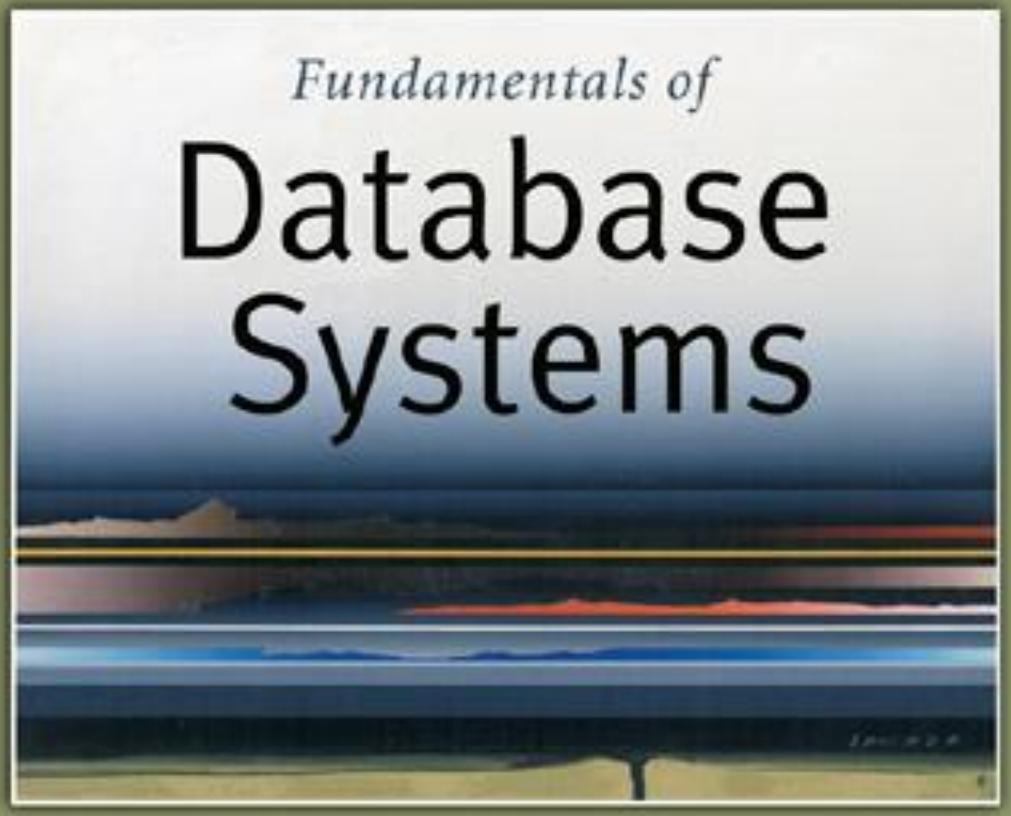
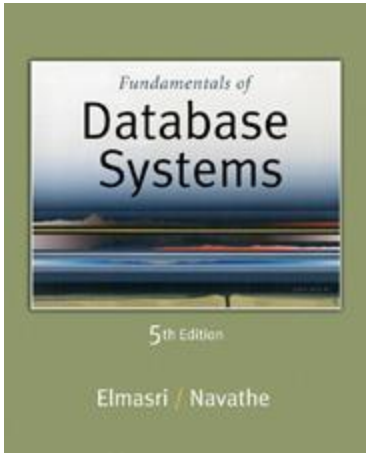


5th Edition

Elmasri / Navathe

Module 4.2

Indexing Structures for Files



Chapter Outline

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

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.

Indexes as Access Paths (contd.)

- The 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 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

Indexes as Access Paths (contd.)

- 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
- 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:
  - index 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

- 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.

Primary index on the ordering key field

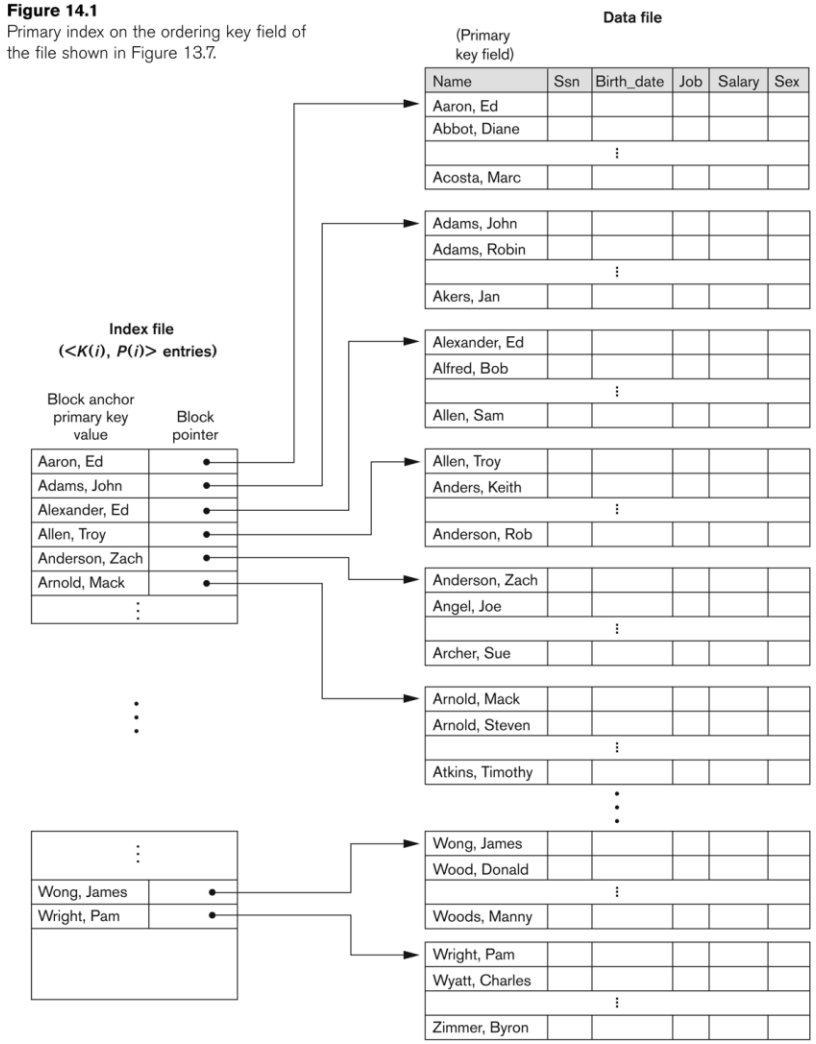


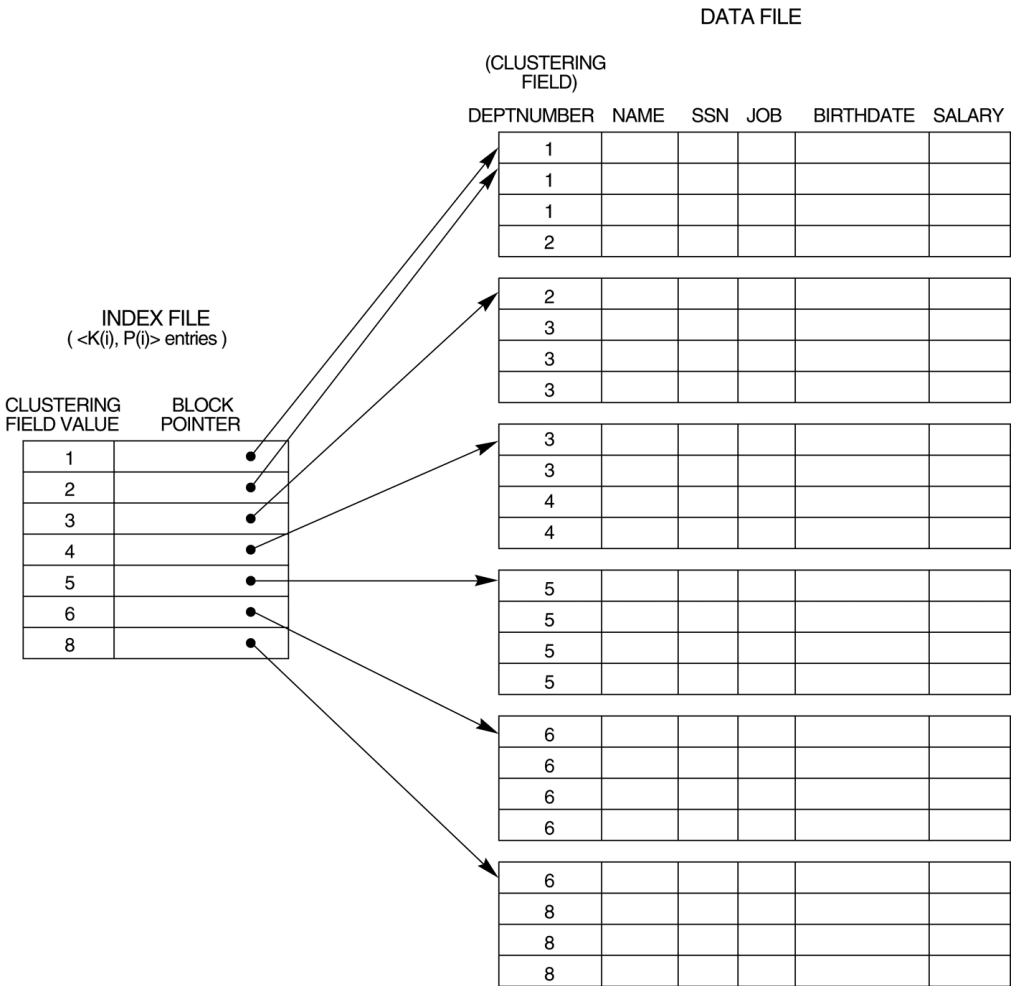
Figure 14.1
Primary index on the ordering key field of the file shown in Figure 13.7.

Types of Single-Level Indexes

- Clustering Index
  - 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.

A Clustering Index Example

- FIGURE 14.2
  A clustering index on the DEPTNUMBER ordering non-key field of an EMPLOYEE file.



Another Clustering Index Example

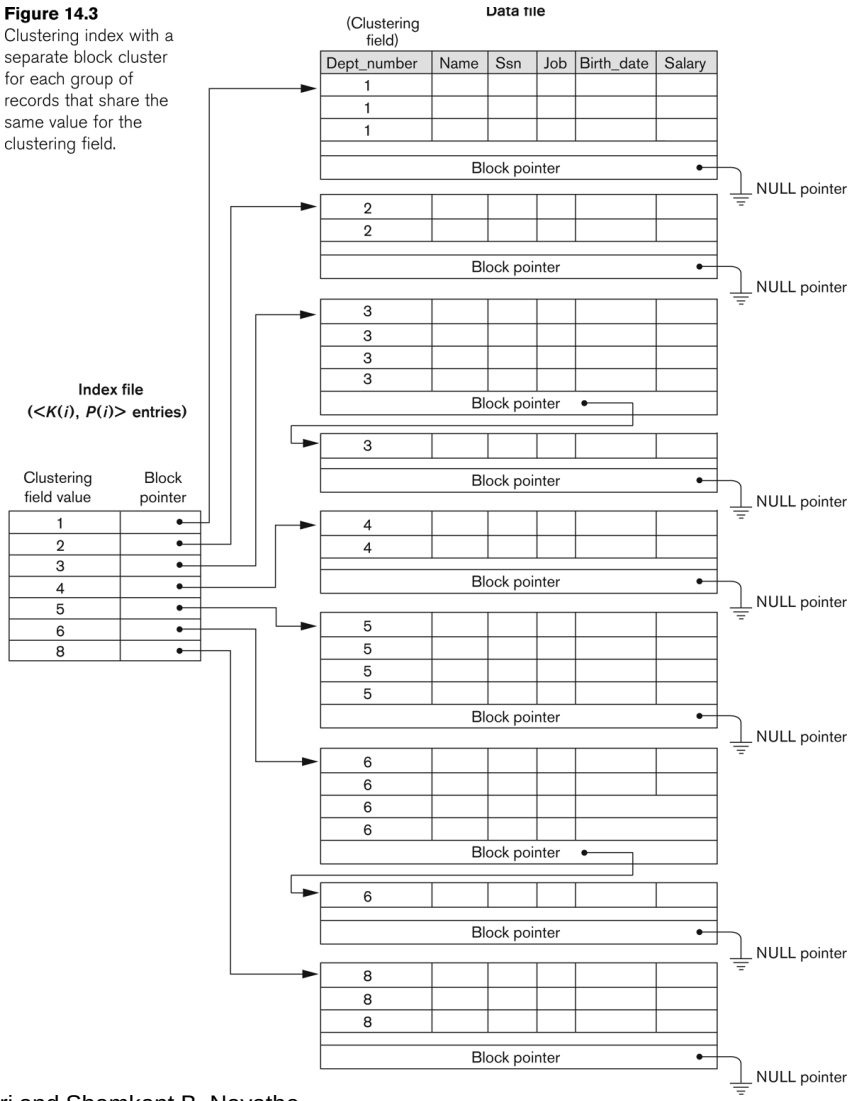


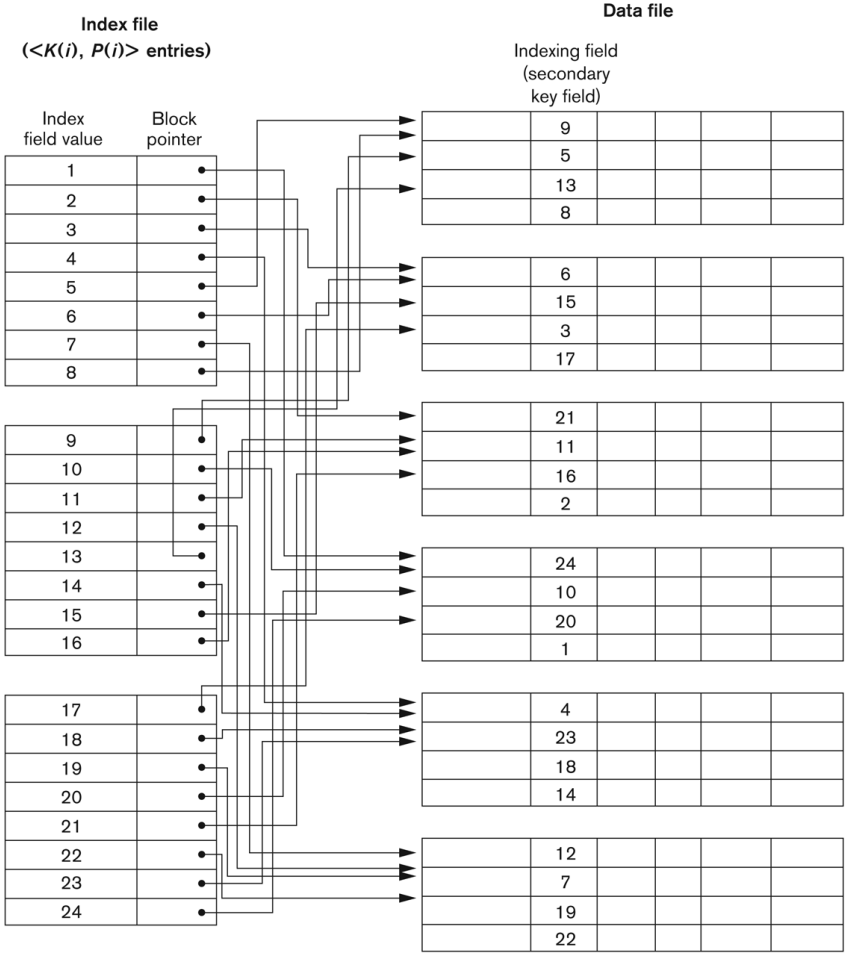
Figure 14.3
Clustering index with a separate block cluster for each group of records that share the same value for the clustering field.

Types of Single-Level Indexes

- Secondary Index
  - A secondary index provides a secondary means of accessing a file for which some primary access already exists.
  - The secondary index may be on a field which is a candidate key and has a unique value in every record, or a non-key with duplicate values.
  - The 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*

Example of a Dense Secondary Index

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



An Example of a Secondary Index

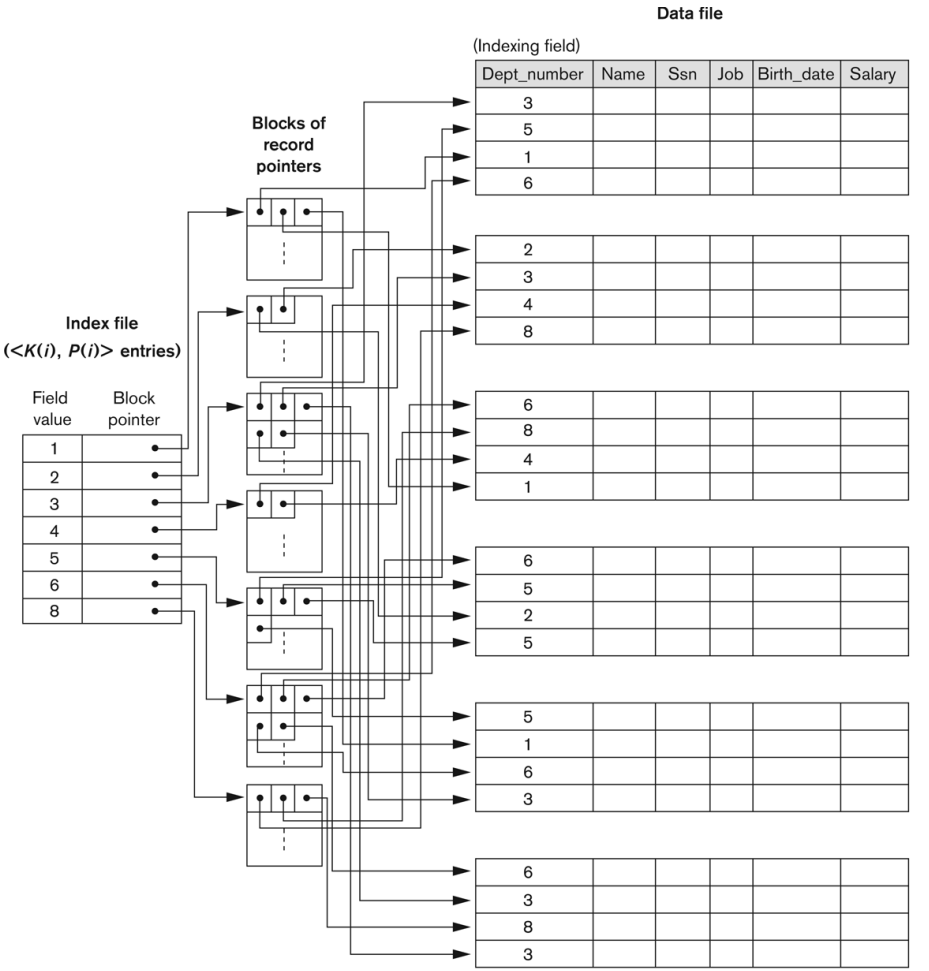


Figure 14.5
A secondary index (with record pointers) on a nonkey field implemented using one level of indirection so that index entries are of fixed length and have unique field values.

Properties of Index Types

TABLE 14.2 PROPERTIES OF INDEX TYPES

TYPE OF INDEX	NUMBER OF (FIRST-LEVEL) INDEX ENTRIES	DENSE OR NONDENSE	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

[a]Yes if every distinct value of the ordering field starts a new block; no otherwise.
[b]For option 1.
[c]For options 2 and 3.

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

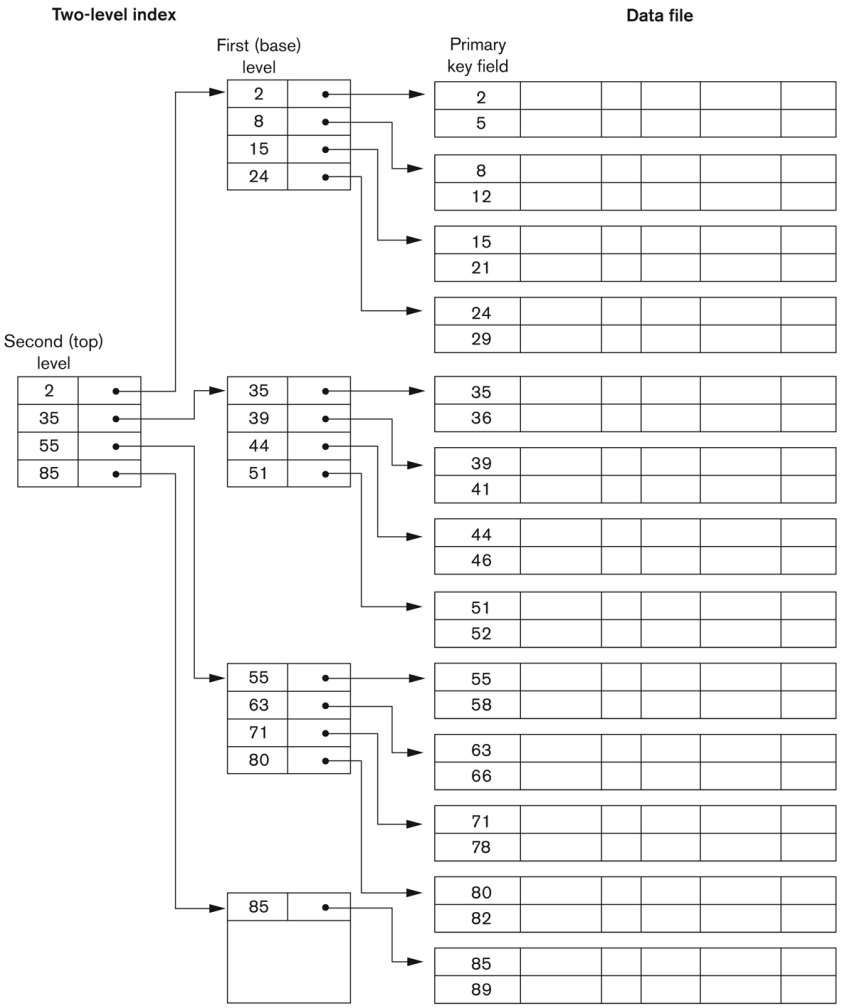


Figure 14.6
A two-level primary index resembling ISAM (Index Sequential Access Method) organization.

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*.

A Node in a Search Tree with Pointers to Subtrees below It

- FIGURE 14.8

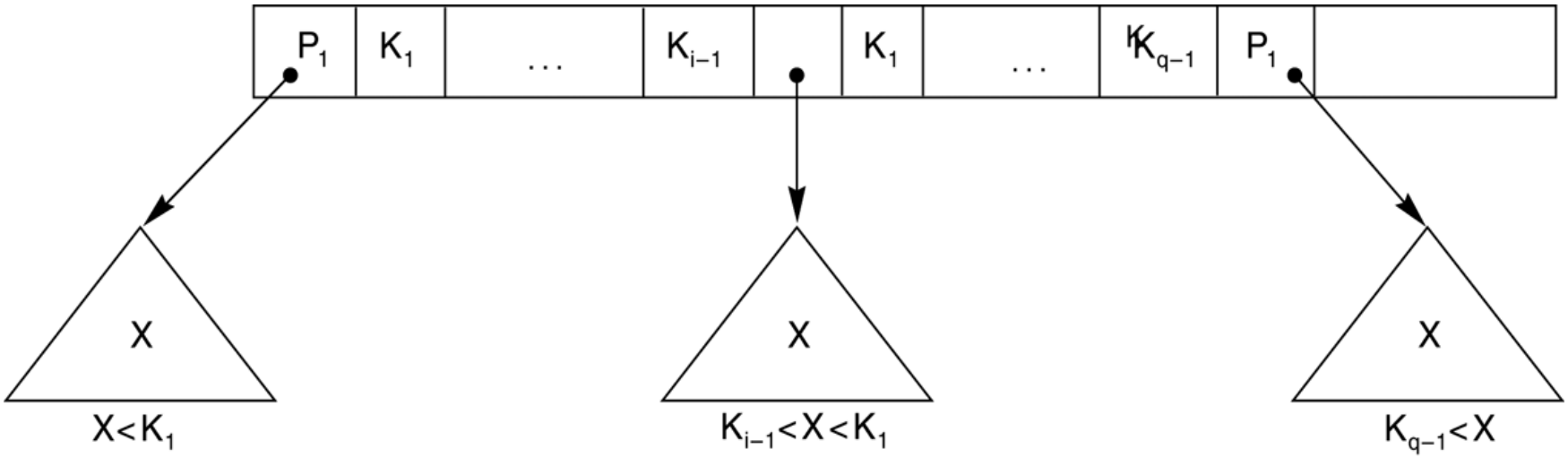
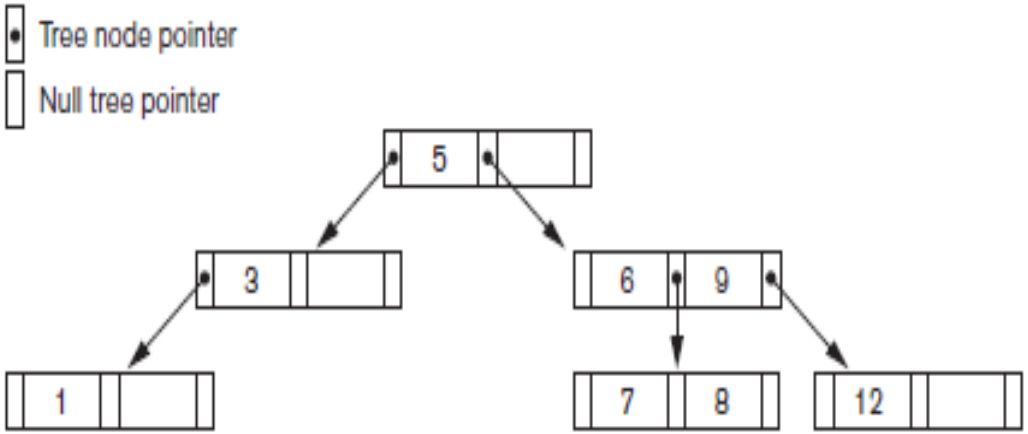


FIGURE 14.9
A search tree of order p = 3.



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 (contd.)

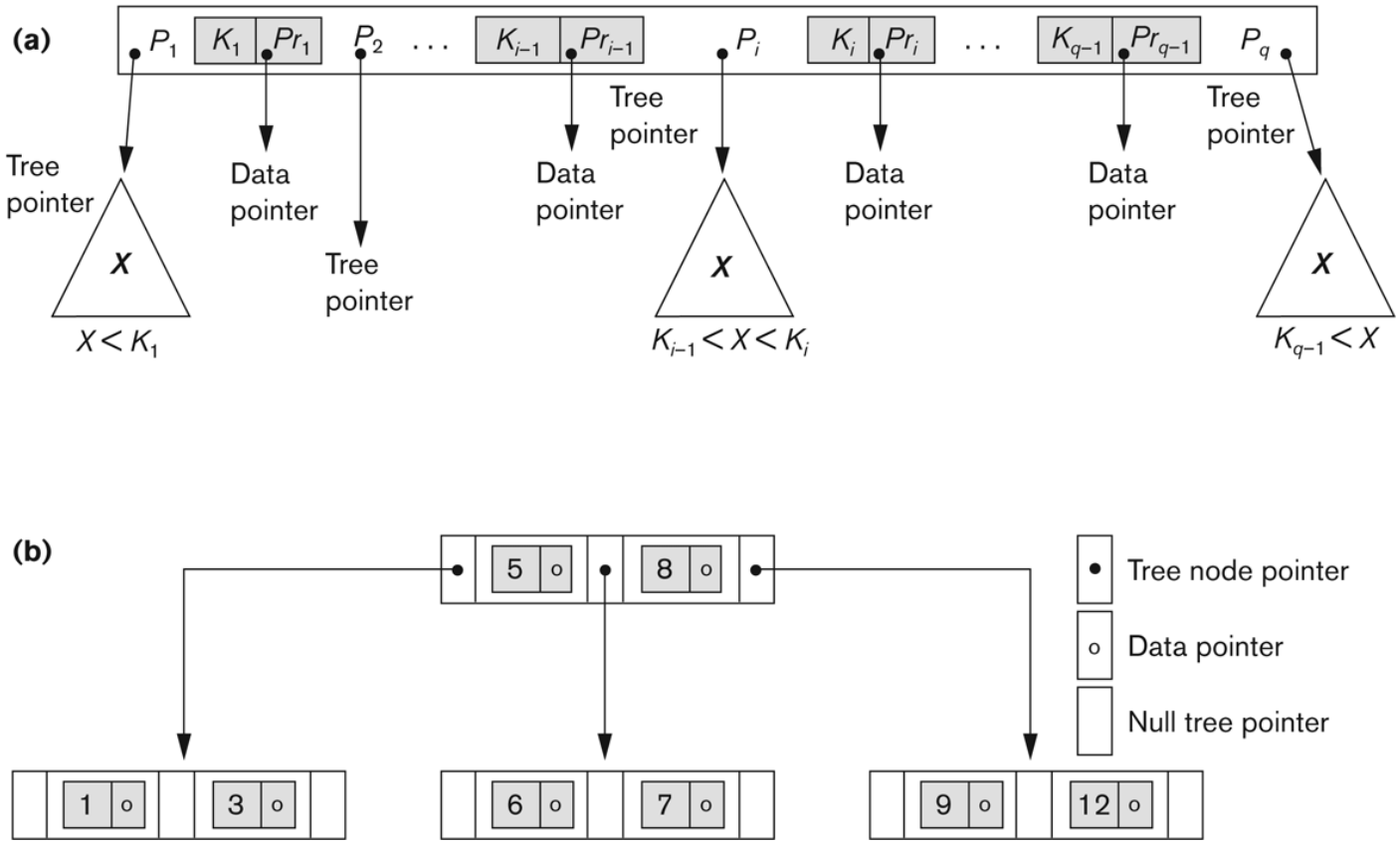
- An insertion into a node that is not full is quite efficient
  - If a node is full the insertion causes a split into two nodes
- Splitting may propagate to other tree levels
- A deletion is quite efficient if a node does not become less than half full
- If a deletion causes a node to become less than 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

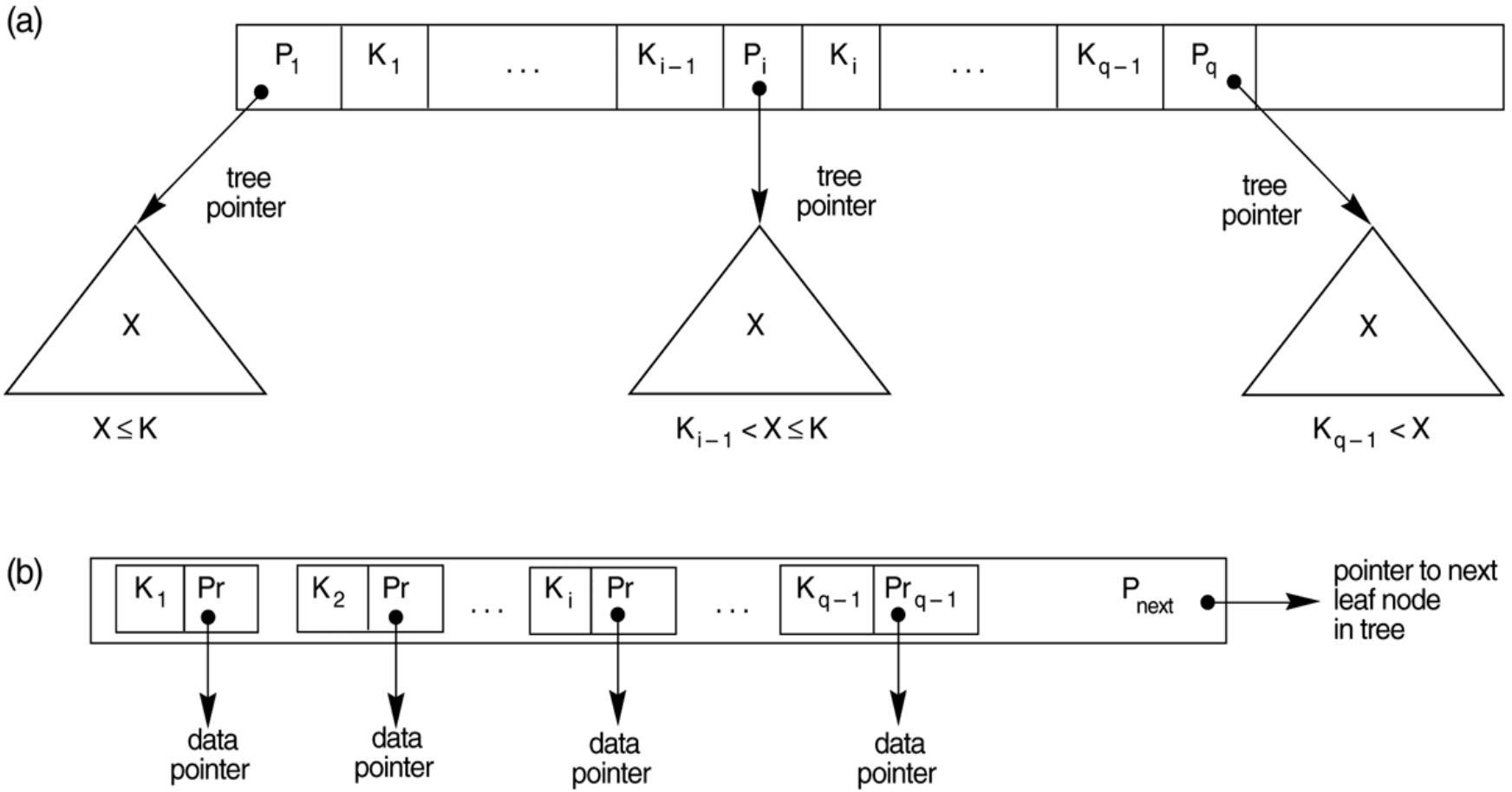
B-tree Structures

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.



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.



B^+- trees

Bayer & McCreight
Acta Informatica 1972

- Balanced search trees (self-balancing)
  - Internal nodes have variable number of children
  - All leaves are at the same level
  - Nodes – internal or leaf – are disk blocks
- 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 record pointers also
  - The fan-out in B-trees is less
- Make sure that blocks are always at least half filled
- Support both random and sequential access of records

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.

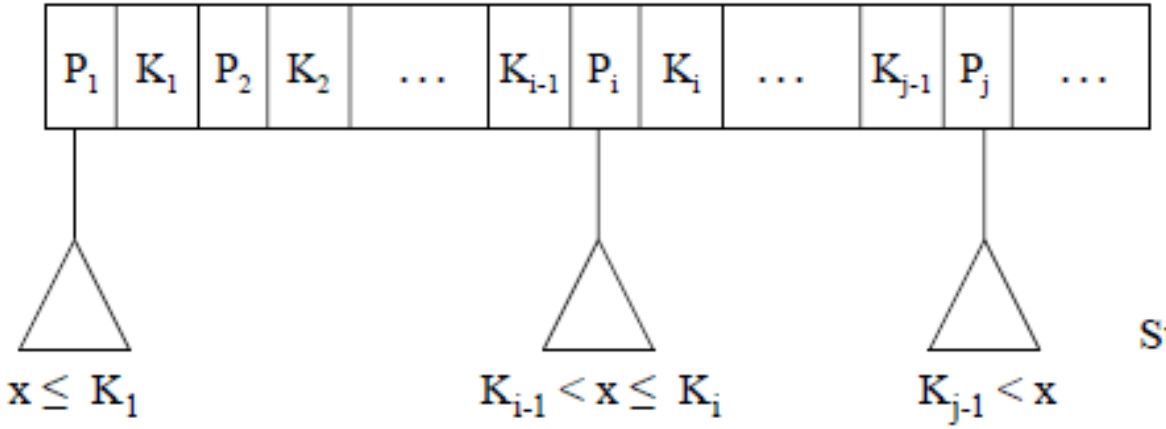
Internal Node Structure

P_i: Tree pointer (Block pointer)

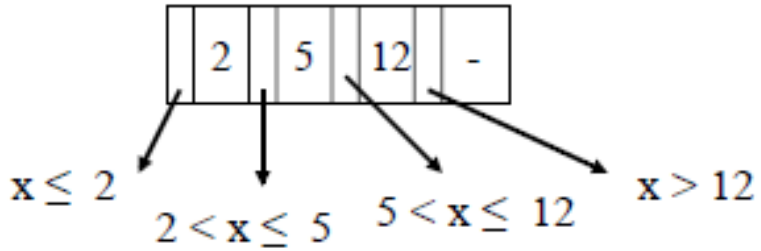
K_i: Key value

m : Order(internal)

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



Example



Internal Nodes

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

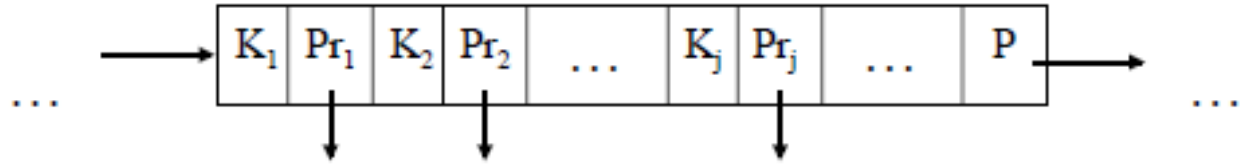
- It contains at least $\lceil \frac{m}{2} \rceil$ pointers, except when it is the root node (Root nodes – a min of 2 pointers is ok)
- It contains at most m pointers.
- If it has $P_1, P_2, \ldots, P_j$ pointers with $K_1 < K_2 < K_3 \ldots < K_{j-1}$ as keys, where $\lceil \frac{m}{2} \rceil \leq j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

Leaf Node Structure

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

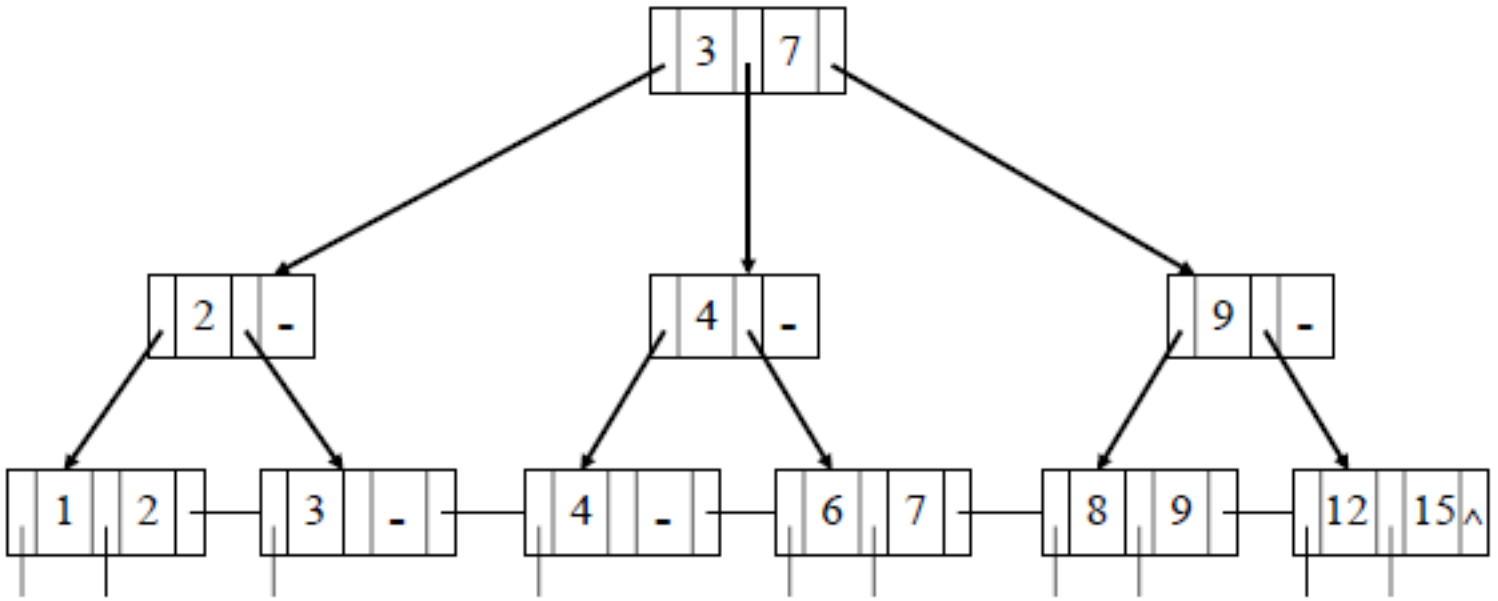
- It contains one block pointer P to point to next leaf node
- At least $\left\lceil \frac{m_{leaf}}{2} \right\rceil$ record pointers and $\left\lceil \frac{m_{leaf}}{2} \right\rceil$ key values
- At most m_{leaf} record pointers and key values
- If a node has keys $K_1 < K_2 < \ldots < K_j$ with $Pr_1,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

Pr_i points to record with K_i as the search field value, $1 \leq i \leq j$
P points to next leaf block



Example B^+- tree

$m = 3 \quad m_{leaf} = 2$



Insertion into B^+- trees

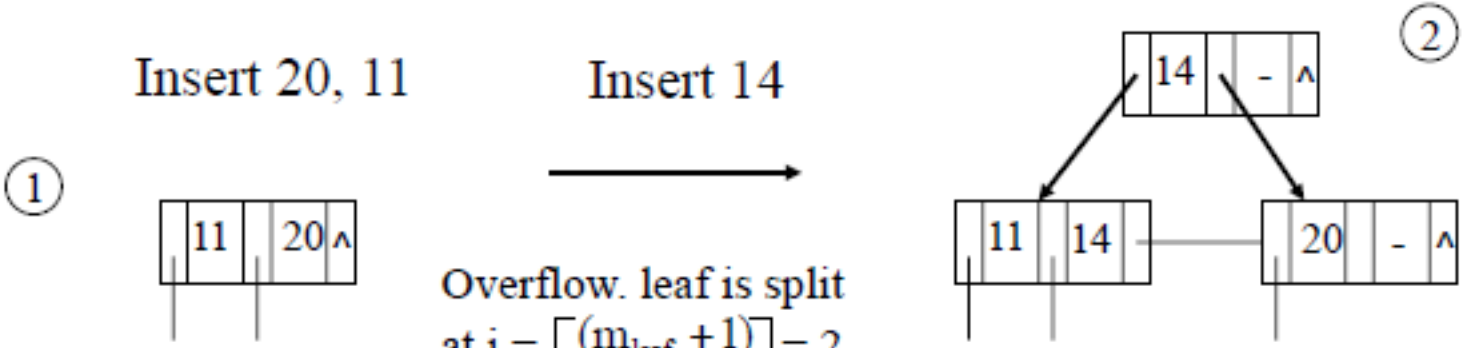
Every (key, record pointer) pair is inserted in an appropriate leaf (Search for it)

- If a leaf node overflows:
  - Node is split at $j = \left\lceil \frac{(m_{leaf} + 1)}{2} \right\rceil$
  - First j entries are kept in original node
  - Entities from j+1 are moved to new node
  - j^{th} key value K_j is *replicated* in the parent of the leaf.
- If an internal node overflows:
  - Node is split at $j = \left\lfloor \frac{(m + 1)}{2} \right\rfloor$
  - Values and pointers up to P_j are kept in the original node
  - j^{th} key value K_j is *moved* to the parent of the internal node
  - P_{j+1} and the rest of entries are moved to a new node.

Example of Insertions $m = 3$ $m_{leaf} = 2$

Insert 20, 11

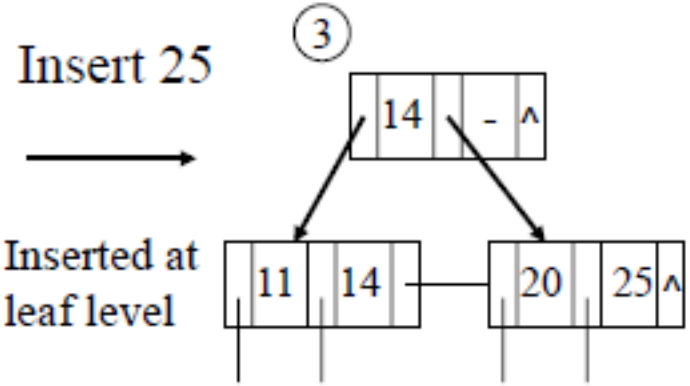
Insert 14



Overflow. leaf is split at $j = \left\lceil \frac{(m_{leaf} + 1)}{2} \right\rceil = 2$

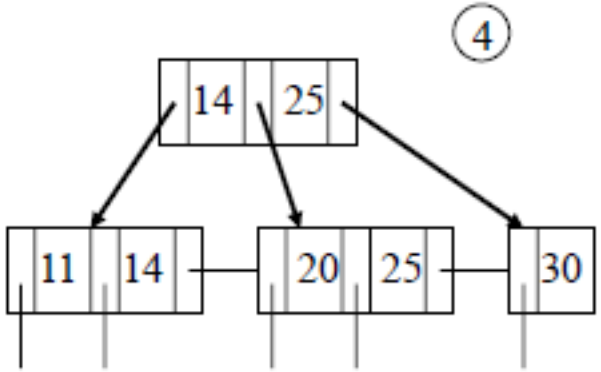
14 is copied to parent level

Insert 25



Inserted at leaf level

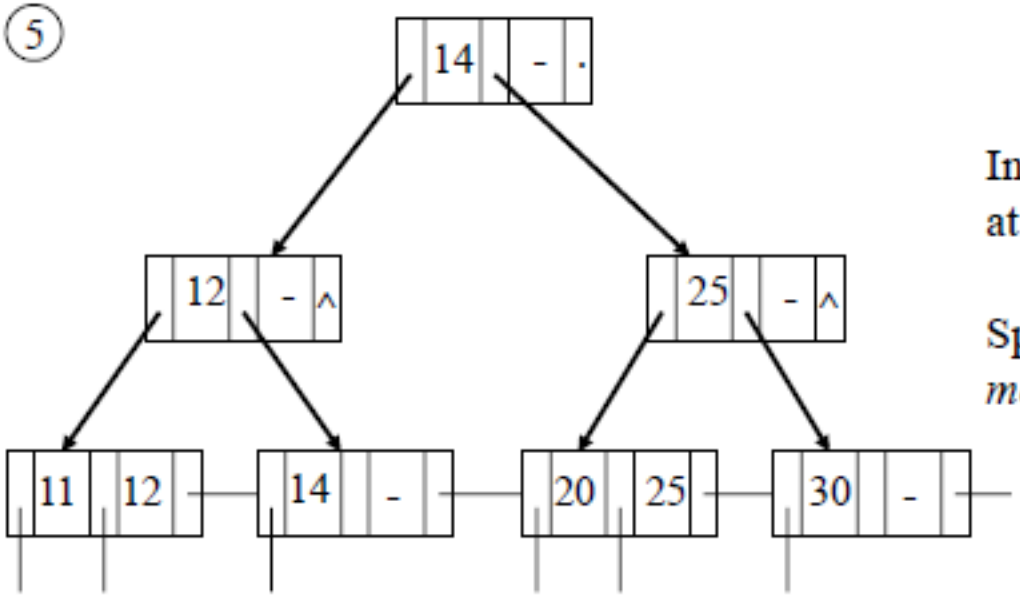
Insert 30



Overflow. split at 25. 25 is copied to upper level

Insert 12 Overflow at leaf level.

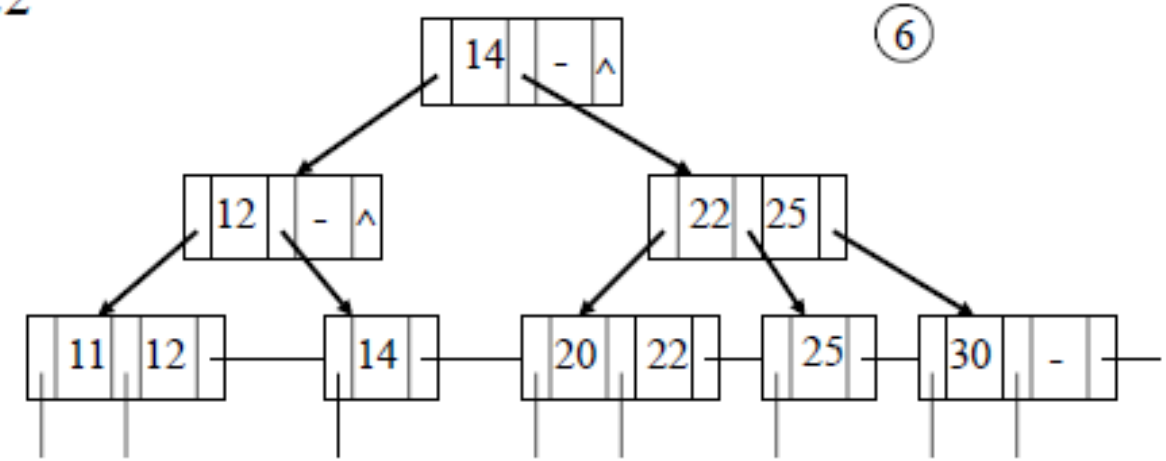
- Split at leaf level,
- Triggers overflow at internal node
- Split occurs at internal node;



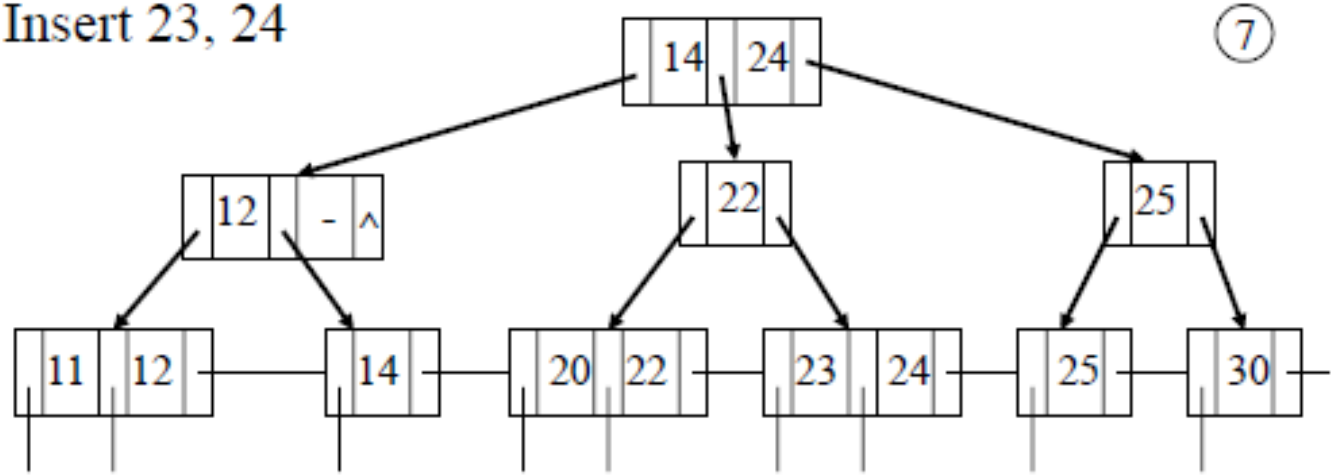
Internal node split at $j = \lceil \frac{m}{2} \rceil$

Split at 14 and 14 is *moved* up

Insert 22



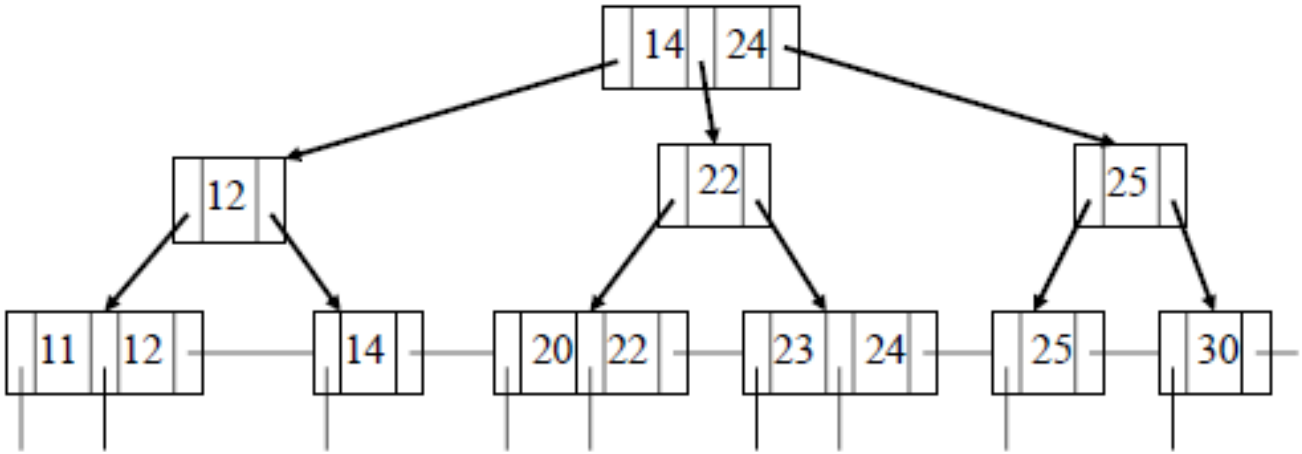
Insert 23, 24



Deletion in B^+- trees

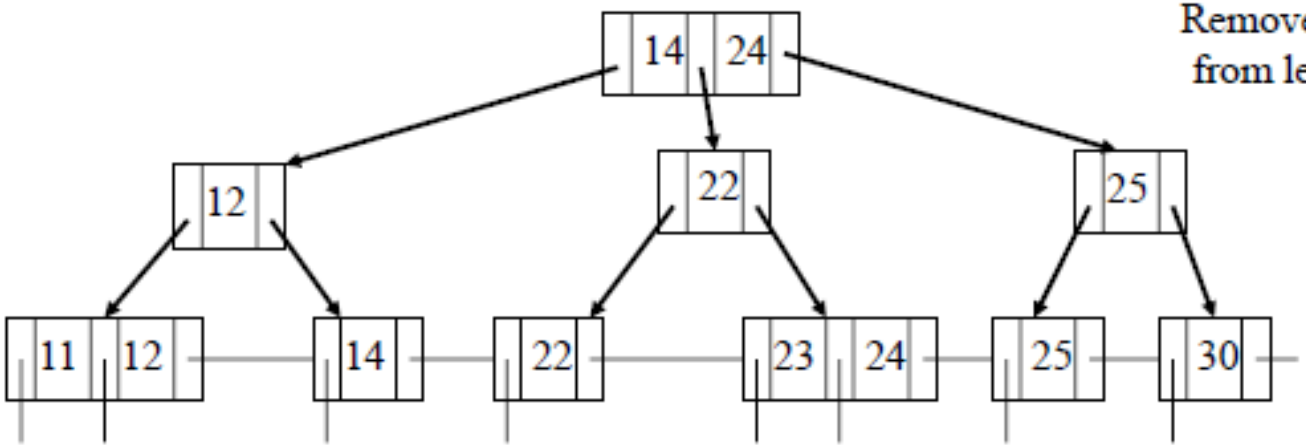
- Delete the entry from the leaf node
- Delete the entry if it is present in Internal node and replace with the entry to its right / right sibling.
- If underflow occurs after deletion
  - Distribute the entries from left sibling
    if not possible – Distribute the entries from right sibling
    if not possible – Merge the node with left and right sibling

Example

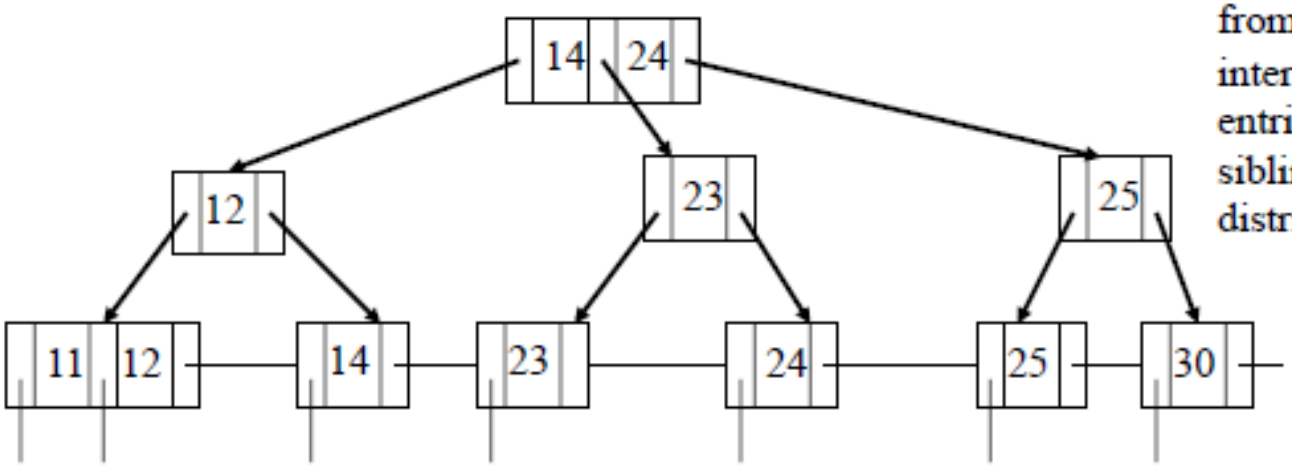


Delete 20

Removed entry from leaf here

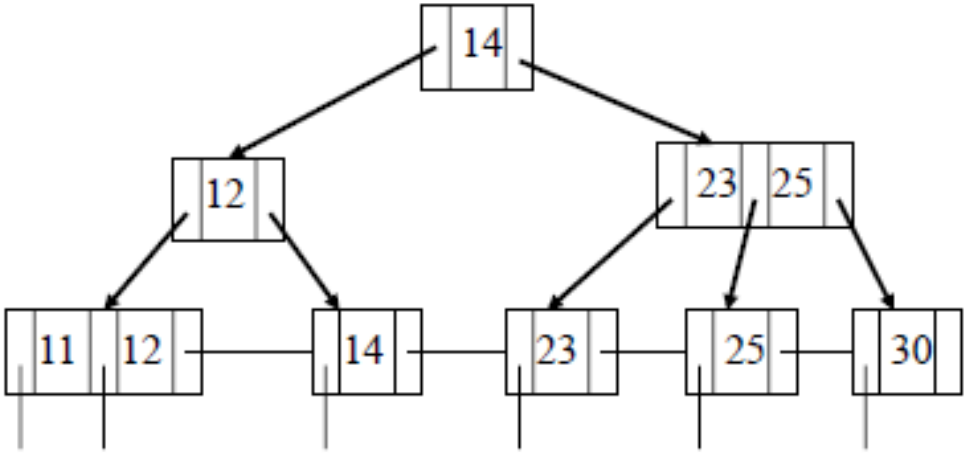


Delete 22

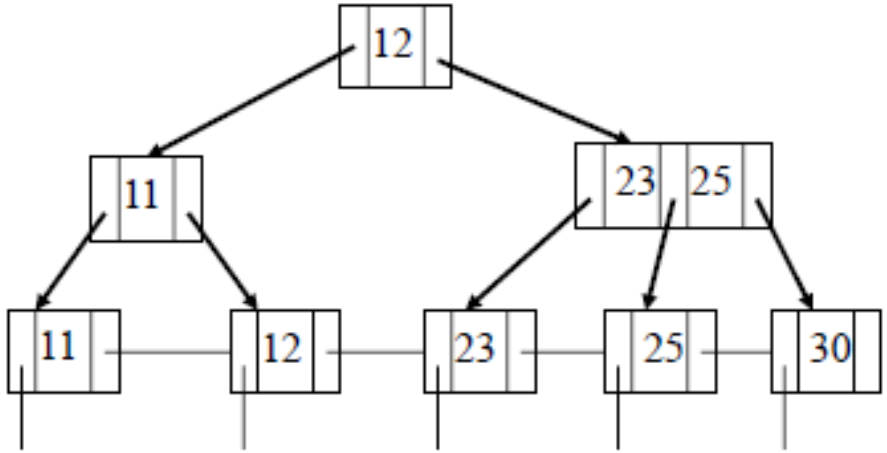


22 is removed from leaf and internal node entries from right sibling are distributed to left

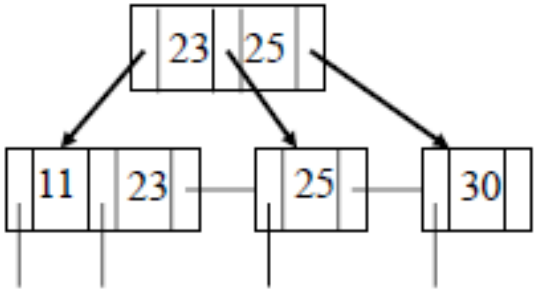
Delete 24



Delete 14



Delete 12



Level drop has occurred

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.

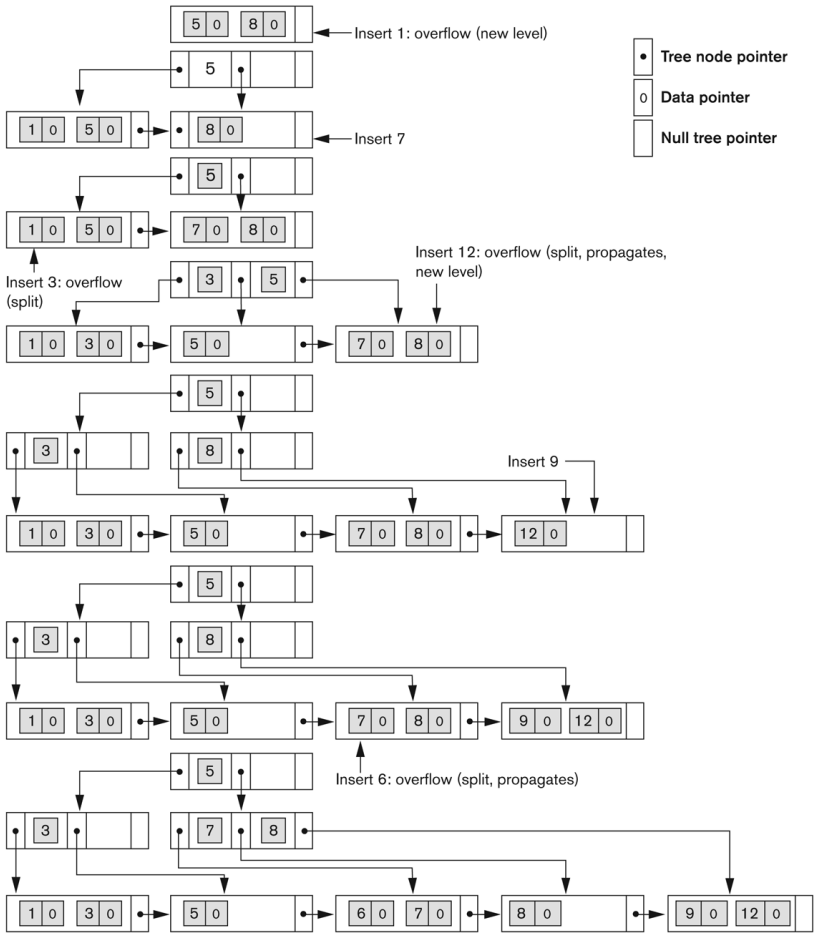
Disadvantages of B^+- trees:

Insert and delete operations are complicated

Root node becomes a *hotspot*

An Example of an Insertion in a B+-tree

Figure 14.12
An example of insertion in a B+-tree with $p = 3$ and $p_{leaf} = 2$.



An Example of a Deletion in a B+-tree

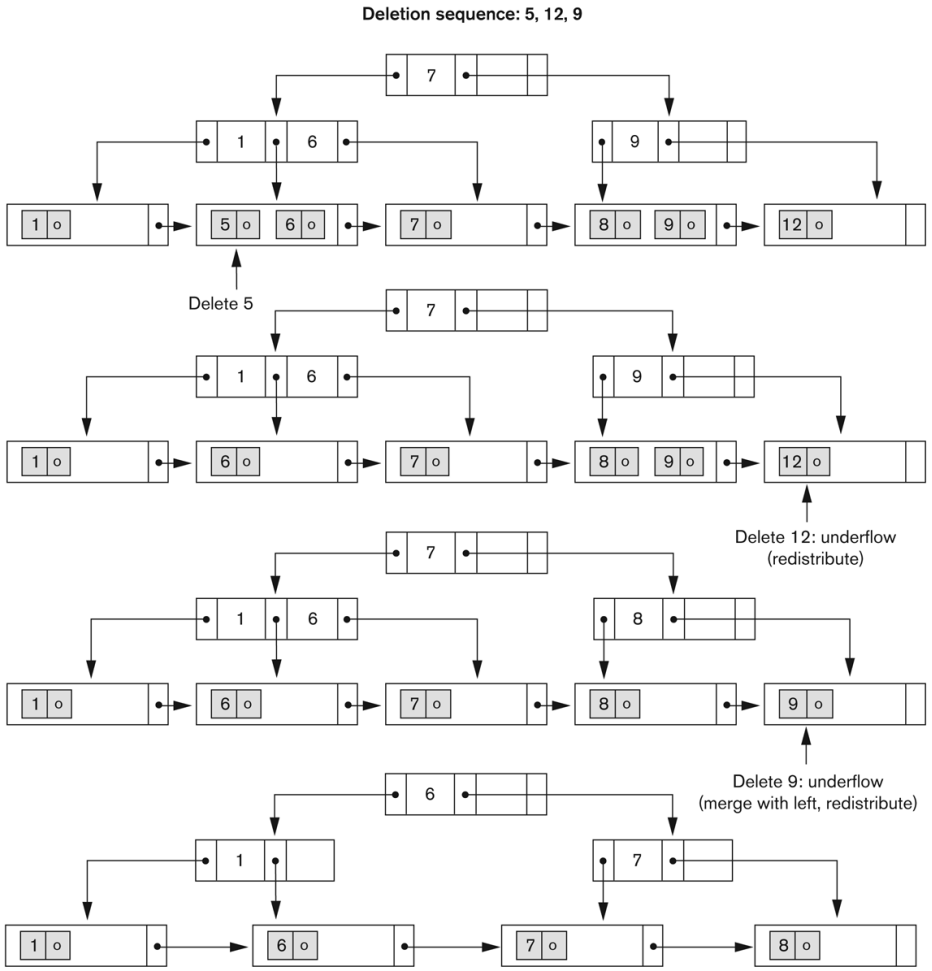


Figure 14.13
An example of deletion from a B+-tree.

Summary

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