Chapter 14, Indexing Structures for Files

- Data file is stored in secondary memory (disk) due to its large size.
- For accessing a record in data file, it may be necessary to read several blocks from disk to main memory before the correct block is retrieved.
- The index structure provides the more efficient access {fewer blocks access} to records in file based on the **indexing fields** that are used to construct the index.

14.1 Types of Single-level Ordered Indexes

- **Primary index**: specified on the ordering key field of an ordered file of records.
- **Clustering index**: if the ordering field is not a key field.
- **Secondary index**: specified on any nonordering field of a file.

14.1.1 Primary Indexes

- The records of the data file are physically ordered by the primary key.
- A **primary index** is an ordered file whose records are of fixed length with two fields.
- The first field in primary index is of the same data type as the ordering key field (primary key) of the data file.
- The second field in primary index is a pointer to a disk block containing the record.
- Total number of entries in the index is the same as the number of disk blocks in the ordered data file. This is an example of a **nondense index**.
- A binary search on the index file requires fewer block access than a binary search on the data file.
- See Figure 14.1 (Fig 6.1 on e3).
- Example 1:

  - r = 30,000 records;
  - block size $B = 1024$ bytes;
  - File records are fixed size with record length $R = 100$ bytes;
  - Blocking factor bfr (records per block) $= \lfloor B/R \rfloor = 10$ records;
  - Number of blocks needed is $b = \lceil r/bfr \rceil = 3000$ blocks;
  - A binary search on the data file would need $\lceil log_2 3000 \rceil = 12$ block accesses;
  - Suppose the ordering key field of the file is $V = 9$ bytes long;
  - A block pointer $P = 6$ bytes long;
  - Thus, the size of each index entry R_i in the primary index will be 15 bytes;
  - The blocking factor for the index is $bfr_i = \lfloor B/R_i \rfloor = 68$ entries per block;
  - The total index entries r_i is equal to the number of blocks of data file, which is 3000;
  - The number of index blocks is hence $b_i = \lceil r_i/bfr_i \rceil = 45$ blocks.
  - The binary search on the index file would need $\lceil log_2 b_i \rceil = 6$ block accesses.
  - To search a record using index, we need one additional block access to the data file for a total $6 + 1 = 7$ block accesses.

- A major problem with a primary index – as with any ordered file – is insertion and deletion of records.

14.1.2 Clustering Indexes

- If records of a file are physically ordered on a nonkey field that does not have a distinct value for each record, that field is called the **clustering field**.
- A clustering index is also an ordered file with two fields.
- The first field is of the same type as the clustering field of the data file.
- The second field is a block pointer.

- There is one entry in the clustering index for each distinct value of the clustering field, containing the value and a pointer to the *first block* in the data file that has a record with that value for its clustering field. See Figure 14.2 and 14.3 (Fig 6.2, 6.3 on e3).
- A clustering index is another example of **nondense index**.
- Insertion and deletion still cause problem, because the data records are physically ordered. Figure 14.3 (Fig 6.3 on e3) shows the way to alleviate the problem)

14.1.3 Secondary Indexes

- A secondary index is also an ordered file with two fields.
- The first field of the index is of the same data type as some *nonordering field* of the data file that is an indexing field.
- The second field of the index is either a block pointer or a record pointer.
- A secondary index can based on a nonordering key field.
  - It is a **dense** index if it is based on nonordering key field, since it contains one entry for each record in the data file.
  - A secondary index usually needs more storage space and longer search time than does a primary index, because of its larger number of entries. But improvement is much more because the alternative is linear search. See Figure 14.4 (Fig 6.4 on e3).
  - Example 2: (continued from Example 1)
    - Number of records r = 30,000;
    - Block size B = 1024 bytes;
    - A fixed record length R = 100 bytes;
    - bfr = 10 records per block;
    - Number of blocks of data file = 3000;
    - 1500 average block accesses for a linear search.

    * Suppose we create a secondary index on a nonordering key field.
    * The nonordering key field has 9 bytes long and the block pointer has 6 bytes long;
    * Each entry R_i in the index has 15 bytes long;
    * The blocking factor $bfr_i = \lfloor B/R_i \rfloor = 68$ entries per block;
    * Because the index is a dense index, the total number of entries r_i of the index is equal to r = 30,000.
    * Thus, the total number of blocks of the index will be $b_i = \lceil r_i/bfr_i \rceil = 442$ blocks.
    * A binary search on the index needs $\lceil log_2 b_i \rceil = 9$ block accesses.
    * 10 block accesses to retrieve a record using the secondary index.

- A secondary index can based on a nonordering nonkey field.
  - **Option 1** is to include several index entries with the same $K(i)$ value – one for each record. This would be a dense index.
  - **Option 2** is to have variable-length records for the index entries, with a repeating field for the pointer.
  - **Option 3** has a fixed length entries and has a single entry for each index field value, but create an extra level of indirection to handle the multiple pointers. Retrieval requires extra block accesses but searching and insertion are straightforward. See Figure 14.5 (Fig 6.5 on e3).

14.1.4 Summary

See Table 14.1 and Table 14.2

14.2 Multilevel Indexes

- The idea here is to reduce the search space by a factor of bfr_i (which is usually >> 2). The blocking factor values bfr_i is called fan out **fo**.
- A multilevel index requires approximately $log_{fo}\ b_i$ block accesses.

- A multilevel index considers the index file, which we will refer to as the **first level** of a multilevel index, as an ordered file with a distinct value of each $K(i)$.
- The **second level** index of a multilevel index is an index to the first level. Since the first level is physically ordered on $K(i)$, the second could have one entry for each block of the first level.
- If the first level has r_1 entries, and the blocking factor – which is also the fan-out – for the index is $bfr_i = fo$, then the first level need $\lceil r_1/fo \rceil$ blocks, which is therefore the number of entries r_2 needed at the second level of the index.
- We can repeat the same process until all the entries of some index level t fit in a single block. This block in t^{th} level is call **top** index level.
- $t = \lceil log_{fo}\ (r_1) \rceil$.
- The multilevel index described here can be used on any type of index, whether it is primary, clustering, or secondary – as long as the first level index has distinct values for $K(i)$, fixed-length entries, and is ordered on $K(i)$. See Figure 14.6 (Fig 6.6 on e3).
- Example 3: (continued from Example 2)
  - Suppose that the dense secondary index of example 2 is converted into a multilevel index.
  - The blocking factor is $bfr_i = 68$ entries per block.
  - The number of blocks in the first level is 442 blocks.
  - The number of blocks in the second level will be $\lceil 442/68 \rceil = 7$ blocks.
  - The number of blocks in the third level will be $\lceil 7/68 \rceil = 1$ block.
  - The total block accesses for a search will be $3 + 1 = 4$.
- To retain the benefits of using multilevel indexing while reducing index insertion and deletion problems, a **dynamic multilevel index** is implemented by using data structures called $B - tree$ and $B^+ - trees$.

14.3 Dynamic Multilevel Indexes Using B-Tree and B$^+$-Trees

14.3.1 Search Trees and B-Trees

- A **search tree** is slightly different from a multilevel index.

  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 $q \leq p$. All search values are assumed to be unique.

  - Within each node, $K_1 < K_2 < \ldots < K_{q-1}$
  - 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 $i = q$. See Figure 14.8 (Fig 6.8 on e3).

- In general, we want insertion/deletion algorithms that keep the tree balanced (i.e., all leaf nodes at the same level)

- **B-Trees:** A B-tree of order p when used as an access structure on a key field to search for records in a data file, can be defined as follows. See Figure 14.10 (Fig 6.10 on e3).

  - **Definition:**

    * Each internal node in the B-tree, in Figure 14.10(a) (Fig 6.10(a) on e3), 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 $q \leq p$.
    * Within each node, $K_1 < K_2 < \ldots < K_{q-1}$
    * $K_{i-1} < X < K_i$ for $1 < i < q$; $X < K_i$ for $i = 1$; and $K_{i-1} < X$ for $i = q$.
    * Each node has at most p pointers.
    * Each node, except the root node, 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).

    - 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.

- What do we do if search key is nonkey in data file ?

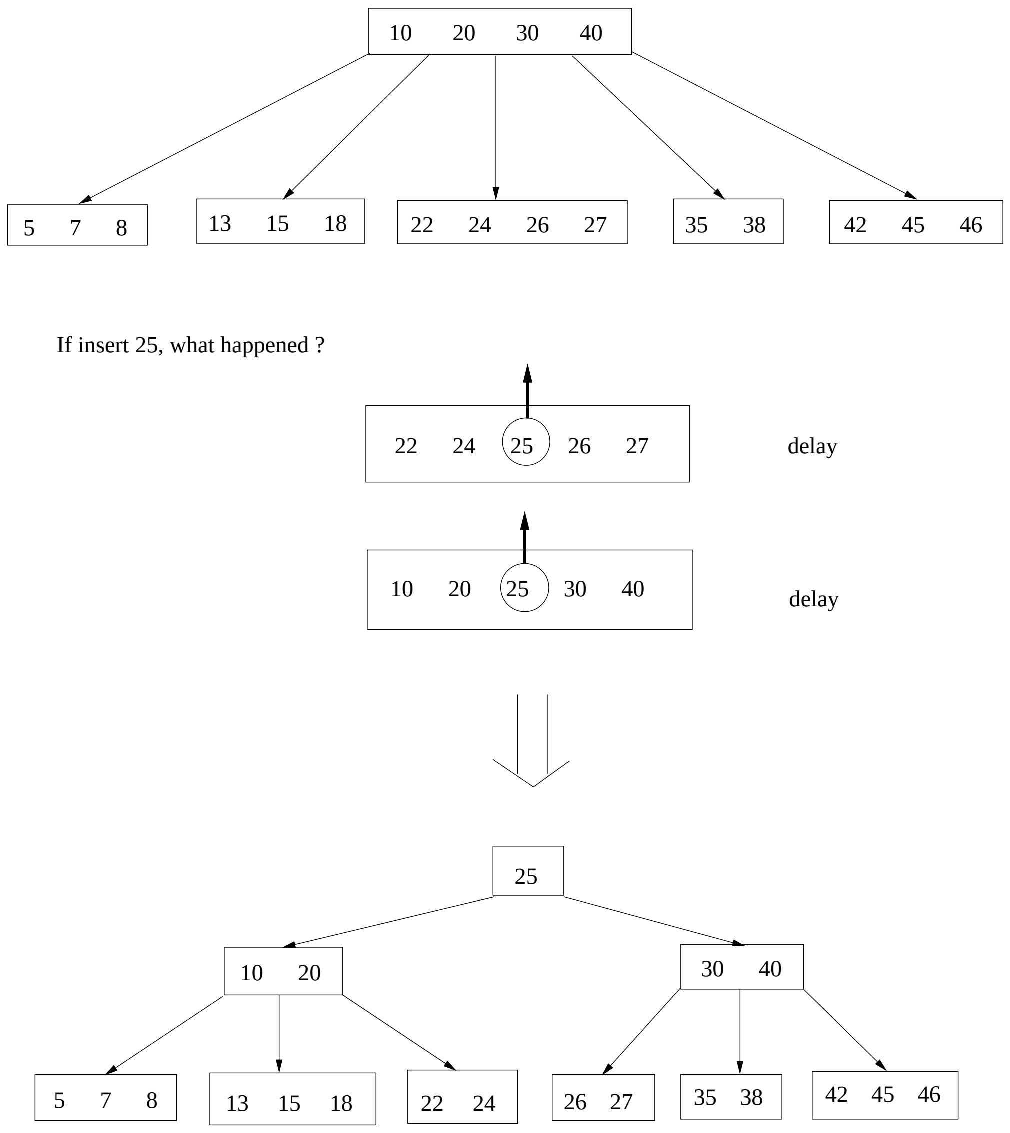
  Keep a linked list of data pointers for each search value.

- **Insertion for B-Tree:** Let p be the order of the B-tree.

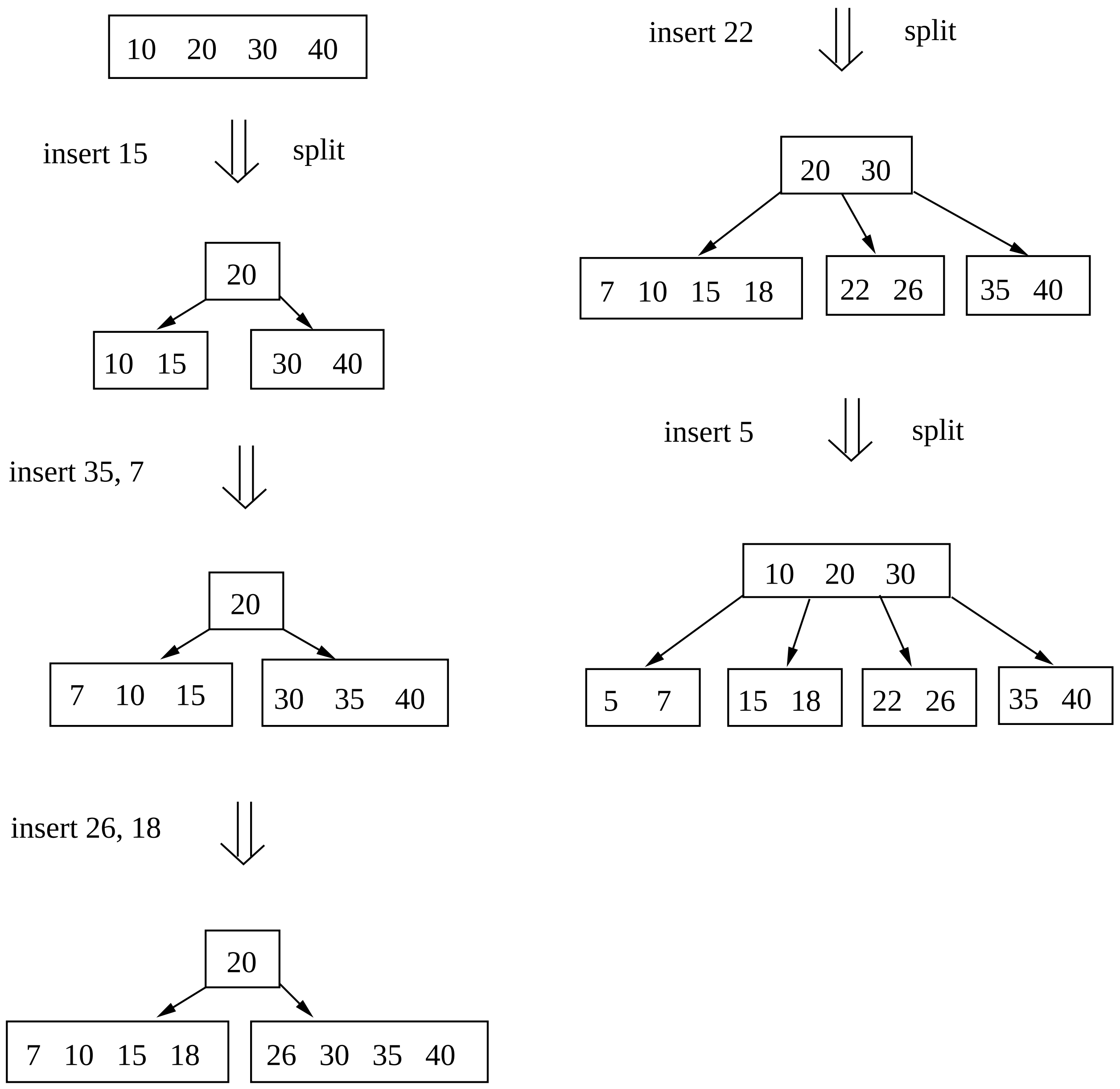
    - B-Tree starts with a single root node at level 0.
    - If any node at level i is full with $p - 1$ search key values and we attempt to insert another value into the node, this node is split into two nodes at the same level i and the middle value goes up to the parent node. If the parent node is full, it is also split. This can propagate all the way to the root, creating a new root if the old root is split.

– Example for B-Tree insertions:

  * If exists a B-Tree with $p = 5$ as below. What happened if we insert 25 ?

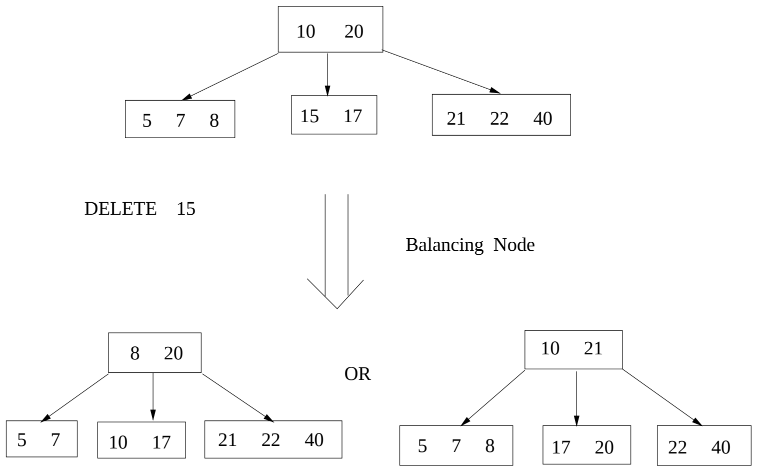


∗ Let $p = 5$; input: 20, 40, 10, 30, 15, 35, 7, 26, 18, 22, 5.

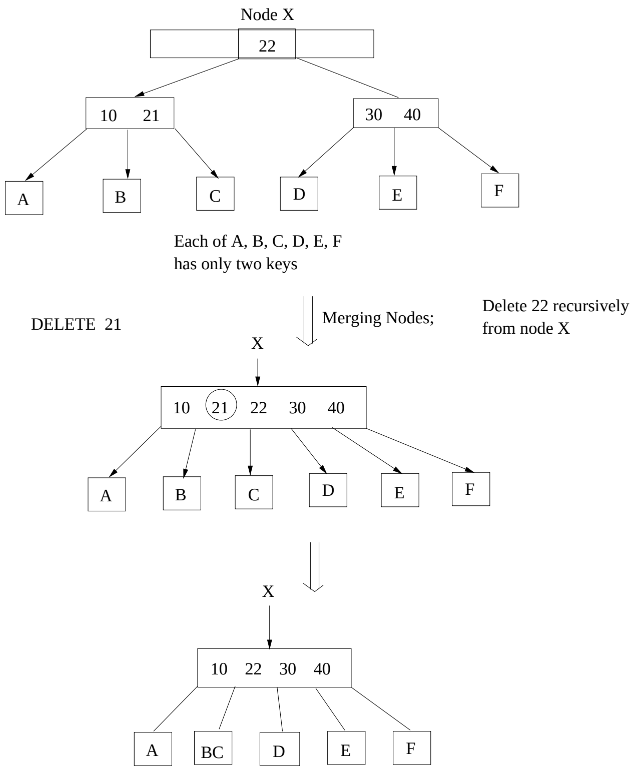


– **Deletion for B-Tree:** Dealing with underflow.

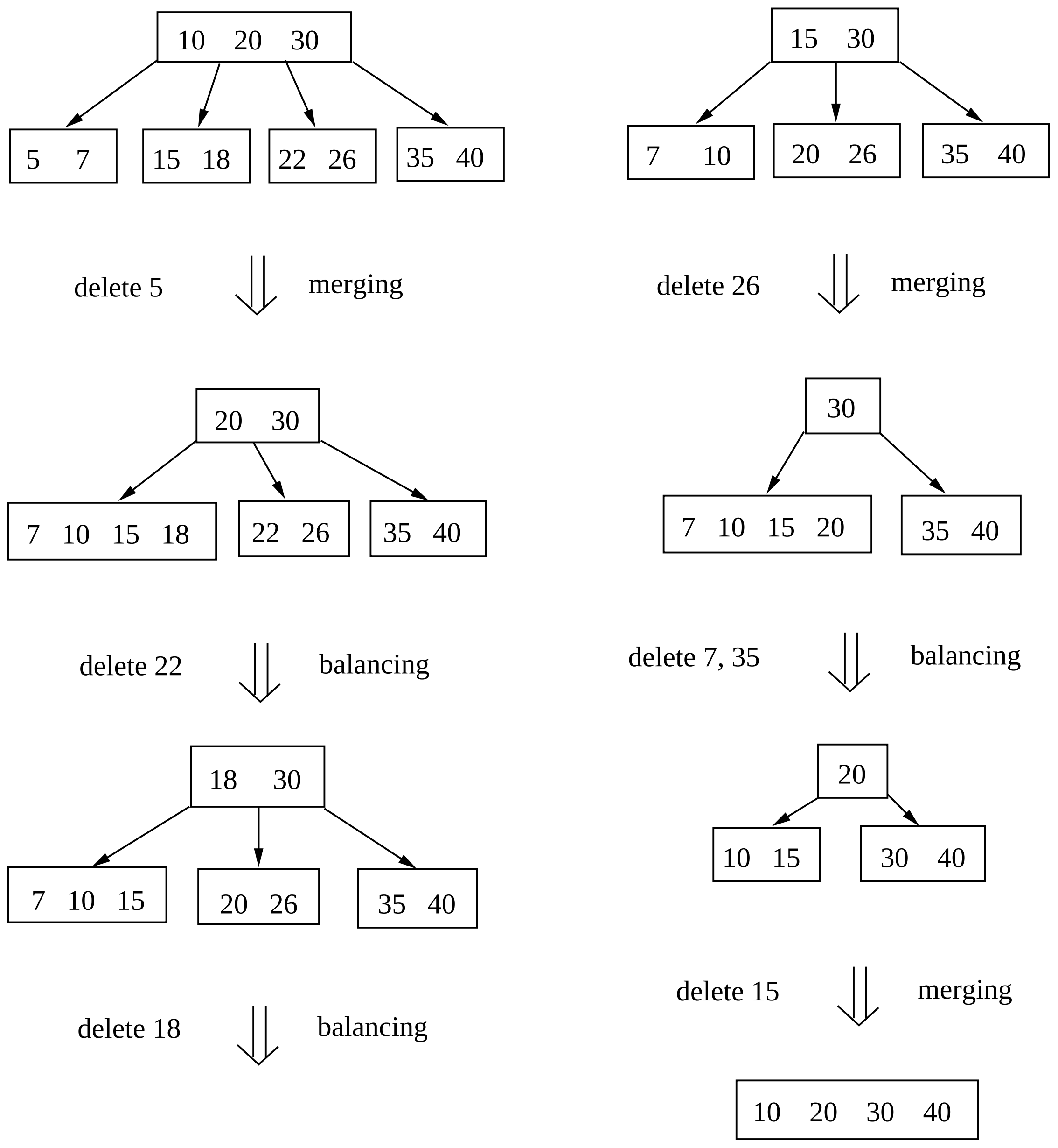
  * **Balancing nodes:** If there are extra keys in its left or right subtrees (left or right siblings if the value being deleted is in the leaf node). Let $p = 5$.



  * **Merging nodes:** If there are no extra keys in its left and right subtrees. Let $p = 5$.



- Example for B-Tree deletions:
    * Let $p = 5$; delete 5, 22, 18, 26, 7, 35, 15 from the B-Tree below.



- Example 4: How to choose p for B-Trees.
  - Suppose the search field is $V = 9$ bytes long.
  - Suppose the disk block size is $B = 512$ bytes.
  - Suppose a record (data) pointer is $P_r = 7$ bytes.
  - Suppose a block pointer is $P = 6$ bytes.
  - Each B-Tree nodes can have at most p tree pointers, $p-1$ data pointers, and $p-1$ search key field values.
  - Suppose each B-Tree node is to correspond to a disk block. Then we have:

$$(p \times P) + ((p-1) \times (P_r + V)) \leq B$$

$$(p \times 6) + ((p-1) \times (7+9)) \leq 512$$

$$(22 \times p) \leq 512$$

    Then, we can choose $p = 23$ (not 24, because we want to have extra space for addition information)

- Example 5: examine the capacity of the B-Tree.
  - Suppose that the search field of Example 4 is a nonordering key field, and we construct a B-Tree on this field.
  - Assume that each node of the B-Tree is 69 percent full.
  - Each node, on the average, will have $p \times 0.69 = 23 \times 0.69$ or approximately 16 pointers and, hence, 15 search key field values.
  - The average fan-out $fo = 16$. Then

Root:	1 nodes	15 entries	16 pointers
Level 1:	16 nodes	240 entries	256 pointers
Level 2:	256 nodes	3840 entries	4096 pointers
Level 3:	4096 nodes	61,440 entries	
Total		65,535 entries	

- B-Tree can also be used as primary file organizations. In this case, whole records are stored within the B-Tree nodes rather than the search key and pointers.

14.3.2 B$^+$-Trees

- Every value of the search field appears once at some level in a B-Tree.
- In a B$^+$-Tree, data pointers are stored only at the leaf nodes; hence, the structure of leaf nodes differs from the structure of internal nodes.
- A leaf node entry: < search value, data pointer >.
- An internal node entry: < search value, tree pointer >.
- Leaf nodes are usually linked together to provide ordered access on the search field to the records.
- Some search field values from the leaf nodes are repeated in the internal nodes to guide the search.
- **Definition:**
  - The structure of **internal nodes:**
    - Each internal node is of the form

      $$< P_1, K_1, P_2, K_2, \ldots, P_{q-1}, K_{q-1}, P_q >$$

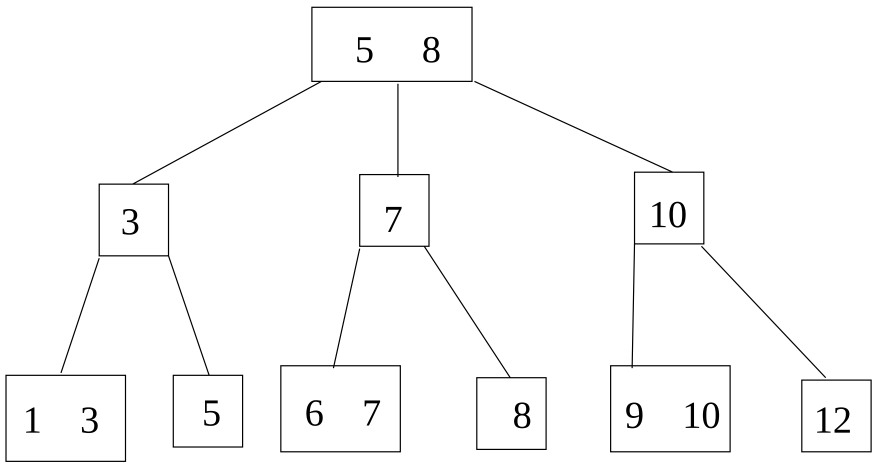
      Where $q \leq p$ and each P_i is a tree pointer.
    - Within each internal node, $K_1 < K_2 < \ldots < K_{q-1}$.
    - For all search field values X in the subtree pointed at by P_i, we have $K_{i-1} < X \leq K_i$ for $1 < i < q$; $X \leq K_i$ for $i = 1$; and $K_{i-1} < X$ for $i = q$, See Figure 14.11(a) (Fig 6.11(a) on e3).
    - Each internal node has at most p tree pointers.
    - Each internal node, except the root, has at least $\lceil p/2 \rceil$ tree pointers. The root node has at least two tree pointers if it is an internal node.
    - An internal node with q pointers, $q \leq p$, has $q - 1$ search field values.
  - The structure of **leaf nodes:**

- ∗ 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, and P_{next} points to the next leaf node of the B$^+$-Tree.
- ∗ Within each leaf node, $K_1 < K_2 < \ldots < K_{q-1}$, where $q \leq p$.
- ∗ 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.
- ∗ Each leaf node has at least $\lceil p/2 \rceil$ values.
- ∗ all leaf nodes are at the same level.

- Example of B$^+$-Tree with $p = 3$ for internal nodes and $p_{leaf} = 2$ for leaf nodes.



- Every value appearing in an internal node also appears as the *rightmost value* in the leaf level of the subtree pointed at by the tree pointer to the left of the value.

- Every key value must exist at the leaf level, because all data pointers are at the leaf level. Only some values exist in internal nodes to guide the search.

- Example 6: To calculate the orders p and p_{leaf} of a B$^+$-Tree.

  - Search key field $V = 9$ bytes.

  - The block size $B = 512$ bytes.

- A record pointer $Pr = 7$ bytes.
- A tree pointer $P = 6$ bytes.
- p tree pointers and $p - 1$ search values in an internal node (block)

$$(p \times P) + ((p-1) \times V) \leq B$$

$$(p \times 6) + ((p-1) \times 9) \leq 512$$

  We choose $p = 34$.

- p_{leaf} data pointers and P_{leaf} search values in a leaf node (block)

$$(p_{leaf} \times (Pr + V)) + P \leq B$$

$$(p_{leaf} \times (7+9)) + 6 \leq 512$$

  we choose $p_{leaf} = 31$.

- Example 7: examine the capacity of the B^{+}-Tree.
  - Assume that each node of the B^{+}-Tree is 69 percent full.
  - Each internal node, on the average, will have $p \times 0.69 = 34 \times 0.69$ or approximately 23 pointers and, hence, 22 search key field values.
  - Each leaf node, on the average, will have $p_{leaf} \times 0.69 = 31 \times 0.69$ or approximately 21 data record pointers and, hence, 21 search key field values.
  - The B^{+}-Tree will have the following average number of entries at each level.

Root:	1 nodes	22 entries	23 pointers
Level 1:	23 nodes	506 entries	529 pointers
Level 2:	529 nodes	11,638 entries	12,167 pointers
Leaf level:	12,167 nodes	255,507 entries	

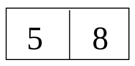
  - Compare to B-Tree (in example 5) which has 65,535 entries.

- **Insertion** for B^{+}-Trees.
  - **Splitting a leaf node because of overflow:** When a leaf node is full and a new value is inserted there, the node **overflows** and must be split. The first

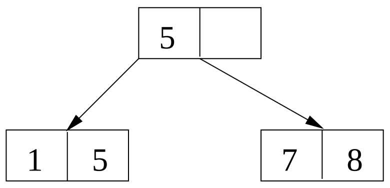
$j = \lceil (P_{leaf}+1)/2 \rceil$ entries are kept in the original node, and the remaining entries are moved to a new leaf node. The j^{th} value is replicated in the parent internal node.

- If the parent internal is full, the new value (the j^{th} value in above step) will cause it to overflow also, so it must be split.
- **Splitting an internal node because of overflow:** the entries in the internal node up to P_j – the j^{th} tree pointer after inserting the new value and pointer, where $j = \lfloor (p+1)/2 \rfloor$ – are kept, while the j^{th} search value in moved to the parent, not replicated. A new internal node will hold the entries from P_{j+1} to the end of the entries in the node.
- This splitting can propagate all the way up to create a new root and hence a new level for the B$^+$-Tree.

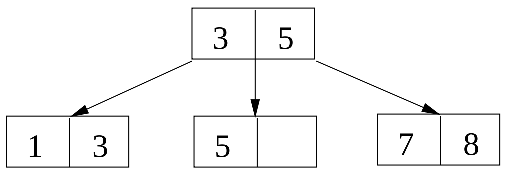
- Example of the B$^+$-Tree insertion: Let a B$^+$-Tree with $p = 3$ and $p_{leaf} = 2$. Try to create the B$^+$-Tree by inserting the following sequence of numbers – 8, 5, 1, 7, 3, 12, 9, 6 (See next page).



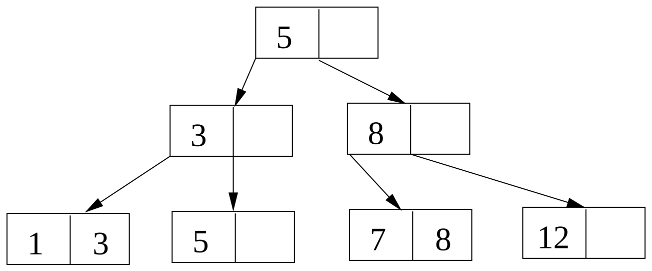
After inserting 8, 5



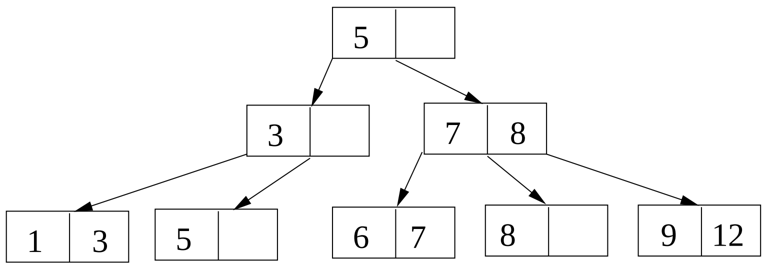
After inserting 1, 7



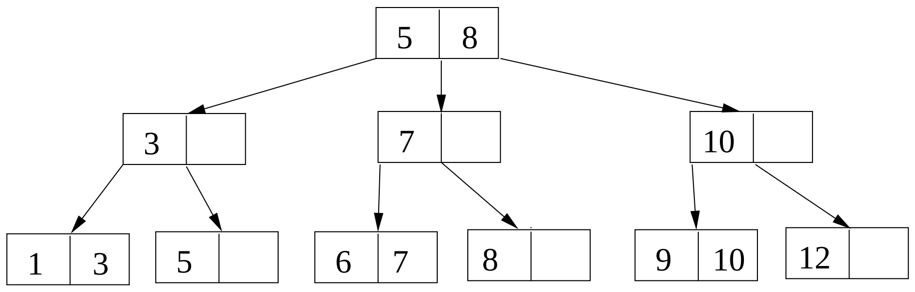
After inserting 3



After inserting 12



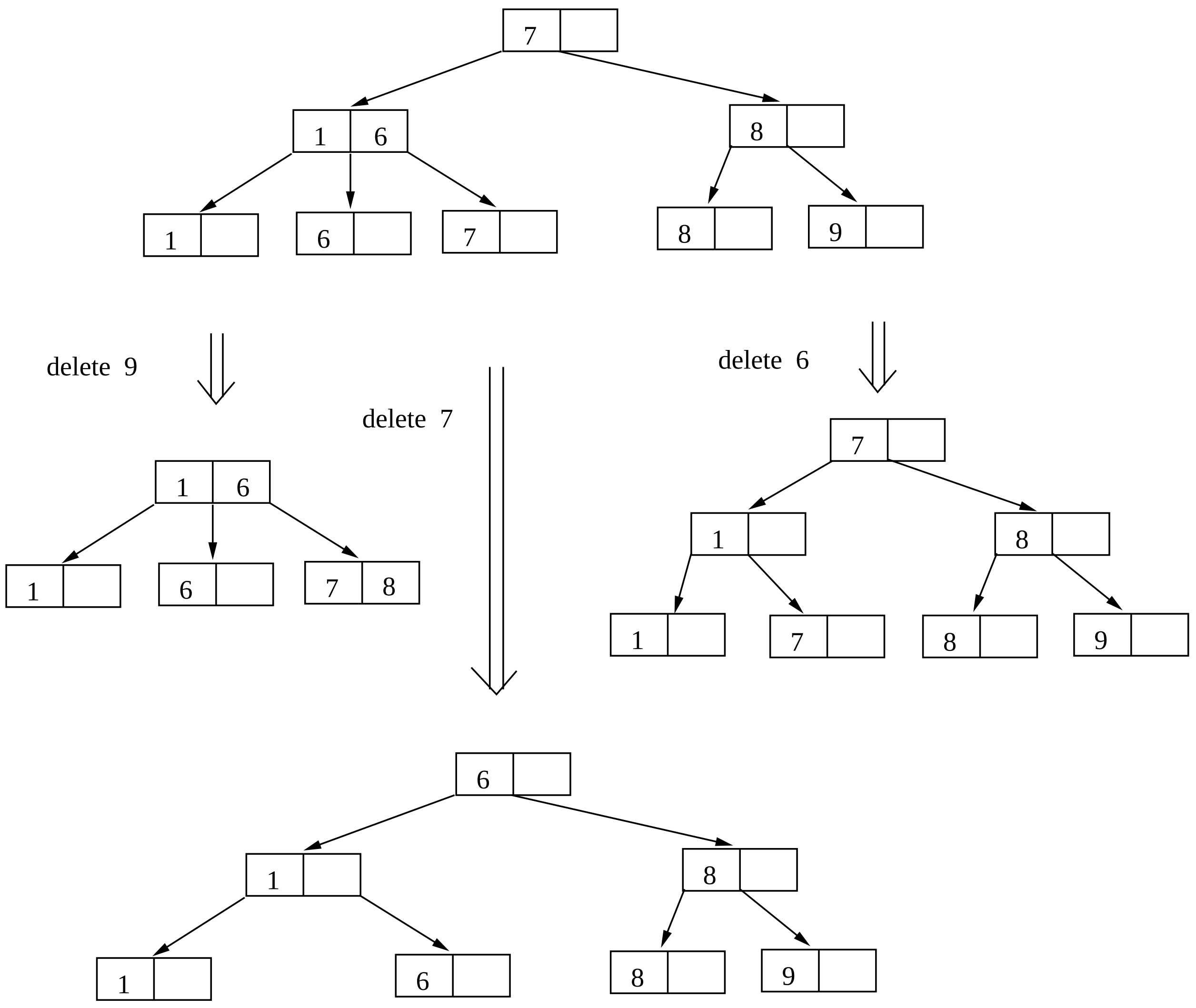
After inserting 9, 6



After inserting 10

- **Deletion** for B$^+$-Trees:
  - When an entry is deleted, it is always removed from the leaf node. If it happens to occur in an internal node, it must also be removed from there. In the latter case, the value to its left in the leaf node must replace it in the internal node.
  - **Underflow in the leaf node:**
    - Try to **redistribute (balancing)** the entries from the left sibling if possible.
    - Try to **redistribute (balancing)** the entries from the right sibling if possible.
    - If the redistribution is not possible, then the three nodes are merged into two leaf nodes. In this case, underflow may propagate to internal nodes because one fewer tree pointer and search value are needed. This can propagate and reduce the tree levels.
- Example of the B$^+$-Tree deletion:
  - see Figure 14.13 (Fig 6.13 on e3).

– Let $p = 3$ and $p_{leaf} = 2$; delete 9 or 6 or 7 from a B$^+$-Tree below.


7
1
6
8
1
6
7
8
9
delete 9
delete 7
delete 6
1
6
1
6
7
8
7
1
8
1
7
8
9
6
1
8
1
6
8
9