• File Organization / Organization of records in a file:

1) **Ordered file Organization**: All the records in the file are ordered on some search key & field.

As it is ordered, Binary search will be used for searching. because of binary search searching is efficient.

Time Complexity:

Best Case = O(1)

Worst Case = O(n)

Maintenance (insertion and deletion) is costly, as it requires re-organization of entire file.

2) **Unordered file Organization**: All the records are inserted usually in the end of the file so not ordered according to any field, Because of this only linear search is possible.

searching is slow.

Maintenance (insertion and deletion) is easy, as it does not require re-organization of entire file.

- **Reason for Indexing:** For a large file when it contains a large no. of records which will eventually acquire large no. of blocks, then its access will become slow.

→ An additional auxiliary access structure is called indexes, a data technique to efficiently retrieve records from the database file based on some attributes on which the indexing has been done. Indexing in database systems is similar to what we see in books.

→ Index typically ~~provit~~ provides secondary access path, which provide alternative way to access the records without affecting the physical placement of records in the main file.

→ Index file is always ordered, irrespective of whether main file is ordered or unordered. So, that we can take the advantage of binary search.

→ Index file always contains two columns one the attribute on which search will be done and other the block or record pointer.

→ The size of index file is very smaller than the of the main file.

→ As the size of the records is very smaller compare to main file, as index file record contain only two column key (attribute in which searching is done) and block pointer (base address of the block of main file which contains the record holding the key). while main file contains all the columns.

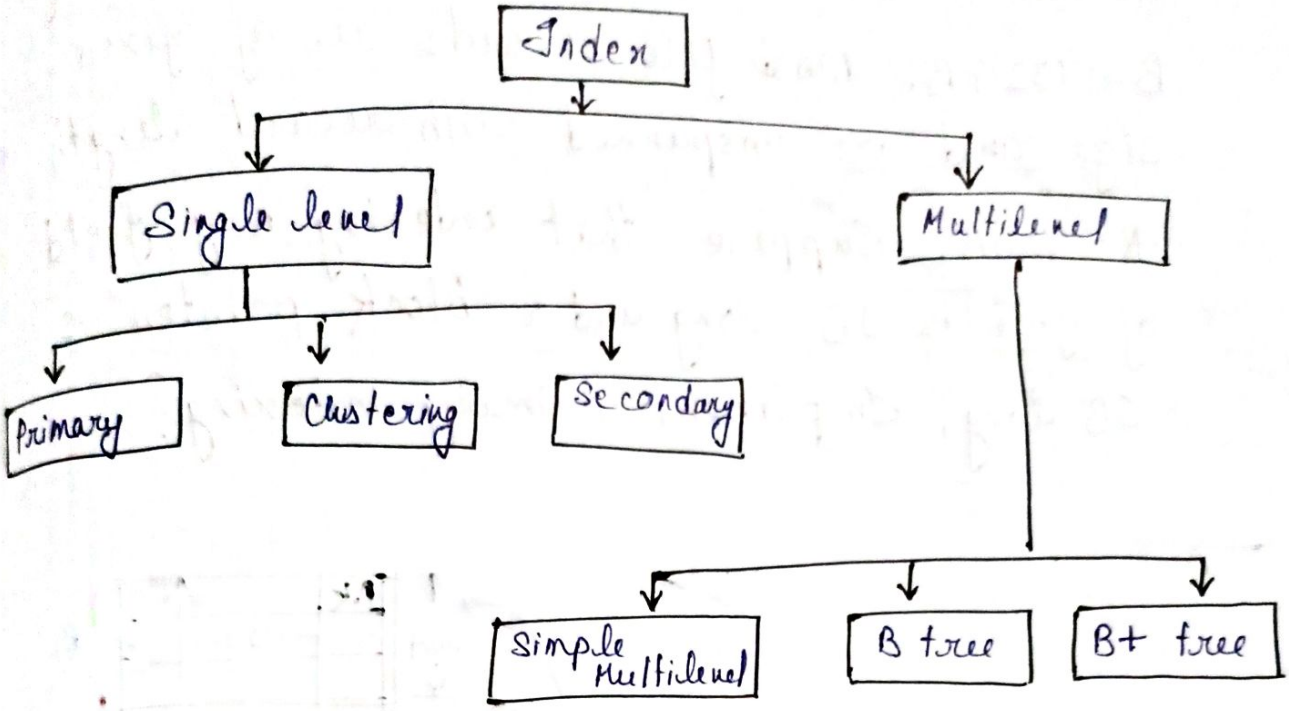
→ Normally, apart from secondary indexing (a type of indexing), the no. of records in index file <= the number of records in the main file.

→ Index can be created on any field of relation (primary key or non-key).

→ No. of access required to search the correct block of main file by index file is

$$\log_2 (\text{No. of blocks in index file}) + 1$$

- Types of Indexing



→ Single level index means we create index file for the main file and stop the process.

→ Multilevel index means, we further index the index file and keep repeating the process until we get one block.

- Basic term used in Indexing :

  - Blocking Factor : No. of records per block

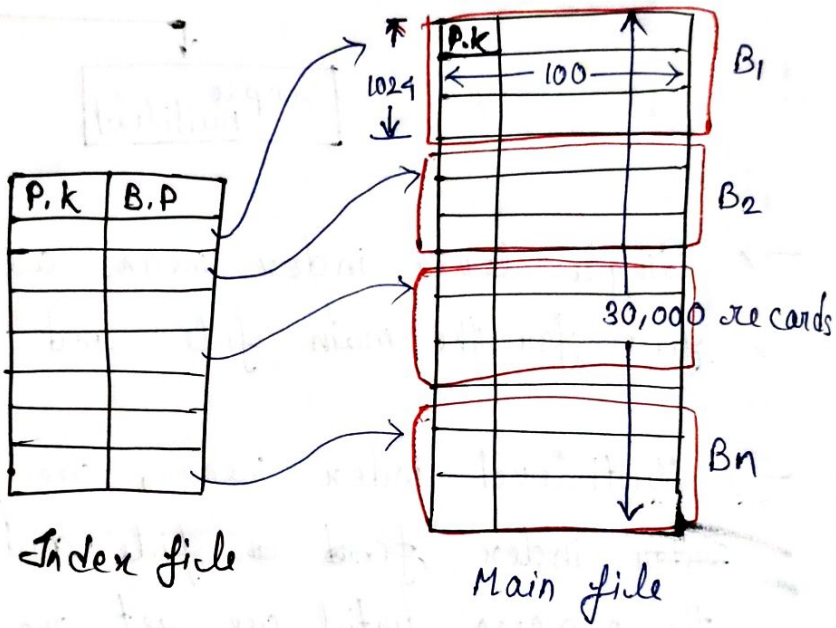
$$= \left[\frac{\text{Block size}}{\text{Record size}} \right]$$

  - No. of blocks required by file :

$$= \left[\frac{\text{no. of records}}{\text{blocking factor}} \right]$$

Q. Suppose we have ordered file with records stored $r = 30,000$ on a disk with Block size $B = 1024\,B$. File records are of fixed size and are unspanned with record length $R = 100B$. Suppose that ordering key field of file is 9B long and a block pointer is 6B long, Implement primary indexing?

→



Main file:

Block factor (no. of record in one block) $= \left\lfloor \frac{\text{Block Size}}{\text{Record length}} \right\rfloor$

$= \left\lfloor \frac{1024}{100} \right\rfloor$

$= 10$

$$\text{No. of Blocks} = \left\lceil \frac{\text{Total records (no. of entries)}}{\text{Block factor}} \right\rceil$$

$$= \frac{30,000}{10}$$

$$= 3000$$

As given main file is sorted (ordered)

$$\text{No. of access} = \lceil \log_2 (\text{No. of Blocks}) \rceil$$

$$= \lceil \log_2 (3000) \rceil$$

$$= 12$$

Index file:

$$\text{Block factor} = \left\lfloor \frac{1024}{15} \right\rfloor = 68$$

$$\text{No. of Blocks} = \left\lceil \frac{\text{Total records (no. of entries)}}{\text{Block factor}} \right\rceil$$

$$= \left\lceil \frac{3000}{68} \right\rceil = 45$$

$$\therefore \text{No. of access} = \lceil \log_2 (45) \rceil + 1$$

(extra for BP)

$$= 6 + 1 = 7$$

Hence, No. of access reduced from 12 to 7. that is almost 50% fast access achieved due to indexing.

9.10 Parallelizing Disk Access Using RAID Technology

With the exponential growth in the performance and capacity of semiconductor devices and memories, faster microprocessors with larger and larger primary memories are continually becoming available. To match this growth, it is natural to expect that secondary storage technology must also take steps to keep up in performance and reliability with processor technology.

A major advance in secondary storage technology is represented by the development of **RAID**, which originally stood for **Redundant Arrays of Inexpensive Disks**. Lately, the "I" in RAID is said to stand for Independent. The RAID idea received a very positive endorsement by industry and has been developed into an elaborate set of alternative RAID architectures (RAID levels 0 through 6). We highlight the main features of the technology below.

The main goal of RAID is to even out the widely different rates of performance improvement of disks against those in memory and microprocessors.[12] While RAM capacities have quadrupled every two to three years, disk *access times* are improving at less than 10 percent per year, and disk *transfer rates* are improving at roughly 20 percent per year. Disk *capacities* are indeed improving at more than 50 percent per year, but the speed and access time improvements are of a much smaller magnitude. Table 9.3 shows trends in disk technology in terms of 1993 parameter values and rates of improvement, as well as where these parameters are in 2003.

A second qualitative disparity exists between the ability of special microprocessors that cater to new applications involving processing of video, audio, image, and spatial data with corresponding lack of fast access to large, shared data sets.

The natural solution is a large array of small independent disks acting as a single higher-performance logical disk. A concept called **data striping** is used, which utilizes *parallelism* to improve disk performance. Data striping distributes data transparently over multiple disks to make them appear as a single large, fast disk. Figure 9.12 shows a file distributed or *striped* over four disks. Striping improves overall I/O performance by

Table 9.3 Trends in Disk Technology

	1993 Parameter Values*	Historical Rate of Improvement Per Year (%)*	Current (2003) Values**
Areal density	50–150 Mbits/sq. inch	27	36 Gbits/sq. inch
Linear density	40,000–60,000 bits/inch	13	570 Kbits/inch
Inter-track density	1500–3000 tracks/inch	10	64,000 tracks/inch
Capacity (3.5" form factor)	100–2000 MB	27	146 GB
Transfer rate	3–4 MB/s	22	43–78 MB/sec
Seek time	7–20 ms	8	3.5–6 msec

*Source: From Chen, Lee, Gibson, Katz, and Patterson (1994), ACM *Computing Surveys*, Vol. 26, No. 2 (June 1994). Reprinted by permission.
**Source: IBM Ultrastar 36XP and 18ZX hard disk drives.

12. This was predicted by Gordon Bell to be about 40 percent every year between 1974 and 1984 and is now supposed to exceed 50 percent per year.

allowing multiple I/Os to be serviced in parallel, thus providing high overall transfer rates. Data striping also accomplishes load balancing among disks. Moreover, by storing redundant information on disks using parity or some other error correction code, reliability can be improved. In Sections 9.10.1 and 9.10.2, we discuss how RAID achieves the two important objectives of improved reliability and higher performance. Section 9.10.3 discusses RAID organizations.

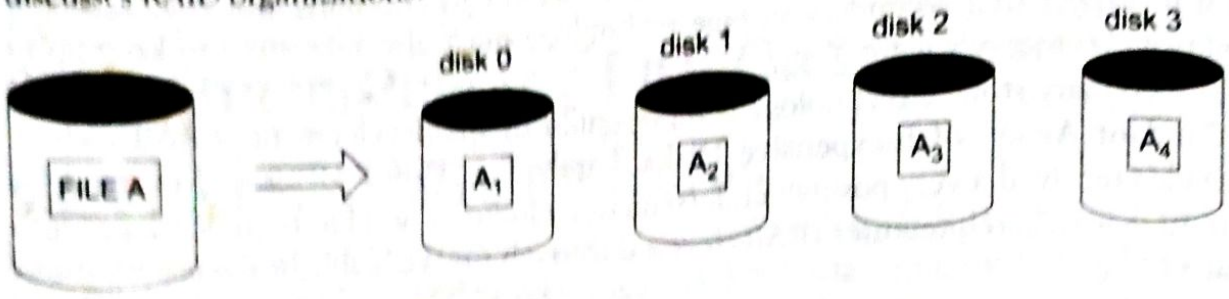


FIGURE 9.12 Data striping. File A is striped across four disks.

9.10.1 Improving Reliability with RAID

For an array of n disks, the likelihood of failure is n times as much as that for one disk. Hence, if the MTTF (Mean Time To Failure) of a disk drive is assumed to be 200,000 hours or about 22.8 years (typical times range up to 1 million hours), that of a bank of 100 disk drives becomes only 2000 hours or 83.3 days. Keeping a single copy of data in such an array of disks will cause a significant loss of reliability. An obvious solution is to employ redundancy of data so that disk failures can be tolerated. The disadvantages are many: additional I/O operations for write, extra computation to maintain redundancy and to do recovery from errors, and additional disk capacity to store redundant information.

One technique for introducing redundancy is called **mirroring** or **shadowing**. Data is written redundantly to two identical physical disks that are treated as one logical disk. When data is read, it can be retrieved from the disk with shorter queuing, seek, and rotational delays. If a disk fails, the other disk is used until the first is repaired. Suppose the mean time to repair is 24 hours, then the mean time to data loss of a mirrored disk system using 100 disks with MTTF of 200,000 hours each is $(200{,}000)^2/(2 * 24) = 8.33 * 10^8$ hours, which is 95,028 years.[13] Disk mirroring also doubles the rate at which read requests are handled, since a read can go to either disk. The transfer rate of each read, however, remains the same as that for a single disk.

Another solution to the problem of reliability is to store extra information that is not normally needed but that can be used to reconstruct the lost information in case of disk failure. The incorporation of redundancy must consider two problems: (1) selecting a technique for computing the redundant information, and (2) selecting a method of distributing the redundant information across the disk array. The first problem is addressed by using error correcting codes involving parity bits, or specialized codes such as Hamming codes. Under the parity scheme, a redundant disk may be considered as having the sum of all the data in the other disks. When a disk fails, the missing information can be constructed by a process similar to subtraction.

For the second problem, the two major approaches are either to store the redundant information on a small number of disks or to distribute it uniformly across all disks. The latter results in better load balancing. The different levels of RAID choose a combination of these options to implement redundancy, and hence to improve reliability.

9.10.2 Improving Performance with RAID

The disk arrays employ the technique of data striping to achieve higher transfer rates. Note that data can be read or written only one block at a time, so a typical transfer contains 512 bytes. Disk striping may be applied at a finer granularity by breaking up a byte of data into bits and spreading the bits to different disks. Thus, **bit-level data striping** consists of splitting a byte of data and writing bit j to the j^{th} disk. With 8-bit bytes, eight physical disks may be considered as one logical disk with an eightfold increase in the data transfer rate. Each disk participates in each I/O request and the total amount of data read per request is eight times as much. Bit-level striping can be generalized to a number of disks that is either a multiple or a factor of eight. Thus, in a four-disk array, bit n goes to the disk which is (n mod 4).

The granularity of data interleaving can be higher than a bit; for example, blocks of a file can be striped across disks, giving rise to **block-level striping**. Figure 9.12 shows blocklevel data striping assuming the data file contained four blocks. With block-level striping, multiple independent requests that access single blocks (small requests) can be serviced in parallel by separate disks, thus decreasing the queuing time of I/O requests. Requests that access multiple blocks (large requests) can be parallelized, thus reducing their response time. In general, the more the number of disks in an array, the larger the potential performance benefit. However, assuming independent failures, the disk array of 100 disks collectively has a $1/100^{th}$ the reliability of a single disk. Thus, redundancy via error-correcting codes and disk mirroring is necessary to provide reliability along with high performance.

9.10.3 RAID Organizations and Levels

Different RAID organizations were defined based on different combinations of the two factors of granularity of data interleaving (striping) and pattern used to compute redundant information. In the initial proposal, levels 1 through 5 of RAID were proposed, and two additional levels—0 and 6—were added later.

RAID level 0 uses data striping, has no redundant data and hence has the best write performance since updates do not have to be duplicated. However, its read performance is not as good as RAID level 1, which uses mirrored disks. In the latter, performance improvement is possible by scheduling a read request to the disk with shortest expected seek and rotational delay. RAID level 2 uses memory-style redundancy by using Hamming codes, which contain parity bits for distinct overlapping subsets of components. Thus, in one particular version of this level, three redundant disks suffice for four original disks whereas, with mirroring—as in level 1—four would be required. Level 2 includes both error detection and correction, although detection is generally not required because broken disks identify themselves.

RAID level 3 uses a single parity disk relying on the disk controller to figure out which disk has failed. Levels 4 and 5 use block-level data striping, with level 5 distributing data and parity information across all disks. Finally, RAID level 6 applies the so-called $P + Q$ redundancy scheme using Reed-Soloman codes to protect against up to two disk failures by using just two redundant disks. The seven RAID levels (0 through 6) are illustrated in Figure 9.13 schematically.

Rebuilding in case of disk failure is easiest for RAID level 1. Other levels require the reconstruction of a failed disk by reading multiple disks. Level 1 is used for critical applications such as storing logs of transactions. Levels 3 and 5 are preferred for large volume storage, with level 3 providing higher transfer rates. Most popular use of RAID technology currently uses level 0 (with striping), level 1 (with mirroring) and level 5 with an extra drive for parity. Designers of a RAID setup for a given application mix have to confront many design decisions such as the level of RAID, the number of disks, the choice of parity schemes, and grouping of disks for block-level striping. Detailed performance studies on small reads and writes (referring to I/O requests for one striping unit) and large reads and writes (referring to I/O requests for one stripe unit from each disk in an error-correction group) have been performed.

- **Types of Single level indexing :**

  i) Primary Indexing

  ii) Secondary Indexing

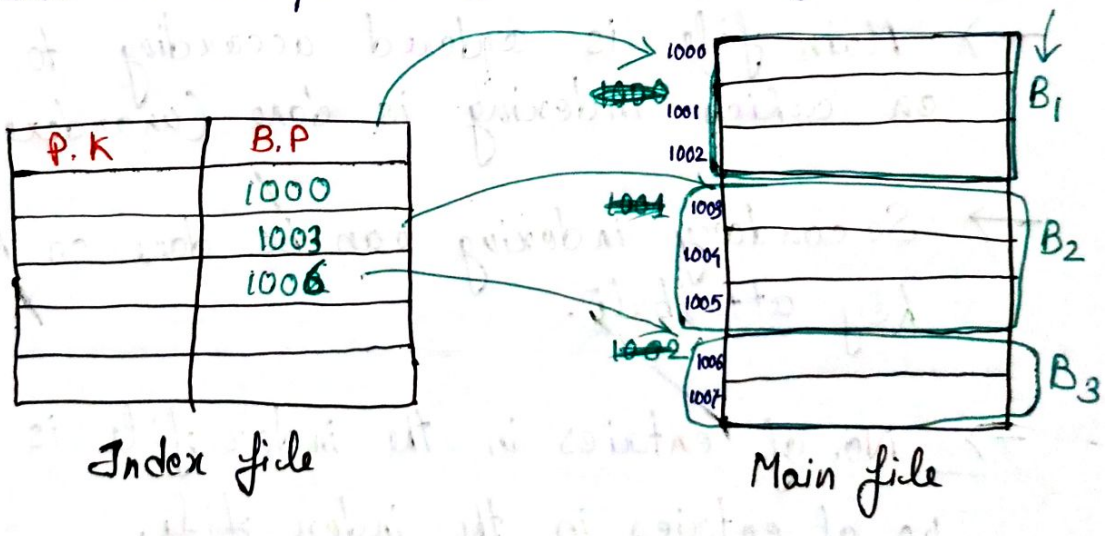
  iii) Clustering Indexing.

i) Primary Indexing :

→ Main file is always sorted according to primary key.

→ Indexing is done on Primary key, Hence, the name Primary Indexing.

→ Index file have two columns, first primary key and second anchor pointer (base address of block)



→ No. of entries in the index file = No. of blocks acquired by the main file.

ii) Clustering Indexing:

→ Main file will be ordered on some non-key attributes

→ No. of entries in the index file = no. of unique values of the attribute on which indexing is done.

iii) Secondary Indexing:

→ Most common scenario, suppose that we already have a primary indexing on primary key, but there is frequent query on some other attributes, so we may decide to have one more index file with some other attribute

→ Main file is ordered according to the attribute on which indexing is done (unordered).

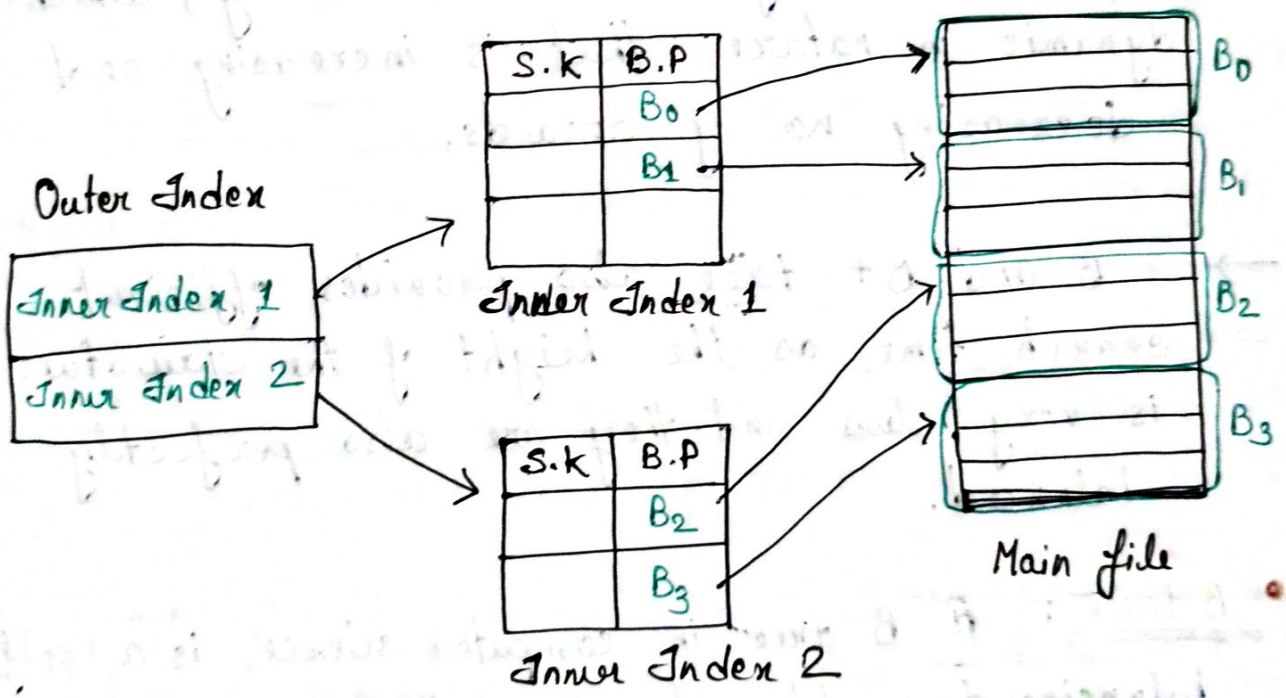
→ Secondary indexing can be done on key or non-key attribute.

→ No. of entries in the index file is same as the no. of entries in the index file.

• Multilevel Indexing:

→ It involves creating multiple layers of indexes, where each level provides more detailed information about the data's location.

→ Multi-level indexing helps in breaking down the index into several down the index into several smaller indices in order to make the outermost level so small that it can ~~cover~~ be stored in a single disk block - 0, which can easily be accomodated anywhere in the main memory.



- **Need of B tree and B+ tree:**

→ An index file is always sorted in nature and will be searched frequently, and sometimes index files can be so large that we want to index file the index file (ie., multilevel index), therefore we must use best data structure to meet our requirements.

→ In general, with multilevel indexing, we require dynamic structure, B and B+ tree is generalized implementation of multilevel indexing, which are dynamic in nature, that is increasing and decreasing no. of records.

→ B and B+ tree also provides efficient search time, as the height of the structure is very less and they are also perfectly balanced.

- **B tree:** B tree in computer science, is a self balancing tree data structure that maintains sorted data and allows searches, sequential access, insertions and deletion in logarithmic time.

→ We can use a search tree as a mechanism to search for records stored in a disk file. The values in the tree can be the values of one of the fields of the file, called the search field.

→ Each key value in the tree is associated with a pointer to the record in the data file having that value.

→ To guarantee that nodes are evenly distributed, so that the depth of the tree is minimized for the given set of keys and that the tree does not get skewed with some nodes being at very deep levels.

→ While minimizing the number of levels in the tree is one goal, another implicit goal is to make sure that index tree does not need too much restructuring as records are inserted into and deleted from the main file. Thus, we want the nodes to be as full as possible and do not want any nodes to be empty, if there are too many deletions. Record deletion may leave some nodes in the tree nearly empty, thus wasting storage space and increasing the no. of levels.

→ The B tree addresses both of these problems by specifying additional constraints on the search tree.

→ The algorithms for insertion and deletion though, become more complex in order to maintain these constraints. Nonetheless, most insertions and deletions are simple processes; they become complicated only under special circumstances — namely, whenever we attempt an insertion into a node that is already full or a deletion from a node that makes it less than half full.

- **B tree :**

→ A B tree of order m if non-empty is an m-way search tree in which :-

→ The root has atleast zero child nodes and atmost m child nodes.

→ The internal nodes except the root have atleast $\lceil \frac{m}{2} \rceil$ child nodes and at most m child nodes.

→ The no. of keys in each internal node is one less than the no. of child nodes and these keys partition the subtrees of the nodes in a manner similar to that of m-way search tree.

→ All leaf nodes are on the same level (perfectly balanced).

Root node

	Max	min
Child	m	0
Data	m-1	1

Internal node

	Max	min
Child	m	$\lceil \frac{m}{2} \rceil$
Data	m-1	$\lceil \frac{m}{2} \rceil - 1$

leaf

	max	min
Child	0	0
Data	m-1	$\lceil \frac{m}{2} \rceil - 1$

- **Insertion in B Tree :**

1) A B tree starts with a single root node (which is a leaf node) at level 0. Once the root nodes is full with $m-1$ search key values and we attempt to insert another entry in the tree, the root node splits into two nodes at level 1.

2) Only the middle value is kept in the root node, and the rest of the values are split evenly between the other two nodes. When a non-root node is full and a new entry is inserted into it, that node is split into two nodes at the same level and the middle entry is moved to the parent node along with two pointers to the new split nodes.

3) If the parent node is full, it is also split. Splitting can propagate all the way to the root node, creating a new level if the ~~rot~~ root is split.

Q. Consider the following elements 5, 10, 12, 13, 14, 1, 2, 3, 4 insert them into an empty B tree of order 3.

5, 10, 12, 13, 14, 1, 2, 3, 4

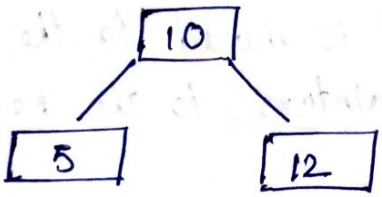
→ Order = m = 3

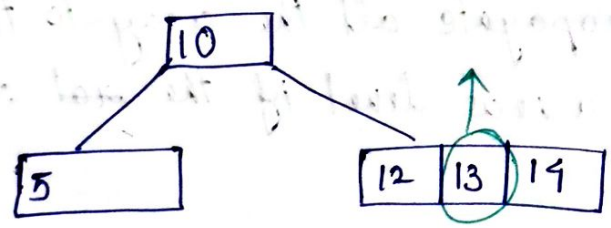
data element = m - 1
= 2

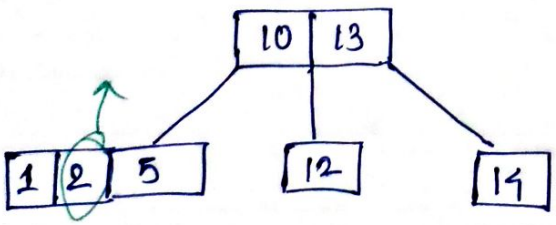
~~5, 10, 10~~

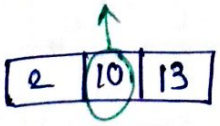
① 

data > 2
∴ shift middle element upwards

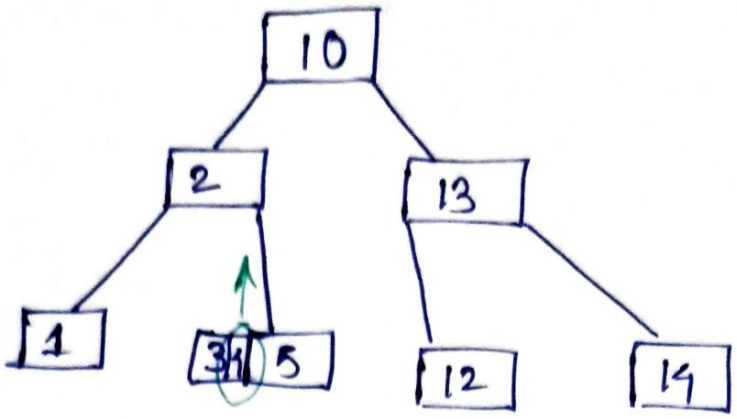


② 

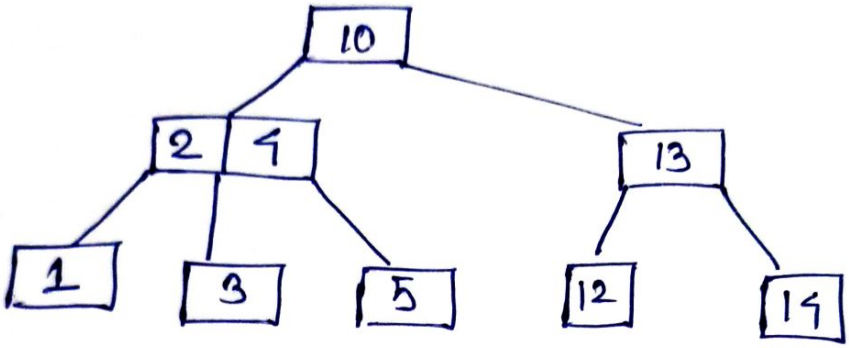
③ 

④ 

(5)



(6)



- **Advantage of B tree :**

1) Memory utilization is very good.
2) Less no. of nodes (blocks) are used and height is also optimized, and access will be very fast.

- **Disadvantage of B tree :**

1) Difficulty of traversing the key sequentially, Means B-Tree do not hold good for range-based queries of database.

- **B+ tree :** Sequential search possible.

→ Most implementation of a dynamic multilevel index use a variation of the B tree data structure called a B + tree.

→ In a B-tree, every value of the search field appears once at some level in the tree, along with a data pointer. But in a B+ tree, data pointers are stored only at the leaf nodes of the tree, hence the structure of leaf node differs from the structure of internal nodes.

→ The leaf nodes of B+ tree are usually linked to provide ordered access on the search field to the records.

• Insertion in B+ tree :

1) Start from root node and proceed towards leaf using the logic of binary search tree.

2) If overflow condition occurs pick the median and push it into the parent node. Also copy the median or key inserted in parent node to the left or right child node.

3) Repeat this procedure until tree is maintained.

Q. Insert the following elements into an empty B+ tree of order 3.

3, 6, 8, 4, 2, 1

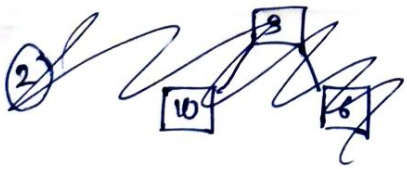
→ Order = m = 3

data element = m − 1 = 2

Using left copy:

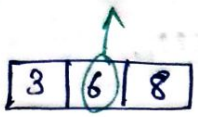
3, 6, 8, 4, 2, 1

if data element > 2, push middle element upwards.

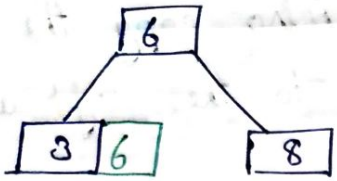


P.T.O

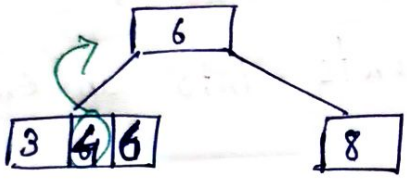
Using Left Copy: 3, 6, 8, 4, 2, 1

1. 

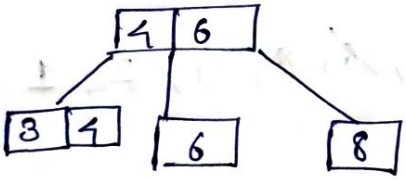
data element > 2, push middle upwards.

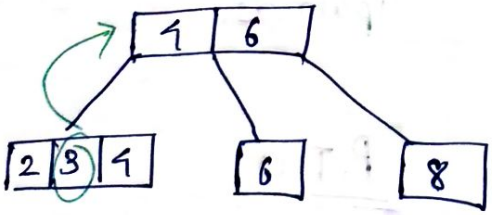
2. 

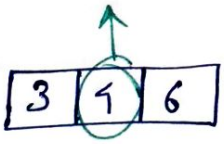
maintain copy of shifted element in left of shifted element.

3. 

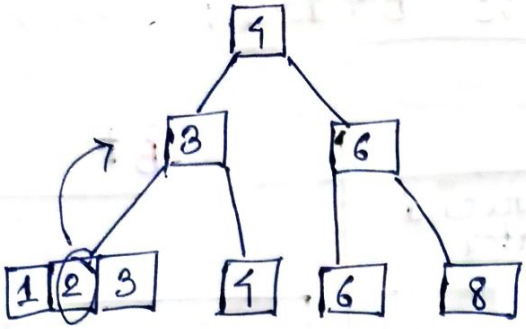
Inserted element 4. but data element is overflowing i.e. > 2. Hence shift middle element upwards. and maintain copy at left.

4. 

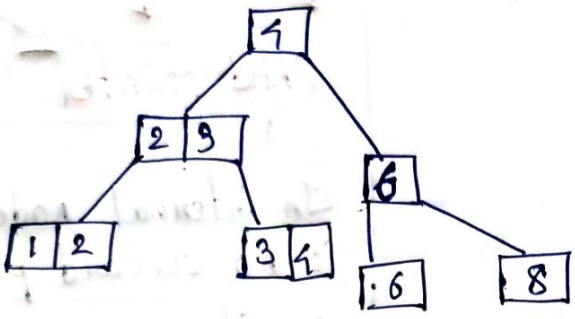
5. 

6. 

(7)

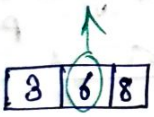


(8)

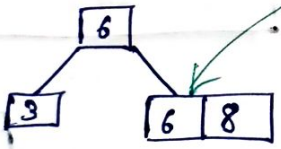


Using Right Copy : 3, 6, 8, 4, 2, 1

(1)

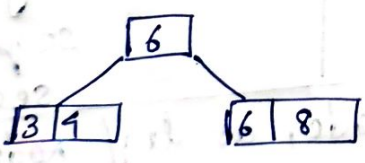


(2)

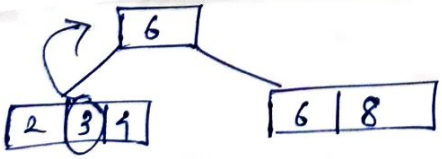


(6) Copy maintained in right of shifted element (6)

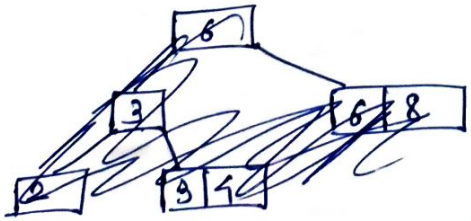
(3)

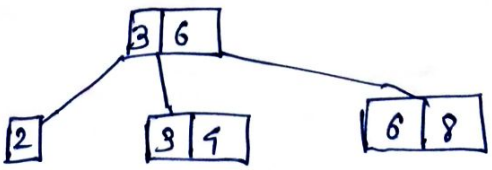


(4)

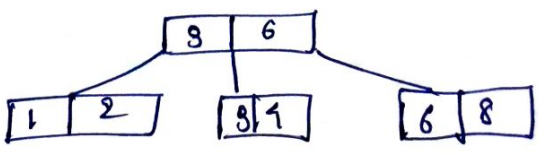


(5)

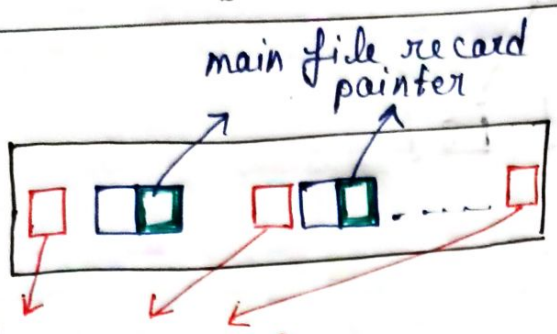
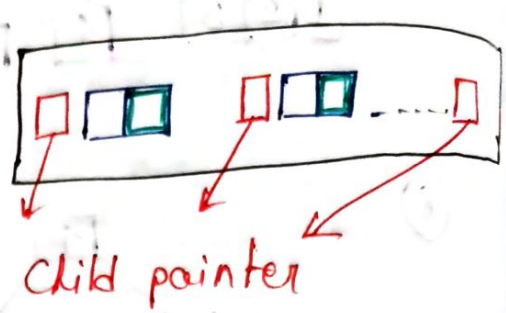
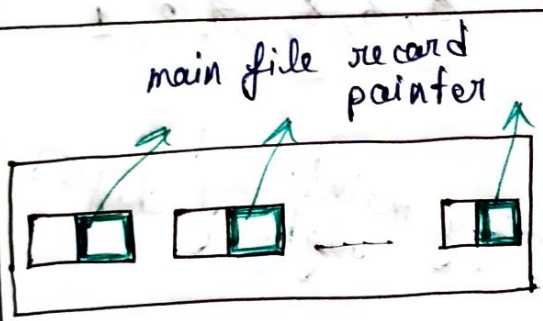
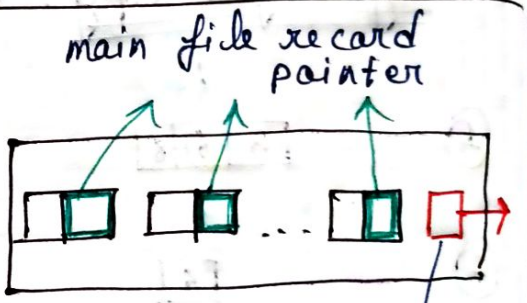




(7)



- Structure of B tree vs B+ tree :

	B	B+
Internal node		 In internal node there is no record pointer so its copy is maintained in leaf node.
Leaf node		

No child pointer in a leaf node, as leaf node does not have any child.