Physical Level of Databases: B+-Trees

Adnan YAZICI
Computer Engineering Department
METU

(Fall 2005)

B^+-Tree Index Files

B^+-tree indices are an alternative to indexed-sequential files.

- Disadvantage of indexed-sequential files: performance degrades as file grows, since many overflow blocks get created. Periodic reorganization of entire file is required.
- Advantage of B^+-tree index files: automatically reorganizes itself with small and local changes, in the face of insertions and deletions. Reorganization of entire file is not required to maintain performance.
- Disadvantage of B^+-trees: extra insertion and deletion overhead, space overhead.
- Advantages of B^+-trees outweigh disadvantages, and they are used extensively.

B^+-Tree Index Files (Cont.)

A B^+-tree is a rooted tree satisfying the following properties:

- All paths from root to leaf are of the same length
- Each node that is not a root or a leaf has between [n/2] and n children.
- Special cases:
  - If the root is not a leaf, it has at least 2 children.
  - If the root is a leaf (that is, there are no other nodes in the tree), it can have between 0 and (n–1) values.

B+-Trees as Dynamic Multi-level Indexes

- **B+-Tree** is a variation of search trees that allow efficient insertion and deletion of new search values.
- In **B+-Tree** , each node corresponds to a disk block.
- Each node is kept between half-full and completely-full.
- 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 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.

B^+-Tree Node Structure

- Typical node

P_1	K_1	P_2	...	P_{n-1}	K_{n-1}	P_n

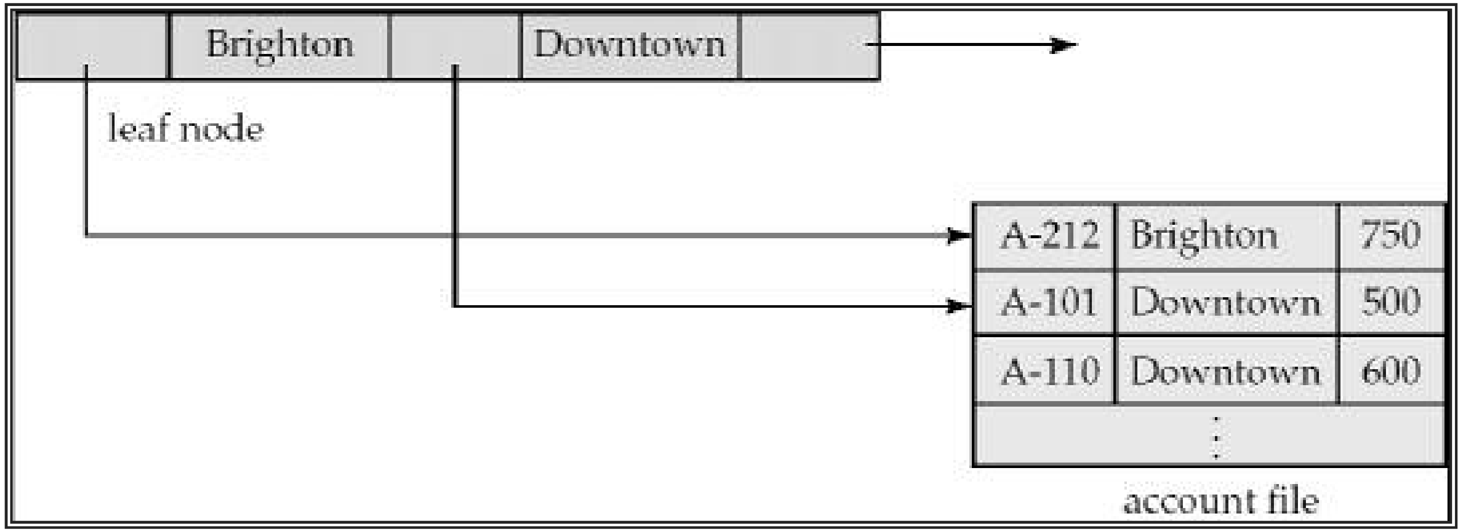
  - K_i are the search-key values
  - P_i are pointers to children (for non-leaf nodes) or pointers to records or buckets of records (for leaf nodes).
- The search-keys in a node are ordered

$$K_1 < K_2 < K_3 < \ldots < K_{n-1}$$

Leaf Nodes in B$^+$-Trees

Properties of a leaf node:

- For $i = 1, 2, \ldots, n$, pointer P_i either points to a file record with search-key value K_i, or to a bucket of pointers to file records, each record having search-key value K_i. Only need bucket structure if search-key does not form a primary key.
- If L_i, L_j are leaf nodes and $i < j$, L_i's search-key values are less than L_j's search-key values
- P_n points to next leaf node in search-key order

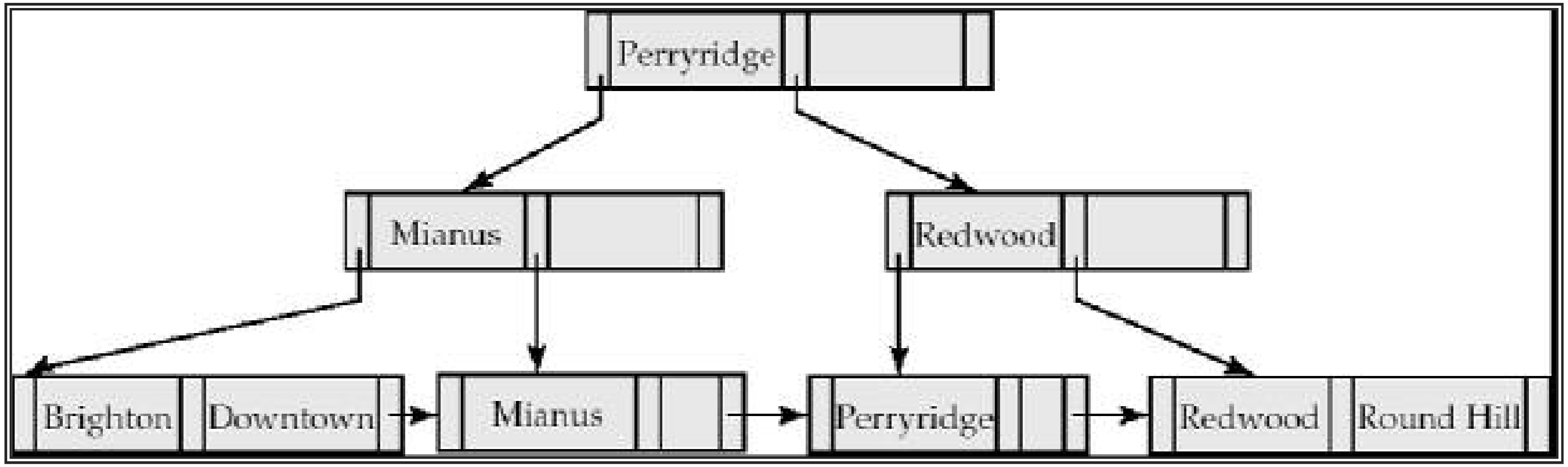


Non-Leaf Nodes in B+-Trees

- Non leaf nodes form a multi-level sparse index on the leaf nodes. For a non-leaf node with *m* pointers:
  - All the search-keys in the subtree to which P_1 points are less than K_1
  - For $2 \leq i \leq n - 1$, all the search-keys in the subtree to which P_i points have values greater than or equal to K_{i-1} and less than K_{m-1}

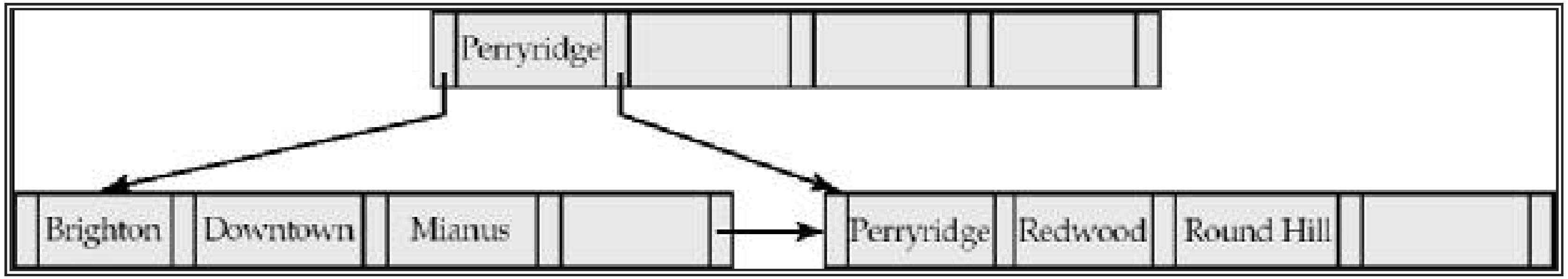
P_1	K_1	P_2	$\ldots$	P_{n-1}	K_{n-1}	P_n

Example of a B^+-tree



B^+-tree for *account* file ($n = 3$)

Example of B$^+$-tree



B$^+$-tree for *account* file (n = 5)

- All nodes must have between 2 and 4 key values ($\lfloor (n)/2 \rfloor$ and n, with n = 5).
- Root must have at least 2 children.

Observations about B^+-trees

- Since the inter-node connections are done by pointers, “logically” close blocks need not be “physically” close.
- The non-leaf levels of the B^+-tree form a hierarchy of sparse indices.
- The B^+-tree contains a relatively small number of levels (logarithmic in the size of the main file), thus searches can be conducted efficiently.
- Insertions and deletions to the main file can be handled efficiently, as the index can be restructured in logarithmic time (as we shall see).

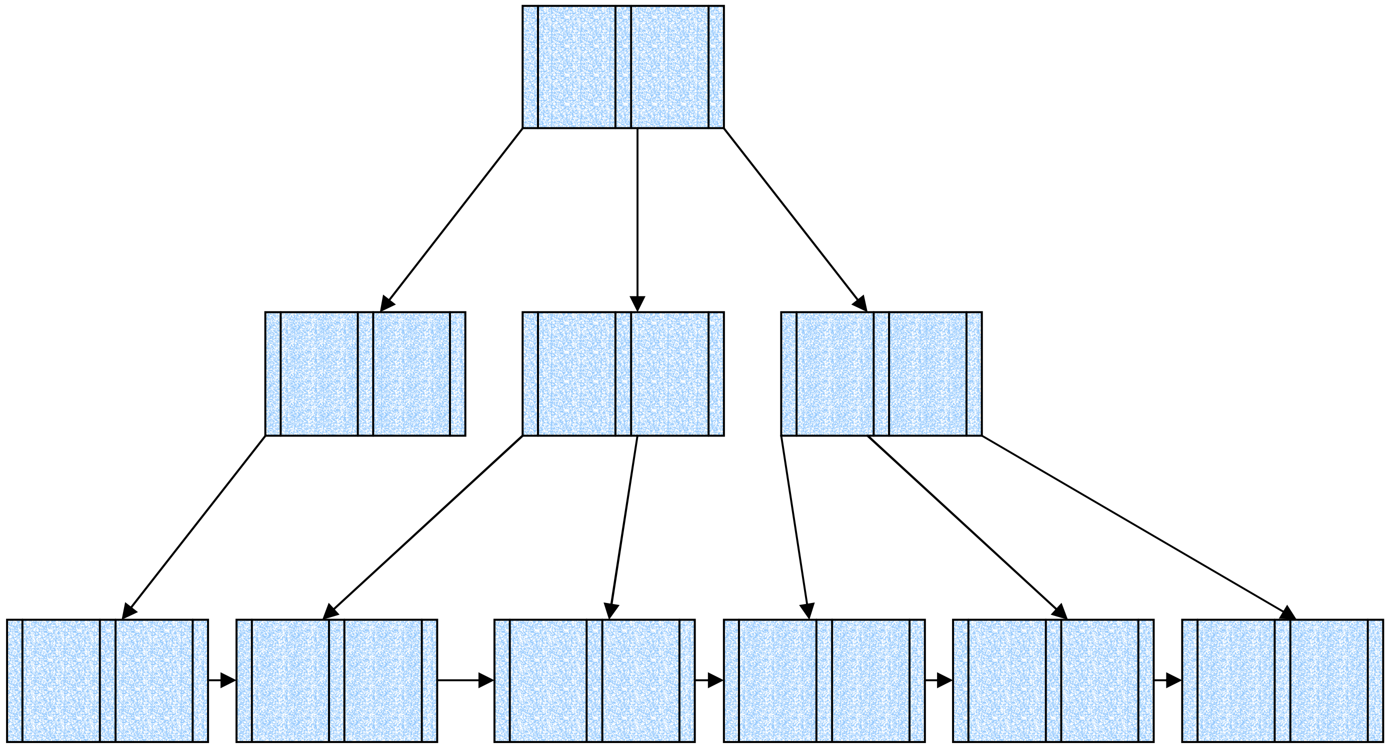
Queries on B^+-Trees

- Find all records with a search-key value of *k.*
  1. Start with the root node
     1. Examine the node for the smallest search-key value > *k.*
     2. If such a value exists, assume it is K_j. Then follow P_i to the child node
     3. Otherwise $k \geq K_{m-1}$, where there are *m* pointers in the node. Then follow P_m to the child node.
  2. If the node reached by following the pointer above is not a leaf node, repeat step 1 on the node
  3. Else we have reached a leaf node.
     1. If for some *i*, key K_i = k follow pointer P_i to the desired record or bucket.
     2. Else no record with search-key value *k* exists.

Queries on B^{+-}Trees (Cont.)

- In processing a query, a path is traversed in the tree from the root to some leaf node.
- If there are K search-key values in the file, the path is no longer than $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$.
- A node is generally the same size as a disk block, typically 4 kilobytes, and n is typically around 100 (40 bytes per index entry).
- With 1 million search key values and n = 100, at most log_{50}(1,000,000) = 4 nodes are accessed in a lookup.
- Contrast this with a balanced binary free with 1 million search key values — around 20 nodes are accessed in a lookup
  - above difference is significant since every node access may need a disk I/O, costing around 20 milliseconds!

B+ Tree

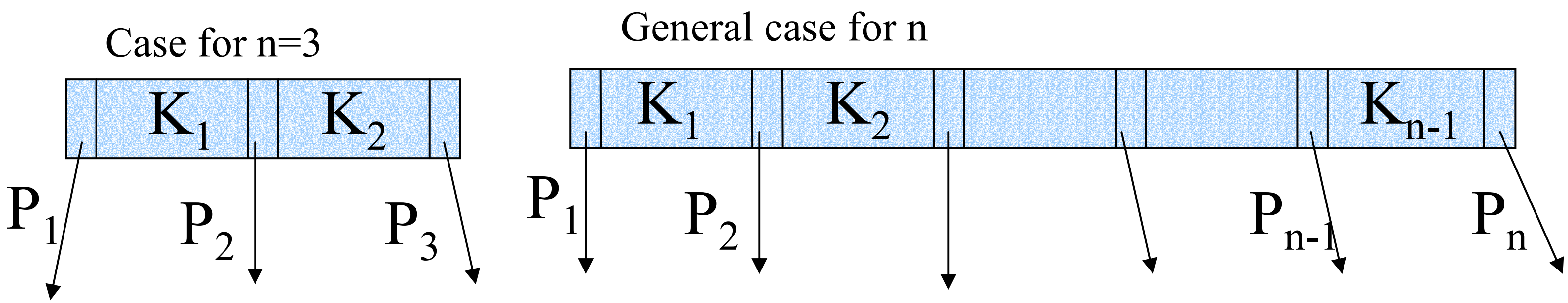


B+ Tree

- B+ Tree Properties
- B+ Tree Searching
- B+ Tree Insertion
- B+ Tree Deletion

B+ Tree

– Balanced Tree
  - Same height for paths from root to leaf
  - Given a search-key K, nearly same access time for different K values
– B+ Tree is constructed by parameter **n**
  - Each Node (except root) has $\lceil n/2 \rceil$ to n pointers
  - Each Node (except root) has $\lceil n/2 \rceil$-1 to n-1 search-key values



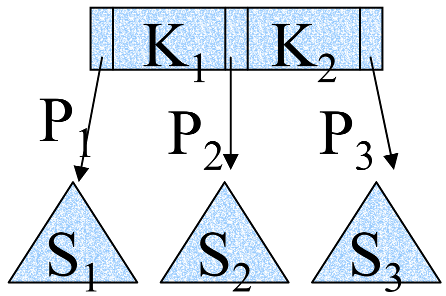
B+ Tree

- Search keys are sorted in order
  - $K_1 < K_2 < \ldots < K_{n-1}$

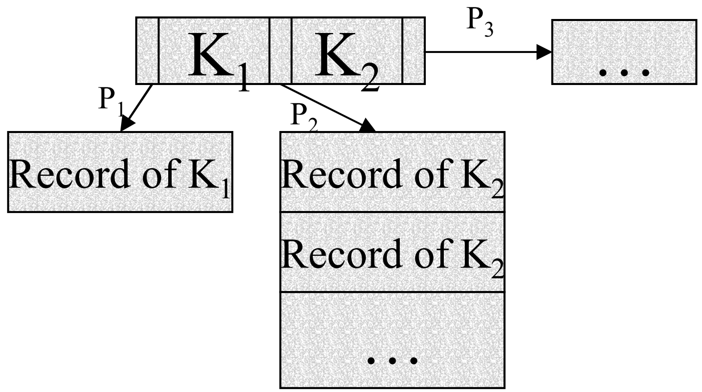
- Non-leaf Node
  - Each key-search values in subtree S_i pointed by $P_i < K_i$, $>= K_{i-1}$

    Key values in $S_1 < K_1$

    $K_1 <=$ Key values in $S_2 < K_2$

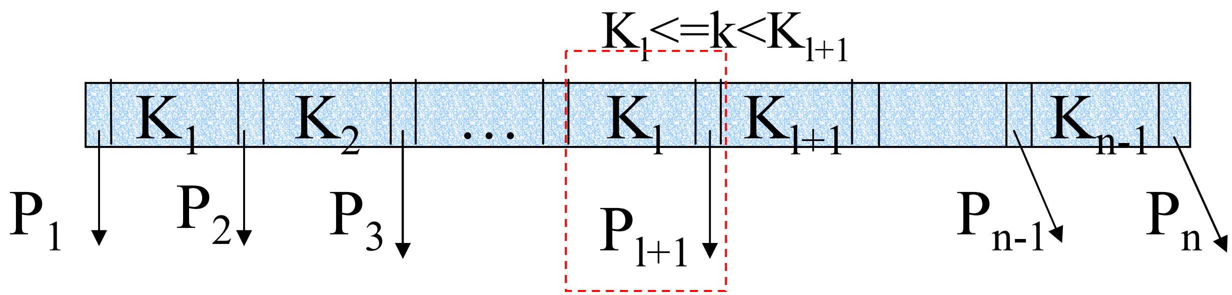


- Leaf Node
  - P_i points record or bucket with search key value K_i
  - P_n points to the neighbor leaf node

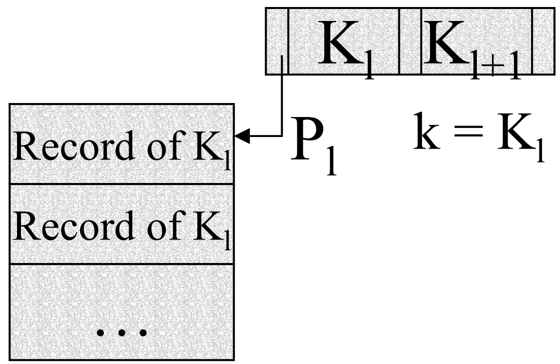


B+ Tree: Search

- Given a search-value k
  - Start from the root, look for the **largest** search-key value (K_l) in the node <= k
  - Follow pointer P_{l+1} to next level, until reach a leaf node

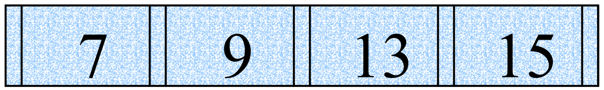


  - If k is found to be equal to K_l in the leaf, follow P_l to search the record or bucket



B+ Tree:Insertion

- Overflow
  - When the number of search-key values exceed n-1



Insert 8

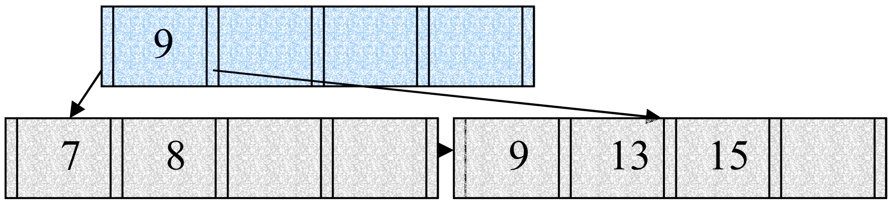
–Leaf Node

•Split into two nodes:

–1^{st} node contains $\lceil (n-1)/2 \rceil$ values

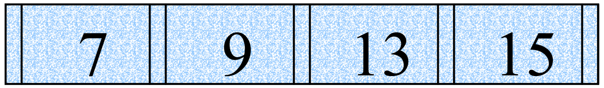
–2^{nd} node contains remaining values

–Copy the smallest search-key value of the 2^{nd} node to parent node



B+ Tree: Insertion

- Overflow
  - When number of search-key values exceed n-1



Insert 8

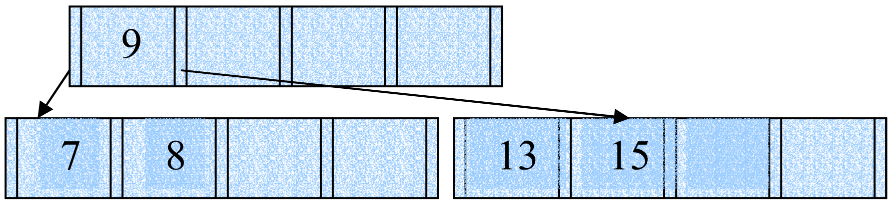
–Non-Leaf Node

•Split into two nodes:

–1st node contains $\lceil n/2 \rceil$-1 values

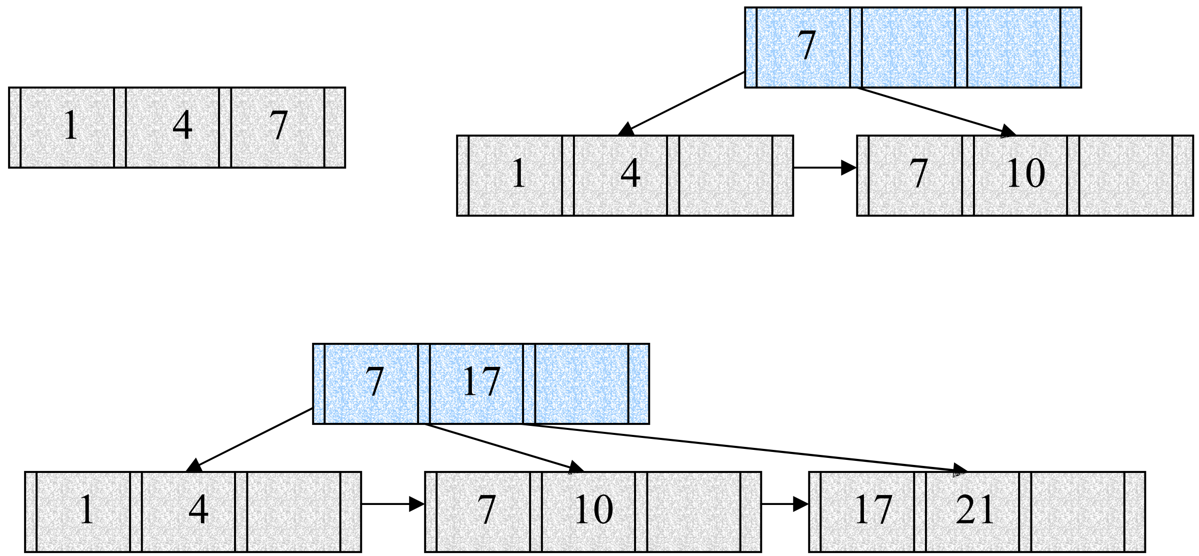
–Move the smallest of the remaining values, together with pointer, to the parent

–2nd node contains the remaining values



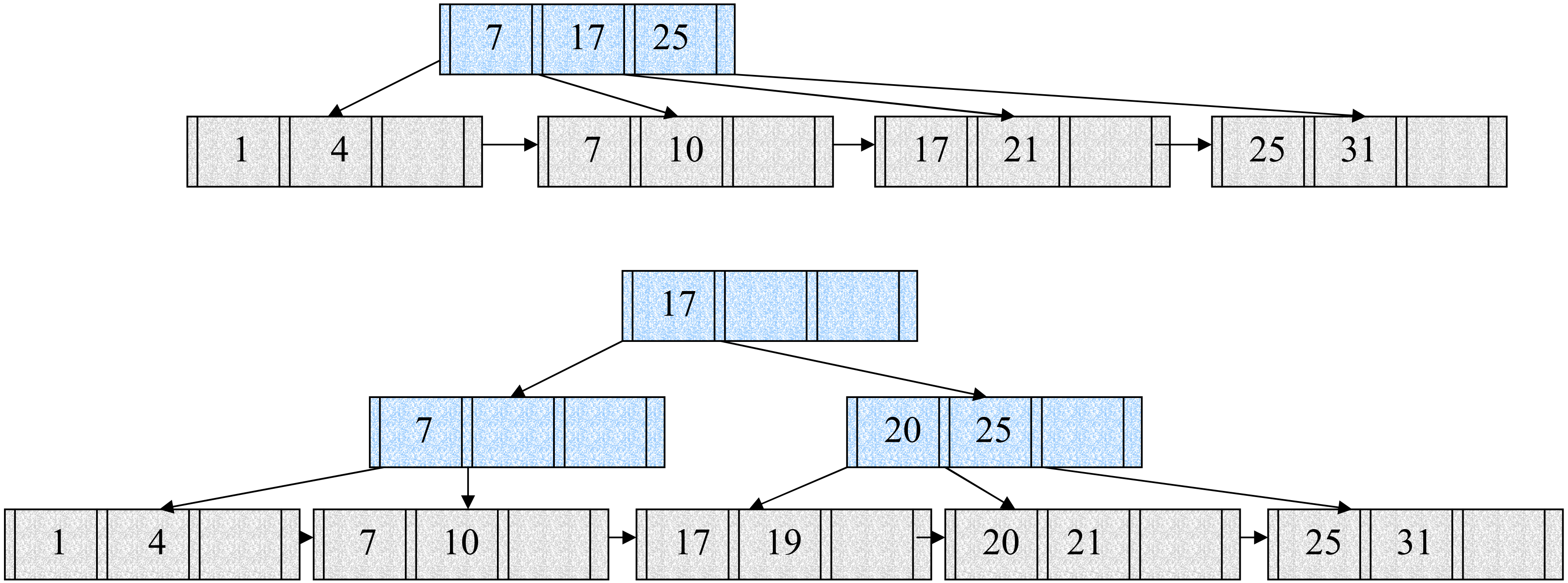
B+ Tree: Insertion

- Example 1: Construct a B^+ tree for (1, 4, 7, 10, 17, 21, 31, 25, 19, 20, 28, 42) with n=4.



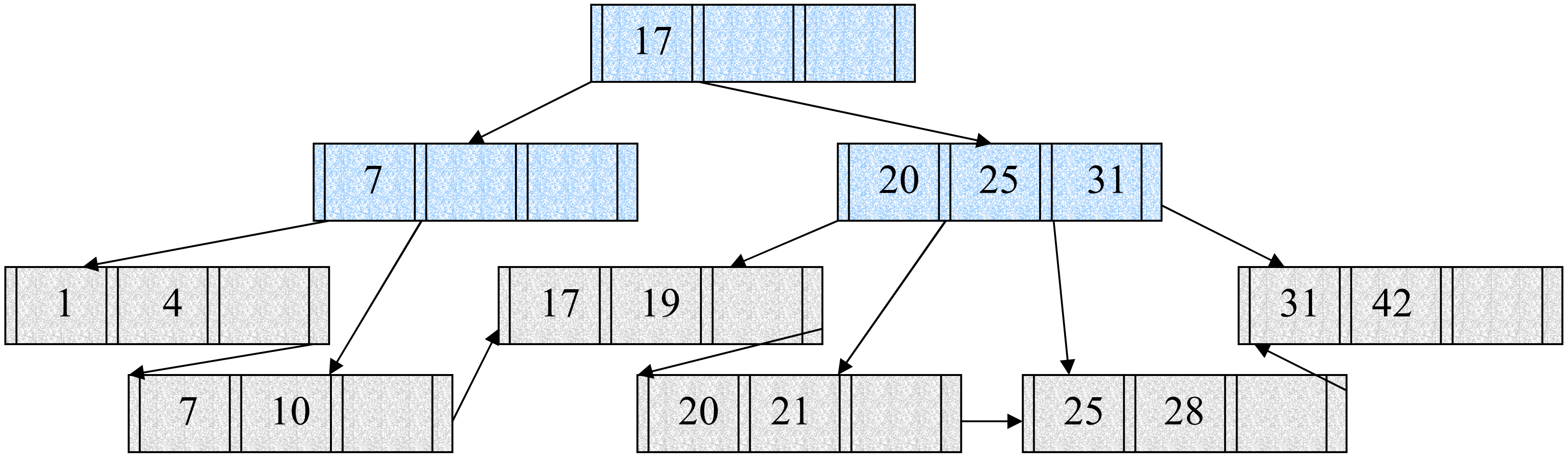
B+ Tree: Insertion

- 1, 4, 7, 10, 17, 21, 31, 25, 19, 20, 28, 42



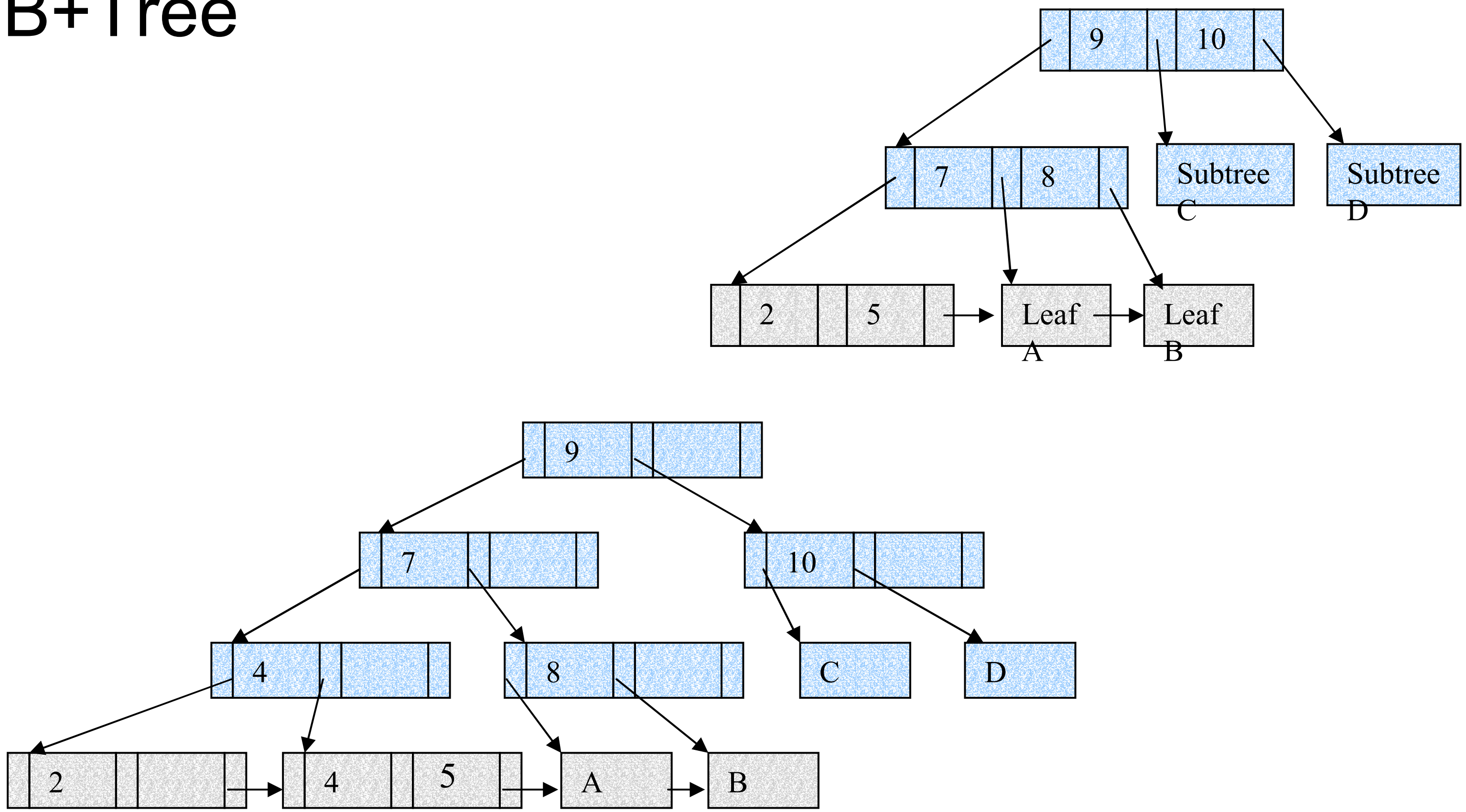
B+ Tree: Insertion

- 1, 4, 7, 10, 17, 21, 31, 25, 19, 20, 28, 42



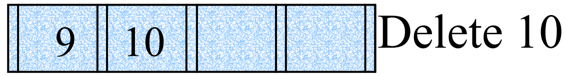
B+ Tree: Insertion

- Example 2: n=3, insert 4 into the following B+Tree



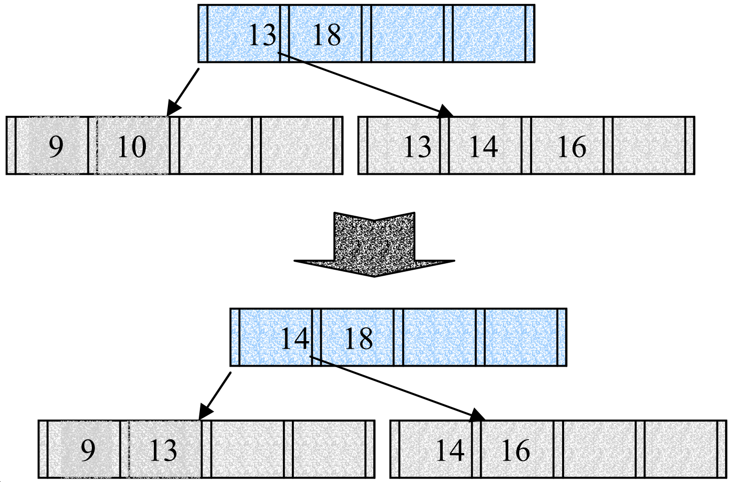
B+ Tree: Deletion

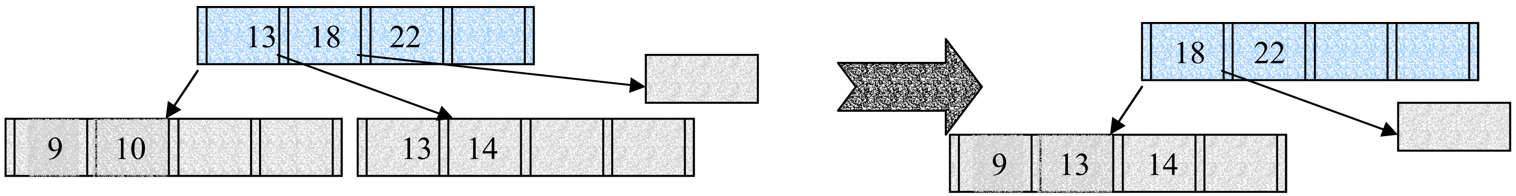
- Underflow
  - When number of search-key values < ⌈n/2⌉-1



–Leaf Node

- •Redistribute to sibling
  - •Right node not less than left node
  - •Replace the between-value in parent by their smallest value of the right node
- •Merge (contain too few entries)
  - •Move all values, pointers to left node
  - •Remove the between-value in parent

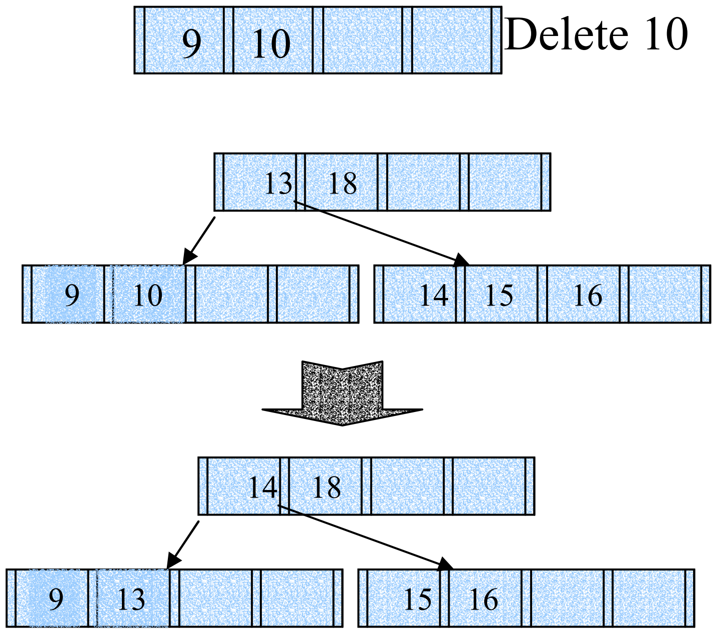


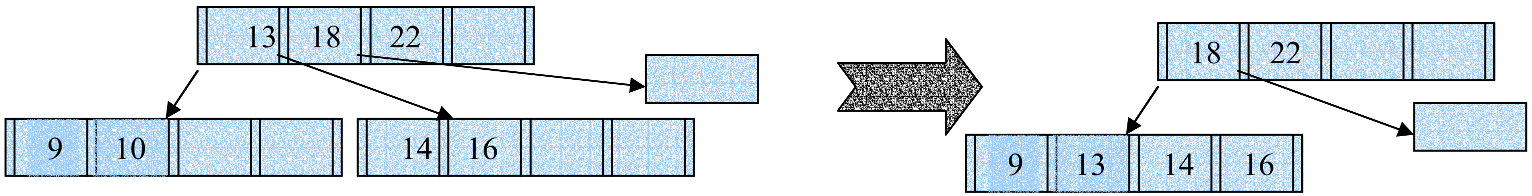


B+ Tree: Deletion

–Non-Leaf Node

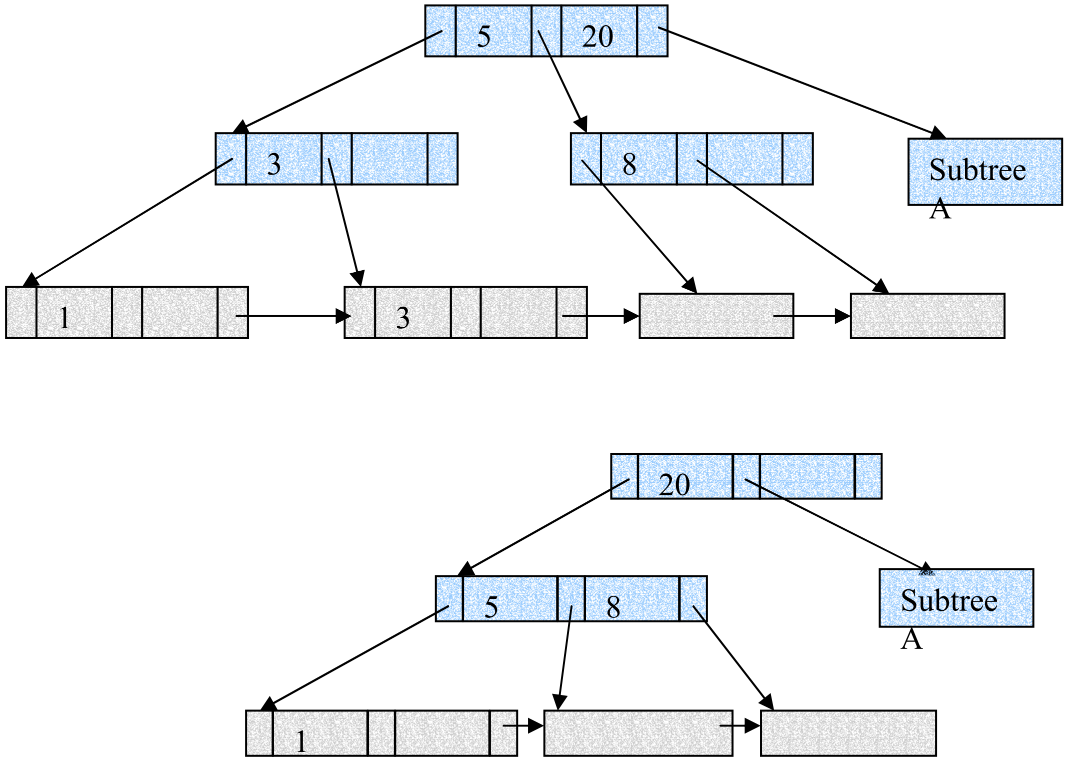
- •Redistribute to sibling
  - •Through parent
  - •Right node not less than left node
- •Merge (contain too few entries)
  - •Bring down parent
  - •Move all values, pointers to left node
  - •Delete the right node, and pointers in parent





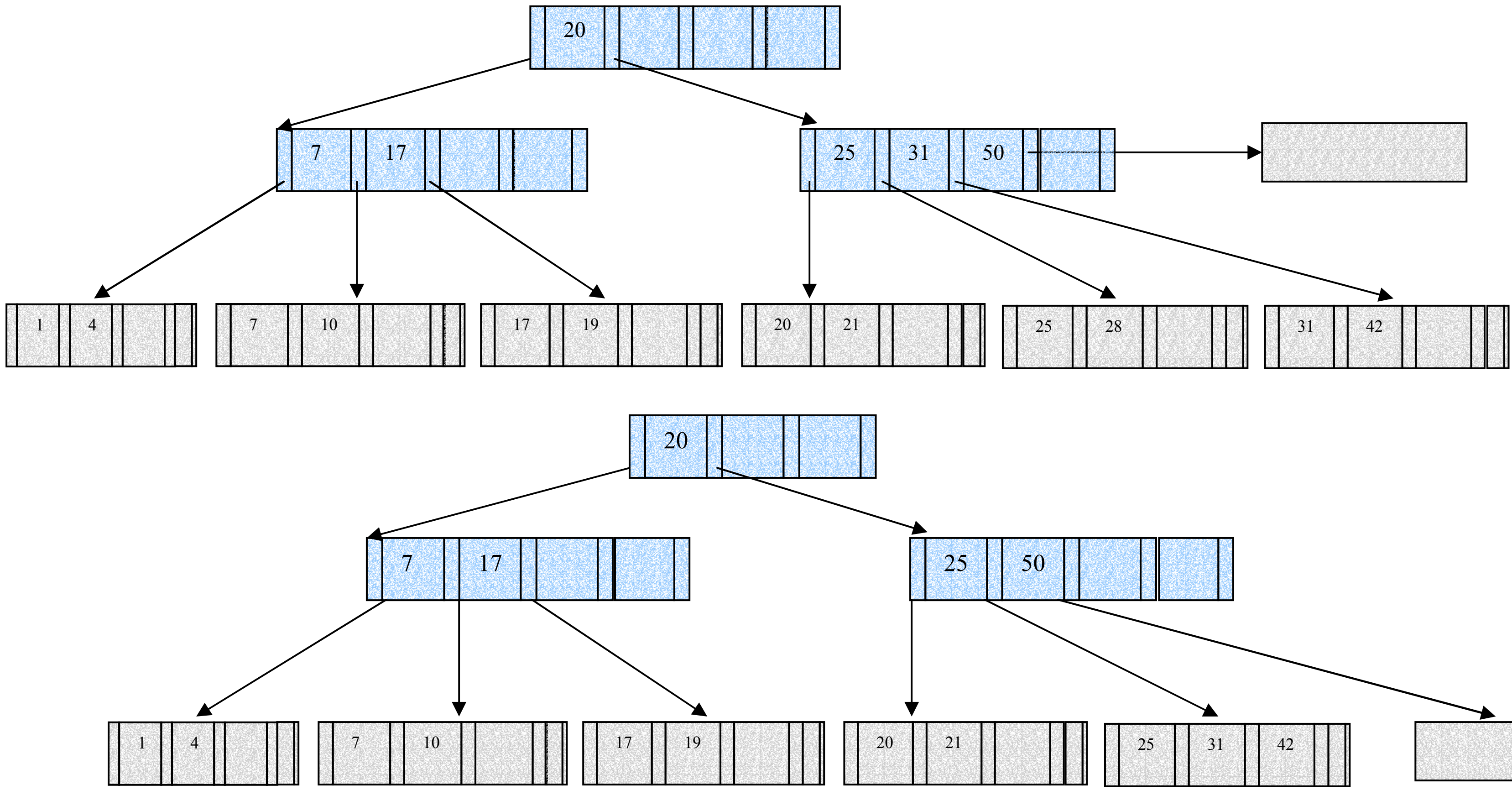
B+ Tree: Deletion

- Example 3: n=3, delete 3



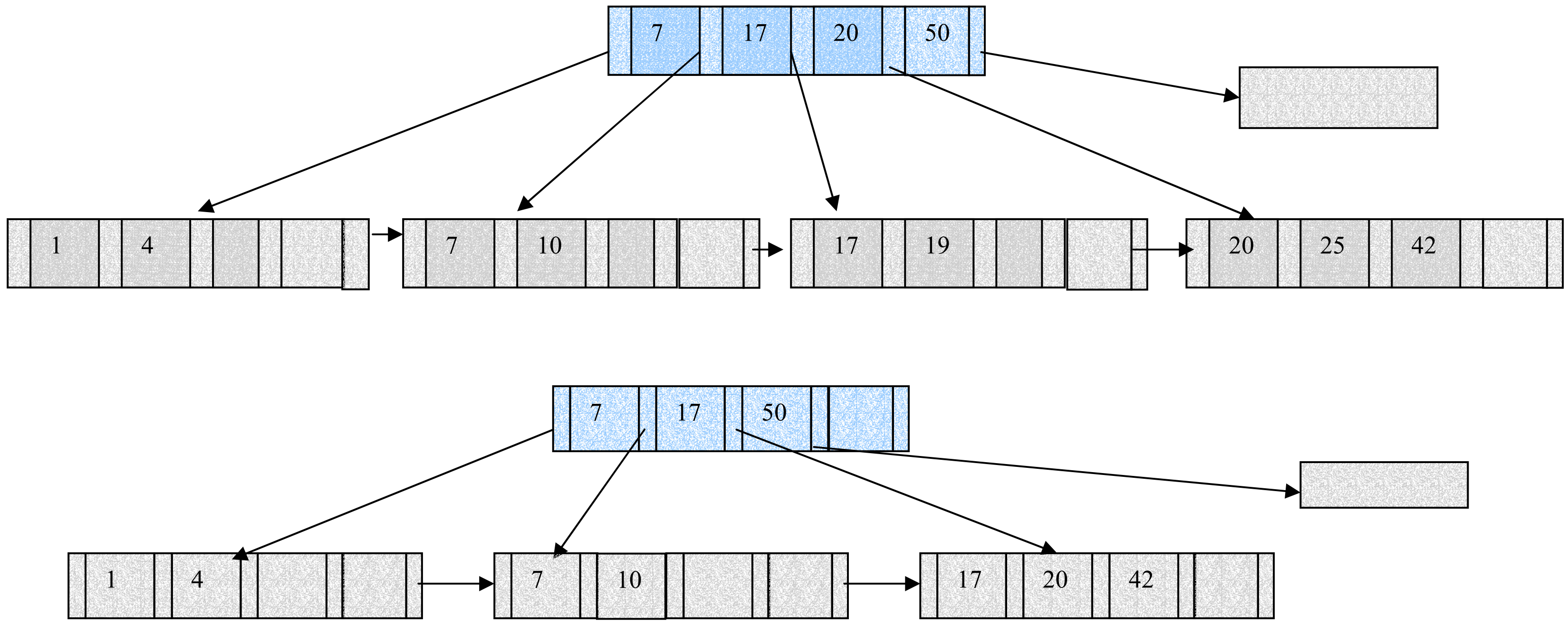
B+ Tree: Deletion

- Example 4: Delete 28, 31, 21, 25, 19



B+ Tree: Deletion

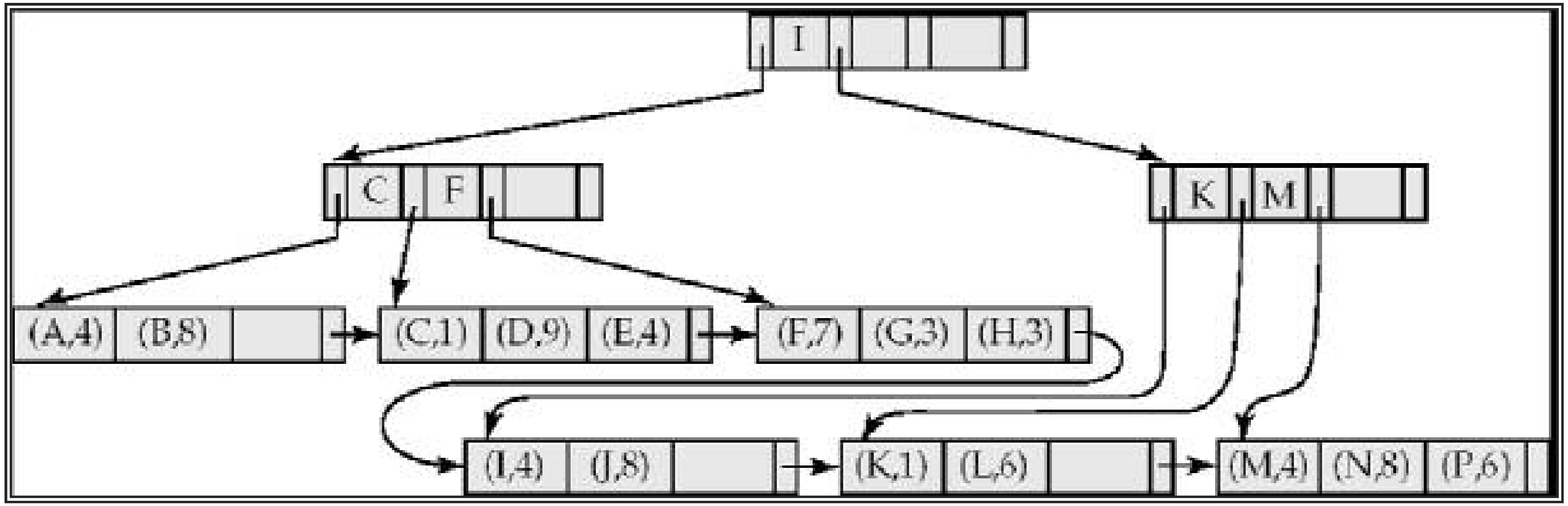
- Example 4: Delete 28, 31, 21, 25, 19



B^+-Tree File Organization

- Index file degradation problem is solved by using B^+-Tree indices. Data file degradation problem is solved by using B^+-Tree File Organization.
- The leaf nodes in a B^+-tree file organization store records, instead of pointers.
- Since records are larger than pointers, the maximum number of records that can be stored in a leaf node is less than the number of pointers in a nonleaf node.
- Leaf nodes are still required to be half full.
- Insertion and deletion are handled in the same way as insertion and deletion of entries in a B^+-tree index.

B^+-Tree File Organization (Cont.)



Example of B^+-tree File Organization

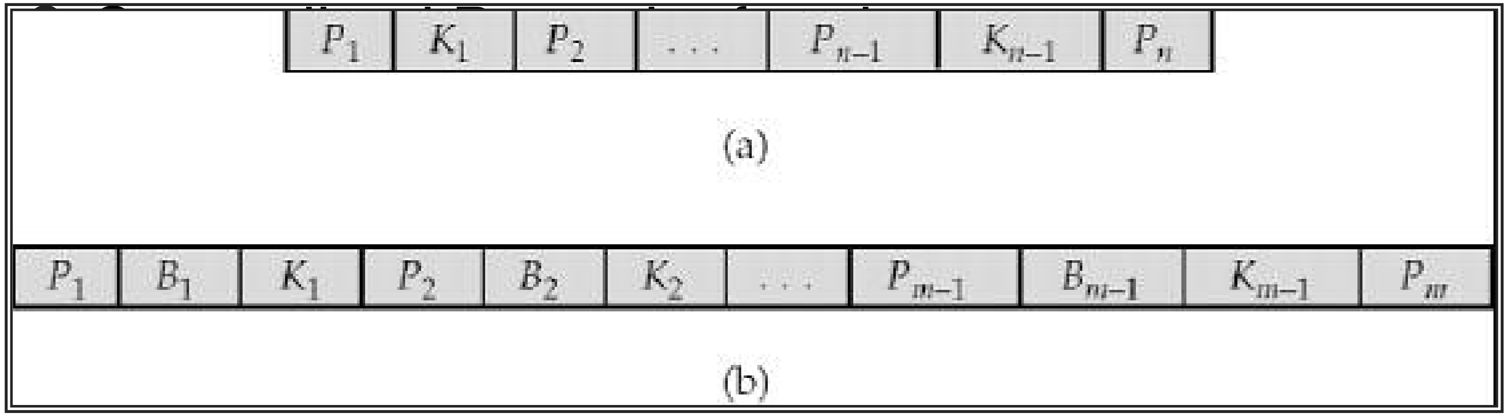
- Good space utilization important since records use more space than pointers.
- To improve space utilization, involve more sibling nodes in redistribution during splits and merges
  - Involving 2 siblings in redistribution (to avoid split / merge where possible) results in each node having at least entries

Indexing Strings

- Variable length strings as keys
  - Variable fanout
  - Use space utilization as criterion for splitting, not number of pointers
- Prefix compression
  - Key values at internal nodes can be prefixes of full key
    - Keep enough characters to distinguish entries in the subtrees separated by the key value
      - E.g. “Silas” and “Silberschatz” can be separated by “Silb”

B-Tree Index Files

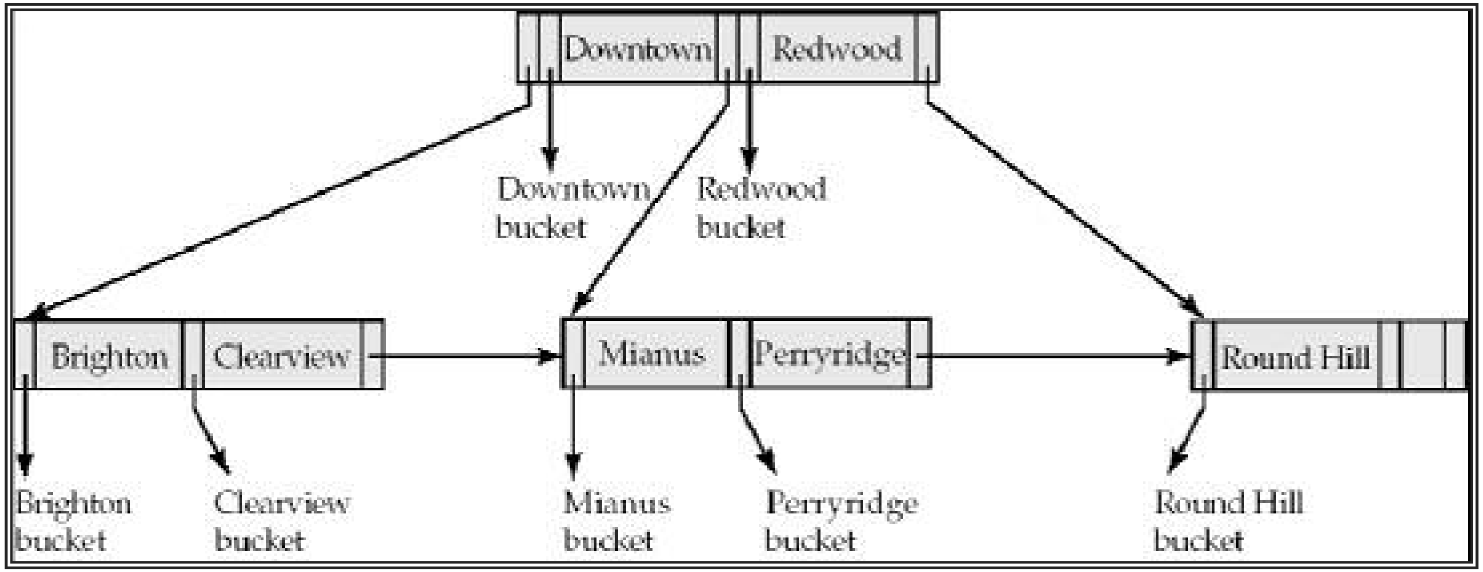
- Similar to B+-tree, but B-tree allows search-key values to appear only once; eliminates redundant storage of search keys.
- Search keys in nonleaf nodes appear nowhere else in the B-tree; an additional pointer field for each search key in a nonleaf node must be included.



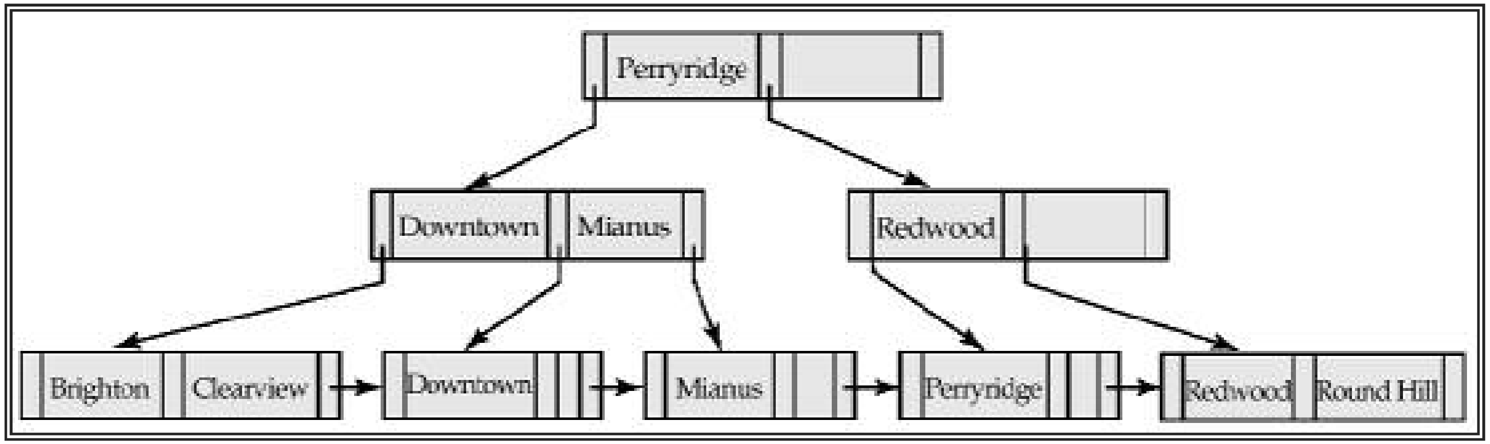
(a) (b)

- Nonleaf node – pointers B_i are the bucket or file record pointers.

B-Tree Index File Example



B-tree (above) and B+-tree (below) on same data



B-Tree Index Files (Cont.)

- Advantages of B-Tree indices:
  - May use less tree nodes than a corresponding B^+-Tree.
  - Sometimes possible to find search-key value before reaching leaf node.
- Disadvantages of B-Tree indices:
  - Only small fraction of all search-key values are found early
  - Non-leaf nodes are larger, so fan-out is reduced. Thus, B-Trees typically have greater depth than corresponding B^+-Tree
  - Insertion and deletion more complicated than in B^+-Trees
  - Implementation is harder than B^+-Trees.
- Typically, advantages of B-Trees do not outweigh disadvantages.

Index Definition in SQL

- Create an index

  create index <index-name> **on** <relation-name> (<attribute-list>)

  E.g.: **create index** *b-index* **on** *branch(branch_name)*
- Use **create unique index** to indirectly specify and enforce the condition that the search key is a candidate key.
  - Not really required if SQL **unique** integrity constraint is supported
- To drop an index

  drop index <index-name>