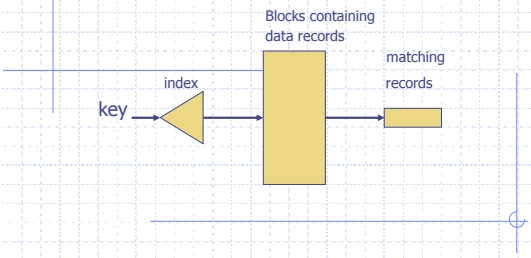
8. Indexes and B-tree

- The needs for indexes in applications like databases.
- Simple indexes, dense indexes, sparse indexes, multi-level indexes.
- B-tree for indexing, its definition, structure, searching, insertion, deletion, and efficiency.

An index



Indexes on sequential files

- In applications of very large amounts of data (e.g. databases, data warehouses, data mining, big data), data records (e.g. for bank transactions, documents, books, products, etc.) are stored in files in secondary storage (mainly hard disks).
- The smallest unit of disk IO is a block. Logically a disk file is a sequence of data blocks.
- A file sorted on one or more attributes (e.g. account number, student ID) is called a sequential file.

Indexes on sequential files

- Compared with main memory (RAM), disk access is much slower.
- Currently, the typical hard disk access time is 5 – 10 milliseconds, while the typical RAM access time is 50-100 nanoseconds. (An SSD has access time of 35 – 100 microseconds.)
  - 1 second = 1,000 milliseconds
  - 1 second = 1,000,000 microseconds
  - 1 second = 1,000,000,000 nanoseconds
- The access time of a hard disk is about 100,000 times longer that that of main memory.

Indexes on sequential files

- Disk access speeds have been the major problem in database (including data warehouse etc) performance.
- A strategy for improving database performance is the use of indexes.
- An index is a collection of [key, pointer] pairs. A pair is an entry in the index. The entries can be organized on the key, e.g. in a sequence sorted on the key for binary search.
- Entries in an index can be organized as trees for even better performance.

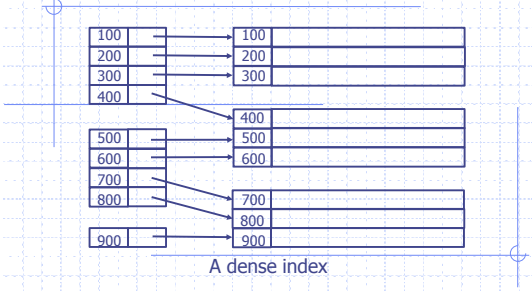
Indexes on sequential files

- An index is much smaller than the data file. In many applications, an index can be kept in Ram. Searching an index entry does not require disk reads.
- An index can be kept in secondary storage. When stored in secondary storage, an index file has much less blocks. Searching an index entry requires a small number of disk reads.
- Once a key-pointer entry is found, a block in the data file can be obtained with a single disk read.

Simple indexes

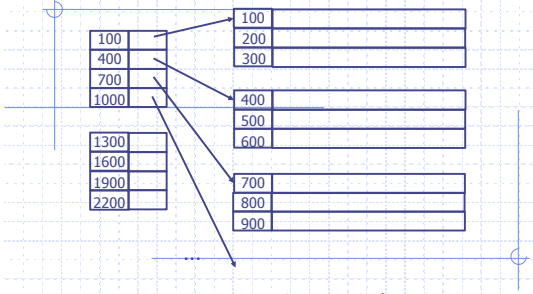
- Dense indexes: In a dense index, there is an entry for every record (or every key) in the data file.
- Sparse indexes: In a sparse index, there are entries for some data records. For example, an entry can be created for the first record in each block.
- When the data file is modified (by insertions, deletions, and updates), the index should also be modified as well.

Simple indexes



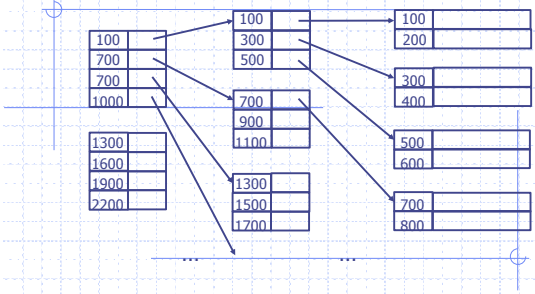
A dense index

Simple indexes



A sparse index

Simple indexes



A two-level index

Simple indexes

- When an index is too big and searching an entry becomes expensive, a higher level index can be created, which is an index of index.
- For a very large number of data records, a multi-level index may be used.
- Repeated insertions and deletions may degrade the index.
- Balanced multi-level index structures are required.
- B-tree is the most commonly used data structure for indexing very large files. A B-tree is a balanced multi-way tree.

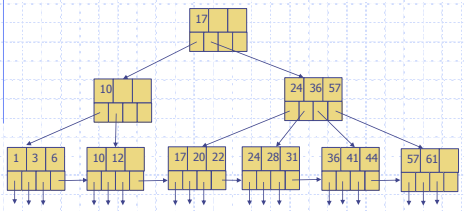
B-tree index structure

Definition:

A B-tree of parameter m is a balanced $(m+1)$-way tree, in which

1. All leaves are on the same level.
2. At a **leaf node**, there are at most m keys and at least $\lceil (m+1)/2 \rceil$ keys.
3. At an **internal node**, there are at most $(m+1)$ pointers (to $(m+1)$ subtrees), and at least $\lceil (m+1)/2 \rceil$ pointers. The number of keys is one less than the number of pointers. The ith key $(1 \le i \le m)$ is the smallest key in the $(i+1)$th subtree.
4. The **root** has at most $(m+1)$ pointers, but may have as few as 2 if it is not a leaf, or none if the tree consists of the root alone.

A B-tree

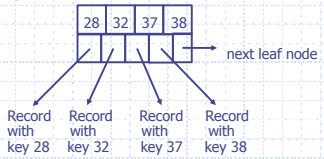


$m = 3$

A leaf node

At a leaf node, when there are q keys $k_1, k_2, \ldots k_q$, the first q pointers are to the q data records with keys $k_1, k_2, \ldots k_q$ respectively. The last pointer is to the next leaf node of the tree.

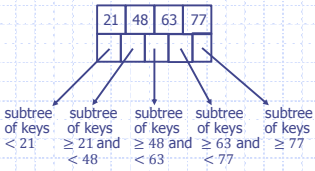
Example (m=4):



An internal node

At an internal node, when there are q keys $k_1, k_2, \ldots k_q$, the first pointer is the subtree of keys less than k_1, the last pointer to the subtree of keys greater than or equal to k_q, and the ith pointer to the subtree of keys greater than or equal to k_{i-1} and less than k_i.

Example (m=4):



Features of a B-tree

- Some keys are duplicated. The same key may appear in both a leaf and an internal nodes.
- At the leaves, each of the keys appears once in order. All the leaves can be seen as a dense index.
- All the internal nodes on a level can be seen as a sparse index.

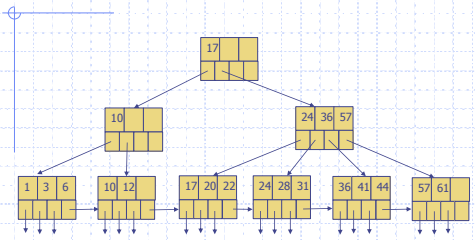
B-Tree as an index

- After inserting a new data record into the data file, insert the key-pointer pair (entry) for the record into the B-tree.
- After deleting a data record from the data file, delete the key-pointer pair from the B-tree when necessary.

Searching key K in a B-tree

- Searching a B-tree starts at the root.
- At a leaf node, if the ith key is equal to K, it is found; If none of the keys is equal to K, it cannot be found in the tree.
- At an internal node, if the ith key is equal to K, search the (i+1)th subtree; if none of the keys in the current node is equal to K,
  - if K is less than the first key in the node, search the first subtree;
  - if K is greater than or equal to the last key, search the last subtree;
  - if K is greater than or equal to the ith and less than the (i+1)th keys, search the (i+1)th subtree.
- With a B-tree index, the search efficiency is O(log n) where n is the number of keys in the index.

A B-tree

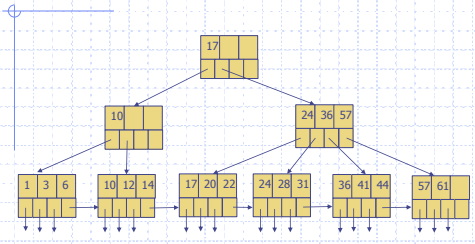


$m = 3$ Search 41, 24

Insertion into a B-tree

- Go down to the appropriate leaf node N using the searching method. We may be in two cases at the leaf before the insertion: The node is full or not full.
- If N is not full before the insertion, insert the new key-pointer pair at a position such that the keys in the node are still in order after the insertion. Adjust the parent node if necessary.

A B-tree



$m = 3$ After inserting 14

Insertion into a B-tree

- If the leaf N is full before the insertion, split it into two new leaf nodes N and M, with M to N's right:
  1. Sort the key-pointer pairs in N and the new one. Partition them into two lists, so that the first $\lceil (m+1)/2 \rceil$ pairs are in the first list and the rest are in the second list.
  2. Store the lower list into N. Store the higher list into M.
  3. Insert the smallest key in M and the pointer to M into the parent of N at the upper level.

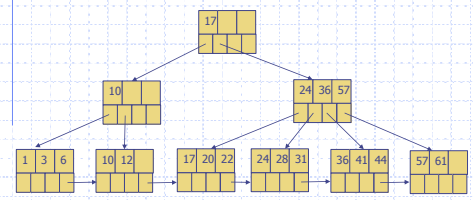
Insertion into a B-tree

- At an internal node, we may also be in two cases before the insertion of a key and a pointer from the lower level: The node is full or not full.
- If the node N is not previously full, insert the key and pointer as a pair at a position such that the keys in the node are still in order. Adjust its parent node if necessary.

Insertion into a B-tree

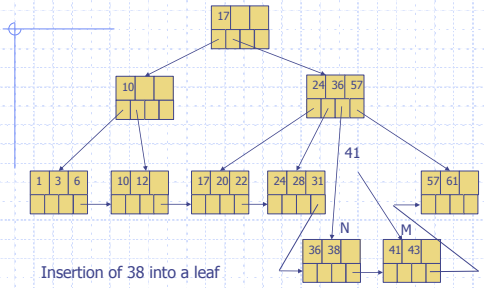
- If the internal node N is previously full, split it into two new nodes N and M, with M to N's right:
  - 1. Sort the key-pointer pairs in N and the new one.
  - 2. Leave the first $\lfloor (m+2)/2 \rfloor$ pointers in N and move the last $\lfloor (m+2)/2 \rfloor$ pointers into M.
  - 3. Leave the first $\lfloor m/2 \rfloor$ keys in N and move the last $\lfloor m/2 \rfloor$ keys into M.
  - 4. Insert the middle key (that is not in N nor in M) and the pointer to M into the upper level.

Insertion into a B-tree



Before the insertion of 38

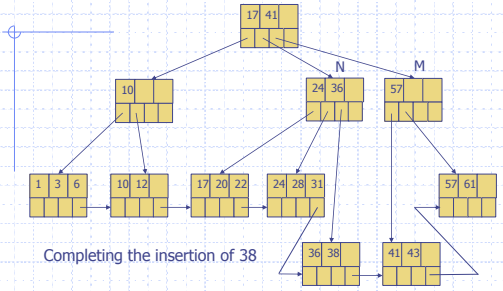
Insertion into a B-tree



Insertion of 38 into a leaf

$\lceil (m+1)/2 \rceil = \lceil (3+1)/2 \rceil = 2$

Insertion into a B-tree



Completing the insertion of 38

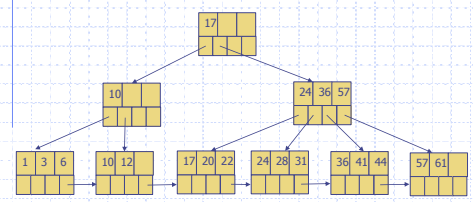
$\lceil (m+2)/2 \rceil = \lceil (3+2)/2 \rceil = 3,\ \lfloor (m+2)/2 \rfloor = \lfloor (3+2)/2 \rfloor = 2$

$\lceil m/2 \rceil = \lceil 3/2 \rceil = 2, \quad \lfloor m/2 \rfloor = \lfloor 3/2 \rfloor = 1$

Deletion from a B-tree:

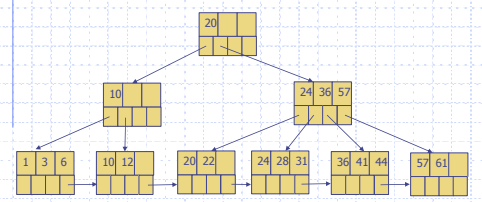
- Locate the key-pointer pair in a leaf node *N*. Delete it from *N*. After the deletion,
  - if the number of keys in *N* is not less than the required minimum, adjust the key in an ancestor that associated with the pointer to this node if necessary, and finish.

Deletion from a B-tree



Before the deletion of 17

Deletion from a B-tree

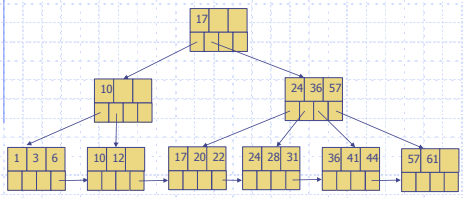


After the deletion of 17

Deletion from a B-tree:

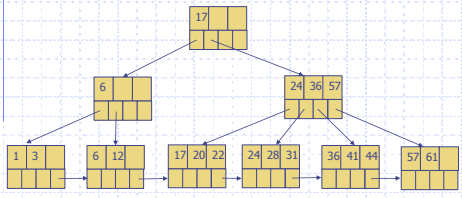
- if the number of keys in leaf node *N* becomes less than the required minimum
  - 1. If the number of keys in an immediate sibling is greater than the required minimum, move the closest key in the sibling into *N*, and adjust the key in an ancestor that associated with the pointer to this node or the sibling. Then finish.

Deletion from a B-tree (cont.)



Before the deletion of 10

Deletion from a B-tree (cont.)



After the deletion of 10

Deletion from a B-tree:

- 2. If none of the immediate adjacent siblings (or sibling) have an extra key, merge *N* with one of the siblings (e.g. *M* to *N*'s right), and then delete a pointer (e.g. the pointer to *M*) from the parent and adjust the keys at the parent node.

Deletion from a B-tree:

At internal node *N*, after the deletion of a pointer,

- if the number of pointers in *N* is not less than the required minimum, make necessary adjustment of the keys in *N*, and finish.

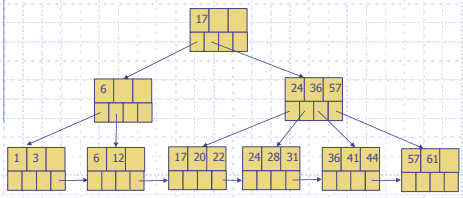
Deletion from a B-tree:

- if the number of pointers in internal node *N* becomes less than the required minimum
  - 1. If the number of pointers in an immediate sibling is greater than the required minimum, move the closest pointer in the sibling into node *N*, and adjust the keys in the current node and the sibling. Then finish.

Deletion from a B-tree:

- 2. If none of the immediate adjacent siblings (or sibling) have an extra pointer, merge one of the siblings (e.g. *M* to *N*'s right), and then delete a pointer (e.g. the pointer to *M*) from the parent and adjust the keys at the parent node.

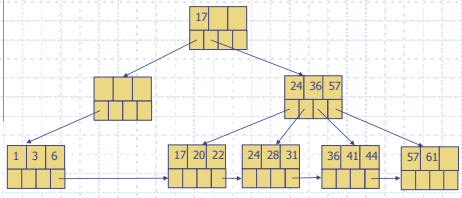
Deletion from a B-tree



m = 3 Before the deletion of 12

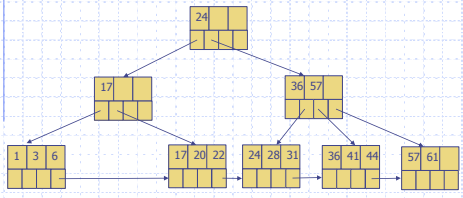
$\lfloor (m+1)/2 \rfloor = \lfloor (3+1)/2 \rfloor = 2$

Deletion from a B-tree



Beginning the deletion of 12

Deletion from a B-tree



Completing the deletion of 12

B-tree Efficiency

- Searching $O(\log n)$
- Insertion $O(\log n)$
- Deletion $O(\log n)$