UNIT-IV

Storage and file structure: Overview of physical storage media, magnetic disk and flash storage, RAID, file organisation, Data dictionary storage, database buffer.

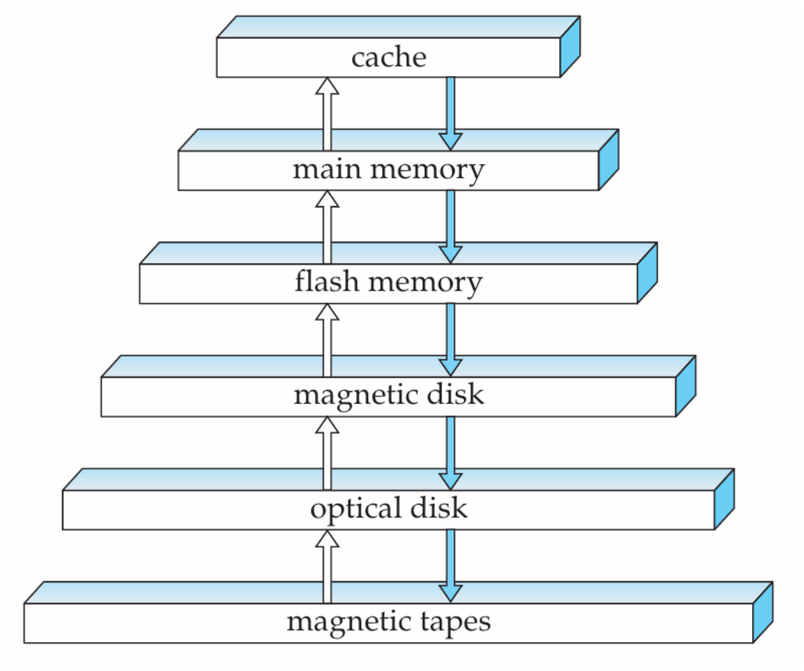
Indexing and hashing: Types of single level ordered indexes, multilevel indexes, dynamic multilevel indexes, hashing, static hashing , dynamic hashing

OVERVIEW OF PHYSICAL STORAGE MEDIA

Several types of data storage exist in most computer systems.

These storage media are classified by

- The speed with which data can be accessed,
- By the cost per unit of data to buy the medium, and
- By the medium’s reliability
  - Data loss on power failure or system crash
  - Physical failure of the storage device



Storage device hierarchy

The various storage media can be organized in a hierarchy as shown in the above figure.
Classification: The memory can be classified into two types.

1. volatile storage: loses contents when power is turned off
2. non-volatile storage: contents persist when power is switched off

Physical Storage Media: Typically,the following physical storage media are available.

*Cache:*The following are the characteristics of Cache memory.

- fastest and most costly form of storage
- volatile
- managed by the hardware and/or operating system

Main Memory: The following are the characteristics of Main memory.

- general purpose instructions operate on data in main memory
- fast access but in general too small to store the entire database (or even an entire relation)
- volatile content of main memory is usually lost if a power failure or system crash occurs

MAGNETIC DISK

Magnetic-Disk Storage: The following are the characteristics of Magnetic-Disk Storage memory.

- primary medium for the long-term storage of data
- typically stores the entire database (i.e., all relations and associated access structures)
- data must be moved from disk to main memory for access and written back for storage (insert, update, delete, select)
- direct-access, i.e., it is possible to read data on disk in any order
- usually survives power failures and system crashes
- disk failure, however, can destroy data, but is much less frequent than system crashes

Optical Storage: The following are the characteristics of Optical Storage memory.

- non volatile
- CD-ROM/DVD most popular form
- Write-Once-Read-Many (WORM) optical disks are typically used for archival storage

Tape Storage: The following are the characteristics of Tape Storage memory.

- non-volatile
- used primarily for backup and export (to recover from disk failures and to restore previous data), and for archival data
- sequential access
- much slower than disk
- very high capacity (8GB tapes are common)
- tape can be removed from drive
- storage cost much cheaper than disk

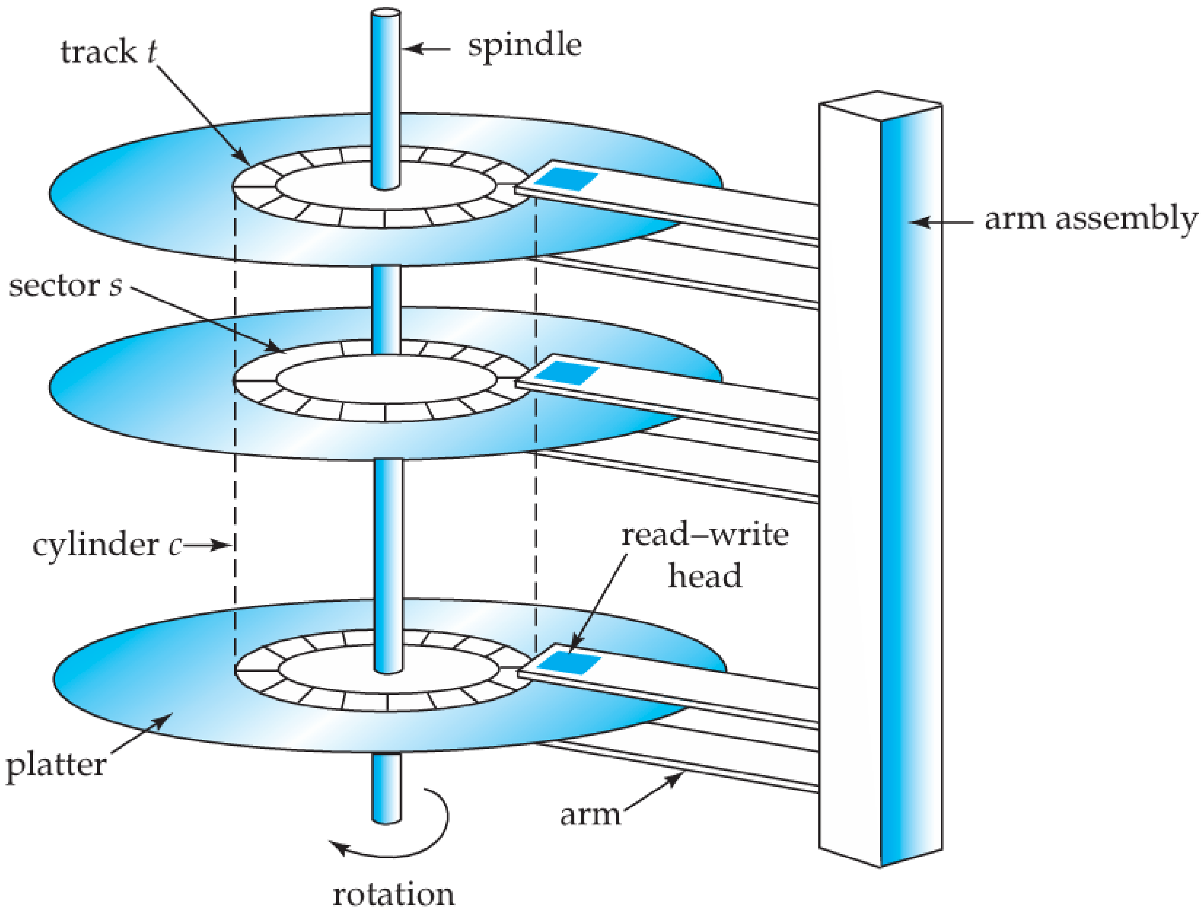
Storage Hierarchy: The various storage media can be organized in a hierarchy according to their speed and their cost. The higher levels are expensive, but arefast. As we move down the hierarchy, the cost per bit decreases, whereas theaccess time increases.

Primary Storage: Fastest media but volatile (cache, main memory)

Secondary Storage: next lower level in hierarchy, non-volatile, moderately fast access time, sometimes also called on-line storage (magnetic disks, ash memory)

Tertiary Storage: lowest level in hierarchy, non-volatile, slow access time, also called off-line storage (magnetic tape, optical storage)

Magnetic disks provide the bulk of secondary storage for modern computer systems.



Moving head disk mechanism.

Physical Characteristics

- Physically, disks are relatively simple
- Each disk platter has a flat, circular shape
- Its two surfaces are covered with a magnetic material
- Platters are made from rigid metal or glass
- Surface of platter is divided into circular tracks and each track is divided into sectors
- A sector is the smallest unit of data that can be read or written

Theread–writeheadstoresinformation onasectormagneticallyasreversals of the direction of magnetization of the magnetic material.

To read/write a sector

- ❖ disk arm swings to position head on right track
- ❖ platter spins continually
- ❖ data is read/written when sector comes under head

The disk platters mounted on a spindle and the heads mounted on a disk arm are together known as head–disk assemblies.

Sincethe headsonalltheplatters move together, when the headononeplatteris ontheith track, the heads on all other platters are also on the ith track of their respective platters. Hence,theithtracksofalltheplatterstogetherarecalledtheithcylinder.

The read–write heads are kept as close as possible to the disk surface to increase the recording density.

Disks are connected to a computersystemthrough ahigh-speedinterconnection.

Commoninterfacesfor connecting disksto computers are:

1. SATA
2. SCSI
3. SAS

- **<u>Performance Measures of Disks</u>:** Themainmeasuresofthequalitiesofadiskarecapacity,accesstime,data-transfer rate, and reliability.

Access time:The time it takes from when a read or write request is issued to when data transfer begins. It consists of:

Seek time: The time it takes to reposition the arm over the correct track. Average seek time is 1/3rd the worst case seek time.

Rotational latency: The time it takes for the sector to be accessed to appear under the head. Average latency is 1/2 of the worst-case latency.

Data-transfer rate:The rate at which data can be retrieved from or stored to the disk.

***Mean time to failure (MTTF)*:**The average time the disk is expected to run continuously without any failure.

<u>Optimization of Disk-Block Access</u>:

Ablockisalogical unit consisting of a fixed number of contiguous sectors. Block sizes range from 512 bytes to severalkilobytes. Data are transferred between disk and main memory in units of blocks.

A number of techniques have been developed for improving the speed of access to blocks.

***Buffering*:** Blocks that are read from disk are stored temporarily in an in memory buffer, to satisfy future requests. Buffering is done by both the operating system and the database system.

***Read-ahead*:**Whenadiskblockisaccessed,consecutiveblocksfromthesame track are read into an in-memory buffer even if there is no pending request for the blocks.

***Scheduling*:**Disk-arm–scheduling algorithms order accesses to tracks so that disk arm movement is minimized (elevator algorithm is often used).

File organization: It optimizes block access time by organizing the blocks to correspond to how data will be accessed. It stores related information on the same or nearby cylinders.

***Non-volatile write buffers*:**It speeds up disk writes by writing blocks to a non-volatile RAM buffer immediately. The controller then writes to disk whenever the disk has no other requests.

Log disk: A disk is devoted to writing a sequential log of block updates.This eliminates seek time. It is used like non-volatile RAM.

<u>FLASH STORAGE</u>

There are two types of flash memory.

1. NOR flash
2. NANDflash

NORflash: It allowsrandomaccesstoindividualwordsofmemory. Its read time is comparable to main memory.

NAND flash:It requires an entire page of data, typically consisting of between 512 and 4096 bytes. Pages in a NAND flash are similar to sectors in a magnetic disk. It is significantly cheaper than NOR flash, and has much higher storage capacity, and is by far the more widely used.

RAID

Full form of RAID is : Redundant Arrays of Inexpensive Disks

- It is a variety of disk organization techniques that take advantage of utilizing large numbers of inexpensive, mass-market disks
- Originally a cost-effective alternative to large, expensive disks
- RAID systemsareusedfor theirhigher reliability and higher performance rate, rather than for economic reasons

<u>Improvement of Reliability via Redundancy</u>: It stores extra information that is not needed normally, but that can be used in the event of failure of a disk to rebuild the lost information.

E.g. Mirroring (or shadowing):

– duplicate every disk. Logical disk consists of two physical disks.

– every write is carried out on both disks

– if one disk in a pair fails, data still available in the other

Improvement in Performance via Parallelism: Two main goals of parallelism in a disk system: are

1. Load balance multiple small accesses to increase throughput
2. Parallelize large accesses to reduce response time

- It improves transfer rate by striping data across multiple disks.

Bit-level striping

– splits the bits of each byte across multiple disks

– In an array of eight disks, write bit i of each byte to disk i

– Each access can read data at eight times the rate of a single disk

– But seek/access time worse than for a single disk

Block-level striping

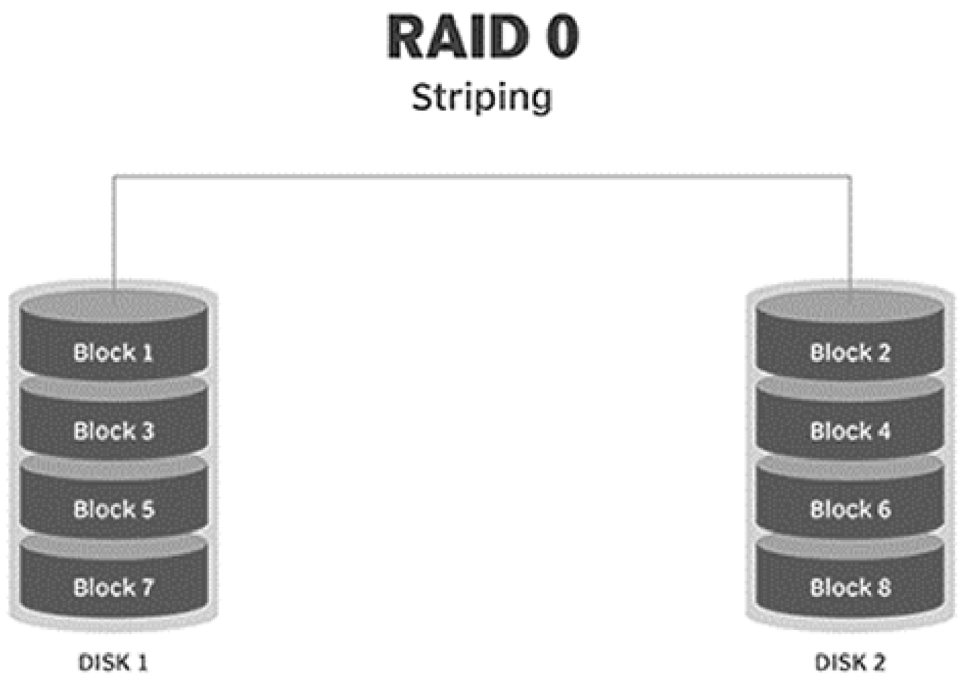
- stripes blocks across multiple disks
- treats the array of disks as a single large disk, and it gives blocks logical numbers
- the block numbers start from 0
- With an array of n disks, block-level striping assigns logical block i of the disk array to disk (i mod n) + 1
- uses the i/nth physicalblock of the disk to store logical block i

Mirroring provides high reliability, but it is expensive. Striping provides high data-transfer rates, but does not improve reliability. Various alternative schemes aim to provide redundancy at lower cost by combining disk striping with “parity” bits. These schemes have different cost–performance trade-offs. The schemes are classified into RAID levels.

RAID Levels

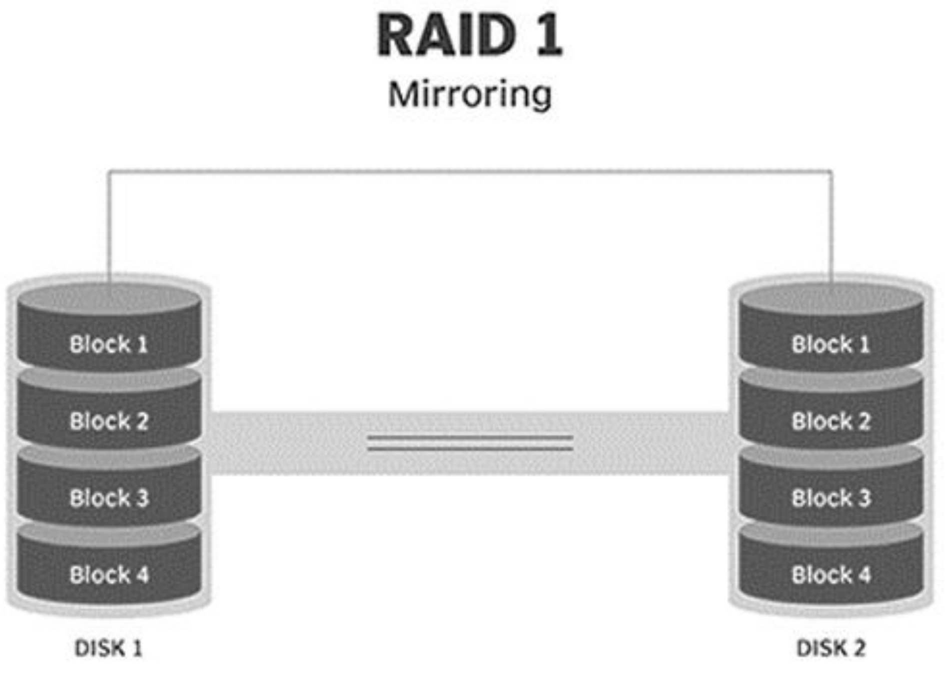
***RAID level 0*:**

- It refers to disk arrays with striping at the level of blocks
- It is non-redundant
- Used in high-performance applications where data loss is not critical
- It offers the best performance, but no fault tolerance.



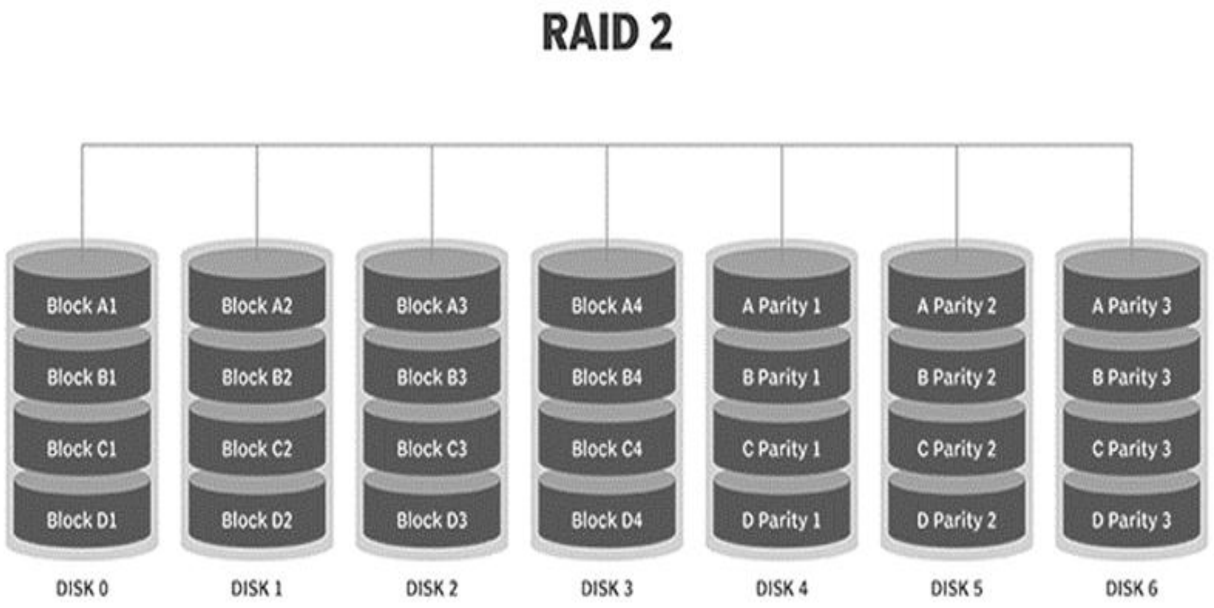
***RAID level 1*:**

- It refers to disk mirroring with block striping
- This configuration consists of at least two drives that duplicate the storage of data.
- It offers best write performance
- Popular for applications such as storing log files in a database system
- Read performance is improved since either disk can be read at the same time



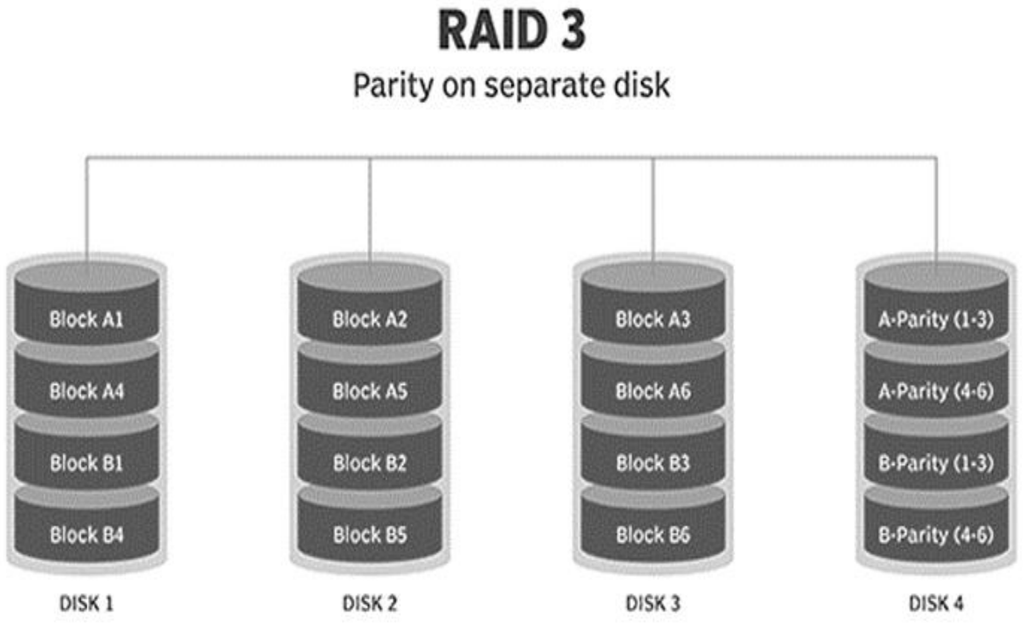
***RAID level 2*:**

- It is known as memory-style error-correcting-code (ECC) organisation, employs parity bits.
- It has no advantage over RAID 3 and is no longer used



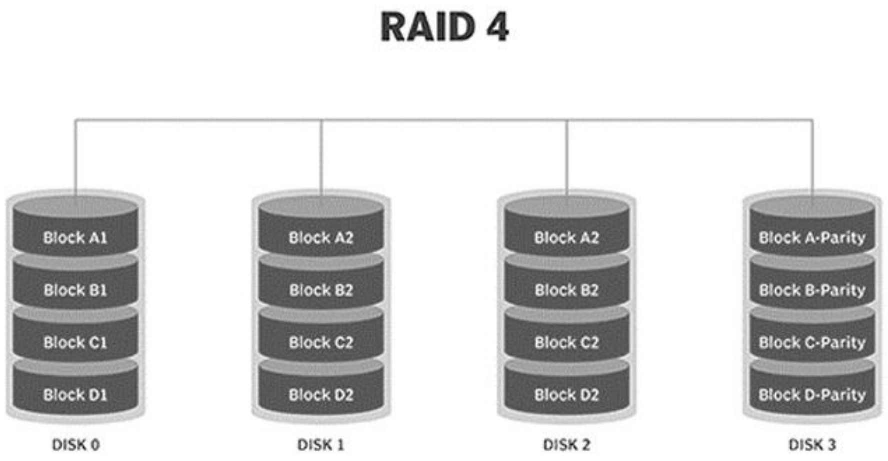
***RAID level 3*:**

- It uses bit-interleaved parity organization
- This technique uses striping and dedicates one drive to storing parity information
- A single parity bit can be used for error correction, not just detection
- Faster data transfer than with a single disk
- It is best for single-user systems with long record applications
- Whenwriting data, parity bit must also be computed and written



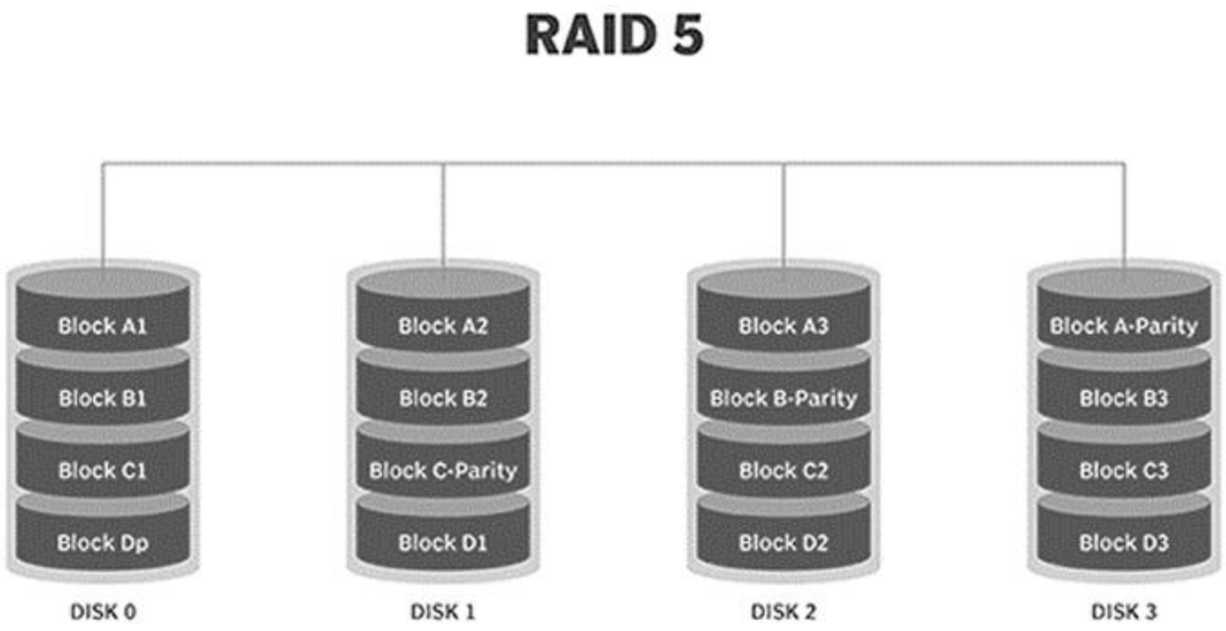
***RAID level 4*:**

- It uses Block-Interleaved Parity
- It uses block-level striping
- It keeps a parity block on a separate disk for corresponding blocks from N other disks
- It provides higher I/O rates for independent block reads than Level 3 (block read goes to a single disk, so blocks stored on different disks can be read in parallel)
- Itprovides high transfer rates for reads of multiple blocks
- Parity block becomes a bottleneck for independent block writes since every block write also writes to parity disk



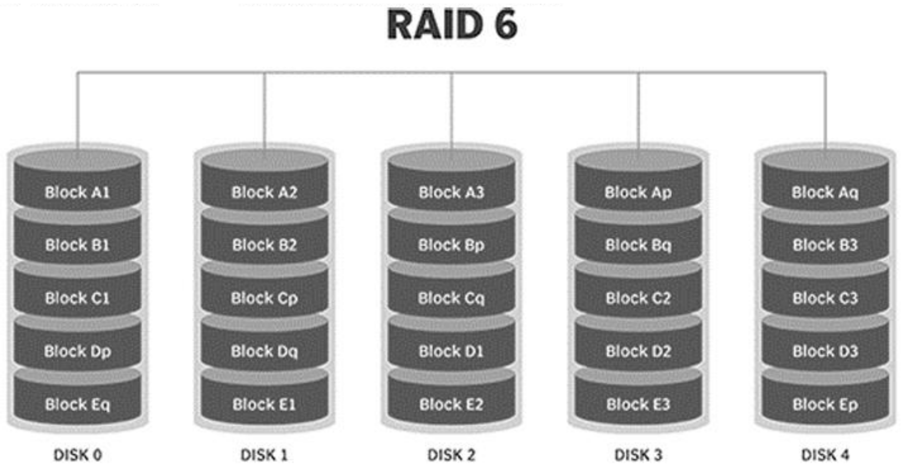
***RAID level 5*:**

- It uses Block-Interleaved Distributed Parity
- It improves on level 4 by partitioning data and parity among all N+1 disks, instead of storing data in Ndisksandparityinonedisk
- The array's architecture allows read and write operations to span multiple drives
- Higher I/O rates than Level 4. (Block writes occur in parallel if the blocks and their parity blocks are on different disks)



***RAID level 6*:**

- It uses P+Q Redundancy scheme
- It stores extra redundant information to guard against multiple disk failures
- Better reliability than Level 5 at a higher cost
- Instead of using parity, level 6 uses error-correcting codes such as the Reed Solomon codes
- RAID 6 arrays have a higher cost per gigabyte (GB) and often have slower write performance than RAID 5 arrays



Choice of RAID Level: The factors to be taken into account in choosing a RAID level are:

- Monetary cost of extra disk-storage requirements
- Performance requirements in terms of number of I/O operations
- Performance when a disk has failed
- Performance during rebuild (that is, while the data in a failed disk are being rebuilt on a new disk)

FILE ORGANIZATION

The database is stored as a collection of files. Each file is a sequence of records. A record is a sequence of fields.

Each file is also logically partitioned into fixed-length storage units called blocks, which are the units of both storage allocation and data transfer.A block may contain several records; the exact set of records that a block contains is determined by the form of physical data organization being used.

One approach to mapping the database to files:

- assume record size is fixed
- each file has records of one particular type only
- different files are used for different relations

There are two possible ways of representing the records:

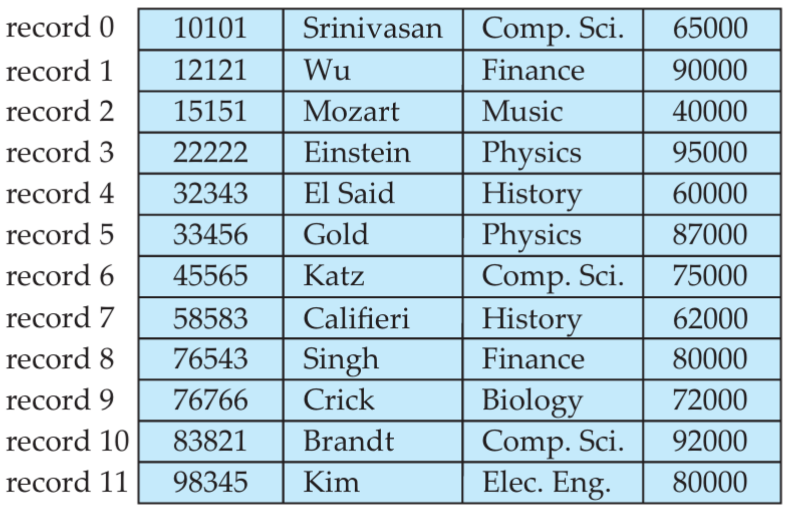
1. Fixed-length records - all the records are exactly the same length
2. Variable-length records - the length of each record varies

<u>Fixed-Length Records</u>: Fixed-length records mean setting a length and storing the records into the file. If the record size exceeds the fixed size, it gets divided into more than one block.

- At the beginning of the file, a certain number of bytes are allocated as a file header.Theheaderwillcontainavarietyofinformationaboutthefile
- Store record i starting from byte n (i − 1), where n is the size of each record
- Record access is simple but records may cross blocks
- Deletion of record i
    - move records i +1..., nto i..., n− 1
    - move record n to i
    - Link all free records on a free list
    - The deleted records form a linked list, which is often referred to as a free list
- Insertion and deletion for files of fixed-length records are simple to implement, because the space made available by a deleted record is exactly the space needed to insert a record

Example: Each record of in a file is defined (in pseudocode) as:

```
type instructor = record
                      ID varchar (5);
                      name varchar(20);
                      dept_name varchar (20);
                      salary numeric (8,2);
                  end
```



record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	12121	Wu	Finance	90000
record 2	15151	Mozart	Music	40000
record 3	22222	Einstein	Physics	95000
record 4	32343	El Said	History	60000
record 5	33456	Gold	Physics	87000
record 6	45565	Katz	Comp. Sci.	75000
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000

File containing *instructor* records.

<u>Variable-Length Records</u>: Variable-length records are the records that vary in size. It requires the creation of multiple blocks of multiple sizes to store them.

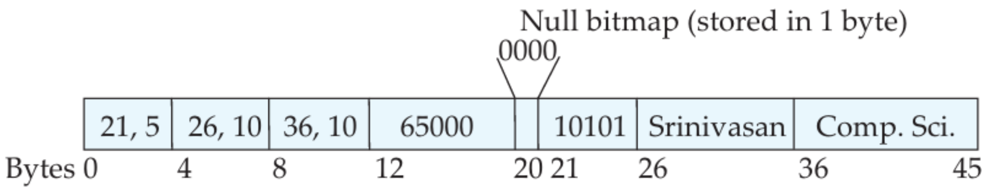
Variable-length records arise in database systems in several ways:

- Storage of multiple record types in a file
- Recordtypes that allow variable lengths for one or more fields
- Recordtypes that allow repeating fields, such as arrays or multisets

Different techniques for implementing variable-lengthrecordsexist.Twodifferent problems must be solved by any such technique:

- How to represent a single record in such a way that individual attributes can be extracted easily
- How to store variable-length records within a block, such that records in a block can be extracted easily

Representing a single record in such a way that individual attributes can be extracted easily:



Representation of variable-length record.

The above figure shows an instructor record, whose first three attributes ID, name,and dept name are variable-length strings, and whose fourth attribute salary is a fixed-sized number. We assume that the offset and length values are stored in two bytes each, for a total of 4 bytes per attribute. The salary attribute is assumed to be stored in 8 bytes, and each string takes as many bytes as it has characters.

The figure also illustrates the use of a null bitmap, which indicates which attributes of the record have a null value.

This representation is particularly useful for certain applications where records have a large number of fields, most of which are null.

Storing variable-length records within a block, such that records in a block can be extracted easily:

The slotted-page structure is commonly used for organizing records within a block.

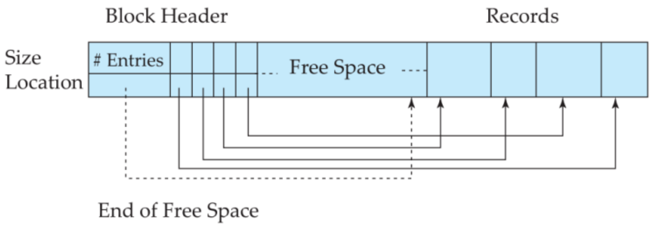


Figure 10.9 Slotted-page structure.

There is a header at the beginning of each block, containing the following information:

1. Thenumberof record entries in the header
2. Theendoffreespaceintheblock
3. Anarray whose entries contain the location and size of each record

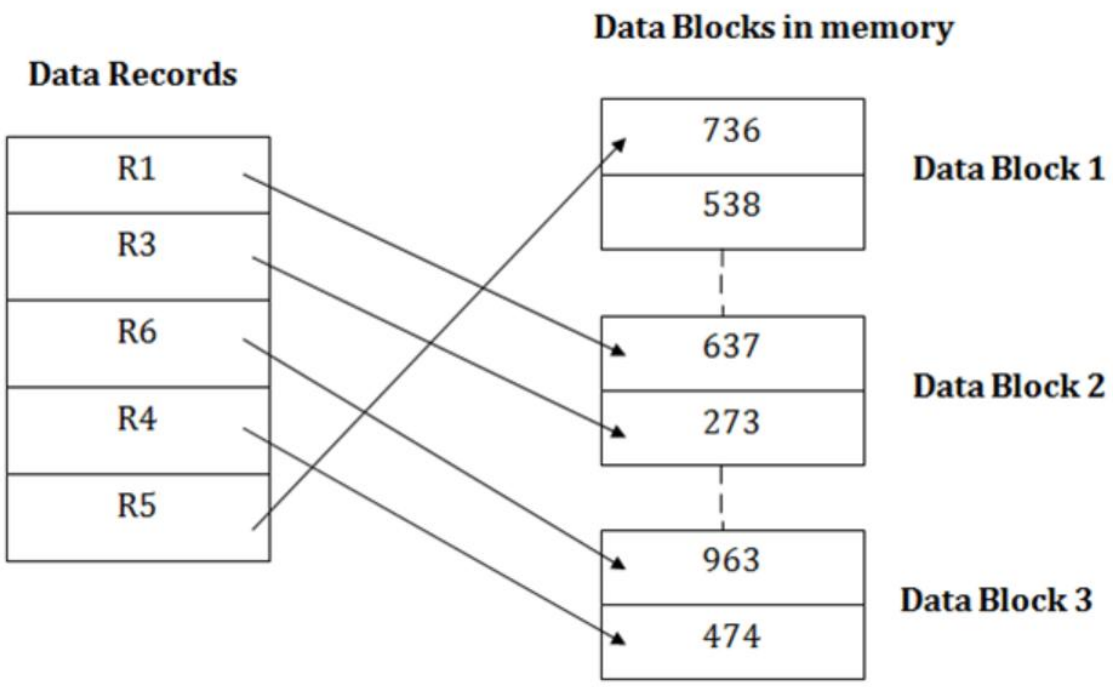
- Records can be moved around within a page to keep them contiguous with no empty space between them; entry in the header must then be updated
- Pointers should not point directly to record - instead they should point to the entry for the record in header

Organization of Records in Files: Several of the possible ways of organizing records in files are:

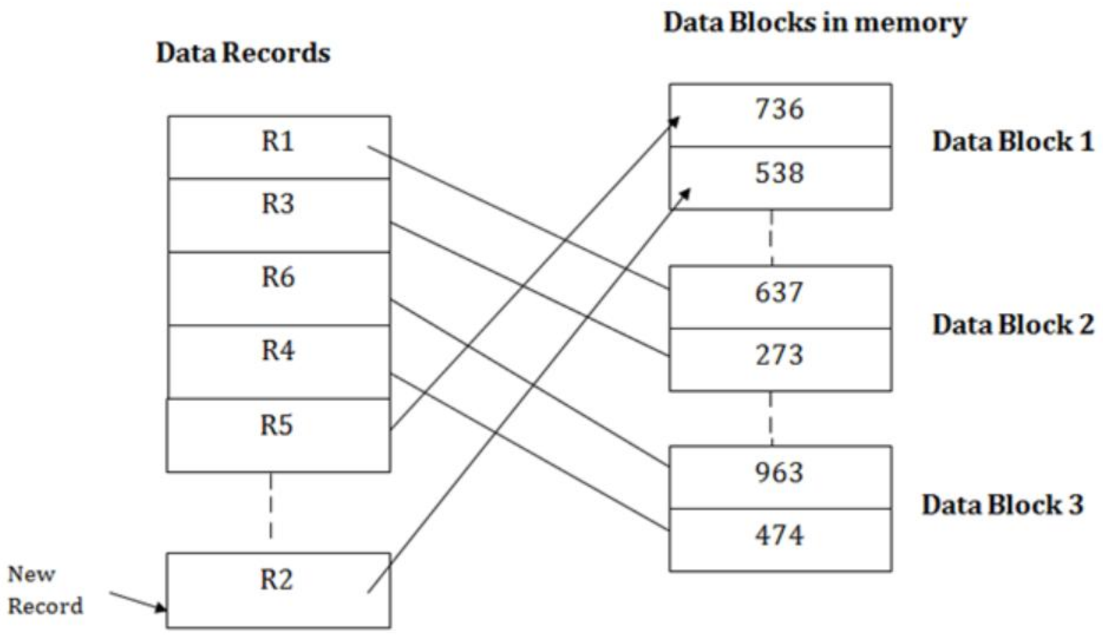
1. **Heap file organization:** Any record can be placed anywhere in the file where there is space for the record. There is no ordering of records. Typically, there is a single file for each relation.
2. **Sequential file organization:** Records are stored in sequential order, according to the value of a "search key" of each record.
3. **Hash file organization:** A hash function is computed on some attribute of each record. The result of the hash function specifies in which block of the file the record should be placed.
4. **Clustering file organization**: Records of several different relations can be stored in the same file; related records are stored on the same block.

Heap file organization: Anyrecordcan beplacedanywhere inthefilewhere there is space for the record.

- It is the simplest and most basic type of file organization
- When the records are inserted, it doesn't require the sorting and ordering of records
- When the data block is full, the new record is stored in some (any)another block
- The heap file is also known as an unordered file
- In the file, every record has a unique id, and every page in a file is of the same size



Insertion of a new record: Suppose we have five records R1, R3, R6, R4 and R5 in a heap and suppose we want to insert a new record R2 in a heap. If the data block 3 is full then it will be inserted in any of the database selected by the DBMS, let's say data block 1.



- If we want to search, update or delete the data in heap file organization, then we need to traverse the data from staring of the file till we get the requested record.
- In case of a small database, fetching and retrieving of records is faster than the sequential record.

- This method is inefficient for large databases.

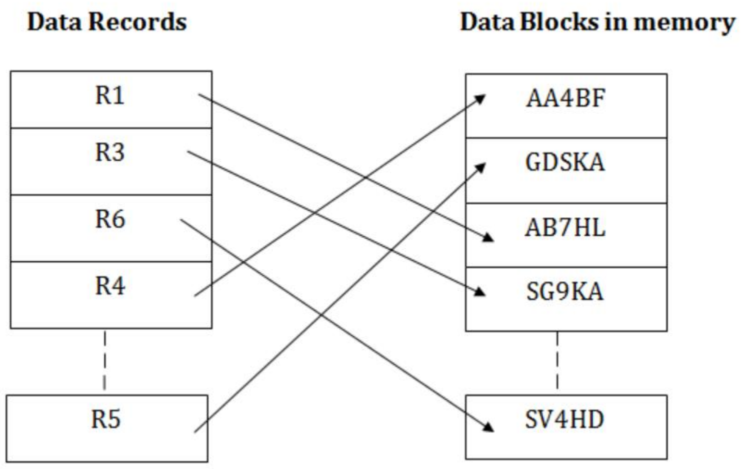
Sequential file organization: Records are stored in sequential order, according to the value of a "search key" of each record.

- Suitable for applications that require sequential processing of the entire file
- The records in the file are ordered by a search-key

Brighton	A-217	750	
Downtown	A-101	500	
Downtown	A-110	600	
Mianus	A-215	700	
Perryridge	A-102	400	
Perryridge	A-201	900	
Perryridge	A-218	700	
Redwood	A-222	700	
Round Hill	A-305	350	

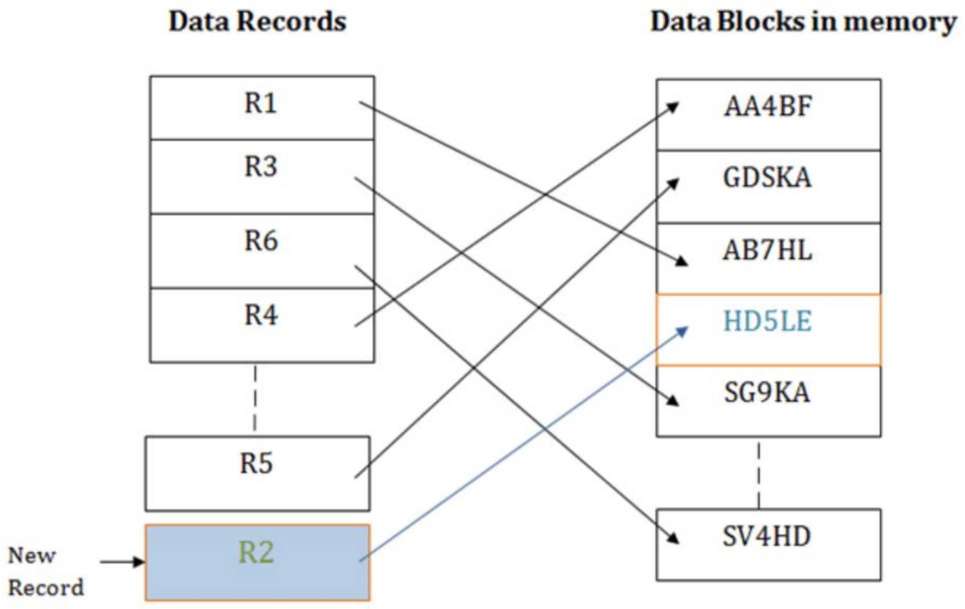
- For Deletion– it uses pointer chains
- For Insertion– it must locate the position in the file where the record is to be inserted
  - – if there is free space insert there
  - – if no free space, insert the record in an overflow block
  - – In either case, pointer chain must be updated
- Need to reorganize the file from time to time to restore sequential order

Hash file organization: A hash function is computed on some attribute of each record. The result of the hash function specifies in which block of the file the record should be placed.



- When a record has to be received using the hash key columns, then the address is generated, and the whole record is retrieved using that address.
- In the same way, when a new record has to be inserted, then the address is generated using the hash key and record is directly inserted.
- The same process is applied in the case of delete and update.

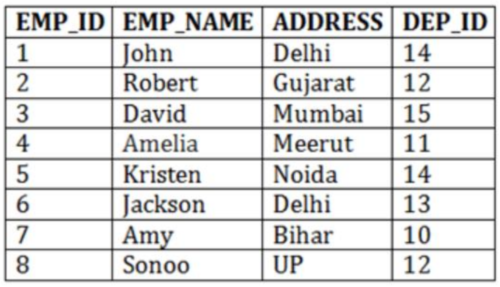
- In this method, there is no effort for searching and sorting the entire file.
- In this method, each record will be stored randomly in the memory.



<u>Clustering file organization</u>: Records of several different relations can be stored in the same file. Related records are stored on the same block.

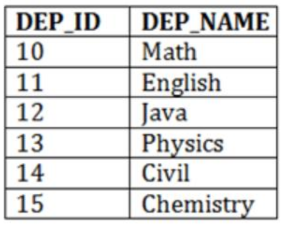
- This method reduces the cost of searching for various records in different files
- It is used when there is a frequent need for joining the tables with the same condition.

EMPLOYEE

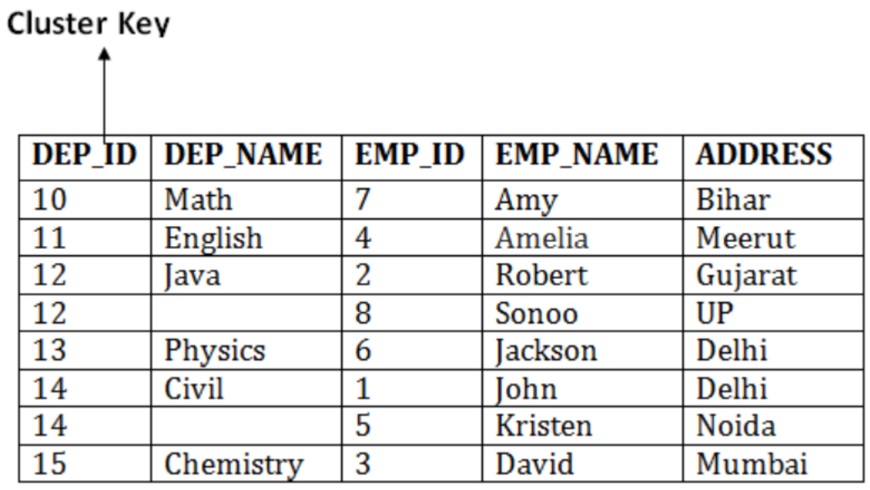


EMP_ID	EMP_NAME	ADDRESS	DEP_ID
1	John	Delhi	14
2	Robert	Gujarat	12
3	David	Mumbai	15
4	Amelia	Meerut	11
5	Kristen	Noida	14
6	Jackson	Delhi	13
7	Amy	Bihar	10
8	Sonoo	UP	12

DEPARTMENT



DEP_ID	DEP_NAME
10	Math
11	English
12	Java
13	Physics
14	Civil
15	Chemistry


Cluster Key

DEP_ID	DEP_NAME	EMP_ID	EMP_NAME	ADDRESS
10	Math	7	Amy	Bihar
11	English	4	Amelia	Meerut
12	Java	2	Robert	Gujarat
12		8	Sonoo	UP
13	Physics	6	Jackson	Delhi
14	Civil	1	John	Delhi
14		5	Kristen	Noida
15	Chemistry	3	David	Mumbai

DATA DICTIONARY STORAGE

In general, such “data about data” is referred to as metadata.Relational schemas and other metadata about relations are stored in a structure called the data dictionary or system catalog. Among the types of in formation that the system must store are these:

- Names of the relations
- Namesoftheattributes of each relation
- Domainsandlengths of attributes
- Namesofviewsdefinedonthe database, and definitions of those views
- Integrity constraints (for example, key constraints)

In addition, many systems keep the following data on users of the system:

- Namesofauthorized users
- Authorization and accounting information about users
- Passwords or other information used to authenticate users

Further,thedatabasemaystorestatisticalanddescriptivedataabouttherelations, such as:

- Numberoftuplesin each relation
- Methodofstorage for each relation (for example, clustered or non-clustered)

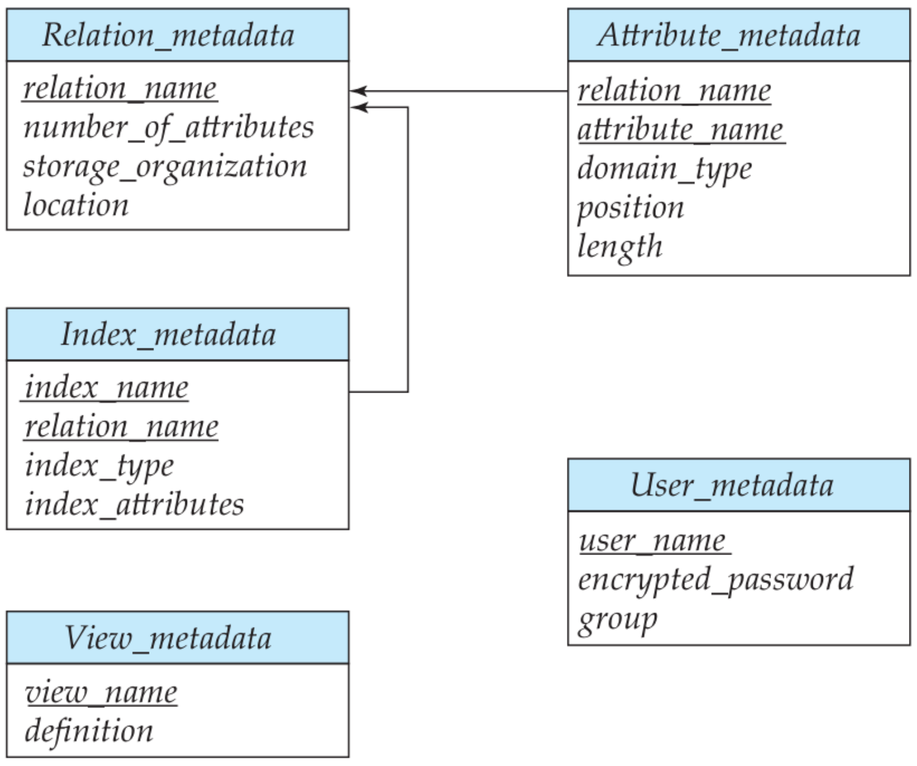
The data dictionary may also note the storage organization (sequential, hash, or heap) of relations, and the location where each relation is stored:

- Ifrelations are stored in operating system files, the dictionary would note the names of the file (or files) containing each relation
- Ifthedatabase stores all relations in a single file, the dictionary may note the blocks containing records of each relation in a data structure such as a linked list

It stores information about each index on each of the relations like:

- Nameoftheindex
- Nameoftherelation being indexed
- Attributes on which the index is defined

- Typeofindexformed



Relational schema representing system metadata.

Database Buffer: The buffer is that part of main memory available for storage of copies of disk blocks.The subsystem responsible for the allocation of buffer space is called the buffer manager.

Buffer Manager: Programs call on the buffer manager when they need a block from disk

- The requesting program is given the address of the block in main memory, if it is already present in the buffer
- If the block is not in the buffer, the buffer manager allocates space in the buffer for the block, replacing (throwing out) some other block, if required, to make space for the new block
- The block that is thrown out is written back to disk only if it was modified since the most recent time that it was written to/fetched from the disk
- Oncespace is allocated in the buffer, the buffer manager reads in the block from the disk to the buffer, and passes the address of the block in main memory to the requester

To serve the database system well, the buffer manager must use techniques more sophisticated than typical virtual-memory management schemes:

- ***Buffer replacement strategy***:When there is no room left in the buffer, a block must be removed from the buffer before a new one can be read in.
  - ❖ Most operating systems replace the block least recently used (LRU).
  - ❖ Toss-immediate strategy– frees the space occupied by a block as soon as the final tuple of that block has been processed.

- ***Pinned blocks*:**A block that is not allowed to be written back to disk is said to be pinned.
- **Forced out put of blocks:** There are situations in which it is necessary to write back the block to disk, even though the buffer space that it occupies is not needed. This write is called the *forced output* of a block.

INDEXING AND HASHING

An index is a data structure that organizes data records on the disk to make the retrieval of data efficient.

- The search key for an index is collection of one or more fields of records using which we can efficiently retrieve the data that satisfy the search conditions.
- The indexes are required to speed up the search operations on file of records.

There are two types of indices -

1. Ordered Indices: This type of indexing is based on sorted ordering values.
2. Hash Indices: This type of indexing is based on uniform distribution of values across range of buckets. The bucket to which a value is assigned is determined by a function, called a hash function.

There are several techniques of for using indexing and hashing. These techniques are evaluated based on following factors –

Access types: The types of access that are supported efficiently. Access types can include finding records with a specified attribute value and finding records whose attribute values fall in a specified range.

Access time: The time it takes to find a particular data item, or set of items, using the technique in question.

Insertion time: The time it takes to insert a data item. This value includes the time it takes to find the correct place to insert the new data item, as well as the time it takes to update the index structure.

Deletion time: The time it takes to delete a data item. This value includes the time it takes to find the item to be deleted, as well as the time it takes to update the index structure.

Space overhead: The additional space occupied by an index structure. Provided that the amount of additional space is moderate, it is usually worthwhile to sacrifice the space to achieve improved performance.

❖ An attribute or set of attributes used to look up records in a file is called a search key.

<u>Ordered Indices</u>

Primary index: An index on a set of fields that includes the primary key is called a primary index. The primary index file should be always in sorted order.

- The primary indexing is always done when the data file is arranged in sorted order and primary indexing contains the primary key as its search key.
- Consider following scenario in which the primary index consists of few entries as compared to actual data file.

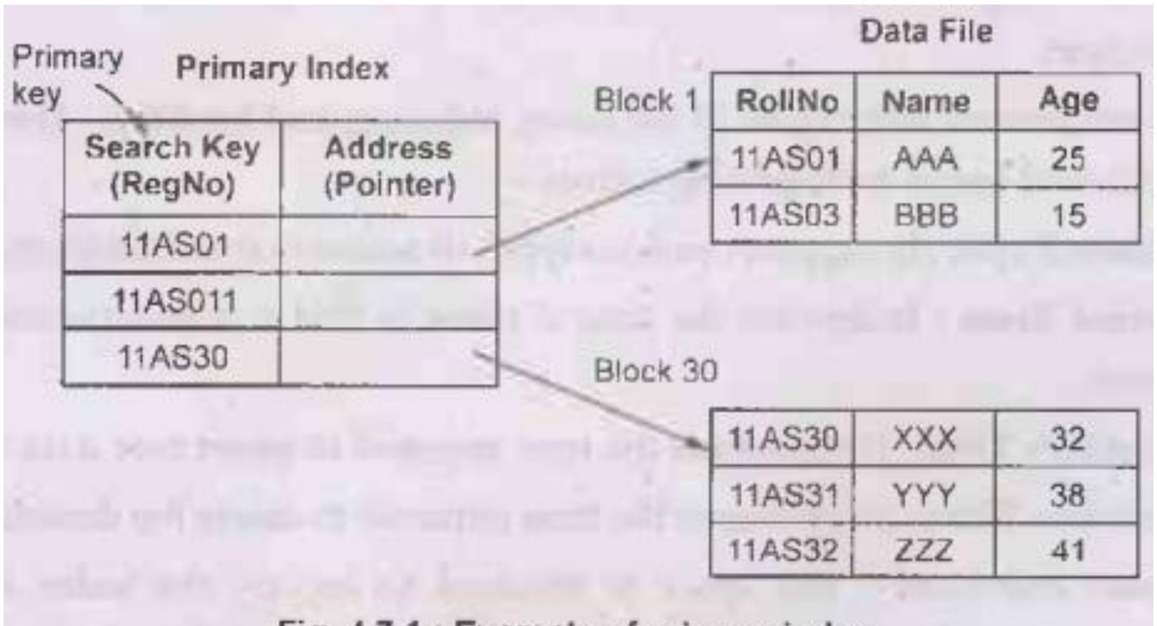


Fig. 4.7.1 : Example of primary index

- Once if you are able to locate the first entry of the record containing block, other entries are stored continuously.
- We can apply binary search technique. Suppose there are n = 300 blocks in a main data file then the number of accesses required to search the data file will be $\log_2 n + 1 = (\log_2 300) + 1 \approx 9$
- If we use primary index file which contains at the most n = 3 blocks then using binary search technique, the number of accesses required to search using the primary index file will be $\log_2 n+1= (\log_2 3)+1=3$
- This shows that using primary index the access time can be reduced to great extent.

Clustered index:

- Group of two or more columns together to get the unique values and create index out of them. This method is known as clustering index.
- When a file is organized so that the ordering of data records is the same as the ordering of data entries in some index then say that index is clustered, otherwise it is an unclustered index.

- The data file need to be in sorted order.
- Basically, records with similar characteristics are grouped together and indexes are created for these groups.
- For example, students studying in each semester are grouped together. i.e.; 1st semester students, 2nd semester students, 3rd semester students etc. are grouped.

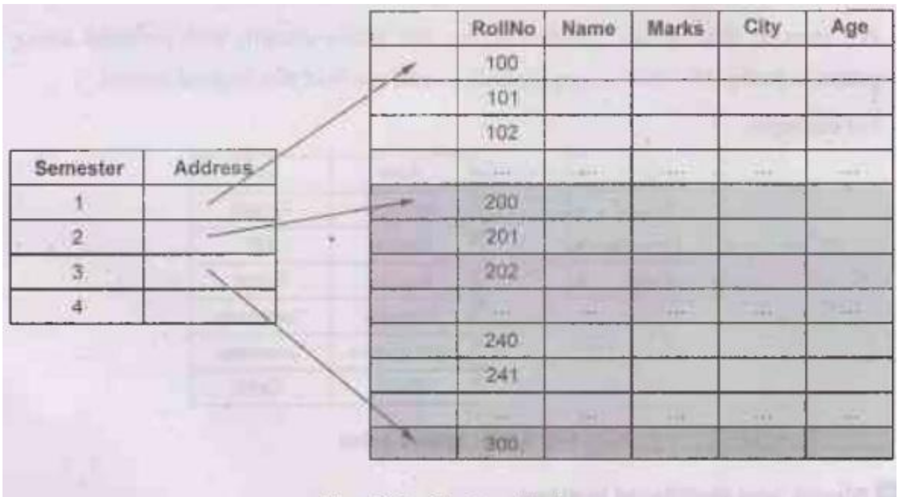


Fig. 4.7.2 : Clustered index

<u>Dense and Sparse Indices</u>: There are two types of ordered indices.

1. Dense index
2. Sparse index

Dense index:

- An index record appears for every search key value in file.
- This record contains search key value and a pointer to the actual record.

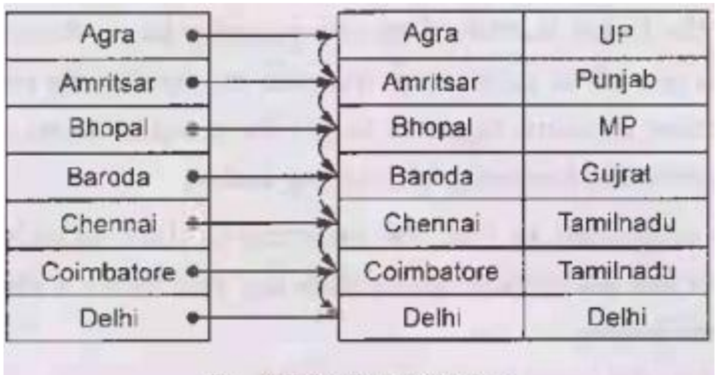


Fig. 4.7.3 : Dense index

Sparse index:

- Index records are created only for some of the records.
- 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.
- We start at that record pointed to by the index record, and proceed along the pointers in the file (that is, sequentially) until we find the desired record.

For example -

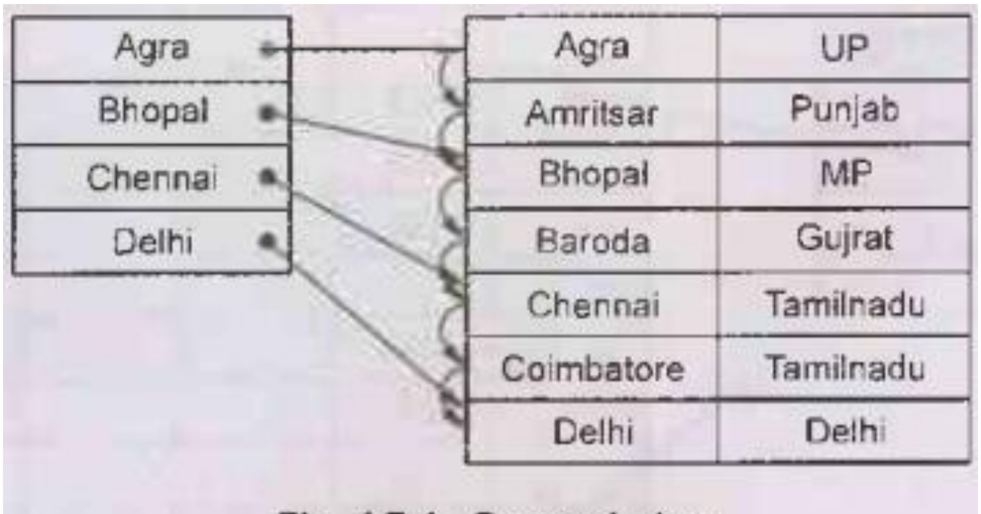
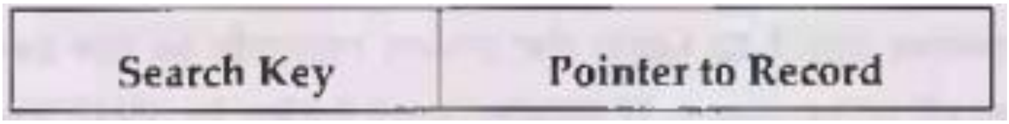


Fig. 4.7.4 : Sparse index

SINGLE LEVEL ORDERED INDEXES

- A single-level index is an auxiliary file that makes it more efficient to search for a record in the data file.
- The index is usually specified on one field of the file (although it could be specified on several fields).
- Each index can be in the following form.



Search Key	Pointer to Record

The index file usually occupies considerably less disk blocks than the data file because its entries are much smaller.

- A binary search on the index yields a pointer to the file record.
- The types of single level indexing can be primary indexing, clustering index or secondary indexing.

Example: Following Figure represents the single level indexing –

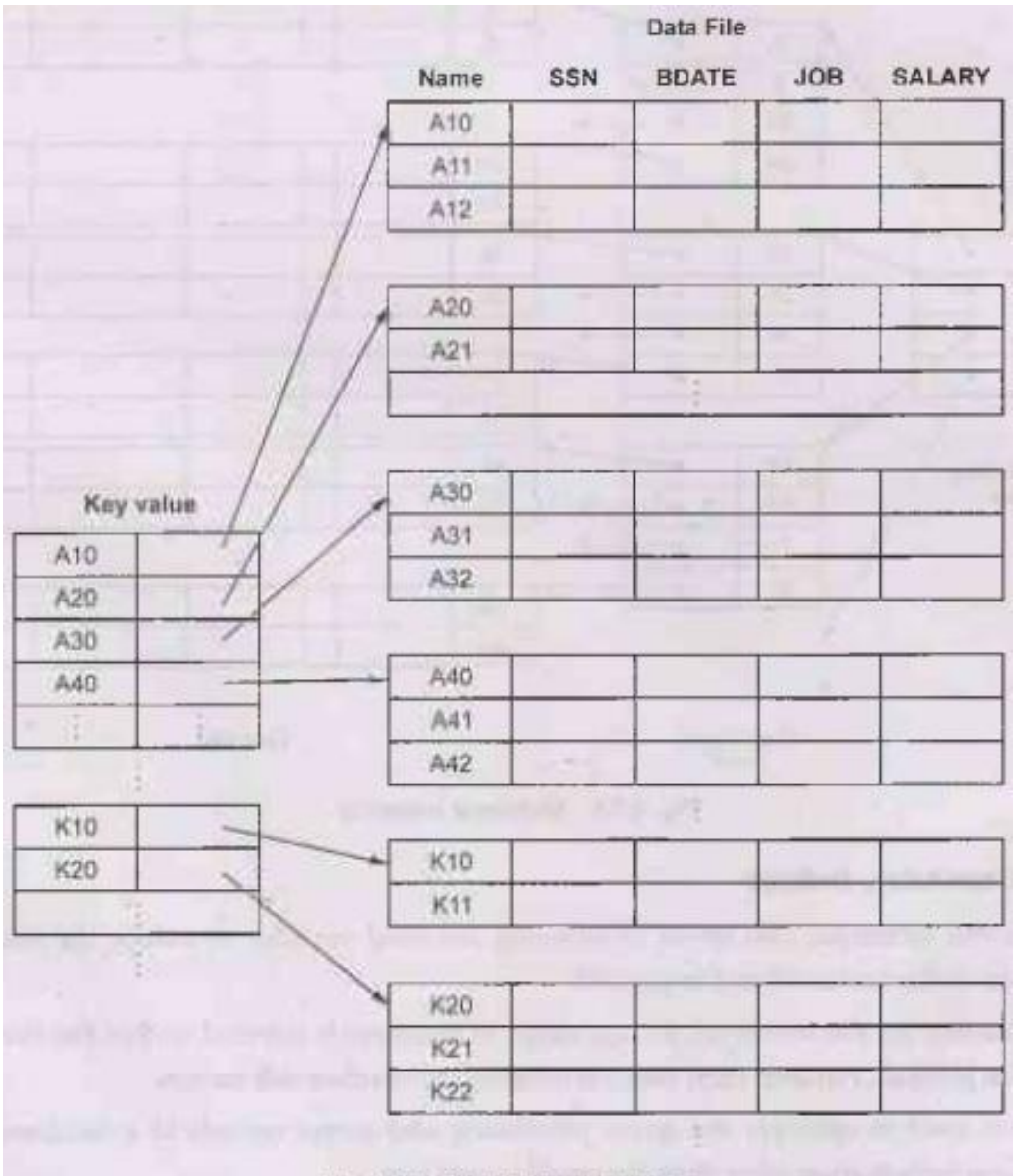


Fig. 4.7.5 : Single level indexing

MULTI LEVEL INDEXES

- There is an immense need to keep the index records in the main memory so as to speed up the search operations. If single-level index is used, then a large size index cannot be kept in memory which leads to multiple disk accesses.
- Multi-level Index helps in breaking down the index into several smaller indices in order to make the outermost level so small that it can be saved in a single disk block, which can easily be accommodated anywhere in the main memory.

The multilevel indexing can be represented by following Figure.

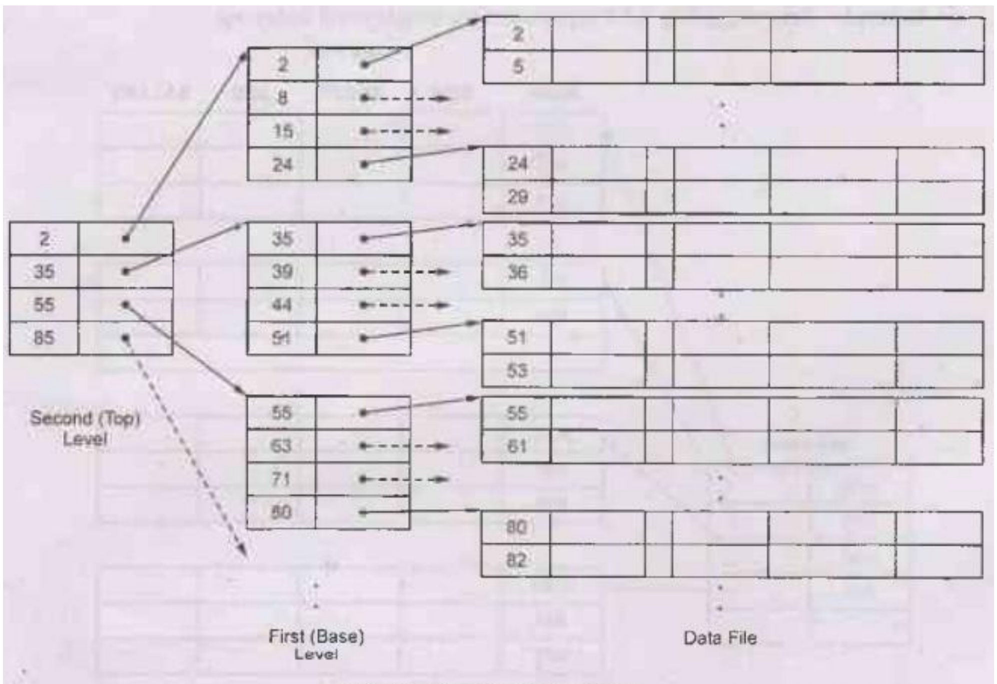


Fig. 4.7.6 : Multilevel indexing

Secondary Indices

- In this technique two levels of indexing are used in order to reduce the mapping size of the first level and in general.
- Initially, for the first level, a large range of numbers is selected so that the mapping size is small. Further, each range is divided into further sub ranges.
- It is used to optimize the query. processing and access records in a database with some information other than the usual search key.

For example -

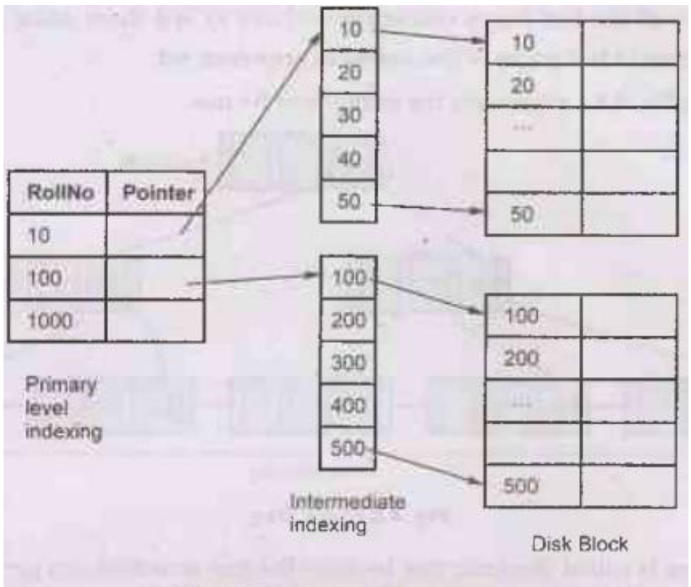
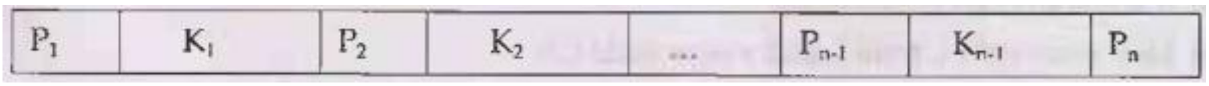


Fig. 4.7.7 : Secondary Indexing

B+ Tree Index Files:

- The B+ tree is similar to binary search tree. It is a balanced tree in which the internal nodes direct the search.
- The leaf nodes of B+ trees contain the data entries.

Structure of B+ Tree:The typical node structure of B+ node is as follows



P_1	K_1	P_2	K_2	...	P_{n-1}	K_{n-1}	P_n

- It contains up to n – 1 search-key values k_1, k_2, ……, k_{n-1} and n pointersP_1, P_2,..., P_n
- The search-key values within a node are kept in sorted order; thus, if $i < j$, then $K_i<K_j$.
- To retrieve all the leaf pages efficiently we have to link them using page pointers. The sequence of leaf pages is also called as sequence set.
- Following Figure represents the example of B+ tree.

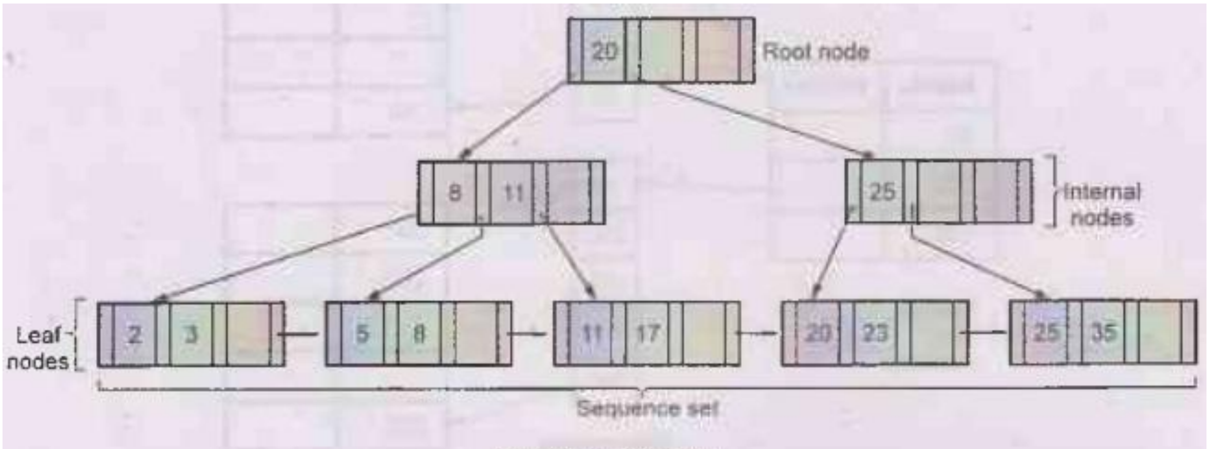


Fig. 4.8.1 : B+ Tree

- The B+ tree is called dynamic tree because the tree structure can grow on insertion of records and shrink on deletion of records.

Characteristics of B+ Tree:Following are the characteristics of B+ tree.

1. The B+ tree is a balanced tree and the operations insertions and deletion keeps the tree balanced.
2. A minimum occupancy of 50 percent is guaranteed for each node except the root.
3. Searching for a record requires just traversal from the root to appropriate leaf.

Insertion Operation

Algorithm for insertion :

Step 1: Find correct leaf L.

Step 2: Put data entry onto L.

i. If L has enough space, done!

ii. Else, must split L (into L and a new node L2)

- Allocate new node
- Redistribute entries evenly
- Copy up middle key.
- Insert index entry pointing to L2 into parent of L.

Step 3: This can happen recursively

- To split index node, redistribute entries evenly, but push up middle key. (Contrast with leaf splits.)

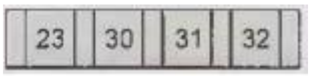
Step 4: Splits "grow" tree; root split increases height.

- Tree growth: gets wider or one level taller at top.

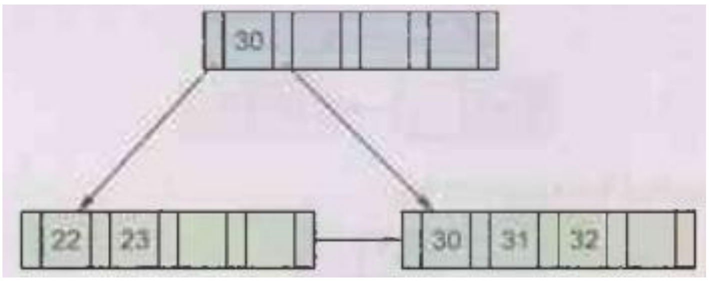
Example: Construct B+ tree for following data. 30,31,23,32,22,28,24,29, where number of pointers that fit in one node are 5.

Solution: In B+ tree each node is allowed to have the number of pointers to be 5. That means at the most 4 key values are allowed in each node.

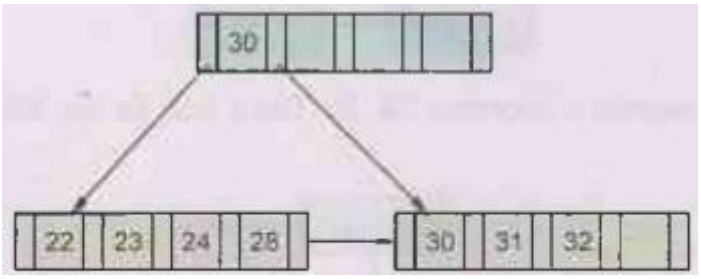
Step 1: Insert 30,31,23,32. We insert the key values in ascending order.



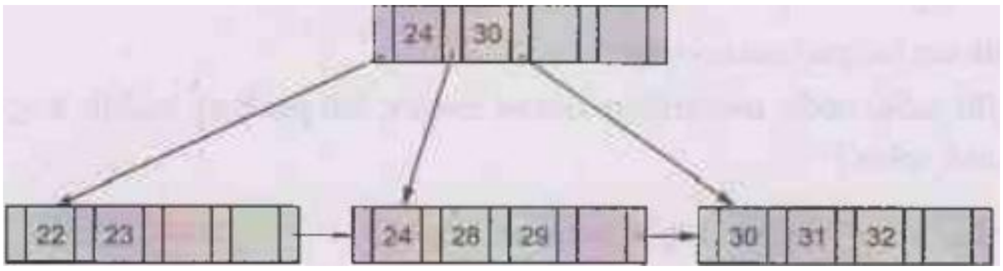
Step 2: Now if we insert 22, the sequence will be 22, 23, 30, 31, 32. The middle key 30, will go up.



Step 3: Insert 28,24. The insertion is in ascending order.



Step 4: Insert 29. The sequence becomes 22, 23, 24, 28, 29. The middle key 24 will go up. Thus we get the B+ tree.

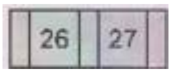


Example: Construct B+ tree to insert the following (order of the tree is 3) 26,27,28,3,4,7,9,46,48,51,2,6

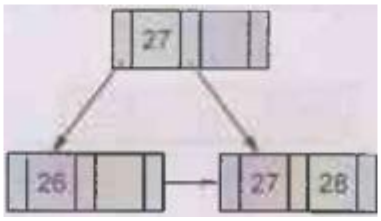
Solution:

Order means maximum number of children allowed by each node. Hence order 3 means at the most 2 key values are allowed in each node.

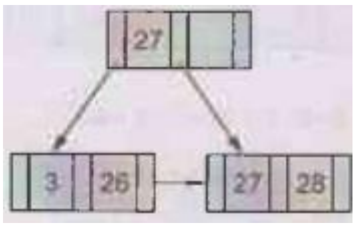
Step 1: Insert 26, 27 in ascending order



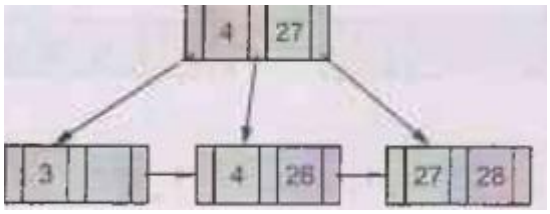
Step 2: Now insert 28. The sequence becomes 26,27,28. As the capacity of the node is full, 27 will go up. The B+ tree will be,



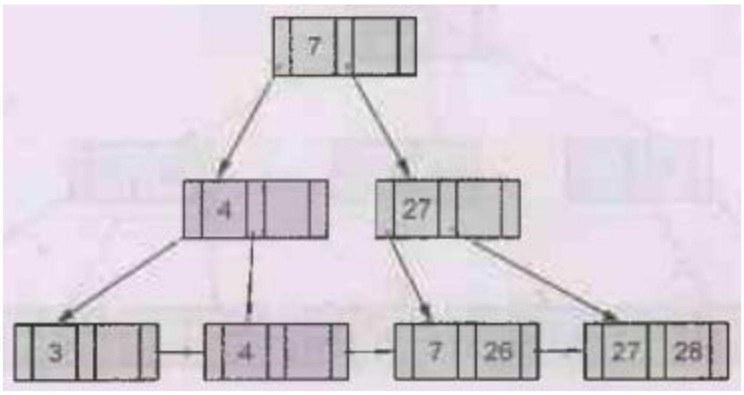
Step 3: Insert 3. The partial B+ Tree will be,



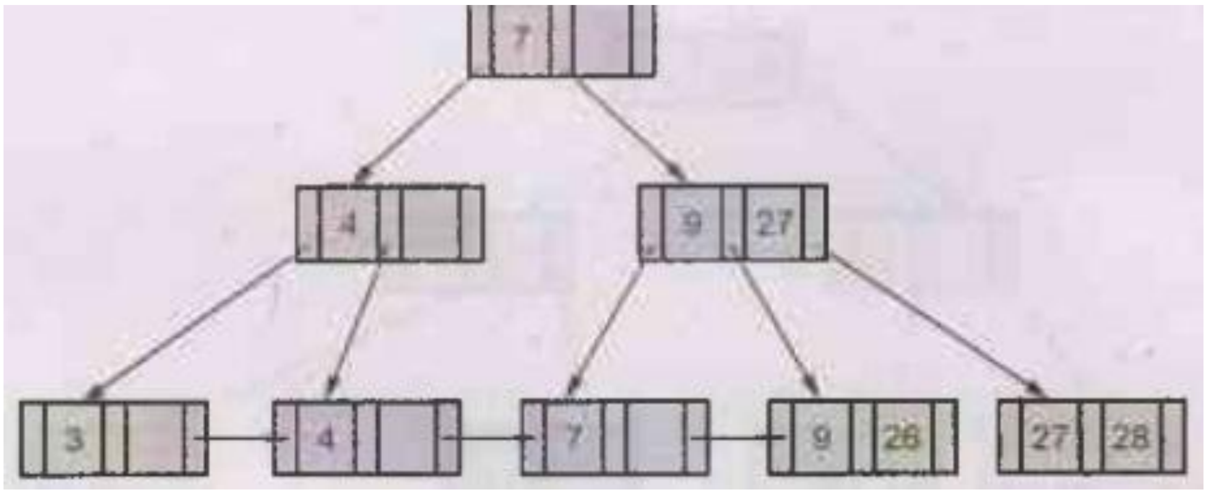
Step 4: Insert 4. The sequence becomes 3,4, 26. The 4 will go up. The partial B+ tree will be –



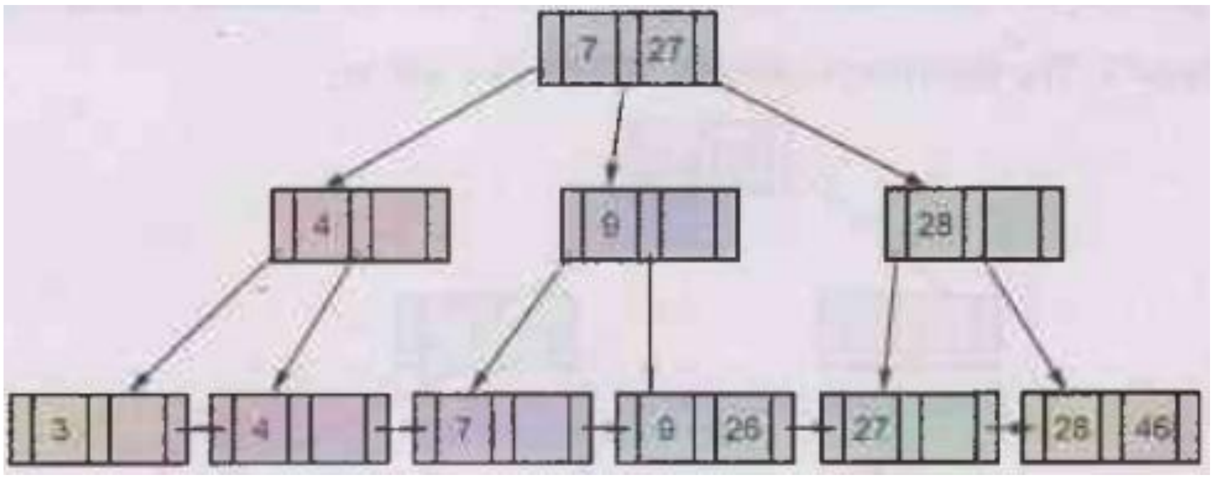
Step 5: Insert 7. The sequence becomes 4,7,26. The 7 will go up. Again from 4,7,27. the 7 will go up. The partial B+ Tree will be,



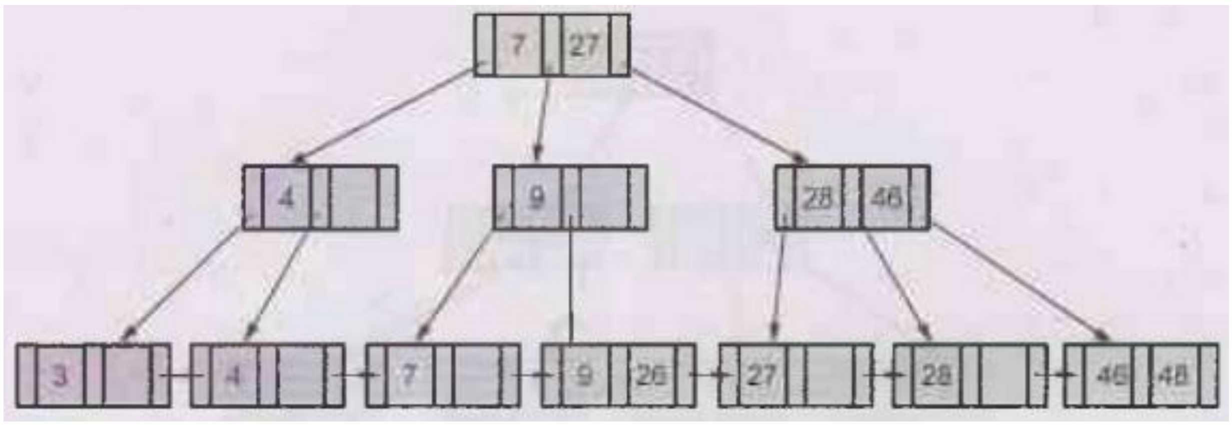
Step 6: Insert 9. By inserting 7,9, 26 will be the sequence. The 9 will go up. The partial B+ tree will be,



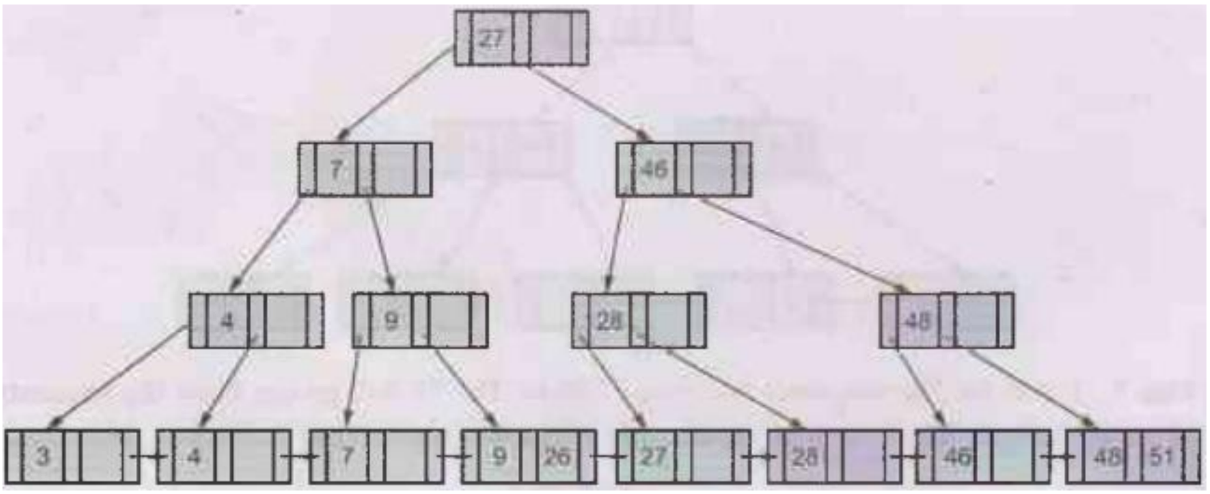
Step 7: Insert 46. The sequence becomes 27,28,46. The 28 will go up. Now the sequence becomes 9, 27, 28. The 27 will go up and join 7. The B+ Tree will be,



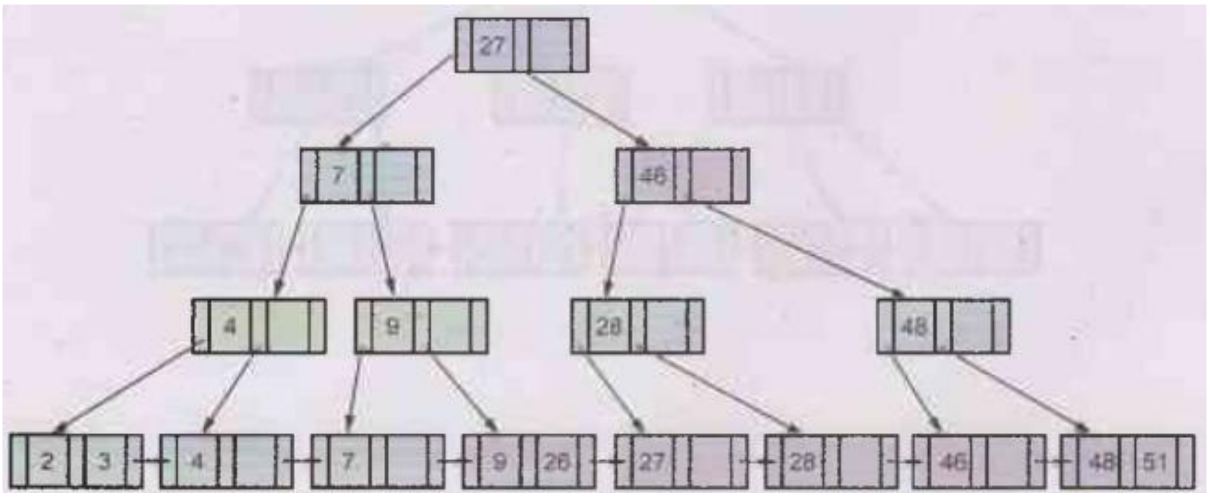
Step 8: Insert 48. The sequence becomes 28,46,48. The 46 will go up. The B+ Tree will become,



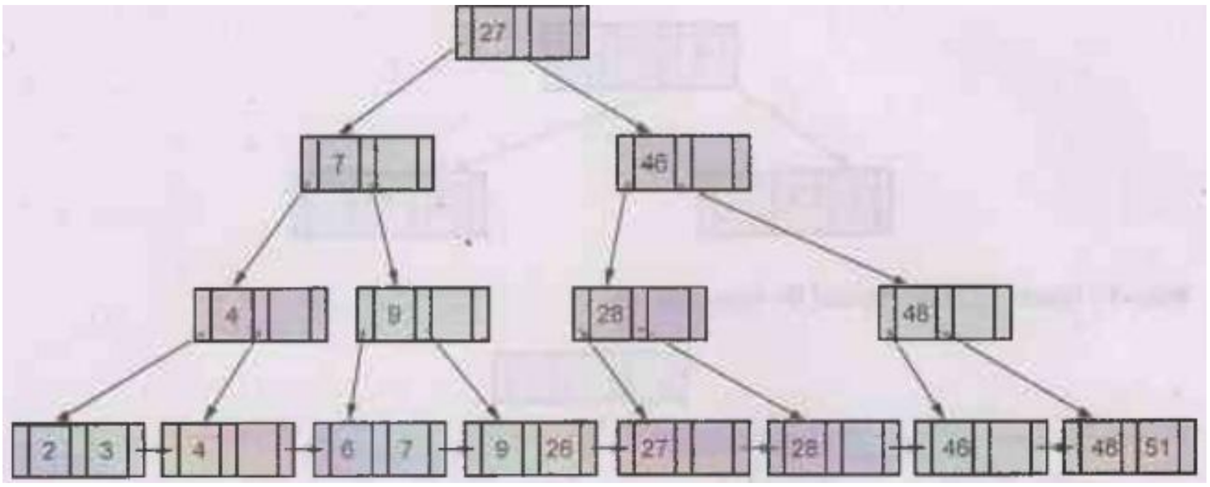
Step 9: Insert 51. The sequence becomes 46,48,51. The 48 will go up. Then the sequence becomes 28, 46, 48. Again the 46 will go up. Now the sequence becomes 7,27, 46. Now the 27 will go up. Thus the B+ tree will be



Step 10: Insert 2. The insertion is simple. The B+ tree will be,



Step 11: Insert 6. The insertion can be made in a vacant node of 7(the leaf node). The final B+ tree will be,



Deletion Operation

Algorithm for deletion:

Step 1: Start at root, find leaf L with entry, if it exists.

Step 2: Remove the entry.

i. If L is at least half-full, done!

ii. If L has only d-1 entries,

- Try to re-distribute, borrowing keys from sibling.
  (adjacent node with same parent as L).
- If redistribution fails, merge L and sibling.

Step 3: If merge occurred, must delete entry (pointing to L or sibling) from parent of L.

Step 4: Merge could propagate to root, decreasing height.

Example: *Construct B+ Tree for the following set of key values*

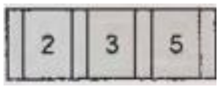
2,3,5,7,11,17,19,23,29,31

Assume that the tree is initially empty and values are added in ascending order.
Construct B+ tree for the cases where the number of pointers that fit one node is four.
After creation of B+ tree perform following series of operations:

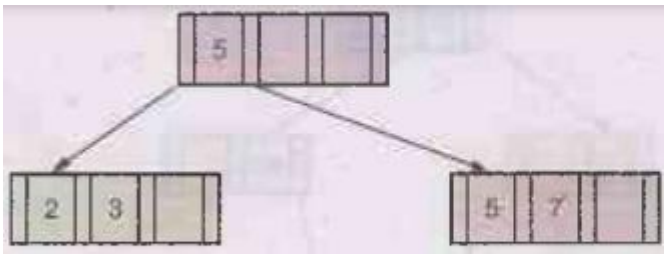
(a) Insert 9. (b) Insert 10. (c) Insert 8. (d) Delete 23. (e) Delete 19.

Solution: The number of pointers fitting in one node is four. That means each node contains at the most three key values.

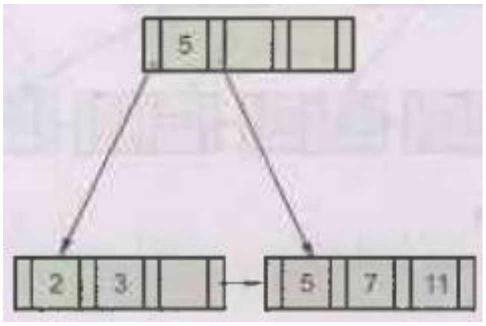
Step 1: Insert 2, 3, 5.



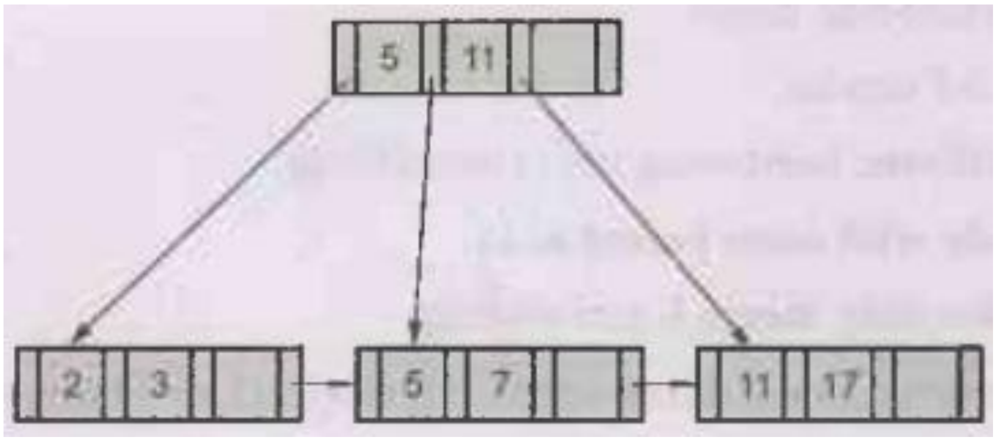
Step 2: If we insert 7, the sequence becomes 2, 3, 5, 7. Since each node can accommodate at the most three keys, the 5 will go up, from the sequence 2, 3, 5, 7.



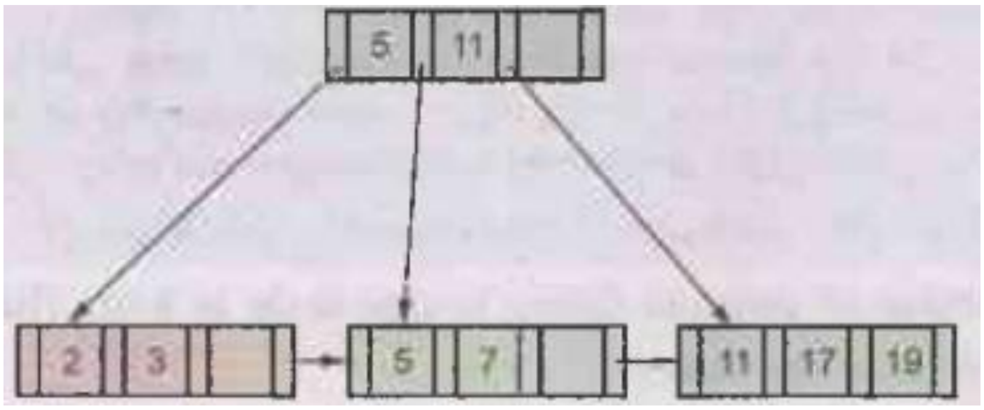
Step 3: Insert 11. The partial B+ tree will be,



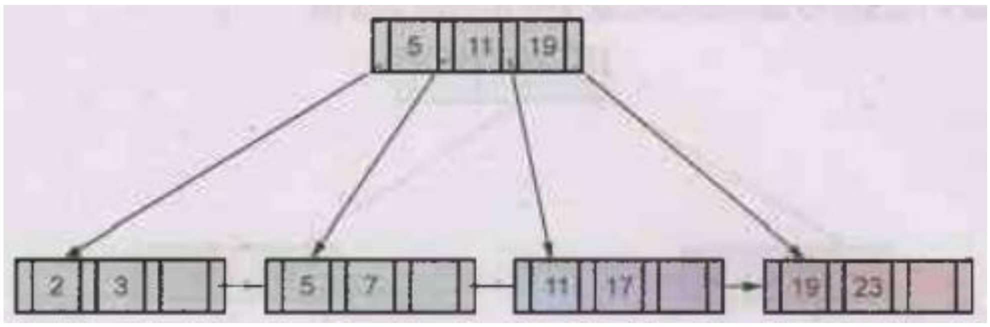
Step 4: Insert 17. The sequence becomes 5,7, 11,17. The element 11 will go up. Then the partial B+ tree becomes,



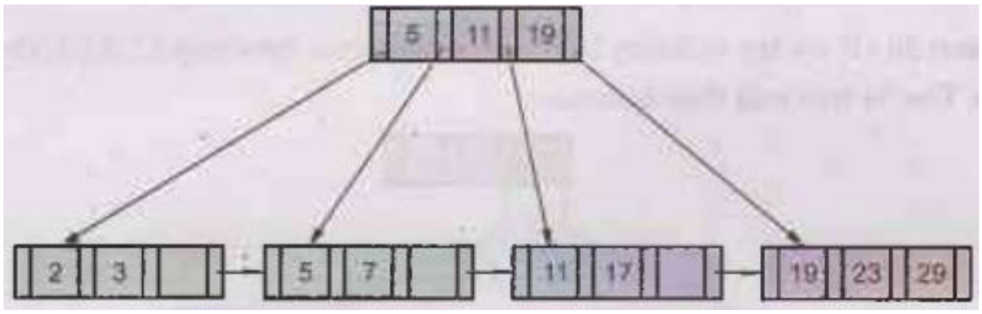
Step 5: Insert 19.



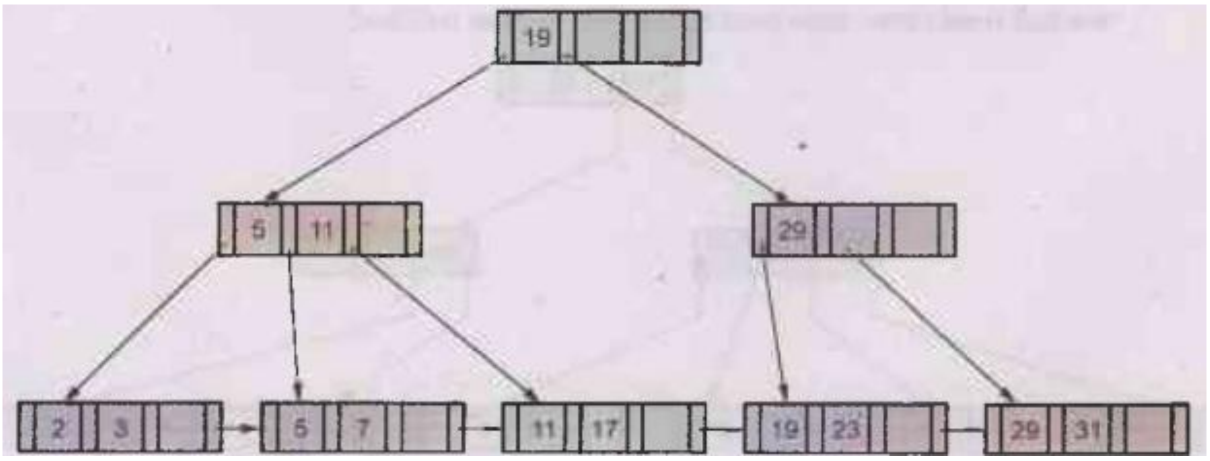
Step 6: Insert 23. The sequence becomes 11,17,19,23. The 19 will go up.



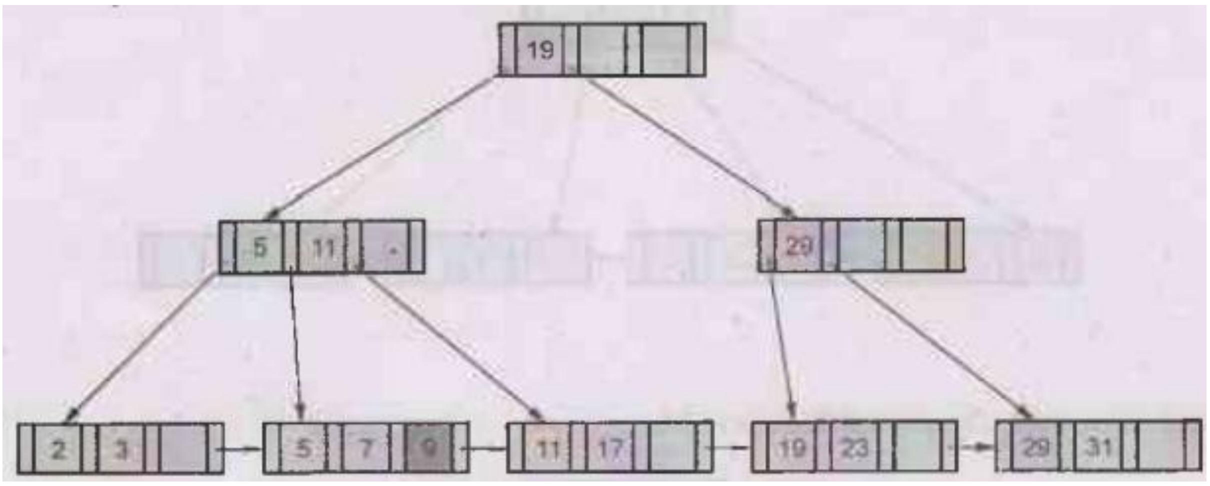
Step 7: Insert 29. The partial B+ tree will be,



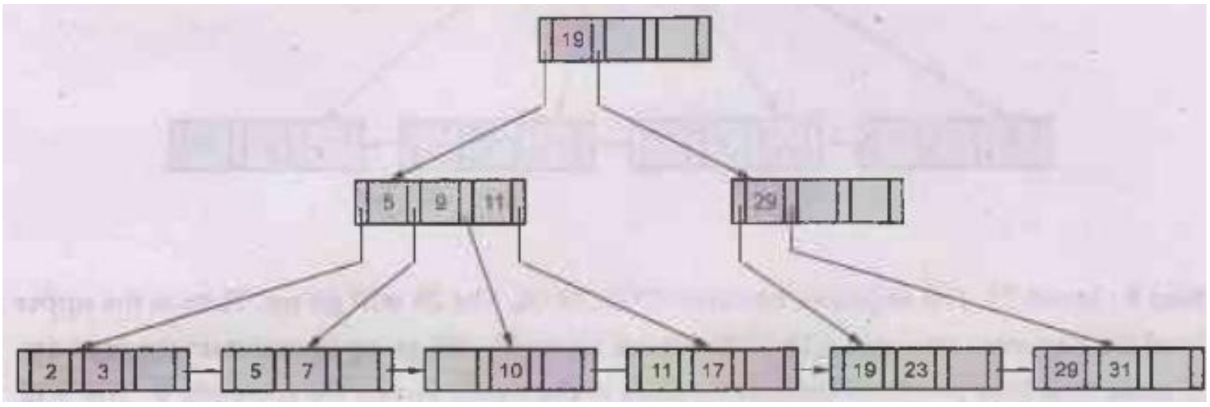
Step 8: Insert 31. The sequence becomes 19,23,29, 31. The 29 will go up. Then at the upper level the sequence becomes 5,11,19,29. Hence again 19 will go up to maintain the capacity of node (it is four pointers three key values at the most). Hence the complete B+ tree will be,



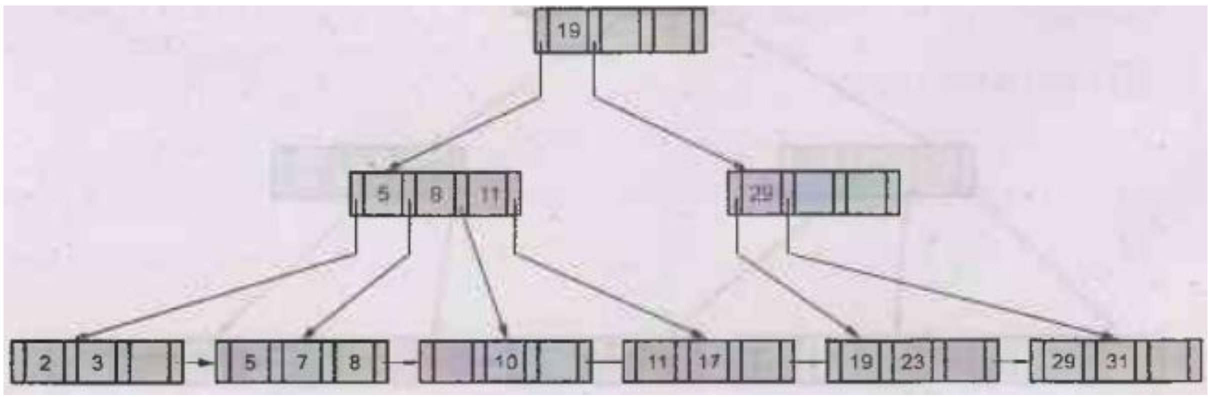
(a) Insertion of 9: It is very simple operation as the node containing 5,7 has one space vacant to accommodate. The B+ tree will be,



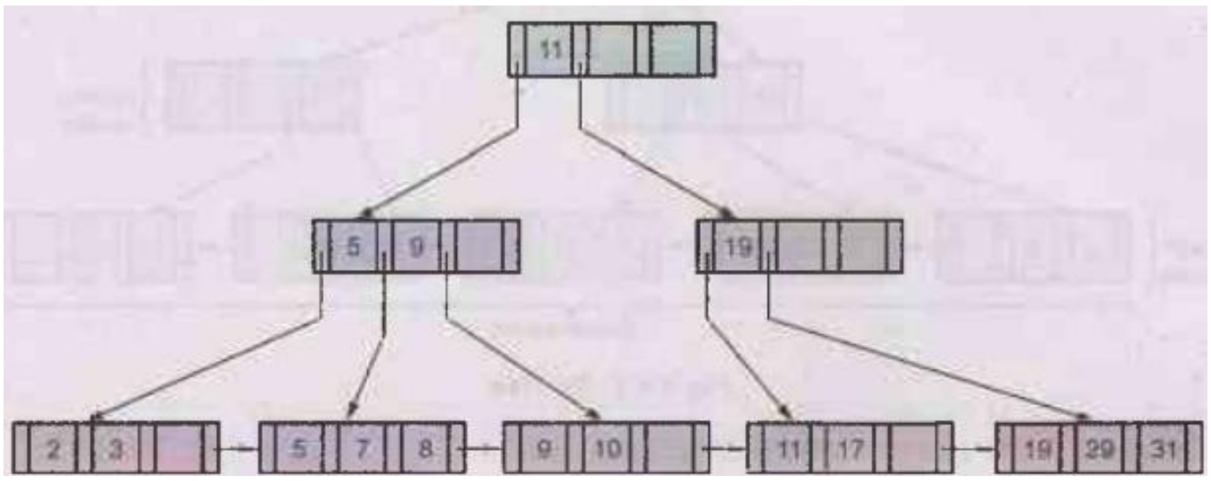
(b) Insert 10: If we try to insert 10 then the sequence becomes 5,7,9,10. The 9 will go up. The B+ tree will then become –



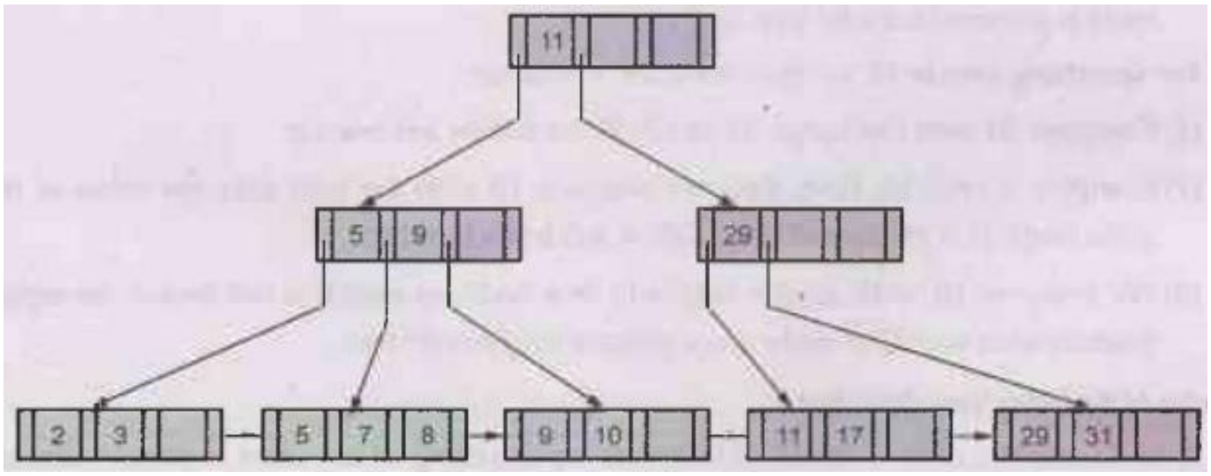
(c) Insert 8: Again insertion of 8 is simple. We have a vacant space at node 5,7. So we just insert the value over there. The B+ tree will be-



(d) Delete 23: Just remove the key entry of 23 from the node 19,23. Then merge the sibling node to form a node 19,29,31. Get down the entry of 11 to the leaf node. Attach the node of 11,17 as a left child of 19.



(e) Delete 19: Just delete the entry of 19 from the node 19,29,31. Delete the internal node key 19. Copy the 29 up as an internal node as it is an inorder successor node.



Search Operation

1. Perform a binary search on the records in the current node.
2. If a record with the search key is found, then return that record.
3. If the current node is a leaf node and the key is not found, then report an unsuccessful search.
4. Otherwise, follow the proper branch and repeat the process.

For example-

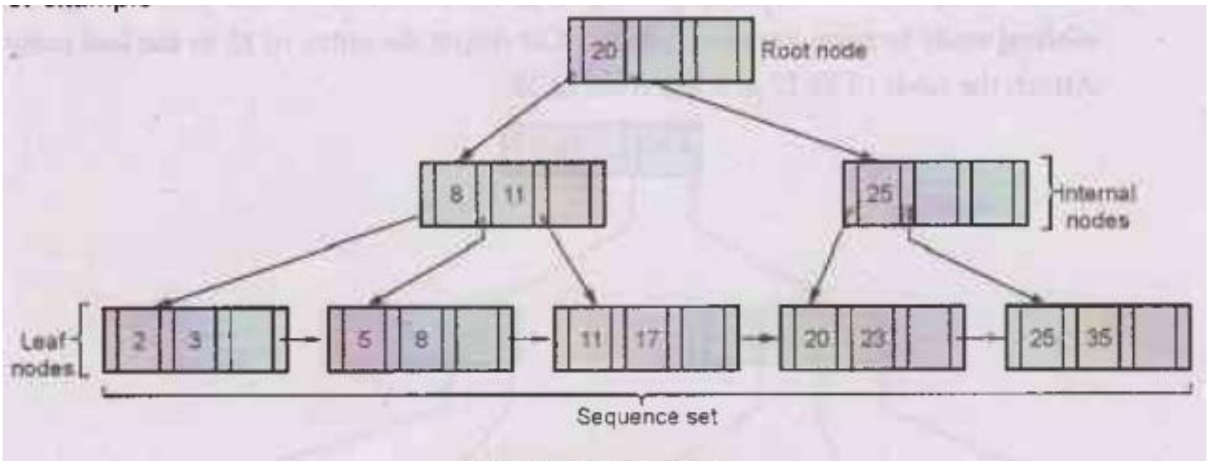


Fig. 4.8.2 : B+ Tree

Consider the B+ tree as shown in above Figure,

For searching a node 25, we start from the root node -

1) Compare 20 with key value 25. As 25>20, move on to right branch.
2) Compare 25 with key value 25. As the match is found we declare, that the given node is present in the B+ tree.

For searching a node 10, we start form the root node -

1) Compare 20 with key value 10, as 10<20, we follow left branch
2) Compare 8 with 10, 10>8, then we compare 10 with the next adjacent value of the same node. It is 11, as 10<11, we follow left branch of 11.
3) We compare 10, with all the values in that node, as match is not found we report unsuccessful search or node is not present in given B+ tree.

<u>**Merits of B+ Index Tree Structure**</u>

1. In B+ tree the data is stored in leaf node so searching of any data requires scanning only of leaf node alone.
2. Data is ordered in linked list.
3. Any record can be fetched in equal number of disk accesses.
4. Range queries can be performed easily as leaves are linked up.
5. Height of the tree is less as only keys are used for indexing.
6. Supports both random and sequential access.

Demerits of B+ Index Tree Structure

1. Extra insertion of non-leaf nodes.
2. There is space overhead.

B Tree Index Files

- 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.
- B-tree is a specialized multiway tree used to store the records in a disk.
- There are number of subtrees to each node. So that the height of the tree is relatively small. So that only small number of nodes must be read from disk to retrieve an item. The goal of B-trees is to get fast access of the data.
- A B-tree allows search-key values to appear only once (if they are unique), unlike a B+-tree, where a value may appear in a non-leaf node, in addition to appearing in a leaf node.

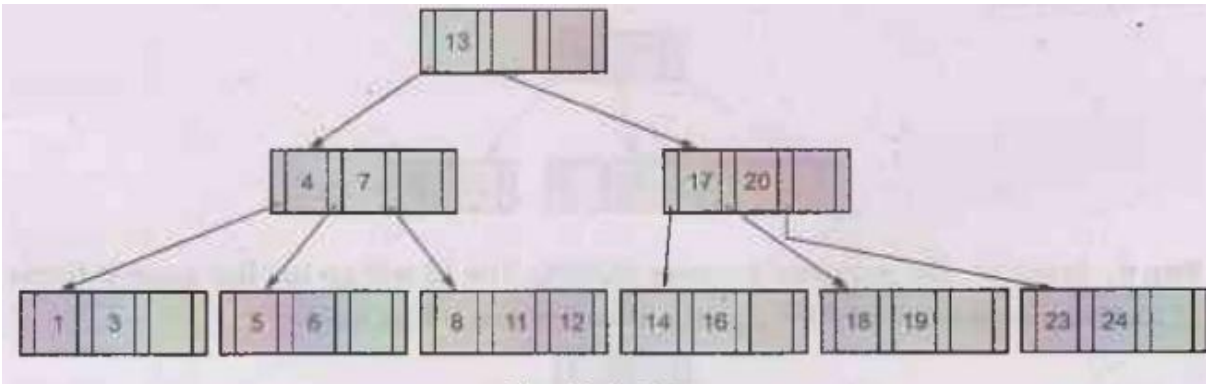
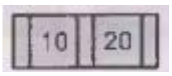


Fig. 4.9.1 : B-Tree

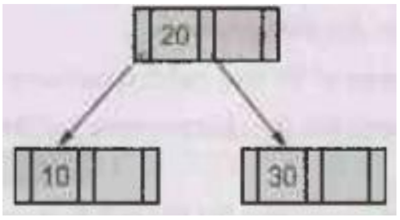
Example: Create B tree of order 3 for following data: **20,10,30,15,12,40,50**.

Solution: The B tree of order 3 means at the most two key values are allowed in each node of B-Tree.

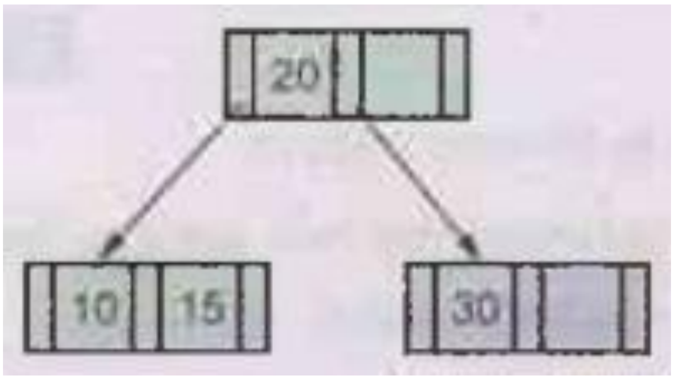
Step 1: Insert 20,10 in ascending order



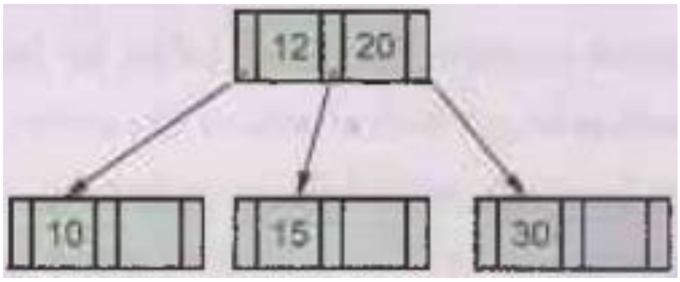
Step 2: If we insert the value 30. The sequence becomes 10,20,30. As only two key values are allowed in each node (being order 3), the 20 will go up.



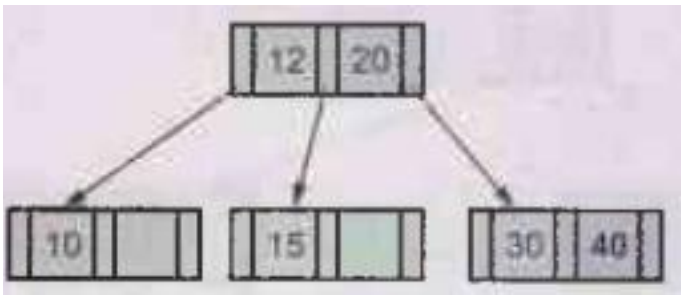
Step 3: Now insert 15.



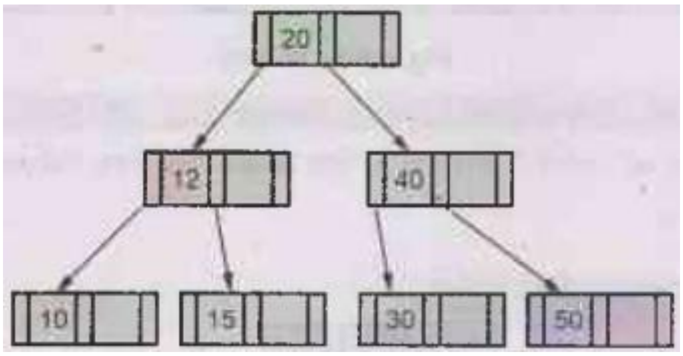
Step 4: Insert 12. The sequence becomes 10, 12, 15. The middle element 12 will go up.



Step 5: Insert 40

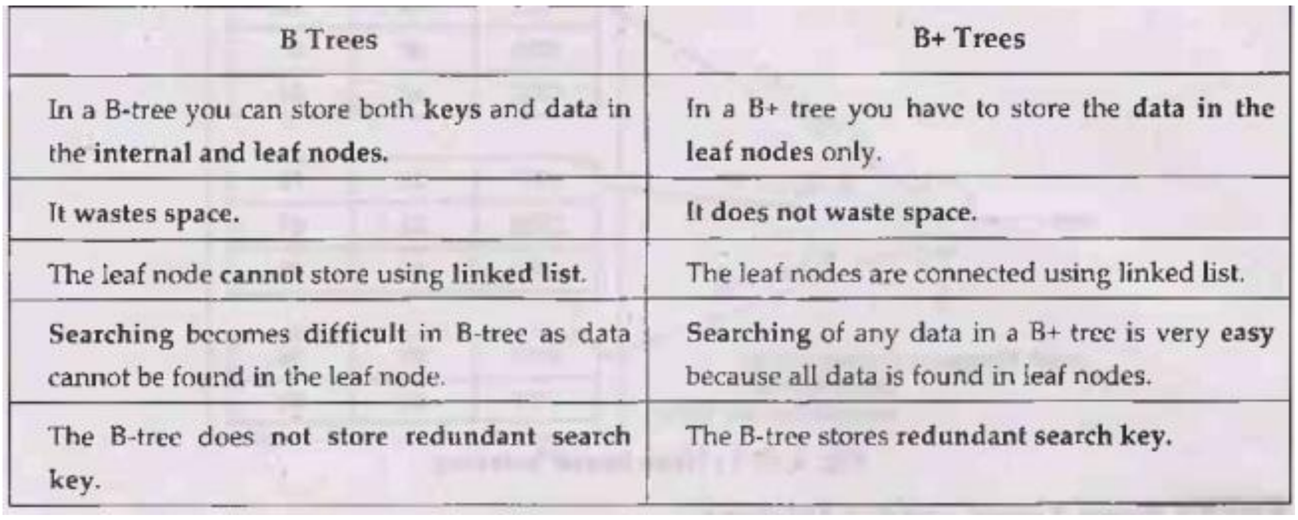


Step 6: Insert 50. The sequence becomes 30,40,50. The 40 will go up. But again it forms 12,20,40 sequence and then 20 will go up. Thus the final B Tree will be,



This is the final B-Tree.

Differences between B Tree and B+ Tree

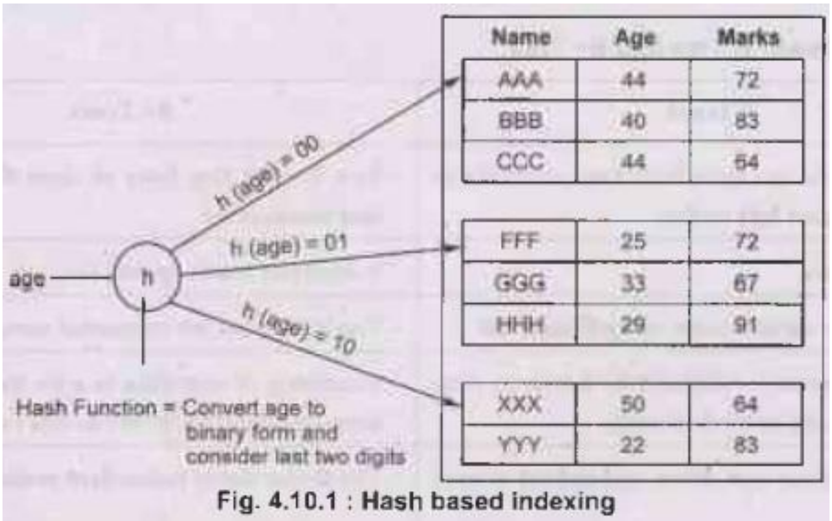


B Trees	B+ Trees
In a B-tree you can store both **keys and data** in the **internal and leaf nodes.**	In a B+ tree you have to store the **data in the leaf nodes** only.
It **wastes space.**	It **does not waste space.**
The leaf node **cannot** store using **linked list.**	The leaf nodes are connected using linked list.
Searching becomes **difficult** in B-tree as data cannot be found in the leaf node.	**Searching** of any data in a B+ tree is very **easy** because all data is found in leaf nodes.
The B-tree **does not** store **redundant search key.**	The B-tree stores **redundant search key.**

Concept of Hashing

- Hash file organization method is the one where data is stored at the data blocks whose address is generated by using hash function.
- The memory location where these records are stored is called as data block or bucket. This bucket is capable of storing one or more records.
- The hash function can use any of the column value to generate the address. Most of the time, hash function uses primary key to generate the hash index - address of the or data block.
- Hash function can be simple mathematical function to any complex mathematical function.

For example - Following figure represents the records of student can be searched using hash based indexing. In this example the hash function is based on the age field of the record. Here the index is made up of data entry k* which is actual data record. For a hash function the age is converted to binary number format and the last two digits are considered to locate the student record.


age
h
h (age) = 00
h (age) = 01
h (age) = 10
Hash Function = Convert age to binary form and consider last two digits

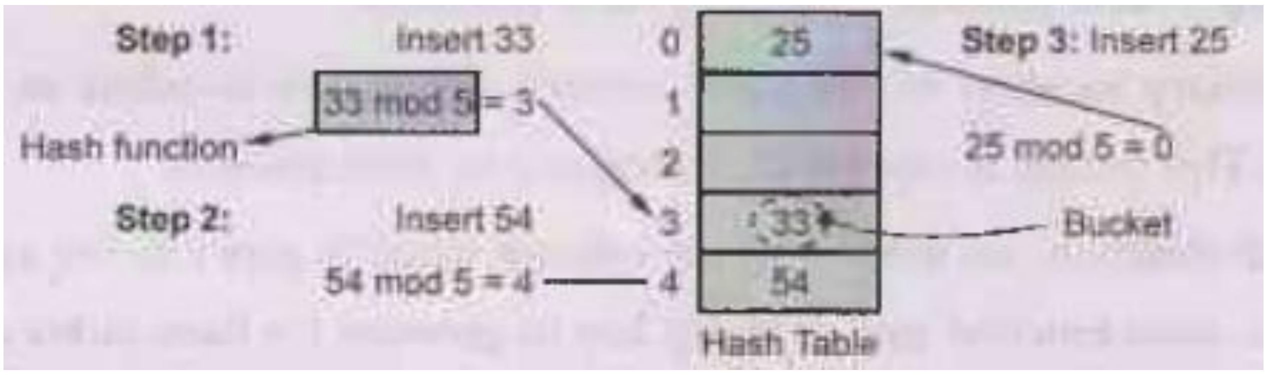
Name	Age	Marks
AAA	44	72
BBB	40	83
CCC	44	54
FFF	25	72
GGG	33	67
HHH	29	91
XXX	50	64
YYY	22	83

Fig. 4.10.1 : Hash based indexing

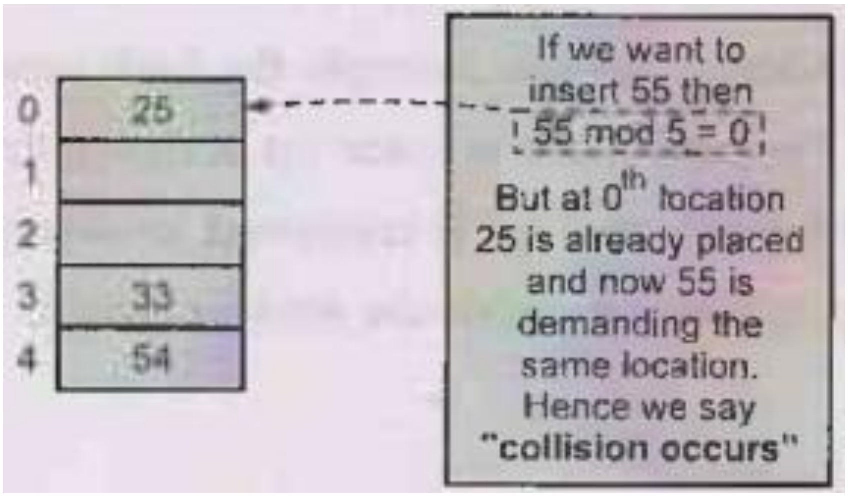
Basic Terms used in Hashing

1. **Hash Table:** Hash table is a data structure used for storing and retrieving data quickly. Every entry in the hash table is made using Hash function.
2. **Hash function:**
   - Hash function is a function used to place data in hash table.
   - Similarly hash function is used to retrieve data from hash table.
   - Thus the use of hash function is to implement hash table.

For example: Consider hash function as key



3. **Bucket:** The hash function H(key) is used to map several dictionary entries in the hash table. Each position of the hash table is called bucket.
4. **Collision:** Collision is situation in which hash function returns the same address for more than one record.
   For example:


0
1
2
3
4
25
33
54
If we want to insert 55 then
55 mod 5 = 0
But at 0th location 25 is already placed and now 55 is demanding the same location. Hence we say "collision occurs"

5. **Probe:** Each calculation of an address and test for success is known as aà probe.
6. **Synonym:** The set of keys that has to the same location are called synonyms.
   For example - In above given hash table computation 25 and 55 are synonyms.

7. **Overflow:** When hash table becomes full and new record needs to be inserted then ore it is called overflow.

   For example

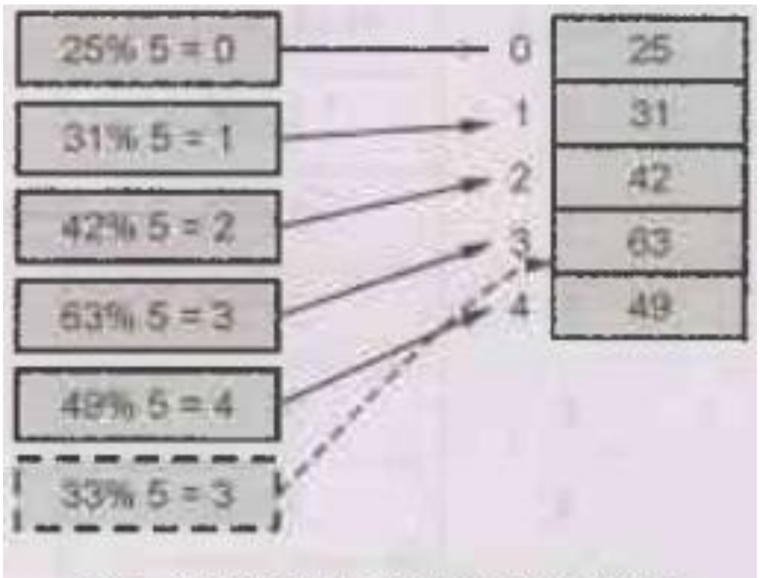
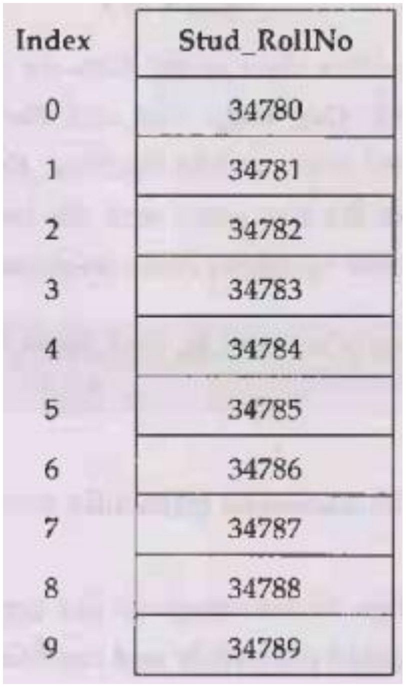


Fig. 4.10.2 : Overflow situation

Static Hashing

- In this method of hashing, the resultant data bucket address will be always same.



Index	Stud_RollNo
0	34780
1	34781
2	34782
3	34783
4	34784
5	34785
6	34786
7	34787
8	34788
9	34789

- That means, if we want to generate address for Stud_RollNo = 34789. Here if we use mod 10 hash function, it always result in the same bucket address 9. There will not be any changes to the bucket address here.

- Hence number of data buckets in the memory for this static hashing remains constant throughout. In our example, we will have ten data buckets in the memory used to store the data.

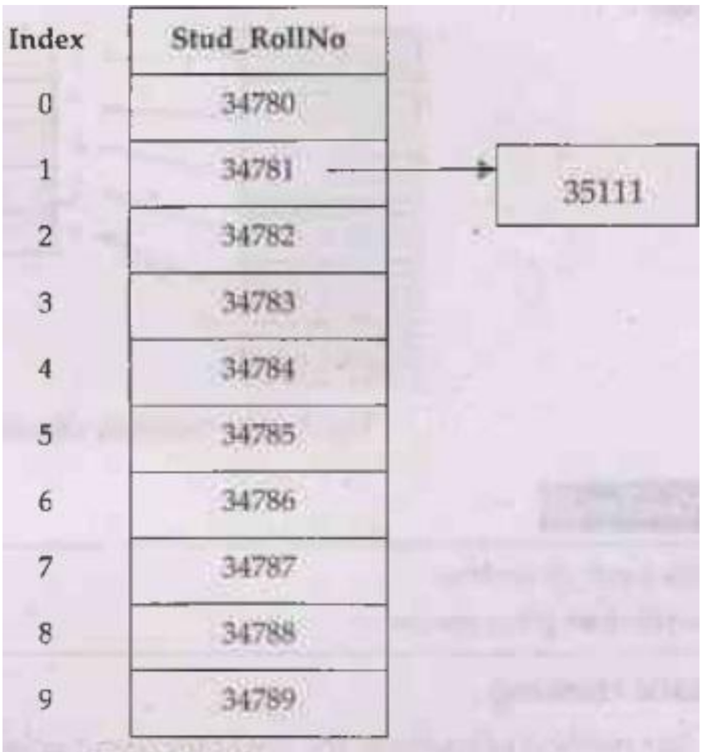


Table 4.11.1

- If there is no space for some data entry then we can allocate new overflow page, put the data record onto that page and add the page to overflow chain of the bucket. For example if we want to add the Stud_RollNo= 35111 in above hash table then as there is no space for this entry and the hash address indicate to place this record at index 1, we create overflow chain as shown in Table.

Advantages of Static Hashing

1. It is simple to implement.
2. It allows speedy data storage.

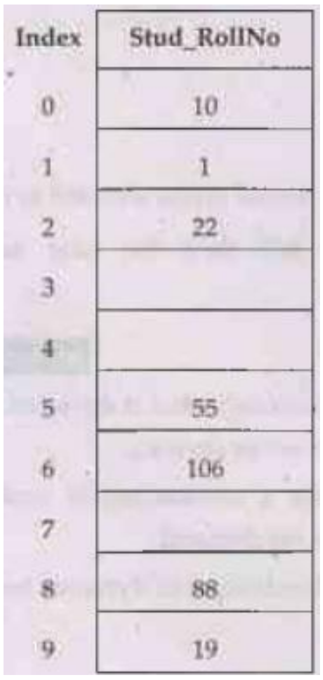
Disadvantages of Static Hashing

1. There are two major disadvantages of static hashing:
2. In static hashing, there are fixed number of buckets. This will create a problematic situation if the number of records grow or shrink.
3. The ordered access on hash key makes it inefficient.

Open Hashing

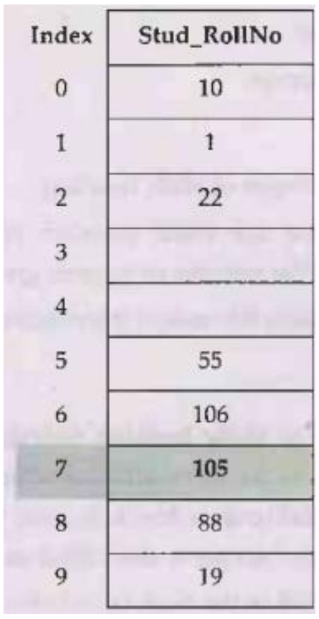
- The open hashing is a form of static hashing technique.
- When the collision occurs, that means if the hash key returns the same address which is already allocated by some data record, then the next available data block is used to enter new record instead of overwriting the old record.
- This technique is also called as linear probing.

For example: Consider insertion of record 105 in the hash table below with the hash function h (key) mod 10.



Index	Stud_RollNo
0	10
1	1
2	22
3	
4	
5	55
6	106
7	
8	88
9	19

The 105 is probed at next empty data block as follows –



Index	Stud_RollNo
0	10
1	1
2	22
3	
4	
5	55
6	106
7	**105**
8	88
9	19

Advantages of Open Hashing:

1. It is faster technique.
2. It is simple to implement.

Disadvantages of Open Hashing:

1. It forms clustering, as the record is just inserted to next free available slot.
2. If the hash table gets full then the next subsequent records can not be accommodated.

Dynamic Hashing

- The problem with static hashing is that it does not expand or shrink dynamically as the size of the database grows or shrinks.
- Dynamic hashing provides a mechanism in which data buckets are added and removed dynamically and on-demand.
- The most commonly used technique of dynamic hashing is extendible hashing.

Extendible Hashing:

The extendible hashing is a dynamic hashing technique in which, if the bucket is overflow, then the number of buckets are doubled and data entries in buckets are re- distributed.

Example:In extendible hashing technique the directory of pointers to bucket is used. Refer following Figure.

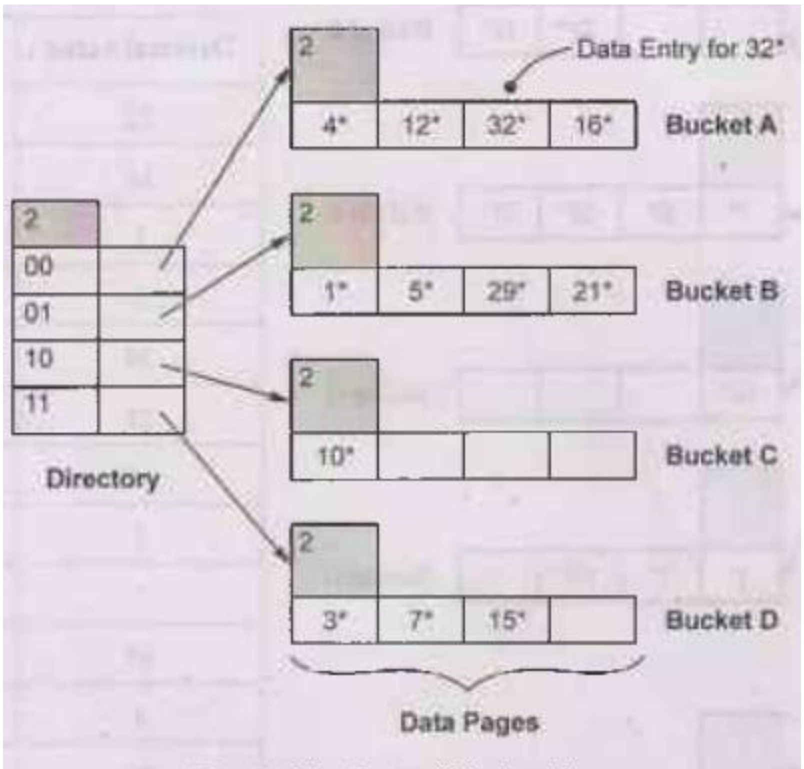
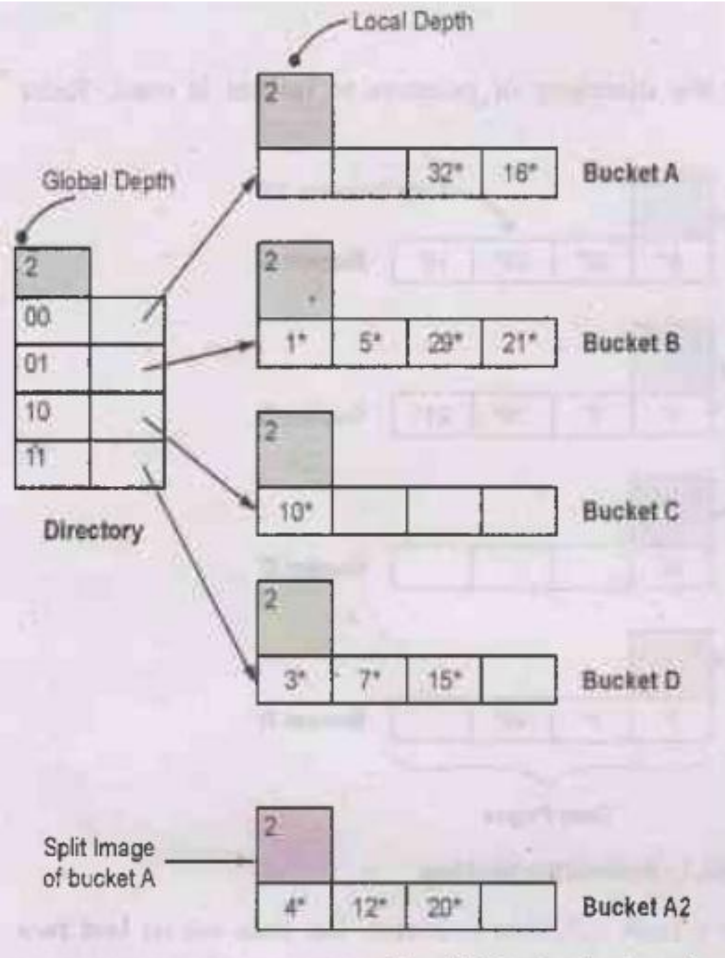


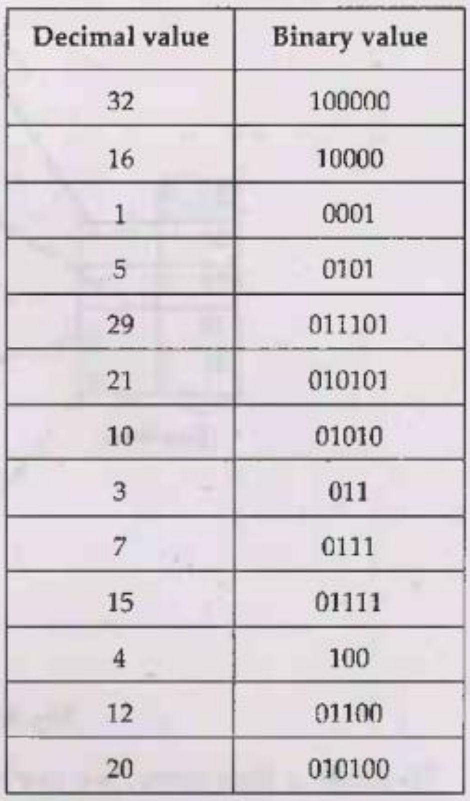
Fig. 4.12.1 : Extendible hashing

To locate a data entry, we apply a hash function to search the data we us last two digits of binary representation of number. For instance binary representation of 32* = 10000000. The last two bits are 00. Hence we store 32* accordingly.

Insertion operation :

- Suppose we want to insert 20* (binary 10100). But with 00, the bucket A is full. So we must split the bucket by allocating new bucket and redistributing the contents, bellsp across the old bucket and its split image.
- For splitting, we consider last three bits of h(r).
- The redistribution while insertion of 20* is as shown in the following Figure.





Decimal value	Binary value
32	100000
16	10000
1	0001
5	0101
29	011101
21	010101
10	01010
3	011
7	0111
15	01111
4	100
12	01100
20	010100

Fig. 4.12.2 : During insertion process

The split image of bucket A i.e. A2 and old bucket A are based on last two bits i.e. 00. Here we need two data pages, to adjacent additional data record. Therefore here it is necessary to double the directory using three bits instead of two bits. Hence,

- There will be binary versions for buckets A and A2 as 000 and 100.
- In extendible hashing, last bits d is called global depth for directory and d is called local depth for data pages or buckets. After insetion of 20*, the global depth becomes

3 as we consider last three bits and local depth of A and A2 buckets become 3 as we are considering last three bits for placing the data records. Refer the following Figure.

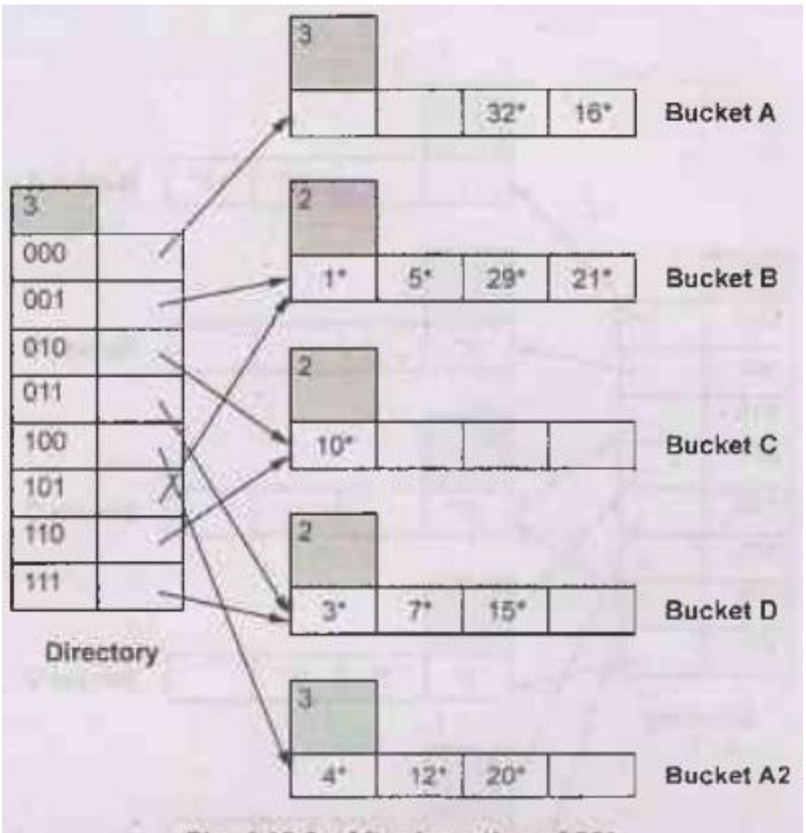


Fig. 4.12.3 : After insertion of 20*

- Suppose if we want to insert 11*, it belongs to bucket B, which is already full. Hence let us split bucket B into old bucket B and split image of B as B2.
- The local depth of B and B2 now becomes 3.
- Now for bucket B, we get and 1=001

  11 100011
- For bucket B2, we get

  5=101

  29 = 11101

  and 21 =10101
- After insertion of 11* we get the scenario as follows,

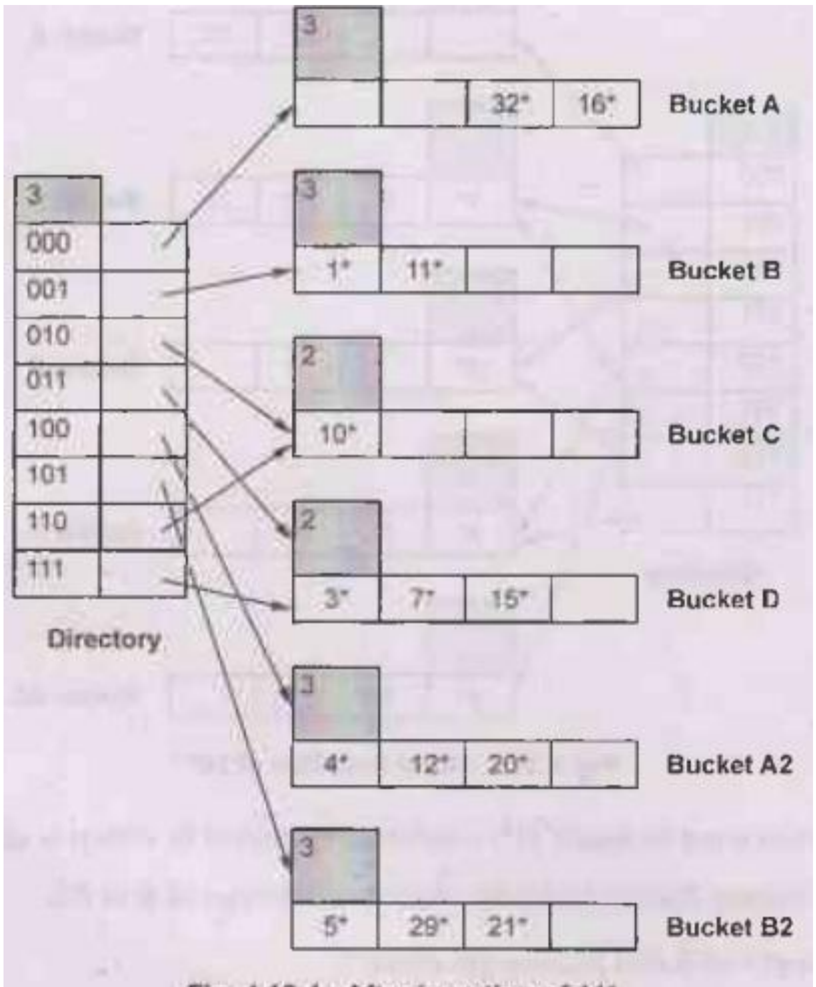
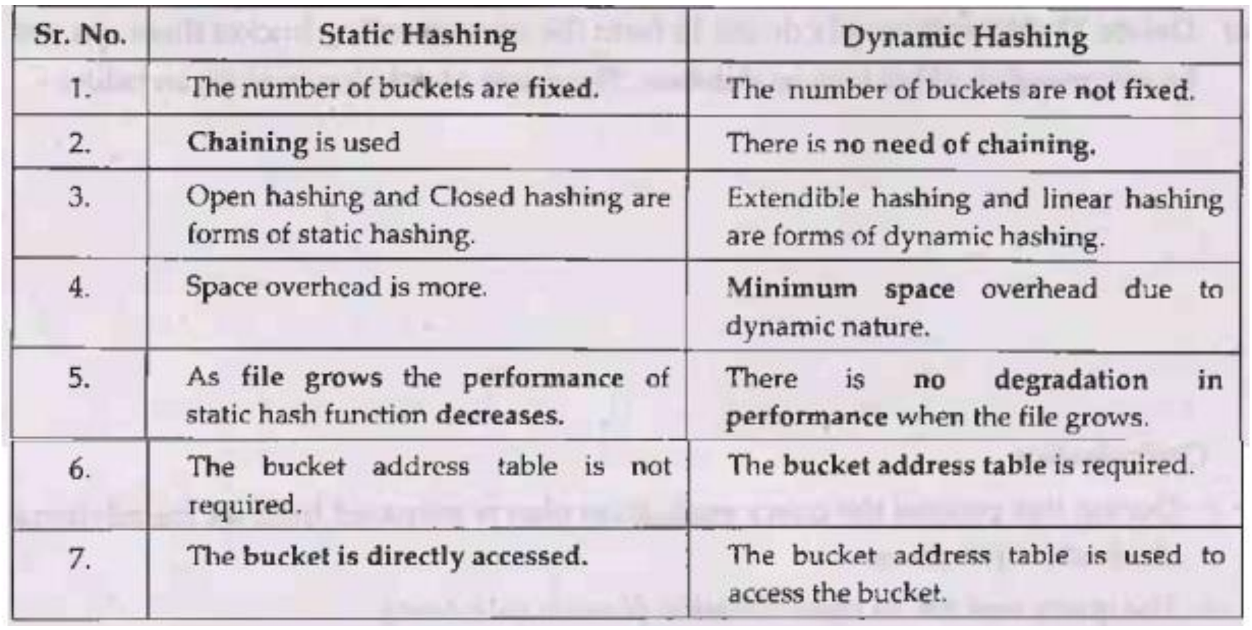


Fig. 4.12.4 : After insertion of 11*

Differences between Static and Dynamic Hashing



Sr. No.	Static Hashing	Dynamic Hashing
1.	The number of buckets are fixed.	The number of buckets are not fixed.
2.	Chaining is used	There is no need of chaining.
3.	Open hashing and Closed hashing are forms of static hashing.	Extendible hashing and linear hashing are forms of dynamic hashing.
4.	Space overhead is more.	Minimum space overhead due to dynamic nature.
5.	As file grows the performance of static hash function decreases.	There is no degradation in performance when the file grows.
6.	The bucket address table is not required.	The bucket address table is required.
7.	The bucket is directly accessed.	The bucket address table is used to access the bucket.

---END---