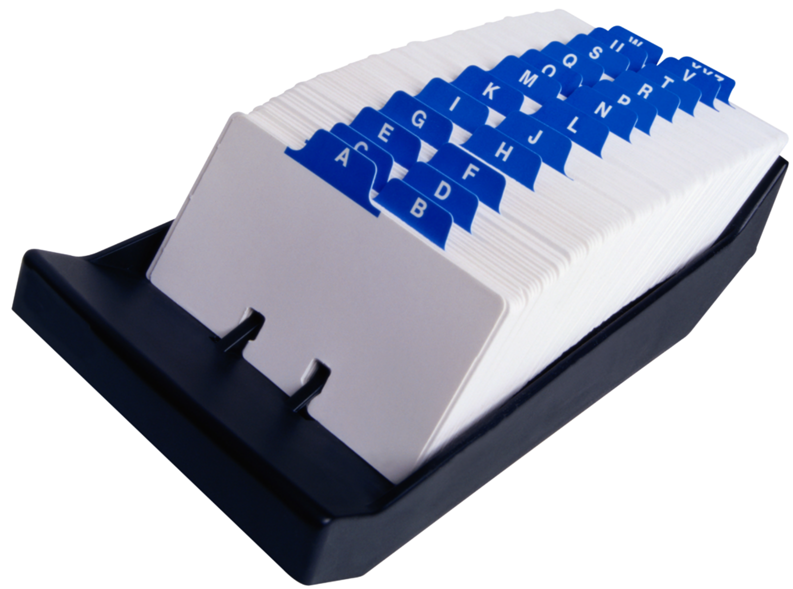
COSC344

Database Theory and Applications

Lecture 19
Indexing (2)



Overview

- Last Lecture
  - Database indexing
    - Single-Level Ordered Index
    - Multi-Level Index
  - Source: Chapter 17
- This Lecture
  - Dynamic Multi-Level Index
    - Trees
    - B -trees
    - B^+-trees
  - Source: Chapter 17
- Next Lecture
  - Database security and integrity
  - Source: Chapter 25

Multilevel Indexes

- First level considered an ordered file
- Second level is a primary index on the first level
  - one entry per block in first level
- Repeat the process for other levels
- bfr_i is the blocking factor for the index
- Used on
  - primary
  - clustering
  - secondary
- Problems with insertions/deletions

Multilevel Indexes (cont.)

The levels except the first level are **primary index**.

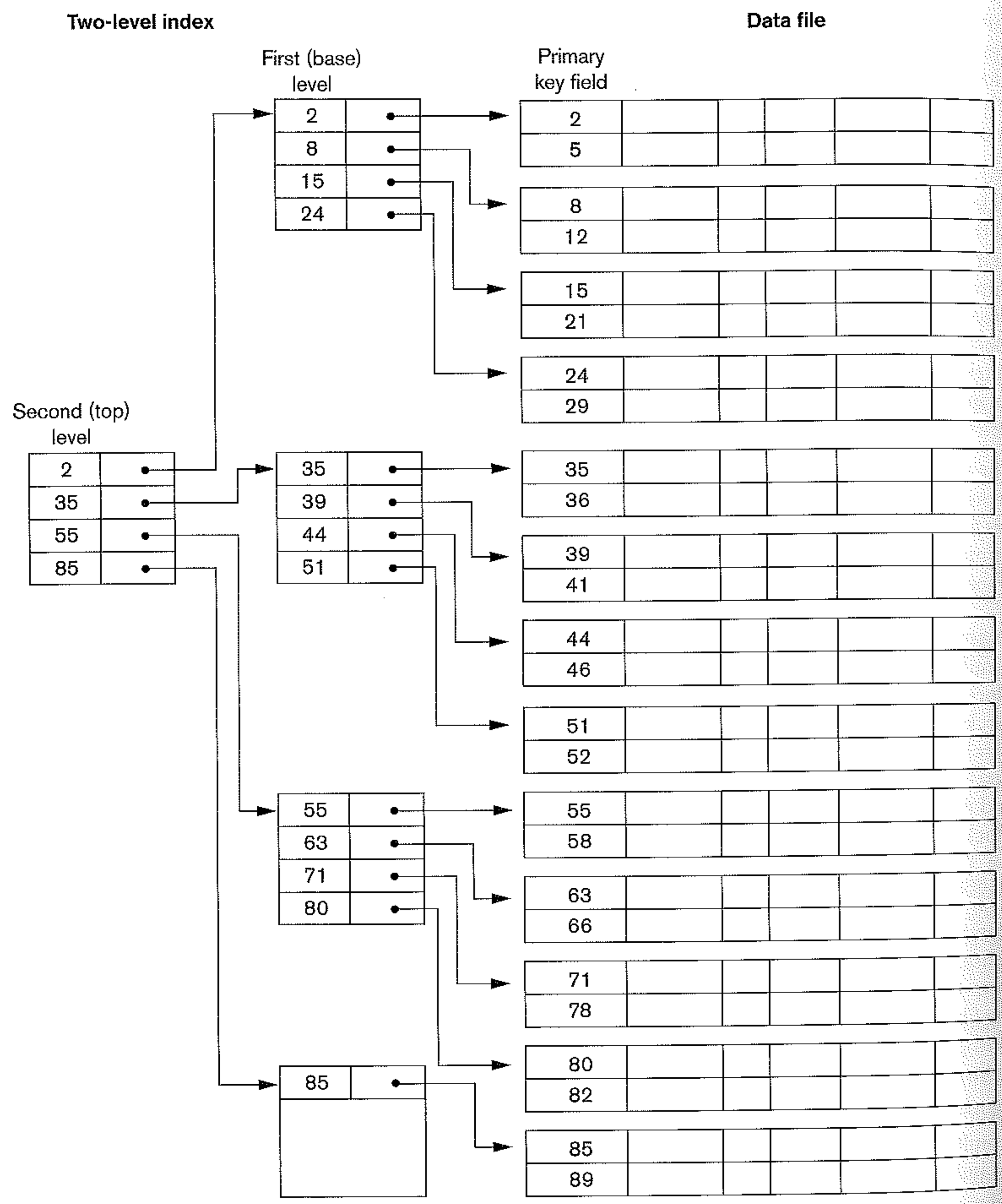


Figure 14.6
A two-level primary index resembling ISAM (Index Sequential Access Method) organization.

Brief Tree Review

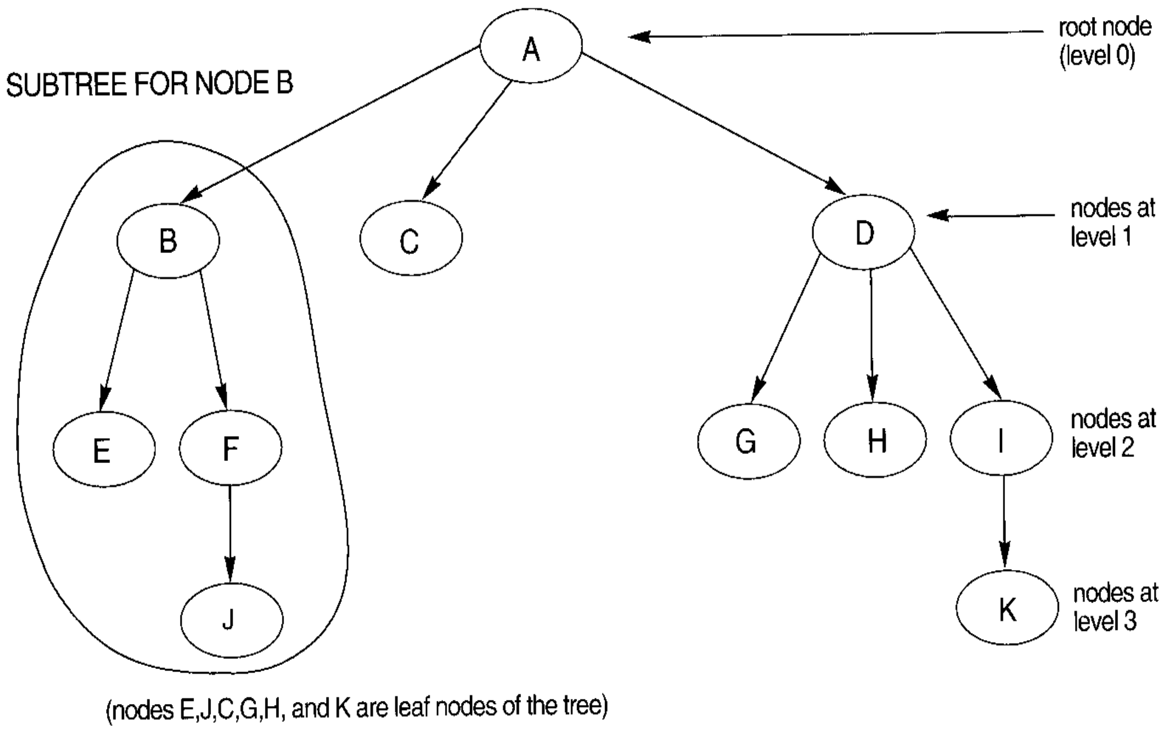
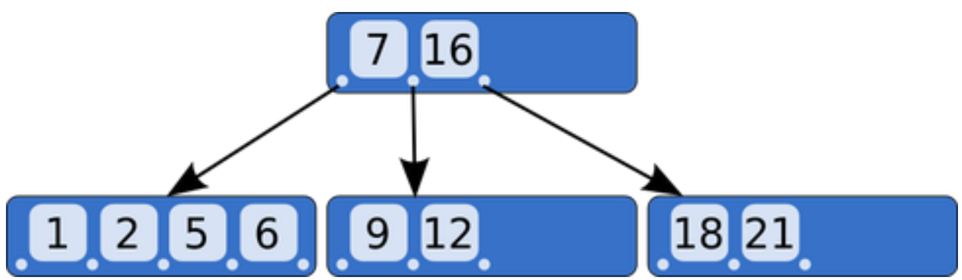


Figure 6.7 A tree data structure that shows an unbalanced tree.

- Root
- Parent and Child
- Leaf node
- Internal node
- Descendant nodes
- Subtree
- Level
- Balanced

Search Trees

- A search tree of order p is
  - Each node contains at most p-1 search values and p pointers in the order <P_1, K_1, P_2, K_2, . . ., P_{q-1}, K_{q-1}, P_q> where $q \leq p$.
  - Each P_i is a pointer to a child or a NULL pointer.
  - Each K_i is a search value
  - All search values are assumed be unique.
  - Two constraints
    - Within each node, $K_1 < K_2$ . . . $< K_{q-1}$
    - For all values X in the subtree pointed by P_i, we have $K_{i-1} < X < K_i$ for $1 < i < q$, $X < K_i$ for $i = 1$, and $K_{i-1} < X$ for $i = q$



Search Trees (cont.)

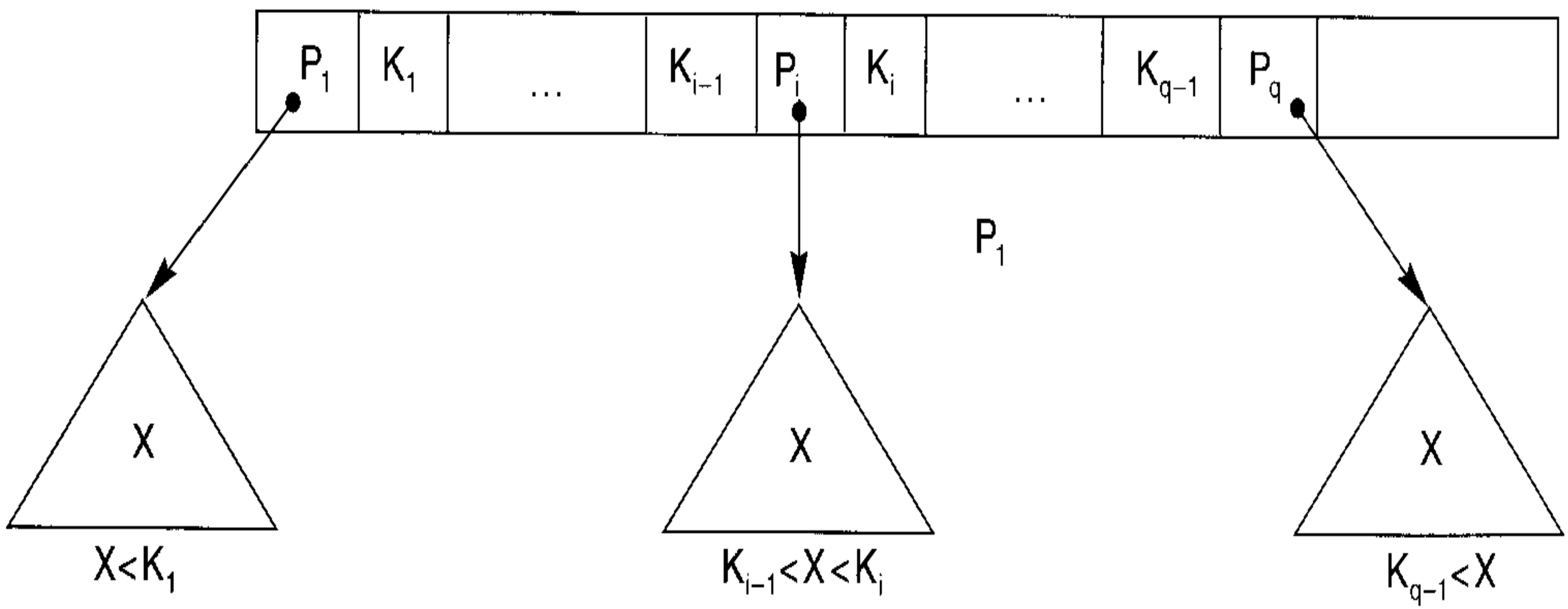


Figure 6.8 A node in a search tree with subtrees below it.

Search Trees (cont.)

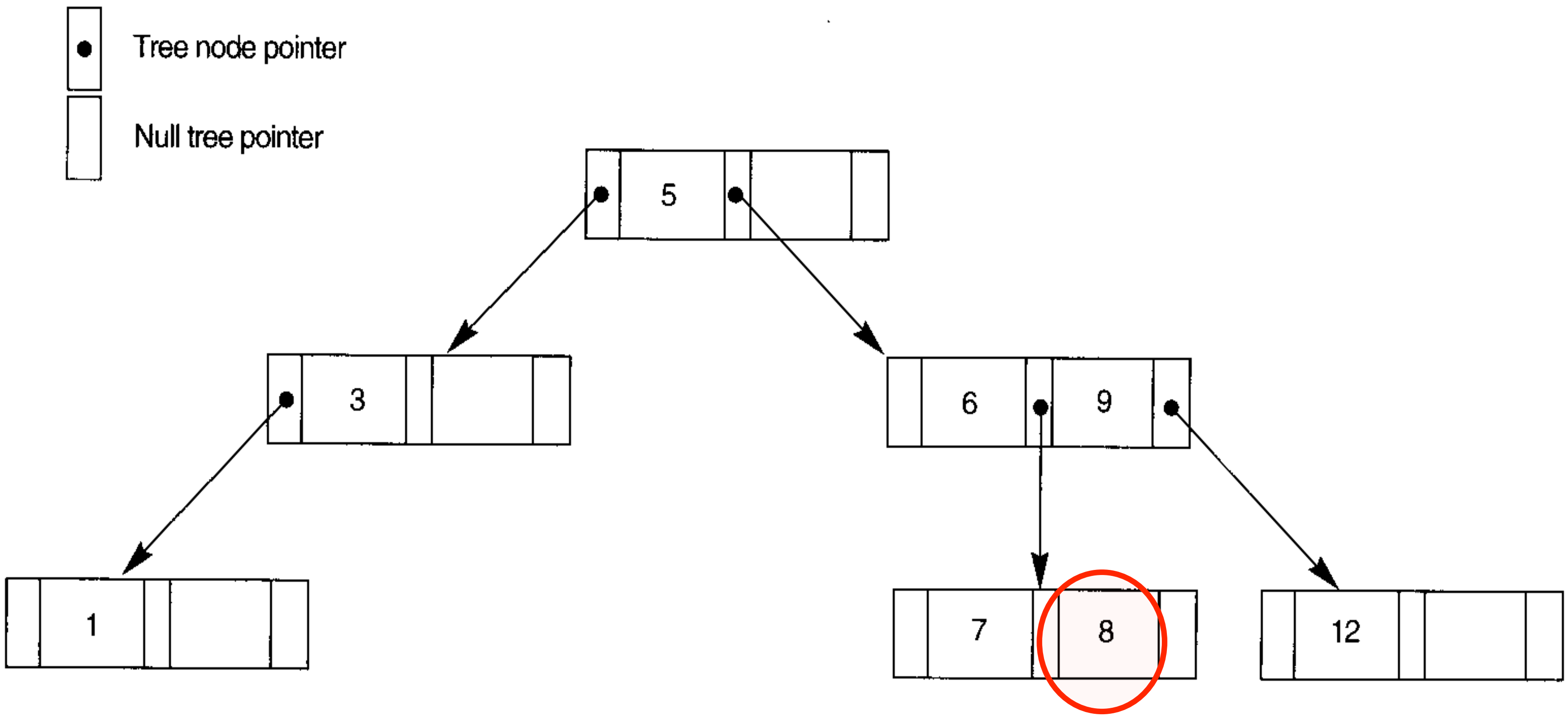
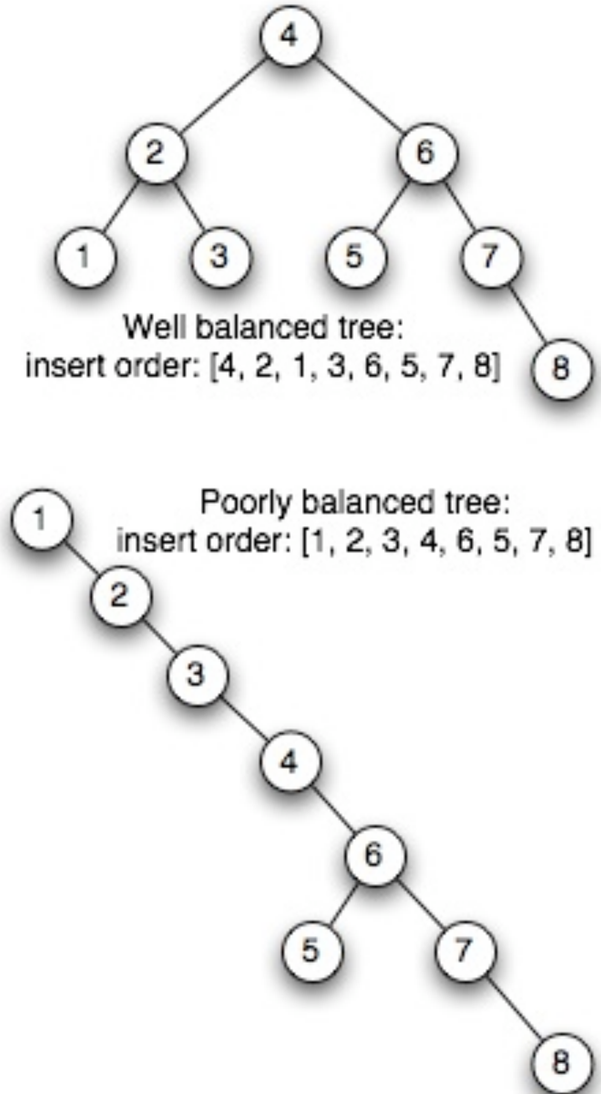


Figure 6.9 A search tree of order p = 3.

Search Trees(cont.)

- Not guaranteed to be balanced
- Tree balance is sensitive to insertion order
- Goals for balancing a search tree
  - Nodes are evenly distributed so that the depth of the tree is minimized
  - Make the search speed uniform
- Deletion may leave near empty nodes
- Another goal
  - The index tree does not need too much restructuring as records are inserted into or deleted



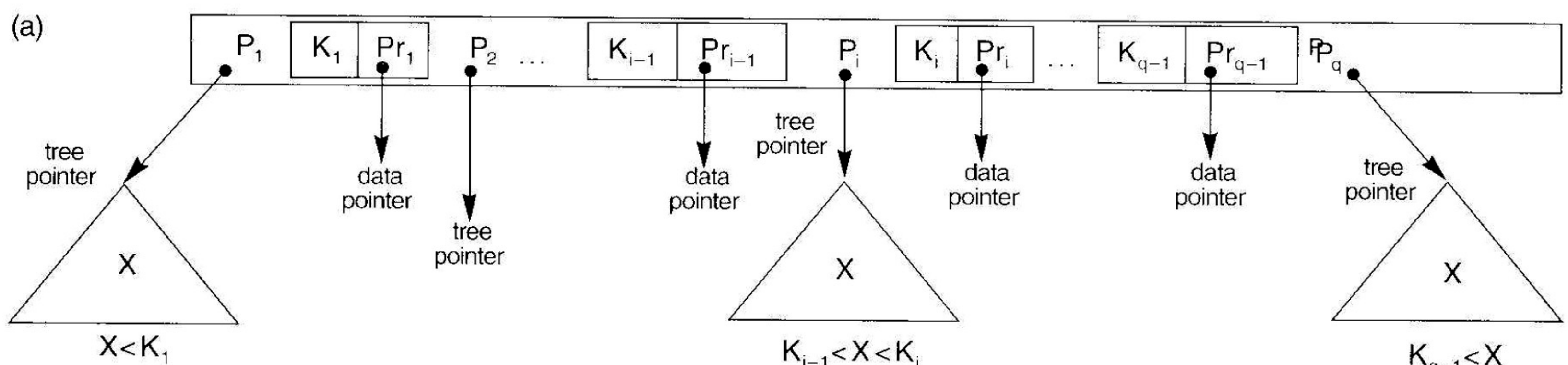
B-Trees

- A B-tree is a search tree with additional constraints:
  - The tree is always balanced.
  - The space wasted by deletion, if any, never becomes excessive.
- Each internal node is of the form

  $<P_1, <K_1,Pr_1>, P_2, <K_2,Pr_2>, \; . \; . \; ., P_{q\text{-}1}<K_{q\text{-}1},Pr_{q\text{-}1}>, P_q>$

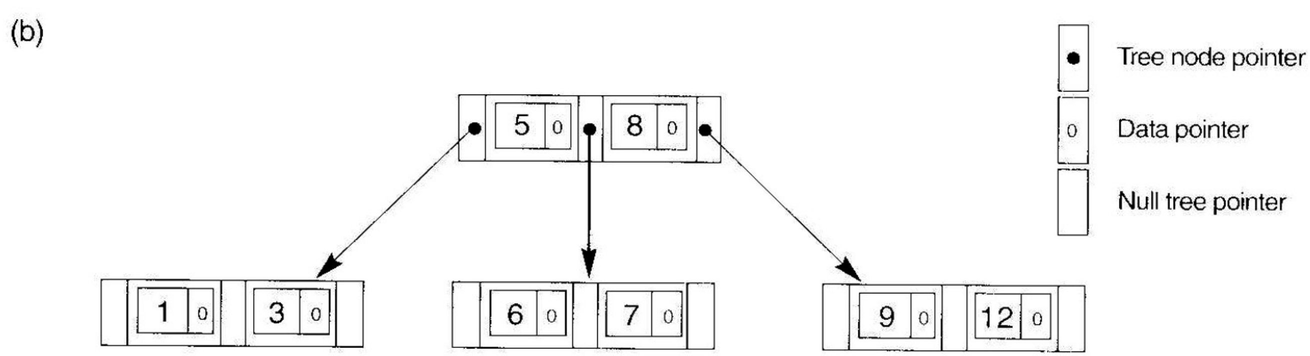
  P_i: a tree pointer to another node in the B-tree

  Pr_i: a data pointer to the record with search key value of K_i



B-Trees (cont.)

- At least **half full**
  - Each node has at most p tree pointers, and has at least $\lceil p/2 \rceil$ tree pointers except the root and leaf nodes.
- A node with q tree pointers has q-1 search key field values and hence q-1 data pointers
- Balanced
  - All leaf nodes are at the same level, and their tree pointers are NULL.



B-Trees (cont.)

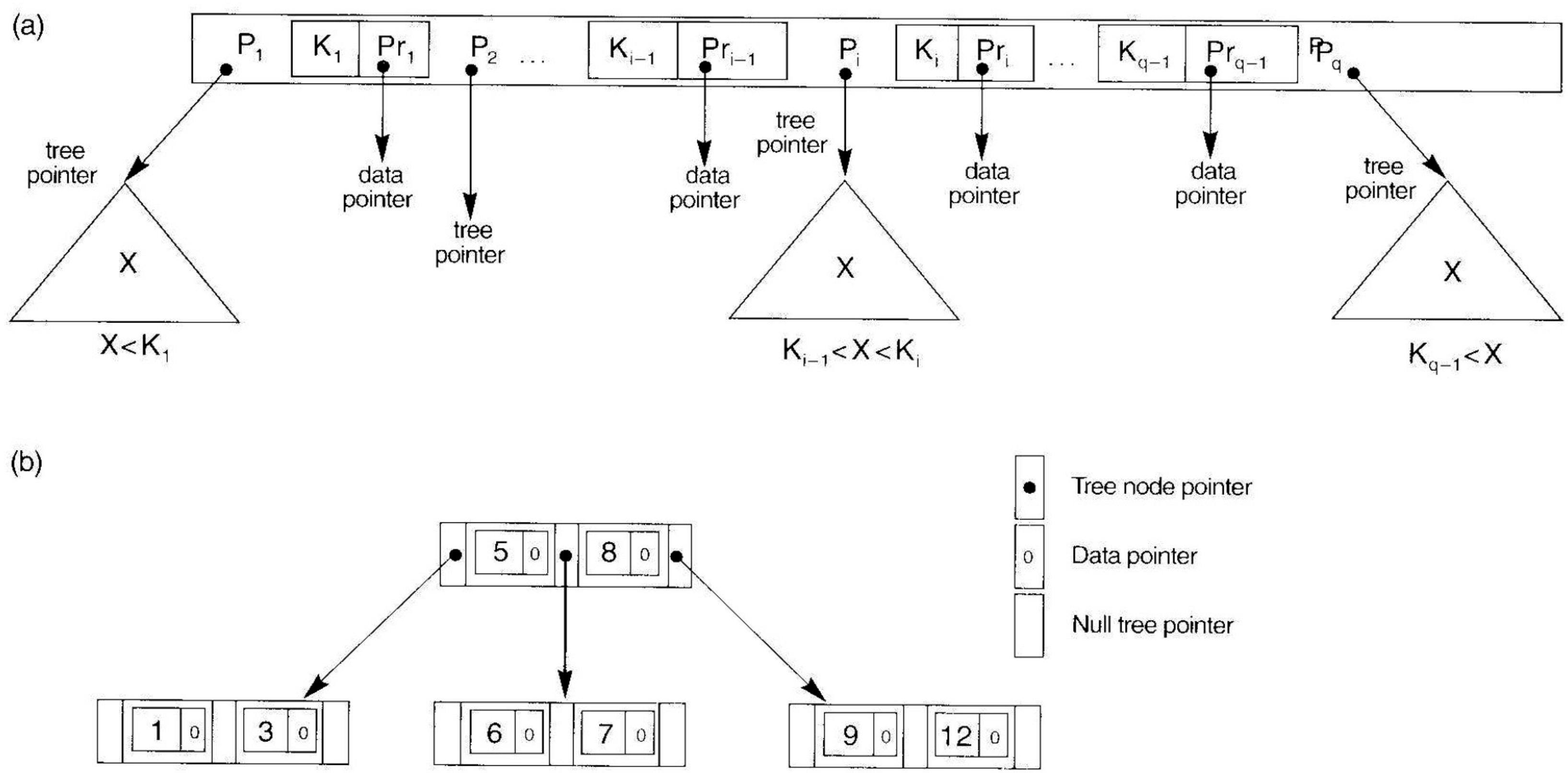


Figure 6.10 B-tree structures. (a) A node in a B-tree with q – 1 search values. (b) A B-tree of order p = 3. The values were inserted in the order 8, 5, 1, 7, 3, 12, 9, 6.

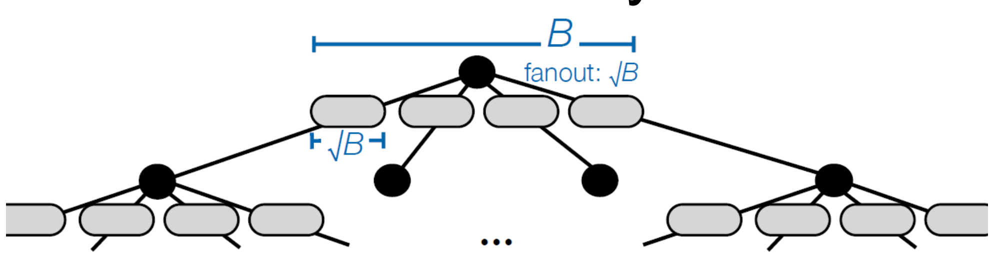
Insertion and deletion become more complex in order to maintain these constraints

B-Trees (cont.)

Example: For a B-tree on a nonordering key field with p=23, assume that each node of the B-tree is 69% full.

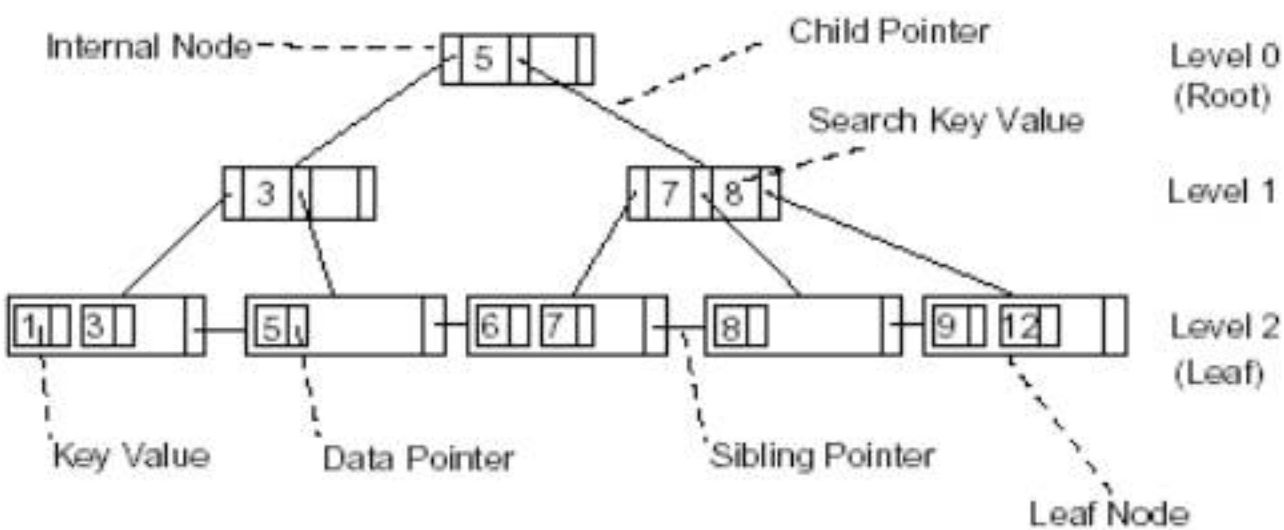
1. Each node, on the average, will have p*0.69=23*0.69≈16 pointers and 15 key field values
2. The average fan-out *fo*=16.
3. The number of pointers and values at each level

Root:	1 node	15 key entries	16 pointers
Level 1:	16 nodes	240 key entries	256 pointers
Level 2:	256 nodes	3840 key entries	4096 pointers
Level 3:	4096 nodes	61440 key entries	



B^+-Trees

- Most implementations of a dynamic multilevel index use a variation of B-tree called B^+-tree
  - Data pointers are stored ***only at the leaf nodes*** of the tree.
  - The structure of leaf node differs from the structure of the internal nodes.
  - Leaf nodes are usually linked together for ordered access on the search field



B^+-Trees (continued)

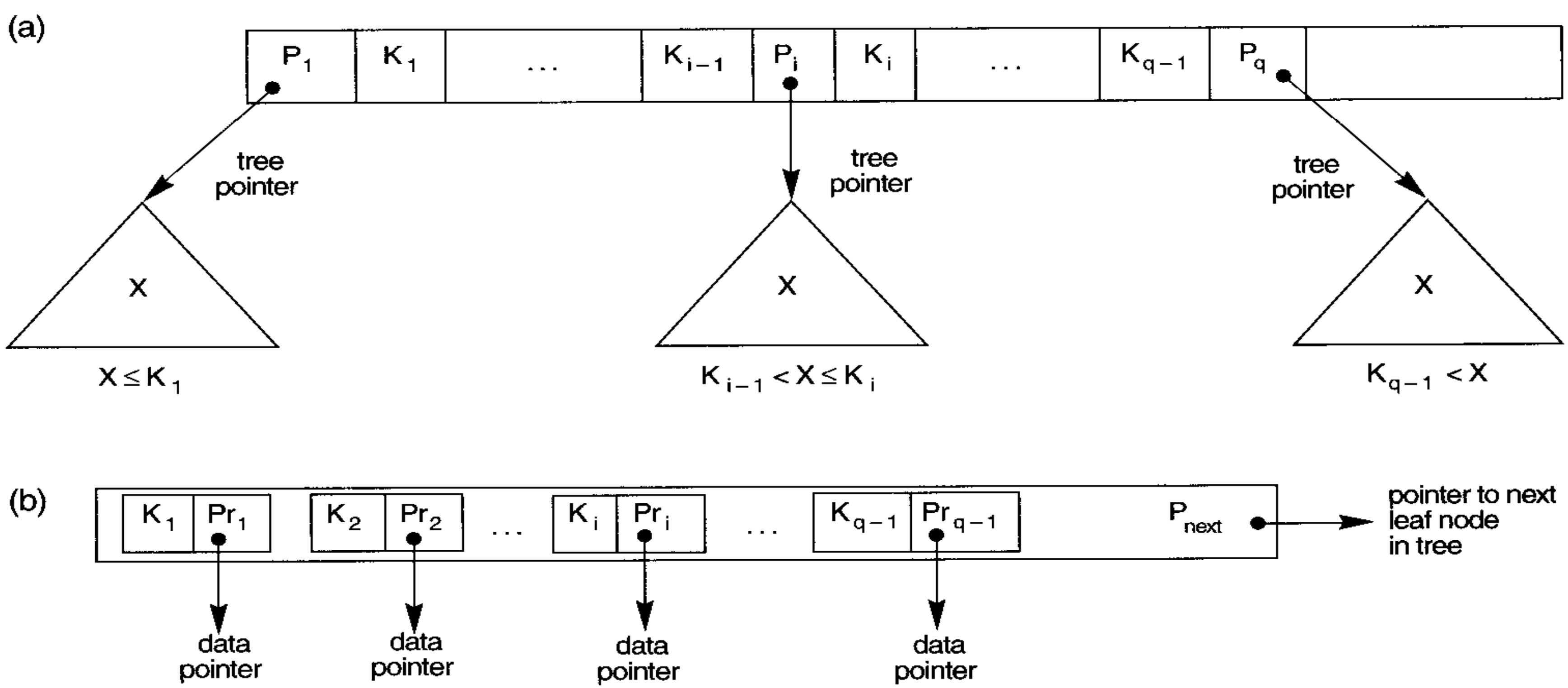
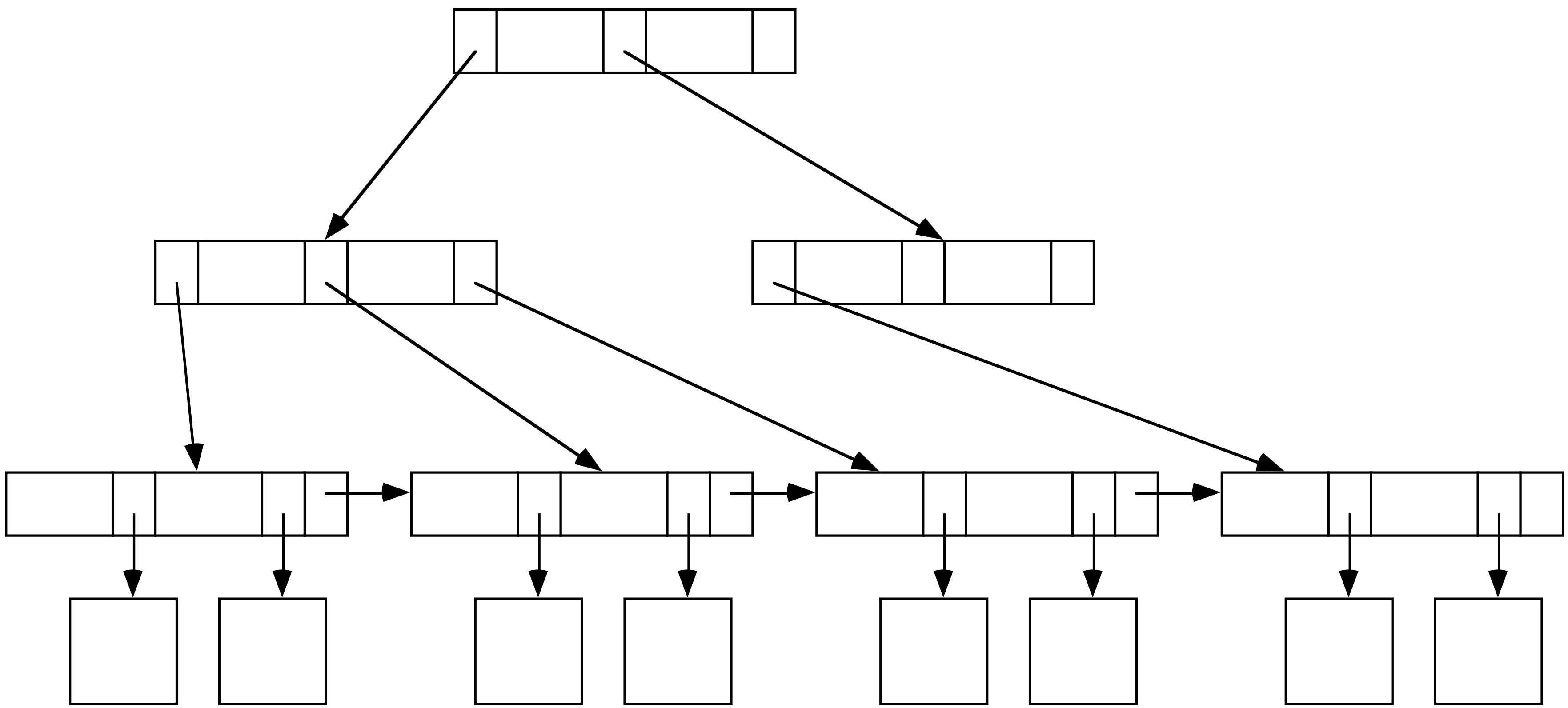


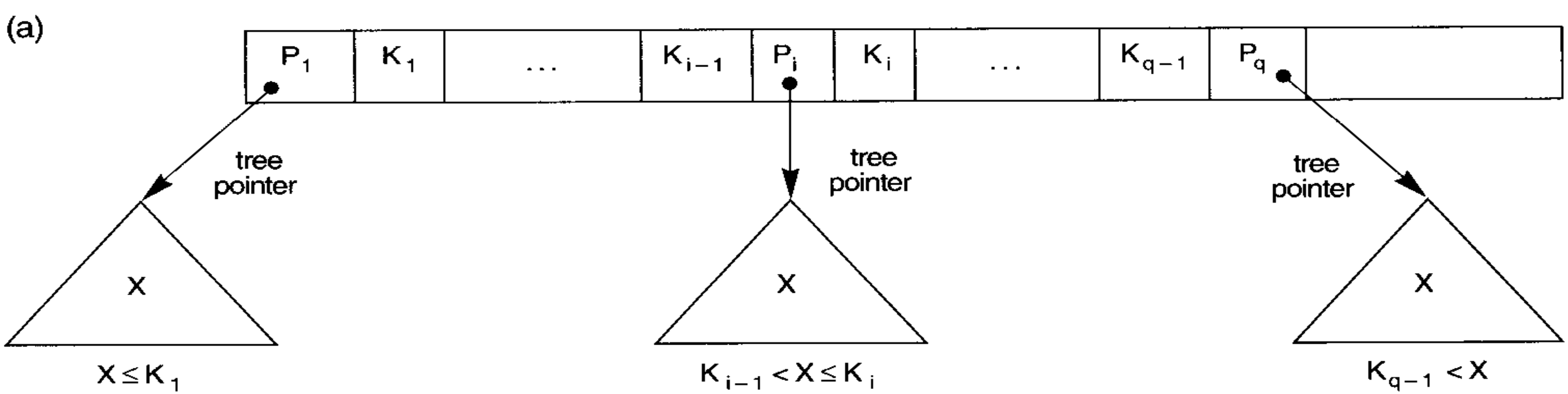
Figure 6.11 The nodes of a B^+-tree. (a) Internal node of a B^+-tree with $q-1$ search values. (b) Leaf node of a B^+-tree with $q-1$ search values and $q-1$ data pointers.

B$^+$-Trees (continued)



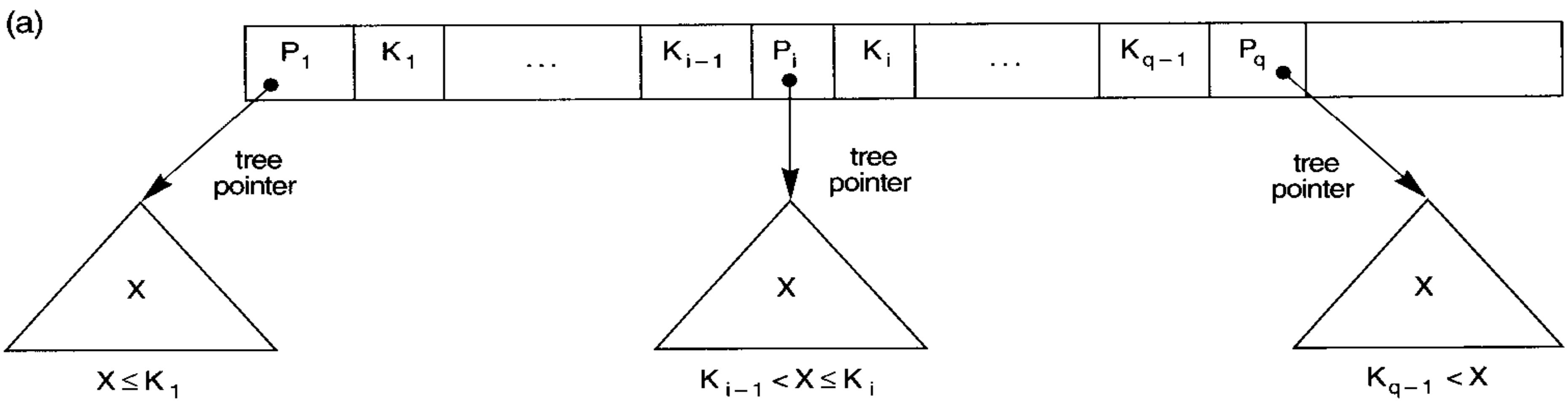
Internal Node of a B+-Tree

- Assume a B+-tree of order p
  - Each internal node is of the form <P_1, K_1, P_2, K_2, . . ., P_{q-1}, K_{q-1}, P_q> where $q \le p$.
  - Each P_i is a tree pointer.
  - Within each node, $K_1 < K_2$ . . . $< K_{q-1}$
  - For all values X in the subtree pointed at by P_i
    - $X \le K_i$ for $i = 1$
    - $K_{i-1} < X \le K_i$ for $1 < i < q$
    - $K_{i-1} < X$ for $i = q$



Internal Node of a B⁺-Tree

- Assume a B⁺-tree of order p

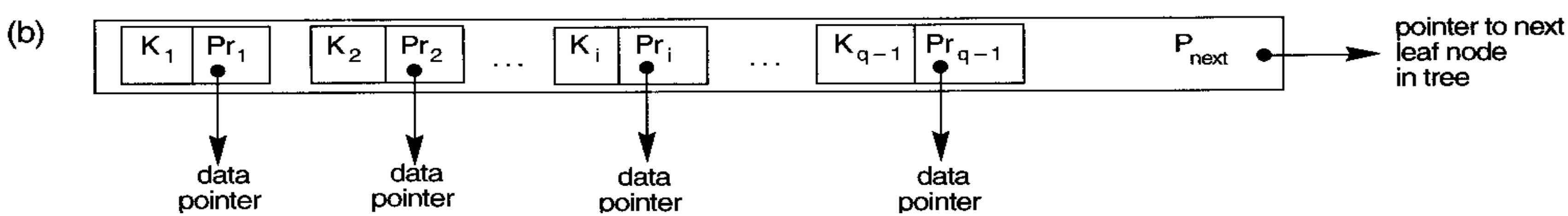


  - Each internal node has at most p tree pointers.
  - Each internal node, except the root and leaf nodes, has at least ⌈p/2⌉ tree pointers. The root node has at least two tree pointers if it is an internal node.
  - An internal node with q tree pointers, $q \le p$, has q-1 search key field values.

Leaf Node of a B^+-Tree

- Each leaf node is of the form
  - $< <K_1, Pr_1>, <K_2, Pr_2>, \ldots, <K_{q-1}, Pr_{q-1}>, P_{next}>$

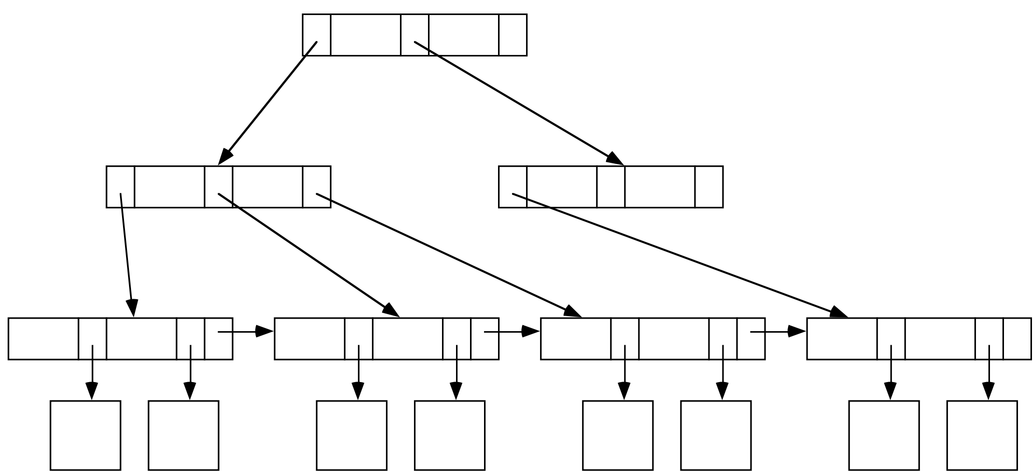
  where $q \leq p$
- Each Pr_i is a data pointer.
- P_{next} points to the next leaf node of the B^+-tree.
- Within each node, $K_1 < K_2 \quad . \; . \; . \quad < K_{q-1}$
- Each Pr_i is a data pointer that points to the record whose search field value is K_i or to a file block containing the record (or to a block of record pointers that point to records whose search field value is K_i if the search field is not a key).
- Each leaf node has at least $\lceil p/2 \rceil$ values.
- All leaf nodes are at the same level.



Characteristics of B^+-Trees

- By starting at the leftmost node, it is possible to traverse leaf nodes as a linked list.

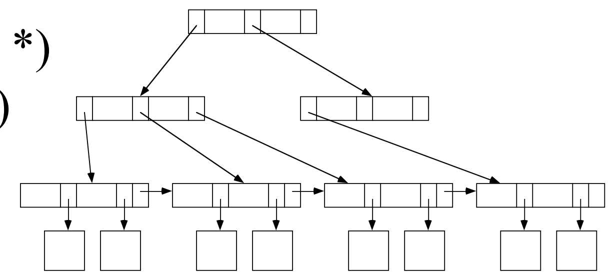
- More entries can be packed into an internal node of a B^+-tree than for a similar B-tree
  – No data pointers in the internal nodes in a B^+-tree
- B^*-Tree - each node must be 2/3 full



Search with B^+ Trees

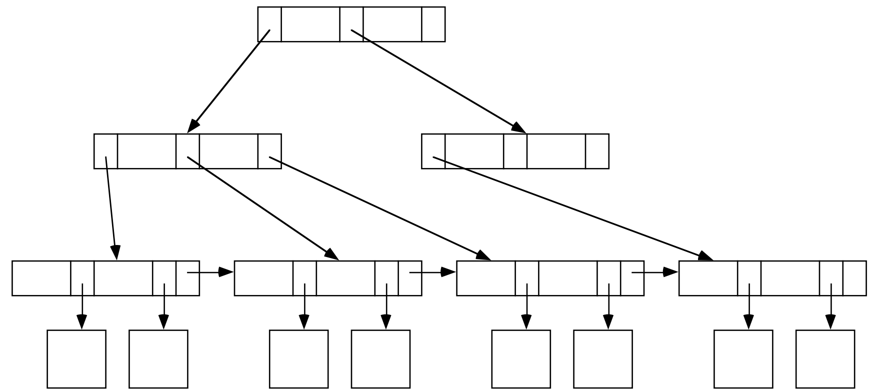
Algorithm: Search for a record with Search Key Field Value K.

```
n <- block containing root node of B+-tree;
read block n;                                   <- block access
while (n is not a leaf node of the B+-tree) do
 begin
   q<- number of tree pointers in node n;
   if  K≤n.K1 (*n.ki refers to the ith search field value in node n *)
   then n<- n.P1 (* n.Pi refers to the ith tree pointer in node n *)
   else if K> n. Kq-1
        then n <- n.Pq
        else  begin
                search node n for an entry i such  that n.Ki-1<K<n.Ki;
                n<- n.Pi;
              end;
   read block n;                                <- block access
 end;
Search block n for entry (Ki, Pri) with K=Ki; (*search leaf node*)
if  found
  then read data file block with address Pri and retrieve record   <- block access
  else the record with search field value K is not in the data file;
```



Search Example

Block size: B=512 bytes
Block pointer: BP=6 bytes
Record pointer: P_R= 7 bytes
Search Key: S = 9 bytes
Num_of_records: R = 30,000



(1) The orders p and p_{leaf} of the B^+ Tree

$p*BP+(p-1)*S \leq B \qquad p \leq 34$

$p_{leaf}*(P_R+S)+BP \leq B \qquad p_{leaf} \leq 31$

(2) The number of leaf-level blocks needed if nodes are 69% full.

ceiling(R/(ceiling(p_{leaf}*0.69))) =1364

Search Example (cont.)

(2) The number of block access needed to search for and retrieve a record from the file based on the primary key using the B^+ tree

fo = ceiling(0.69*p) = ceiling(0.69*34) = ceiling(23.46) = 24

Level 0: ceiling(3/fo) = 1
Level 1: ceiling(57/fo) = 3
Level 2: ceiling(1364/fo) = 57
Level 3: 1364

Block access is 4+1=5

Disadvantages of Indexes

- Indexes add overhead when inserting, deleting, and possibly updating.
- In a mass load, add the index *after* the data is loaded.

Indexes in SQL

```
CREATE INDEX <name>
  ON <table> (column [ASC | DESC]
     [, column [ASC | DESC]] ...);
```

```
CREATE INDEX salary_index
  ON employee (salary);
```

```
DROP INDEX <name>;
```

NOTE - Oracle automatically creates an index on the primary key.

Questions to Ponder

What are some issues in database security?

Consider the company database - How might we restrict operations among user groups?