UNIT-5

UNIT-5

DATA STORAGE AND INDEXING

→ This chapter initiates a study of the internals of an RDBMS. Interms of DBMS architecture it covers

1. Disk Space Manager
2. Buffer Manager
3. The layer that supports the abstraction of file of records.

→ Data in a DBMS is stored on storage devices such as disks and tapes.

→ The disk space manager is responsible for keeping track of available disk space.

→ File Manager:-

File Manager provides the abstraction of file of records to higher levels of DBMS code, It issues requests to the disk space manager to obtain and relinquish space on disk. The file manager layer requests and frees disk space in units of page.

→ The File management layer is responsible for keeping track of the pages in a file and for arranging records within pages. A record has a unique identifier called "RecordId".

→ When a record is needed for processing, it must be fetched from disk to main memory. The page on which the record resides is determined by the file manager.

Buffer Manager:-

After identifying the required page, the file manager requests for the page to layer of DBMS code.

The Buffer manager fetches a request page from disk into a region of main memory called Buffer pool and tells the file manager the location of the request page.

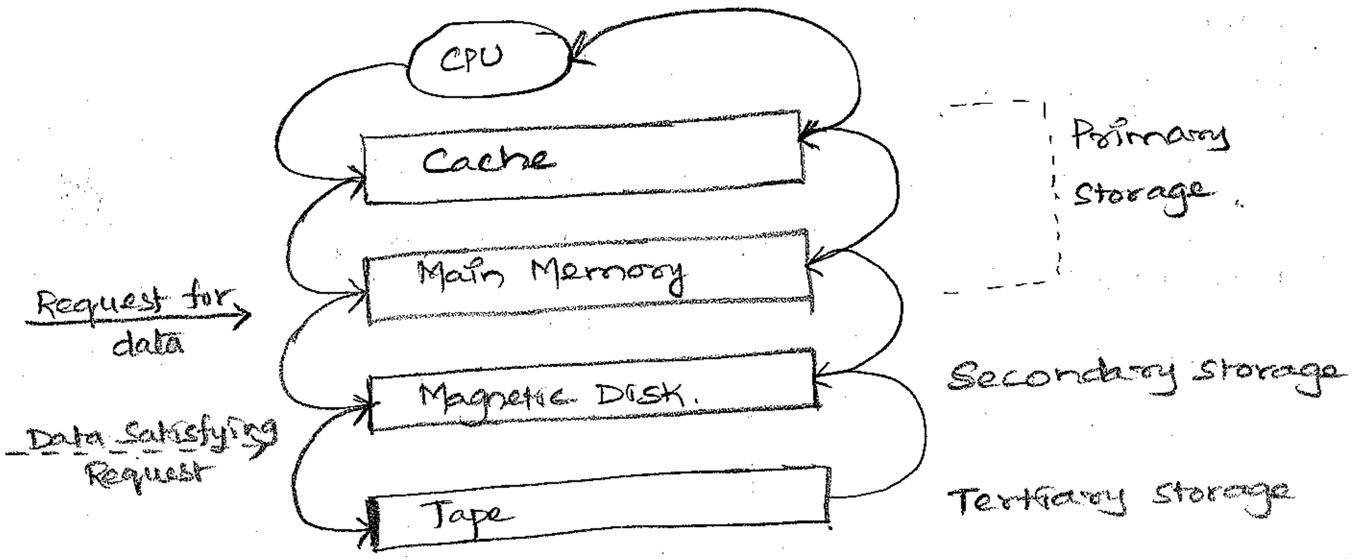
THE MEMORY HIERARCHY :-

Memory in a computer system is arranged in a Hierarchy

1. Primary storage
2. Secondary storage
3. Tertiary storage

. Primary storage consists of cache and main memory and provides very fast access to data.

· Secondary storage consists of slower devices such as magnetic disk, and tertiary storage devices are the slowest class. of devices such as optical disks and tapes.



THE MEMORY HIERARCHY.

Magnetic Disks:-

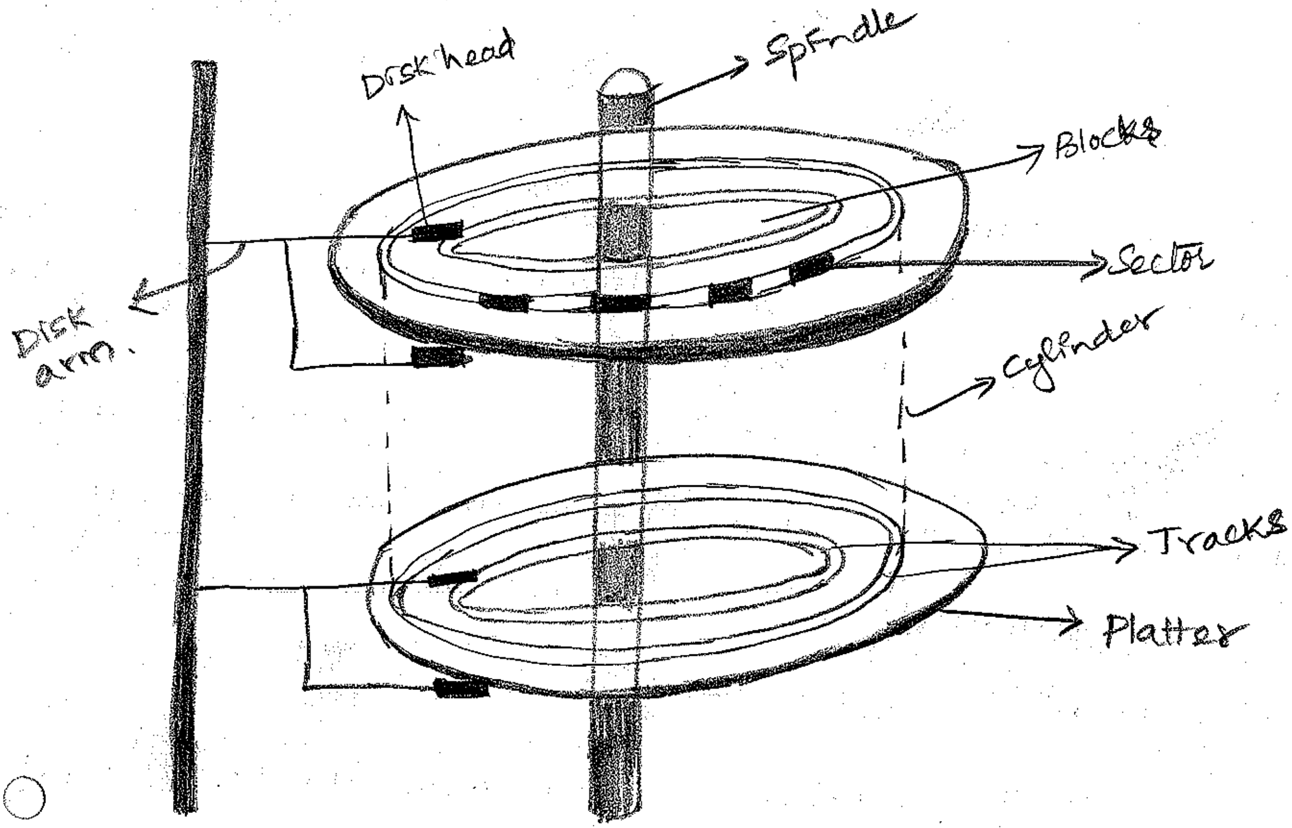
Magnetic disks supports direct access to a desired location and which are widely used for database applications.

A DBMS provides seamless access to data on disk;

The structure of a disk consists of:

1. Disk Blocks
2. Tracks
3. Cylinder.
4. Sectors. etc.

5-2



Structure of Disk.

Disk. Blocks:-
Data is stored on disks in units called Disk Blocks. A disk Block is a contiguous sequence of bytes and, the unit in which data is written to a disk and read from a disk.

Tracks:-
Block are arranged in concentric ring called Tracks, on one (or) more platters. Tracks can be recorded on one (or) both surfaces of a platter, The platters are single sided or double sided accordingly

Cylinder:-
The set of all tracks with the same diameter is called a cylinder, ~~A~~ cylinder contains one track per platter surface

Sector:-
Each track is divided into arcs called Sectors,

A sector size of the disk can not be changed. The size of the disk block can be set when the disk is initialized as a multiple of the sector size.

→ An array of disk heads, one per recorded surface, is moved as a unit, when one head is positioned over a block, the other heads are in identical positions with respect to their platters. To read or write a block, a disk head must be positioned on top of the block. As the size of the platter decreases, seek time also decrease since we have to move a disk head a smaller distance.

→ A Disk controller interfaces a disk drive to the computer. It implement commands to read or write a sector by moving the arm assembly and transferring data to and from the disk surfaces.

→ A checksum is computed for when data is written to a sector and stored with the sector. The checksum is computed again when the data on the sector is read back.
If the sector is corrupted or the read is faulty the checksum ... computed when the sector was written. The controller computes checksums and if it detects an error, it tries to read the sector again.

→ Seek time:-
Seek time is the time taken to move the disk heads to the track on which a desired block is located.

→ Rotational Delay:-
It is the waiting time for the desired block to rotate under the disk head. It is the time required for half a rotation on average and is usually less than seek time.

5.7

Transfer time:-

It is the time to actually read or write the data in the block once the head is positioned.

(i.e) the time for the disk to rotate over the block.

* Performance implications of Disk structure:-

1). Data must be in memory for the DBMS to operate on it

2) The unit of data transfer between disk and main memory is a block;
If a single item on a block is needed, the entire block is transferred.

∴ Reading or writing a disk block is called an I/O operation
(I/O → Input/output).

3) The time to read or write a block varies, depending on the location of the data.

$$\boxed{\text{access time} = \text{seek time} + \text{rotational delay} + \text{transfer time}}$$

RAID - Redundant Array of Independent Disk.

A disk array is an arrangement of several disks, organized so as to increase performance and improve reliability of the resulting storage system.

→ Reliability is improved through redundancy. Instead of having a single copy of the data, redundant information is maintained. The redundant information is carefully organized so that incase of disk failure, it can be used to reconstruct the contents of the failed disk.

→ Disk arrays that implements a combination of data striping and redundancy are called " Redundant Array of Independent Disks (RAID).

Some Important terms:-

Mean time to failure (MTTF)

It is a measure of reliability of the disk.

MTTF of the disk (or) system is the average amount of time the system to run continuously without any failure.

→ Reliability of a disk array can be increased by storing redundant information.

→ We have make 2 choices, when incorporating redundancy into a disk array design.

(1) We have to decide where to store the redundant information. We can either store the redundant information on a small number of check disks (or) distribute the redundant information uniformly over all disks.

(2) We have to make a choice of how to compute a redundant information. Most disk arrays stores parity information;
(i.e) The parity scheme, an extra check disk contains information that can be used to recover from failure of any one disk in the array.

→ In the RAID system, the disk array is partitioned into reliability groups.

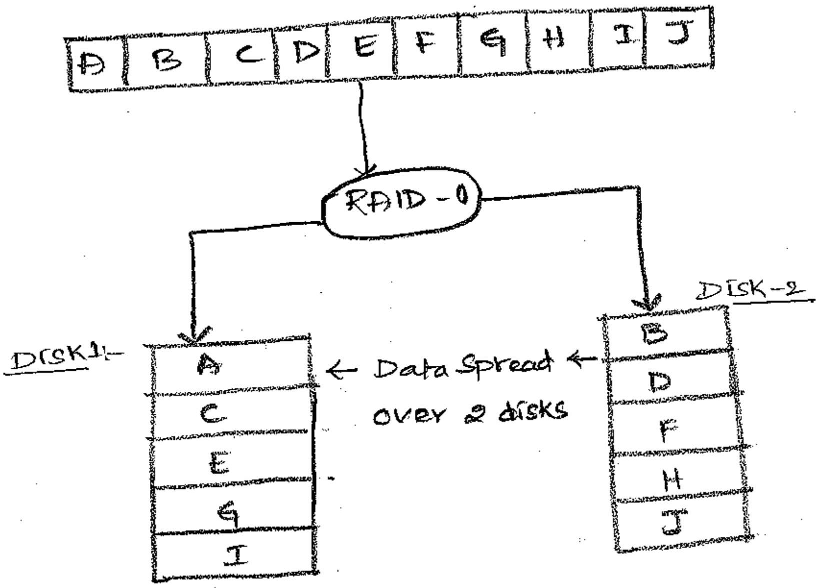
A Reliability group consisting of a set of data disks and a set of check disks.

→ A common redundancy scheme is applied on each group, the no. of check disks depends on the RAID Level choosen.

RAID Levels:-

Level-0 – stripping:-

A Raid level-0 system uses data stripping to increase the maximum bandwidth available. No redundant information is maintained



→ Data striped across multiple disks. Level-0 is not fault-tolerant. Since RAID-0 provides no redundancy, the failure of one disk can cause the entire array to fail.

Level-1: Mirroring.

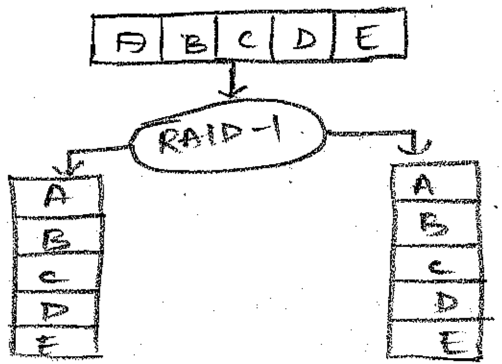
It is the most expensive solution. Here, we are copying the same data in two disks.

(i.e) We are storing the same data in 2 seperate disks

This type of redundancy is often called "Mirroring".

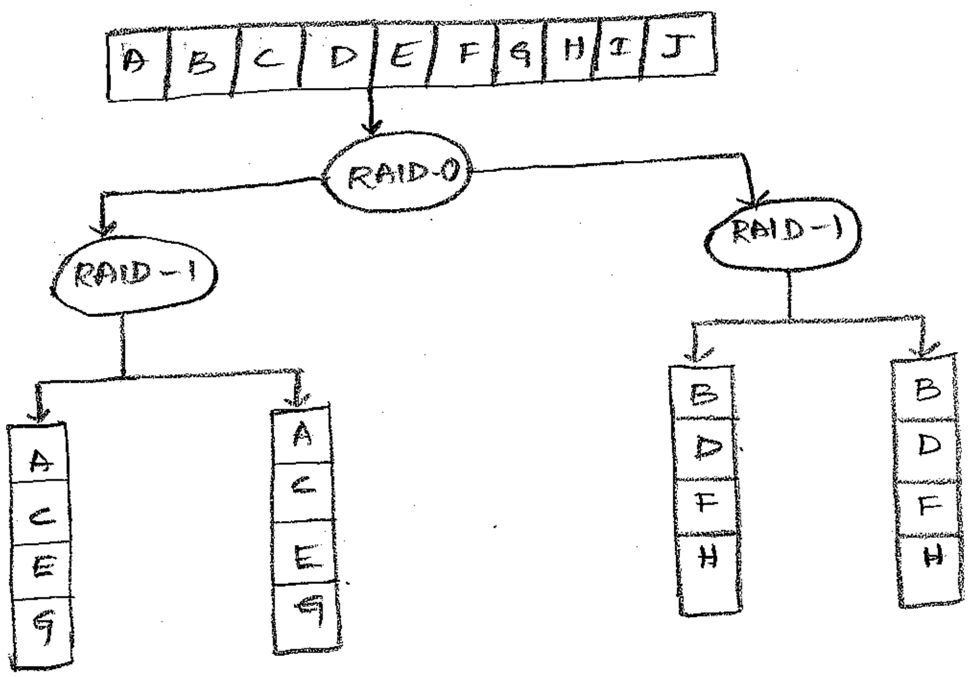
Every write on a disk block involves a write on both disks

→ We write a block on one disk first and then write the other copy on the mirrored disk.



Level 0+1 (or) Level-10 → striping and Mirroring.

This level is a combination of RAID-0 and RAID-1. RAID-1, read requests of the size of the disk block can be scheduled both to disk and its mirror image.



RAID-0+1

Level-2 : Error-Correcting Codes:-

In this level, the striping unit is a single unit. The redundancy scheme used is Hamming Code.

Suppose, if there are four data disks, then 3 check disks are needed. In general the no. of. check disks grows. Logarithmically with the no. of. data disks.

Level-3 :- Bit Interleaved parity:-

Instead of using several disks to store Hamming code, RAID-3 has a single check disk with parity information. Thus, the reliability overhead for RAID-3 is a single disk, the lowest overhead possible.

Level-4 :- Block Interleaved parity:-

It has a striping unit of a disk block, instead of a single bit as in. RAID-3. Block level striping has an advantage that read request of the size of the disk block can be served entirely by the disk where the requested block resides.

Level-5: Block-Interleaved Distributed Parity:-

RAID-5 improves on RAID-4 by distributing the parity blocks uniformly over all disks, instead of storing them on a single check disk. This distribution has 2 advantages.

1) several write requests could be processed in parallel, since the bottleneck of a unique check disks has been eliminated.
2) Read requests have a higher level parallelism. since the data is distributed all disks, read requests involves all disks, where in systems with a dedicated check disk, the check disk never participates in reads.

Level-6: P+Q Redundancy:-

→ RAID-6 system uses Reed-solomon codes. to be able to recover from upto two simultaneous disk failures.

→ RAID-6 requires. two check disks, but it also uniformly distributes redundant information at the block level as in RAID-5.

FILE ORGANIZATIONS AND INDEXING:-

→ The file of records is an important abstraction in a DBMS, and is implemented by the files and access methods Layer.

→ A file can be created, destroyed, records inserted, and deleted from it, and scan operation.

→ A relation is typically stored as a file of records.

A disk page is a collection of records in a file, disk page keeps track of pages allocated to each file, and records are inserted and deleted from a file, it also keep track.

available space with in pages allocated to the file.

→ The simplest file structure is Heap file (or) an unordered file Records in a heap file are stored in random order across the pages of the file.

A Heap file organization supports retrieval all records (or) retrieval of a particular record specified by its record id (rid).

The File manager must keep track of the files allocated for the file.

→ <u>Index</u> is a data structure that organizes the data records on disk to optimise certain kinds of retrieval operations.

→ We use term <u>data entry</u> to refer to the records stored in an index file. A data entry with a search key value K; denoted as <u>K*</u>, It contains enough information to locate data records with search key value K.

→ There 3 alternatives for what to store as a data entry in the index:

(1) A data entry <u>K*</u> is an actual data record (with search key value K).

(2) A data entry is a <K, rid> pair, where <u>rid</u> is a record id of a data record with search key value K.

(3) A data entry is a <K, rid_list> pair, where rid-list is a list of record ids of data records with search key value K.

TREE STRUCTURED INDEXING

Intuition for Tree Indexes:-

Consider a file of records sorted by "gpa" of students Table. To answer a range selection such as:

" Find all students with gpa higher than 3.0".

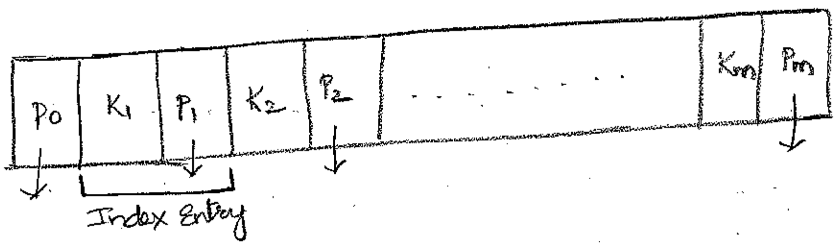
→ Firstly, we must identify such student by doing a binary search of the file and then scan the file from that point on.

If the file is large, (i.e) it contains huge no. of records the binary search can take a lot of time to search, this will be cost-effective process, and the cost is proportional to no. of pages fetched.

→ We can improve this method by using keys and Indexing. One method is to create a second file with one record per page in the original (data) file of the form

< first key on page, pointer to page >.

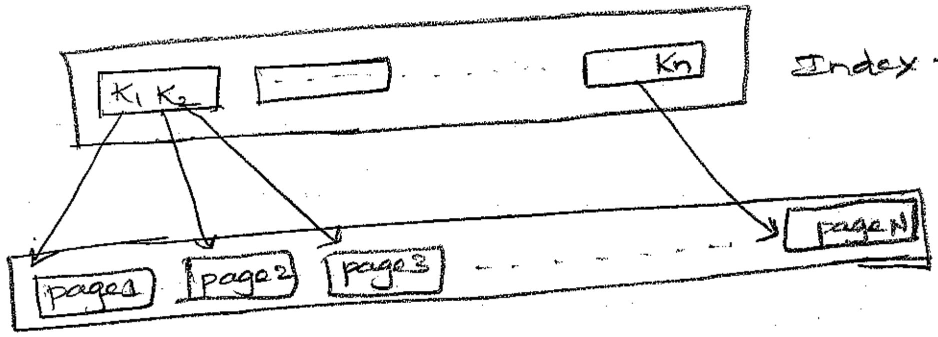
This again sorted by the key attribute (i.e) gpa.



Format of Index page

→ The index entry (or) entry to pairs of the form. <key, pointer>. Each index page contains "one pointer more than the no. of keys". (If 2 keys are there, 3 pointers will be there).

Each key serves as a seperator for the contents of the pages pointed to by the pointers to its left and right.



We can do binary search of the index file to identify the page containing the first key (gpa) value that satisfies the range selection. (i.e. student gpa over 3.0), and follow the pointer to the page containing the first data record with that key value.

We then scan the data file sequentially from that point on to retrieve the other qualifying records.

→ The size of an index entry in the index file is likely to be much smaller than the size of the page and only one such entry per page of data file is exist.

∴ A binary search of an index file is much faster than of a data file.

→ However, a binary search of index file could still costly (or) expensive, if the index file has large no. of entries, so that inserts and deletes to a index file be very taking process. (i.e) expensive process for cpu.

⇒ The potential large size of the index file motivates the tree indexing.

∴ The repeated construction of one level index leads to a tree structure with several levels of non-leaf pages.

5-8

To gain a fast random access to records in a file, we can use an Index structure.

Index structure is associated with a particular search key.

Just like the index of a book or a library catalog, an ordered index stores the values of search keys in sorted order, and associates with each search key the records that contains it.

→ The records in the indexed file may themselves be stored in some sorted order just as Books in a Library.

→ A file may have several indices, on different search keys.

→ If the file containing the records is sequencially ordered, a <u>clustering index</u> is an index whose search key defines the sequencial order of the file.

→ Clustering indizes are also called "<u>primary indices</u>", the term primary index may denote an index on a primary key, but such indices can infact be built on any search key.

→ A search key of a clustering index is often the primary key.

<u>Non-clustering Indizes:</u>— Indices whose search keys specifies an order different from the sequencial order.

→ Non clustering indices are also called secondary indizes.

(or)

An index on a set of fields that includes the primary key called a primary index; Other indexes are called Secondary Indexes.

→ If the index is clustered, the rids in qualifying data entries point to a contiguous collection of records, and we need to retrieve only a few data pages.

→ If the index is unclustered, each qualifying data entry could contain a rid that points to a distinct data page, leading to as many data page I/Os as the no. of data entries that match with the range selection.

→ Two data entries are said to be <u>duplicates</u> if they have same value for the <u>search key field</u> associated with an index.

→ A primary index is guaranteed not to contain duplicates. But index on other fields can contain duplicates, a secondary index can contain duplicates.

→ If we know that no duplicates exists, (that is) we know that search key contains some candidate key, we call such an index as a "<u>unique index</u>".

5-6

Index Data Structures :-

1) Hash Based Indexing

2) Tree - Based Indexing

Hash - Based Indexing :-

We can organize the records using a technique called "Hashing", to quickly find records that have a given search key value.

Ex:- If the employee file records is hashed on the name field, we can retrieve all records about employee name "joe".

→ In this approach the records are grouped in "buckets", where a bucket consists of primary page and additional pages linked in a chain.

→ The bucket to which a record belongs, can be determined by applying a special function called a Hash function, to the search key.

→ Given a bucket number, a hash based index structure allows us to retrieve the primary page for the bucket in one or two disk I/O's.

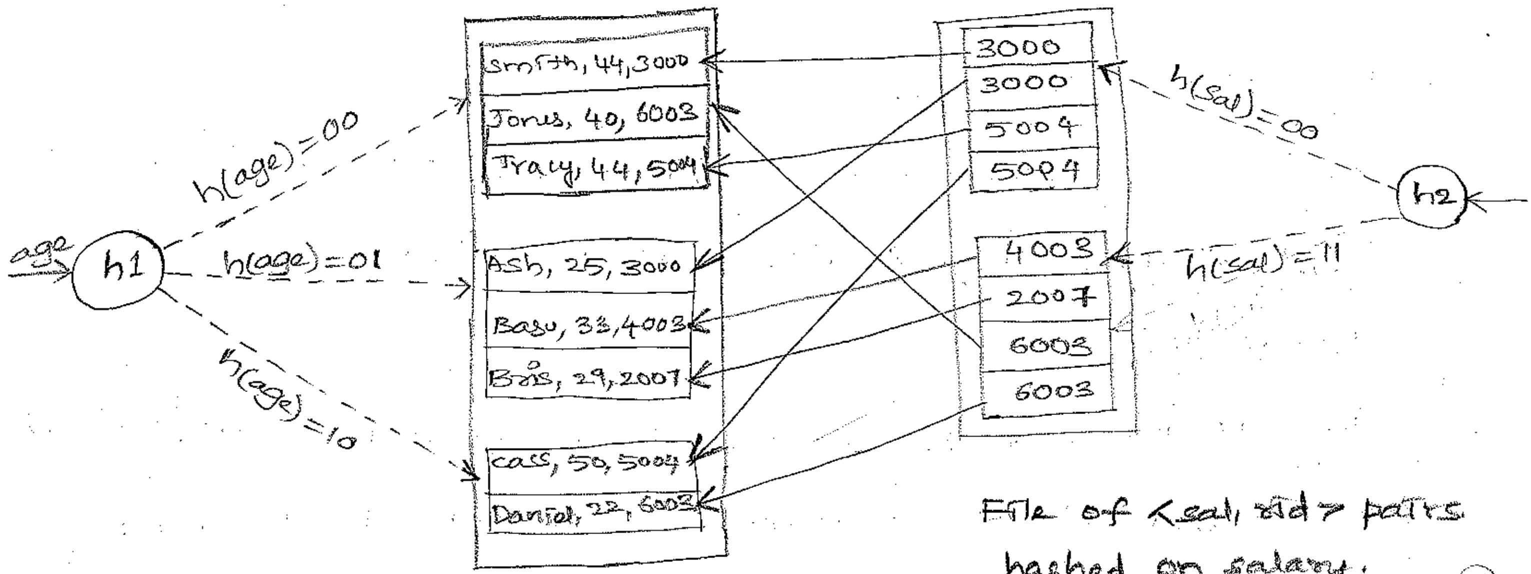
Insert :-

On Insert, the record is inserted into the appropriate bucket, with 'overflow' pages allocated as necessary

Search :-

To search for a record with a given search key value, we apply the hash function to identify the bucket to which such records belongs to., and look at all pages in that bucket.

→ If we donot have the search key value for the record, for Ex:- the index is based on sal and we want records with a given age value, we have to scan all pages in a file



File of <sal, rid> pairs hashed on salary.

Employee file Hashed on age

Index - Organized File Hashed on age, with Auxiliary Index on salary

→ In the above Hash Indexing, the data is stored in a file that is hashed on age; the data entries in first index file are the actual data records.

→ Applying the Hash function to the age field identifies the page that the record belongs to.

the Hash function h for this example is:

It converts the search key value to its binary representation and uses the two least significant bits as the bucket identifier.

→ Note that the search key for an index can be any sequence of one or more fields, and it need not uniquely identify records.

Ex:- Salary Index, two data entries have same search key "6003".

5-D

TREE BASED INDEXING :-

→ In this method, the data entries are arranged in sorted order by search key value, and a hierarchical search data structure is maintained that directs searches to the correct page of data entries.

→ This structure allows us to efficiently locate all data entries with search key values in a desired range.

→ All searches begin at the topmost node, called the Root, the contents of pages in non-leaf level directs searches to correct leaf page, which is a lowest level of the tree. (i.e) Leaf level.

→ The leaf level contains the data entries.

For Ex:- The employee records.

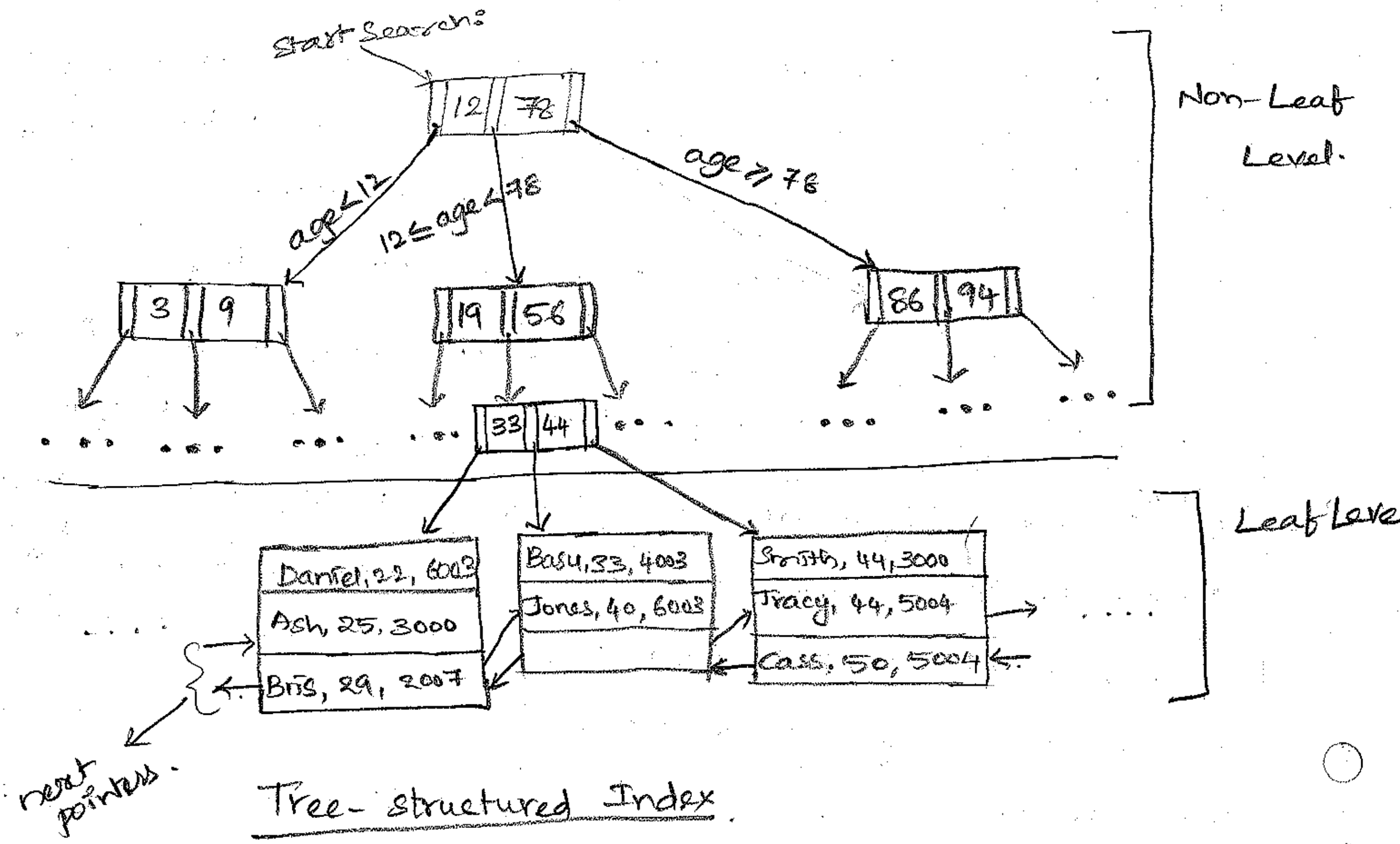
→ The Non-Leaf pages contain Node pointers seperated by search key values.

*** → The Node pointer to the left of a key value K points the subtree that contains only data entries less than K

→ The node pointer to the right of a key value K points to a subtree that contains only data entries greater than or equal to K.

Ex:-

• If there is a record with age = 22, and we want to add some additional record, with age less than 22, that would appear in leaf pages to the left of 22. and records with age greater than 50, would appear in leaf page to the right of 50.



Tree-structured Index.

→ The above tree structured index, have employee records organized with search key "age".

Each node in this figure (Leaf or Non-leaf) is a physical page and retrieving a node involves a disk I/o.

→ All leaf pages are maintained in doubly-linked list, thus for efficient searching and retrieval of qualifying records.

→ The tree structure must be balanced, to keep the tree balanced, we use B+ tree.

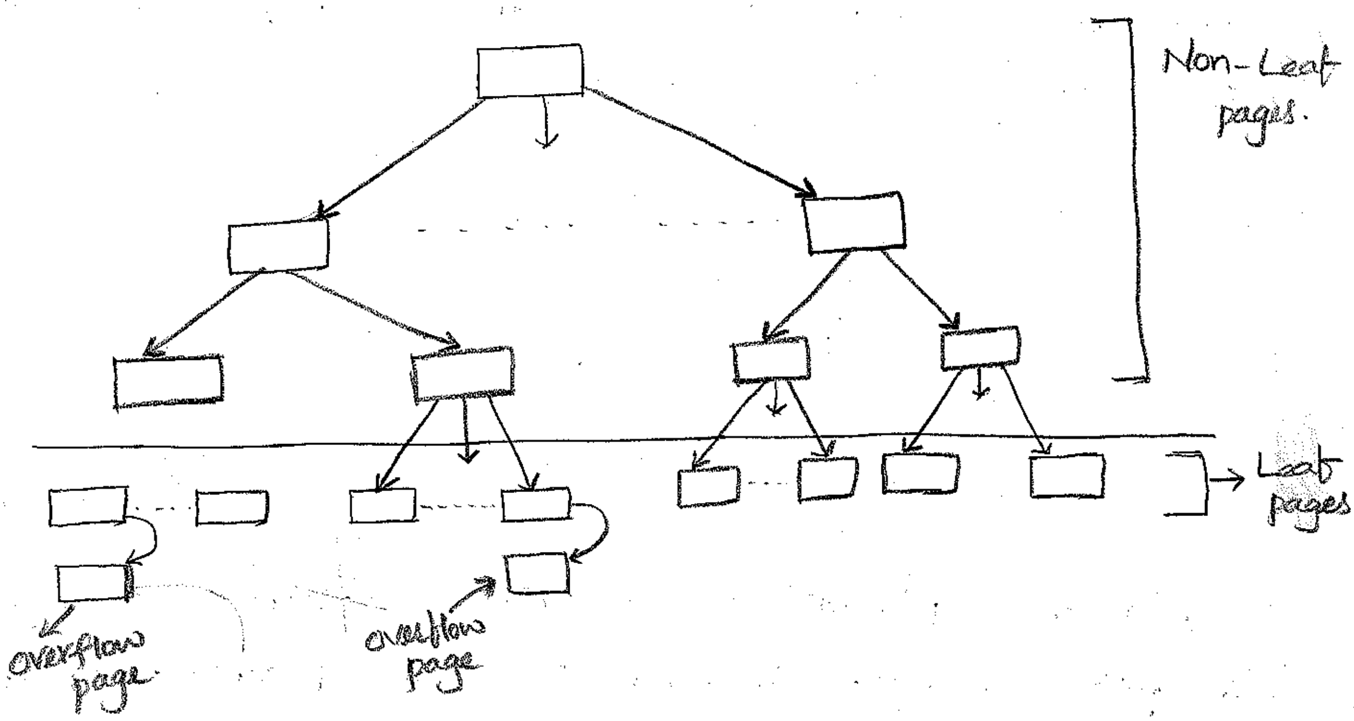
B+ tree is an index structure that ensures that all paths from the root to leaf in a given tree are of same length; (i.e) the structure is always balanced in height.

→ The height of a balanced tree is the length of the path from root to leaf.

→ The Height of above tree is "**3**".

5-E

Indexed Sequencial Access Method (ISAM)



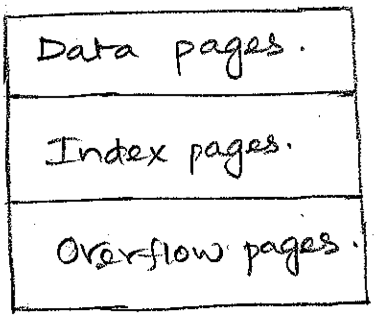
ISAM Index Structure

The data entries. of the ISAM index are in the leaf pages of the tree and additional overflow pages chained to some leaf page.

→ Database systems carefully organizes the layout of pages so that page boundaries correspond closely to the physical characteristics. of the underlying storage devices.

→ ISAM is static structure, except for overflow pages. and facilitates low-level organization

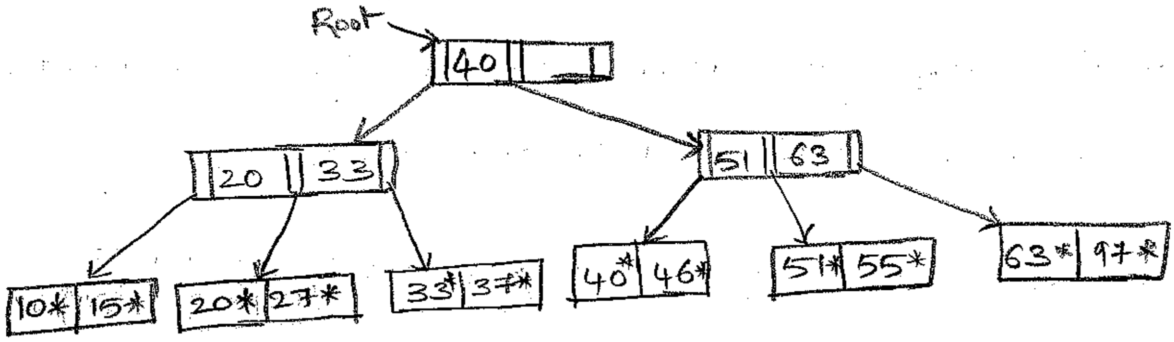
→ Each tree node is a disk page, and all the data resides in the leaf pages. When the file is created, all leaf pages. are allocated sequencially and stored on the search key value, as mentioned in the above figure, and the non-leaf pages are then allocated. If there are several inserts to the file subsequently, so that more entries. are inserted into a leaf than will fit onto a single page, additional pages are needed because the index structure is static. These additional pages are allocated from an overflow area.



Page allocation in ISAM

→ The basic operations insertion, deletion, search are all quite straight forward. For an equality selection search, we start at the root node and determine which subtree to search, by comparing the value in the search field of the given record with the key values in the node.

→ For a range query, the starting point in the data Level (or) Leaf Level is determined. and the data pages are then retrieved sequencially.



Sample ISAM Tree

→ For inserts and deletes, the appropriate page is determined as for a search, and the record is inserted or deleted with overflow pages added if necessary.

→ In the above figure, the root node have value "40"

5-F

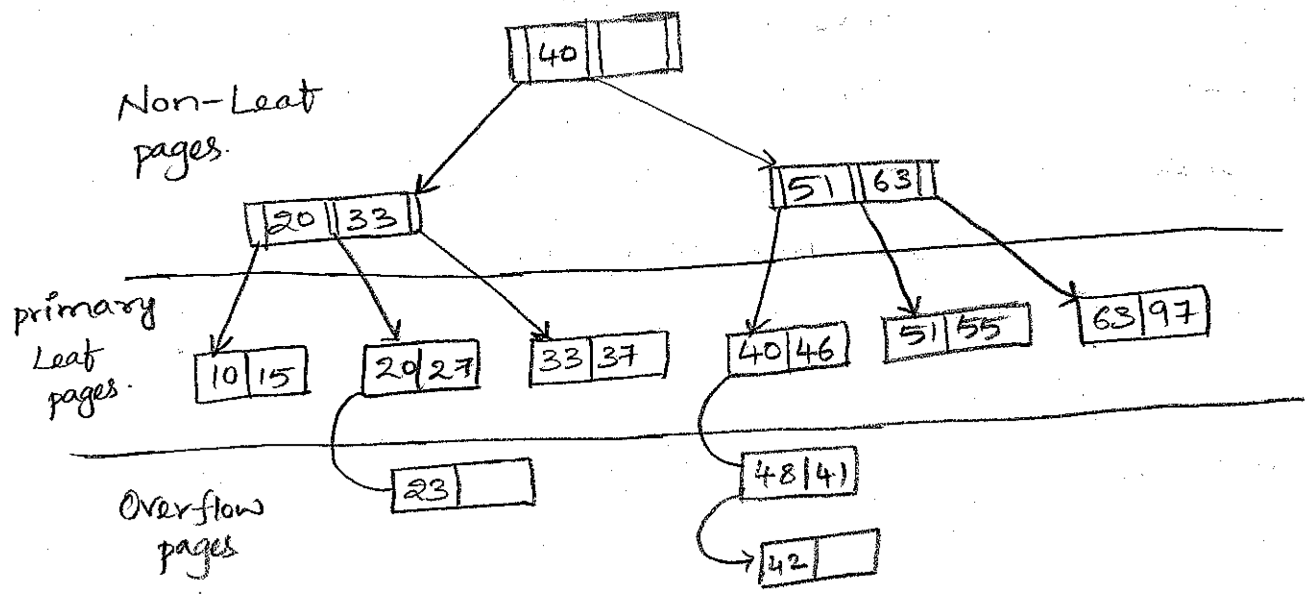
- All searches begins at the root. ~~For~~

For Ex:- We want search 27 record.. (27 is a key value here).

→ We start at the root and follow the left pointer, since 27<40:

→ We then follow the middle pointer, since 20<=27<33.

For a range search, we find the first qualifying data entry "27".



ISAM tree with overflow pages.

→ In the above figure, the primary leaf pages are allocated sequentially, the leaf pages dont have the 'Next Leaf page' pointer.

→ Each leaf page have two entries in above figure, If we now insert a record with key value 23, the entry 23, belongs to second data page (i.e) $\boxed{20|27}$, it is already full and has no space to insert.

→ We deal with this situation, by adding an overflow page and putting 23 in overflow page.

→ chains of overflow pages can easily developed as shown in the figure 48, 41, 42 leads to overflow chain of 2 pages.

Deletion in ISAM ~~tree~~:-

The deletion of entry $k*$ is handled by simply removing the entry. If this entry is on an overflow page, and that overflow page becomes empty, the page can be removed

→ If the entry is on the primary page, and the deletion makes primary page empty, the simplest approach is simply leave the empty primary page as it is.

→ It serves as a placeholder for future insertions and possibly for non-empty overflow pages, because we donot move records from the overflow page to the primary page when deletions on the primary page create space. Thus the no. of primary pages are fixed at the file creation time.

Problems with overflow pages:-

Once the ISAM file is created, inserts and deletes affects only the contents of leaf pages.

A consequence of this design is that long overflow chains could develop, if a no. of inserts made to a single leaf.

→ These chains can significantly affect the time to retrieve a record because overflow chain has to be searched as well, when the search gets to this leaf.

→ To deal with this problem, the tree is initially created so that 20 percent of each page is free. However, once the page free space is filled with inserted records, unless the space is freed again through deletes, overflow chains can be eliminated only by a complete reorganization of ~~a~~ the file.

ISAM - Concurrent Access and Locking :-

→ The fact that only leaf pages are modified also has an important advantage with respect to concurrent access.

→ When a page is accessed it is typically "Locked" by requests to ensure that it is not concurrently modified by other users of the page.

→ To modify a page, it must be locked in exclusive mode which is permitted only no one else holds a lock on the page.

(*)→ Locking can lead to "queues" of users (transactions) waiting to get access to a page and these QUEUES can be a significant performance bottleneck, especially for heavily accessed pages near the root of an index structure.

→ In ISAM pages, the index-level pages are never modified we can safely omit the locking step.

→ Not Locking index-level pages is an important advantage of ISAM over a dynamic structure like a B+ tree.

*** → ISAM are preferable only when.

- the data distribution and size are relatively static, (ie) the overflow chains are rare.

B+ Trees: A DYNAMIC INDEX STRUCTURE

→ ISAM is a static index structure.

B+ tree is a dynamic index structure.

→ ISAM static structure suffers problem of long overflow chains, that leads to poor performance.

This problem is avoided by B+ trees.

→ A B+ tree search structure, which is widely used is a balanced tree in which the internal nodes direct the search and the leaf nodes contains the data entries.

→ B+ tree structure grows and shrinks dynamically. To retrieve all leaf pages efficiently, we have to link them using page pointers, by organizing them in ~~to~~ a doubly linked list, we can easily traverse the sequence of leaf pages in either direction.

Characteristics of B+ tree :-

- Operations such as inserts, deletes on the tree keep it balanced.
- A minimum occupancy of 50% is guaranteed for each node except the root.

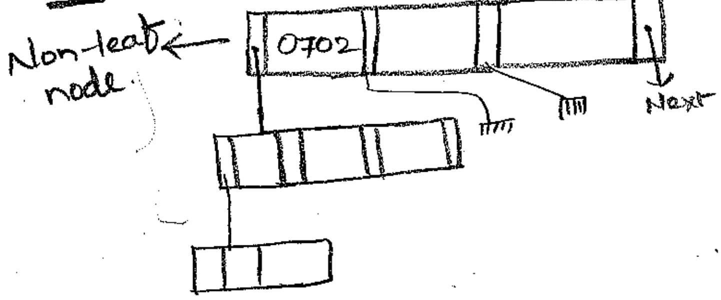
  (i-e) All Index blocks have to be atleast half full.

Ex:- • Out of 4 pointers, 2 pointers have to be points to valid index blocks.

• Out of 3 pointers, 2 pointers have to be points to valid index blocks.

→ Searching for a record record requires just a traversal from the root to the appropriate leaf.

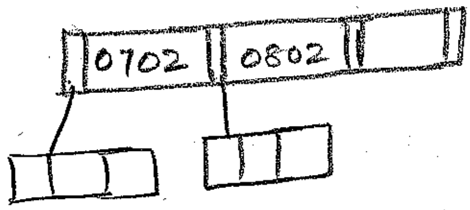
Ex:- 1)



Out of 4 pointers, 1 pointer is pointed to other index block, ∴ which is not allowed.

(⏚ represents NULL).

②



Out of 4 pointers, 2 pointers are pointed to index blocks, so, which is allowed.

→ Height of a tree is the no. of edges in between root node to leaf node.

→ A tree with single index level (Root node) and single leaf level the height of the tree is 1.

→ A tree with only root node, the height = 0.

Order of a B+ tree (d):-

The value d is a parameter of the B+ tree, every node contains m entries.

where $d \leq m \leq 2d$.

Order of a tree is a measure of capacity of a tree node.

→ The root node is the only exception to this require-ment on the no. of entries.

For Root: $1 \leq m \leq 2d$.

Comparison of B+ tree and ISAM:-

→ B+ tree typically maintains 67% of occupancy in space.

→ B+ trees are preferable to ISAM, because inserts are handled gracefully without overflow chains.

→ ISAM is prefered to B+ trees, when

(1) the leaf pages are allocated in sequence.

(2) The locking overhead of ISAM is less than that of B+ trees.

<u>Structure of B+ tree:-</u>

The Format of Node in <u>B+</u> tree is same as for ISAM

→ Non-leaf pages with <u>m</u> index entries contains <u>m+1</u> pointers to <u>children</u>.

→ Pointer <u>P_i</u> points to a subtree in which all the key values K are such that <u>"$K_i < K < K_{i+1}$"</u>.

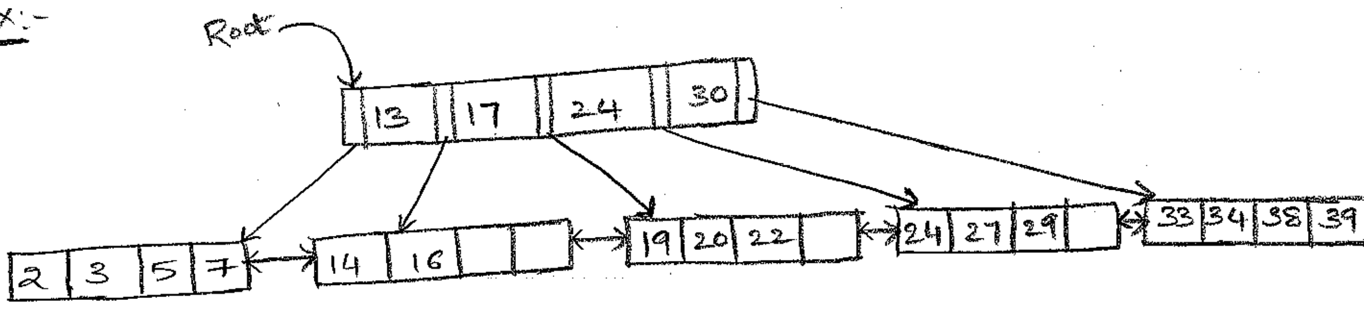
→ (i.e) (i) <u>P_0</u> points to a tree in which all the key values are less than K_1. <u>and</u>,

(ii) <u>P_m</u> points to a tree in which all the key values are greater than or equal to K_m.

iii) For leaf nodes the entries are denoted by <u>K*</u>.

→ <u>B+ tree</u> leaf nodes contains <u>data entries</u>, the leaf nodes (or) Leaf pages are chained together in a doubly Linked List.

<u>Ex:-</u>



<u>B+ tree</u>, order d=2, height=1.

In the above figure the left side of pointer points to Less than 13. and right side of pointer 30, points to greater than 30 (i.e) 33, 34, 38, 39 leaf node.

The non-leaf node contains m index entries with m+1 pointers;

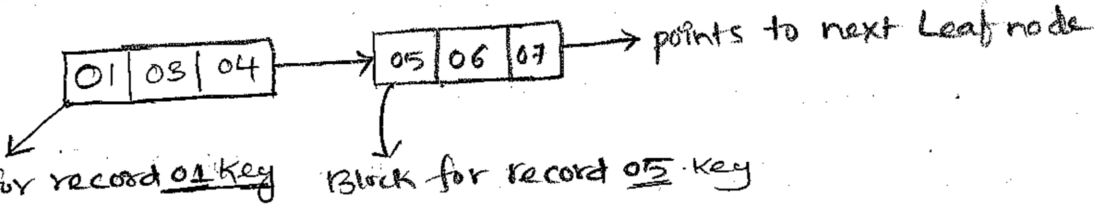
Ex:- For 2 index entries, these must be 3 pointers,

For 3 index entries, there must be 4 pointer

blocks are points to atleast 2 other nodes.

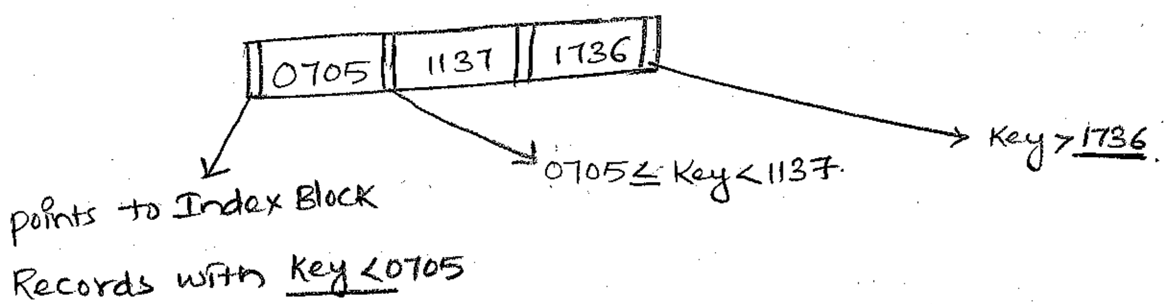
* The actual data entries (i.e files or records) which are attached to leaf nodes.

Structure of a leaf node:-



The leaf nodes points to the actual Block records.

Structure of Non-Leaf node:-



→ The Non-leaf nodes are also called as Internal nodes.

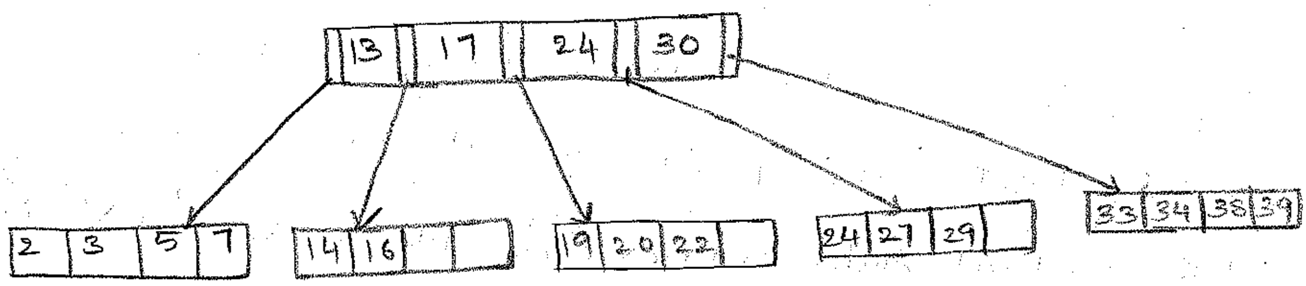
The leaf nodes are called as "External nodes".

B+ Tree Operations:-

1) Search:-

- The searching can be done in either linear search or binary search. depends on the no-of. entries in the code.
- Let us consider a B+ tree with the order d=2. That is, each node contains entries between 2 and 4.
- Each non-leaf entry is a < keyvalue, nodepointer >. pair.

→ At the leaf level, the entries are data records. that we denoted by K*.



B+ tree with order-2

→ To search for entry 5, we follow the left most child pointer, since $5<13$.

→ To search for the entries 14 and 15, we follow the second pointer $13 \leq 14 < 17$. and $13 \leq 15 < 17$.

→ To find 24, we follow the fourth child pointer, since "$24 \leq 24 < 30$".

INSERT:-

→ The algorithm for insertion takes an entry, finds the leaf node, where it belongs, and inserts it there.

→ The insertion procedure results in going down to the leaf node where the entry belongs, placing the entry there, and returning all the way back to the root node.

→ If a node is full and it must be split.

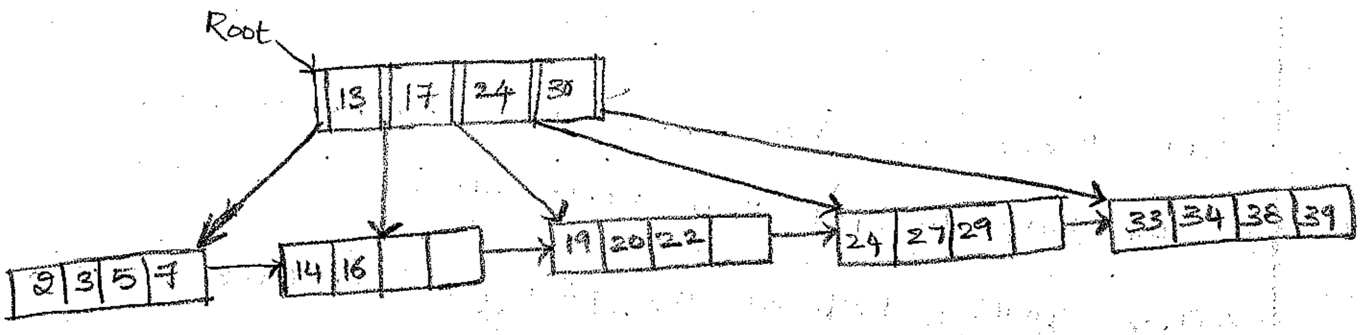
→ When the node is split, an entry pointing to the node created by the split must be inserted into its parent.

→ This entry is pointed to by the pointer variable "newchildentry".

→ If the old root split, a new root node is created and the height of the tree increases by 1.

Example:-

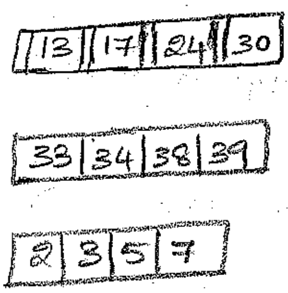
Let us take a tree, below.



B+ tree with order d=2.

In the above tree, there are full nodes, (i.e) there is no place to insert any key.

Full Nodes are :-

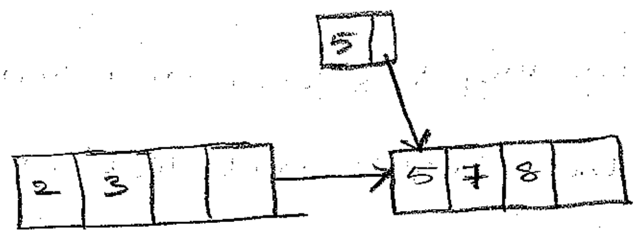


We can not insert any value in these nodes, if we want to insert we have to split the nodes.

* If we insert 8, it belongs to the left most leaf, (i.e) $\boxed{2|3|5|7}$. But this node is already full.

So, this insertion causes a split of a leaf node, and the node values must be adjusted.

(i.e) we have to adjust the nodes that are atleast half-full



split pages during the insert of entry 8

After inserting 8, the node is splitted into 2 nodes.

Now we have to copy the value 5 to parent node.

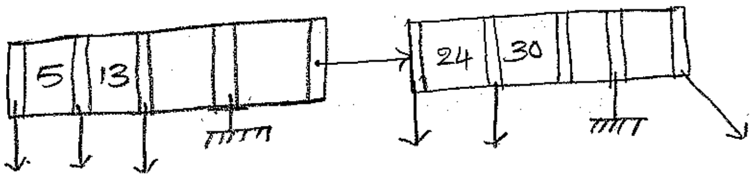
(i.e) | 13 | 17 | 24 | 30 |

But there is no space to copy up 5 in the above node

So, we have to split that node also.

After splitting, half of the pointers must have to point other nodes. in non-leaf nodes.

So, the split nodes will be

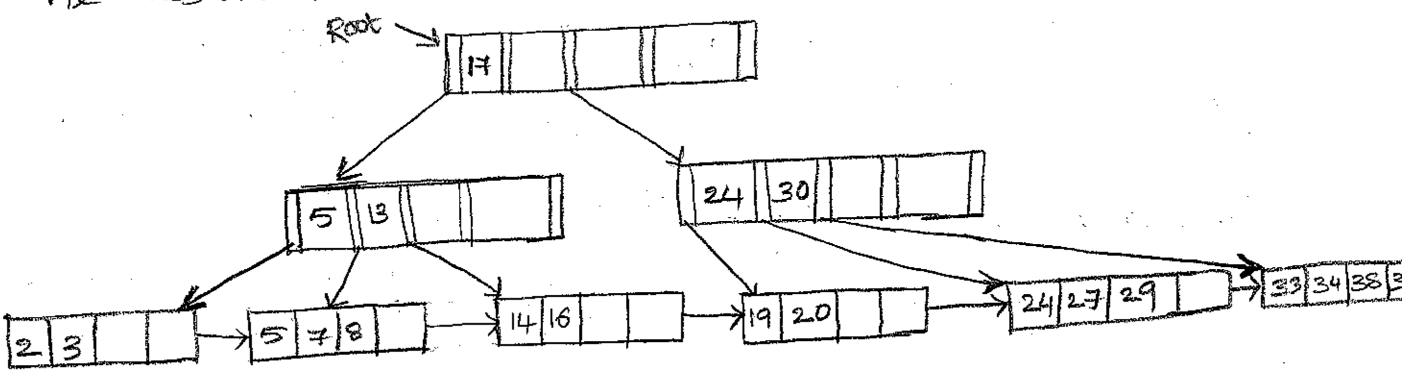


There are two nodes in non-leaf node. Before splitting of these node, the node is actually a root node

→ So, we have to set a root node for tree, we insert 17, in node | 5 | 13 | 17 | |

→ The root node now have the value 17, (i.e) the 17 is pushup to the parent node.

The resultant tree will be:



The difference in handling leaf level and index level splits. arises from the B+ tree requirement that all data entries. k' must reside in the leaves. This requirement prevents us from 'pushing up 5' and leads to slight redundancy of having some key value appearing in the leaf level as well as in some index level.

In dealing with the index levels, we have more flexibility, and we 'pushup 17' to avoid two copies of 17 in the index levels.

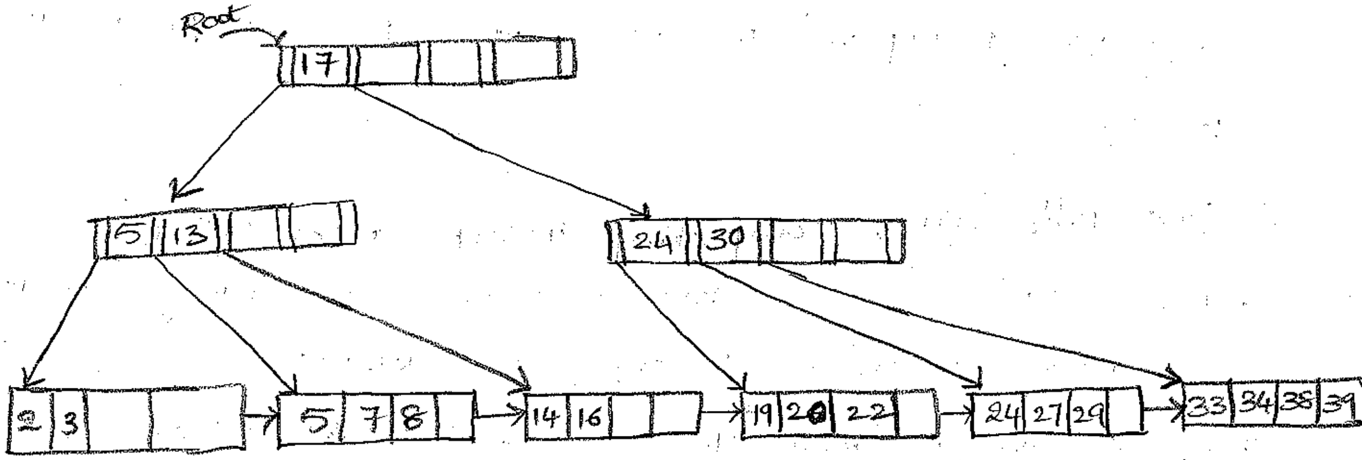
→ DELETE:-

- The algorithm for deletion takes an entry, finds the leaf node, where it belongs and deletes it.
- We usually go down to the leaf node, where the entry belongs, removes the entry from there and return all the way back to the root node.
- Occasionally, a node is at minimum occupancy before the deletion, and the deletion causes it go below the occupancy threshold.
- When this happens, we must either redistribute the entries from an adjacent sibling (or) merge the node with a sibling to maintain the minimum occupancy.
- If entries are redistributed between the two nodes, their parent node must be updated to reflect:

  The key value in the index entry pointing to the second node must be changed to the lowest search key in the second node.
- If two nodes are merged their parents must be updated to reflect this by deleting the entry for the second node, this index entry is pointed to, by the pointer variable (old child entry) when the delete call returns to the parent node.

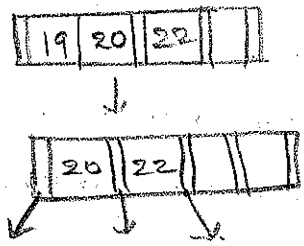
• If the last entry in the root node is deleted in this manner because one of its children was deleted, the height of the tree is decreased by 1.

<u>Let us take an example using the last B+ tree</u>



→ To delete an entry 19, we simply remove it from leaf page on which it appears. and we are done with the deletion process because the leaf still contain two entries.

→ (i.e)

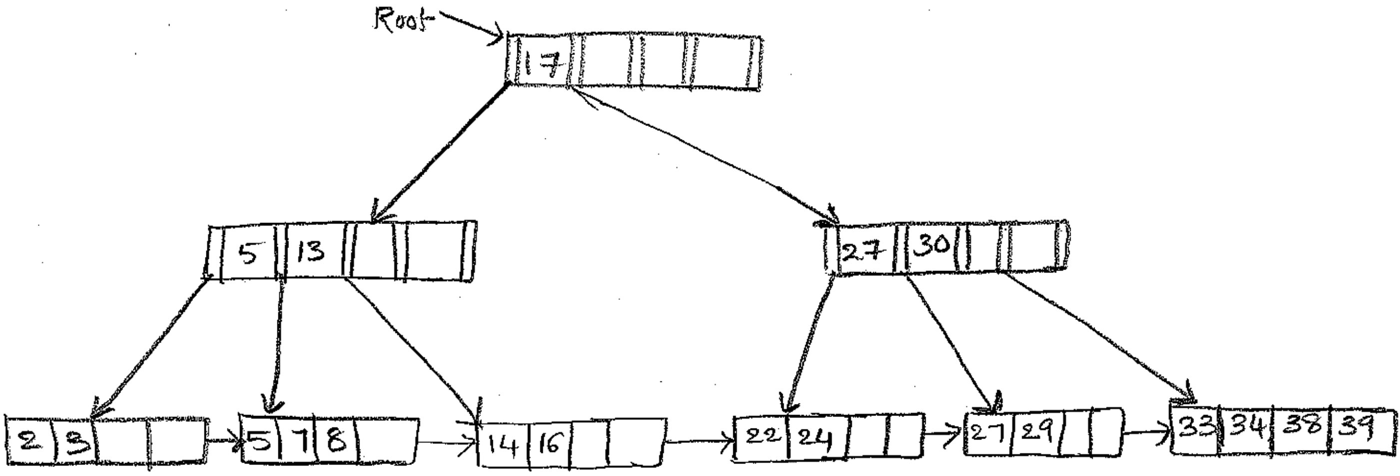


<u>after deletion</u>:- , (there is no need to redistribute the node (3 pointers and 2 entries it satisfies half of node is full property)

→ But if we delete 20, the node contains 22, only. (i.e) the node only contains one entry 22 after deletion, the sibling of leaf node (i.e. the next node) contains 3 entries, we can therefore redistribute the nodes.

→ we move entry 24 to the leaf page that contained 20, and copy the new splitting key (27), which is a new <u>low key value</u> of the leaf from which we borrowed 24 will go into the parent.

5V

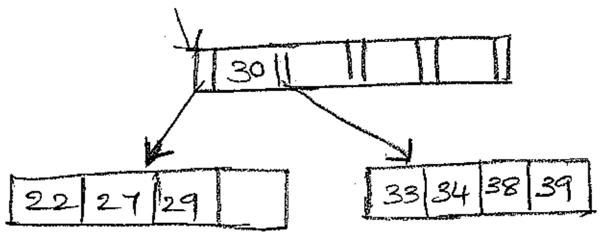


Suppose, that we now delete an entry 24, the affected leaf contains only one entry 22. after deletion and the only sibling contains just two entries 27 and 29.

∴ we can not redistribute entries.

However these two leaf nodes together contain only 3 entries and can be merged.

→ While merging we can toss the entry 27 (and its pointer) in the parent, which pointed to second leaf page, because the second leaf page is empty after the merge and can be discarded.



after deleting <u>24</u>.

Consider the merging of two non-leaf nodes, and the sibling to be merged containing only three entries, and they have a total of five pointers to leaf nodes. To merge the two nodes, we also need to pulldown the index entry in their parent that currently discriminates between these nodes.

5-M

HASH-BASED INDEXING

The Hash-Based Indexing uses Hash-function, which maps values in a search field into a range of bucket numbers to find the page on which a desired data entry belongs.

3 types of Hashing techniques.

1) Static Hashing
2) Extendable Hashing
3) Linear Hashing

Hash Based Indexing techniques. can not support range-searches, where as tree-based Indexing can supports the range searches.

Static Hashing :-

In static hashing scheme, the pages containing the data can be viewed as Buckets, with one primary page and additional overflow pages per bucket.

→ A file consists of buckets 0 through N-1, with one primary page per bucket initially.

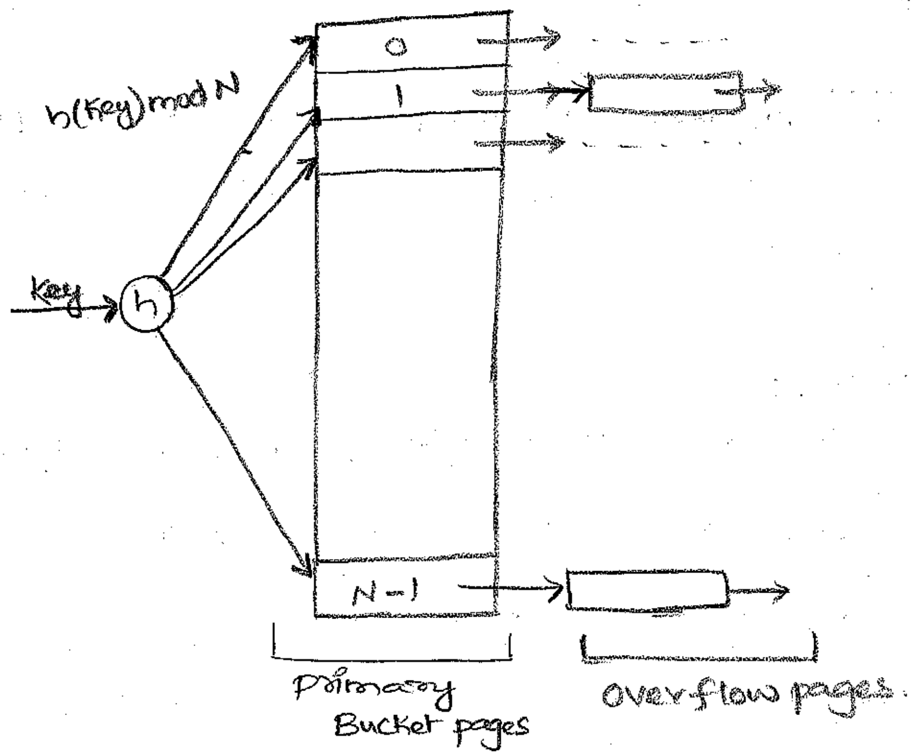
→ Buckets contain Data entries, to search for a data entry, we apply a hash function h to identify the bucket to which it belongs to, and then search this bucket.

→ To speed the search of a bucket, we can maintain data entries in sorted order by search key value

→ To Insert a data entry, we use a hash function to identify the correct bucket and then put the data entry there. If there is no space for this entry, we allocate a new overflow page, put the data entry on this page, and add the page to overflow chain of the bucket.

→ To delete a data entry, we use the hash function to identify the correct bucket, locate the data entry by searching the bucket, and then remove it.

* → If this data entry is last in the overflow page, the overflow page is removed from the overflow chain of the bucket. and added to the list of free pages.



—: STATIC HASHING :—

→ The hash function is the important component of hashing approach. It must distribute the values in the domain of search field uniformly over the collection of buckets.

→ In static Hashing, the no. of buckets in file is known when the file is created, the primary pages can be stored on successive disk pages.

5-N *→ Hence a search ideally requires one disk I/O &
Insert and delete requires two I/Os.
(i.e. one I/o is for read, other I/o is for write), In search process we are only reading the value, no updations are done in search, Hence it only take one I/o).

→. The cost would be high with the overflow pages, as. the files grows long with overflow chains, it is not easy task to search for a value.

→ Since, searching a bucket requires to search all pages in its overflow chain, this leads to performance degradation.

→ We can avoid this problem, by keeping pages 80% full, but only if the file does not grow too much, but in general the only way to get rid of overflow chain is to create a new file with more buckets, but in static hashing the no. of buckets are fixed. Hence it is the main problem.

Disadvantages with static Hashing:-

1) If the file shrinks greatly, a lot of space is wasted
2) If a file grows a lot, long overflow chains can be develop, which will resulting to poor performance.

Comparison of IASM and static Hashing:-

Like ISAM, static Hashing is also sufferred from overflow chains if they grow long incase of insertions to the same page.

Extendible Hashing :-

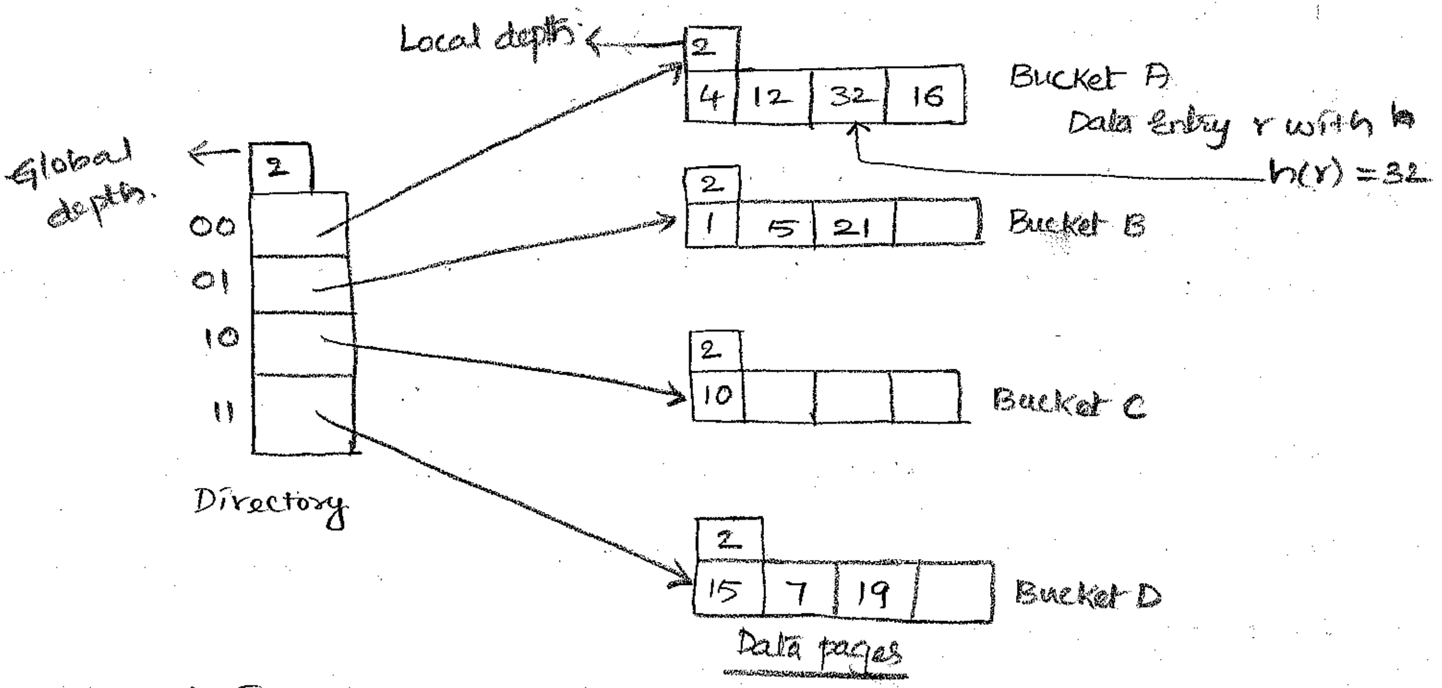
→ Extendible Hashing uses a directory of pointers to buckets. These directory of pointers can be used to avoid the problem of overflow chains.

→ In static Hashing if a bucket is full and to insert a new data entry, we need to add a overflow page, if we do not want to add the overflow page, one solution is to reorganise the file at this point by doubling the no. of buckets, and redistributing the entries across a new set of buckets.

Issues with above process:-

During the above process the entire file has to be read and twice as many pages have to be written to achieve the reorganization.

*** This problem can be overcome by using Directory of pointers to buckets and double the size of the no. of buckets by doubling just the directory and splitting only the bucket that is overflowed.



: Extendible Hashing :-

5-0 → The directory consisting of an array of size 4. with each element being a pointer to a bucket

→ To locate a data entry., we apply a hash function to the search field and take last 2 bits of its binary representation. to get the number between <u>0</u> and <u>3</u>. (0-3).

→ Each bucket can hold 4 data entries, the pointer in this array position gives us the desired bucket.

<u>Ex:-</u> ∴ To locate a data entry with hash value 5, it binary format <u>5 = 101</u>, we look at the directory element <u>01</u> and follow the pointer to the data page, (i.e) Bucket B in the above figure

→ <u>Insert -13 :-</u>

The hash value -<u>13</u> , we examine directory element 01 and go to the page containing the data entries 1, 5, 21. since the page has space for additional data entry, we are done after we insert the entry.

<u>Before Insert-13</u>

2			
1	5	21	

Bucket B

<u>After Insert-13</u>

2			
1	5	21	13

Bucket B

→ <u>Insert - 20</u> :-

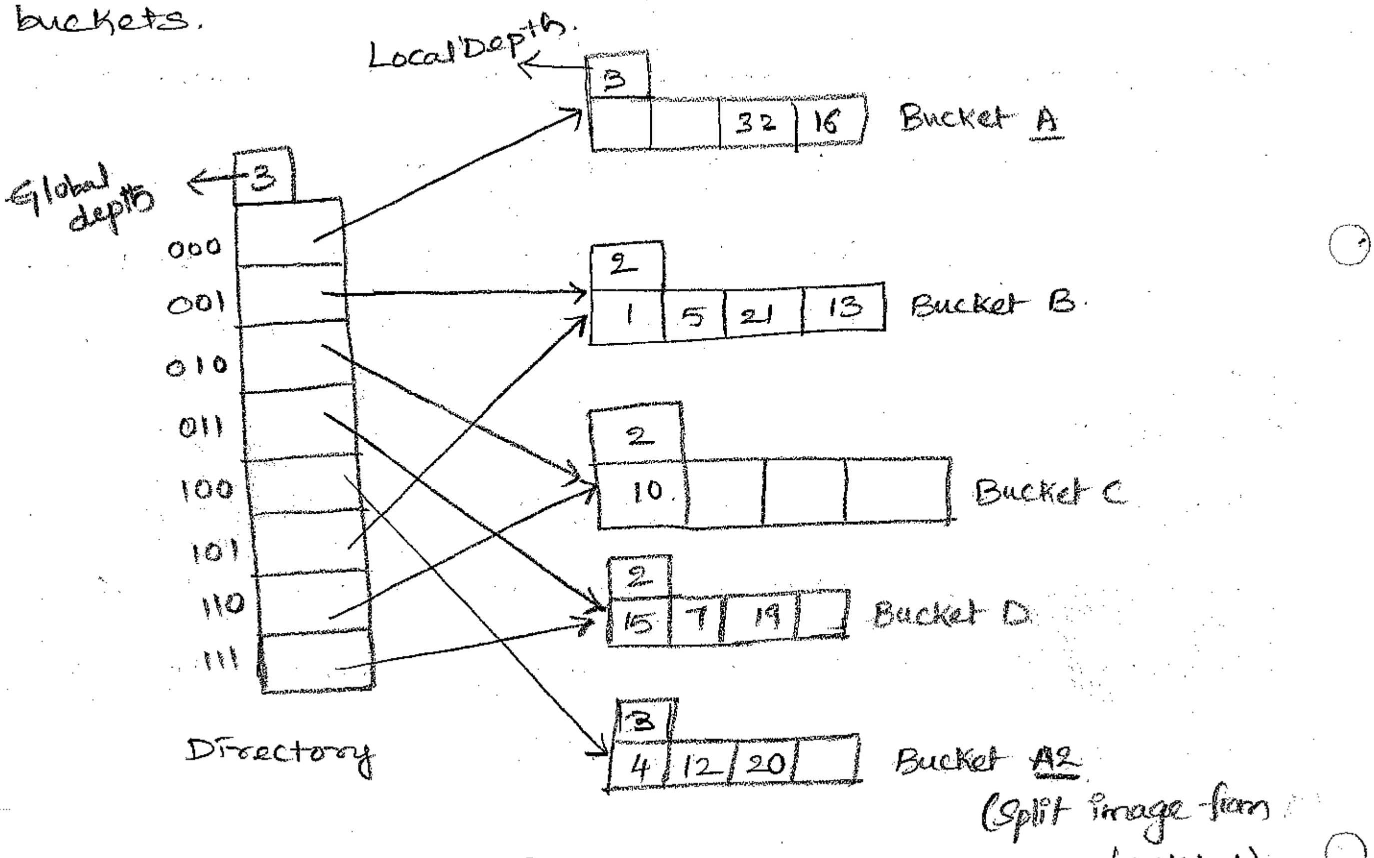
. Binary representation of 20 = 101<u>00</u>

The last 2 digits are <u>00</u>, which belongs to <u>Bucket A</u>, and which is full.

So, we must split the bucket by allocating a new bucket and redistributing the contents including the new entry to be inserted. across the old bucket and split bucket.

* → To redistribute entries across old bucket and split one we consider the last 3 bits of $\underline{h(r): 100}$

In the above 3 bits, the last 2 bits indicating a data entry that belongs to one of these two buckets, and the $\underline{3^{rd}\text{ bit}}$ discriminates between these buckets.



After Inserting 20 (ie) $\boxed{h(r)=20}$

In the above figure, the bucket A splitted in Bucket A & A2. and 20 is inserted on A2.

→ Here we are doubling the directory. The elements that differs only in the 3^{rd} bit from the end are said to correspond:

Corresponding elements of the directory point to the same bucket with the exception of elements corresponding to split bucket.

5-8 In the above example bucket 0 was split; so new directory element 000 points to one of the split versions and new element 100 points to other.

∴ Doubling the file require allocating a new bucket page, writing both this page and the old bucket page that is being split, and doubling the directory array.

→ The directory is likely to be much smaller than the file itself because each element is just a page-Id, and can be doubled by simply copying it over and adjusting the elements for the split bucket.

Global Depth and Local depth:-

In this Extendable Hashing. the result of applying a hash. function h as a binary number and interpret last "d" bits where 'd' depends on the size of the directory, as an offset into directory.

→ In the above example, initially d=2, because we only have 4 buckets.

After the split d=3, and we now have eight buckets.

→ When distributing entries across a bucket and its split image, that should be done based on the d'th bit.

The number 'd' is called "global depth" of the hash file, and it is kept as a part of the header of the file, in our example global depth is top of the directory. It is used every time, we need to locate a data entry

Insert-9:-

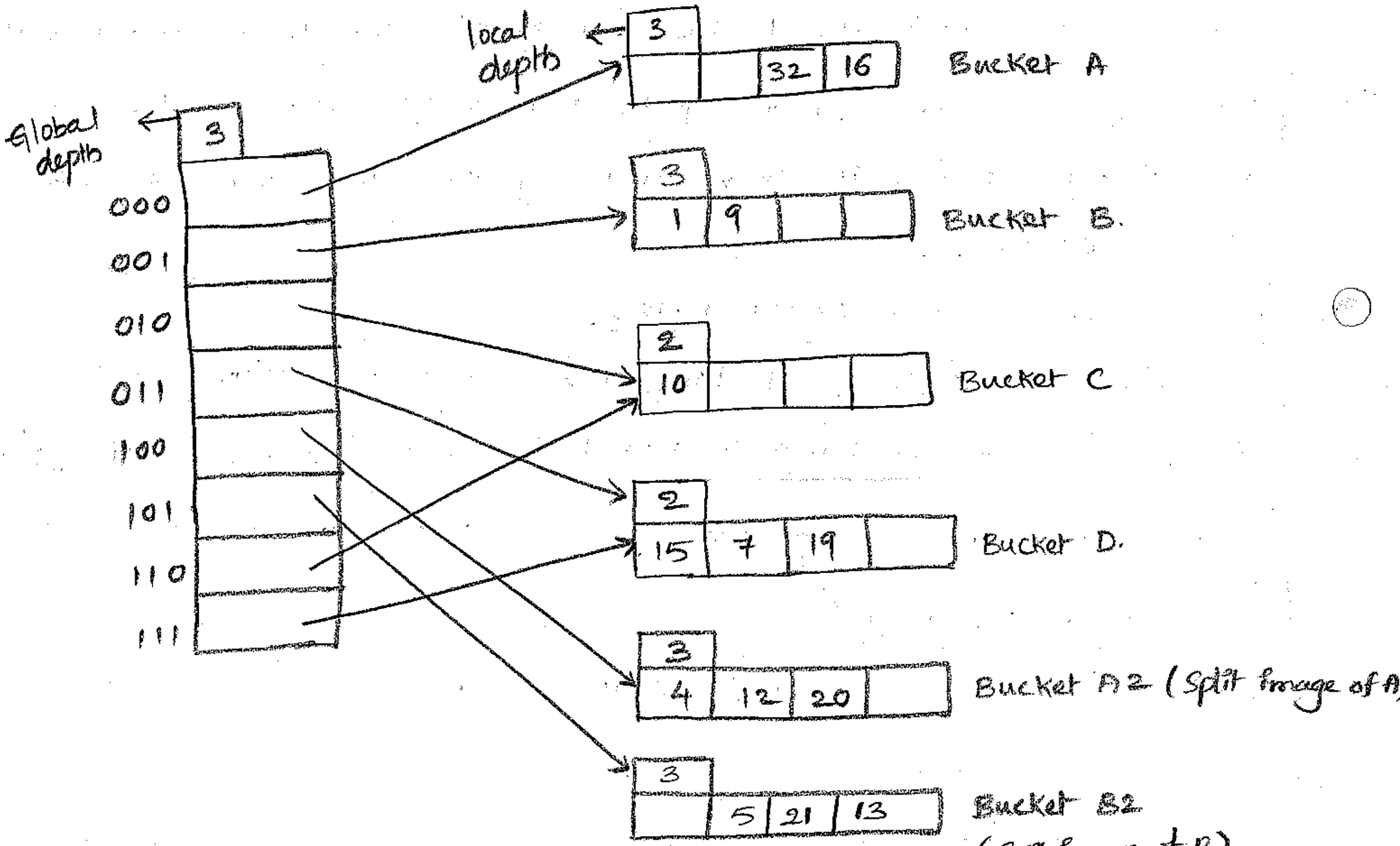
Binary Representation of 9 is 1001

The last 2 digits are 01, and it belongs to Bucket B, but Bucket B is already full.

So, we have to split the bucket B.

Note:- It is important to note that, the splitting bucket
*.
*. does not always necessiates directory doubling.
*.

→ So, insert 9, we split the bucket B. and using directory elements 001 and 101 to point to the bucket B and its split image B2.



After Inserting '9' (ie) h(v)=9

→ To determine whether the directory doubling is needed or not, we need to maintain a Local depth for each bucket.

- If a bucket whose local depth is equal to global depth is split, then the directory must be doubled.

→ Initially all local depths are equals to the global depth.

*→ We increment the global depth by 1 each time when the directory doubles.

5A We increment the local depth by 1 when the bucket splits and we assign this incremented local depth value to its newly created split image.

Deletion :-

For deletes, the data entry is located and removed. If the delete leaves the bucket empty, it can be merged with its split image.

Merging the buckets decreases the local depth.

→ For each directory element points to the same bucket as its split image, we can half the directory and reduce the global depth.

Issues with collisions:-

→ 'Collision' means data entries with same hash value.

→ Collisions caus problem and must be handle specially; when more data entries than will fit on a page have the same hash value, we need overflow pages.

Linear Hashing:-

→ Linear Hashing a dynamic hashing technique, like Extendible Hashing, adjusting gracefully to inserts and deletes.

→ In contrast to extendible Hashing, it does not require a directory, and deals naturally with the collisions; and offers a lot of flexibility with respect to timing of bucket splits.

(ie) allowing us to trade off slightly greater overflow chains for higher average utilization.

→ **Problem with Linear Hashing:-**

If the data distribution is very skewed (non-uniform) however, overflow chains could cause Linear Hashing performance to be worse than that of Extendible Hashing

Process of Linear Hashing:-

The scheme utilizes a family of Hash function h_0, h_1, h_2 with the property that function range is twice that of its predecessor.

(ie). If h_i maps a data entry into one of M buckets, h_{i+1} maps a data entry into one of $2M$ buckets.

→ Such a family typically obtained by choosing a hash function h and initial number N of bucket, and defining

$$\therefore \boxed{h_i(\text{value}) = h(\text{value}) \bmod (2^i N)}$$

If N is choosen to be power of 2 (ie) 2^N then we apply h and look at the last d_i bits;

d_0 is the no. of bits needed to represent N, and.

$\boxed{d_i = d_0 + i}$.

∴ We choose h to be a function. that maps a data entry to some integer.

→ Suppose that, we set the initial number N of buckets to be 32.

(i.e) $2^i = 32 \Rightarrow \boxed{2^i = 2^5}$

Hence $\boxed{i=5}$

In this case $d_0 = 5$ and h_0 is "$h \bmod 32$".

(i.e) the number range = 0 to 31.

The value $\boxed{d_1 = d_0 + 1 = 6.}$ and

$h_1 \Rightarrow h \bmod (2 * 32)$.

(i.e). $h_1 = (h) \bmod (64)$

hence h_1 range is 0 to 63.

Similarly, h_2 range is 0 to 127 .. so on.

Process to Double the Buckets:-

→ It is uses the rounds of splitting. During the round number Level, only hash function h_{Level} and $h_{Level+1}$ are in use.

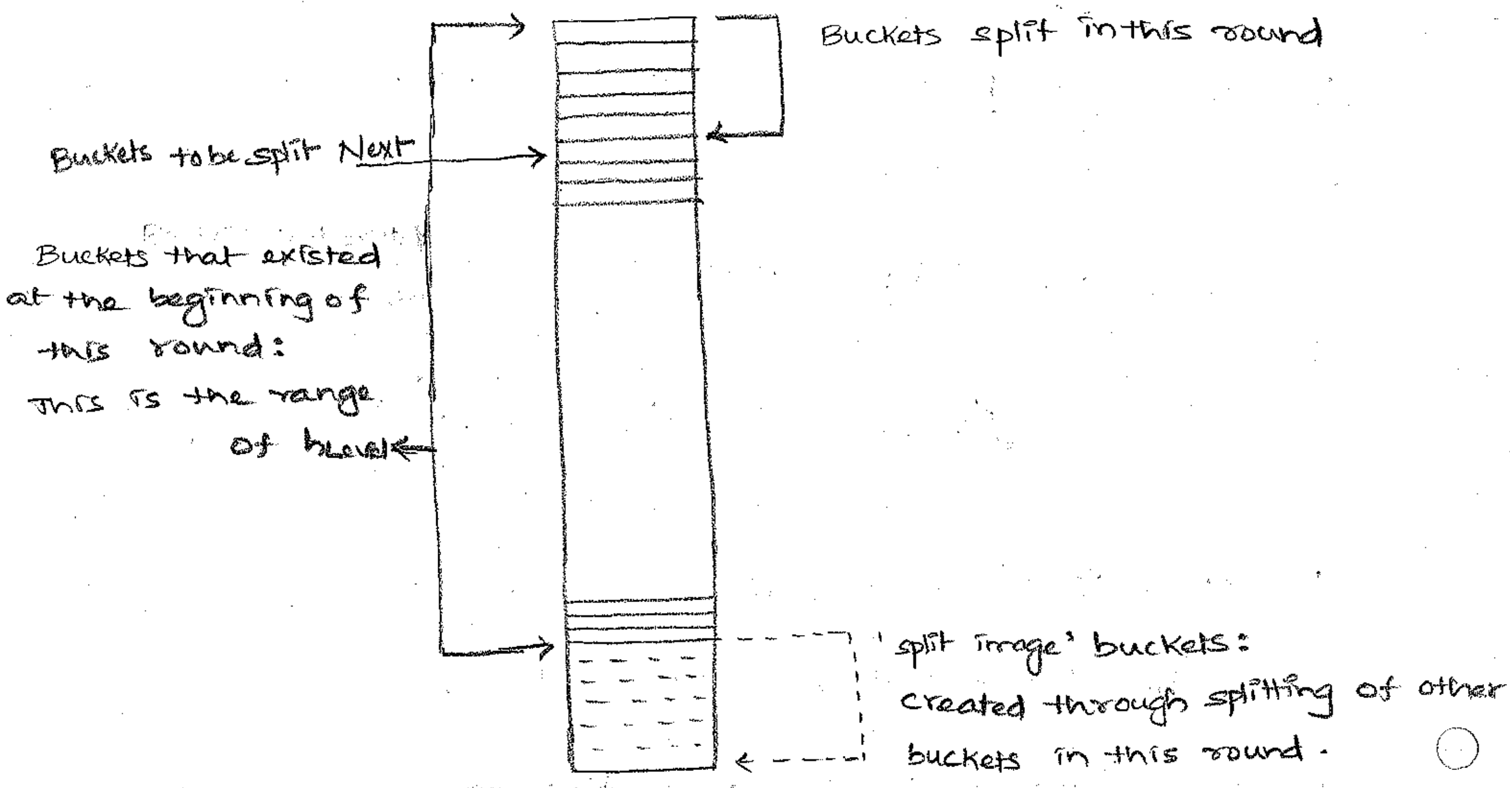
→ The buckets in the file at the beginning of round are split, one by one from the first to last bucket, there by doubling the number buckets.

→ At any point within a round, we have

(i) buckets that have been split

(ii) buckets that are yet to be split

(iii) buckets created by splits in this round.



Buckets during a round in Linear Hashing.

Process of Insertion :-

Consider the search for the data entry with a given search key value.

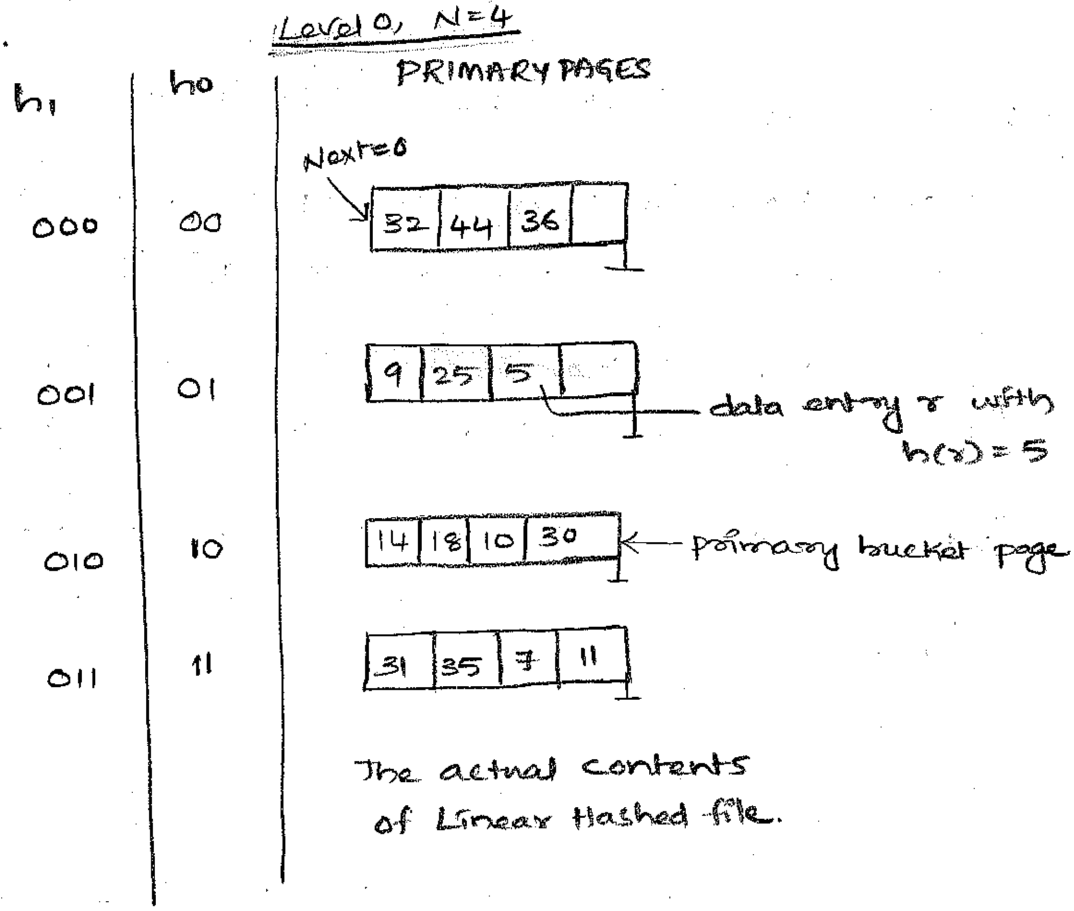
We apply a hash function h_{Level}, and if this leads us to one of the unsplit buckets, we simply look there.

→ If it leads to one of the split buckets, the entry may be there or it may have been moved to the new bucket created earlier in this round by splitting this bucket, to determine which of the two buckets contains the entry, we apply $h_{Level+1}$.

Note :- Unlike Extendible Hashing, when an insert trigger split, the bucket into which the data entry inserted is not necessarily the bucket that is split.
An overflow page is added to store the newly inserted data entry, which triggered the split as in Static Hashing.

5-5 → The buckets are choosen "round-robin fashion", eventually all buckets are split, there by redistributing the data entries in overflow chain. before the chains get to be more than one or two pages long

→ Let us take a Linear Hashed File, each bucket can hold four data entries, and the file initially contains four buckets.



Linear Hashed file

→ In The above figure,

- Level is a counter, is used to indicate the current round number and it is initialized to zero (0).

  ∴ Level=0

- Next → The bucket to split is denoted by next. Next is initialized to zero, which is a first bucket.

  Next=0.

- N_{Level} → The no. of buckets in the file at the beginning of round Level.

N_0 or N :- the no. of buckets at the begining of round 0.

→ We can split by using overflow pages. We can split whenever a new overflow page is added, or we can impose additional conditions based on conditions such as space utilization.

(i.e) We can split, when inserting a new data entry causes the creation of overflow page

→ Whenever a split is triggered the next bucket is split and hash function $h_{Level+1}$ redistribute the entries between the bucket and its split image.

∴ Suppose, the bucket number = b,
b's split image bucket number "$b + N_{Level}$".

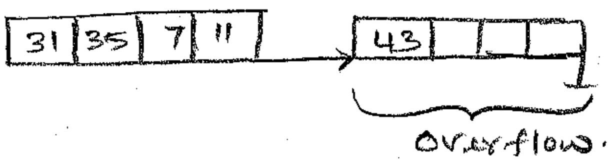
* After splitting the bucket, the value of Next is incremented by 1.

Examples:-

Insert 43:-

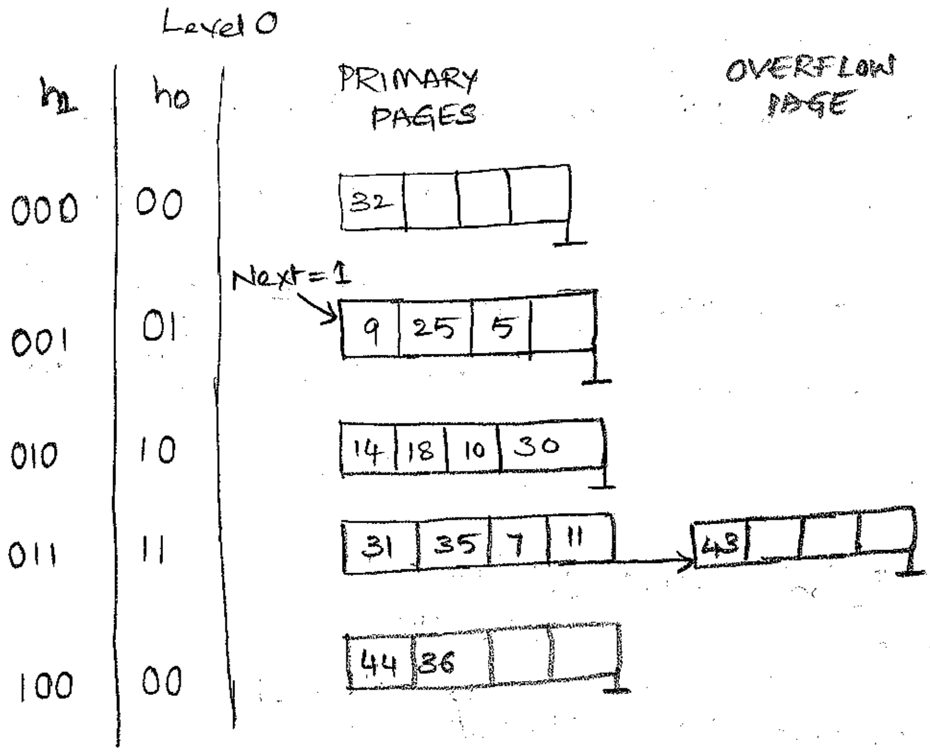
43 → In binary representation 1010<u>11</u>.

43 belongs to 11. Hence, 43 will be added to bucket



Now the bucket is full, we need to add an overflow page. At any time in the middle of the round Level, all buckets above "Next", have been split, and the file contains the buckets that are their split images, located at the bottom. The complete figure is shown as:

5-8



(∴ Next is incre-mented, & Extra bucket is added).

Fig: After Inserting Record r with h(r) = 43

→ Buckets Next through N_{Level} have yet not been split.
If we use h_{Level} on data entry and obtain a number b, in the range Next through N_{Level}, the data entry belongs to Bucket "b".

Ex:-
$h_0(18)$ → Binary Representation → 10010
The last two digits are 10. (ie) 2.
Since this value is between Next(=1) and N_1(=4), (ie) b/w 1 and 4, this bucket has not been split.

→ However if we obtain a number b, between the range 0 through Next, the data entry may be in this bucket (or) in its split image (which is bucket number $b+N_{Level}$). We have to use $h_{Level+1}$ to determine which of these two buckets the data entry belongs to.
(ie) we have to look at one more bit of data entry's hash value.

For Ex:-

$h_0(32)$ and $h_0(44)$ are both having the last two digits in their binary format are 00.

since next is currently equals to **1** (Next=1), which indicates, a bucket that has been split, we have to apply h1.

$\therefore \underset{(i.e. 0)}{\underline{h_1(32) = 000}}$ and $\underset{(i.e)(4)}{\underline{h1(44) = 100}}$

Thus 32 belongs to bucket A &
44 belongs to bucket A2, which is a split image of A

Note:- Not All Insertions trigger a split:-

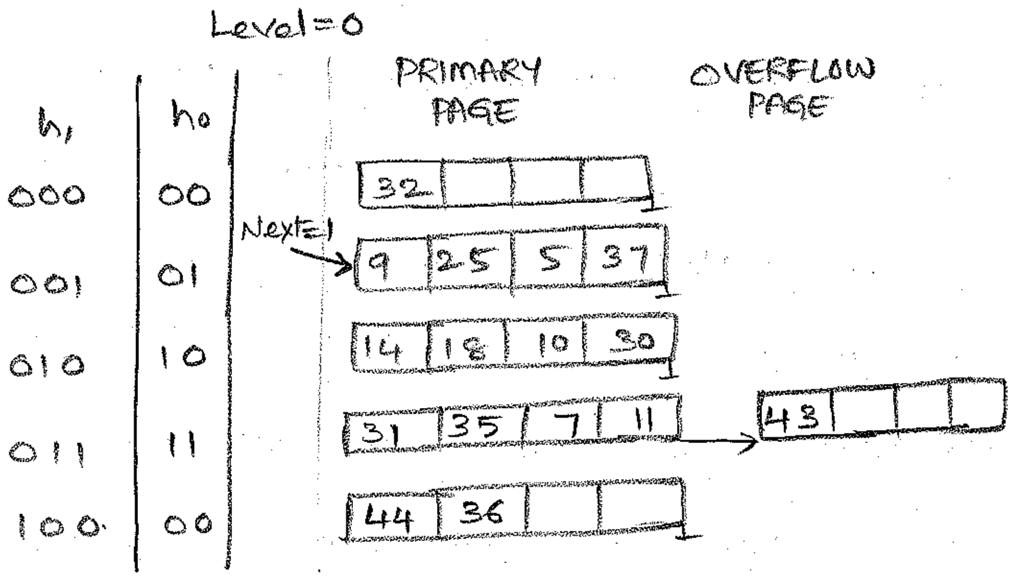
Insert 37:-

37 → 10101 (binary).

The last two digits are 01, which belongs to bucket

9	25	5	

there is a space in the bucket so, we can insert 37, which does not trigger a split.



After inserting Record r with h(r)=37.

5.T Note:

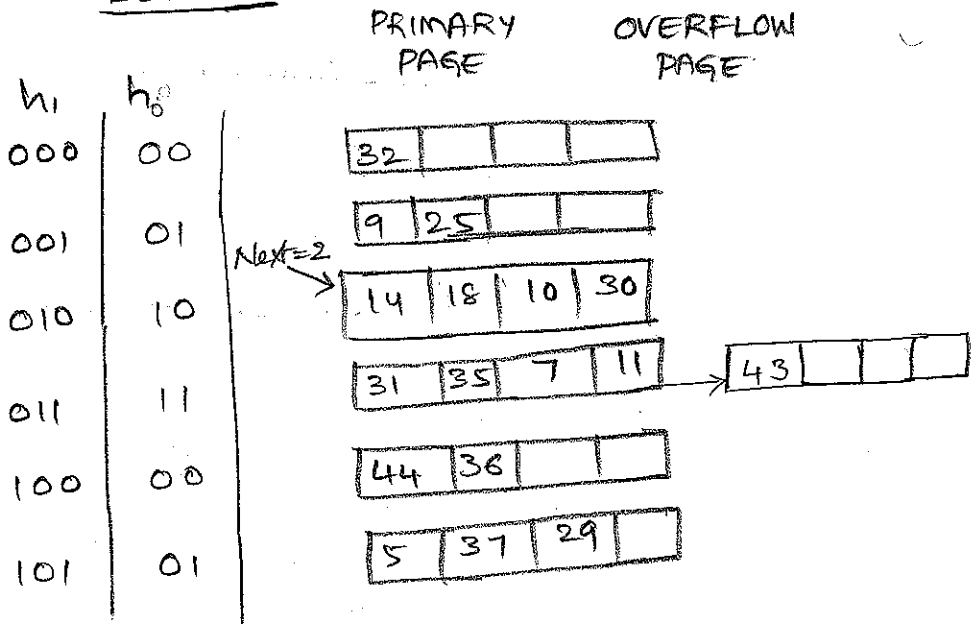
No need for overflow bucket when a bucket is full:-

Some time the buckets pointed to Next (the current bucket for splitting) is full. In this case a split is triggered, but we do not need a overflow bucket.

Ex:- Insert 29.

29 = 11101.

Level 0



After Inserting Record r with h(r)=29

When a level is incremented?

When Next is equal to $N_{Level-1}$ and a split is triggered, we split the last of the buckets that were present in the file at the beginning of "round Level."

The no. of buckets after the split is twice the number at the begining of the round, and we start a new round with: "Level is incremented by 1 and next is reset to 0".

(i.e) If Next = $N_{Level-1}$.

then Level = 1

Next = 0.

Incrementing Level amounts to doubling the effective range into which keys are hashed.

EX:- Insert 22, 66, 34

22 → 10110
66 → 1000010
34 → 100010

→ 66 and 34 belongs to 010 (the last 3 digits) bucket, but 22 belongs 110 bucket.

Level 0

h_1	h_0		PRIMARY PAGES	OVERFLOW PAGES
000	00		32	
001	01		9, 25	
010	10		66, 18, 10, 34	
011	11	Next=3 →	31, 35, 7, 11	→ 43
100	00		44, 36	
101	01		5, 37, 29	
110	10		14, 30, 22	

After inserting records with h(r) = 22, 66, 34.

Insert - 50 - which causes a split that increments a Level.

50 → 110010

The Last digits are 010, which belongs to 3rd Bucket

Level 1.

h_1	h_0		PRIMARY PAGES	OVERFLOW PAGES.
000	00	Next=0 →	32	
001	01		9, 25	
010	10		66, 18, 10, 34	→ 50
011	11		43, 35, 11	
100	00		44, 36	
101	01		5, 37, 29	
110	10		14, 30, 22	
111	11		31, 7	

After inserting record r with h(r) = 50.

Cost Comparison and Problems with Linear Hashing:-

- An Equality search costs one disk I/O unless the bucket has overflow pages.

The uniform distribution of data entries. average cost is. 1.2 disk accesses.

Problem:-

The cost can be considerably worse, when the no.of data entries in the file are Linear, (i.e) If the data distribution is non-uniform (very skewed).

The space utilization is also very poor with skewed (non-uniform) data distribution.

Inserts requires reading or writing a single page unless a split is triggered.

Process of Deletion:-

→ Deletion is inverse of Insertion.

- If the last bucket in the file is empty, it can be removed and Next can be decremented.

- If Next=0 and the last bucket becomes empty, Next is made to point to bucket "(M/2)-1", and Level decremented.

(where M= current number of buckets)

and the empty bucket is removed.

- While deletion, there is a time where we use merging, (i.e) combining the last bucket with its split image even when it is not empty.

→ The merging criteria is typically based on the occupancy of the file and merging can be done to improve space utilization.