UNIT – V

Data on External Storage, File Organization and Indexing, Cluster Indexes, Primary and Secondary Indexes, Index data Structures, Hash Based Indexing, Tree base Indexing, Comparison of File Organizations, Indexes and Performance Tuning, Intuitions for tree Indexes, Indexed Sequential Access Methods (ISAM), B+ Trees: A Dynamic Index Structure.

1. DATA ON EXTERNAL STORAGE

Primary memory has limited storage capacity and is volatile. To overcome this limitation, secondary memory is also termed as external storage devices are used. External storage devices such as disks and tapes are used to store data permanently.

The Secondary storage devices can be fixed or removable. Fixed Storage device is an internal storage device like **hard disk** that is fixed inside the computer. Storage devices that are portable and can be taken outside the computer are termed as removable storage devices such as **CD, DVD, external hard disk**, etc.

Magnetic/optical Disk: It supports random and sequential access. It takes less access time.

Magnetic Tapes: It supports only sequential access. It takes more access time.

In DBMS, processing a query and getting output need accessing random pages. So, disks are preferable than magnetic tapes.

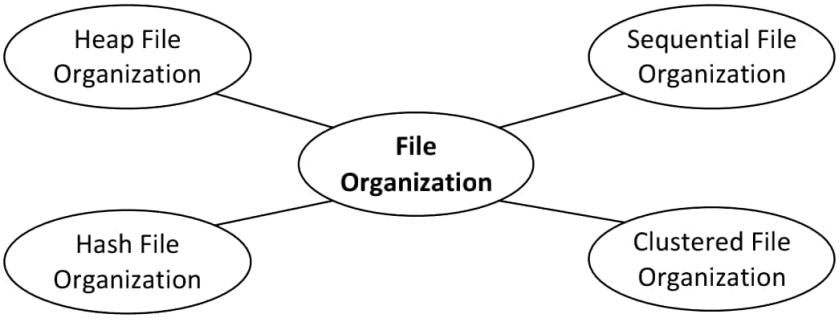
2. FILE ORGANIZATION

The database is stored as a collection of files. Each file contains a set of records. Each record is a collection of fields. For example, a student table (or file) contains many records and each record belongs to one student with fields (attributes) such as Name, Date of birth, class, department, address, etc.

File organization defines how file records are mapped onto disk blocks.

The records of a file are stored in the disk blocks because a block is the unit of data transfer between disk and memory. When the block size is larger than the record size, each block will contain more than one record. Sometimes, some of the files may have large records that cannot fit in one block. In this case, we can store part of a record on one block and the rest on another. A pointer at the end of the first block points to the block containing the remainder of the record.

The different types of file organization are given below:



Heap File Organization: When a file is created using Heap File Organization mechanism, the records are stored in the file in the order in which they are inserted. So the new records are inserted at the end of the file. In this type of organization inserting new records is more efficient. It uses linear search to search records.

Sequential File Organization: When a file is created using Sequential File Organization mechanism, all the records are ordered (sorted) as per the primary key value and placed in the file. In this type of organization inserting new records is more difficult because the records need to be sorted after inserting every new record. It uses binary search to search records.

Hash File Organization: When a file is created using Hash File Organization mechanism, a hash function is applied on some field of the records to calculate hash value. Based on the hash value, the corresponding record is placed in the file.

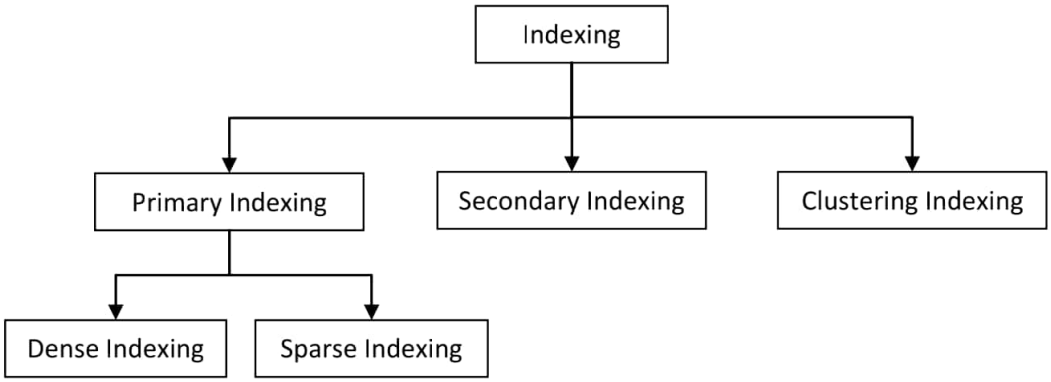
Clustered File Organization: Clustered file organization is not considered good for large databases. In this mechanism, related records from one or more relations are kept in a same disk block, that is, the ordering of records is not based on primary key or search key.

3. INDEXING

If the records in the file are in sorted order, then searching will become very fast. But, in most of the cases, records are placed in the file in the order in which they are inserted, so new records are inserted at the end of the file. It indicates, the records are not in sorted order. In order to make searching faster in the files with unsorted records, indexing is used.

Indexing is a data structure technique which allows you to quickly retrieve records from a database file. An Index is a small table having only two columns. The first column contains a copy of the primary or candidate key of a table. The second column contains a set of disk block addresses where the record with that specific key value stored.

Indexing in DBMS can be of the following types:

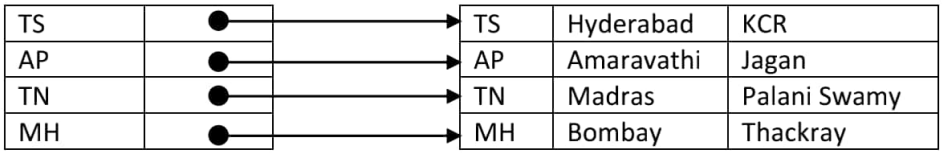


i. Primary Index

- If the index is created by using the primary key of the table, then it is known as primary indexing.
- As primary keys are unique and are stored in a sorted manner, the performance of the searching operation is quite efficient.
- The primary index can be classified into two types: dense index and sparse index.

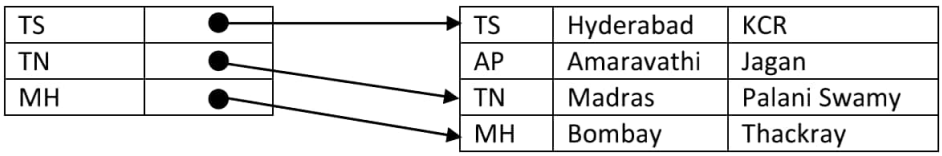
Dense index

- If every record in the table has one index entry in the index table, then it is called dense index.
- In this, the number of records (rows) in the index table is same as the number of records (rows) in the main table.
- As every record has one index entry, searching becomes faster.



Sparse index

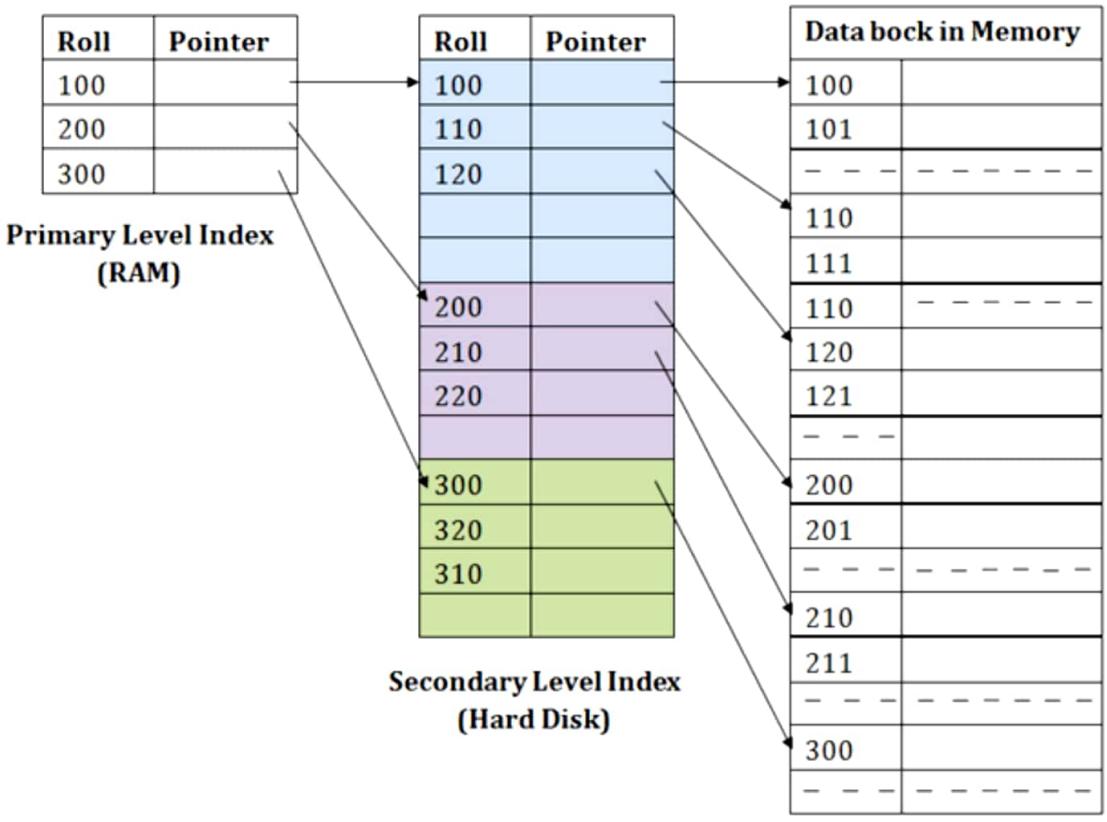
- If only few records in the table have index entries in the index table, then it is called sparse index.
- In this, the number of records (rows) in the index table is less than the number of records (rows) in the main table.
- As not all the record have index entries, searching becomes slow for records that does not have index entries.



ii. Secondary Index

When the size of the main table grows, then size of index table also grows. If the index table size grows then fetching the address itself becomes slower. To overcome this problem, secondary indexing is introduced.

- In secondary indexing, to reduce the size of mapping, another level of indexing is introduced.
- It contains two levels. In the first level each record in the main table has one entry in the first-level index table.
- The index entries in the fisrt level index table are divided into different groups. For each group, one index entry is created and added in the second level index table.

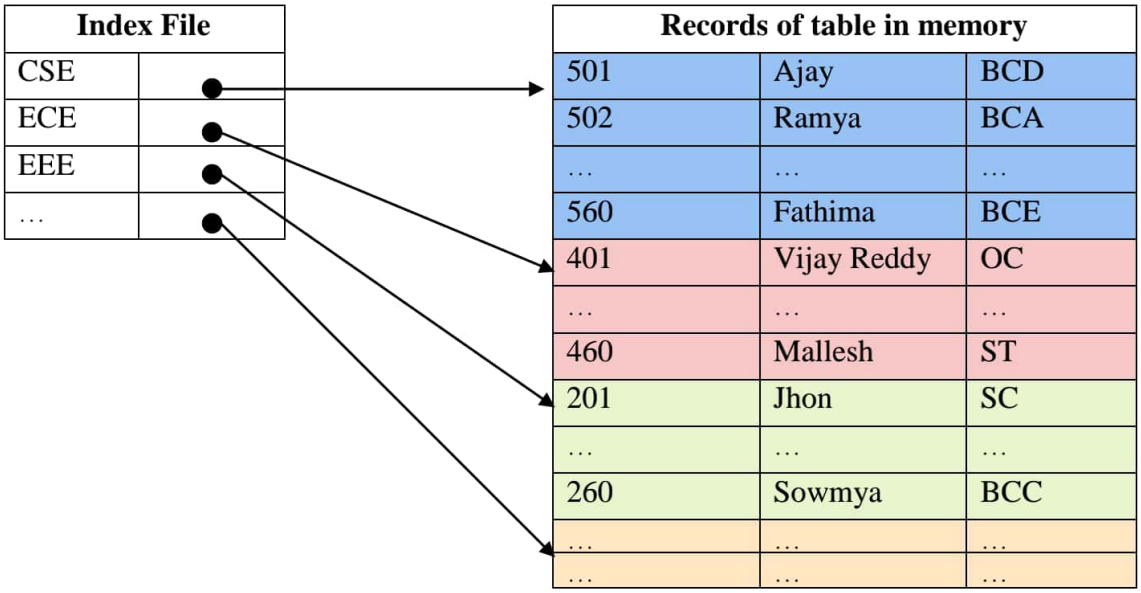


Multi-level Index: When the main table size becomes too large, creating secondary level index improves the search process. Even if the search process is slow; we can add one more level of indexing and so on. This type of indexing is called multi-level index.

iii. Clustering Index

- Sometimes the index is created on non-primary key columns which may not be unique for each record.
- In this case, to identify the record faster, we will group two or more columns to get the unique value and create index out of them. This method is called a clustering index.
- The records which have similar characteristics are grouped, and indexes are created for these group.

Example: Consider a college contains many students in each department. All the students belong to the same Dept_ID are grouped together and treated as a single cluster. One index pointers point to the one cluster as a whole. The idex pointer points to the first record in each cluster. Here Dept_ID is a non-unique key.

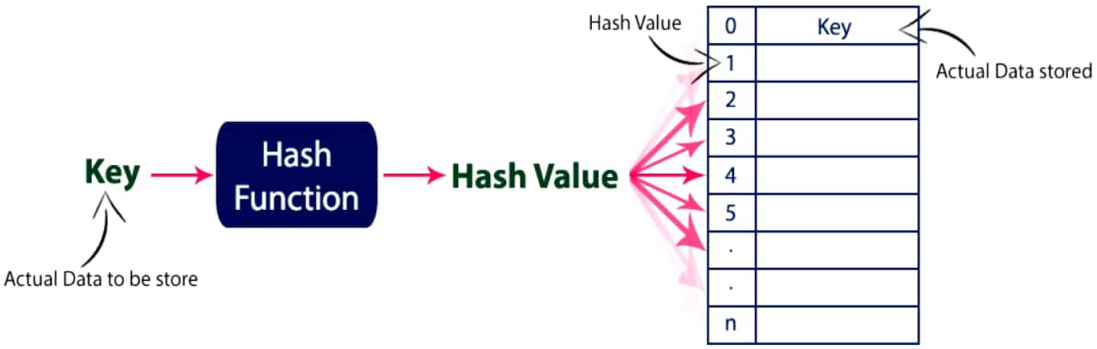


In above diagram we can see that, indexes are created for each department in the index file. In the data block, the students of each department are grouped together to form the cluster. The address in the index file points to the beginning of each cluster.

4. HASH BASED INDEXING

Hashing is a technique to directly search the location of desired data on the disk without using index structure. Hash function is a function which takes a piece of data (key) as input and produces a hash value as output which maps the data to a particular location in the hash table.

The concept of hashing and hash table is shown in the below figure



There are mainly two types of hashing methods:

i. Static Hashing

ii. Dynamic Hashing

- Extended hashing
- Linear hashing

5. STATIC HASHING

In static hashing, the hash function produce only fixed number of hash values. For example consider the hash function

$$f(x) = x \bmod 7$$

For any value of x, the above function produces one of the hash value from {0, 1, 2, 3, 4, 5, 6}. It means static hashing maps search-key values to a fixed set of bucket addresses.

Example: Inserting 10, 21, 16 and 12 in hash table.

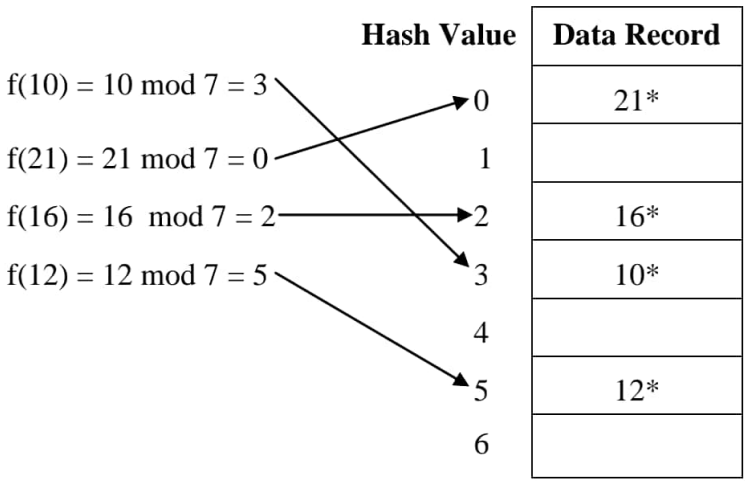


Figure 5.1: *Static hashing*

Suppose, latter if we want to insert 23, it produce hash value as 2 (23 mod 7 = 2). But, in the above hash table, the location with hash value 2 is not empty (it contains 16*). So, a collision occurs. To resolve this collision, the following techniques are used.

- Open addressing
- Separate Chaining or Closed addressing

i. Open Addressing:

Open addressing is a collision resolving technique which stores all the keys inside the hash table. No key is stored outside the hash table. Techniques used for open addressing are:

- Linear Probing
- Quadratic Probing
- Double Hashing

➢ Linear Probing:

In linear probing, when there is a collision, we scan forwards for the next empty slot to fill the key's record. If you reach last slot, then start from beginning.

Example: Consider figure 1. When we try to insert 23, its hash value is 2. But the slot with 2 is not empty. Then move to next slot (hash value 3), even it is also full, then move once again to next slot with hash value 4. As it is empty store 23 there. This is shown in the below diagram.

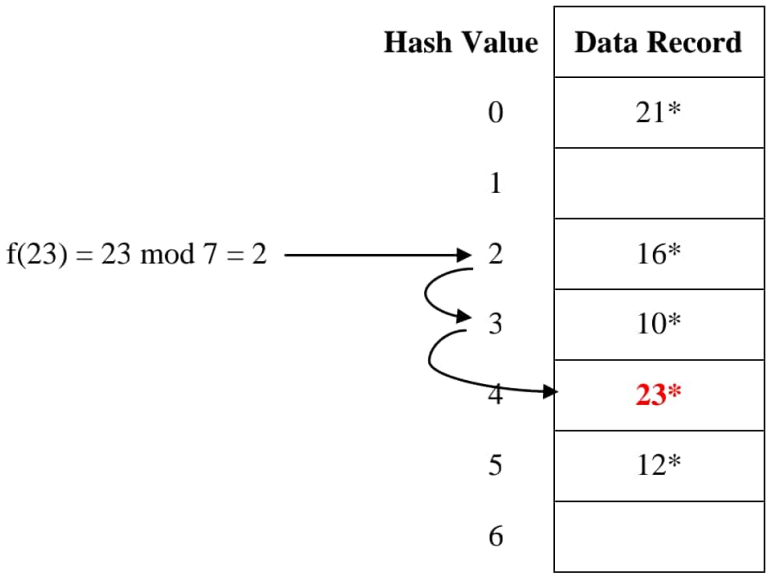


Figure 5.2: *Linear Probing*

➢ Quadratic Probing:

In quadratic probing, when collision occurs, it compute new hash value by taking the original hash value and adding successive values of quadratic polynomial until an open slot is found. If here is a collision, it use the following hash function: $h(x) = (f(x) + i^2)$ mod n , where I = 1, 2, 3, 4,…. and f(x) is initial hash value.

Example: Consider figure 1. When we try to insert 23, its hash value is 2. But the slot with hash value 2 is not empty. Then compute new hash value as $(2 + 1^2)$ mod 7 =3, even it is also full, and then once again compute new hash value as $(2 + 2^2)$ mod 7 = 6. As it is empty store 23 there. This is shown in the below diagram.

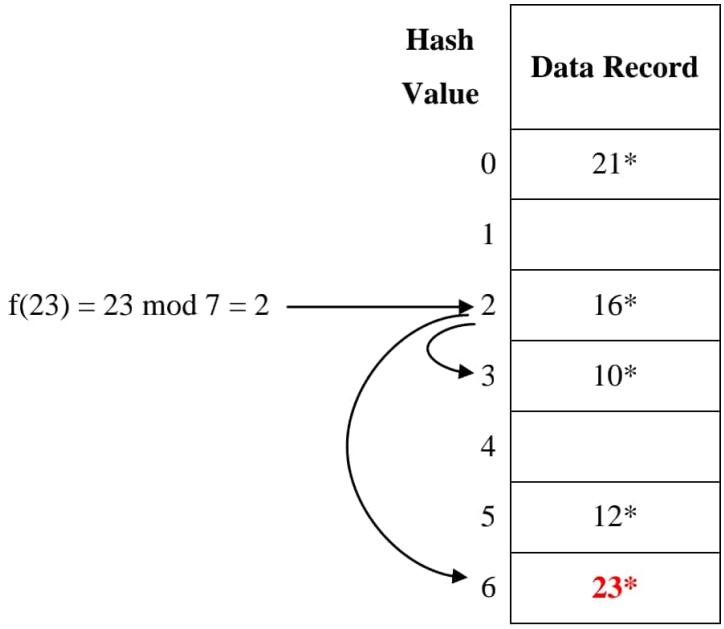


Figure 5.3: *Quadratic Probing*

➢ Double Hashing

In double hashing, there are two hash functions. The second hash function is used to provide an offset value in case the first function causes a collision. The following function is an example of double hashing: (firstHash(key) + i * secondHash(key)) % tableSize. Use i = 1, 2, 3, …

A popular second hash function is : secondHash**(key) = PRIME – (key % PRIME)**

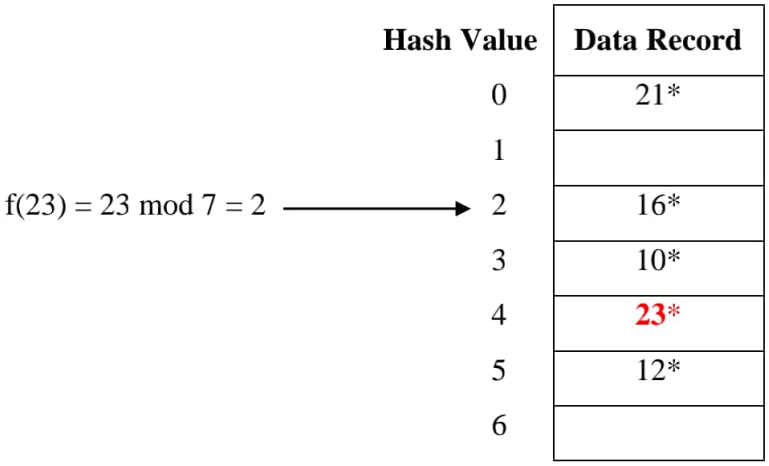
where PRIME is a prime smaller than the TABLE_SIZE.

Example: Consider figure 1. When we try to insert 23, its hash value is 2. But the slot with hash value 2 is not empty. Then compute double hashing value as

secondHash (key) = PRIME – (key % PRIME) → secondHash (23) = 5 – (23 % 5) = 2

Double hashing: (firstHash(key) + i * secondHash(key)) % tableSize → (2+1*2))%7 =4

As the slot with hash value 4 is empty, store 23 there. This is shown in the below diagram.



Hash Value	Data Record
0	21*
1	
2	16*
3	10*
4	23*
5	12*
6	

Figure 5.4: *Double Probing*

ii. Separate Chaining or Closed addressing:

To handle the collision, This technique creates a linked list to the slot for which collision occurs. The new key is then inserted in the linked list. These linked lists to the slots appear like chains. So, this technique is called as *separate chaining*. It is also called as closed addressing.

Example: Inserting 10, 21, 16, 12, 23, 19, 28, 30 in hash table.

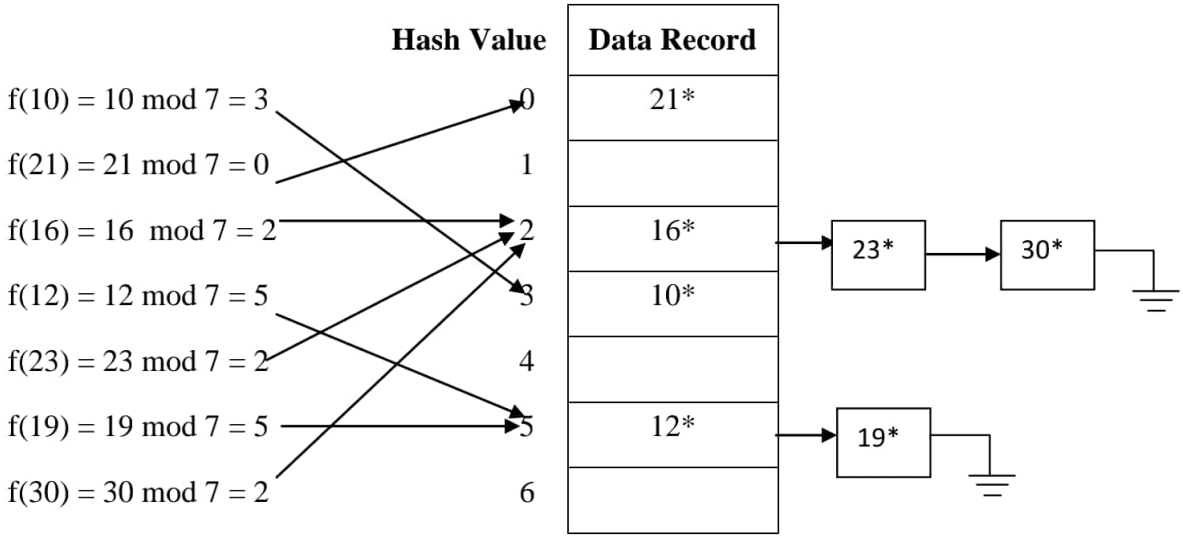


Figure 5.5: *Separate chaining example*

6. 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. Dynamic hashing can be implemented using two techniques. They are:

- Extended hashing
- Linear Hashing

i. Extendable hashing

In extendable hashing, a separate *directory* of pointers to buckets is used. The number bits used in directory is called global depth (gd) and number entries in directory = 2^{gd}. Number of bits used for locating the record in the buckets is called *local depth*(ld) and each bucket can stores up to 2^{ld} entries. The hash function use last few binary bits of the key to find the bucket. If a bucket overflows, it splits, and if local depth greater than global depth, then the table doubles in size. It is one form of dynamic hashing.

Example: Let global depth (gd) = 2. It means the directory contains four entries. Let the local depth (ld) of each bucket = 2. It means each bucket need two bits to perform search operation. Let each Bucket capacity is four. Let us insert 21, 15, 28, 17, 16, 13, 19, 12, 10, 24, 25 and 11.

21	=	10101	19	=	10011
15	=	01111	12	=	01100
28	=	11100	10	=	01010
17	=	10001	24	=	11000
16	=	10000	25	=	11101
13	=	01101	11	=	01011

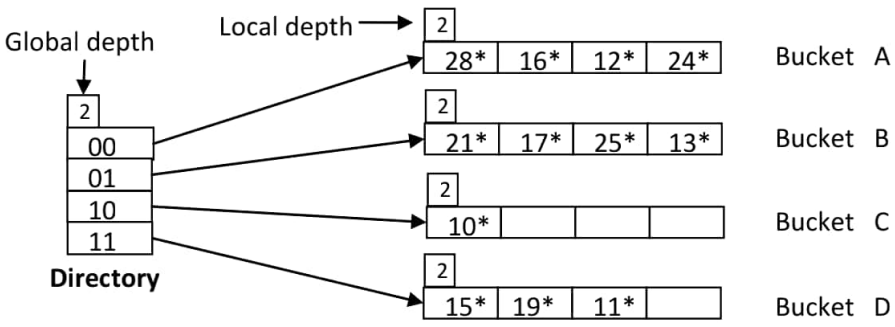


Figure 6.1: *Extendible hashing example*

Now, let us consider insertion of data entry 20* (binary 10100). Looking at directory element 00, we are led to bucket A, which is already full. So, we have to **split** the bucket by

allocating a new bucket and redistributing the contents (including the new entry to be inserted) across the old bucket and its 'split image (new bucket). To redistribute entries across the old bucket and its split image, we consider the last *three bits;* the last two bits are 00, indicating a data entry that belongs to one of these two buckets, and the third bit discriminates between these buckets. That is if a key's last three bits are 000, then it belongs to bucket A and if the last three bits are 100, then the key belongs to Bucket A2. As we are using three bits for A and A2, so the local depth of these buckets becomes 3. This is illustrated in the below Figure 6.2.

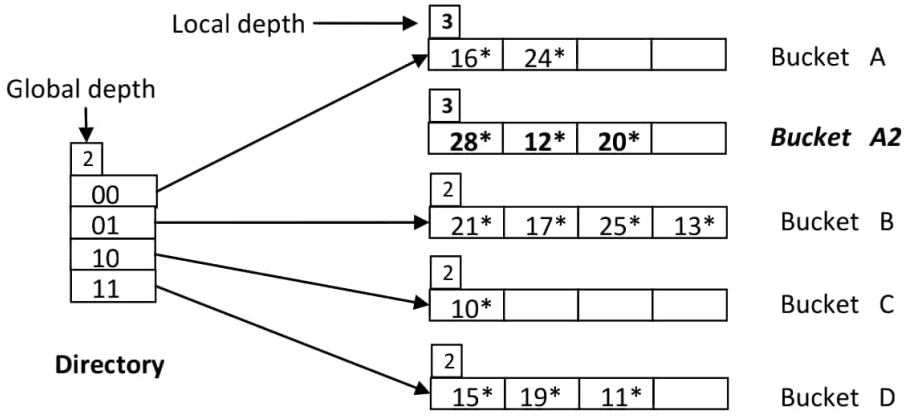


Figure 6.2: *After inserting 20 and splitting Bucket A*

After split, Bucket A and A2 have local depth greater than global depth (3 > 2), so double the directory and use three bits as global depth. As Bucket A and A2 has local depth 3, so they have separate pointers from the directory. But, Buckets B, C and D use only local depth 2, so they have two pointers each. This is shown in the below diagram.

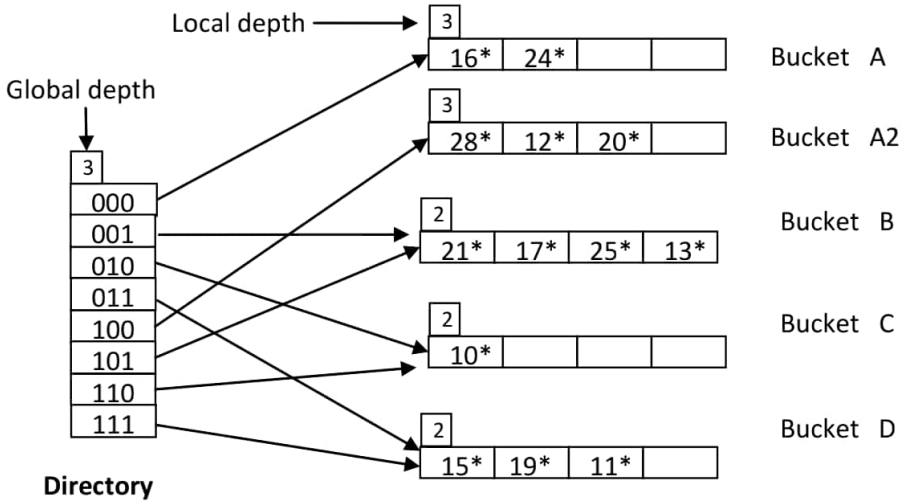


Figure 6.3: *After inserting 20 and doubling the directory*

An important point that arises is whether splitting a bucket necessitates a directory doubling. Consider our example, as shown in Figure 6.3. If we now insert 9* (01001), it belongs in bucket B; this bucket is already full. We can deal with this situation by splitting the bucket and using directory elements 001 and 101to point to the bucket and its split image. This is shown in the below diagram. As Bucket B and its split image now have local depth three and it is not greater than global depth. Hence, a bucket split does not necessarily require a directory doubling. However, if either bucket A or A2 grows full and an insert then forces a bucket split, we are forced to double the directory once again.

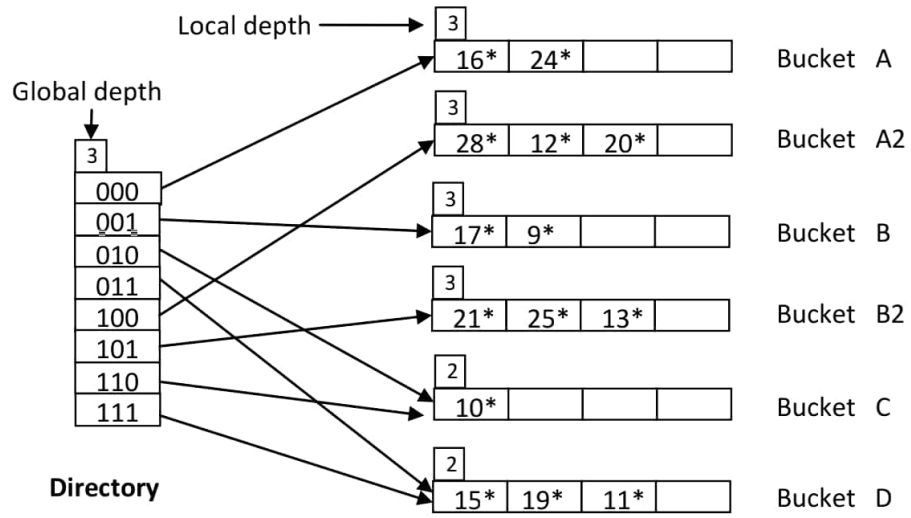


Figure 6.4: *After inserting 9*

Key Observations:

- A Bucket will have more than one pointers pointing to it if its local depth is less than the global depth.
- When overflow condition occurs in a bucket, all the entries in the bucket are rehashed with a new local depth.
- If new Local Depth of the overflowing bucket is equal to the global depth, only then the directories are doubled and the global depth is incremented by 1.
- The size of a bucket cannot be changed after the data insertion process begins.

ii. Linear Hashing

Linear hashing is a dynamic hashing technique that linearly grows or shrinks number of buckets in a hash file without a directory as used in *Extendible Hashing*. It uses a family of hash functions instead of single hash function.

This scheme utilizes a *family* of hash functions h_0, h_1, h_2, ... , with the property that each function's range is twice that of its predecessor. That is, if h_i maps a data entry into one of N buckets, h_{i+1} maps a data entry into one of $2N$ buckets. One example of such hash function family can be obtained by: $h_{i+1}(key) = key \bmod (2^i N)$ where N is the initial number of buckets and i = 0,1,2,….

Initially it use N buckets labelled 0 through N–1 and an initial hashing function $h_0(key) = key \% N$ is used to map any key into one of the N buckets. For each overflow bucket, one of the buckets in serial order will be splited and its content is redistributed between it and its split image. That is, for first time overflow in any bucket, bucket 0 will be splited, for second time overflow in any bucket; bucket 1 will be splited and so on. When number of buckets becomes 2N, then this marks the end of splitting round 0. Hashing function h_0 is no longer needed as all 2N buckets can be addressed by hashing function h_1. In new round namely splitting-round 1, bucket split once again starts from bucket 0. A new hash function h_2 will be used. This process is repeated when the hash file grows.

Example: Let N = 4, so we use 4 buckets and hash function $h_0(key) = key \% 4$ is used to map any key into one of the four buckets. Let us initially insert 4, 13, 19, 25, 14, 24, 15, 18, 23, 11, 16, 12 and 10.This is shown in the below figure.

Bucket#	*h_1*	*h_0*	*Primary pages*				*Overflow pages*
0	000	00	4*	24*	16*	12*	
1	001	01	13*	25*			
2	010	10	14*	18*	10*		
3	011	11	19*	15*	23*	11*	

Next, when 27 is inserted, an overflow occurs in bucket 3. So, bucket 0 (first bucket) is splited and its content is distributed between bucket 0 and new bucket. This is shown in below figure.

Bucket#	*h_1*	*h_0*	*Primary pages*				*Overflow pages*			
0	000	00	24*	16*						
1	001	01	13*	25*						
2	010	10	14*	18*	10*					
3	011	11	19*	15*	23*	11*	27*			
4	100	00	4*	12*						

Next, when 30, 31 and 34 is inserted, an overflow occurs in bucket 2. So, bucket 1 is splited and its content is distributed between bucket 1 and new bucket. This is shown in below figure.

Bucket#	*h1*	*h0*	*Primary pages*	*Overflow pages*
0	000	00	24*, 16*	
1	001	01	13*	
2	010	10	14*, 18*, 10*, 30*	34*
3	011	11	19*, 15*, 23*, 11*	27*, 31*
4	100	00	4*, 12*	
5	101	01	25*	

When 32, 35, 40 and 48 is inserted, an overflow occurs in bucket 0. So, bucket 2 is splited and its content is distributed between bucket 2 and new bucket. This is shown in below figure.

Bucket#	h_1	h_0	*Primary pages*	*Overflow pages*
0	000	00	24*, 16*, 32*, 40*	48*
1	001	01	13*	
2	010	10	18*, 10*, 34*	
3	011	11	19*, 15*, 23*, 11*	27*, 31*, 35*
4	100	00	4*, 12*	
5	101	01	25*	
6	110	10	14*, 30*	

When 26, 20 and 42 is inserted, an overflow occurs in bucket 0. So, bucket 3 is splited and its content is distributed between bucket 3 and new bucket. This is shown in below figure.

Bucket#	*h1*	*h0*	*Primary pages*	*Overflow pages*
0	000	00	24*, 16*, 32*, 40*	48*
1	001	01	13*	
2	010	10	18*, 10*, 34*, 26*	42
3	011	11	19*, 11*, 27*, 35*	
4	100	00	4*, 12*, 20*	
5	101	01	25*	
6	110	10	14*, 30*	
7	111	11	15*, 23*, 31*	

This marks the end of splitting round. Hashing function h_0 is no longer needed as all 2N buckets can be addressed by hashing function h_1. In new round namely splitting-round 1, bucket split once again starts from bucket 0. A new hash function h_2 will be used. This process is repeated.

7. INTUITIONS FOR TREE INDEXES

We can use tree-like structures as index as well. For example, a binary search tree can also be used as an index. If we want to find out a particular record from a binary search tree, we have the added advantage of binary search procedure, that makes searching be performed even faster. A binary tree can be considered as a **2-way Search Tree**, because it has two pointers in each of its nodes, thereby it can guide you to two distinct ways. Remember that for every node storing 2 pointers, the number of value to be stored in each node is one less than the number of pointers, i.e. each node would contain 1 value each.

The abovementioned concept can be further expanded with the notion of the m-Way Search Tree, where m represents the number of pointers in a particular node. If m = 3, then each node of the search tree contains 3 pointers, and each node would then contain 2 values. We use mainly two tree structure indexes in DBMS. They are:

- Indexed Sequential Access Methods (ISAM)
- B+ Tree

8. INDEXED SEQUENTIAL ACCESS METHODS (ISAM)

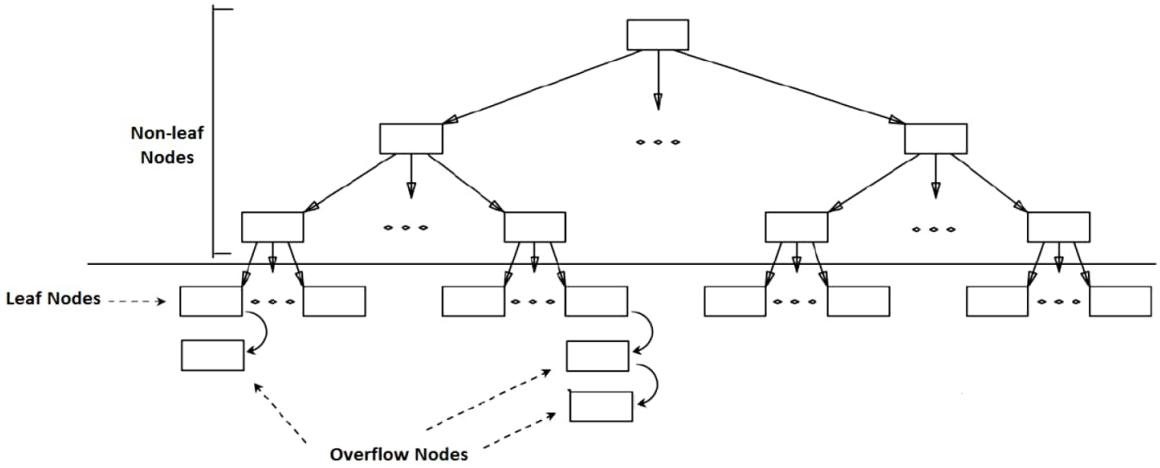
ISAM is a tree structure data that allows the DBMS to locate particular record using index without having to search the entire data set.

- The records in a file are sorted according to the primary key and saved in the disk.
- For each primary key, an index value is generated and mapped with the record. This index is nothing but the address of record.
- A sorted data file according to primary index is called an indexed sequential file.
- The process of accessing indexed sequential file is called ISAM.
- ISAM makes searching for a record in larger database is easy and quick. But proper primary key has to be selected to make ISAM efficient.

- ISAM gives flexibility to generate index on other fields also in addition to primary key fields.

ISAM contain three types of nodes:

- **Non-leaf nodes:** They store the search index key values.
- **Leaf nodes:** They store the index of records.
- **Overflow nodes:** They also store the index of records but after the leaf node is full.



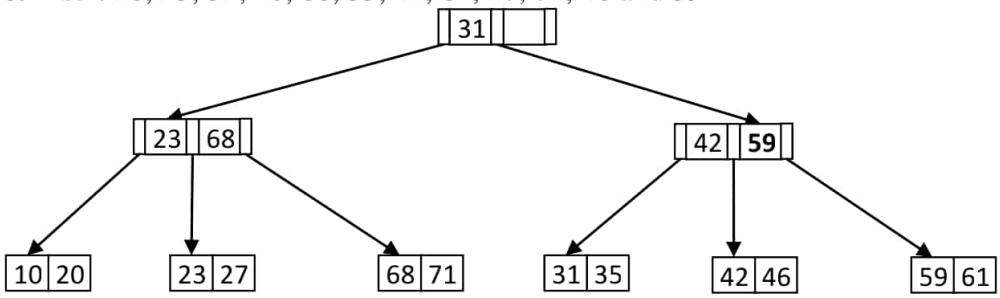
On ISAM, we can perform search, insertion and deletion operations.

Search Operation: It follows binary search process. The record to be searched will be available in the leaf nodes or in overflow nodes only. The non-leaf nodes are used to search the value.

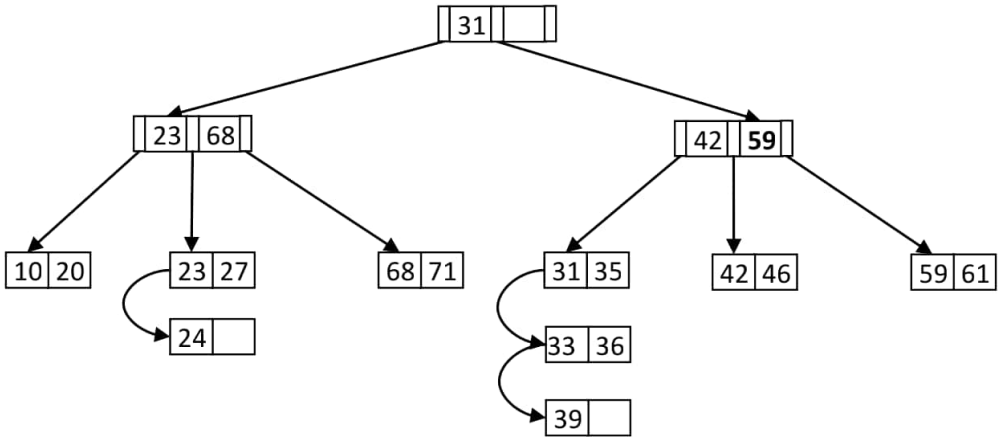
Insertion operation: First locate a leaf node where the insertion to be take place (use binary search). After finding leaf node, insert it in that leaf node if space is available, else create an overflow node and insert the record index in it, and link the overflow node to the leaf node.

Deletion operation: First locate a leaf node where the deletion to be take place (use binary search). After finding leaf node, if the value to be deleted is in leaf node or in overflow node, remove it. If the overflow node is empty after removing the deleted value, then delete overflow node also.

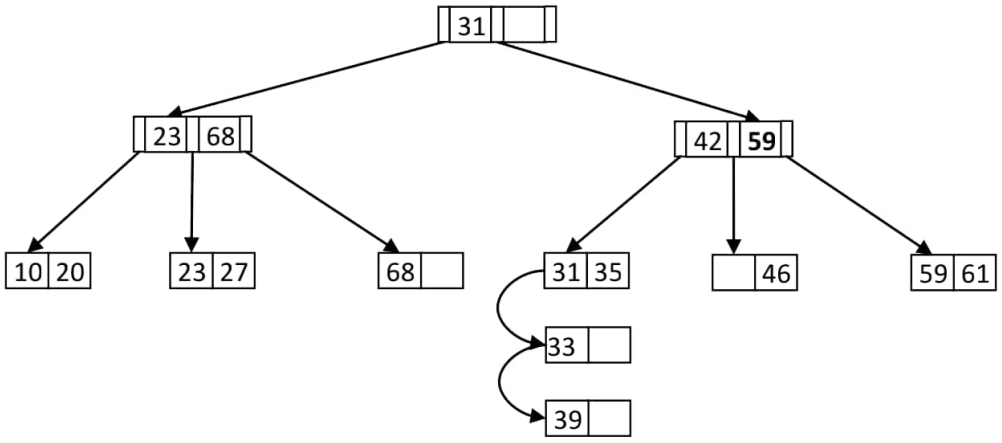
Example: Insert 10, 23, 31, 20, 68, 35, 42, 61, 27, 71, 46 and 59



After inserting 24, 33, 36, and 39 in the above tree, it looks like



Deletion: From the above figure, after deleting 42, 71, 24 and 36



9. B+ TREE

B+ Tree is an extension of Binary Tree which allows efficient insertion, deletion and search operations. It is used to implement indexing in DBMS. In B+ tree, data can be stored only on the leaf nodes while internal nodes can store the search key values.

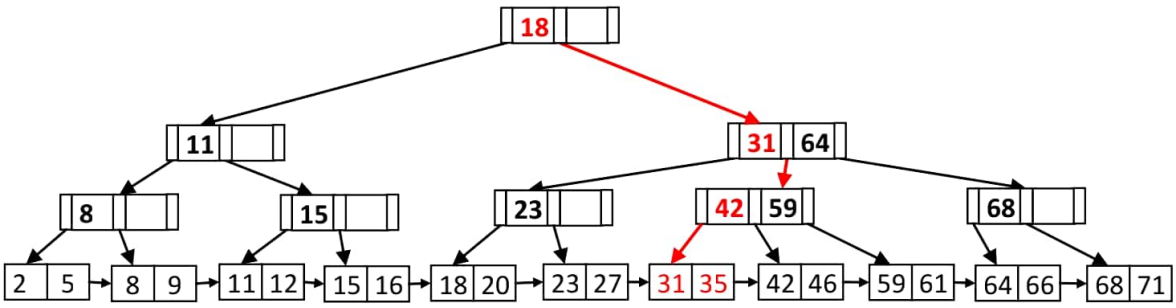
1. B+ tree of an order *m* can store max *m-1* values at each node.
2. Each node can have a maximum of *m* children and at least *m/2* children (except root).
3. The values in each node are in sorted order.
4. All the nodes must contain at least half full except the root node.
5. Only leaf nodes contain values and non-leaf nodes contain search keys.

B+ Search:

Searching for a value in the B+-Tree always starts at the root node and moves downwards until it reaches a leaf node. The search procedure follows binary tree search procedure.

1. Read the value to be searched. Let us say this value as *X*.
2. Start the search process from root node
3. At each non-leaf node (including root node),
    a. If all the values in the non-leaf node are greater than X, then move to its first child
    b. If all the values in the non-leaf node are less than or equal to X, then move to its last child
    c. If for any two consecutive values in the non-leaf node, left value is less and right value greater than or equal to X, then move to the child node whose pointer is in between these two consecutive values.
4. Repeat step-3 until a leaf node reaches.
5. At leaf node compare the key with the values in that node from left to right. If the key value is found then display found. Otherwise display it is not found.

Example: Searching for 35 in the below given B+ tree. The search path is shown in red color.



B+ Insertion:

1. Apply search operation on B+ tree and find a leaf node where the new value has to insert.
2. If the leaf node is not full, then insert the value in the leaf node.
3. If the leaf node is full, then

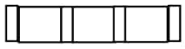
a. Split that leaf node including newly inserted value into two nodes such that each contains half of the values (In case of odd, 2nd node contains extra value).
b. Insert smallest value from new right leaf node (2nd node) into the parent node. Add pointers from these new leaf nodes to their parent node.
c. If the parent is full, split it too. Add the middle key (In case of even,1st value from 2nd part)of this parent node to its parent node.
d. Repeat until a parent is found that need not split.

4. If the root splits, create a new root which has one key and two pointers.

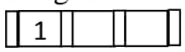
Example: Insert 1,5,3,7,9,2,4,6,8,10 into B+ tree of an order 4.

B+ tree of order 4 indicates there are maximum 3 values in a node.

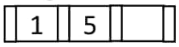
Initially



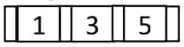
After inserting 1



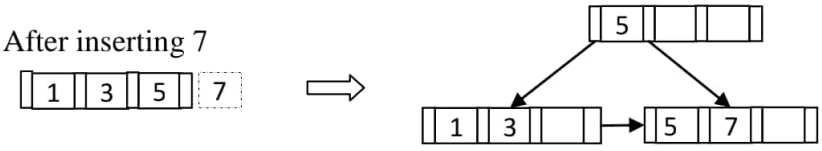
After inserting 5



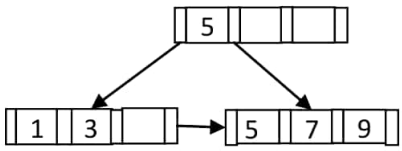
After inserting 3



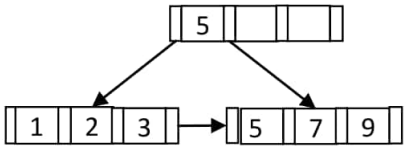
After inserting 7



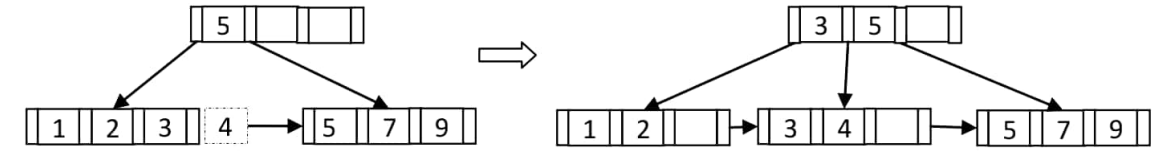
After inserting 9



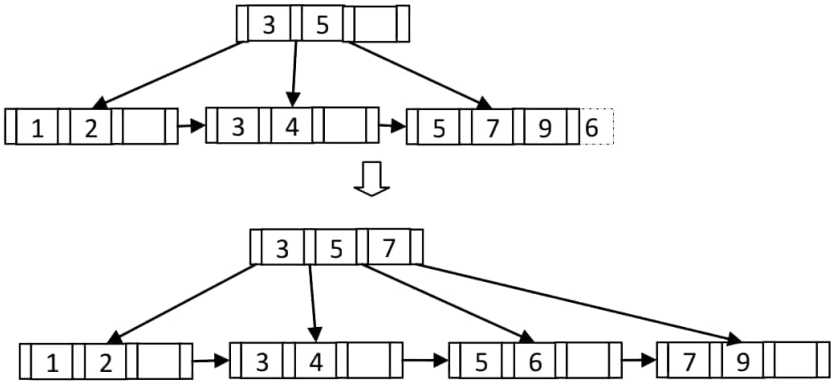
After inserting 2



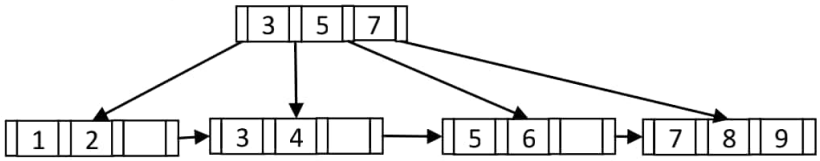
After inserting 4



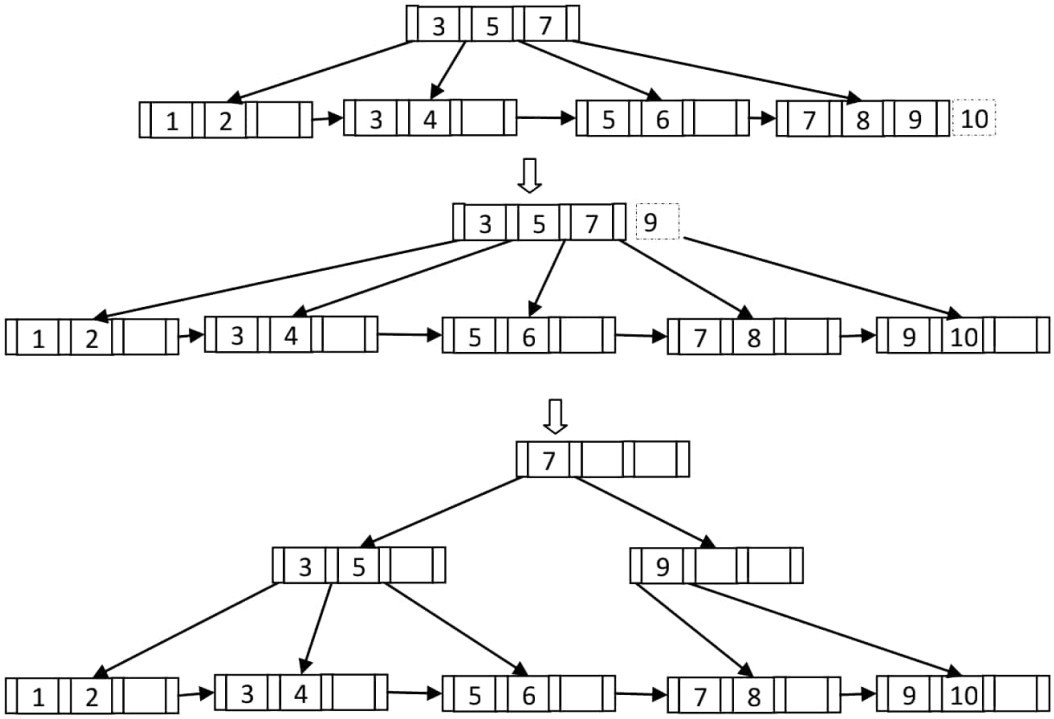
After inserting 6



After inserting 8



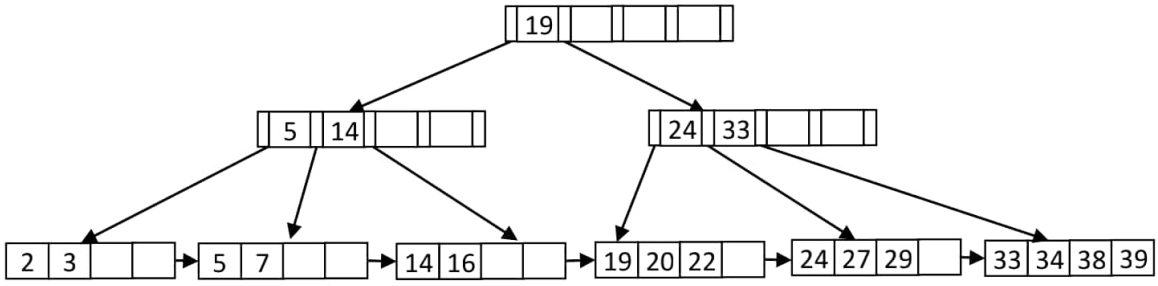
After inserting 10



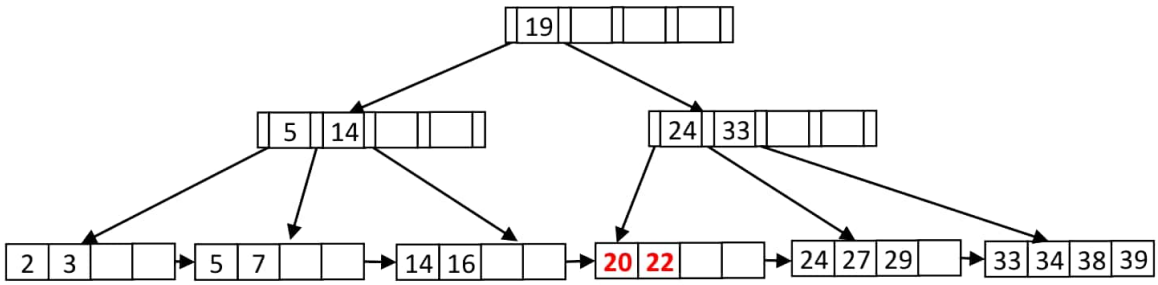
B+ Deletion

- Identify the leaf node L from where deletion should take place.
- Remove the data value to be deleted from the leaf node L
- If L meets the "half full" criteria, then its done.
- If L does not meets the "half full" criteria, then
    - If L's right sibling can give a data value, then move smallest value in right sibling to L (After giving a data value, the right sibling should satisfy the half full criteria. Otherwise it should not give)
    - Else, if L's left sibling can give a data value, then move largest value in left sibling to L (After giving a data value, the left sibling should satisfy the half full criteria. Otherwise it does not give)
    - Else, merge L and a sibling
    - If any internal nodes (including root) contain key value same as deleted value, then delete those values and replace with it successor. This deletion may propagate up to root. (If the changes propagate up to root then tree height decreases).

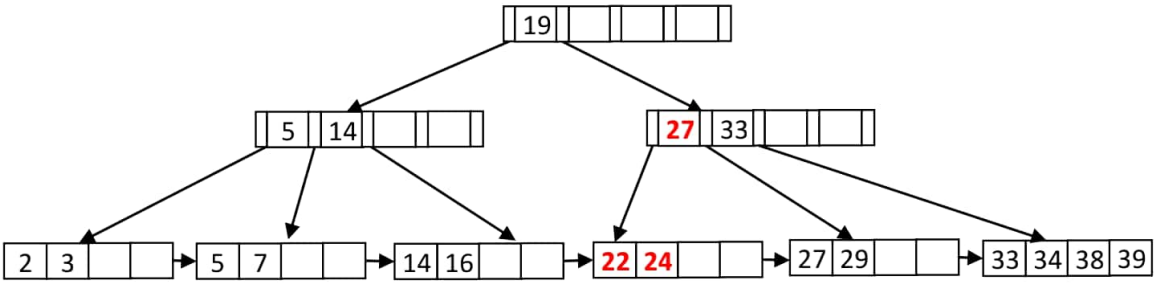
Example: Consider the given below tree and delete 19,



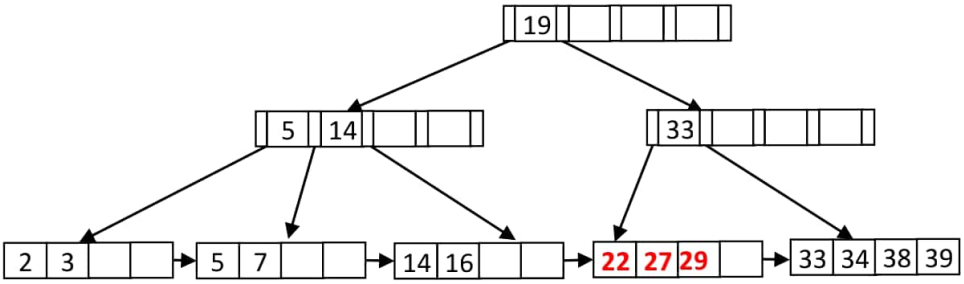
Delete 19 : Half full criteria is satisfied even after deleting 19, so just delete 19 from leaf node



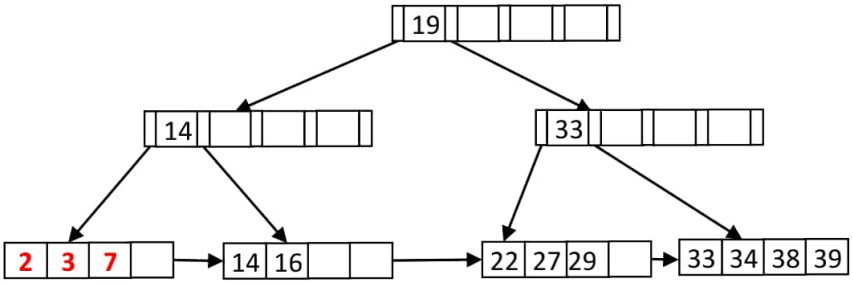
Delete 20: Half full criteria is not satisfied after deleting 20, so bring 24 from its right sibling and change key values in the internal nodes.



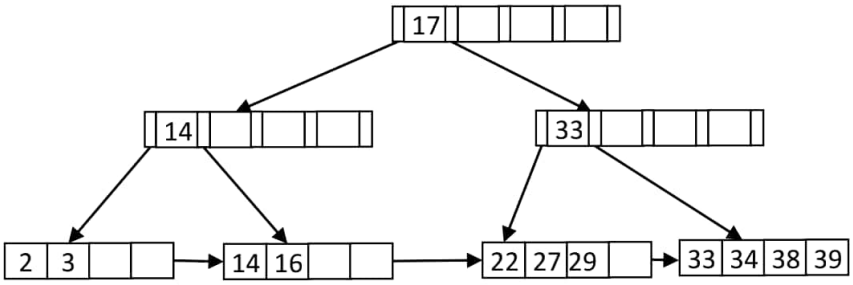
Delete 24: Half full criteria is not satisfied after deleting 24, bringing a value from its siblings also not possible. Therefore merge it with its right sibling and change key values in the internal nodes.



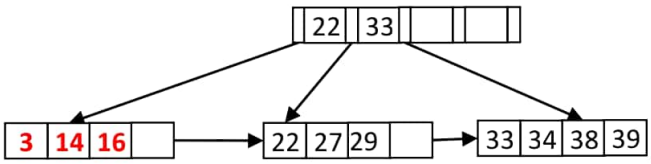
Delete 5: Half full criteria is not satisfied after deleting 5, bringing a value from its siblings also not possible. Therefore merge it with its left sibling (you can also merge with right) and change key values in the internal nodes.



Delete 7: Half full criteria is satisfied even after deleting 7, so just delete 7 from leaf node.



Delete 2: Half full criteria is not satisfied after deleting 2, bringing a value from its siblings also not possible. Therefore merge it with its right sibling and change key values in the internal nodes.



9. INDEXES AND PERFORMANCE TUNING

Indexing is very important to execute DBMS query more efficiently. Adding indexes to important tables is a regular part of performance tuning. When we identify a frequently executed query that is scanning a table or causing an expensive key lookup, then first consideration is if an index can solve this problem. If yes add index for that table.

While indexes can improve query execution speed, the price we pay is on index maintenance. Update and insert operations need to update the index with new data. This means that writes will slow down slightly with each index we add to a table. We also need to monitor index usage and identify when an existing index is no longer needed. This allows us to keep our indexing relevant and trim enough to ensure that we don't waste disk space and I/O on write operations to any unnecessary indexes. To improve performance of the system, we need to do the following:

- Identify the unused indexes and remove them.
- Identify the minimally used indexes and remove them.
- An index that is scanned more frequently, but rarely finds the required answer. Modify the index to reach the answer.
- Identify the indexes that are very similar and combine them.

- o O 0 O o –

Best of Luck *– Ravindar Mogili*