Chapter – 3

File Organization and Indexing

Outline

- Types of Single-level Ordered Indexes
  - Primary Indexes
  - Clustering Indexes
  - Secondary Indexes
- Multilevel Indexes
- Dynamic Multilevel Indexes Using B-Trees and B+-Trees

Introduction

- Indexes
  - Additional auxiliary **access structures -** used to speed up the retrieval of records in response to certain search conditions
  - Enable efficient access to records based on the **indexing fields** that are used to construct the index
  - Any field of the file can be used to create an index, and multiple indexes on different fields-as well as indexes on multiple fields-can be constructed on the same file
  - A variety of indexes are possible; each of them uses a particular data structure to speed up the search
    - To find a record or records in the data file based on a search condition on an indexing field, the index is searched, which leads to pointers to one or more disk blocks in the data file where the required records are located

Indexes as Access Paths

- A single-level index is an auxiliary file that makes it more efficient to search for a record in the data file
- The index is usually specified on one field of the file (although it could be specified on several fields)
- One form of an index is a file of entries **<field value, pointer to record>**, which is ordered by field value
- The index is called an access path on the field

- Index file - usually occupies considerably less disk blocks than the data file because its entries are much smaller
- A binary search on the index yields a pointer to the file record

Indexes as Access Paths

- Indexes can also be characterized as dense or sparse
  - A **dense index** has an index entry for every search key value (and hence every record) in the data file
  - A **sparse (or nondense) index**, on the other hand, has index entries for only some of the search values
- **Example:** Given the following data file EMPLOYEE(NAME, SSN, ADDRESS, JOB, SAL, ...)
- Suppose that
  - Record size R=150 bytes   Block size B=512 bytes   r=30000 records
- Then, we get:
  - blocking factor Bfr= B div R= 512 div 150= 3 records/block
  - number of file blocks b= (r/Bfr)= (30000/3)= 10000 blocks

Indexes as Access Paths

- For an index on the SSN field, assume the field size V_{SSN}=9 bytes, assume the record pointer size P_R=7 bytes. Then:
  - Idex entry size $R_I=(V_{SSN}+P_R)=(9+7)=16$ bytes
  - Index blocking factor Bfr_I= B div R_I= 512 div 16= 32 entries/block
  - Number of index blocks b= (r/ Bfr_I)= (30000/32)= 938 blocks
  - Binary search needs $\log_2 bI = \log_2 938$= 10 block accesses
  - This is compared to an average linear search cost of:
    - (b/2)= 30000/2= 15000 block accesses
  - If the file records are ordered, the binary search cost would be:
    - $\log_2 b = \log_2 30000$= 15 block accesses

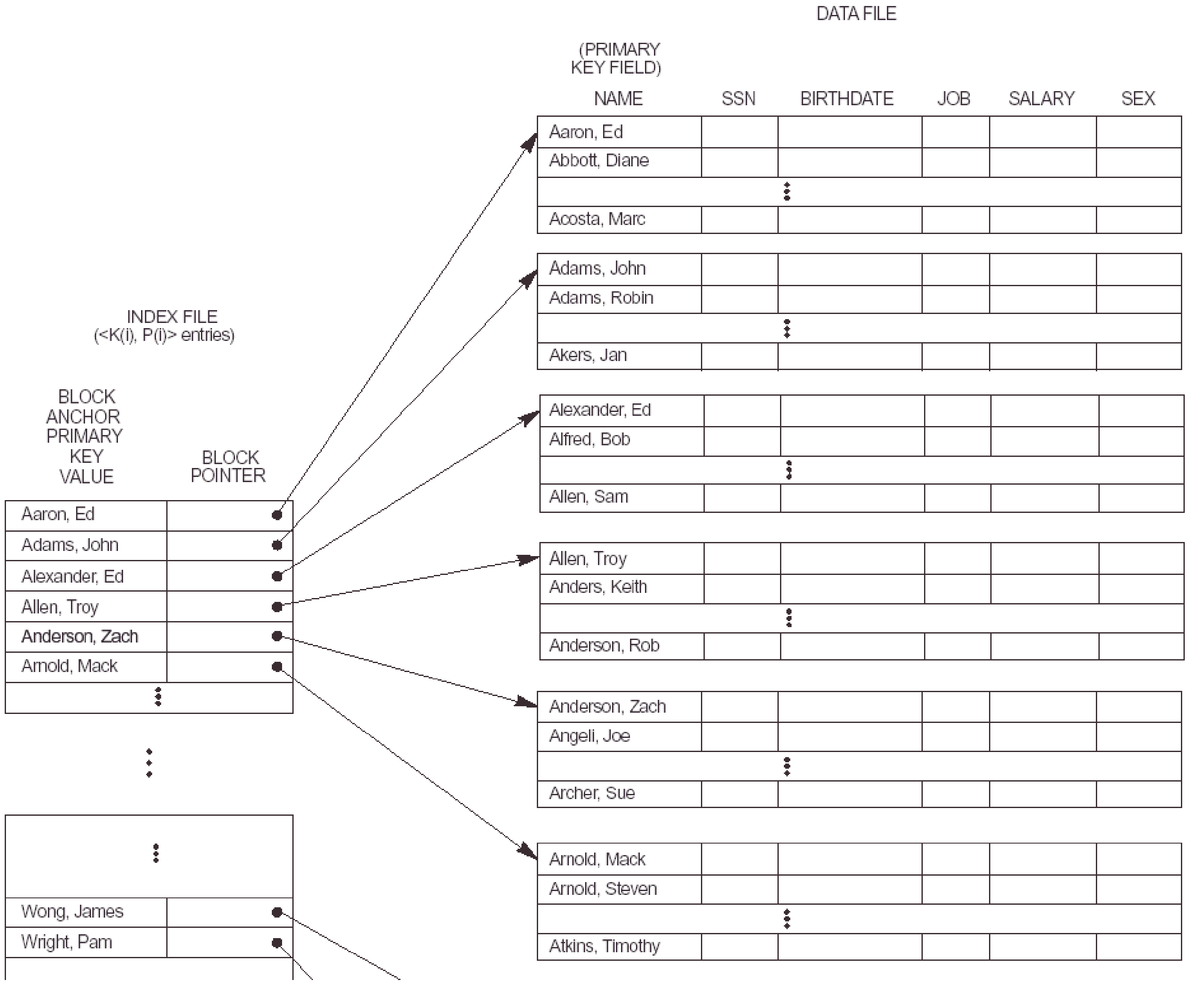
Types of Single-Level Indexes

- Types of Single-level Indexes
  - Primary Indexes
  - Clustering Indexes
  - Secondary Indexes

Types of Single-Level Indexes

- Primary Index
  - Defined on an ordered data file
  - The data file is ordered on a **key field**
  - Includes one index entry *for each block* in the data file
    - The index entry has the **key field value for the *first record*** in the block, which is called the *block* ***anchor***
  - A similar scheme can use the *last record* in a block
  - A primary index is a nondense (sparse) index, since it includes an entry for each disk block of the data file and the keys of its anchor record rather than for every search value
- Each index entry has the value of the **primary key field for the *first* record** in a block and **a pointer to that block** as its two fields : **index entry *i* as <*K*(*i*), *P*(*i*)>**

Types of Single-Level Indexes



Primary Index

Can be built on ordered / sorted files

Index attribute – ordering key field (OKF)

Index Entry:

value of OKF for the first record of a block B_j	disk address of B_j

Index file: ordered file (sorted on OKF)

size-no. of blocks in the data file

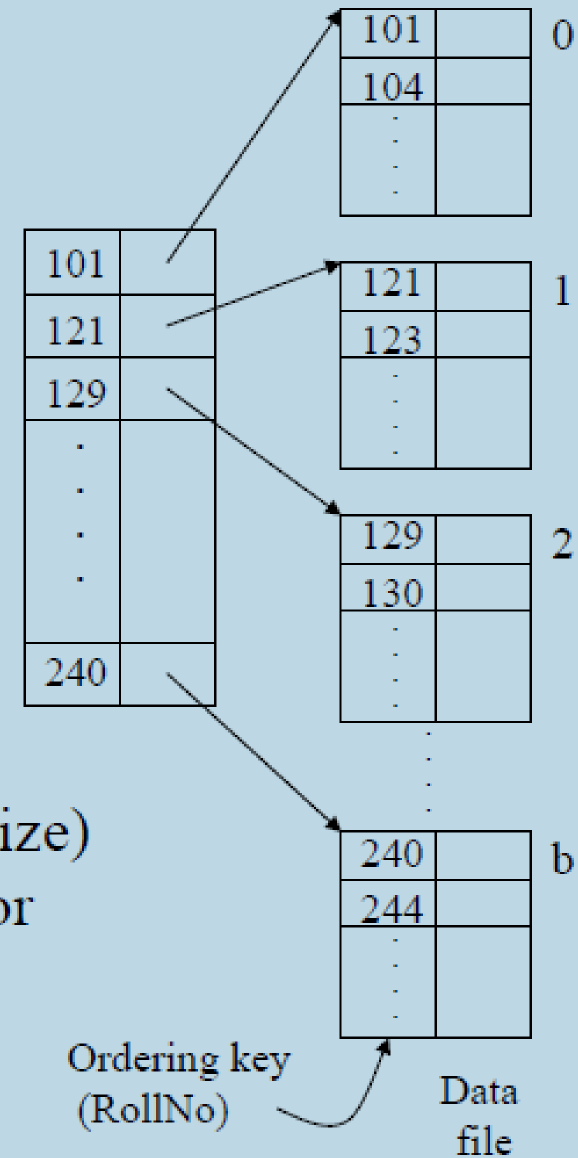
Index file blocking factor $BF_i = \lfloor B/(V+P) \rfloor$

(B-block size, V-OKF size, P-block pointer size)

- generally more than data file blocking factor

No of Index file blocks $b_i = \lceil b/BF_i \rceil$

(b - no. of data file blocks)



An Example

Data file:

- No. of blocks $b = 9500$
- Block size $B = 4KB$
- OKF length $V = 15$ bytes
- Block pointer length $p = 6$ bytes

Index file

- No. of records $r_i = 9500$
- Size of entry $V + P = 21$ bytes
- Blocking factor $BF_i = \lfloor 4096/21 \rfloor = 195$
- No. of blocks $b_i = \lceil r_i/BF_i \rceil = 49$

Max No. of block accesses for getting record using the primary index: $1 + \lceil \log_2 b_i \rceil = 7$

Max No. of block accesses for getting record without using primary index: $\lceil \log_2 b \rceil = 14$

Types of Single-Level Indexes

- Clustering Indexes
  - Defined on an ordered data file
  - The data file is ordered on a *non-key field* unlike primary index, which requires that the ordering field of the data file have a distinct value for each record
  - Includes one index entry *for each distinct value* of the field; the index entry points to the first data block that contains records with that field value.
  - It is another example of *nondense* index where Insertion and Deletion is relatively straightforward with a clustering index

Built on ordered files where ordering field is *not a key*
Index attribute: ordering field (OF)

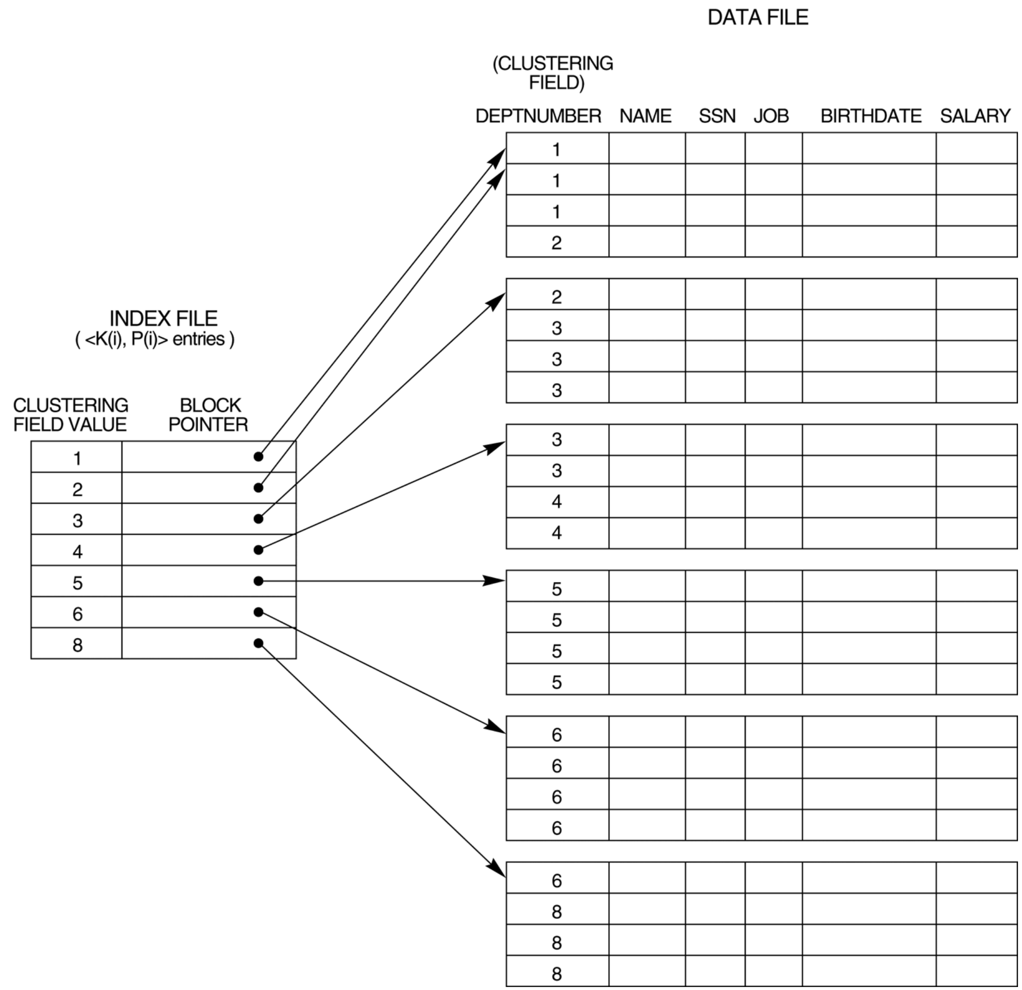
Index entry:

Distinct value V_i of the OF	address of the first block that has a record with OF value V_i

Index file: Ordered file (sorted on OF)
size – no. of distinct values of OF

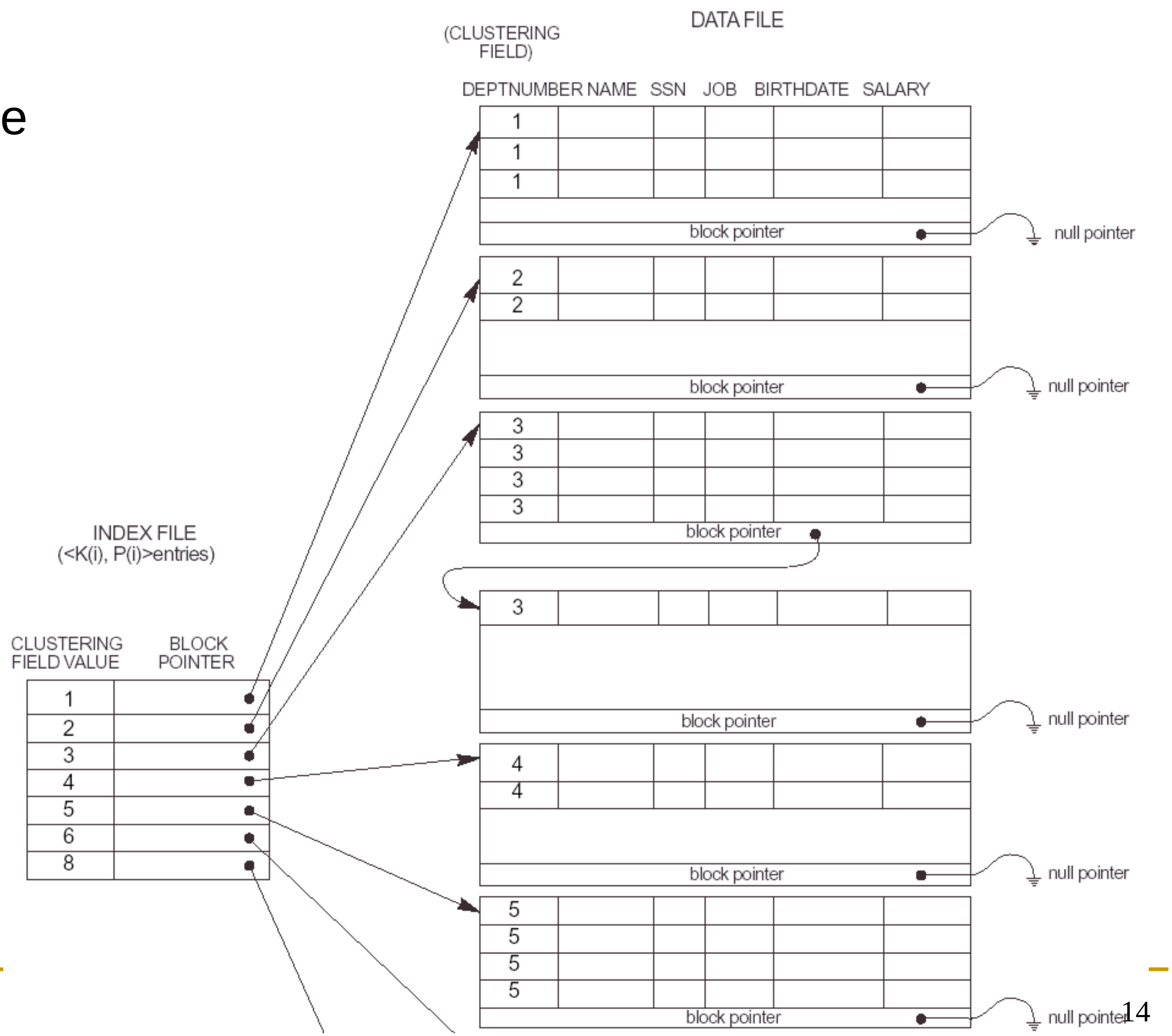
Types of Single-Level Indexes

- Figure : A Clustering index on the DEPTNUMBER ordering non-key field of an EMPLOYEE file



Types of Single-Level Indexes

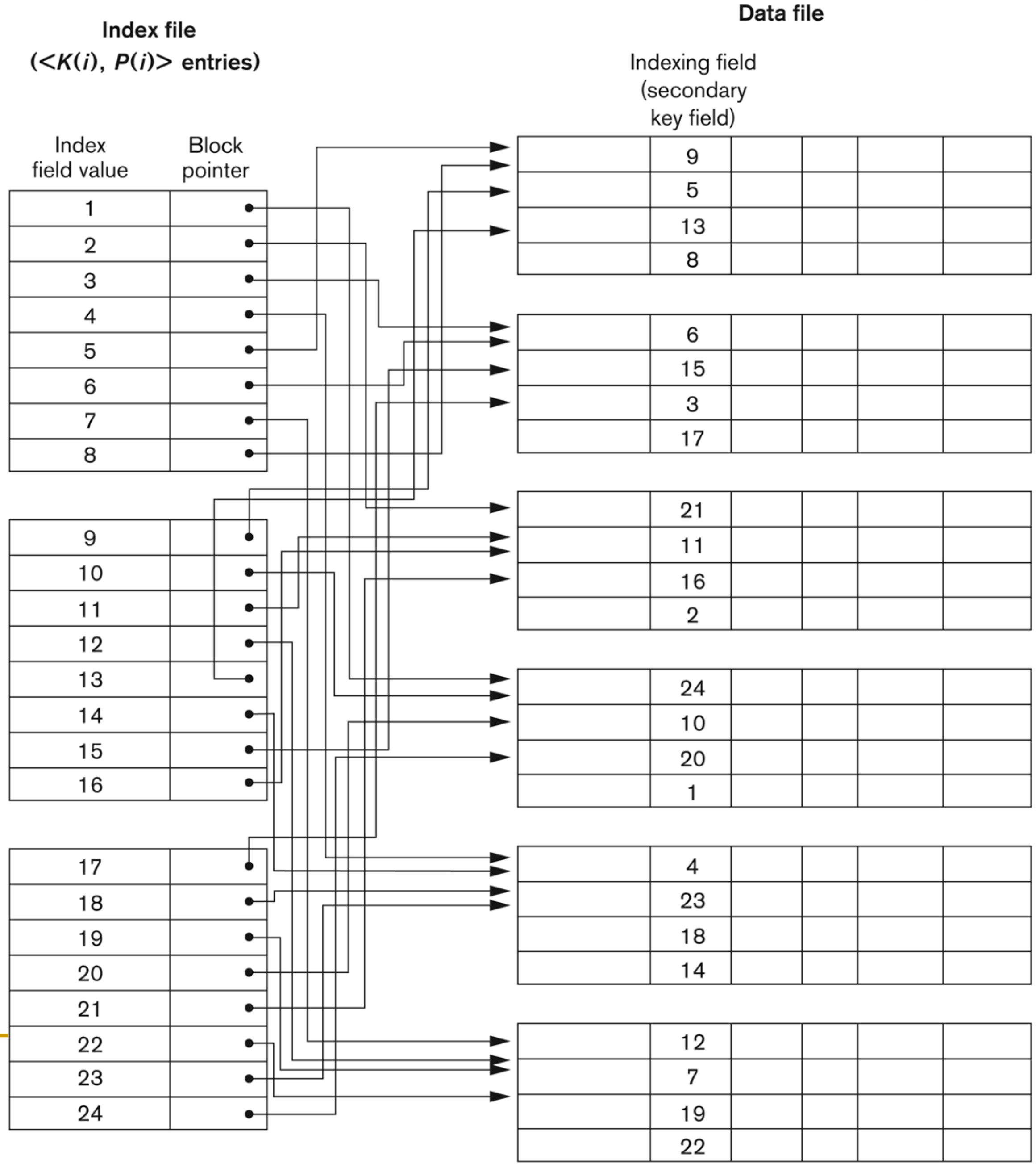
- Another Example



Types of Single-Level Indexes

- Secondary Index
  - Defined on an *unordered* data file
  - The secondary index may be on
    - a key field (with a unique value) in every record, or
    - a non-key with duplicate values
  - **A Secondary index is an ordered file with two fields**
    - The first field is of the same data type as some **non-ordering field** of the data file that is an indexing field
    - The second field is either a **block** pointer or a record pointer
    - There can be *many* secondary indexes (and hence, indexing fields) for the same file
  - Includes one entry *for each record* in the data file; hence, it is a *dense index*

Figure 14.4
A dense secondary index (with block pointers) on a nonordering key field of a file.



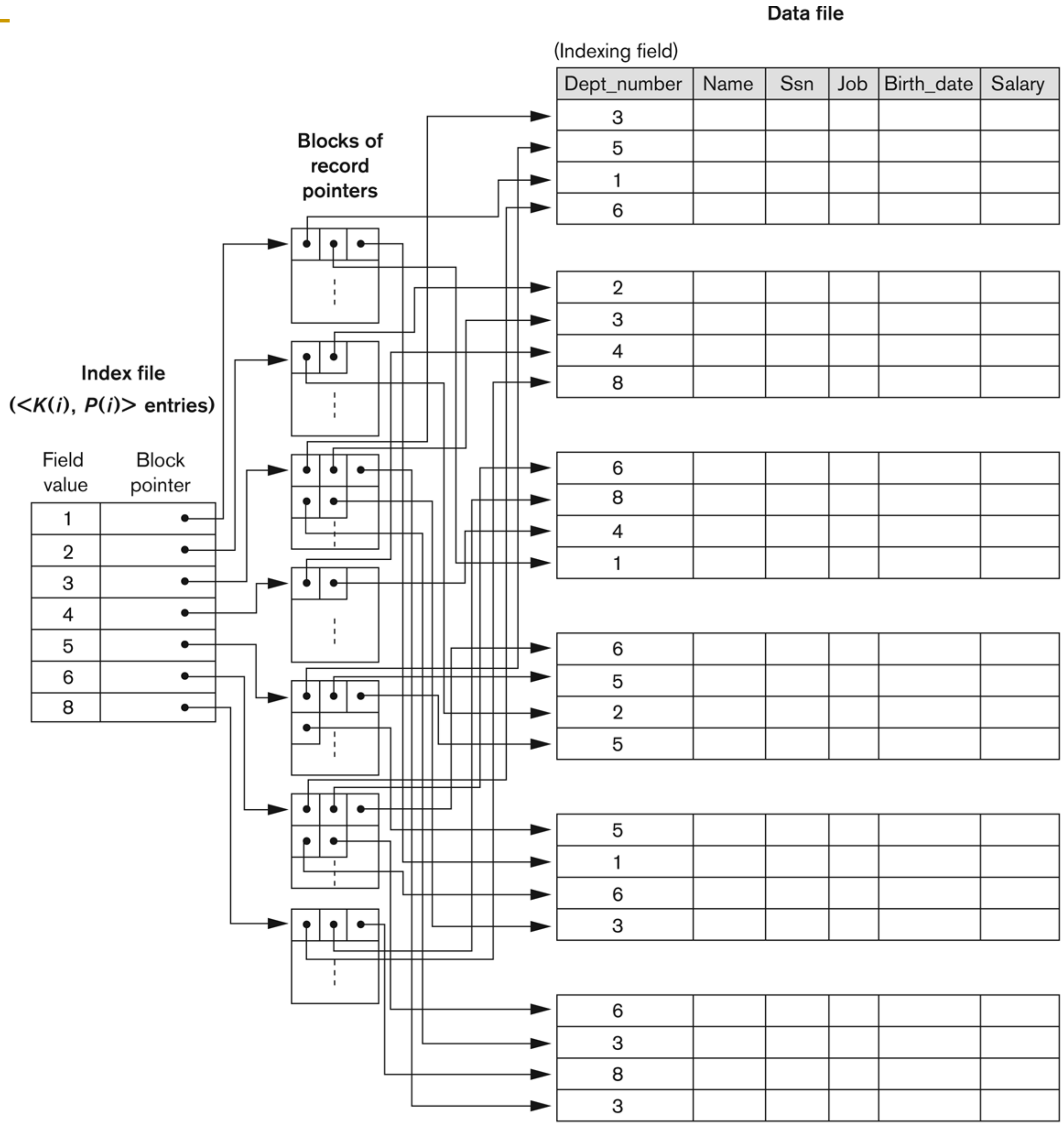


Figure 14.5
A secondary index (with record pointers) on a nonkey field implemented using one level of indirection so that index entries are of fixed length and have unique field values.

Secondary Index (key)

Can be built on ordered and also other type of files

Index attribute: non-ordering key field

Index entry:

value of the NOF V_i	pointer to the *record* with V_i as the NOF value

Index file: ordered file (sorted on NOF values)

No. of entries – same as the no. of *records* in the data file

Index file blocking factor $Bf_i = \lfloor B/(V+P_r) \rfloor$

(B: block size, V: length of the NOF,

P_r: length of a record pointer)

Index file blocks $= \lceil r/Bf_i \rceil$

(r – no. of records in the data file)

An Example

Data file:

No. of records $r = 90{,}000$ Block size $B = 4KB$

Record length $R = 100$ bytes $BF = \lfloor 4096/100 \rfloor = 40$,

$b = \lceil 90000/40 \rceil = 2250$

NOF length $V = 15$ bytes length of a record pointer $P_r = 7$ bytes

Index file :

No. of records $r_i = 90{,}000$ record length $= V + P_r = 22$ bytes

$BF_i = \lfloor 4096/22 \rfloor = 186$ No. of blocks $b_i = \lceil 90000/186 \rceil = 484$

Max no. of block accesses to get a record using the secondary index $1 + \lceil \log_2^{b_i} \rceil = 10$

Avg no. of block accesses to get a record without using the secondary index $b/2 = 1125$

A very significant improvement

Properties of Index Types

Table 18.2 Properties of Index Types

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

Multi-Level Indexes

- Because a single-level index is an ordered file, we can create a primary index *to the index itself*;
  - In this case, the original index file is called the *first-level index* and the index to the index is called the *second-level index*
- We can repeat the process, creating a third, fourth, ..., top level until all entries of the *top level* fit in one disk block
- A multi-level index can be created for any type of first-level index (primary, secondary, clustering) as long as the first-level index consists of *more than one* disk block

A Two-level Primary Index

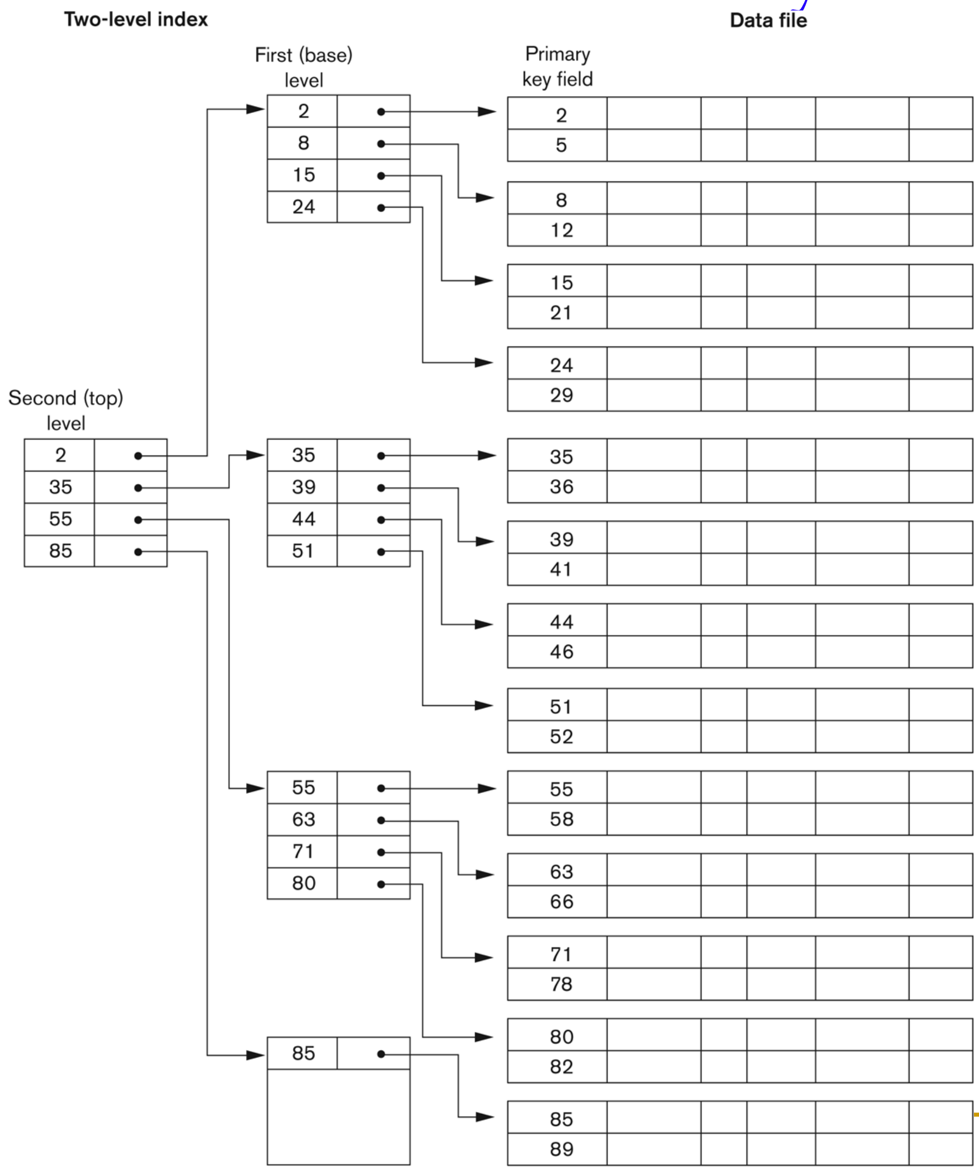


Figure 14.6
A two-level primary index resembling ISAM (Index Sequential Access Method) organization.

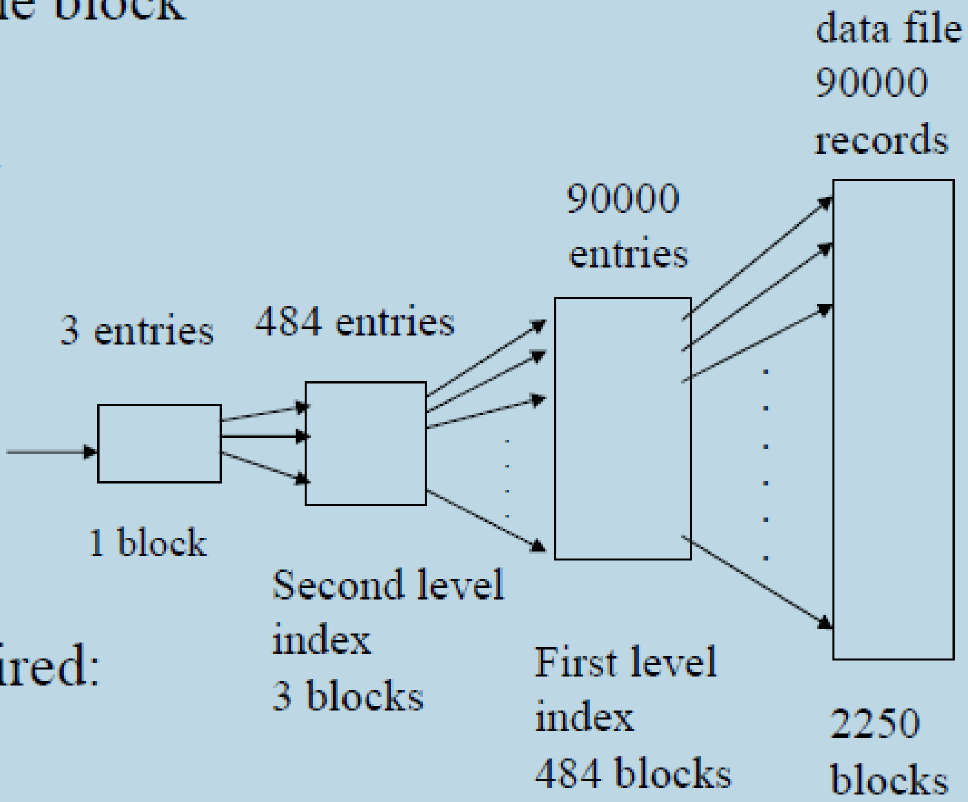
Making the Secondary Index Multi-level

Multilevel Index –

Successive levels of indices are built till the last level has one block

height – no. of levels

block accesses: height + 1



For the example data file:

No of block accesses required:

multi-level index: 4

single level index: 10

Multi-Level Indexes

- Such a multi-level index is a form of *search tree*
  - However, insertion and deletion of new index entries is a severe problem because every level of the index is an *ordered file*

Index Sequential Access Method (ISAM) Files

ISAM files –

Ordered files with a multilevel primary/clustering index

Insertions:

Handled using overflow chains at data file blocks

Deletions:

Handled using deletion markers

Most suitable for files that are relatively static

If the files are dynamic, we need to go for dynamic multi-level index structures based on B^+- trees

Dynamic Multilevel Indexes Using B-Trees and B^+-Trees

- Most multi-level indexes use B-tree or B+-tree data structures because of the insertion and deletion problem
  - This leaves space in each tree node (disk block) to allow for new index entries
- These data structures are variations of search trees that allow efficient insertion and deletion of new search values
- In B-Tree and B+-Tree data structures, each node corresponds to a disk block
- Each node is kept between half-full and completely full

Dynamic Multilevel Indexes Using B-Trees and B^+-Trees

- An insertion into a node that is not full is quite efficient
  - If a node is full the insertion causes a split into two nodes
- Splitting may propagate to other tree levels

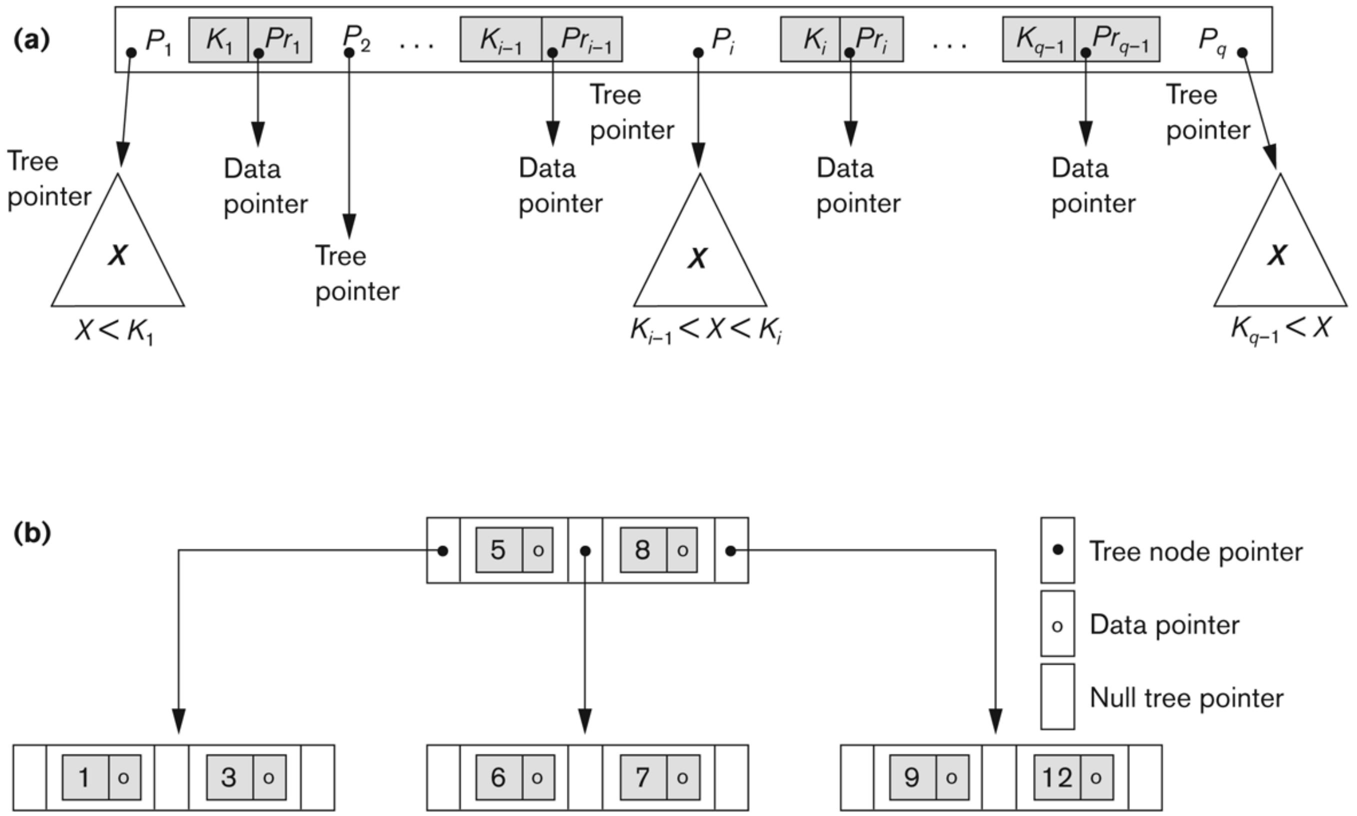
- A deletion is quite efficient if a node does not become less than half full
  - If a deletion causes a node to become less than half full, it must be merged with neighboring nodes

Difference between B-tree and B$^+$-tree

- In a B-tree, pointers to data records exist at all levels of the tree
- In a B+-tree, all pointers to data records exists at the leaf-level nodes
- A B+-tree can have less levels (or higher capacity of search values) than the corresponding B-tree

B-tree Structures

Figure 14.10
B-Tree structures. (a) A node in a B-tree with $q-1$ search values. (b) A B-tree of order $p = 3$.The values were inserted in the order 8, 5, 1, 7, 3, 12, 9, 6.



B^+- trees

- Balanced search trees
  - all leaves are at the same level
- Leaf node entries point to the actual data records
  - all leaf nodes are linked up as a list
- Internal node entries carry only index information
  - In B-trees, internal nodes carry data records also
  - The fan-out in B-trees is less
- Makes sure that blocks are always at least half filled
- Supports both random and sequential access of records

B^+-tree

Order

Order (m) of an Internal Node

- Order of an internal node is the maximum number of tree pointers held in it.
- Maximum of (m-1) keys can be present in an internal node

Order (m_{leaf}) of a Leaf Node

- Order of a leaf node is the maximum number of record pointers held in it. It is equal to the number of keys in a leaf node.

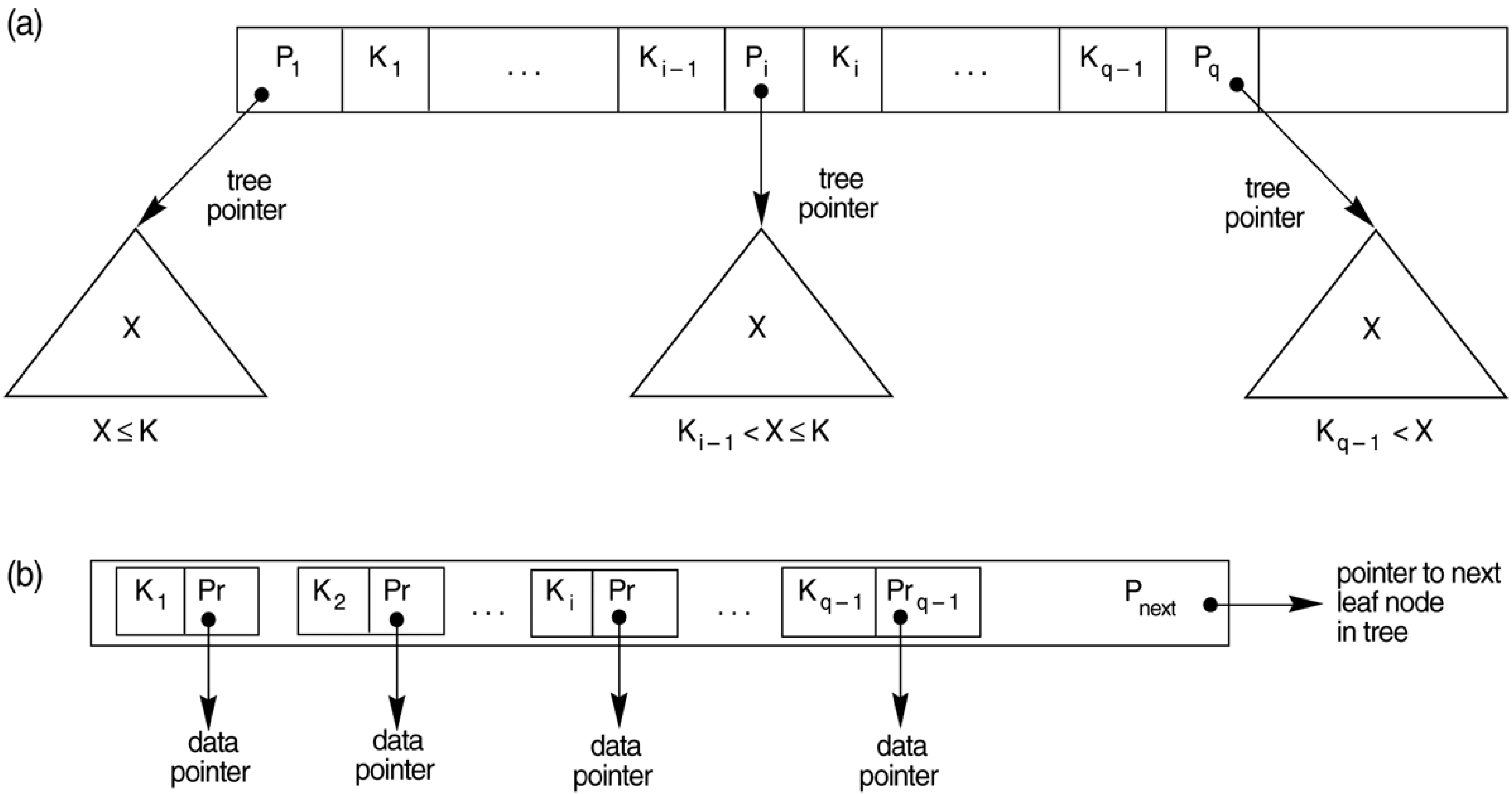
Internal Nodes

An internal node of a B^+- tree of order m:

- It contains at least $\left\lceil \frac{m}{2} \right\rceil$ pointers, except when it is the root node
- It contains at most m pointers.
- If it has $P_1, P_2, \ldots, P_j$ pointers with $K_1 < K_2 < K_3 \ldots < K_{j-1}$ as keys, where $\left\lceil \frac{m}{2} \right\rceil \le j \le m$, then
  - P_1 points to the subtree with records having key value $x \le K_1$
  - P_i $(1 < i < j)$ points to the subtree with records having key value x such that $K_{i-1} < x \le K_i$
  - P_j points to records with key value $x > K_{j-1}$

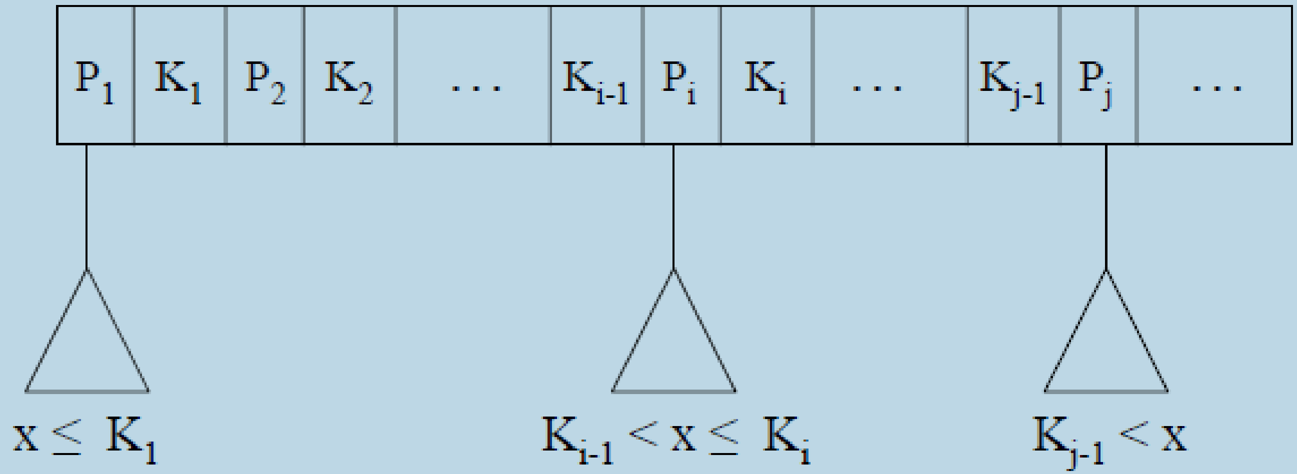
The Nodes of a B$^+$-tree

- FIGURE 14.11 The nodes of a B+-tree
  - (a) Internal node of a B+-tree with q –1 search values.
  - (b) Leaf node of a B+-tree with q – 1 search values and q – 1 data pointers.

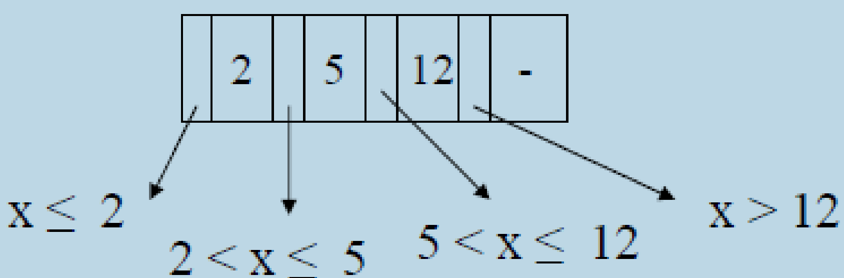


Internal Node Structure

$$\left\lceil \frac{m}{2} \right\rceil \leq j \leq m$$



Example



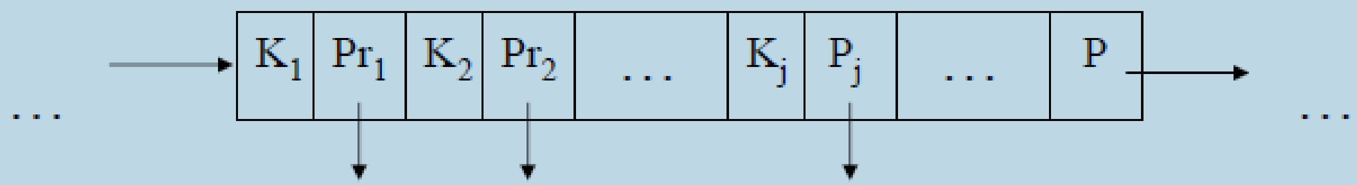
Leaf Node Structure

Structure of leaf node of B^+- of order m_{leaf} :

- It contains one block pointer P to point to next leaf node
- At least $\left\lceil \frac{m_{leaf}}{2} \right\rceil$ record pointers and $\left\lceil \frac{m_{leaf}}{2} \right\rceil$ key values
- At most m_{leaf} record pointers and key values
- If a node has keys $K_1 < K_2 < \ldots < K_j$ with $Pr_1, Pr_2 \ldots Pr_j$ as record pointers and P as block pointer, then

Pr_i points to record with K_i as the search field value, $1 \le i \le j$

P points to next leaf block



Advantages of B^+- trees:

1) Any record can be fetched in equal number of disk accesses.

2) Range queries can be performed easily as leaves are linked up

3) Height of the tree is less as only keys are used for indexing

4) Supports both random and sequential access.

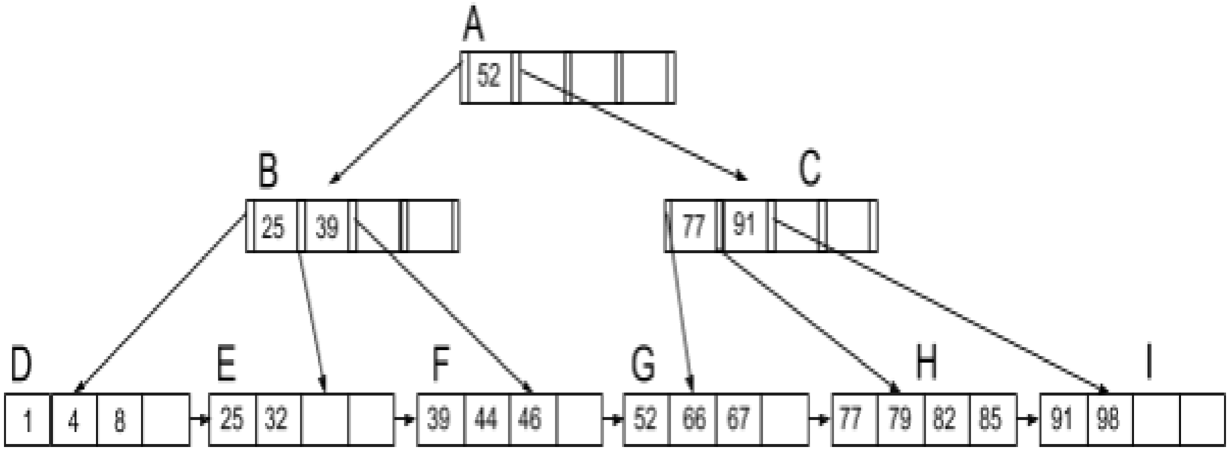
Disadvantages of B^+- trees:

Insert and delete operations are complicated

Root node becomes a *hotspot*

In-class Exercise

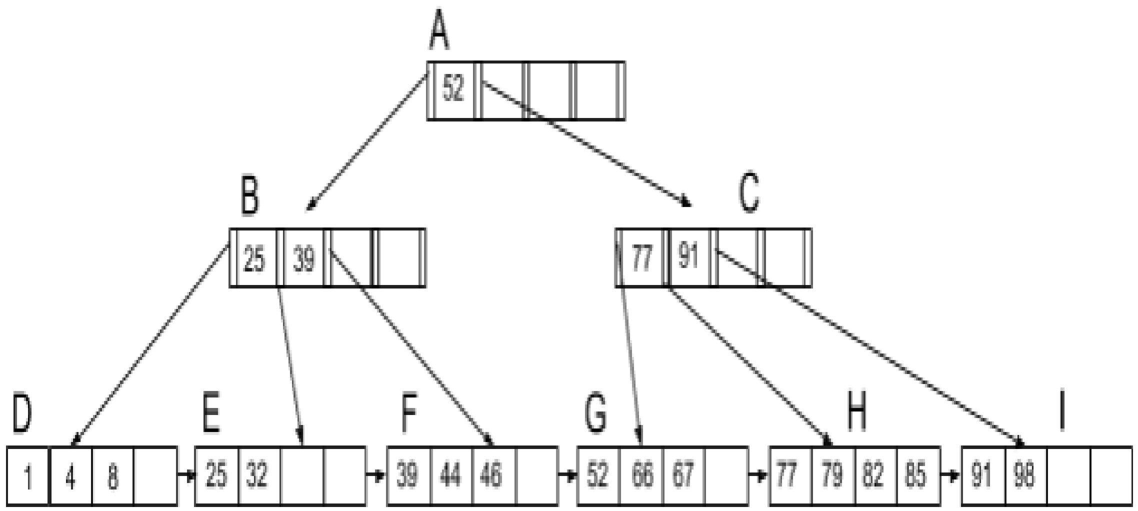
- Consider the following B+ tree index on the "price" field of a relation ***Products(itemID int, price int)***



- **Calculate the number of disk I/Os if B+ tree index is used to answer the following query. Brief your calculation.**

  SELECT * FROM R WHERE PRICE ≥ 39 AND PRICE ≤ 78;

In-class Exercise



- We need a total of 12 I/Os
- 5 index page I/Os: A -> B -> F -> G -> H
- 7 file page I/Os to fetch records: 39, 44, 46, 52, 66, 67, 77