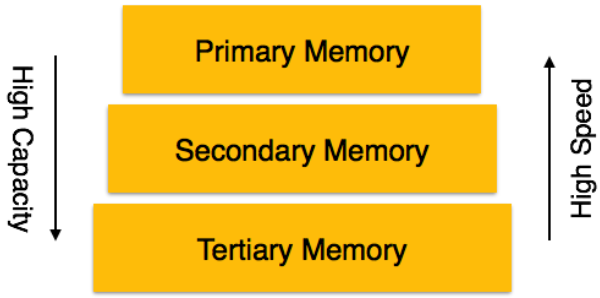
Unit 4

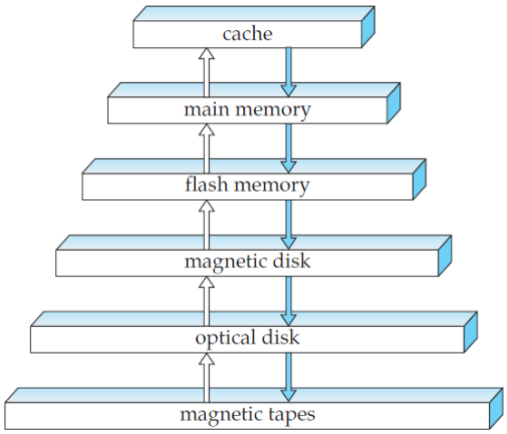
DBMS - Storage System

Databases are stored in file formats, which contain records. At physical level, the actual data is stored in electromagnetic format on some device. These storage devices can be broadly categorized into three types −



- **Primary Storage** − The memory storage that is directly accessible to the CPU comes under this category. Eg. registers, fast memory (cache), and main memory (RAM). This storage is typically very small, ultra-fast, and volatile.
- **Secondary Storage** − Secondary storage includes memory devices that are not a part of the CPU chipset or motherboard, for example, magnetic disks, optical disks (DVD, CD, etc.), hard disks, flash drives, and magnetic tapes.
- **Tertiary Storage** − Tertiary storage is used to store huge volumes of data. These storage devices are mostly used to take the back up of an entire system. Optical disks and magnetic tapes are widely used as tertiary storage.

Memory Hierarchy:



Redundant Arrays of Independent Disks (RAID)

RAID, or "Redundant Arrays of Independent Disks" is a technique which makes use of a combination of multiple disks instead of using a single disk for increased performance, data redundancy or both.

Need for RAID:

- Improvement of Reliability via Redundancy
  The simplest approach to introducing redundancy is to duplicate every disk. This technique is called mirroring.

- Improvement in Performance via Parallelism
  With multiple disks, the transfer rate can be improved by striping data across multiple disks. In its simplest form, data striping consists of splitting the bits of each byte across multiple disks; such striping is called bit level striping

RAID LEVELS:

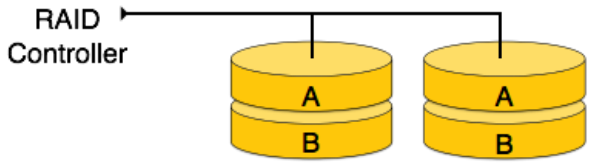
RAID 0 – Non Redundant Stripping
RAID 1 – Mirrored disks
RAID 2 - Memory style error correcting code
RAID 3 – Bit interleaved Parity
RAID 4 – Block interleaved Parity
RAID 5 - Block interleaved distributed Parity
RAID 6 – P + Q Redundancy

RAID level 0:

- The data is broken down into blocks and all blocks are distributed among all disks.
- Each disk receives a block of data to write/read in parallel. This enhances the speed and performance of storage device.
- There is no parity and backup in Level 0.

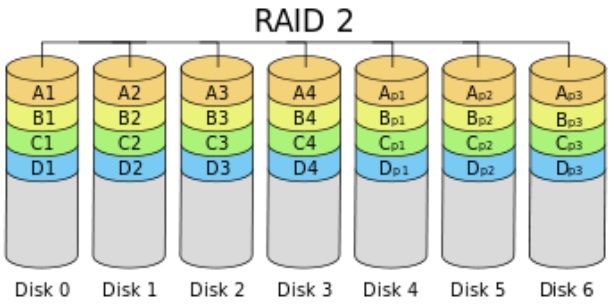
RAID LEVEL 1:

- This level uses mirroring techniques. When data is sent to RAID controller it sends a copy of data to all disks in array.
- RAID level 1 is also called mirroring and provides 100% redundancy in case of failure



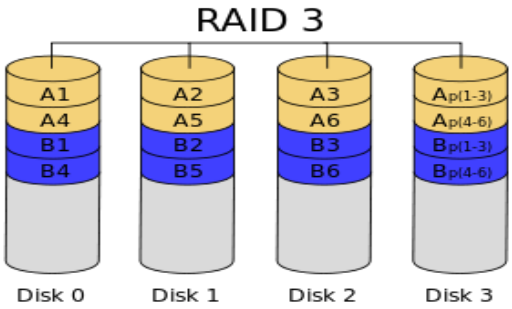
RAID LEVEL 2:

- RAID 2 consists of bit-level striping using hamming code parity. In this level, each data bit in a word is recorded on a separate disk and ECC code of data words is stored on different set disks.
- Due to its high cost and complex structure, this level is not commercially used.



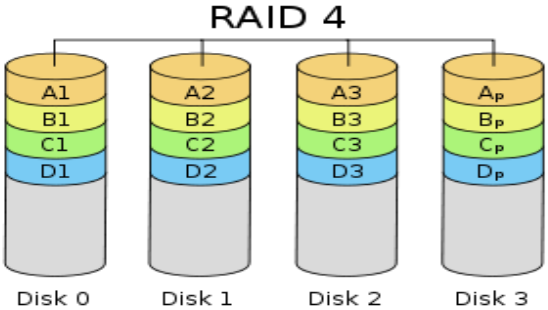
RAID LEVEL 3:

- RAID 3 consists of byte-level striping with dedicated parity. In this level, the parity information is stored for each disk section and written to a dedicated parity drive.



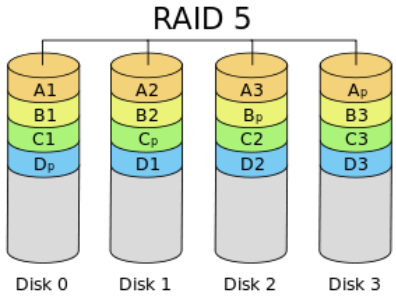
RAID LEVEL 4:

RAID 4 consists of block-level stripping with a parity disk. Instead of duplicating data, the RAID 4 adopts a parity-based approach.



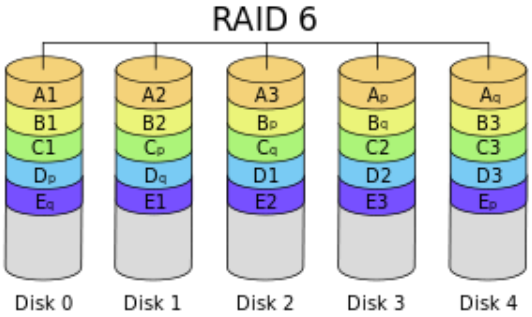
RAID LEVEL 5:

- RAID 5 is a slight modification of the RAID 4 system. The only difference is that in RAID 5, the parity rotates among the drives.
- It consists of block-level striping with DISTRIBUTED parity



RAID LEVEL 6:

- This level is an extension of RAID 5. It contains block-level stripping with 2 parity bits.
- In RAID 6, the system can survive 2 concurrent disk failures.



File organization:

File organization is a Method of arranging a file of records on external storage.

FILE: A collection of pages, each containing a collection of records.

Record Formats:

Fixed-length records - all the records are exactly the same length
Variable-length records - the length of each record varies

Advantages of variable length Record:

- Variable-length records save disk space. Because, to use fixed-length records, you the record length of all the records is chosen based on the length of the longest record. If

the application generates many short records with occasional long ones, using fixed-length records wastes a lot of disk space, so variable-length records would be a better choice.

In DBMS,

- File corresponds to Table
- Record corresponds to Row
- Field corresponds to attribute

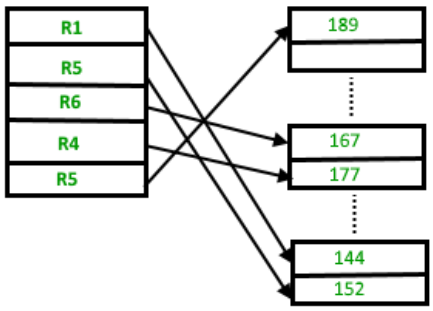
Types of File organization :

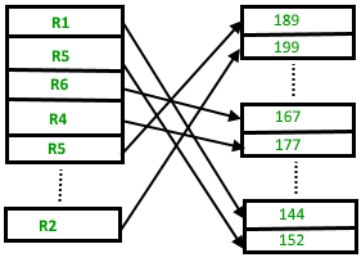
- Heap File organization
- Sequential Order file organization
- B+ tree file organization
- Hash file organization

1) Heap File organization

- It is the simplest and most basic type of organization. In heap file organization, the records are inserted at the file's end. When the records are inserted, it doesn't require the sorting and ordering of records.

To insert a new record: **R2**





Pros –

- When there is a huge number of data needs to be loaded into the database at a time, then this method of file Organization is best suited.

Cons –

- Inefficient for larger databases.

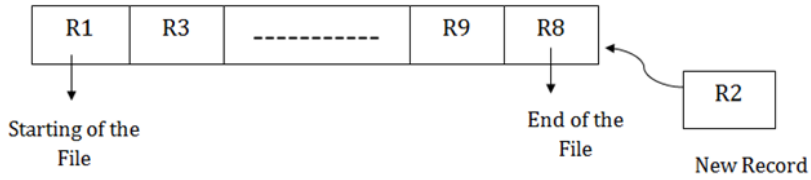
2) Sequential order file organization:

In this method, files are stored sequentially. This method can be implemented in two ways:

1. Pile File Method
2. Sorted File Method

Pile File Method:

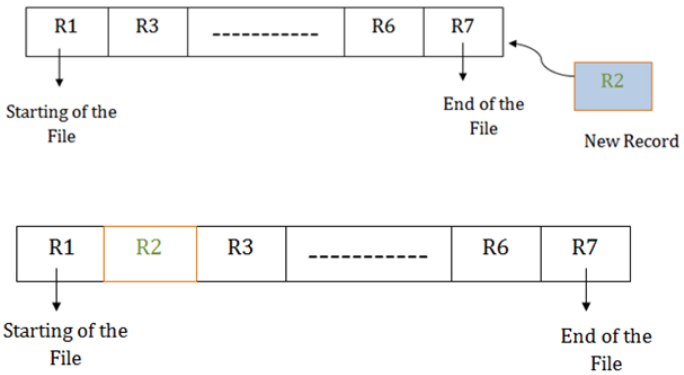
In this method, the record will be inserted in the order in which they are inserted into tables.



In case of updating or deleting of any record, the record will be searched in the memory blocks. When it is found, then it will be marked for deleting.

Sorted File Method:

In this method, the new record is always inserted at the file's end, and then it will sort the sequence in ascending or descending order. Sorting of records is based on any primary key or any other key.



Pros of sequential file organization:

- In this method, files can be easily stored in cheaper storage mechanism like magnetic tapes.

- It is simple in design.
- It requires no much effort to store the data.

Cons of sequential file organization:

- It will waste time as we cannot jump on a particular record that is required but we have to move sequentially which takes our time.
- Sorted file method takes more time and space for sorting the records.

3) Hash File Organization:

Hash File Organization uses the computation of hash function on some fields of the records. The hash function's output determines the location of disk block where the records are to be placed.

4) B+ tree file organization

B+ tree file organization is the advanced method of an indexed sequential access method. It uses a tree-like structure to store records in File.

File Operations :

Operations on database files can be broadly classified into two categories −

- Update Operations (Insert, delete, update)
- Retrieval Operations (select)

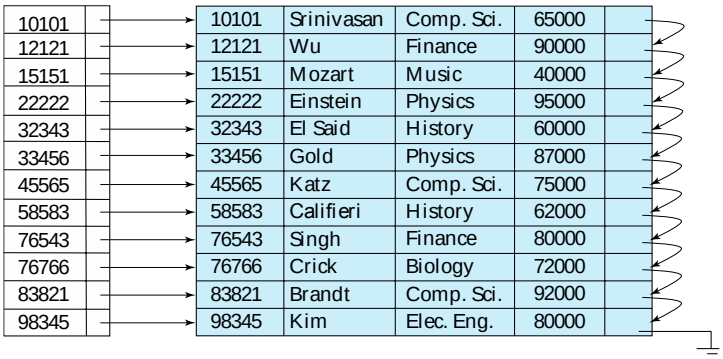
In addition to that

- Open
- Locate
- Read
- Write
- close

Indexing

- Indexing is used to speed up the retrieval of records or access to desired data in response to certain search conditions.
- In indexing, the records in the indexing file are in the sorted order, making it easy to find the record looking for.
- An **index file** consists of records (called **index entries**) of the form

Search Key - attribute to set of attributes used to look up records in a file.
Pointers - Index files are typically much smaller than the original file



Index Evaluation Metrics:

Access types: The types of access that are supported efficiently.
Access time: The time it takes to find a particular data item, or set of items.
Insertion time: The time it takes to insert a new data item.
Deletion time: The time it takes to delete a data item.
Space overhead: The additional space occupied by an index structure.

SQL facilities for Index:

Creation of Index using SQL.
CREATE INDEX *index_name*
ON *table_name* (*column1*, *column2*, ...);

Example:
CREATE INDEX idx_lastname
ON Persons (LastName);

Index on multiple columns:
CREATE INDEX idx_pname
ON Persons (LastName, FirstName);

Types of indices:

- **Ordered indices:** Based on a sorted ordering of the values.
- **Hash indices:** Based on a uniform distribution of values across a range of buckets. The bucket to which a value is assigned is determined by a function, called a hash function

Ordered indices:

- In an **ordered index,** index entries are stored sorted on the search key value.

Types of Ordered Index:

- **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**.**

Primary index:

It is a sequentially ordered file, the index whose search key specifies the sequential order of the file.
Example:
Consider the following when no indexing used:

- An ordered file with r= 30,000 records, block size B= 1024 bytes.
- File records are of fixed size and are un spanned, with record length R=100 bytes.
- Blocking factor (bfr) = B/R = 1024/ 100 = 10 records per block;
- The number of blocks needed for the file is b= r / bfr = 30,000 /10 =3000 blocks.
- For an binary search $\log_2 b = \log 2\ 3000 = 12$ block access.

For primary indexes:

- The ordering key field v=9 bytes long , block pointer p=6 bytes long
- The size if the index entry = Ri = (9+6) = 15 bytes.
- so, bfri = 1024 / 15 = 68 entries per block.
- The total no. of index entries ri is equal to the number of blocks in the data file, which is 3000.
- No. of index blocks bi = ri / bfri (3000 / 68) =45 blocks.
- For an binary search $\log_2 45 = 6$ block access

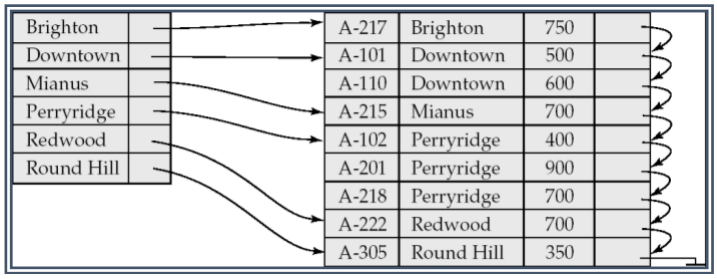
Types:

- Dense Index
- Sparse Index

Dense Index:

- In a dense index, an index entry appears for every search-key value in the file.

- In dense primary index, the index record contains the search key value and a pointer to the first data record with that search key value.
- The rest of the records with the same search key value would be sorted sequentially after the first record.



Sparse index:

In sparse index, an ordered record appears for only some of the search key values.

- 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

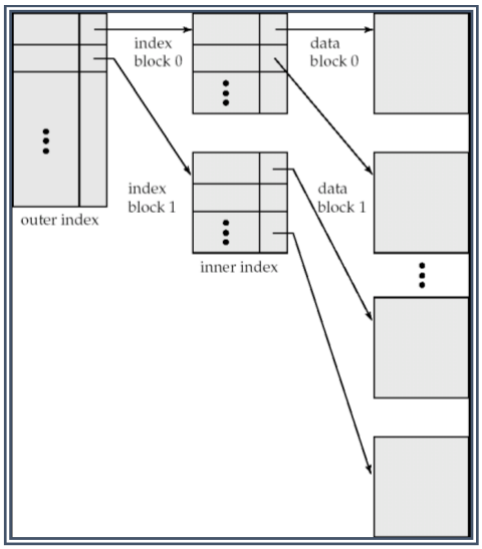
Comparison of Dense Index and Sparse Index:

S.No	Dense Index	Sparse Index
1	An index record appears for every search key value in file	Index records are created only for some of the records.
2	This record contains search key value and a pointer to the actual record. Sparse Index	To locate a record, we find the index record with the largest search key value less than or equal to the search key value we are looking for.
3	Faster to locate a record	Less Faster to locate a record than dense.
4.	requires more space and more maintenance over head	requires less space and less maintenance over head.

Multilevel Index:

If primary index does not fit in memory, access becomes expensive.

- Solution: 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



Record Deletion in Primary Index (Single level index):

- **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.**

Deletion in **Dense indices** – deletion of search-key: similar to file record deletion.

Deletion in **Sparse indices** :

- If deleted key value exists in the index, the value is replaced by 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.

Record Insertion:

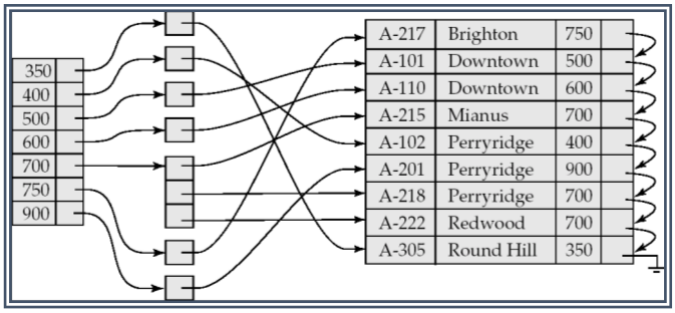
- Perform a lookup using the key value from inserted record

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.

If a new block is created, the first search-key value appearing in the new block is inserted into the index.

Secondary Indices:

- 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



Eg,

- r=30000 fixed length records.
- Size R=100 bytes
- Block size B=1024 bytes.
- File has b =3000 blocks.
- Secondary index key field v=9 bytes long.
- Block pointer= 6 bytes long
- Bfri = 1024 / 15 = 68 entries per block

- Total no. of index entries ri equal to the number of records in the data file = 30,000

- **Number blocks needed for index = 30,000/ 68 = 442 blocks.**
  for an binary search log 2 442 = 9 block access

Disadvantage of indexed-sequential files:

1) Performance degrades as file grows, since many overflow blocks get created.

2) Periodic reorganization of entire file is required.

Hashing

Hashing is an effective technique to calculate the direct location of a data record on the disk without using index structure. Hashing uses hash functions with search keys as parameters to generate the address of a data record.

Hash File Organization

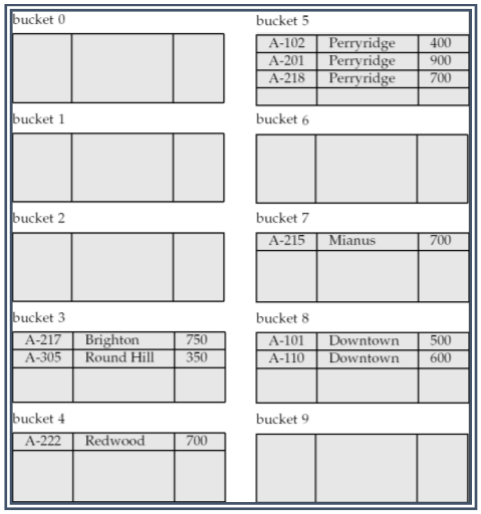
Bucket:

A hash file stores data in bucket format. Bucket is considered a unit of storage. A bucket typically stores one complete disk block, which in turn can store one or more records.

Hash Function:

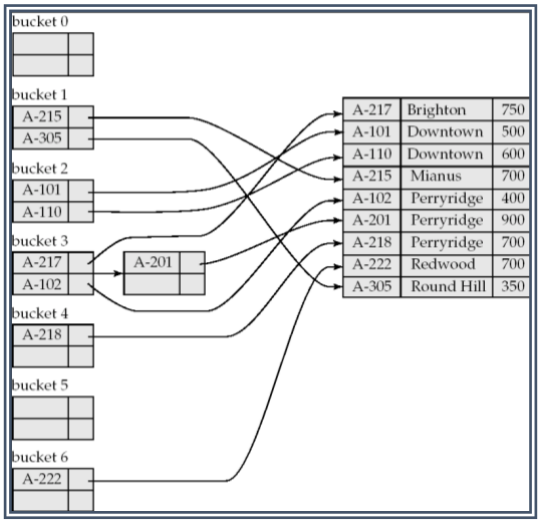
A hash function, h, is a mapping function that maps all the set of search-keys K to the address where actual records are placed. It is a function from search keys to bucket addresses.

- Worst hash function maps all search-key values to the same bucket; this makes access time proportional to the number of search-key values in the file.
- An ideal hash function is **uniform**, i.e., each bucket is assigned the same number of search-key values from the set of *all* possible values.
- Ideal hash function is **random**, so each bucket will have the same number of records assigned to it irrespective of the *actual distribution* of search-key values in the file.



Hash Indices:

- Hashing can be used not only for file organization, but also for index-structure creation.
- A **hash index** organizes the search keys, with their associated record pointers, into a hash file structure.



- Hash indices are always secondary indices

Types of Hashing:

- Static Hashing

- Dynamic Hashing

Static Hashing:

- In static hashing, when a search-key value is provided, the hash function always computes the same address. That means if we generate an address for EMP_ID =103 using the hash function mod (5) then it will always result in same bucket address 3. Here, there will be no change in the bucket address.

- The number of buckets provided remains unchanged at all times.

Operations on static Hashing:

Insertion − When a record is required to be entered using static hash, the hash function h computes the bucket address for search key K, where the record will be stored.
Bucket address = h(K)

Search − When a record needs to be retrieved, the same hash function can be used to retrieve the address of the bucket where the data is stored.

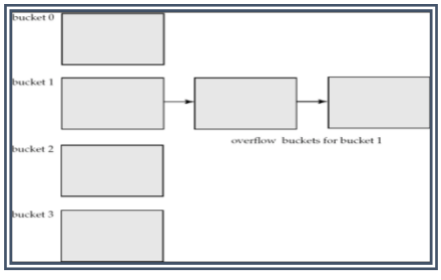
Delete − This is simply a search followed by a deletion operation.

Deficiencies of Static Hashing:

- Bucket overflow can occur because of
  - Insufficient buckets
  - Skew in distribution of records. This can occur due to two reasons:
    - multiple records have same search-key value
    - chosen hash function produces non-uniform distribution of key values

Handling of Bucket Overflows:

- **Overflow chaining** – the overflow buckets of a given bucket are chained together in a linked list. Above scheme is called closed hashing**.**



- An alternative, called open hashing, which does not use overflow buckets, is not suitable for database applications.

- Databases grow or shrink with time. In static Hashing,
  - If initial number of buckets is too small, and file grows, performance will degrade due to too much overflows.

- If space is allocated for anticipated growth, a significant amount of space will be wasted initially (and buckets will be underfull).
- If database shrinks, again space will be wasted.

Best solution: **allow the number of buckets to be modified dynamically.**

Dynamic Hashing

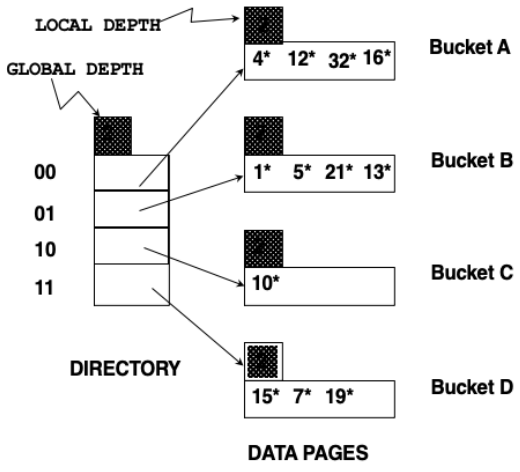
Dynamic hashing provides a mechanism in which data buckets are added and removed dynamically and on-demand. Dynamic hashing is also known as **Extended hashing. It is** Good for database that grows and shrinks in size

Extendable hashing.

The prefix of an entire hash value is taken as a hash index. Only a portion of the hash value is used for computing bucket addresses. Every hash index has a depth value (Global depth and local depth) to signify how many bits are used for computing a hash function. These bits can address 2^n buckets. When all these bits are consumed − that is, when all the buckets are full − then the depth value is increased linearly and twice the buckets are allocated.

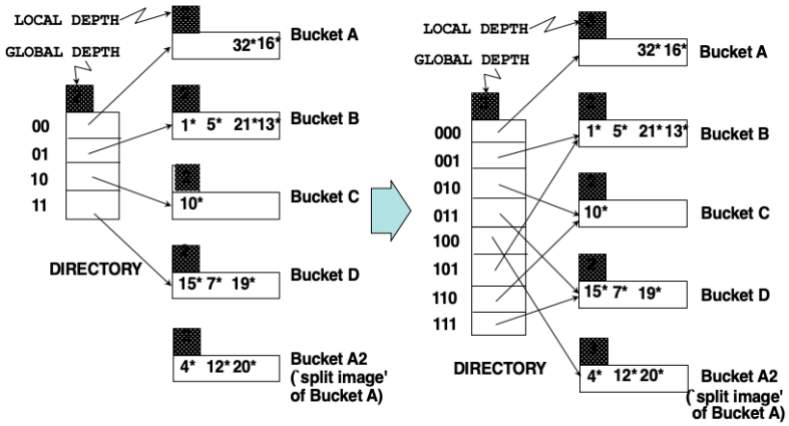
Example: Directory is array of size 4.

- To find bucket for *r*, take last `*global depth*' # bits of **h**(*r*); we denote *r* by **h**(*r*).
  - If **h**(*r*) = 5 = binary 101, it is in bucket pointed to by 01.



To Insert 20,

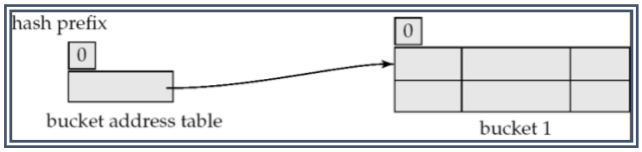
The global depth has increased from 22 to 23, representing 8 states. Now the size of the overflow buckets has increased twice.

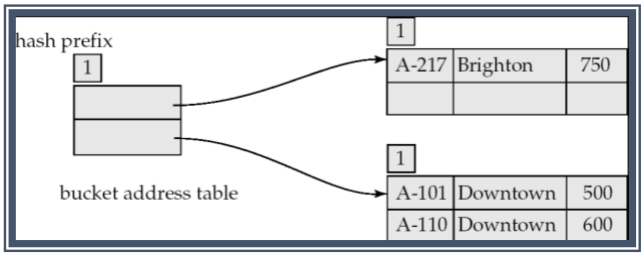


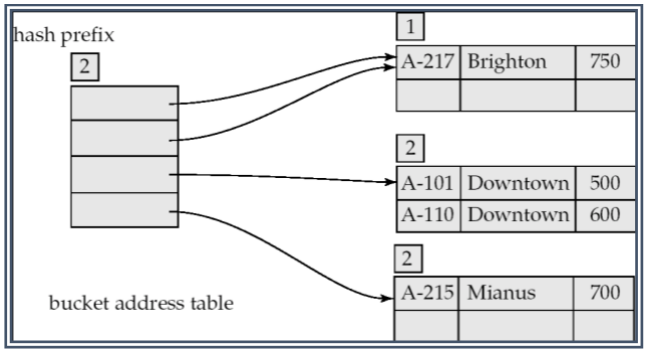
Example of Dynamic Hashing:

To insert the following:

Brighton
Downtown
Downtown
Mianus
Perryridge
Perryridge
Perryridge
Round Hill







Benefits of extendable hashing:

- Hash performance does not degrade with growth of file
- Minimal space overhead

Disadvantages of extendable hashing:

- Bucket address table may itself become very big (larger than memory)
- Cannot allocate very large contiguous areas on disk either

  Solution: B+-tree file organization to store bucket address table

---

Comparison of Ordered Indexing and Hashing:

- Hashing is generally better at retrieving records having a specified value of the key.
- If range queries are common, ordered indices are to be preferred.

B^+ Tree indexing

A tree is formed of nodes. Each node in the tree, except for a special node called the root, has one parent node and zero or more child nodes. The root node has no parent. A node that does not have any child nodes is called a leaf node; a non-leaf node is called an internal node.

The B+ -tree index structure is the most widely used of several index structures that maintain their efficiency despite insertion and deletion of data. A B^+-tree index takes the form of a balanced tree in which every path from the root of the tree to a leaf of the tree is of the same length. Each non-leaf node in the tree has between ⌈n/2⌉ and n children, where n is fixed for a particular tree.

Structure of a B^+-Tree:

A B+-tree index is a multilevel index

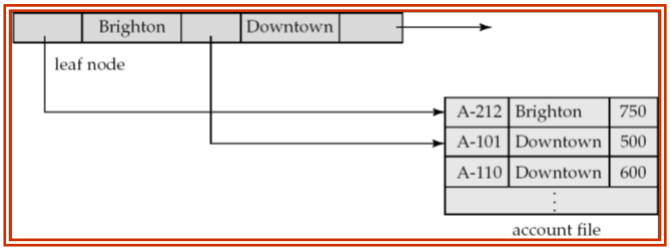
P_1	K_1	P_2	…	P_{n-1}	K_{n-1}	P_n

- Ki are the search-key values
- Pi are pointers to children (for non-leaf nodes) or pointers to records or buckets of records (for leaf nodes).

It contains up to n − 1 search-key values K1, K2, . . . , Kn − 1, and n pointers P1, P2,...,Pn.The search-key values with in a node are kept in sorted order

Structure of Leaf Node:

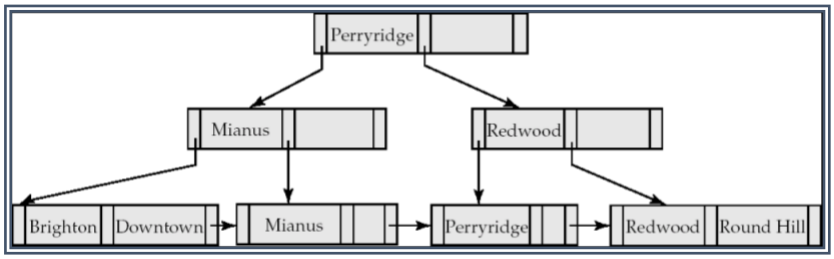
In leaf node, the pointer Pi points to a file record with search-key value Ki .



Non-leaf Nodes:

- The non-leaf nodes of the B^+-tree form a multilevel (sparse) index on the leaf nodes. The pointers of non-leaf nodes pointing only to tree nodes not file record. A non-leaf node may hold up to n pointers, and must hold at least ⌈n/2⌉ pointers.
- The number of pointers in a node is called the fanout of the node.
- Non-leaf nodes are also referred to as internal nodes. The non-leaf node contains the redundant values that represent how to reach the leaf node.

Example of B^+ Tree

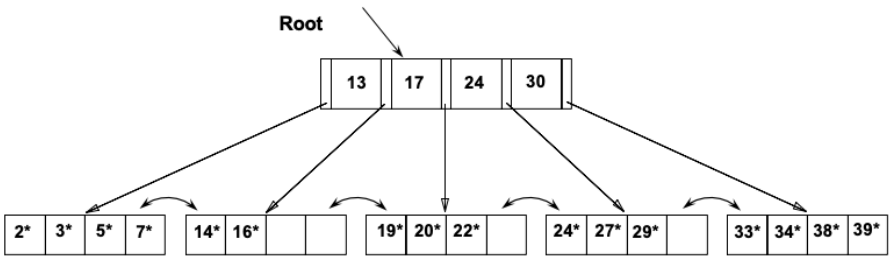


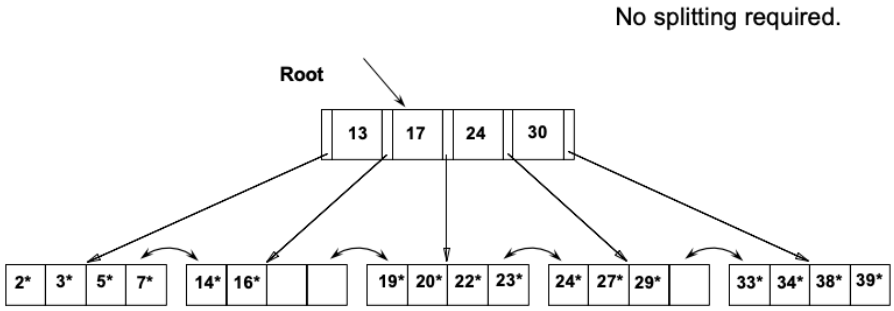
Updates on B+-Trees:

- When a record is inserted into, or deleted from a relation, indices on the relation must be updated correspondingly.
- For Insertion and deletion it may be necessary to **split** a node that becomes too large as the result of an insertion, or to **coalesce nodes** (that is, combine nodes) if a node becomes too small (fewer than ⌈n/2⌉ pointers).

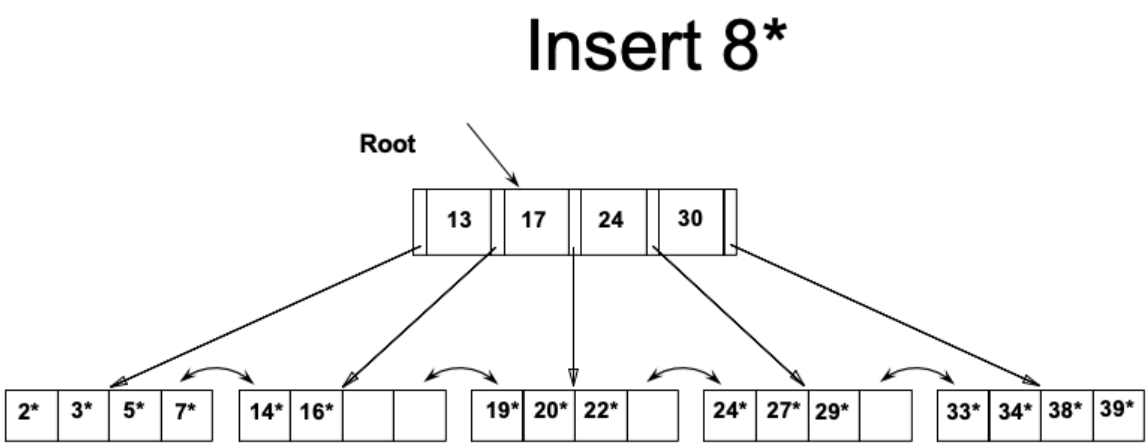
Insertion:

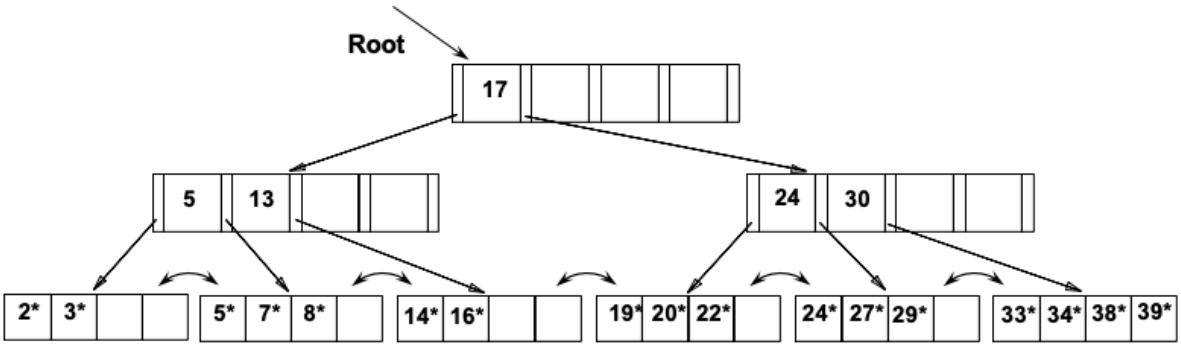
To insert 23,





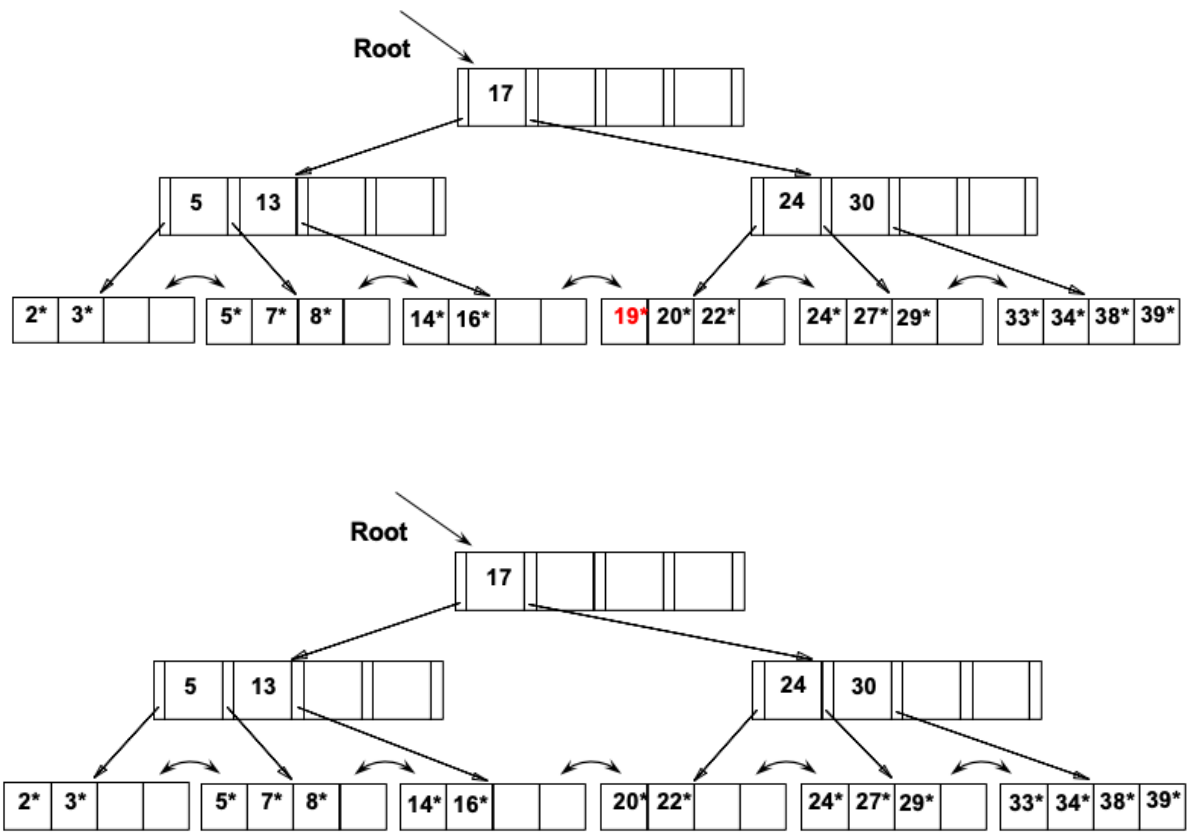
To insert 8:



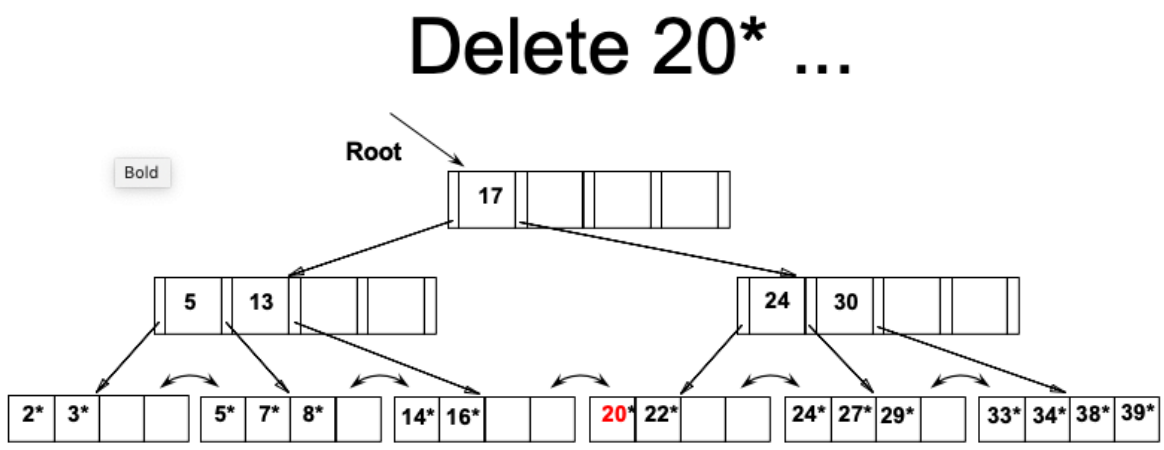


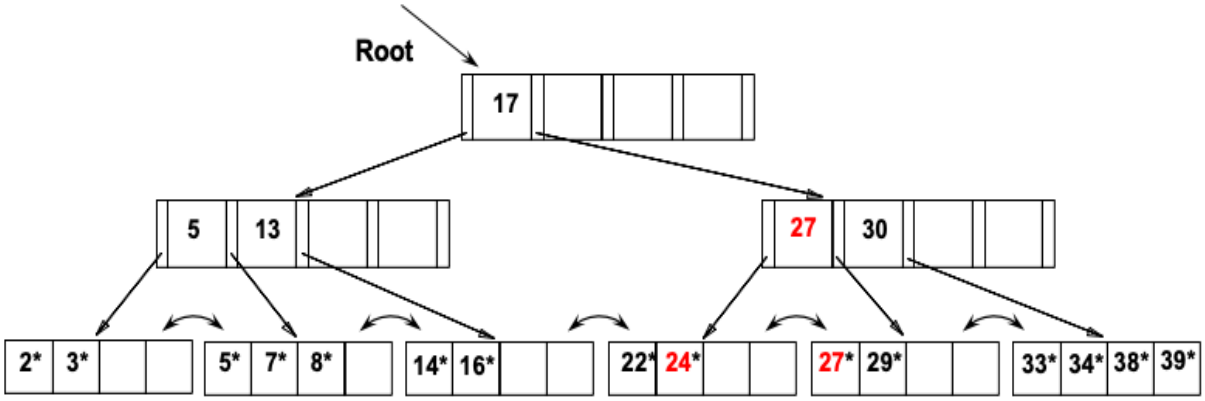
Deletion:

To delete 19:

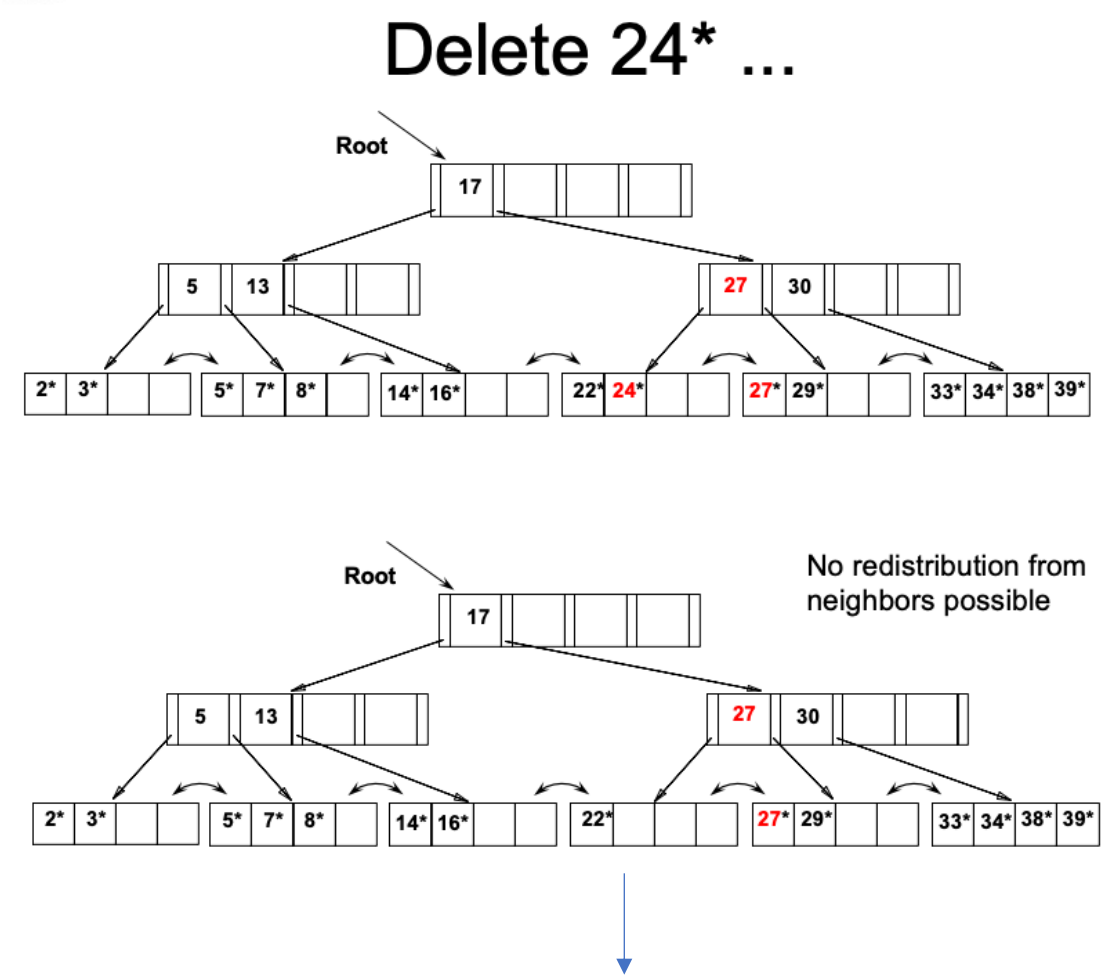

Delete 19*
Root
17
5
13
24
30
2*
3*
5*
7*
8*
14*
16*
19*
20*
22*
24*
27*
29*
33*
34*
38*
39*
Root
17
5
13
24
30
2*
3*
5*
7*
8*
14*
16*
20*
22*
24*
27*
29*
33*
34*
38*
39*

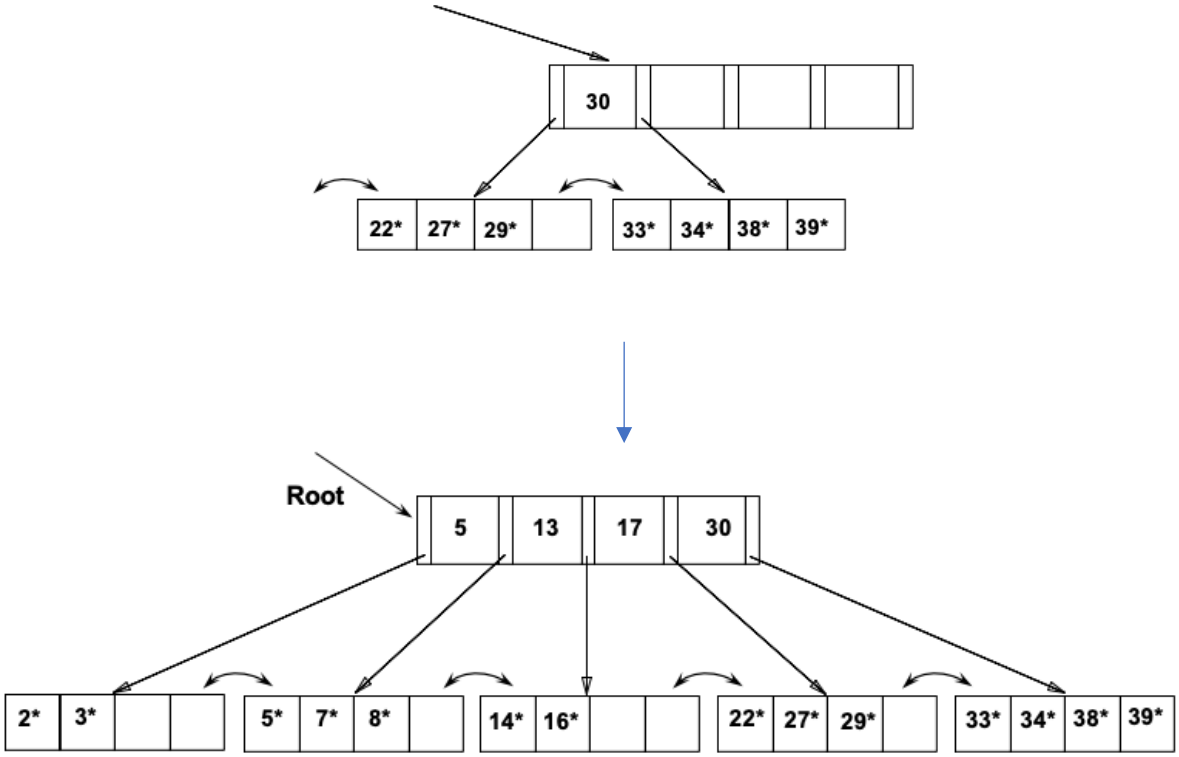
To delete 20:


Delete 20* ...
Bold
Root
17
5
13
24
30
2*
3*
5*
7*
8*
14*
16*
20*
22*
24*
27*
29*
33*
34*
38*
39*


Root
17
5
13
27
30
2*
3*
5*
7*
8*
14*
16*
22*
24*
27*
29*
33*
34*
38*
39*

To delete 24:


Delete 24* ...
Root
17
5
13
27
30
2*
3*
5*
7*
8*
14*
16*
22*
24*
27*
29*
33*
34*
38*
39*
Root
No redistribution from
neighbors possible
17
5
13
27
30
2*
3*
5*
7*
8*
14*
16*
22*
27*
29*
33*
34*
38*
39*



Algorithm for B+ Tree Insertion

- B+ trees are filled from bottom and each entry is done at the leaf node.
- If a leaf node overflows −
- Split node into two parts.
  Partition at i = ⌊(m+1)/2⌋.
- First i entries are stored in one node.
- Rest of the entries (i+1 onwards) are moved to a new node.
- i^{th} key is duplicated at the parent of the leaf.

If a non-leaf node overflows

- Split node into two parts.
  Partition the node at i = ⌈(m+1)/2⌉.
- Entries up to i are kept in one node.
- Rest of the entries are moved to a new node.

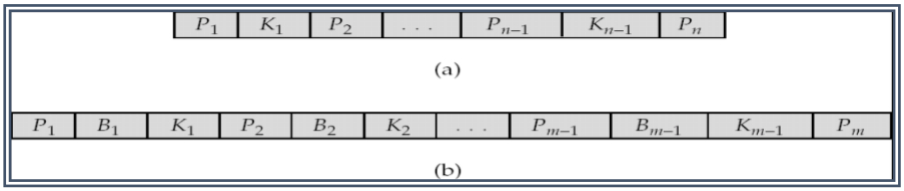
Algorithm for B+ Tree Deletion:

- B+ tree entries are deleted at the leaf nodes.
- The target entry is searched and deleted.
  If it is an internal node, delete and replace with the entry from the left position.
- After deletion, underflow is tested,
  If underflow occurs, distribute the entries from the nodes left to it.
- If distribution is not possible from left, then
  Distribute from the nodes right to it.
- If distribution is not possible from left or from right, then
  Merge the node with left and right to it.

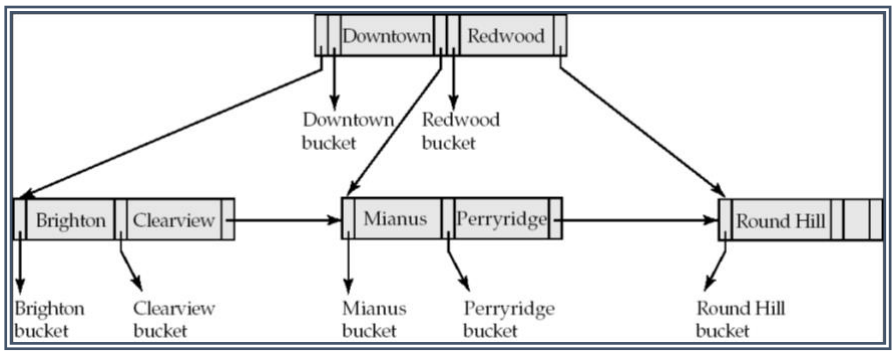
B Tree Indexing

B-tree indices are similar to B+-tree indices. The primary distinction between the two approaches is that a B-tree eliminates the redundant storage of search-key values. A B-tree allows search-key values to appear only once. Since search keys are not repeated in the B-tree, we may be able to store the index in fewer tree nodes than in the corresponding B+-tree index.

Structure of a B Tree



Example:



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:

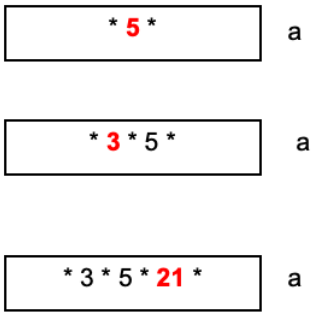
- Only small fraction of all search-key values are found early
- Non-leaf nodes are larger, so fan-out is reduced. Thus, B-Trees typically have greater depth than corresponding B^+-Tree
- Insertion and deletion more complicated than in B^+-Trees
- Implementation is harder than B^+-Trees.

B Tree operations:

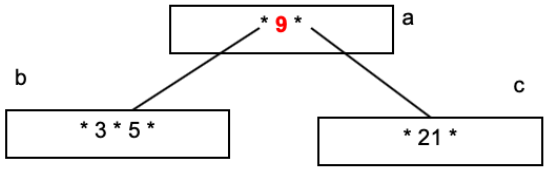
- B-Tree of order 4
  - Each node has at most 4 pointers and 3 keys, and at least 2 pointers and 1 key.
- Insert: 5, 3, 21, 9, 1, 13, 2, 7, 10, 12, 4, 8
- Delete: 2, 21, 10, 3, 4

Insertion:

To insert 5,3,21

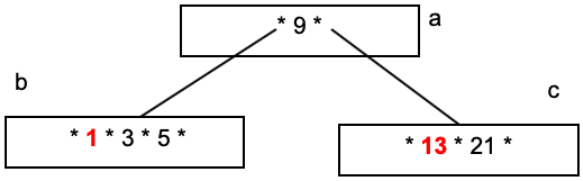


To insert 9:

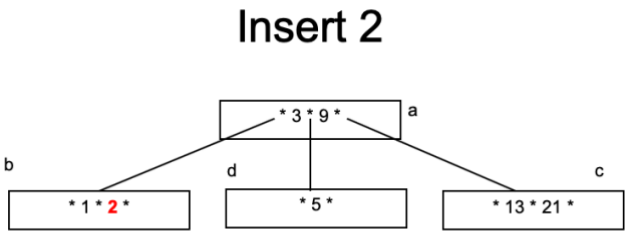


Node a splits creating 2 children: b and c

To insert 1,13:

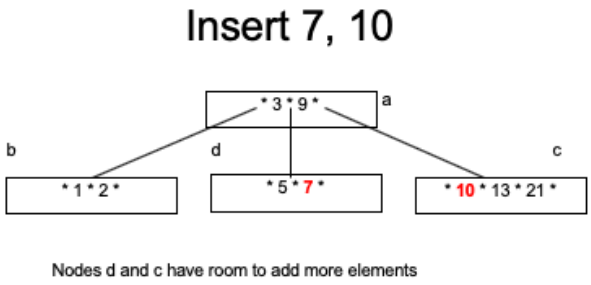


To insert 2:

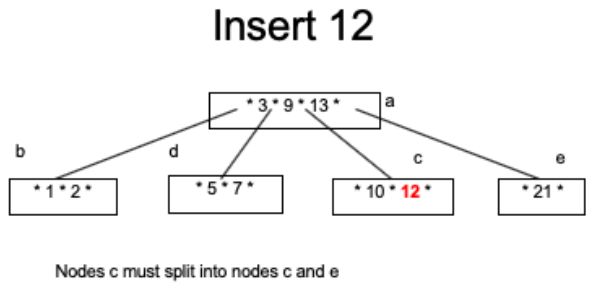


Node b has no more room, so it splits creating node d.

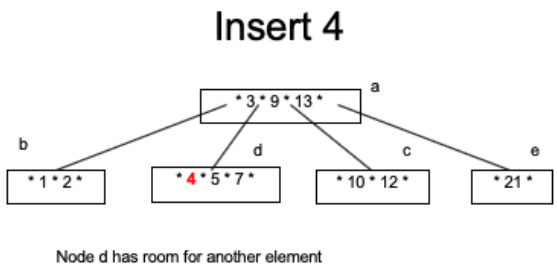
To insert 7,10:



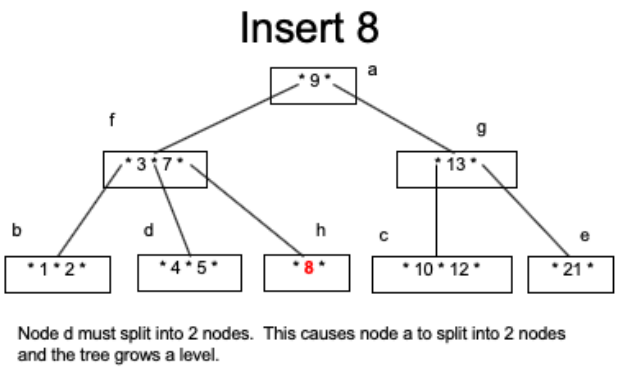
To insert 12:



To insert 4:

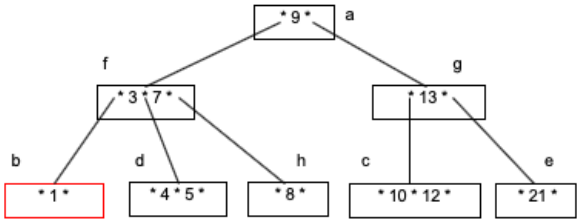


To insert 8:



Deletion:

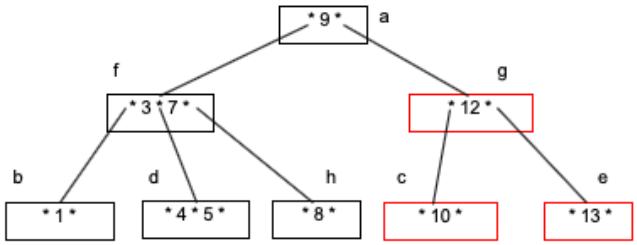
Delete 2



Node b can loose an element without underflow.

To delete 21:

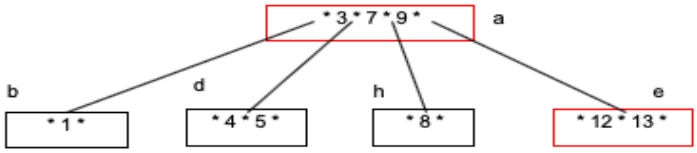
Delete 21



Deleting 21 causes node e to underflow, so elements are redistributed between nodes c, g, and e

To delete 10:

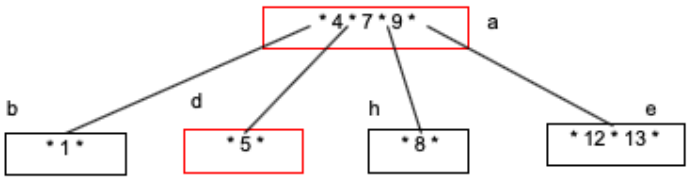
Delete 10



Deleting 10 causes node c to underflow. This causes the parent, node g to recombine with nodes f and a. This causes the tree to shrink one level.

To delete 3:

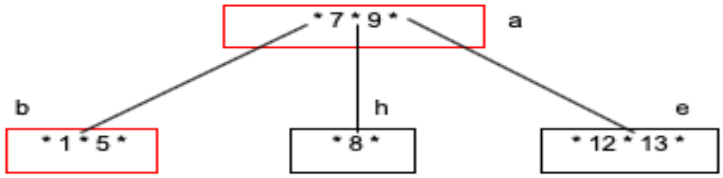
Delete 3



Because 3 is a pointer to nodes below it, deleting 3 requires keys to be redistributed between nodes a and d.

To delete 4:

Delete 4

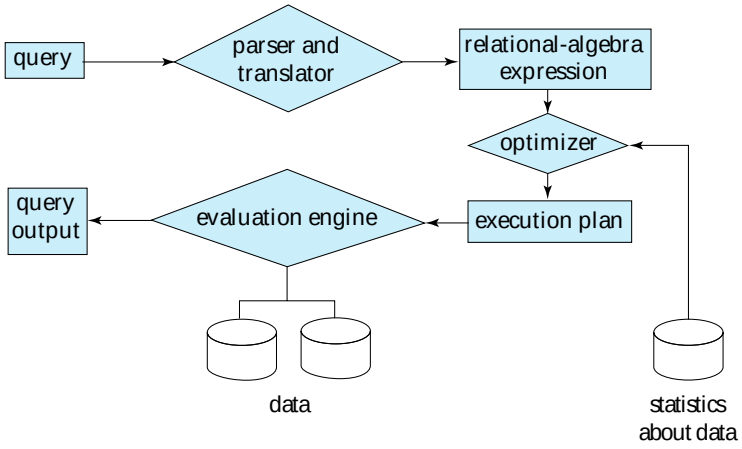


Deleting 4 requires a redistribution of the keys in the subtrees of 4; however, nodes b and d do not have enough keys to redistribute without causing an underflow. Thus, nodes b and d must be combined.

Query Processing

Query processing refers to the range of activities involved in extracting data from a database. The activities include translation of queries in high-level database languages into expressions, a variety of query-optimizing transformations, and actual evaluation of queries.

The steps involved in processing a query are:
1. Parsing and translation.
2.Optimization.
3. Evaluation.



Parsing and translation: Parsing can be done in three stages. They are

- Scanning
- Parsing
- Validating

The scanner identifies the query tokens—such as SQL keywords, attribute names, and relation names—that appear in the text of the query.

The parser checks the query syntax to determine whether it is formulated according to the syntax rules (rules of grammar) of the query language.

The query must also be validated by checking that all attribute and relation names are valid and semantically meaningful names in the schema of the particular database being queried.

The system constructs a parse-tree representation of the query, which it then translates into a relational-algebra expression.

Query optimization:

The DBMS must then devise an **execution strategy or query plan** for retrieving the results of the query from the database files. A query typically has many possible execution strategies, and the process of choosing a suitable one for processing a query is known as query optimization. The query optimizer module has the task of producing a good execution plan.

Evaluation:

The **code generator** generates the code to execute that plan. The **runtime database processor** has the task of running (executing) the query code, whether in compiled or interpreted mode, to produce the query result. If a runtime error results, an error message is generated by the runtime database processor.

Algorithms for SELECT and JOIN operations:

1) SELECT OPERATIONS:

In query processing, the file scan is the lowest-level operator to access data. Consider a selection operation on a relation whose tuples are stored together in one file. The most straightforward way of performing a selection is as follows:

(A1) Linear search: In a linear search, the system scans each file block and tests all records to see whether they satisfy the selection condition. An initial seek is required to access the first block of the file

Search algorithms that use an index are referred to as index scans. Index structures are referred to as access paths, since they provide a path through which data can be located and accessed.

(A2)Primary index, equality on key: For an equality comparison on a key attribute with a primary index, we can use the index to retrieve a single record that satisfies the corresponding equality condition.

(A3)Primary index, equality on non-key: We can retrieve multiple records by using a primary index when the selection condition specifies an equality comparison on a non-key attribute,

(A4)Secondary index, equality: Selections specifying an equality condition can use a secondary index. This strategy can retrieve a single record if the equality condition is on a key;

Selections Involving Comparisons

(A5)Primary index, Comparison : A primary ordered index (for example, a primary B+-tree index) can be used when the selection condition is a comparison.

A6)Secondary index, comparison: We can use a secondary ordered index to guide retrieval for comparison conditions involving $<$, $\leq$, $\geq$, or $>$

	Algorithm	Cost	Reason
A1	Linear Search	$t_S + b_r * t_T$	One initial seek plus b_r block transfers, where b_r denotes the number of blocks in the file.
A1	Linear Search, Equality on Key	Average case $t_S + (b_r/2) * t_T$	Since at most one record satisfies condition, scan can be terminated as soon as the required record is found. In the worst case, b_r blocks transfers are still required.
A2	Primary B^+-tree Index, Equality on Key	$(h_i + 1) * (t_T + t_S)$	(Where h_i denotes the height of the index.) Index lookup traverses the height of the tree plus one I/O to fetch the record; each of these I/O operations requires a seek and a block transfer.
A3	Primary B^+-tree Index, Equality on Nonkey	$h_i * (t_T + t_S) + b * t_T$	One seek for each level of the tree, one seek for the first block. Here b is the number of blocks containing records with the specified search key, all of which are read. These blocks are leaf blocks assumed to be stored sequentially (since it is a primary index) and don't require additional seeks.
A4	Secondary B^+-tree Index, Equality on Key	$(h_i + 1) * (t_T + t_S)$	This case is similar to primary index.
A4	Secondary B^+-tree Index, Equality on Nonkey	$(h_i + n) * (t_T + t_S)$	(Where n is the number of records fetched.) Here, cost of index traversal is the same as for A3, but each record may be on a different block, requiring a seek per record. Cost is potentially very high if n is large.
A5	Primary B^+-tree Index, Comparison	$h_i * (t_T + t_S) + b * t_T$	Identical to the case of A3, equality on nonkey.
A6	Secondary B^+-tree Index, Comparison	$(h_i + n) * (t_T + t_S)$	Identical to the case of A4, equality on nonkey.

Figure 12.3 Cost estimates for selection algorithms.

Implementation of Complex Selections

- **Conjunction:** A *conjunctive selection* is a selection of the form:

$$\sigma_{\theta_1 \wedge \theta_2 \wedge \cdots \wedge \theta_n}(r)$$

- **Disjunction:** A *disjunctive selection* is a selection of the form:

$$\sigma_{\theta_1 \vee \theta_2 \vee \cdots \vee \theta_n}(r)$$

A7 (Conjunctive selection using one index): We first determine whether an access path is available for an attribute in one of the simple conditions. If one is, one of the selection algorithms A2 through A6 can retrieve records satisfying that condition.

A8(Conjunctive selection using composite index): An appropriate composite index (that is, an index on multiple attributes) may be available for some conjunctive selections. If the selection specifies an equality condition on two or more attributes, and a composite index exists on these combined attribute fields, then the index can be searched directly.

A10(Disjunctive selection by union of identifiers): If access paths are available on all the conditions of a disjunctive selection, each index is scanned for pointers to tuples that satisfy the individual condition. The union of all the retrieved pointers yields the set of pointers to all tuples that satisfy the disjunctive condition.

2) Join Operation

The term equijoin to refer to a join of the form $r \bowtie_{r.A=s.B} s$, where A and B are attributes or sets of attributes of relations r and s, respectively.

- Number of records of *student*: $n_{student}$ = 5, 000.
- Number of blocks of *student*: $b_{student}$ = 100.
- Number of records of *takes*: n_{takes} = 10, 000.
- Number of blocks of *takes*: b_{takes} = 400.

Nested-Loop Join

A simple algorithm to compute the theta join $r \bowtie_\theta s$ of two relations r and s is shown below. This algorithm is called the nested-loop join algorithm, since it basically consists of a pair of nested for loops. Relation r is called the outer relation and relation s the inner relation of the join, since the loop for r encloses the loop for s.

```
for each tuple t_r in r do begin
    for each tuple t_s in s do begin
        test pair (t_r, t_s) to see if they satisfy the join condition θ
        if they do, add t_r · t_s to the result;
    end
end
```

Block nested-loop join:

Block nested-loop join, which is a variant of the nested-loop join where every block of the inner relation is paired with every block of the outer relation

```
for each block B_r of r do begin
    for each block B_s of s do begin
        for each tuple t_r in B_r do begin
            for each tuple t_s in B_s do begin
                test pair (t_r, t_s) to see if they satisfy the join condition
                if they do, add t_r · t_s to the result;
            end
        end
    end
end
```

Indexed Nested-Loop Join

For each tuple t_r in the outer relation r, the index is used to look up tuples in s that will satisfy the join condition with tuple t_r . This join method is called an indexed nested-loop join; it can be used with existing indices.

Merge Join

The merge-join algorithm (also called the sort-merge-join algorithm) can be used to compute natural joins and equijoins. Let r(R) and s(S) be the relations whose natural join is to be computed, and let R ∩ S denote their common attributes

```
pr := address of first tuple of r;
ps := address of first tuple of s;
while (ps ≠ null and pr ≠ null) do
    begin
        t_s := tuple to which ps points;
        S_s := {t_s};
        set ps to point to next tuple of s;
        done := false;
        while (not done and ps ≠ null) do
            begin
                t_s' := tuple to which ps points;
                if (t_s'[JoinAttrs] = t_s[JoinAttrs])
                    then begin
                            S_s := S_s ∪ {t_s'};
                            set ps to point to next tuple of s;
                         end
                    else done := true;
            end
        t_r := tuple to which pr points;
        while (pr ≠ null and t_r[JoinAttrs] < t_s[JoinAttrs]) do
            begin
                set pr to point to next tuple of r;
                t_r := tuple to which pr points;
            end
        while (pr ≠ null and t_r[JoinAttrs] = t_s[JoinAttrs]) do
            begin
                for each t_s in S_s do
                    begin
                        add t_s ✶ t_r to result;
                    end
                set pr to point to next tuple of r;
                t_r := tuple to which pr points;
            end
    end.
```

Evaluation of Expressions:

Two approaches are used for evaluating the expressions:

- Materialized evaluation
- Pipelining

Materialized evaluation

The result of each evaluation is materialized in a temporary relation for subsequent use. A disadvantage to this approach is the need to construct the temporary relations,

Pipelining:

In a pipeline, with the results of one operation passed on to the next, without the need to store a temporary relation.

Query optimization

Query optimization is the process of selecting the most efficient query-evaluation plan from among the many strategies usually possible for processing a given query.

Types of optimization:

- Heuristic based optimization
- Cost based optimization

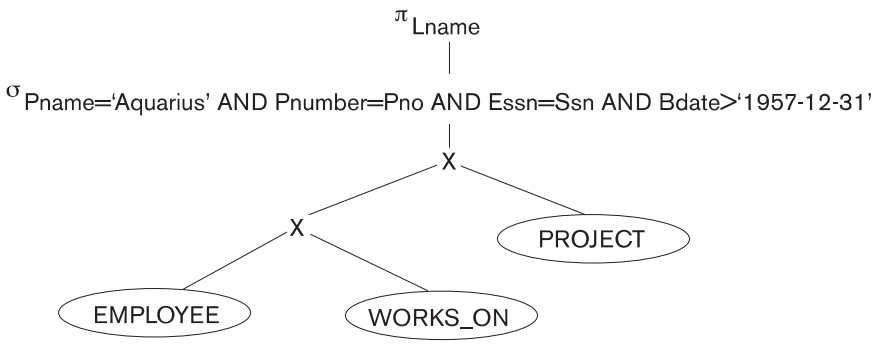
Heuristic based optimization:

Heuristic based optimization relies on heuristic reordering of relational algebra operations. One of the main heuristic rules is to apply SELECT and PROJECT operations before applying the JOIN or other binary operation

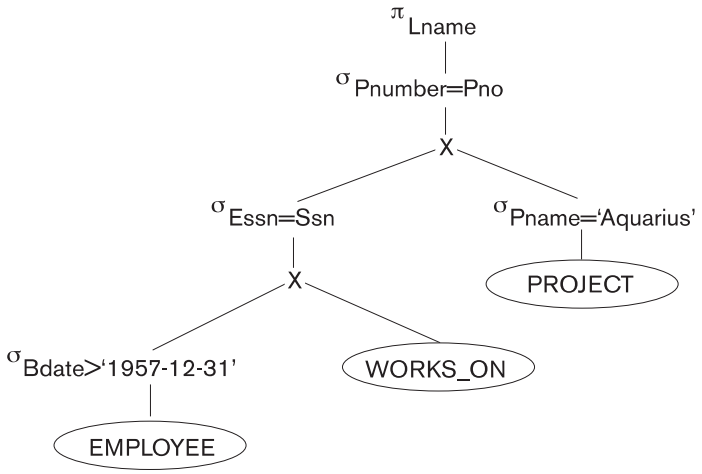
```
SELECT  Lname
FROM    EMPLOYEE, WORKS_ON, PROJECT
WHERE   Pname='Aquarius' AND Pnumber=Pno AND Essn=Ssn
        AND Bdate > '1957-12-31';
```

Steps in converting a query tree during heuristic optimization.
(a) Initial (canonical) query tree for SQL query Q.
(b) Moving SELECT operations down the query tree.
(c) Applying the more restrictive SELECT operation first.
(d) Replacing CARTESIAN PRODUCT and SELECT with JOIN operations.
(e) Moving PROJECT operations down the query tree.

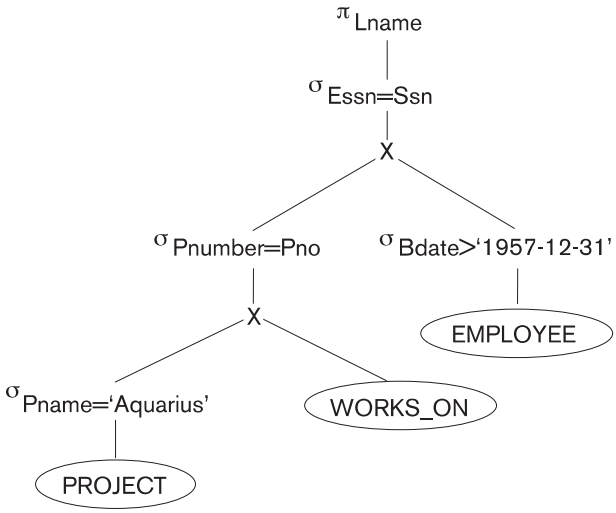
(a)

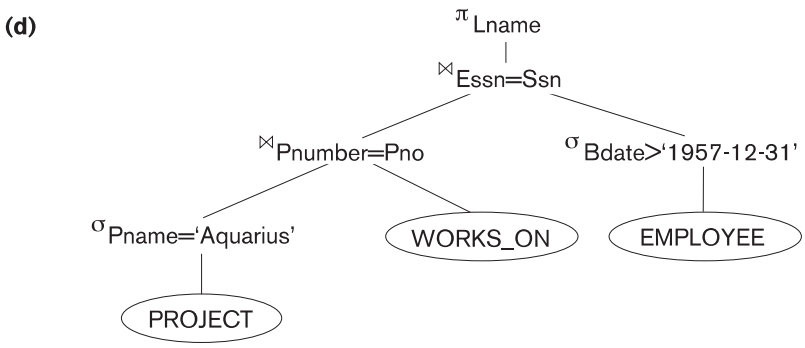

π Lname
σ Pname='Aquarius' AND Pnumber=Pno AND Essn=Ssn AND Bdate>'1957-12-31'
X
X
PROJECT
EMPLOYEE
WORKS_ON

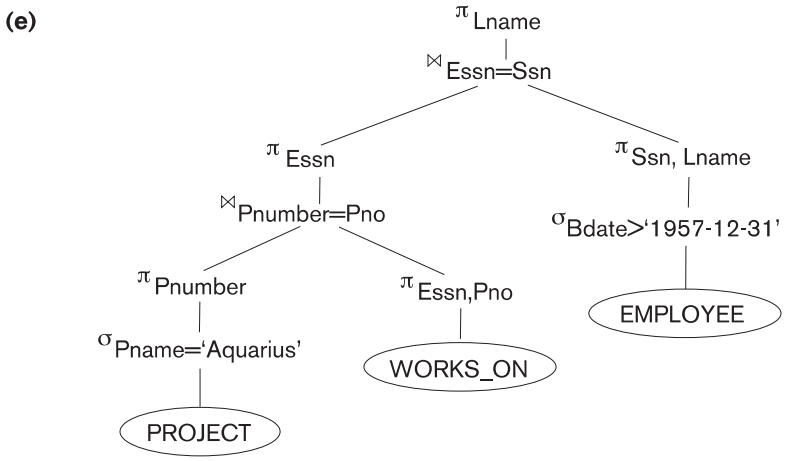
(b)


π Lname
σ Pnumber=Pno
X
σ Essn=Ssn
σ Pname='Aquarius'
X
PROJECT
σ Bdate>'1957-12-31'
WORKS_ON
EMPLOYEE

(c)


π Lname
σ Essn=Ssn
X
σ Pnumber=Pno
σ Bdate>'1957-12-31'
X
EMPLOYEE
σ Pname='Aquarius'
WORKS_ON
PROJECT





General Transformation Rules for Relational Algebra Operations.

1. **Cascade of σ** A conjunctive selection condition can be broken up into a cascade (that is, a sequence) of individual σ operations:
   $\sigma_{c_1 \text{ AND } c_2 \text{ AND } \ldots \text{ AND } c_n}(R) \equiv \sigma_{c_1}(\sigma_{c_2}(\ldots(\sigma_{c_n}(R))\ldots))$
2. **Commutativity of σ.** The σ operation is commutative:
   $\sigma_{c_1}(\sigma_{c_2}(R)) \equiv \sigma_{c_2}(\sigma_{c_1}(R))$
3. **Cascade of π.** In a cascade (sequence) of π operations, all but the last one can be ignored:
   $\pi_{\text{List}_1}(\pi_{\text{List}_2}(\ldots(\pi_{\text{List}_n}(R))\ldots)) \equiv \pi_{\text{List}_1}(R)$
4. **Commuting σ with π.** If the selection condition c involves only those attributes $A_1, \ldots, A_n$ in the projection list, the two operations can be commuted:
   $\pi_{A_1, A_2, \ldots, A_n}(\sigma_c(R)) \equiv \sigma_c(\pi_{A_1, A_2, \ldots, A_n}(R))$

Commutativity of set operations. The set operations ∪ and ∩ are commutative but − is not.

Associativity of ⋈, ×, ∪, and ∩. These four operations are individually associative; that is, if θ stands for any one of these four operations (throughout the expression), we have:
$(R \theta S) \theta T \equiv R \theta (S \theta T)$

Cost based optimization:

Cost based optimization involves the systematic estimating the cost of different execution plan and choosing the best plan. It estimates and compares the costs of executing a query using different execution strategies and algorithms, and it then chooses the strategy with the lowest cost estimate. This approach is generally referred to as cost-based query optimization. The cost functions used in query optimization are estimates and not exact cost functions, so the optimization may select a query execution strategy that is not the optimal

Cost Components for Query Execution

The cost of executing a query includes the following components:

Access cost to secondary storage: This is the cost of transferring (reading and writing) data blocks between secondary disk storage and main memory buffers.

Disk storage cost: This is the cost of storing on disk any intermediate files that are generated by an execution strategy for the query.

Computation cost: This is the cost of performing memory operations on the records within the data buffers during query execution.

Memory usage cost: This is the cost pertaining to the number of main memory buffers needed during query execution.

Communication cost: This is the cost of shipping the query and its results from the database site to the site or terminal where the query originated.

Cost Functions for Cost based optimization

To estimate the costs of various execution strategies, we must keep track of any information that is needed for the cost functions.

Consider,

N**umber of records (tuples) – *r***
Record size – *R*
Number of file blocks - *b*
Blocking factor - *bfr*

Cost Functions for SELECT

- Linear search (brute force) approach: CS1a = b.
- Binary search. This search accesses approximately CS2 = log2b + $\lceil (s/bfr) \rceil - 1$ file blocks
- Using a primary index to retrieve a single record : CS3a = x + 1.

The following strategies can be used in cost based optimization:

- Statistics
- Cost
- Selectivity
