Database Management Systems (DBMS)

Lec 21: Query processing and optimization (Contd.)

Ramesh K. Jallu
IIIT Raichur

Date: 16/04/21

Recap

- An example for outer join and the OUTER UNION operation
- Introduction to indexing
  - Primary Index, Clustered Index, and Secondary Index
- A few algorithms for SELECTION operation

Overview

- Multilevel indexing
- Search trees
- B-tree

Multilevel indexing

- The indexing schemes we have described thus far involve an ordered index file
- A binary search requires approximately $log_2 b_i$ block accesses for an index with b_i blocks
- The ***blocking factor*** (denoted by ***f***) is the number of records per block in indexing
- The idea behind a ***multilevel index*** is to reduce the part of the index that we continue to search by bfr_i, for $f > 2$
- We divide it n-ways (where $n = f$) at each search step using the multilevel index, thus a search requires $log_n b_i$ block accesses

Multilevel indexing (Contd.)

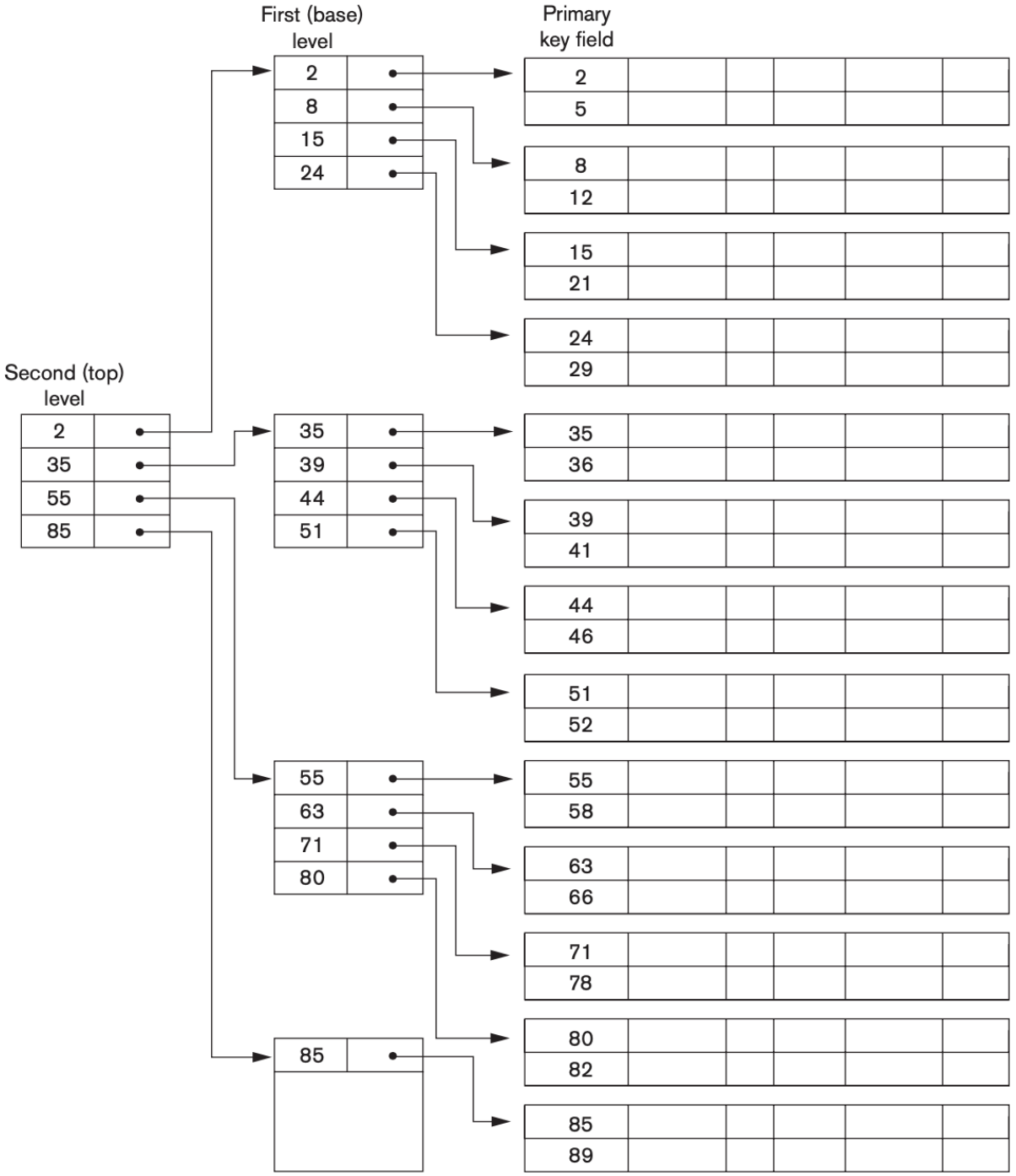
- A multilevel index considers the index file, which we refer to as the ***first level***
- Since the first-level index file as a sorted data file, we can create a primary index for the first level
  - This index to the first level is called the ***second level*** of the multilevel index
  - We can use block anchors so that the second level has one entry for *each block* of the first level
  - Observe that there is no change in ***f*** in this level and subsequent levels

Multilevel indexing (Contd.)

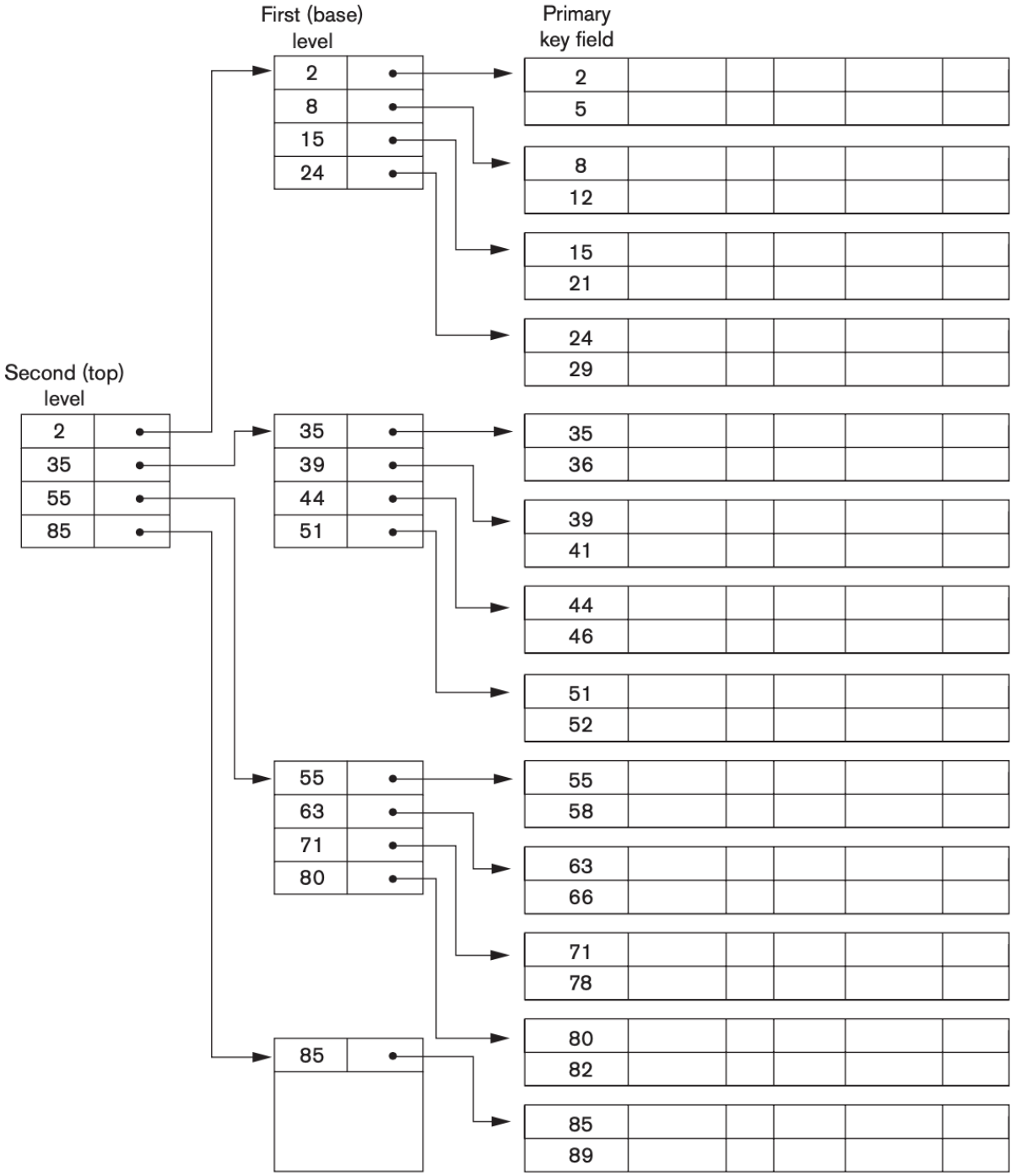
- The ***third level***, which is a primary index for the second level, has an entry for each second-level block
- We can repeat the preceding process until all the entries of some index level ***t*** fit in a single block
  - This block at the t^{th} level is called the ***top*** index level
- Each level reduces the number of entries at the previous level by a factor of ***f***
- If the first level contains r_1 records, then this level needs r_1/f blocks, which is the number of entries r_2 needed at the second level of the index

Multilevel indexing (Contd.)

- The number of blocks at the top index level $r_1/(f)^t = 1$; Implies, $t = log_f r_1$
- When searching the index, a single disk block is retrieved at each level
  - Hence, *t* disk blocks are accessed for an index search
- The multilevel scheme described here can be used on any type of index as long as the first-level index has ***distinct values for K(i)*** *and* ***fixed-length entries***
- Multilevel indexing can be seen as a tree, where each node can have as many as ***f*** pointers and ***f*** key values
- The index field values in each node guide us to the next node, until we reach the data file block that contains the required records

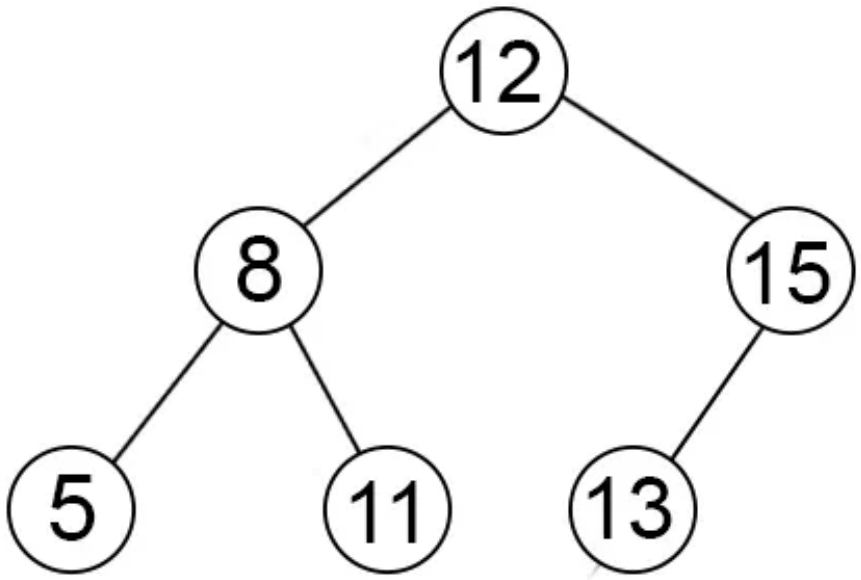


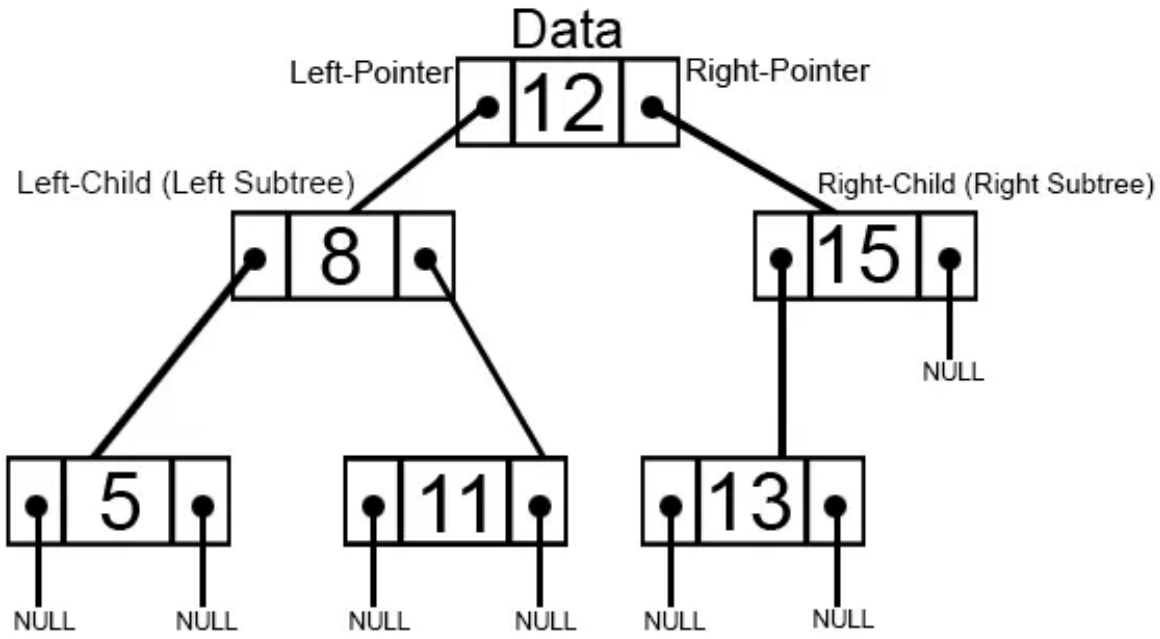
- Suppose that we have an ordered file with r = 300,000 records stored on a disk
- Block size B = 4,096 bytes and each record length R = 100 bytes
- Now, suppose that the ordering key field of the file is V = 9 bytes long, a block pointer is P = 6 bytes long
- The size of each index entry is 15 bytes, so 273 (= f) entries per block as each block is 4MB



- The number of blocks needed for the first-level index $\lceil (300{,}000/273) \rceil$ = 1,099 blocks
- The number of blocks for the second level is $\lceil (1{,}099/273) \rceil = 5$
- The number of blocks for the third level $\lceil (5/273) \rceil = 1$
- To access a record by searching the multilevel index, we must access one block at each level plus one block from the data file, so we need $t + 1 = 3 + 1 = 4$ block accesses

Binary search tree

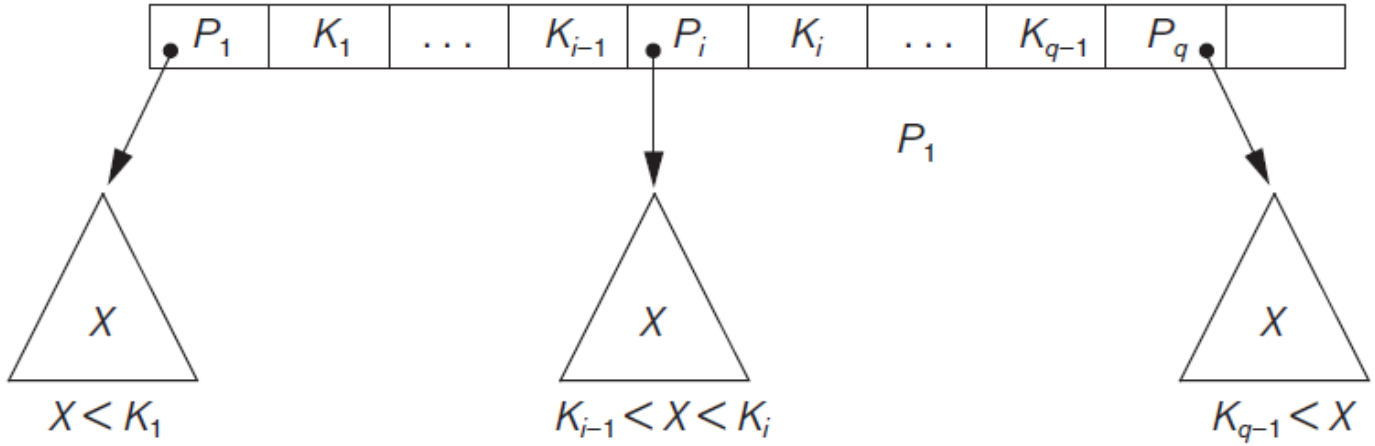




Search tree

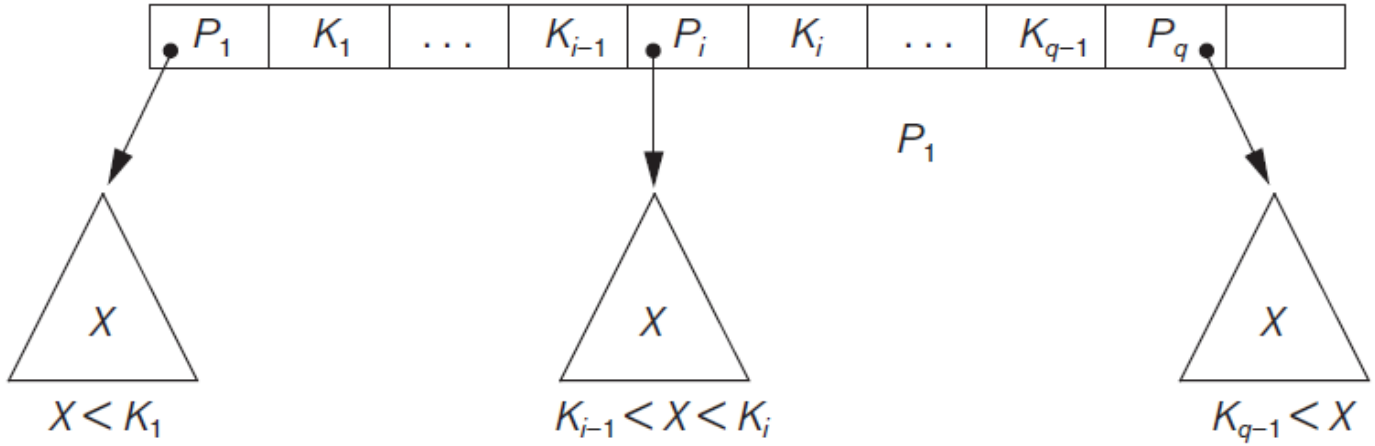
- A ***search tree*** is a special type of tree that is used to guide the search for a record

- A search tree of order p is a tree such that each node contains at most p-1 search values and p pointers in the order $<P_1, K_1, P_2, K_2, \ldots, P_{q-1}, K_{q-1}, P_q>$, where $q \leq p$



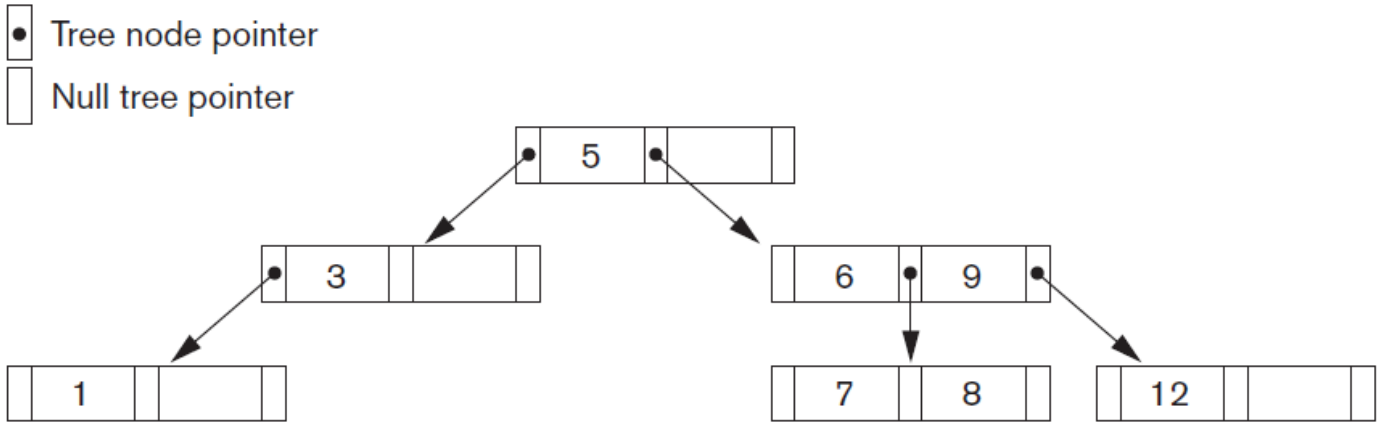
Search tree (Contd.)

- Each P_i is a pointer to a child node (or a **NULL** pointer), and each K_i is a search value from some ordered set of values
- Two constraints must hold at all times on the search tree
  1. Within each node, $K_1 < K_2 < \ldots < K_{q-1}$
  2. For all values X in the subtree pointed at 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 tree (Contd.)

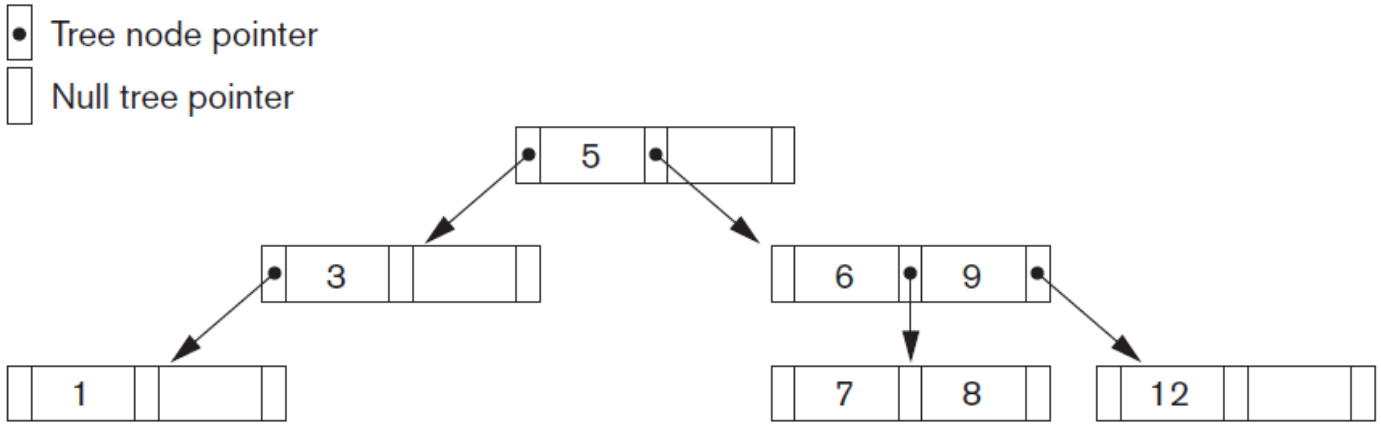
- We can use a search tree as a mechanism to search for records stored in a disk file
- The values in the tree can be the values of one of the fields of the file, called the search field
- Each key value in the tree is associated with a pointer to the record/disk block in the data file having that value



Search tree (Contd.)

- The search tree itself can be stored on disk by assigning each tree node to a disk block

- When a new record is inserted in the file, we must update the search tree by inserting an entry in the tree containing the search field value of the new record and a pointer to the new record



Disadvantages of search trees

- The algorithms to insert/delete search values in search tree do not guarantee that a search tree is balanced
- It demands restructuring as records are inserted into and deleted from the main file
- We cannot guarantee that the nodes are full
  - Record deletion may leave some nodes in the tree nearly empty, thus wasting storage space and increasing the number of levels

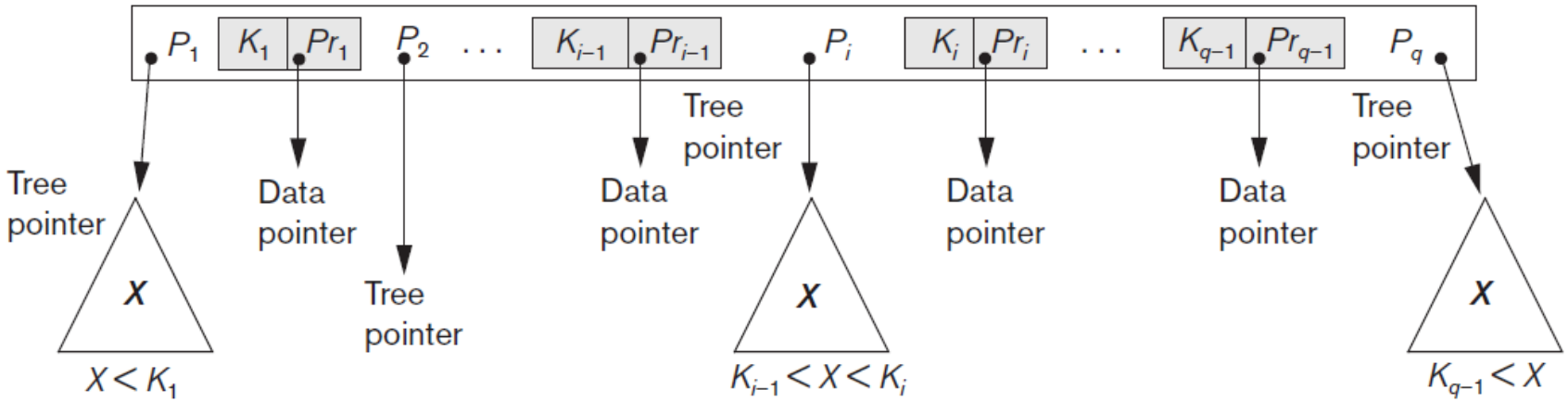
B-trees

- The ***B-tree*** addresses the problems with search tree by specifying additional constraints on the search tree
- B-trees were invented by ***Rudolf Bayer*** and ***Edward McCreight*** while working at Boeing Research Labs
- The B-tree has additional constraints that ensure that the tree is always balanced
  - The space wasted by deletion, if any, never becomes excessive

B-trees (Contd.)

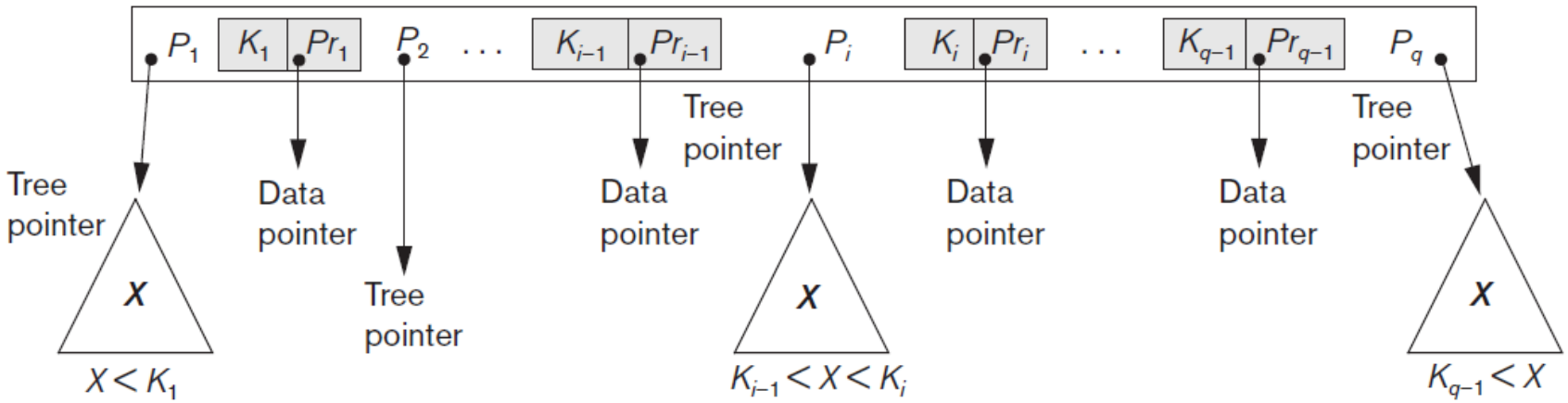
- A B-tree of order p, can be defined as follows:
  - Each internal node in the B-tree is of the form
$<P_1, <K_1, Pr_1>, P_2, <K_2, Pr_2>, \ldots, <K_{q-1}, Pr_{q-1}>, P_q>$, where $q \leq p$ and each P_i is a tree pointerand each Pr_i is a ***data pointer***

- Within each node, $K_1 < K_2 < \ldots < K_{q-1}$



B-trees (Contd.)

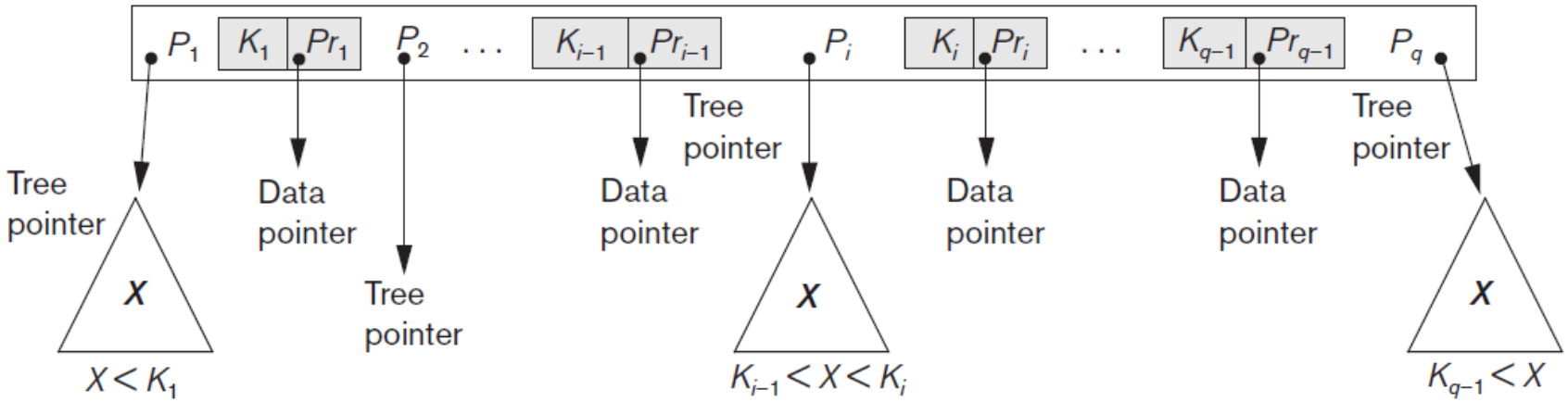
- For all search key field values X in the subtree pointed at by P_i, we have $K_{i-1} < X < K_i$ for $1 < i < q$; $X < K_i$ for $i = 1$; and $K_{i-1} < X$ for $i = q$
- Each node has at most p tree pointers



B-trees (Contd.)

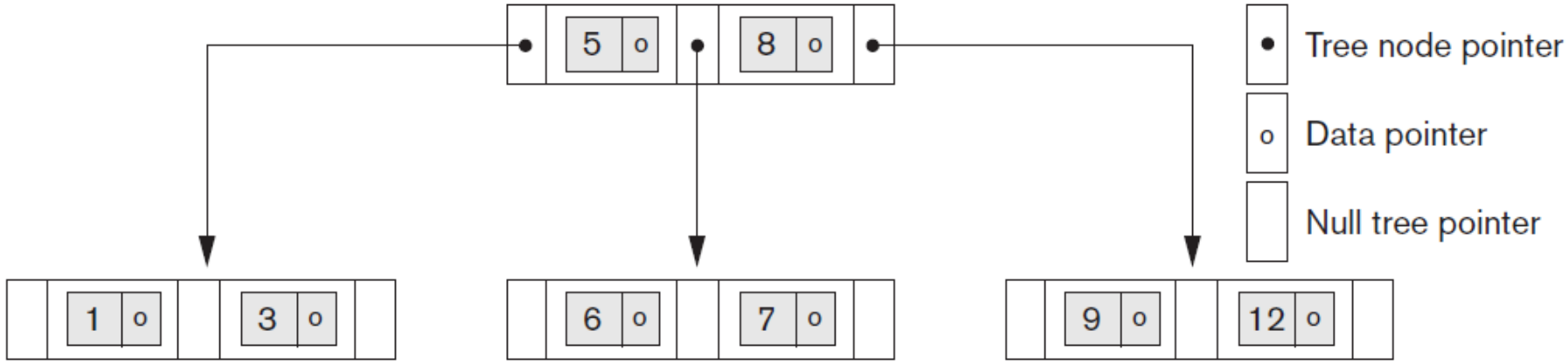
- Each node, except the root and leaf nodes, has at least $\lceil p/2 \rceil$ tree pointers
  - The root node has at least two tree pointers unless it is the only node in the tree

- A node with q tree pointers, $q \leq p$, has $q - 1$ search key field values (and hence has $q - 1$ data pointers)



B-trees (Contd.)

- All leaf nodes are at the same level
- Leaf nodes have the same structure as internal nodes except that all of their tree pointers P_i are **NULL**

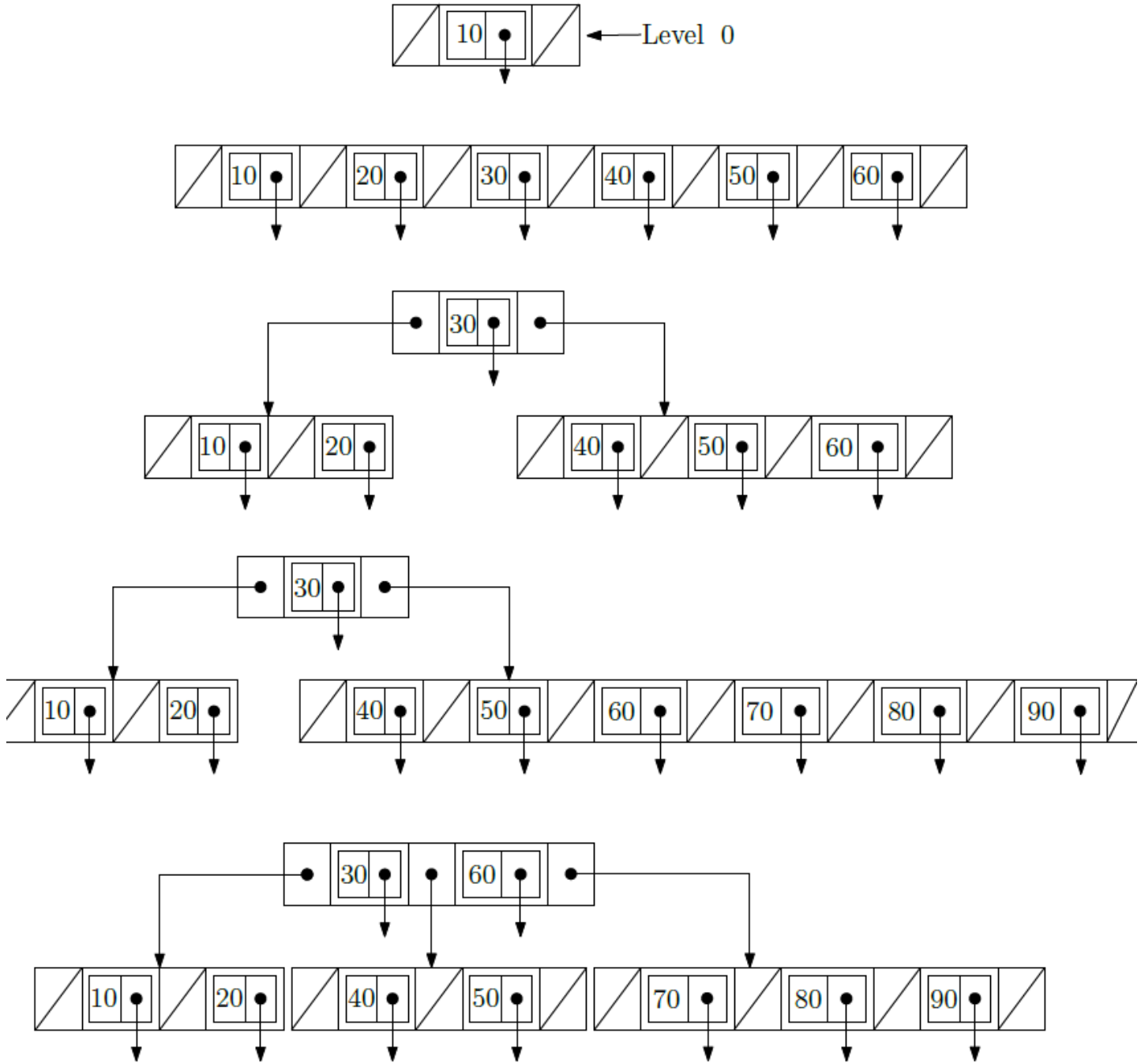


Insertion in B-trees

- A B-tree starts with a single root node (which is also a leaf node) at level 0
- Once the root node is full with $p - 1$ search key values and we attempt to insert another entry in the tree, the root node splits into two nodes at level 1
- Only the middle value is kept in the root node, and the rest of the values are split evenly between the other two nodes

Insertion in B-trees (Contd.)

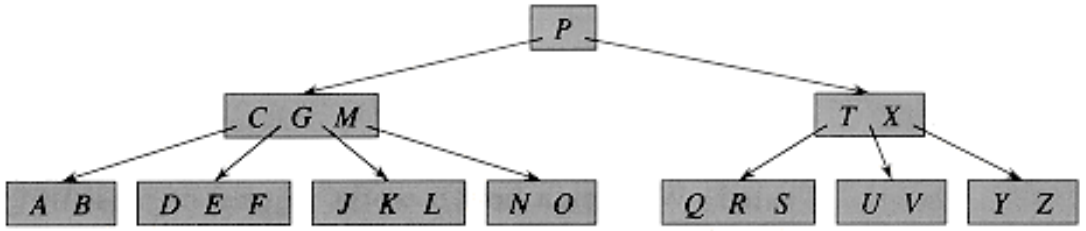
- When a nonroot node is full and a new entry is inserted into it, then
  - The nonroot node is split into two nodes at the same level, and the middle entry is moved to the parent node along with two pointers to the new split nodes
  - If the parent node is full, it is also split
  - Splitting can propagate all the way to the root node, creating a new level if the root is split

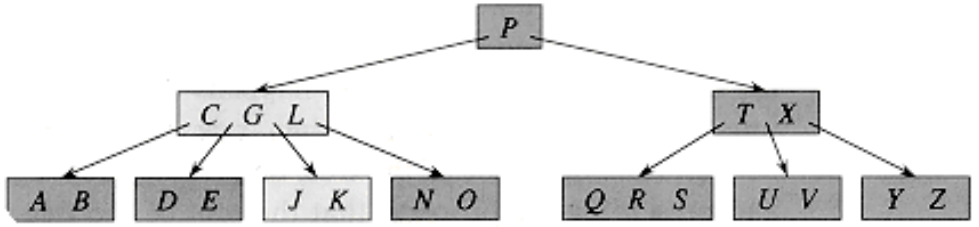

10
Level 0
10
20
30
40
50
60
30
10
20
40
50
60
30
10
20
40
50
60
70
80
90
30
60
10
20
40
50
70
80
90

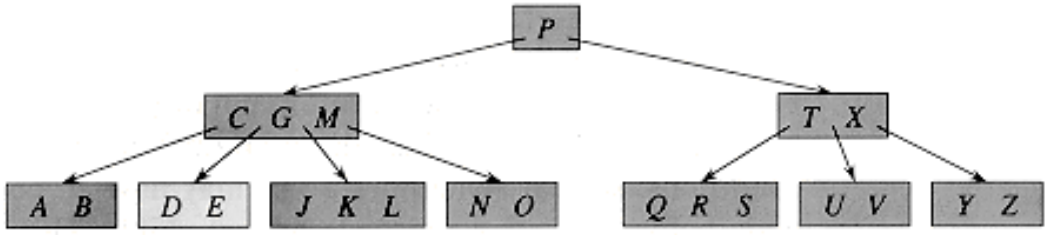
Deletion in B-trees (Contd.)

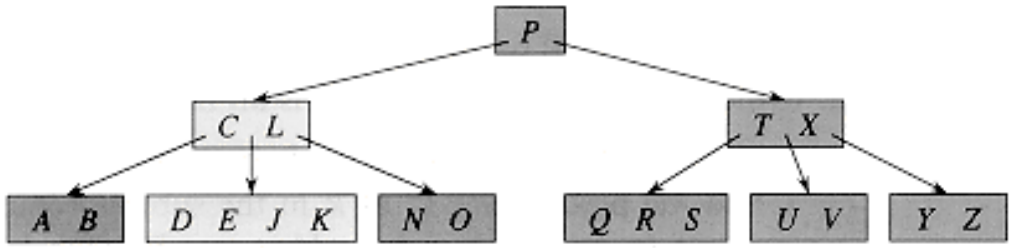
- If deletion of a value causes a node to be less than half full, it is combined with its neighboring nodes, and this can also propagate all the way to the root
  - Deletion can reduce the number of tree levels

Deletion in B-trees (Contd.)









Thank you!