Table of Contents -- sequential storage order

Index -- based on key terms, find random access to specific items

Also, analogy of card catalog

- Indexing mechanisms used to speed up access to desired data.
  - E.g., author catalog in library
- **Search Key** - attribute to set of attributes used to look up records in a file.
- An **index file** consists of records (called **index entries**) of the form

search-key	pointer

- Index files are typically much smaller than the original file
- Two basic kinds of indices:
  - **Ordered indices:** search keys are stored in sorted order
  - **Hash indices:** search keys are distributed uniformly across "buckets" using a "hash function".

Metrics for Evaluation

- Access types supported efficiently. E.g.,
  - records with a specified value in the attribute
  - or records with an attribute value falling in a specified range of values.
- Access time
- Insertion time
- Deletion time
- Space overhead

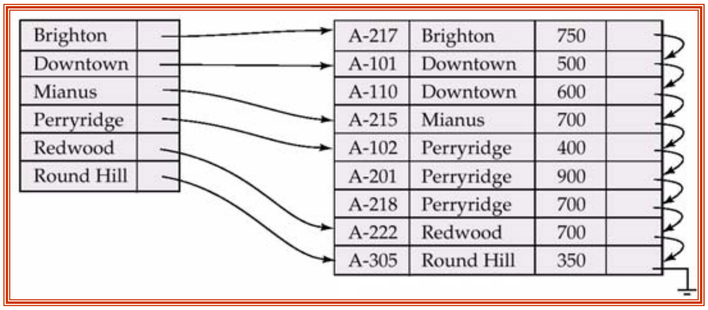
Ordered Indices

Search Key -- may be different from primary key

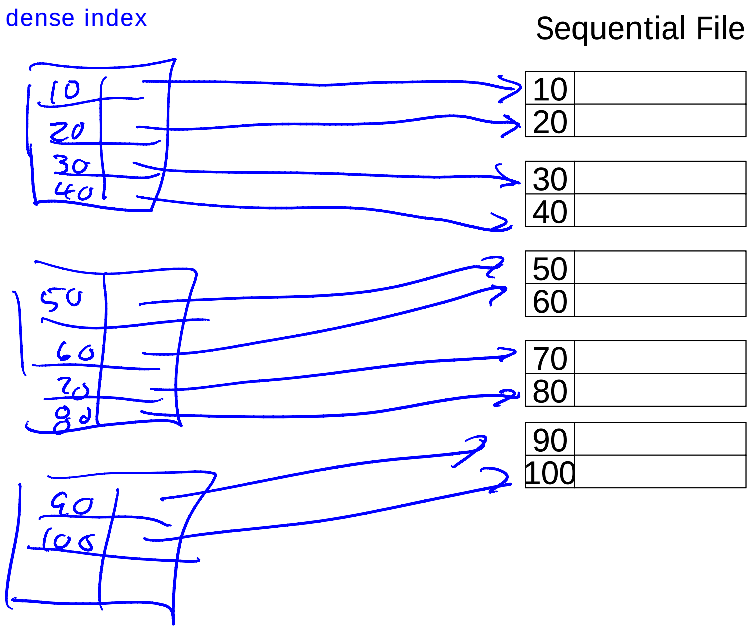
Indexing techniques evaluated on basis of:

- In an **ordered index,** index entries are stored sorted on the search key value. E.g., author catalog in library.
- **Primary index:** in a sequentially ordered file, the index whose search key specifies the sequential order of the file.
  - Also called **clustering index**
  - The search key of a primary index is usually but not necessarily the primary key.
- **Secondary index**: an index whose search key specifies an order different from the sequential order of the file. Also called non-clustering index**.**
- Index-sequential file**:** ordered sequential file with a primary index.

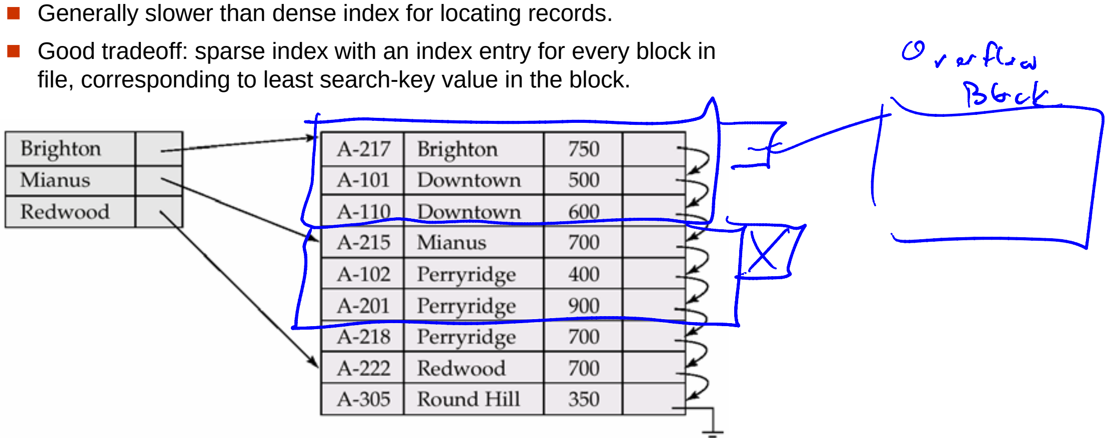
- Dense index — Index record appears for every search-key value in the file.



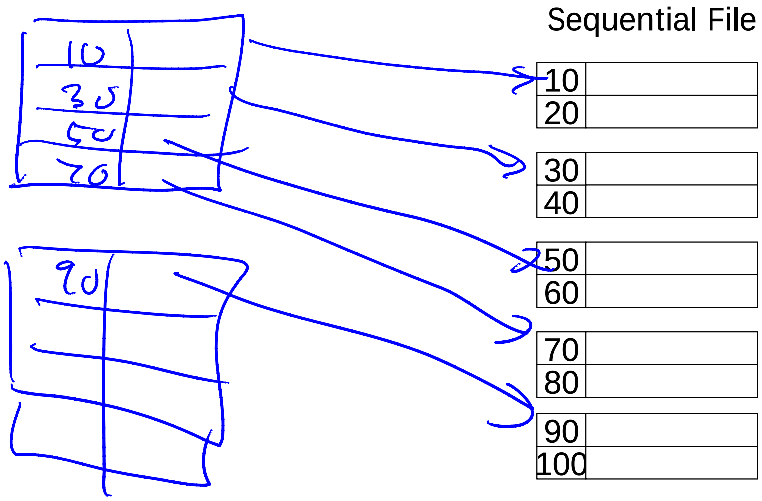
Build a dense index



- Sparse Index: contains index records for only some search-key values.
  - Applicable when records are sequentially ordered on search-key
- To locate a record with search-key value *K* we:
  - Find index record with largest search-key value < *K*
  - Search file sequentially starting at the record to which the index record points
- Less space and less maintenance overhead for insertions and deletions.
- Generally slower than dense index for locating records.
- Good tradeoff: sparse index with an index entry for every block in file, corresponding to least search-key value in the block.

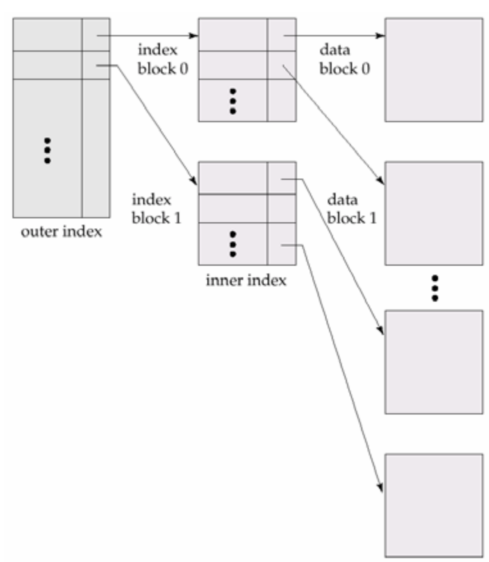


Build a sparse index



Multilevel Index

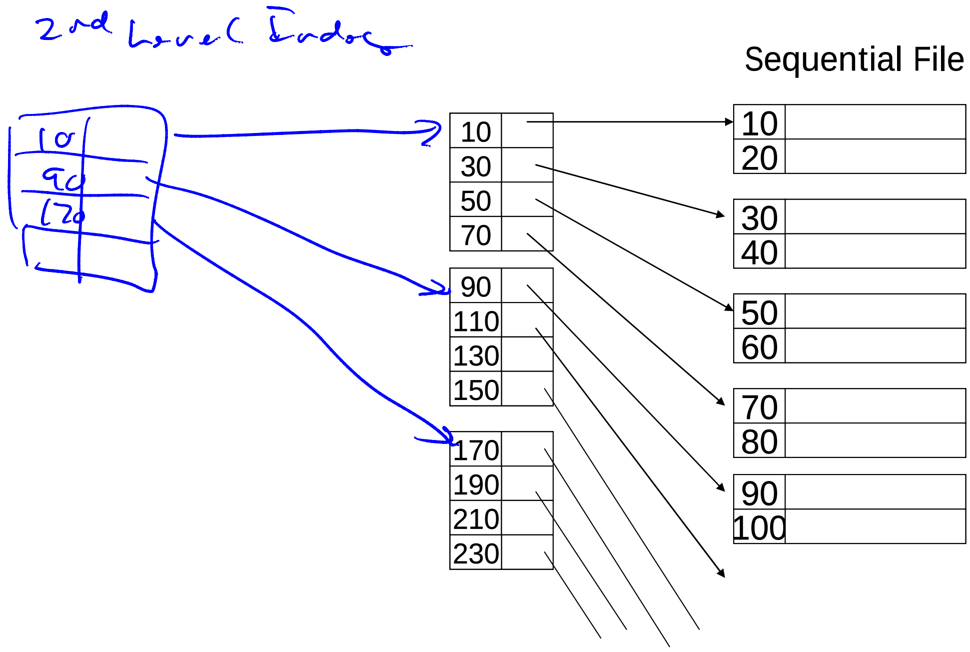
- If primary index does not fit in memory, access becomes expensive.
- To reduce number of disk accesses to index records, treat primary index kept on disk as a sequential file and construct a sparse index on it.
  - outer index – a sparse index of primary index
  - inner index – the primary index file
- If even outer index is too large to fit in main memory, yet another level of index can be created, and so on.
- Indices at all levels must be updated on insertion or deletion from the file.



Build a Second-Level Index

2nd Level Index

Sequential File

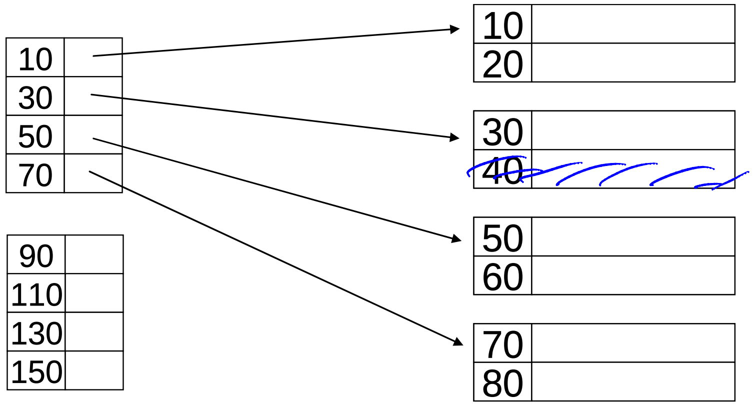


Index Record Deletion

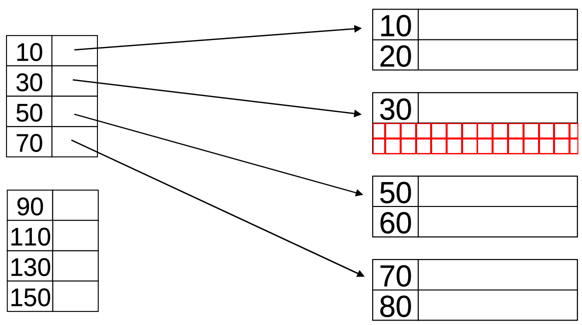
- If deleted record was the only record in the file with its particular search-key value, the search-key is deleted from the index also.
- Single-level index deletion:
  - Dense indices – deletion of search-key is similar to file record deletion.
  - Sparse indices – if an entry for the search key exists in the index, it is deleted by replacing the entry in the index with the next search-key value in the file (in search-key order). If the next search-key value already has an index entry, the entry is deleted instead of being replaced.

Insertion

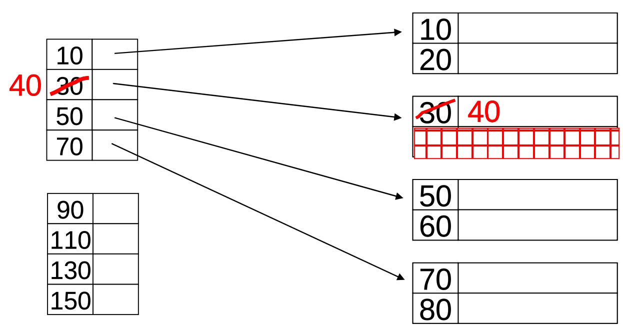
- Single-level index insertion:
  - Perform a lookup using the search-key value appearing in the record to be inserted.
  - Dense indices – if the search-key value does not appear in the index, insert it.
  - Sparse indices – if index stores an entry for each block of the file, no change needs to be made to the index unless a new block is created. In this case, the first search-key value appearing in the new block is inserted into the index.
- Multilevel insertion (as well as deletion) algorithms are simple extensions of the single-level algorithms



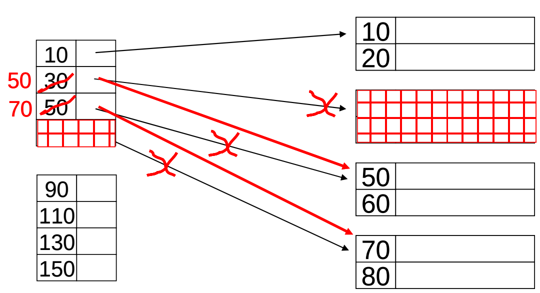
– delete record 40



– delete record 30

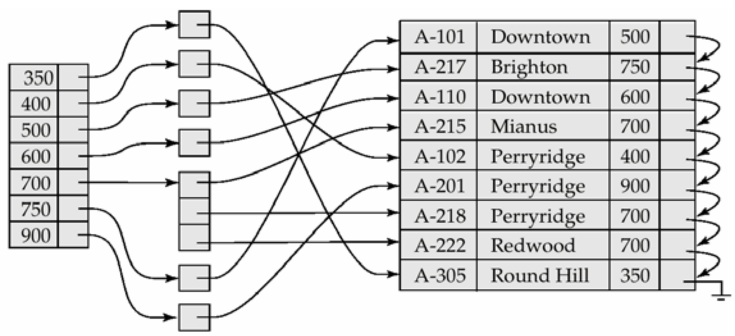


– delete records 30 & 40

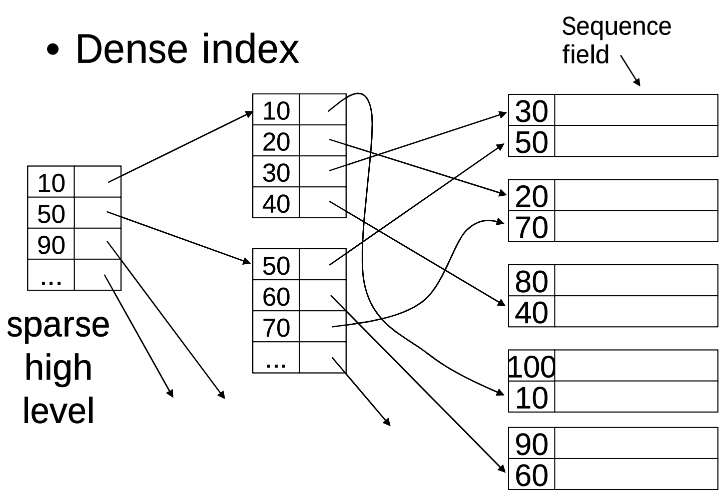


Secondary Indices

- Frequently, one wants to find all the records whose values in a certain field (which is not the search-key of the primary index satisfy some condition.
  - Example 1: In the *account* database stored sequentially by account number, we may want to find all accounts in a particular branch
  - Example 2: as above, but where we want to find all accounts with a specified balance or range of balances
- We can have a secondary index with an index record for each search-key value; index record points to a bucket that contains pointers to all the actual records with that particular search-key value.



- Secondary indices have to be dense.
- Indices offer substantial benefits when searching for records.
- When a file is modified, every index on the file must be updated, Updating indices imposes overhead on database modification.
- Sequential scan using primary index is efficient, but a sequential scan using a secondary index is expensive
  - each record access may fetch a new block from disk



Conventional indexes

Advantage:

- Simple
- Index is sequential file
  good for scans

Disadvantage:

- Inserts expensive, and/or
- Lose sequentiality & balance

Solution: Use a more general data structure that allows us to enforce balance and has good insertion/deletion characteristics

-- The B+ tree

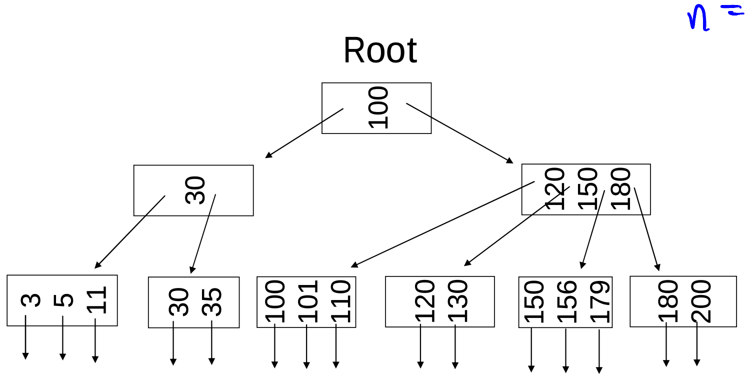
n

- Automatically maintain as many levels of index as is appropriate for the size of the file being indexed
- Block space management ensures that every block is between half and completely full. No overflow blocks are needed.

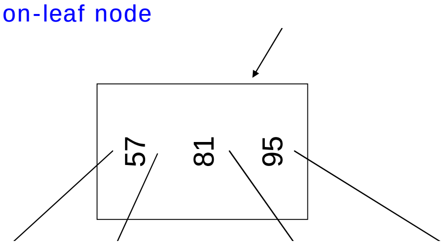
B-tree properties

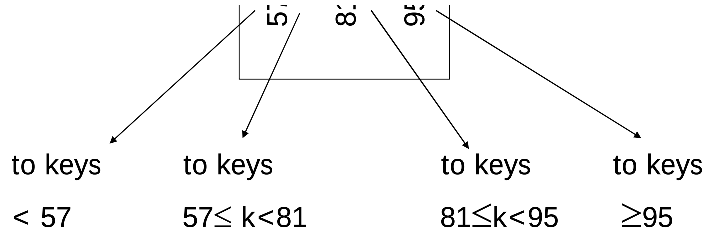
- All paths from root to leaf have same length
- Every block has space for n search keys and n+1 pointers
- The keys in leaf nodes are copies of keys from the data file. These keys are distributed among the leaves in sorted order, from left to right.
- At the root, there are at least two used pointers. All pointers point to B-tree blocks at the level below.
- At a leaf, the last pointer points to the next leaf block to the right. At least floor((n+1)/2)) point to data blocks.
- At interior nodes, all n+1 pointers can be used to point to B-tree blocks

n = 3

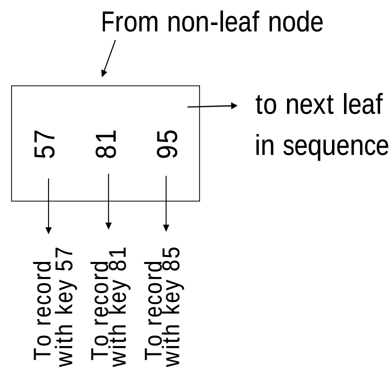


Sample non-leaf node





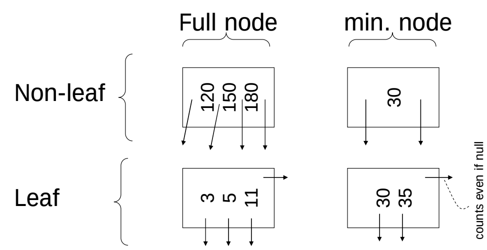
Sample leaf node



- Use at least

Non-leaf: $\lceil (n+1)/2 \rceil$ pointers

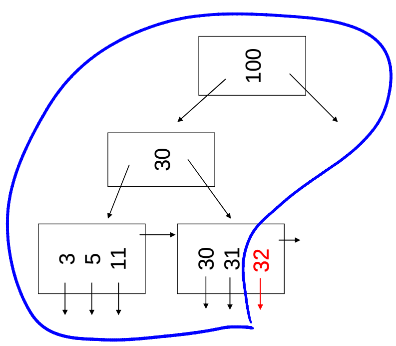
Leaf: $\lfloor (n+1)/2 \rfloor$ pointers to data



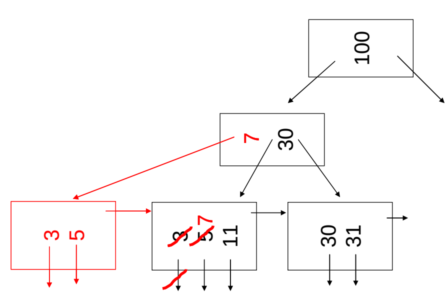
Insert into B+tree

(a) simple case
– space available in leaf
(b) leaf overflow
(c) non-leaf overflow
(d) new root

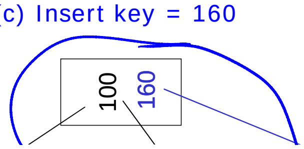
(a) Insert key = 32

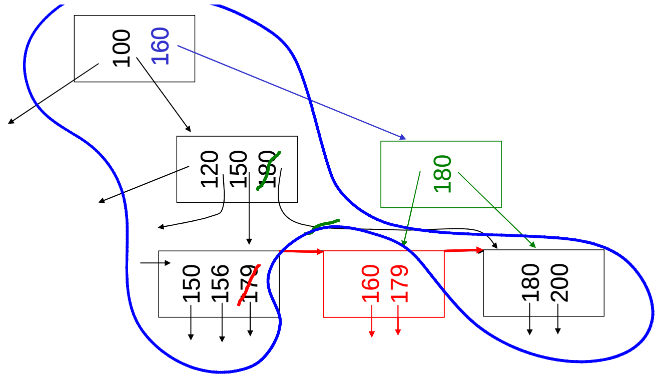


(b) Insert key = 7



(c) Insert key = 160





(d) New root, insert 45