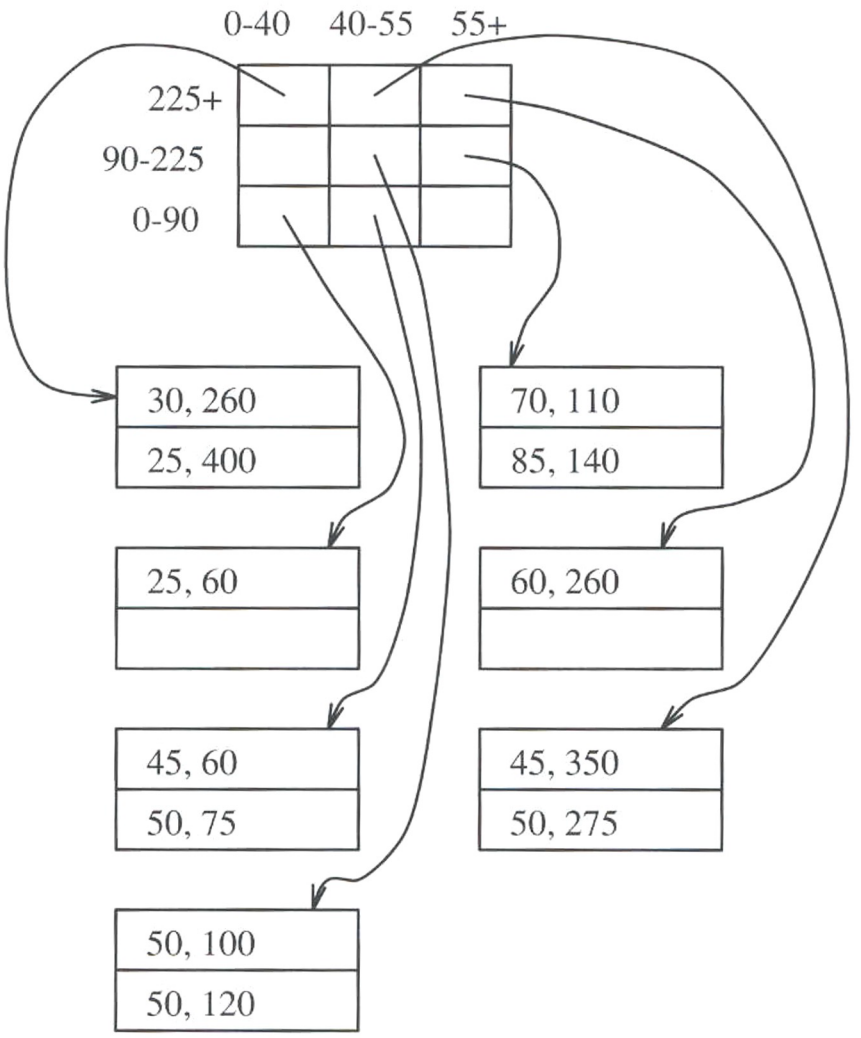

000
001
010
011
100
101
110
111
30, 260
50, 100
50, 120
60, 260
70, 110
50, 75
50, 275
25, 60
25, 400
45, 60
85, 140
45, 350
1 bit
2 bits
age
salary

Grid Files

- Partitioned the multidimensional space into grids
- In each dimension, grid lines partition space into stripes
- Points falling on a grid line are considered to belong to the stripe for which that grid line is the lower boundary
- #(grid lines) vary in different dimensions
- Different spacings between adjacent grid lines


500K
225K
90K
0
Salary
0
40
55
100
Age


500K
225K
90K
0
Salary
0
40
55
100
Age

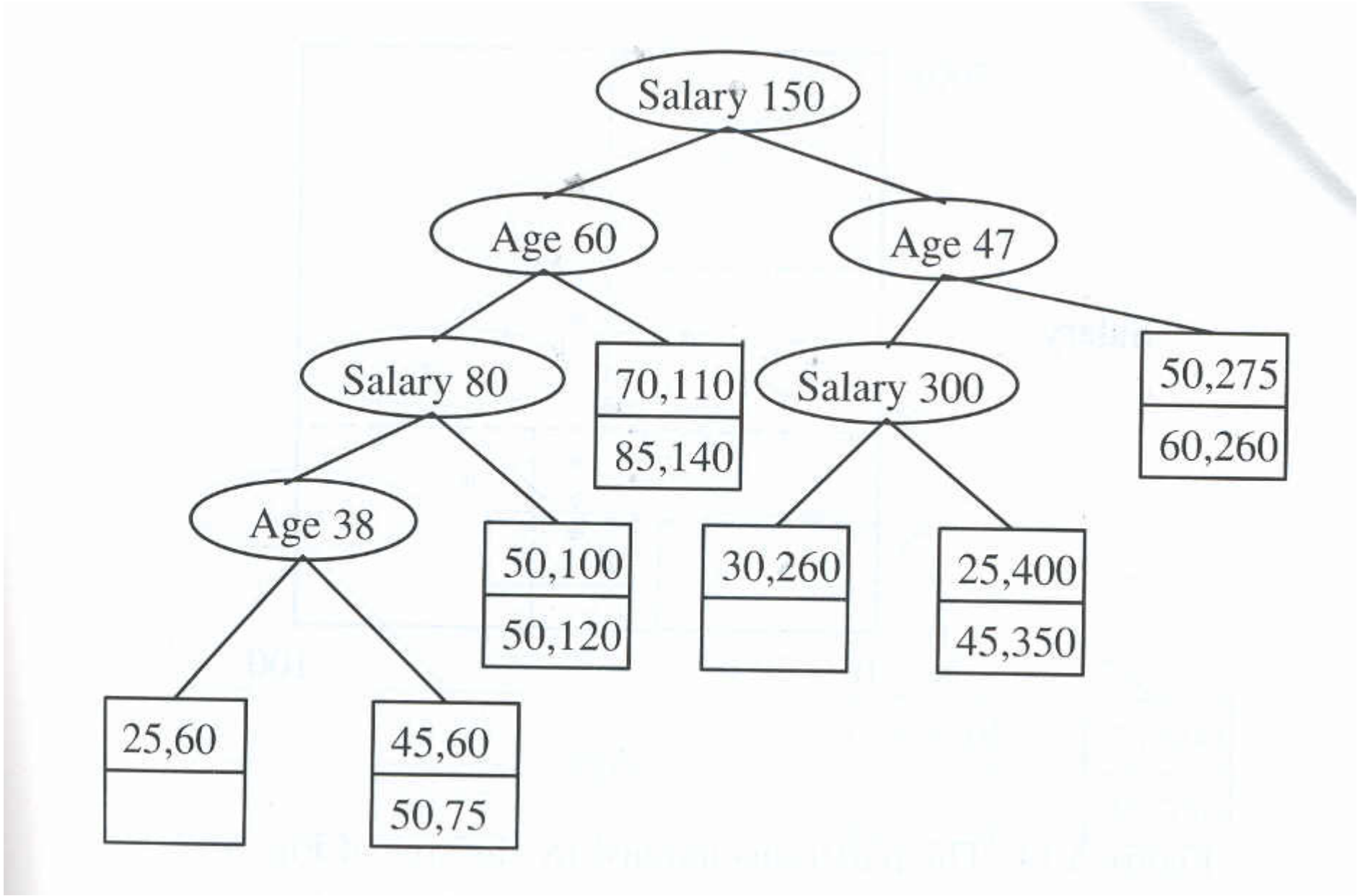

0-40
40-55
55+
225+
90-225
0-90
30, 260
25, 400
70, 110
85, 140
25, 60
60, 260
45, 60
50, 75
45, 350
50, 275
50, 100
50, 120

Operations on Grid Files

- Lookup
  - A multidimensional array of buckets
  - Look at each dimension of the point and determine the position of the point in the grid for that dimension
  - The positions of the point in each of the dimensions together determine the bucket
- Insertion
  - Overflow
    - Add overflow buckets
    - Reorganize the structure
      - by adding or moving the grid lines
      - Adding a grid line splits all the buckets along that line
      - Choose the best spilt line

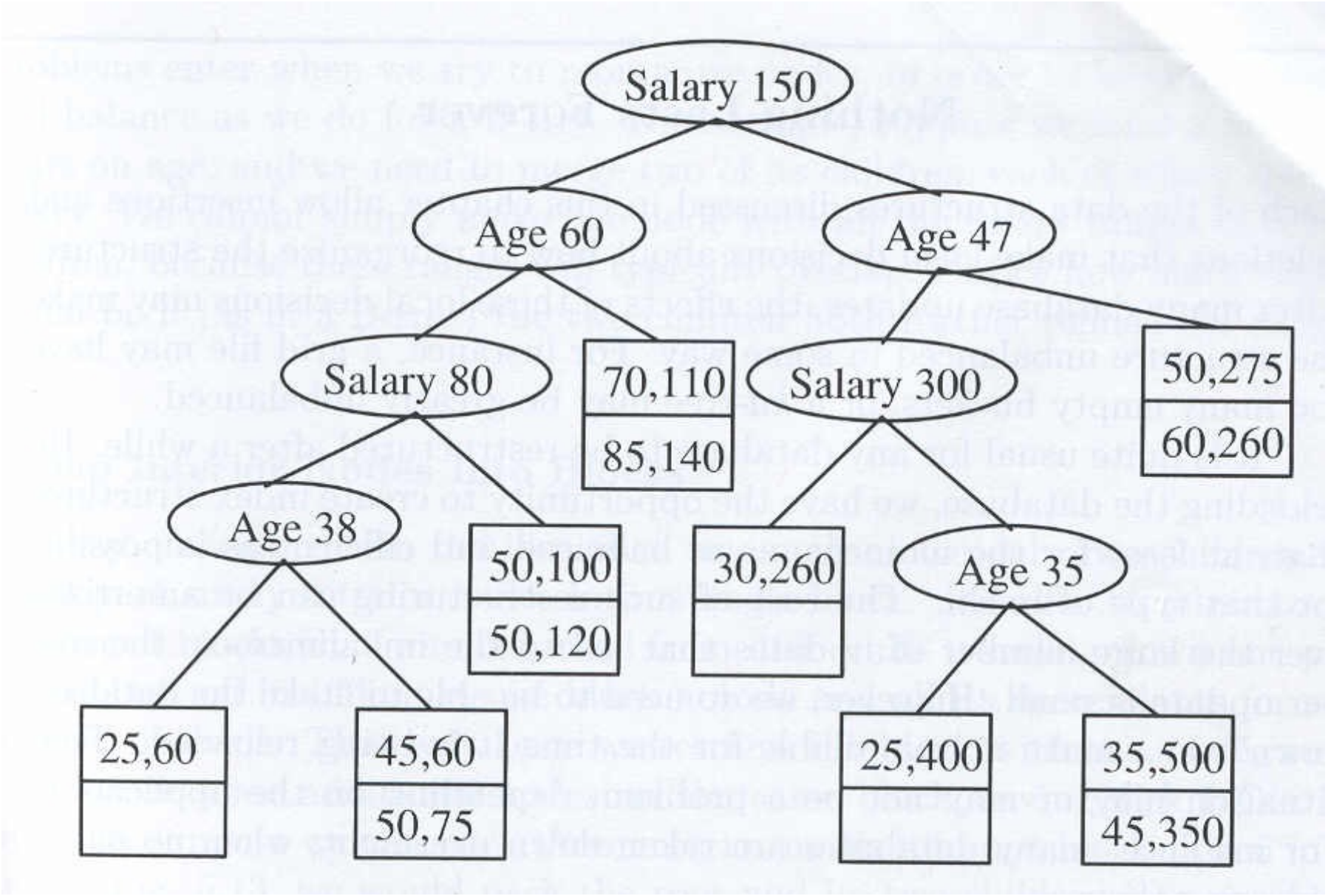
kd-Trees

- kd-tree in main memory
  - K-dimensional binary search tree (proposed by J. L. Bentley)
  - Interior nodes have an associated attribute and a value the splits the data into 2 parts
  - The attribute at different levels are different, with levels rotating among attributes of all dimensions


Salary 150
Age 60
Age 47
Salary 80
70,110
85,140
Salary 300
50,275
60,260
Age 38
50,100
50,120
30,260
25,400
45,350
25,60
45,60
50,75

Operations on kd-Trees

- Lookup
  - proceeds as in a binary search tree
  - Make a decision which way to go at each interior node
- Insertion
  - Proceed as for a lookup
  - If there is a room in the leaf, insert new data
  - Otherwise, split the block and divide its contents according to whatever attribute is appropriate at the level of the split leaf, create a new interior node whose children are 2 new blocks

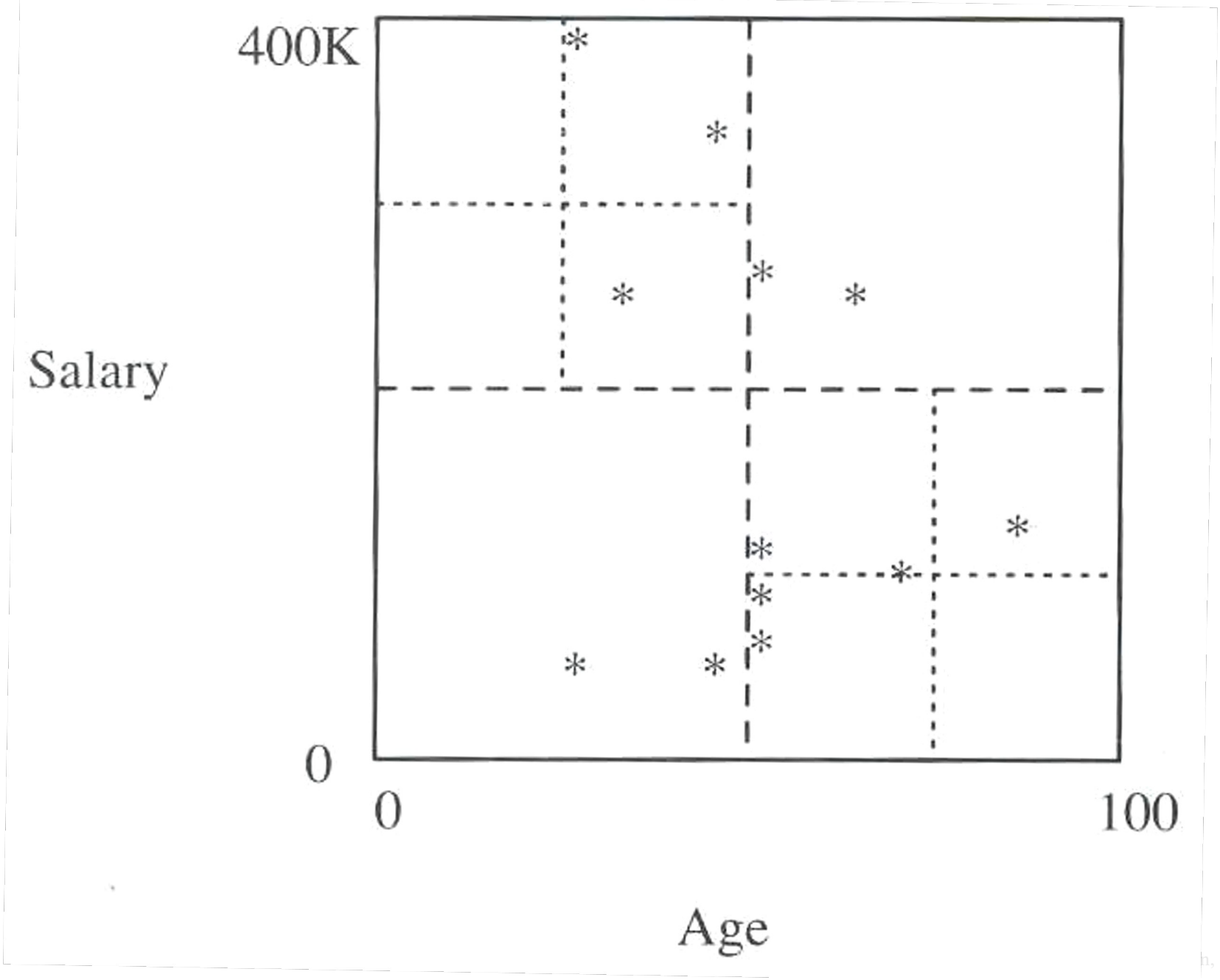

Salary 150
Age 60
Age 47
Salary 80
70,110
85,140
Salary 300
50,275
60,260
Age 38
50,100
50,120
30,260
Age 35
25,60
45,60
50,75
25,400
35,500
45,350

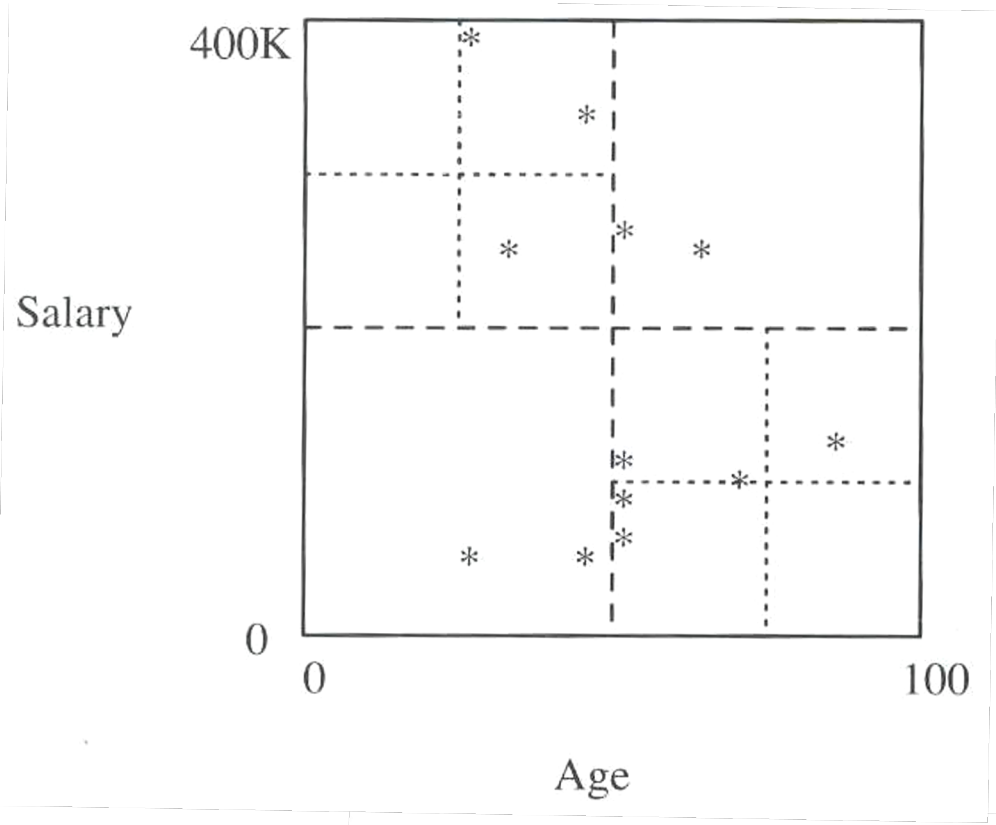
Adapting kd-trees to Secondary Storage

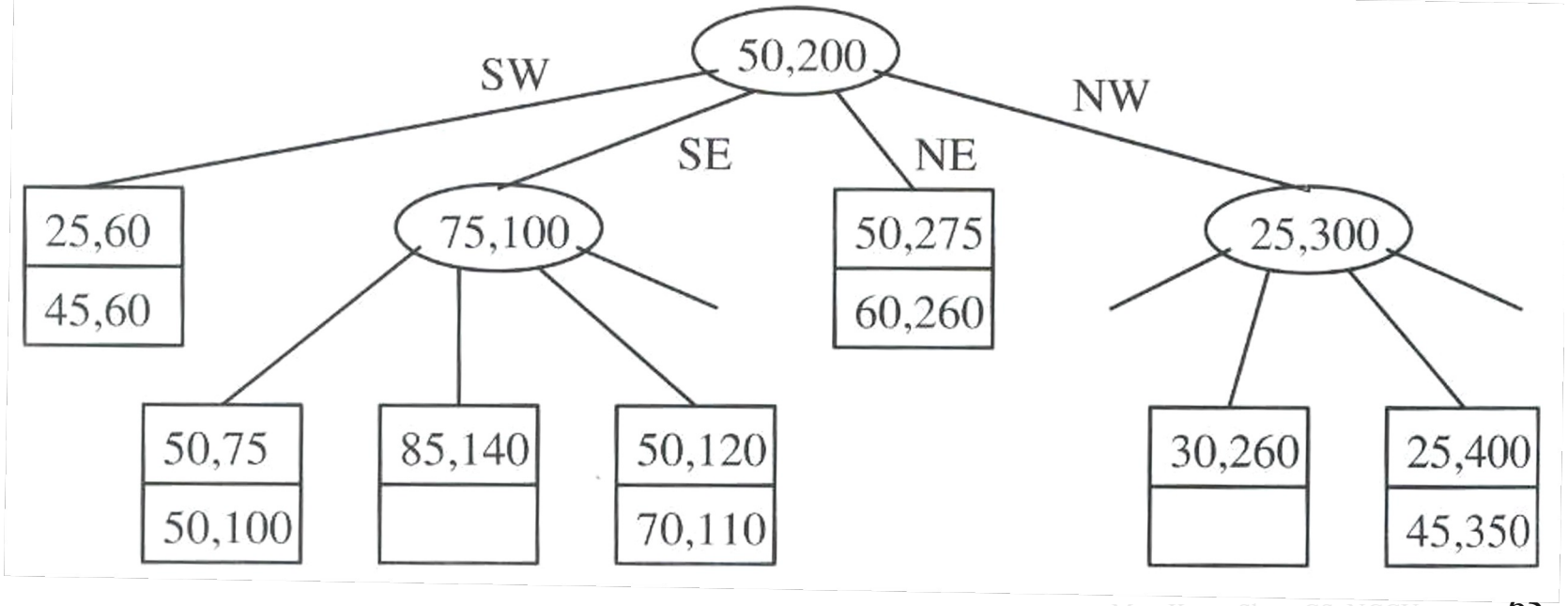
- Multiway branches at interior nodes
- Group interior nodes into blocks

Quad-Trees

- Quad-tree (proposed by H. Samet)
  - Each interior node corresponds to a square region in 2-dimensions or a k-dimensional cube in k-dimensions.
  - If #(points) in a square is no larger than what will fit in a block => this square as a leaf
  - If #(points) in a square is larger than block factor, treat the square as an interior node with children corresponding to its 4 quadrants


400K
Salary
0
0
100
Age

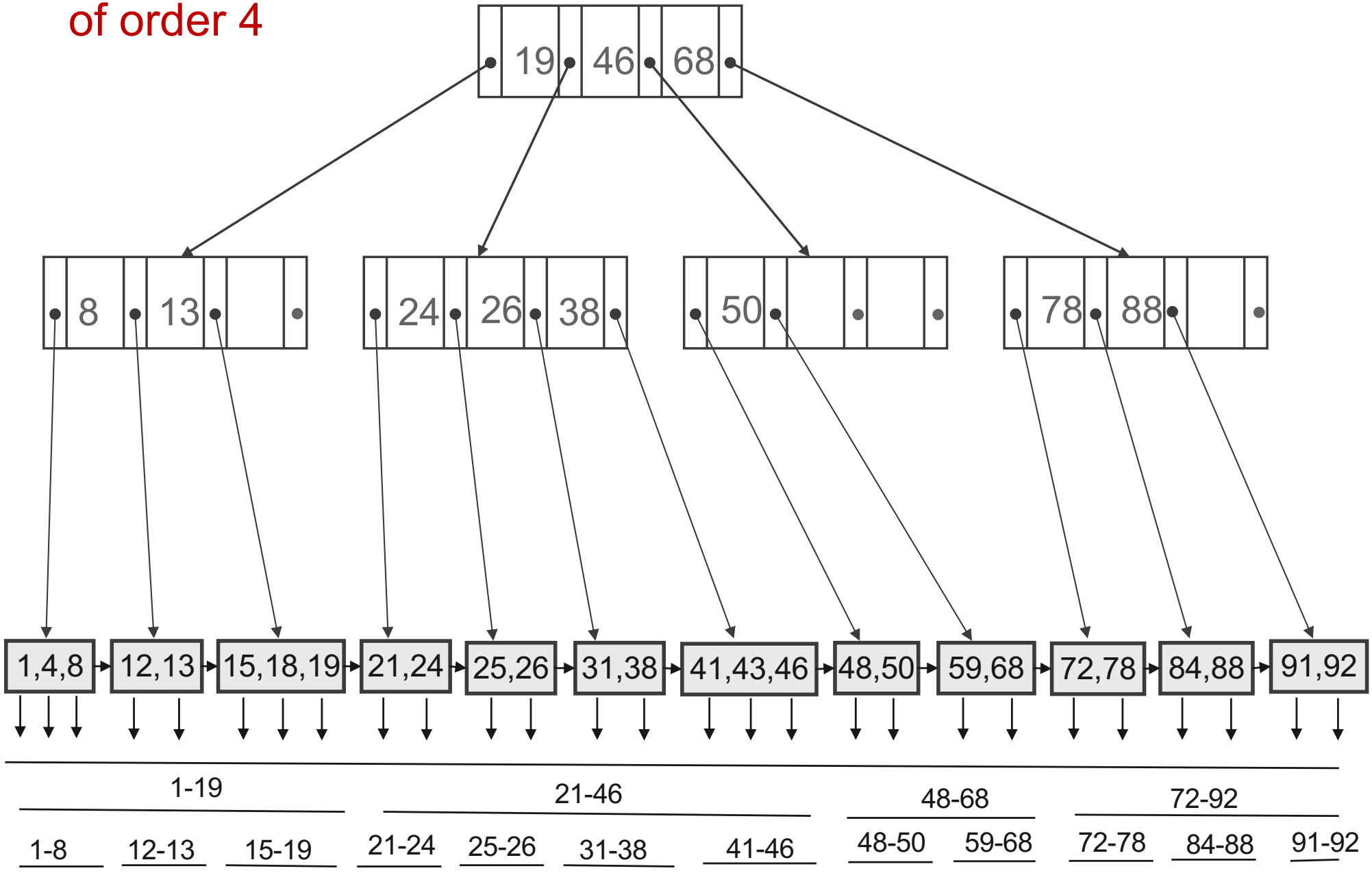

400K
Salary
0
0
100
Age

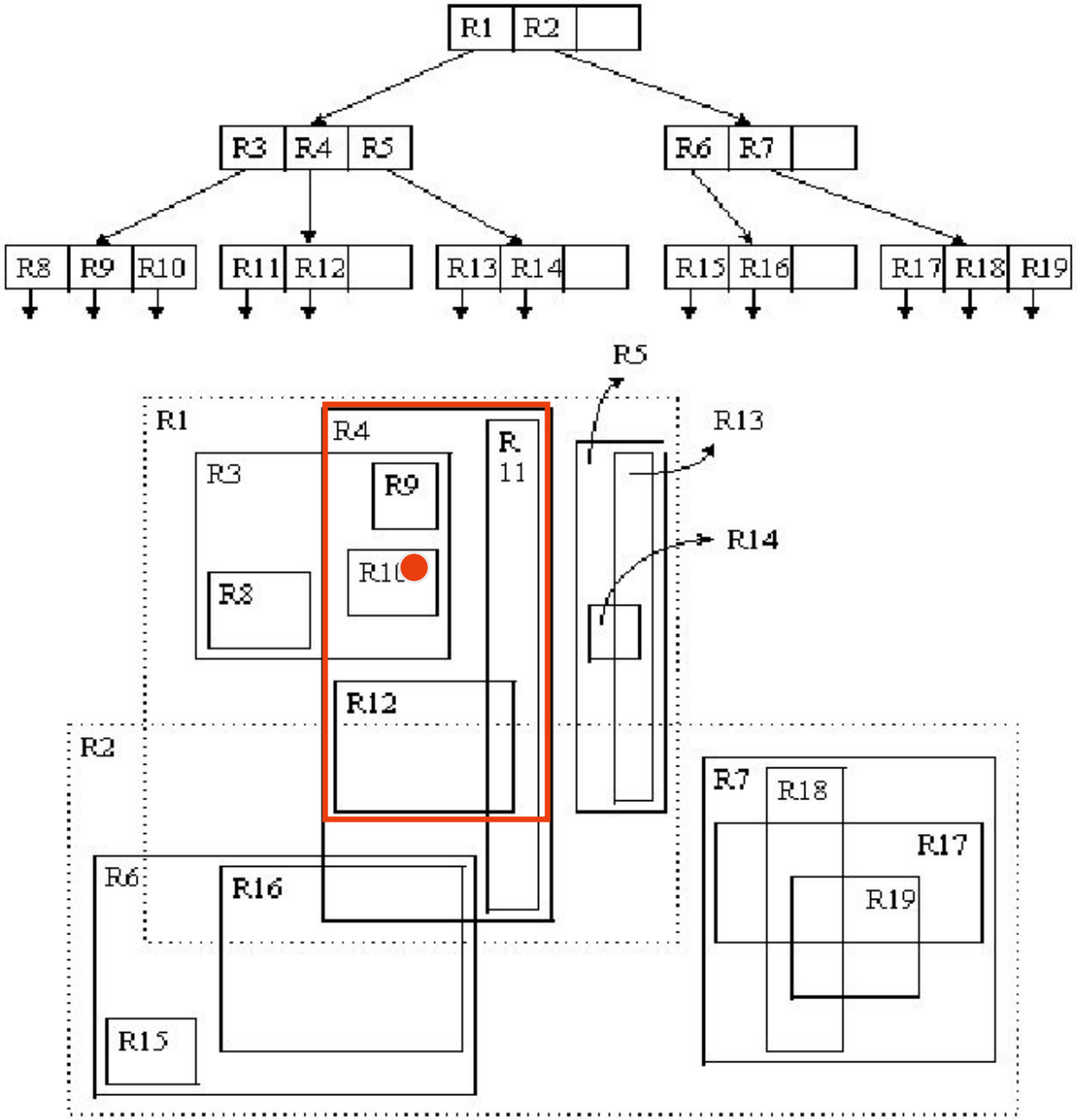

50,200
SW
SE
NE
NW
25,60
45,60
75,100
50,275
60,260
25,300
50,75
50,100
85,140
50,120
70,110
30,260
25,400
45,350

R-trees

- R-tree (region tree)
  - Multidimensional B+-tree, provided by PostgreSQL
  - Balanced multi-way search tree
  - Data records are stored in the leaves while interiors are index nodes
  - Represents data that consists of multidimensional regions
  - Interior node corresponds to some interior region (not data region)
  - Each node has sub-regions that represent contents of its children
  - Sub-regions are allowed to overlap (though it is desirable to keep the overlap small)

B+-Tree of order 4




R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R13
R14
R15
R16
R17
R18
R19

Operations on R-trees

- Where-am-I query
  - Start at the root
  - Recursively search each sub-regions which contains the query point
  - Multiple paths visited from the root
- Region query
  - Start at the root
  - Recursively visit sub-regions which intersect with the query region
  - Multiple paths visited from the root

Operations on R-trees (cont.)

- Overlap region
  - affect search performance
  - Minimize overlap regions
- Insertions
  - Start from the root
  - Insert into
    - the subregions that contains the inserted region

      * What if multiple qualified region ?
    - What if no subregions containing the inserted region
- Overflow: how to split
- Improvement: R+-tree, R*-tree

Content

- Single-level ordered indexes
- Multilevel indexes
- Dynamic multilevel indexes using B-trees and B+-trees
- Multiple dimensional Indexes
- Other types of indexes
- Issues Concerning Indexes

Bitmap Indexes

- Bitmap index
  - For multi-attribute query
  - Built on one particular value of a particular field
  - Index is an array of bots
  - For multi-attribute search query
    - Logical AND operation: AND query
    - Logical OR operation: OR query

record number	*name*	*gender*	*address*	*income_level*
0	John	m	Perryridge	L1
1	Diana	f	Brooklyn	L2
2	Mary	f	Jonestown	L1
3	Peter	m	Brooklyn	L4
4	Kathy	f	Perryridge	L3

Bitmaps for *gender*

m	10010
f	01101

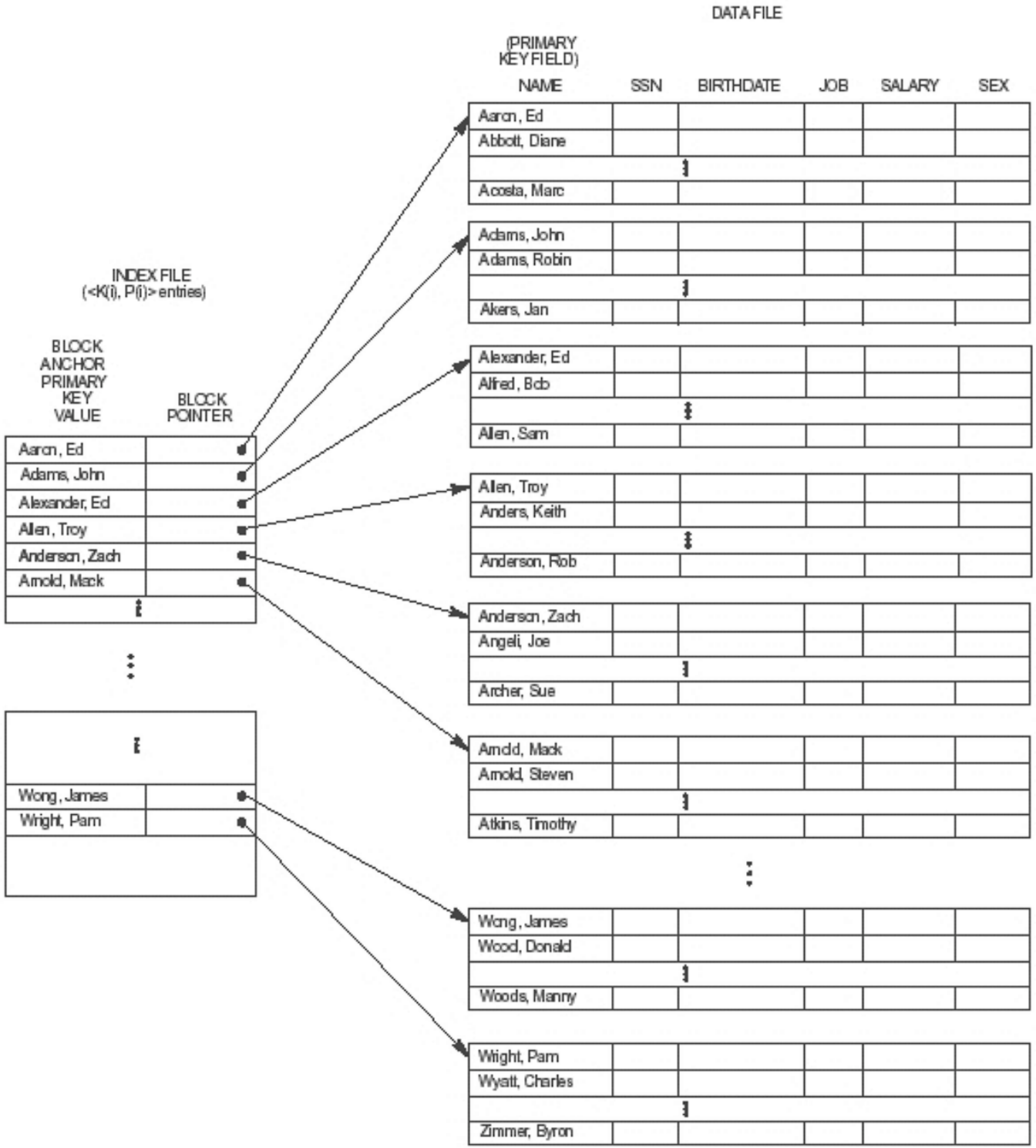
Bitmaps for *income_level*

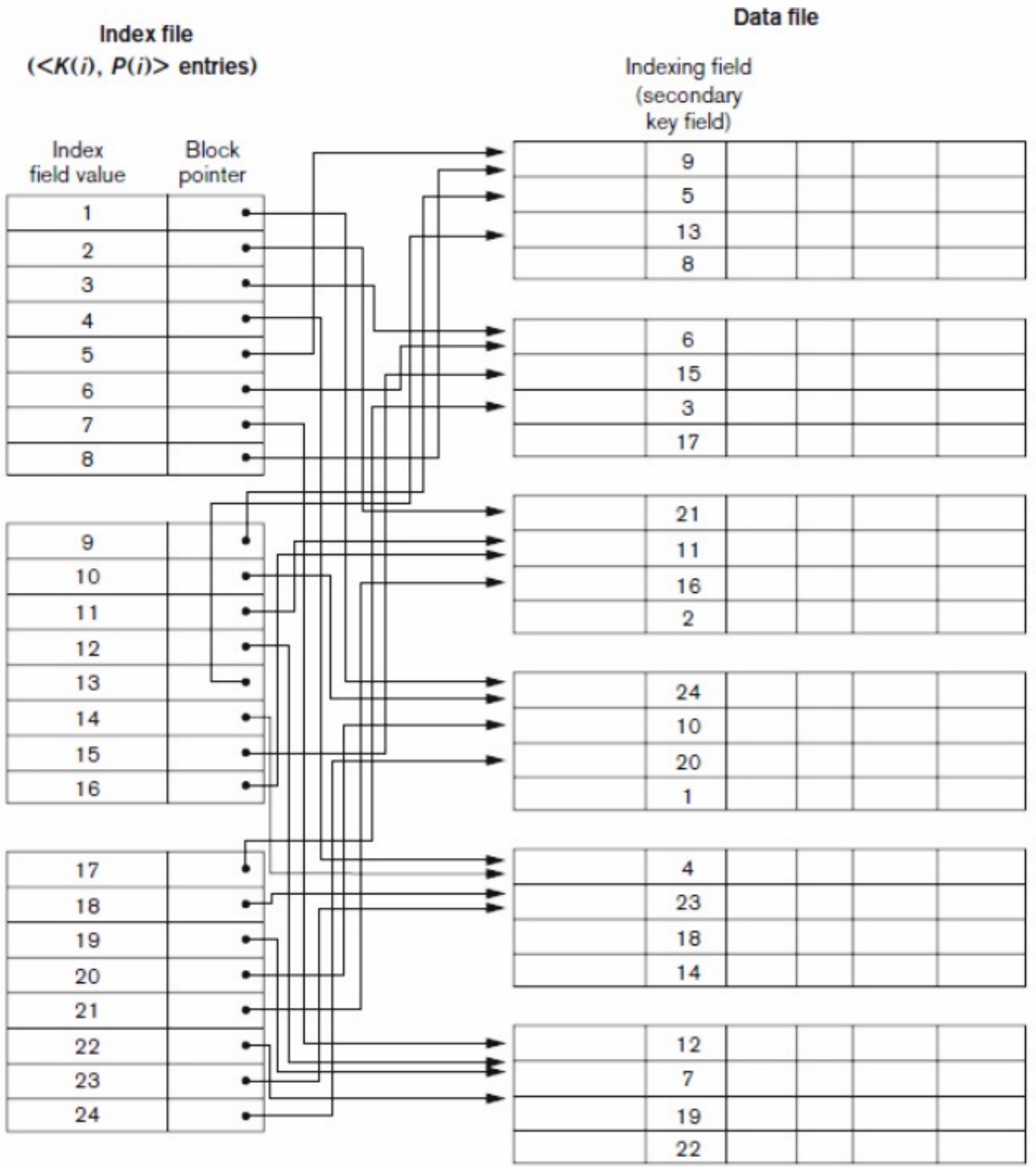
L1	10100
L2	01000
L3	00001
L4	00010
L5	00000

- **Find records where gender is 'f' AND income_level is 'L1'**:
  - Gender 'f': 01101
  - Income_level 'L1': 10100
  - AND operation: 01101 AND 10100 = 00100
  - Result: Record number 2 (Mary)
- **Find records where gender is 'm' OR income_level is 'L3'**:
  - Gender 'm': 10010
  - Income_level 'L3': 00001
  - OR operation: 10010 OR 00001 = 10011
  - Result: Record numbers 0, 3, and 4 (John, Peter, Kathy)

Content

- Single-level ordered indexes
- Multilevel indexes
- Dynamic multilevel indexes using B-trees and B+-trees
- Multiple dimensional Indexes
- Other types of indexes
- Issues Concerning Indexes


DATA FILE
(PRIMARY KEY FIELD)
NAME
SSN
BIRTHDATE
JOB
SALARY
SEX
Aaron, Ed
Abbott, Diane
Acosta, Marc
Adams, John
Adams, Robin
Akers, Jan
Alexander, Ed
Alfred, Bob
Allen, Sam
Allen, Troy
Anders, Keith
Anderson, Rob
Anderson, Zach
Angeli, Joe
Archer, Sue
Arnold, Mack
Arnold, Steven
Atkins, Timothy
Wong, James
Wood, Donald
Woods, Manny
Wright, Pam
Wyatt, Charles
Zimmer, Byron
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
Wong, James
Wright, Pam


Index file
(<K(i), P(i)> entries)
Index field value
Block pointer
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
Data file
Indexing field (secondary key field)
9
5
13
8
6
15
3
17
21
11
16
2
24
10
20
1
4
23
18
14
12
7
19
22

Index Creation

- General form of the command to create an index

```
CREATE [ UNIQUE ] INDEX <index name>
ON <table name> ( <column name> [ <order> ] { , <column name> [ <order> ] } )
[ CLUSTER ] ;
```

  - Unique and cluster keywords optional
  - Order can be ASC or DESC
- Secondary indexes can be created for any primary record organization
  - Complements other primary access methods

Indexing of Strings

- Strings can be variable length
- Strings may be too long, limiting the fan-out
- Prefix compression
  - Stores only the prefix of the search key adequate to distinguish the keys that are being separated and directed to the subtree

Conclusions

- Indexes are access structures that improve efficiency of record retrieval from a data file
- Ordered single-level index types
  - Primary, clustering, and secondary
- Multilevel indexes can be implemented as B-trees and B+ -trees
  - Dynamic structures
- Multiple key access methods