Indexing:

❖The main goal of designing the database is faster access to any data in the database and quicker insert/delete/update to any data.

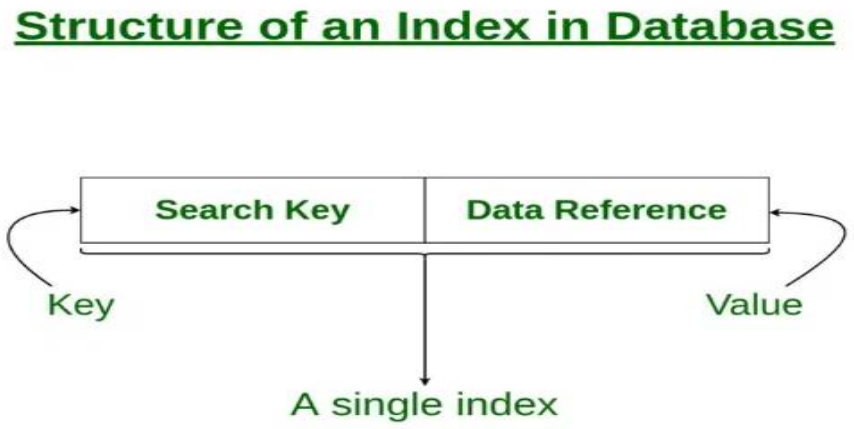
❖In order to reduce the time spent in transactions, Indexes are used. Indexes are similar to book catalogues in library or even like an index in a book.

❖Indexing is a data structure technique which allows you to quickly retrieve records from a database file.

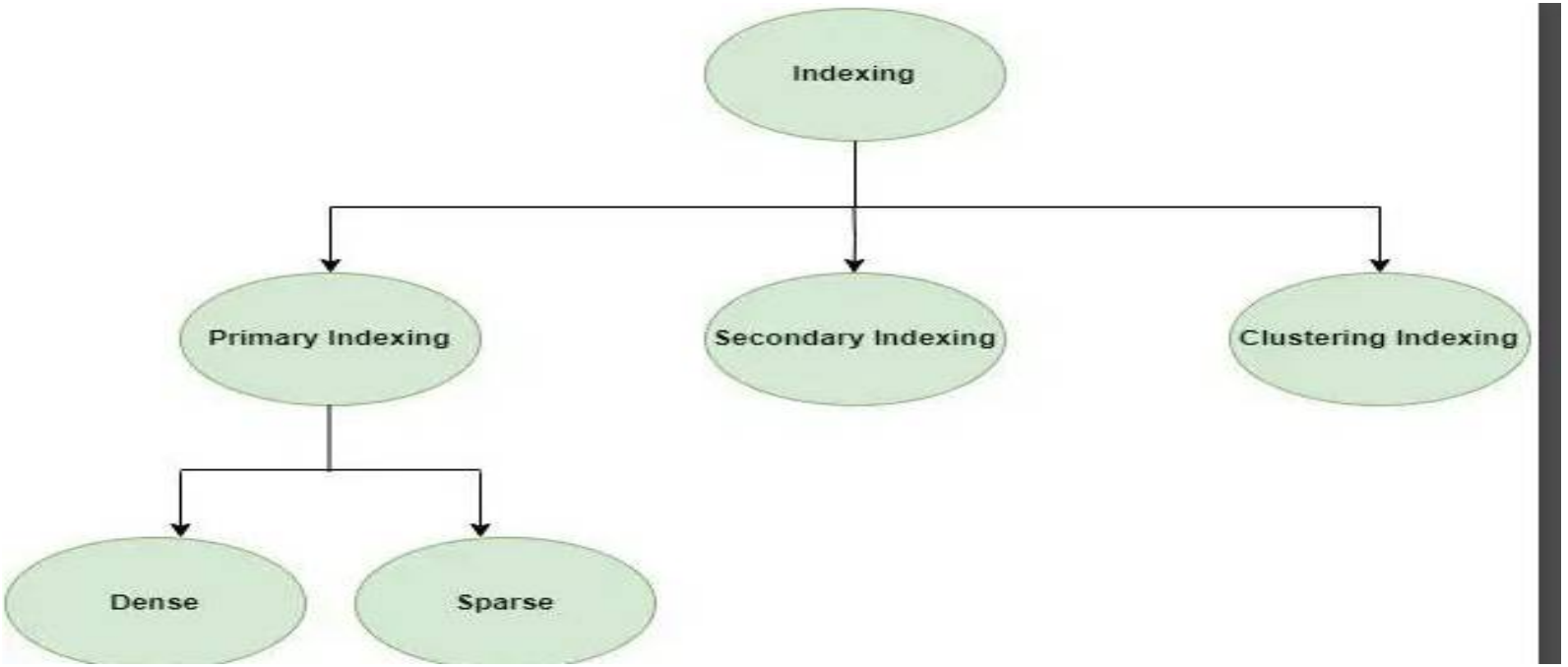
❖An Index is a small table having only two columns.

✓The first column comprises a copy of the primary or candidate key of a table.

✓Its second column contains a set of pointers for holding the address of the disk block where that specific key value stored



Types of Indexes



Primary Index

❖Primary Index is an ordered file which is fixed length size with two fields. The first field is the same a primary key and second, filed is pointed to that specific data block.

❖The primary Indexing in DBMS is also further divided into two types.

❖Dense Index

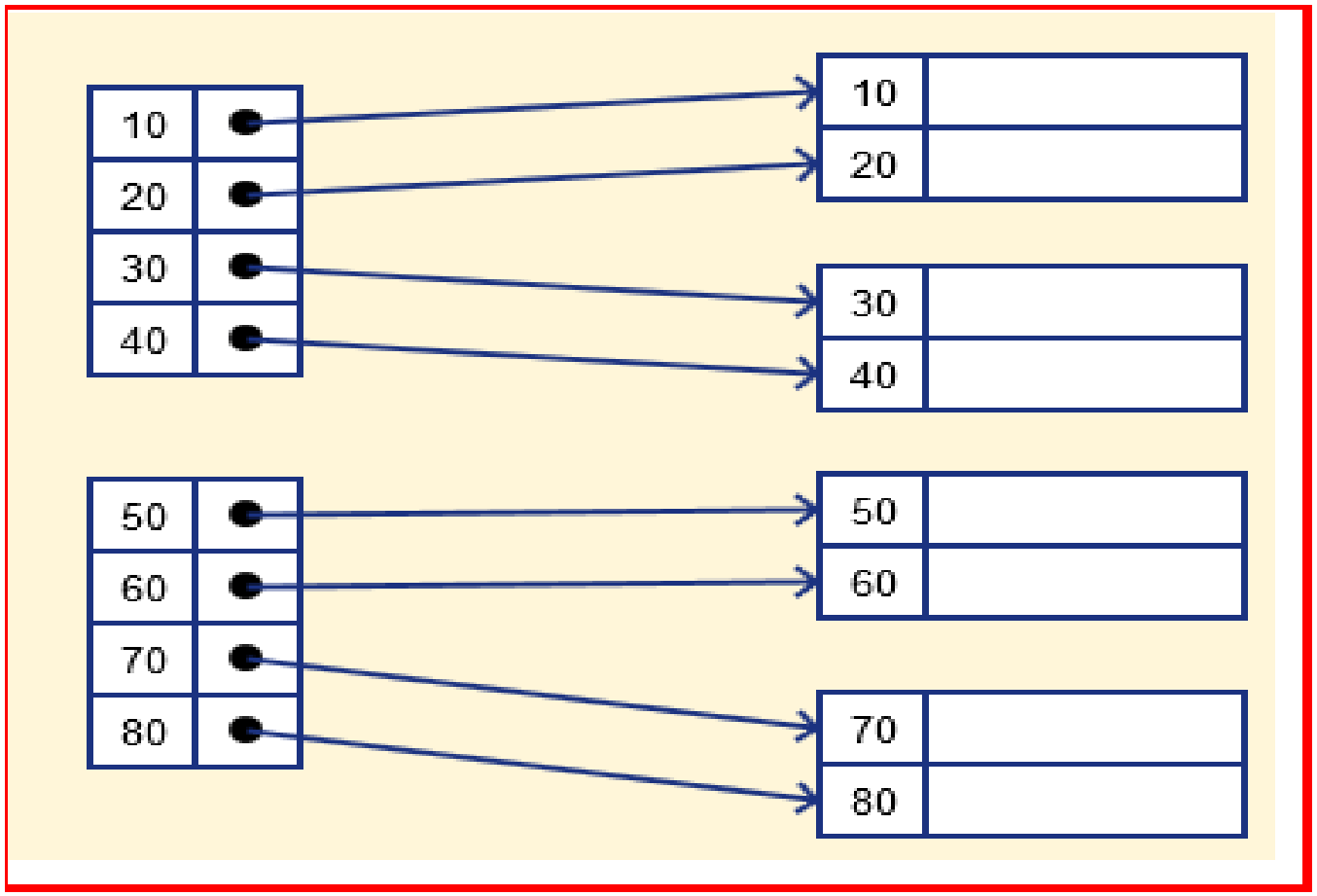
❖Sparse Index

Dense Index:

❖In a dense index, a record is created for every search key valued in the database.

❖This helps you to search faster but needs more space to store index records.

❖In this Indexing, method records contain search key value and points to the real record on the disk.


10
20
30
40
50
60
70
80
10
20
30
40
50
60
70
80

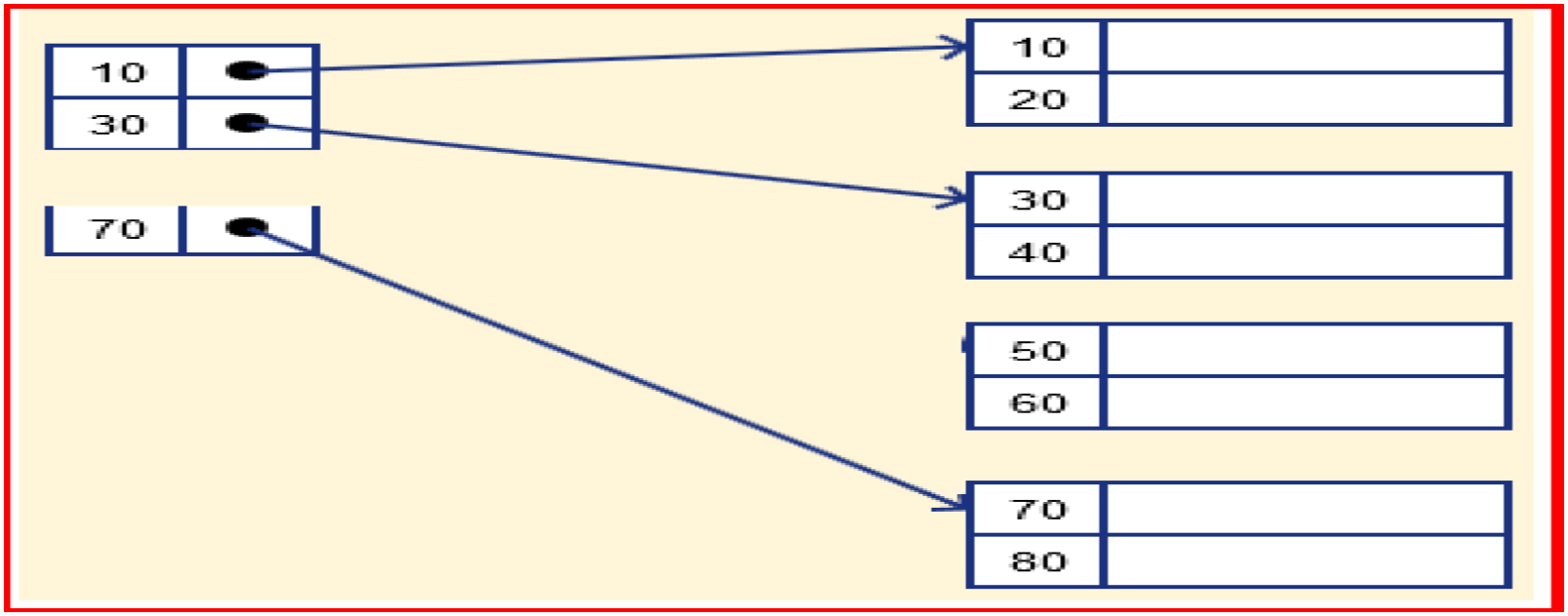
Sparse Index

❖It is an index record that appears for only some of the values in the file.

❖In this method of indexing technique, a range of index columns stores the same data block address, and when data needs to be retrieved, the block address will be fetched.

❖It needs less space, less maintenance overhead for insertion, and deletions .

❖but It is slower compared to the dense Index for locating records.

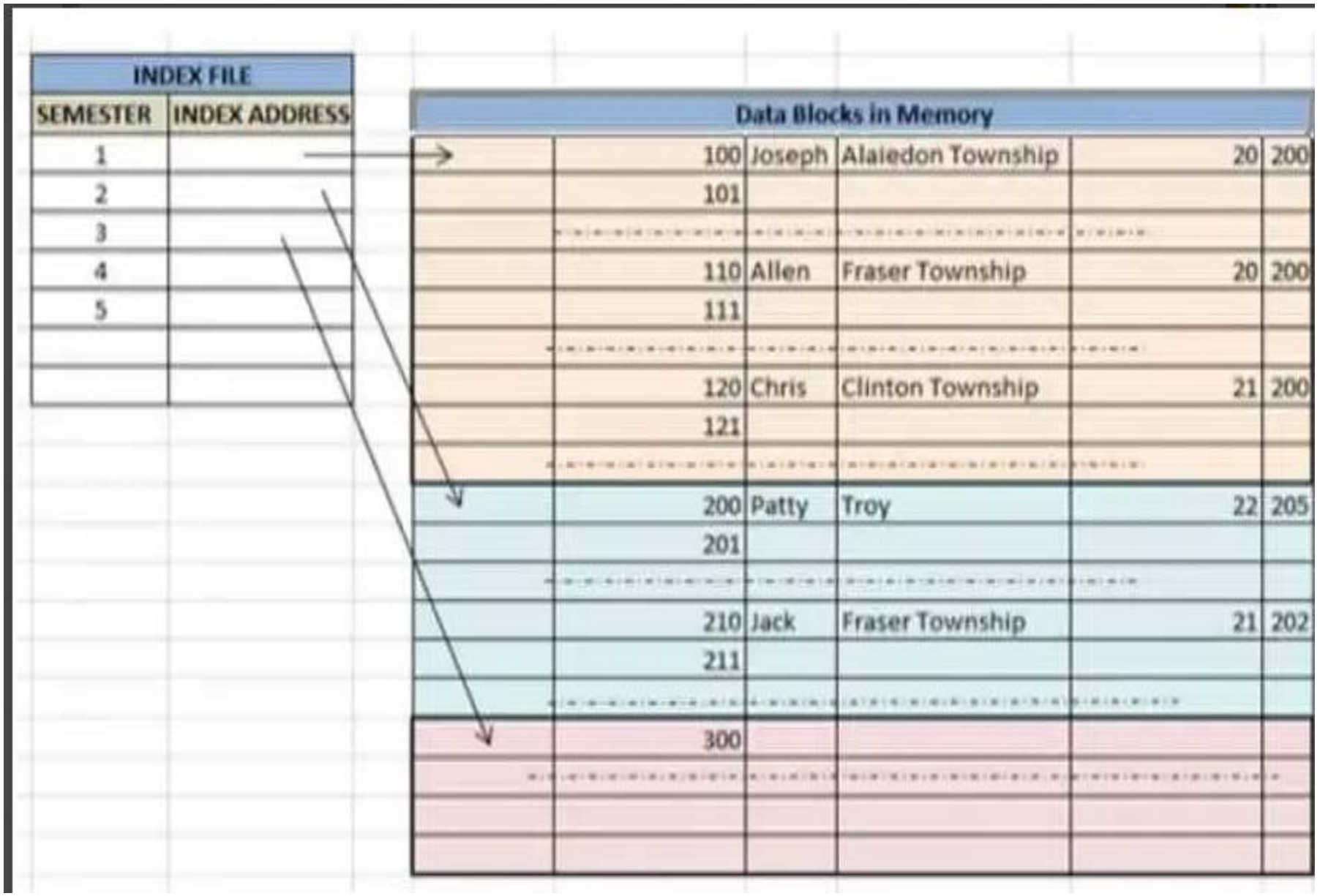


Clustered Indexing

❖Clustering index is defined on an ordered data file. The data file is ordered on a non-key field.

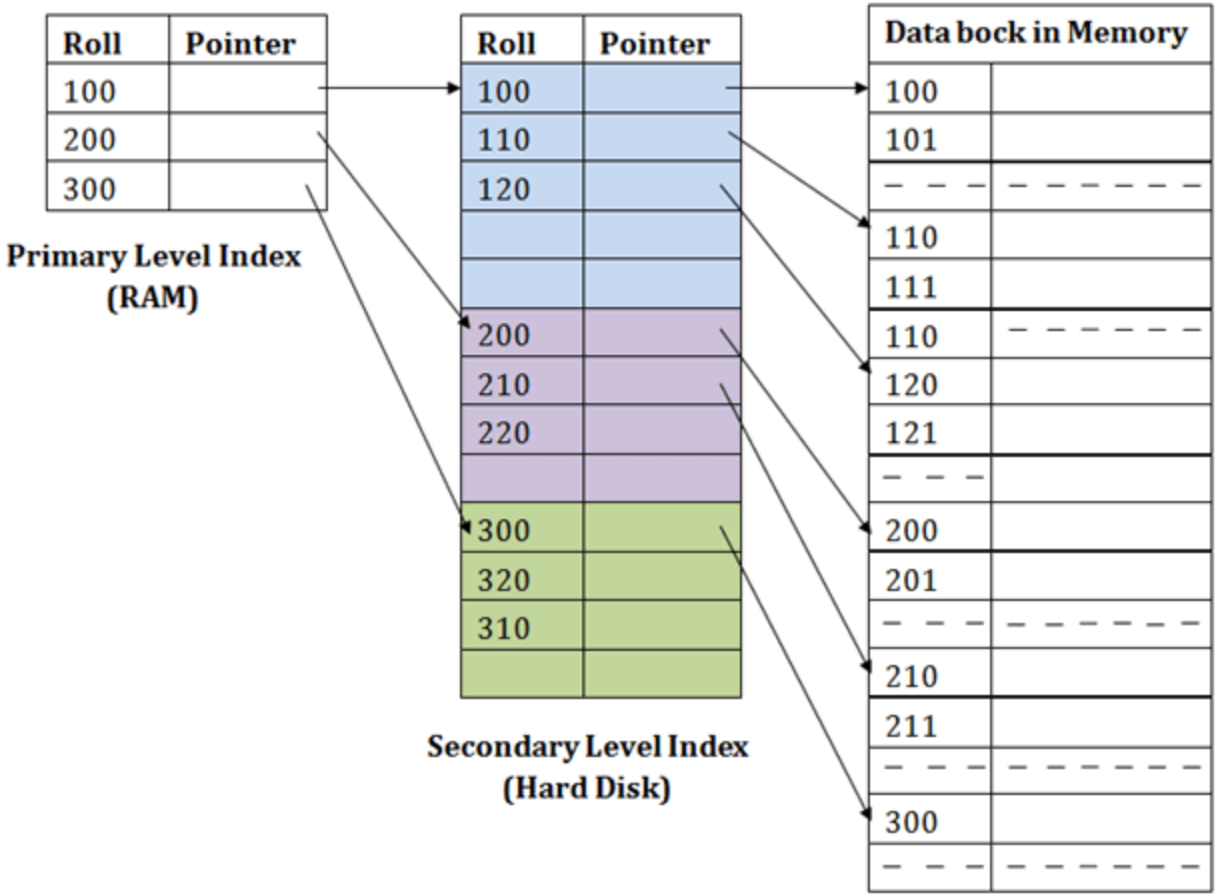
❖In some cases, the index is created on non-primary key columns which may not be unique for each record.

❖In such cases, in order to identify the records faster, we will group two or more columns together to get the unique values and create index out of them. This method is known as **clustering index**.


INDEX FILE
SEMESTER
INDEX ADDRESS
1
2
3
4
5
Data Blocks in Memory
100 Joseph Alaiedon Township 20 200
101
110 Allen Fraser Township 20 200
111
120 Chris Clinton Township 21 200
121
200 Patty Troy 22 205
201
210 Jack Fraser Township 21 202
211
300

Secondary Index:

Secondary index may be generated from a field which is a candidate key and has a unique value in every record, or a non-key with duplicate values.

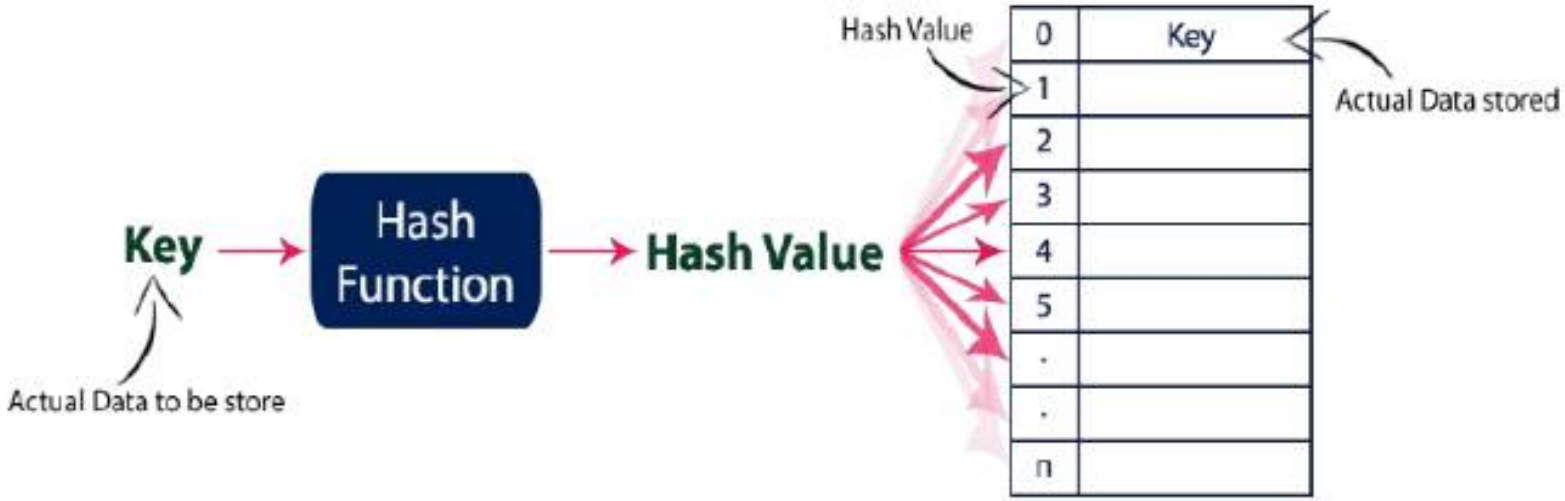


Hash Based Indexing

❖Hashing is the technique of the database management system, which directly finds the specific data location on the disk **without using the concept of index structure.**

❖In the database systems, **data is stored at the blocks whose data address is produced by the hash function**. That location of memory where hash files stored these records is called as **data bucket or data block.**

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



Terms used in Hashing:

Data Bucket: That location of memory where the hash file stored the records. It is also referred to as a Unit of storage**.**

Key: A key in the Database Management system (DBMS) is a field or set of fields that helps the relational database users to uniquely identify the row/records of the database table.

Hash function: This mapping function matches all the set of search keys to those addresses where the actual records are located. It is an easy mathematics function.

Linear Probing: It is a concept in which the next available block of data is used for inserting the new record instead of overwriting the older data block.

Quadratic Probing: It is a method that helps users to determine or find the address of a new data bucket.

Double Hashing: Double hashing is a computer programming method used in hash tables to resolve the issues of has a collision.

Bucket Overflow: When a record is inserted, the address generated by the hash function is not empty or data already exists in that address .This critical situation is called bucket overflow.

There are mainly two types of SQL hashing methods:

•Static Hashing

•Dynamic Hashing

Static Hashing:

❖In the static hashing, the resultant data bucket address will always remain the same.

❖Therefore, if you generate an address for say Student_ID = 10 using hashing function mod(3), the resultant bucket address will always be 1. So, you will not see any change in the bucket address.

❖Therefore, in this static hashing method, the number of data buckets in memory always remains constant.

Static Hash Functions

Inserting a record: When a new record requires to be inserted into the table, you can generate an address for the new record using its hash key. When the address is generated, the record is automatically stored in that location.

Searching: When you need to retrieve the record, the same hash function should be helpful to retrieve the address of the bucket where data should be stored.

Delete a record: Using the hash function, you can first fetch the record which is you wants to delete. Then you can remove the records for that address in memory.

Static Hashing Example

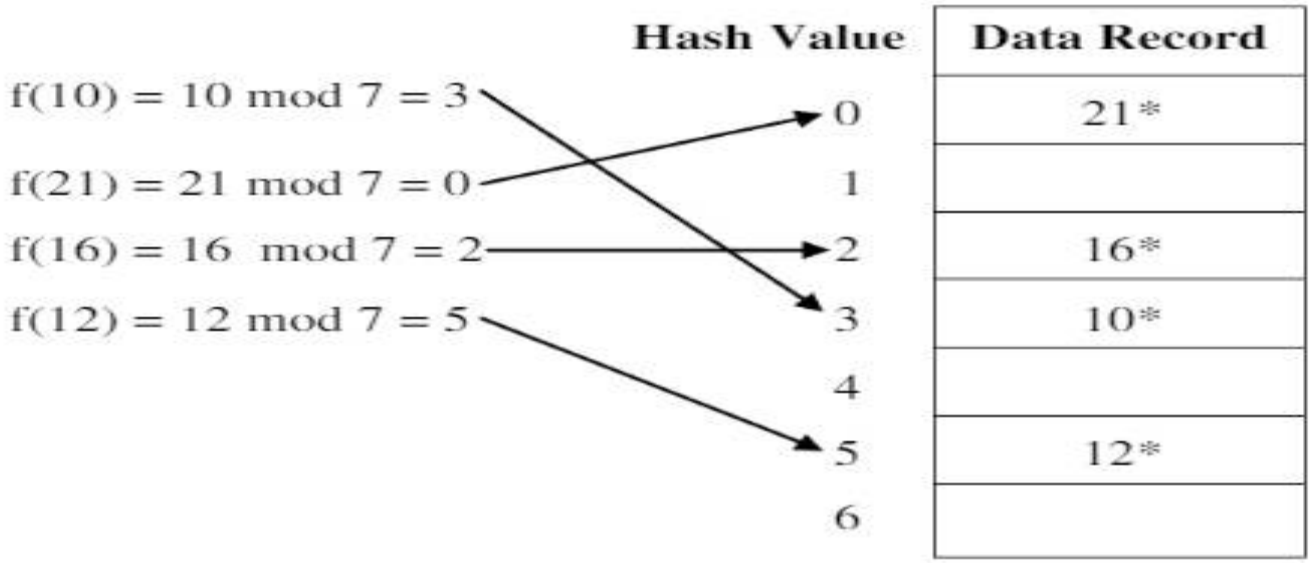


Figure 5.1: *Static hashing*

❖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.

1. Open Addressing
2. Closed Addressing

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

- When a hash function generates at which data is already stored, then the next bucket will be allocated to it.
- This mechanism is called Linear probing.

➢**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.

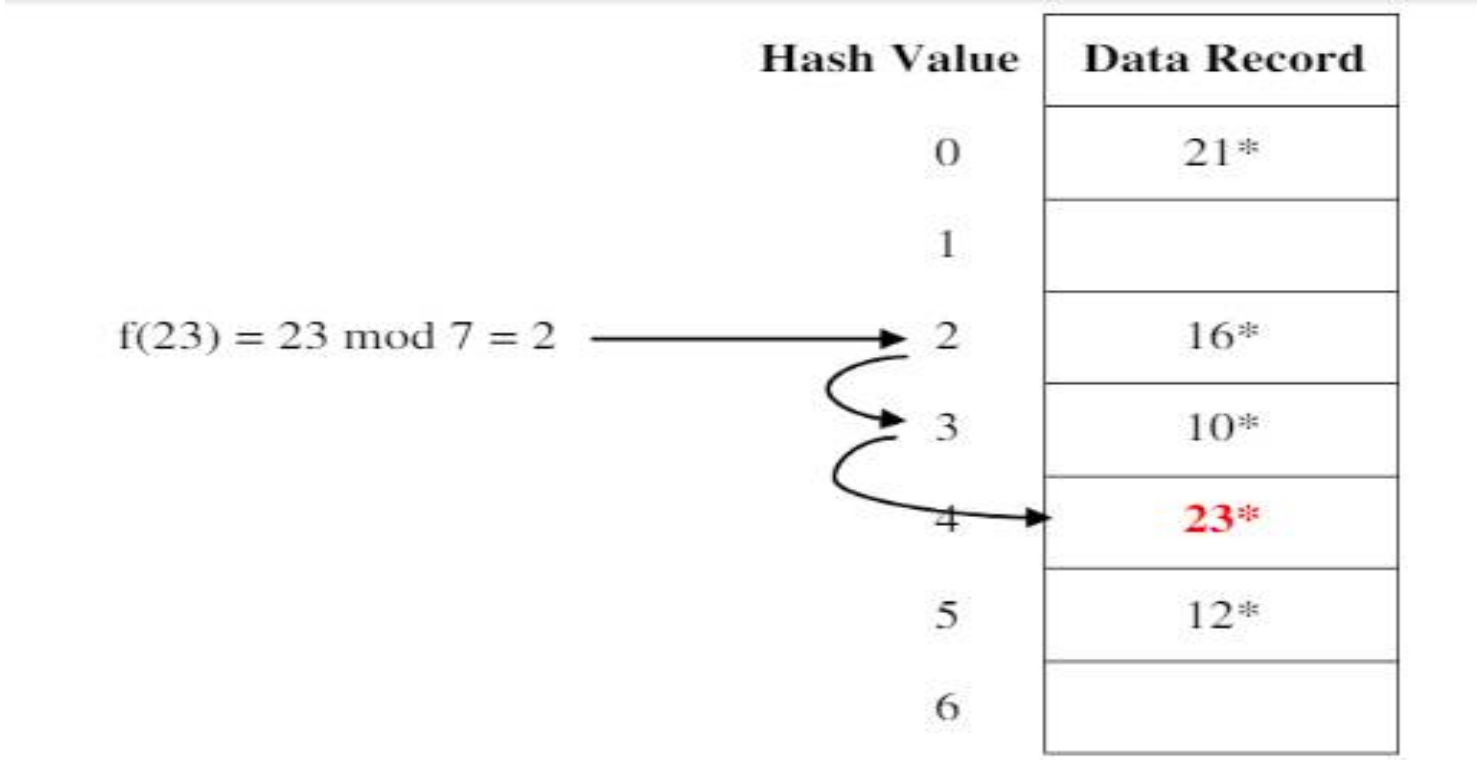


Figure 5.2: *Linear Probing*

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.

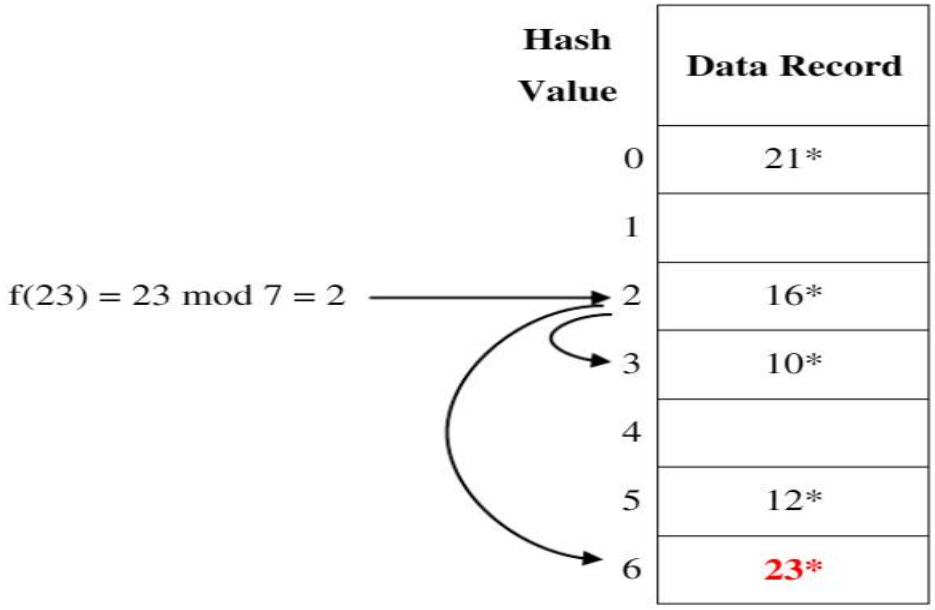


Figure 5.3: *Quadratic Probing*

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

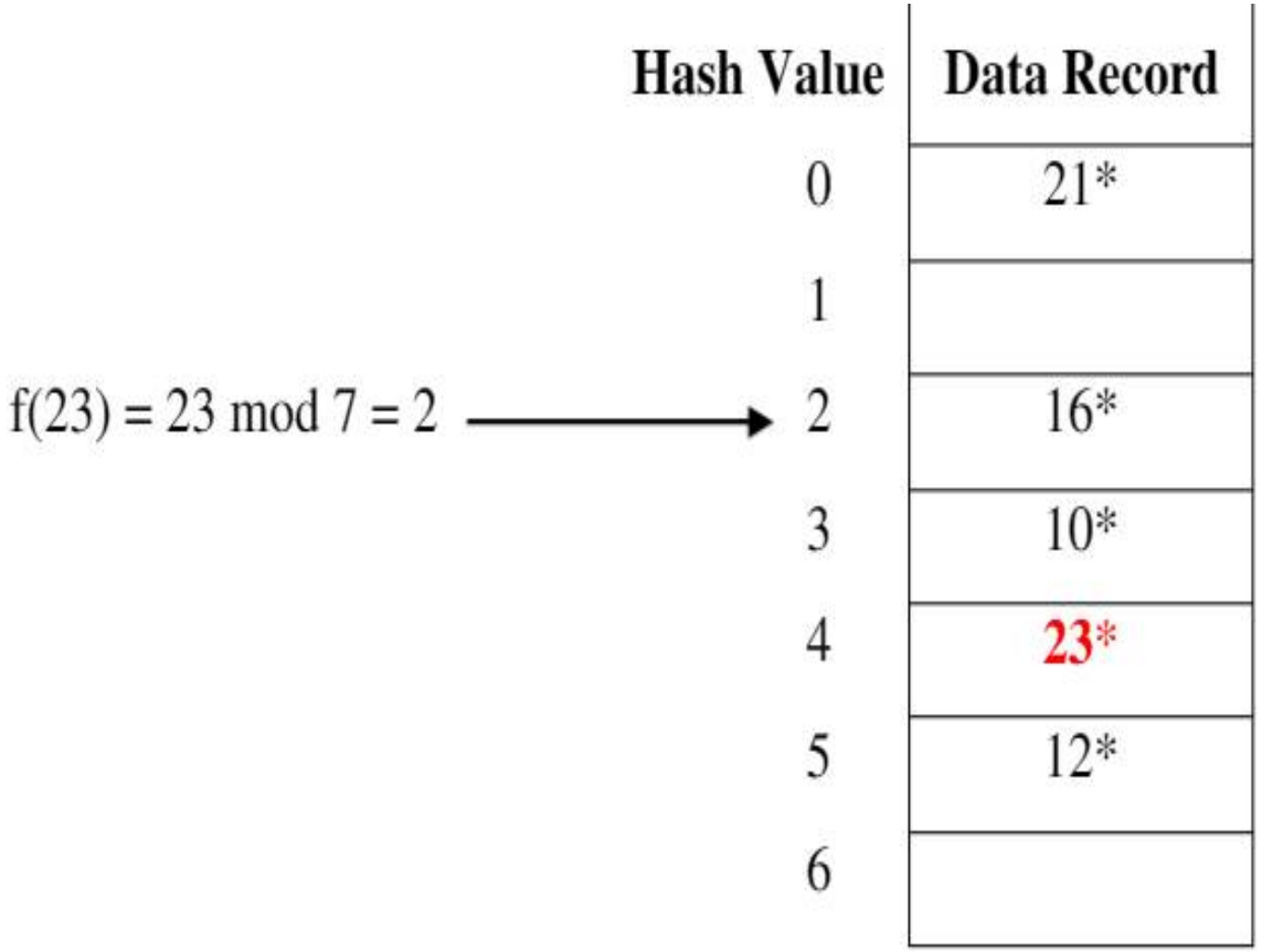


Figure 5.4: *Double Probing*

Closed Addressing or Seperate Chaining

❖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.

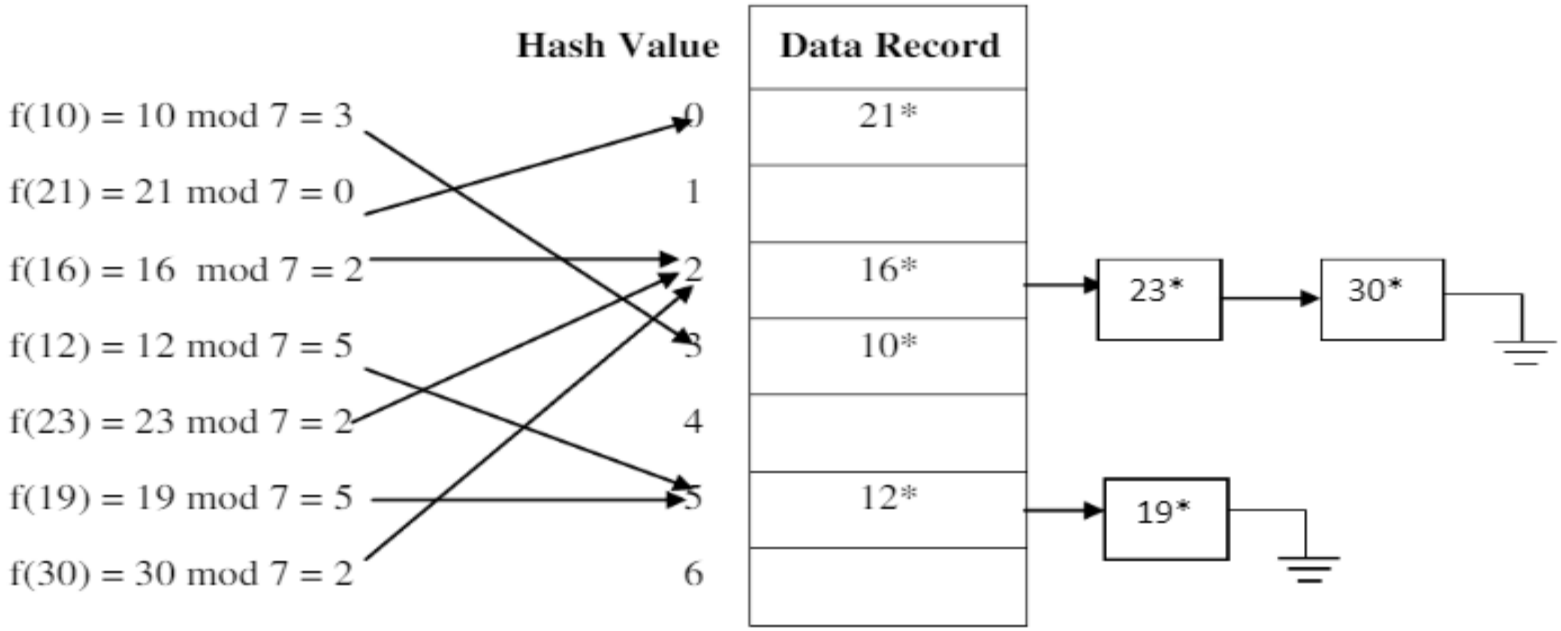


Figure 5.5: *Separate chaining example*

Dynamic Hashing

- The dynamic hashing method is used to overcome the problems of static hashing like bucket overflow.
- In this method, data buckets grow or shrink as the records increases or decreases. This method is also known as Extendable hashing method.
- This method makes hashing dynamic, i.e., it allows insertion or deletion without resulting in poor performance.

How to search a key

- First, calculate the hash address of the key.
- Check how many bits are used in the directory, and these bits are called as i.
- Take the least significant i bits of the hash address. This gives an index of the directory.
- Now using the index, go to the directory and find bucket address where the record might be.

How to insert a new record

- Firstly, you have to follow the same procedure for retrieval, ending up in some bucket.
- If there is still space in that bucket, then place the record in it.
- If the bucket is full, then we will split the bucket and redistribute the records.

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	=	11001
13	=	01101	11	=	01011

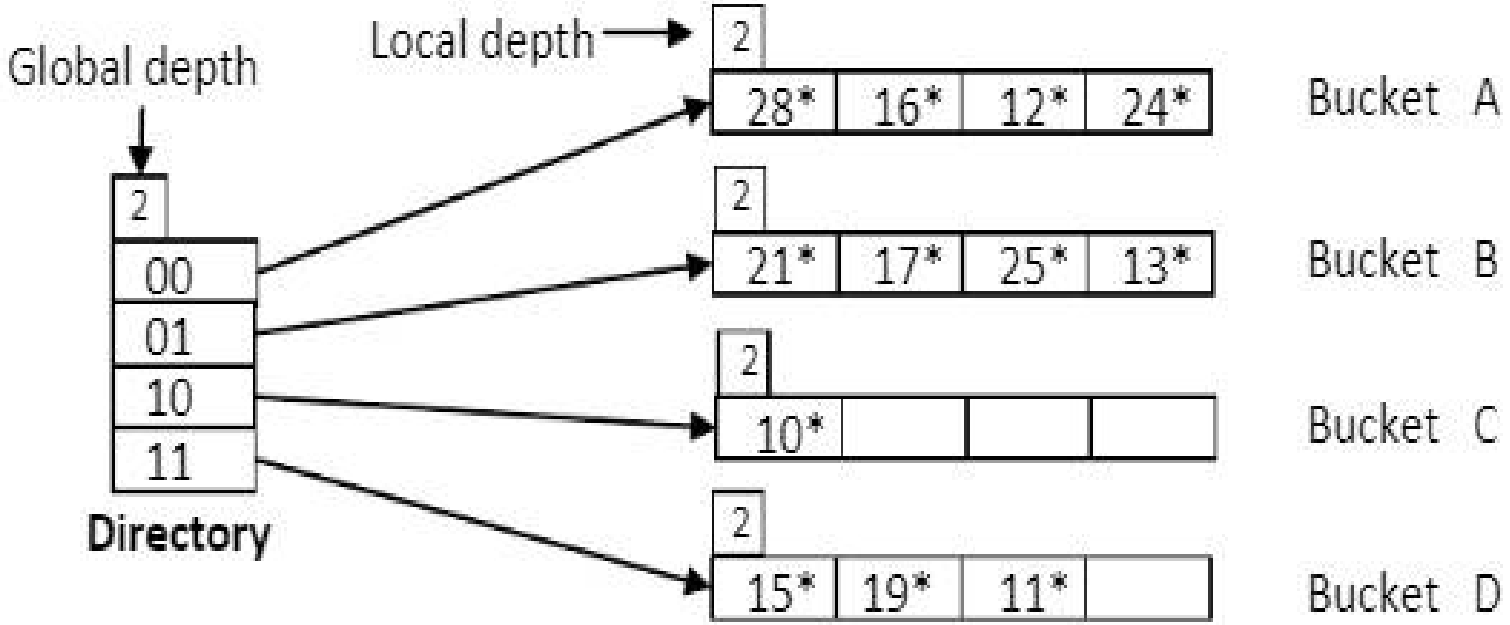


Figure 6.1: *Extendible hashing example*

INSERT 20* (binary 10100)

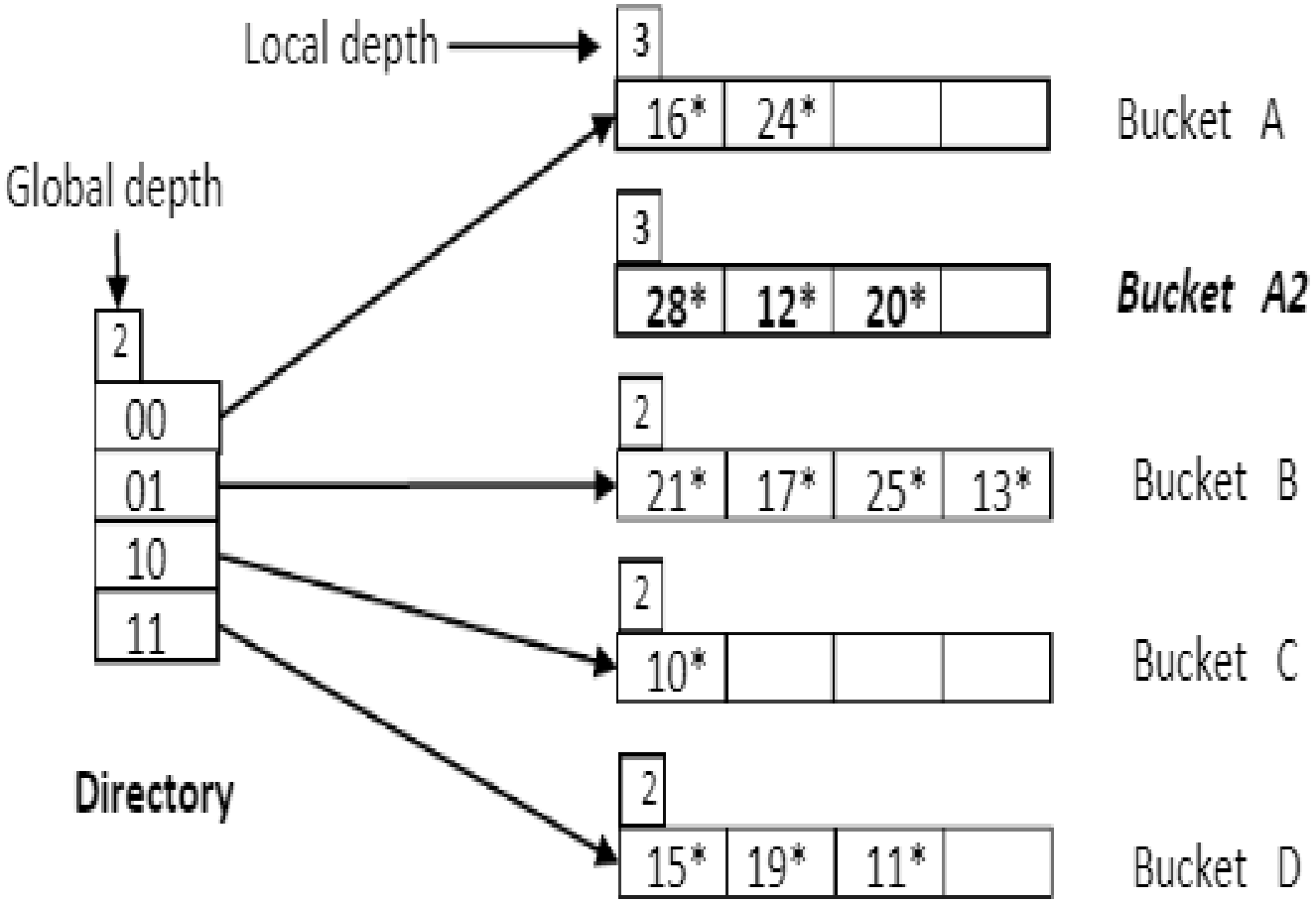


Figure 6.2: After inserting 20 and splitting Bucket A

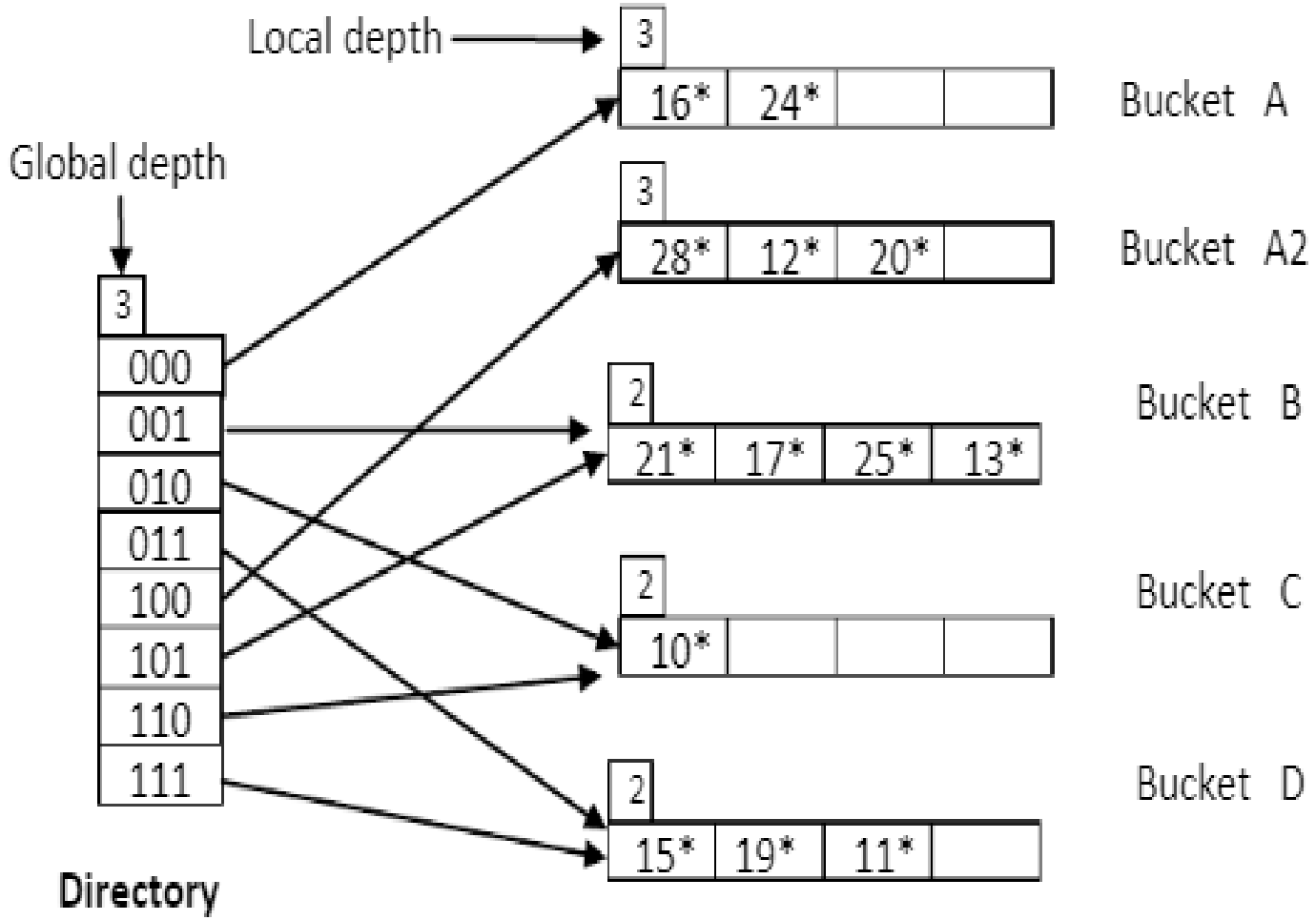


Figure 6.3: *After inserting 20 and doubling the directory*

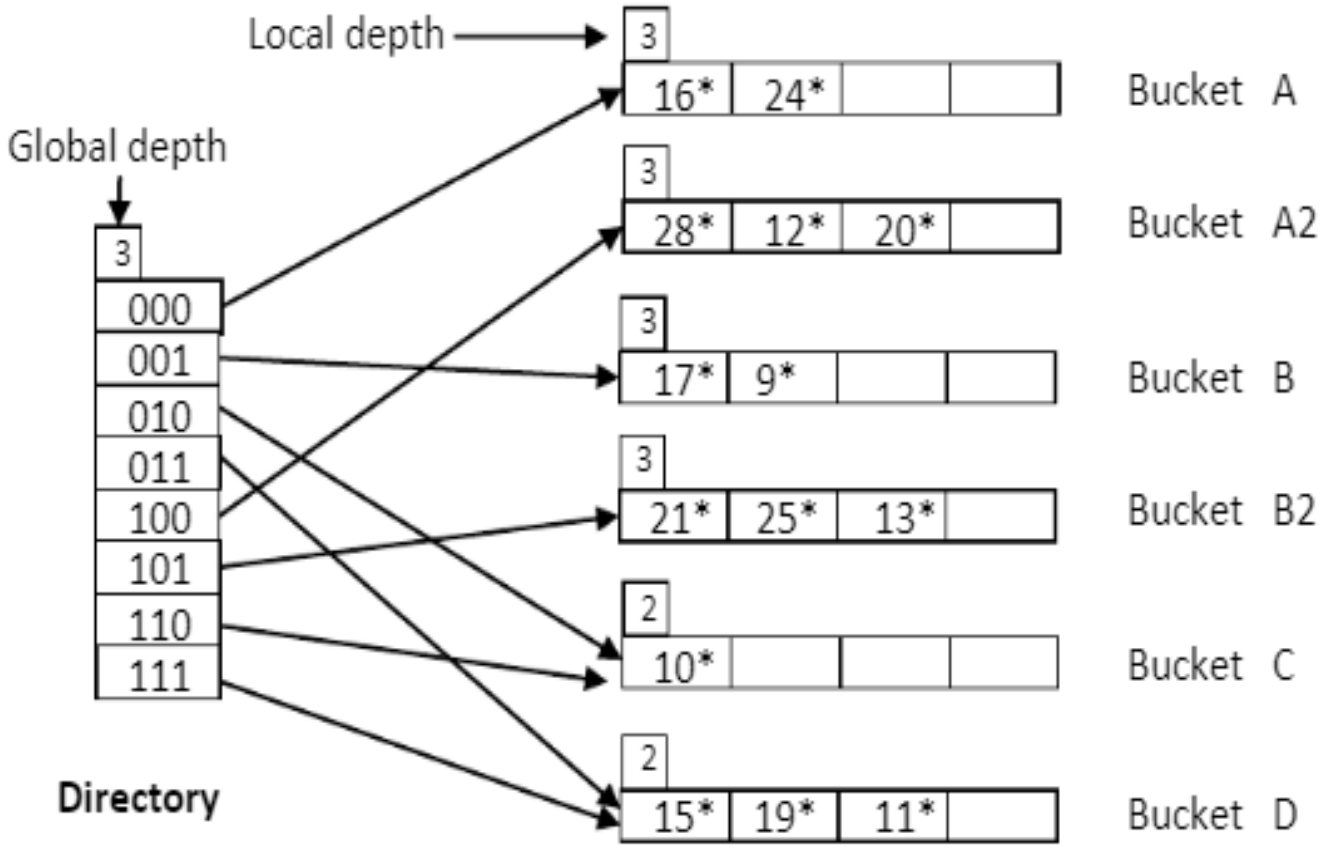


Figure 6.4: *After inserting 9*

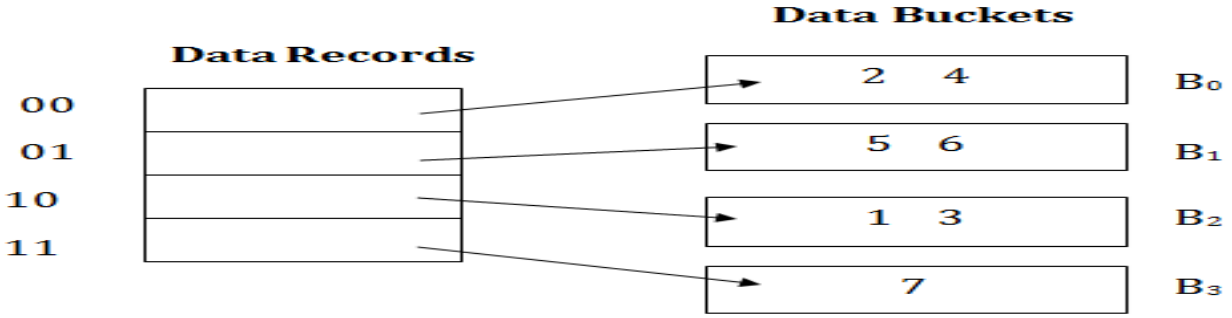
Dynamic hashing

EXTRA SIMPLE Example:

Consider the following grouping of keys into buckets, depending on the prefix of their hash address:

Key	Hash address
1	11010
2	00000
3	11110
4	00000
5	01001
6	10101
7	10111

The last two bits of 2 and 4 are 00. So it will go into bucket B0.
The last two bits of 5 and 6 are 01, so it will go into bucket B1.
The last two bits of 1 and 3 are 10, so it will go into bucket B2.
The last two bits of 7 are 11, so it will go into B3.



Insert key 9 with hash address 10001 into the above structure:

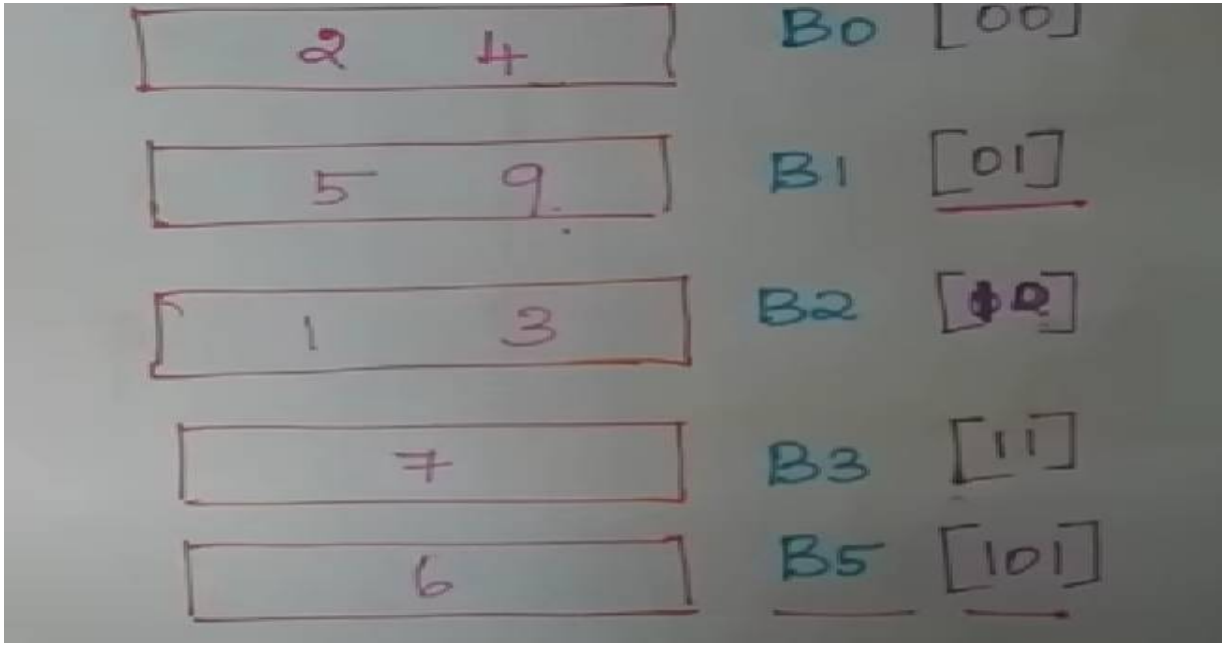
•Since key 9 has hash address 10001, it must go into the first bucket. But bucket B1 is full, so it will get split.

•The splitting will separate 5, 9 from 6 since last three bits of 5, 9 are 001, so it will go into bucket B1, and the last three bits of 6 are 101, so it will go into bucket B5.

•Keys 2 and 4 are still in B0. The record in B0 pointed by the 000 and 100 entry because last two bits of both the entry are 00.

•Keys 1 and 3 are still in B2. The record in B2 pointed by the 010 and 110 entry because last two bits of both the entry are 10.

•Key 7 are still in B3. The record in B3 pointed by the 111 and 011 entry because last two bits of both the entry are 11.



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 h0, h1, h2, ... , 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 h0(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.

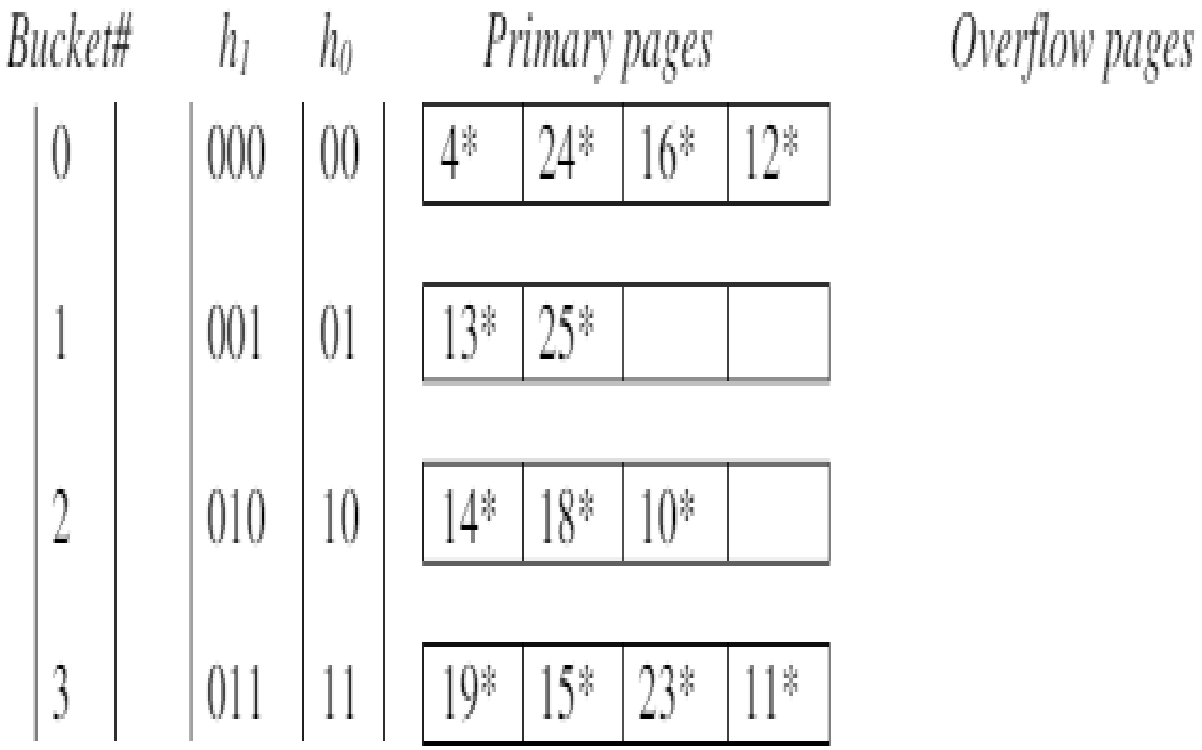
❖That is, **for first time overflow in any bucket**, bucket 0 will be splited, for **second time overflow in any bucket**; bucket 1 will be splited and so on.

❖When number of buckets **becomes 2N**, then this marks the **end of splitting round 0.**

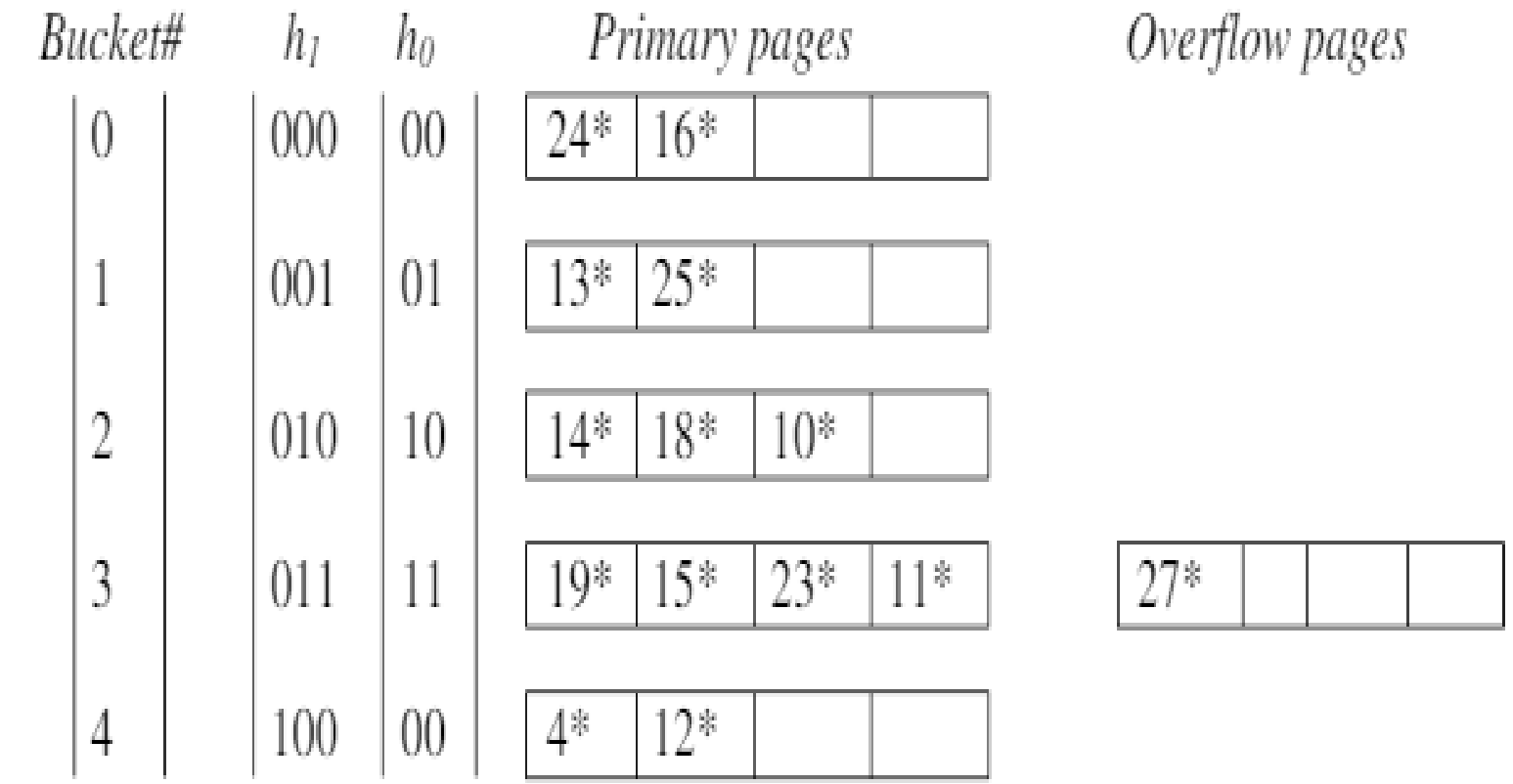
❖Hashing function **h0 is no longer needed as all 2N buckets can be addressed by hashing function h1.**

❖In new round namely splitting-round 1, bucket split once again starts from bucket 0. A new hash function h2 will be used. This process is repeated when the hash file grows.

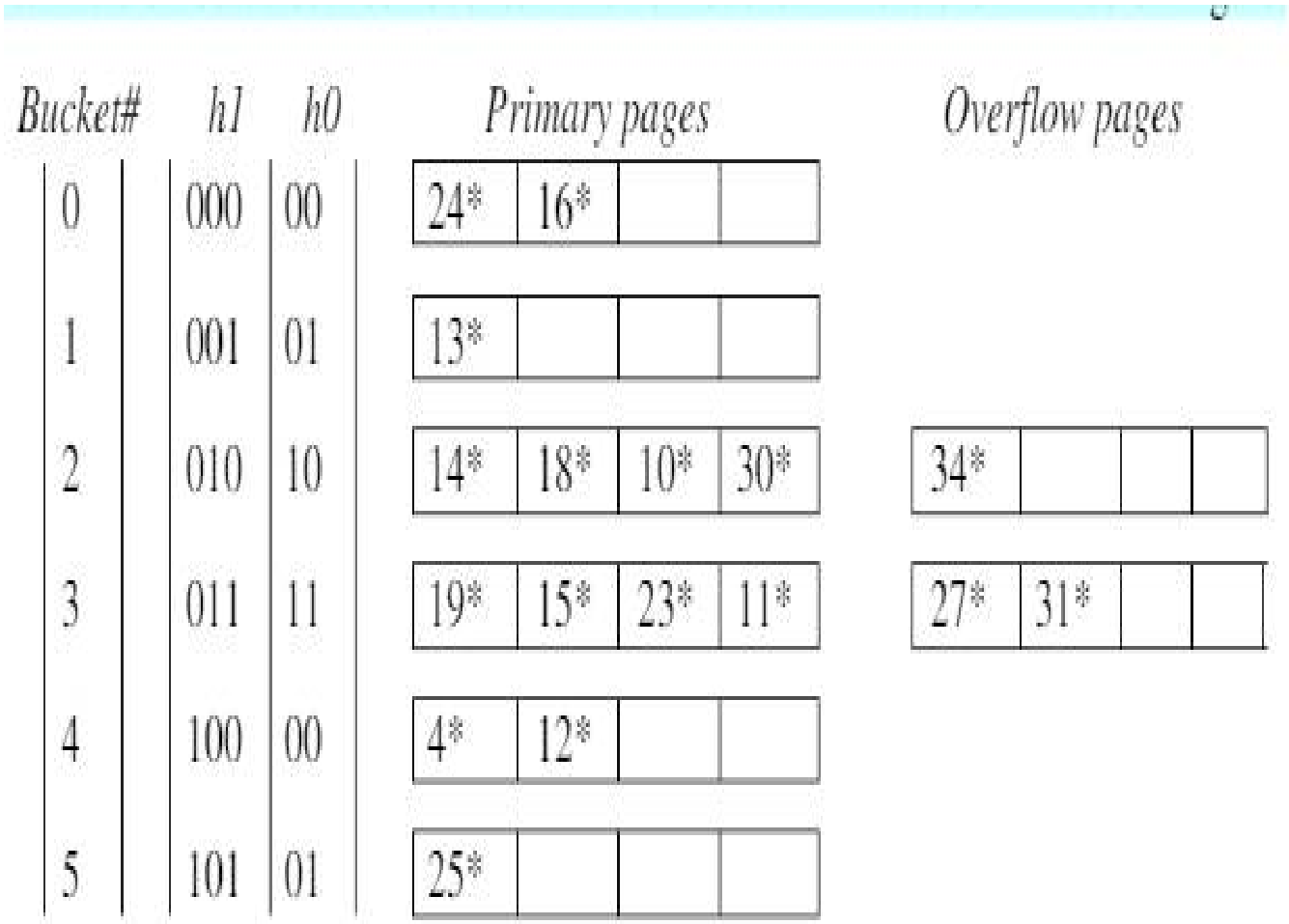
Example: Let N = 4, so we use 4 buckets and hash function h0(key) = key % 4 is used to map any key into one of the four buckets. Let us initially insert 4, 13, 19, 25, 14, 24, 15, 18, 23, 11, 16, 12 and 10.This is shown in the below figure.



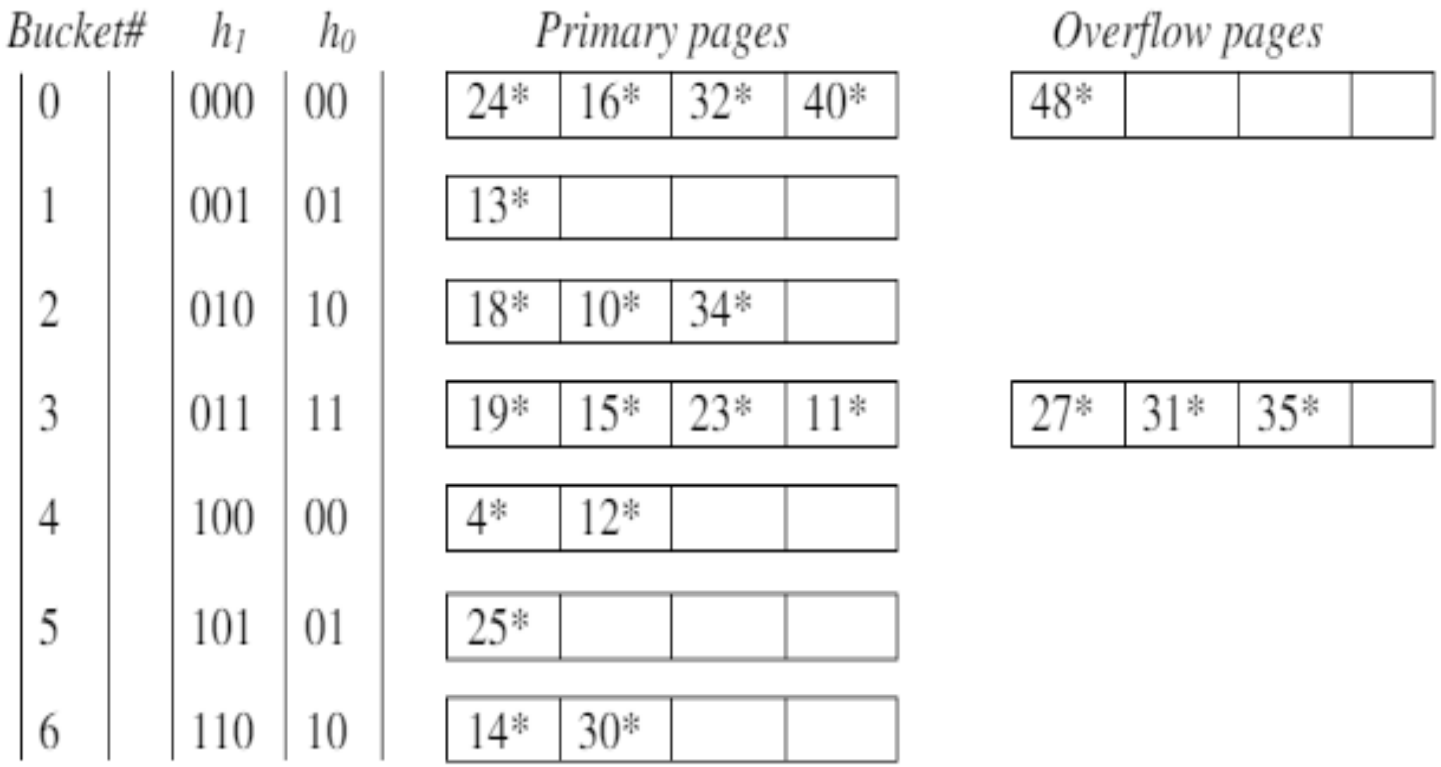
❖Next, when 27 is inserted, an overflow occurs in bucket 3. So, bucket 0 (first bucket) is splited and its content is distributed between bucket 0 and new bucket. This is shown in below figure



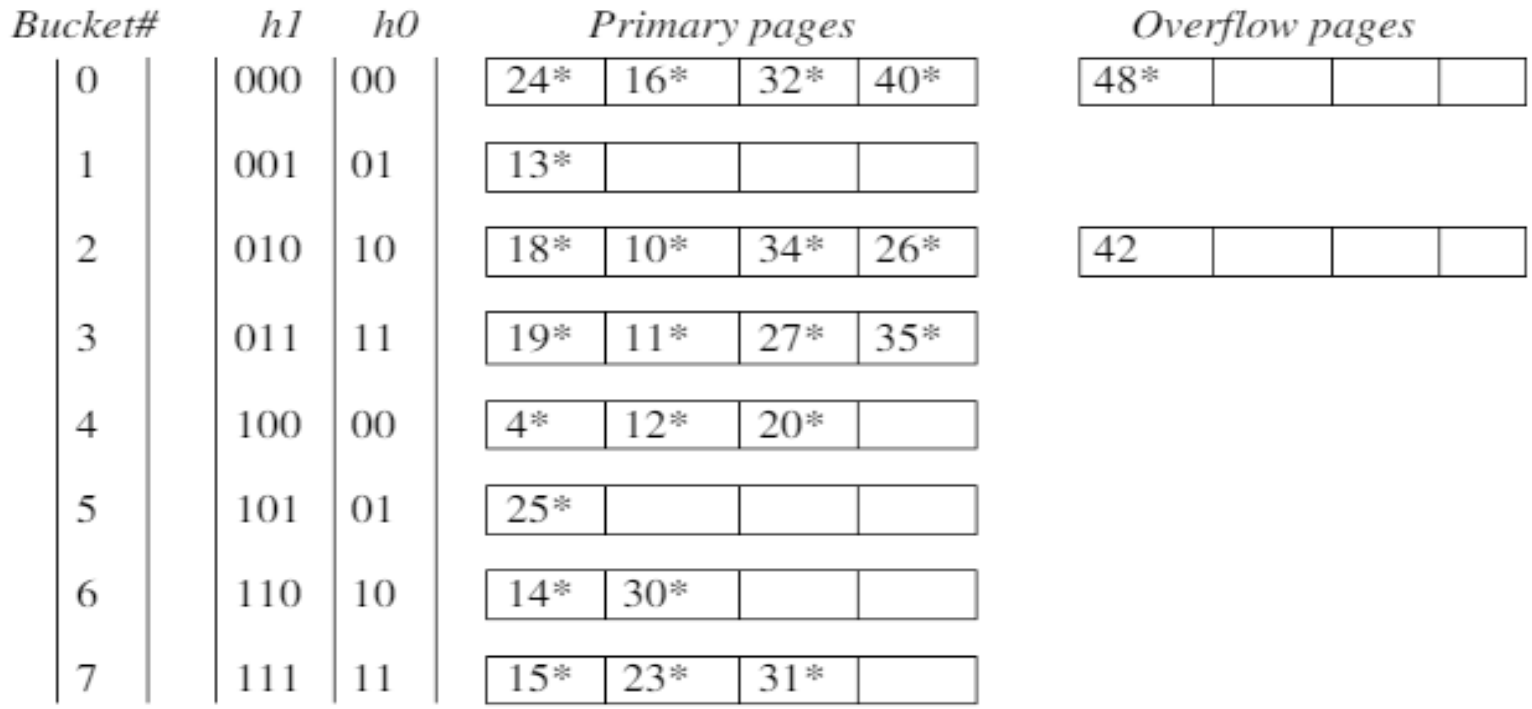
Next, when 30, 31 and 34 is inserted, an overflow occurs in bucket 2. So, bucket 1 is splited and its content is distributed between bucket 1 and new bucket. This is shown in below figure.



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.



❖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 **h0 is no longer needed as all 2N buckets can be addressed by hashing function h1**. In new round namely splitting-round 1, bucket split once again starts from bucket 0. A new hash function h2 will be used. This process is repeated

INTUITIONS FOR TREE INDEXES

❖We can **use tree-like structures as index as well**. For example, a binary search tree(BST) 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 above mentioned 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**

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.

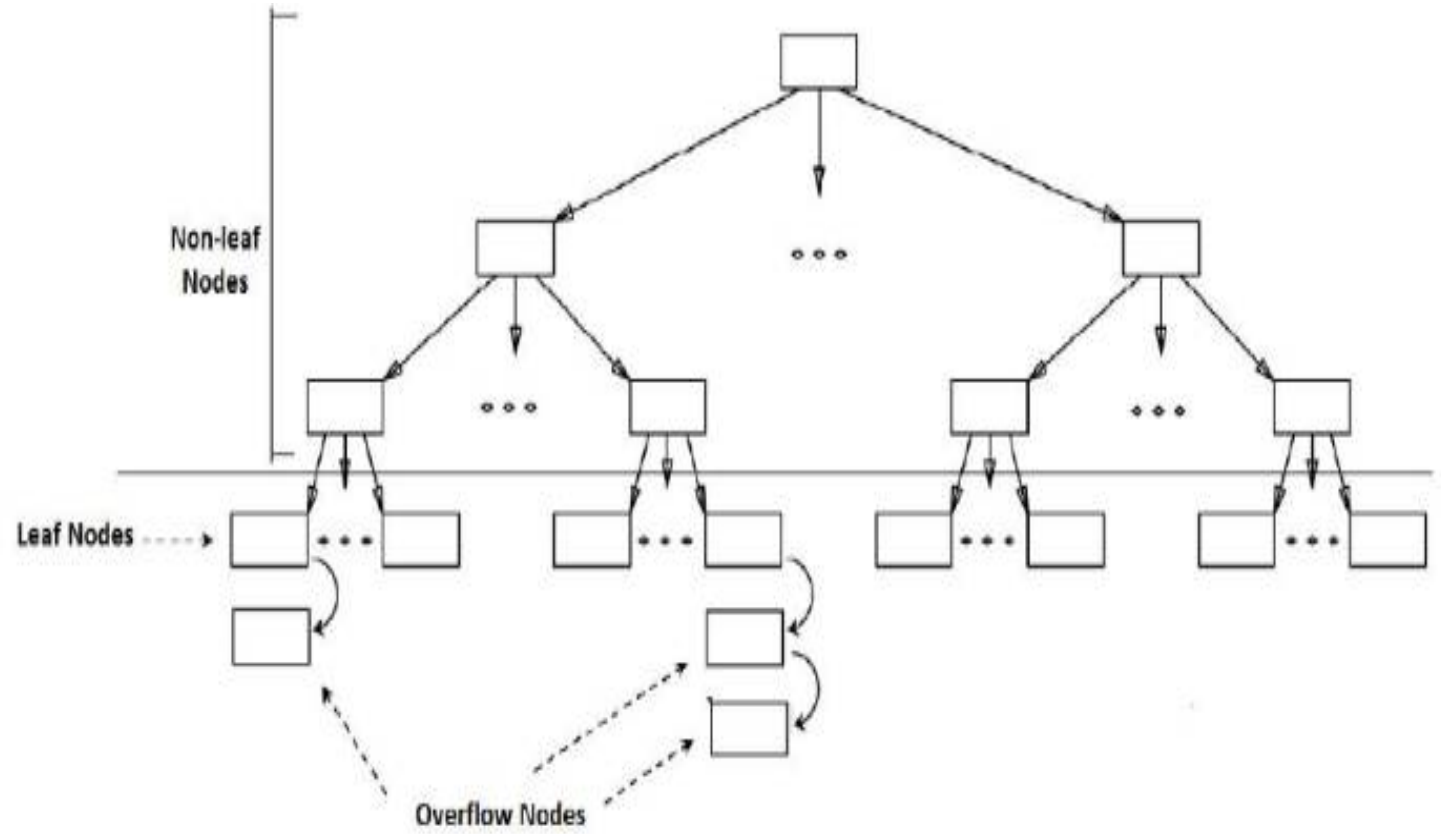
❖ISAM gives flexibility to generate index on other fields also in addition to primary key fields.

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.



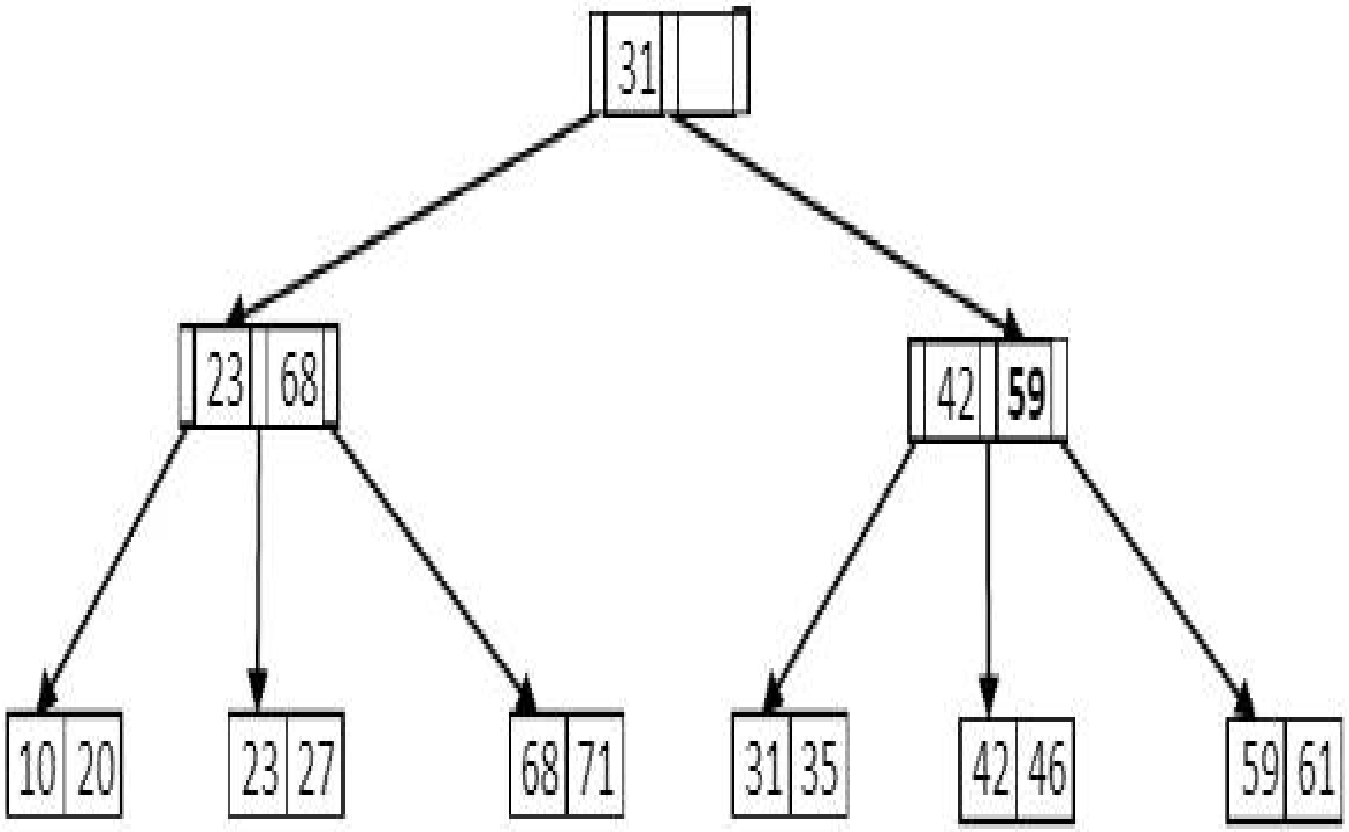
On ISAM, we can perform search, insertion and deletion operations.

❖**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.

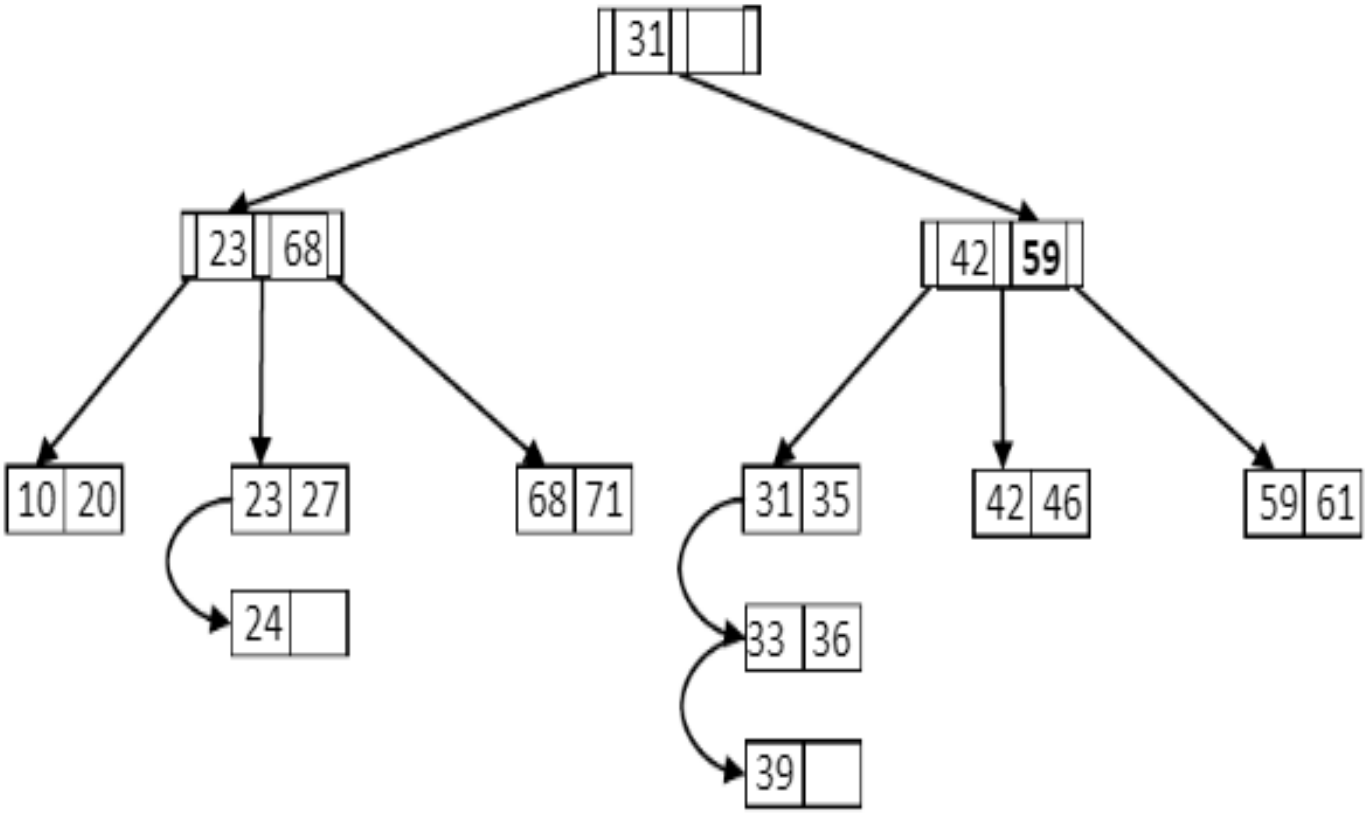
❖**Insertion operation**: First locate a leaf node where the insertion to be take place (use binary search). After finding leaf node, insert it in that leaf node if space is available, else create an overflow node and insert the record index in it, and link the overflow node to the leaf node.

❖**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.

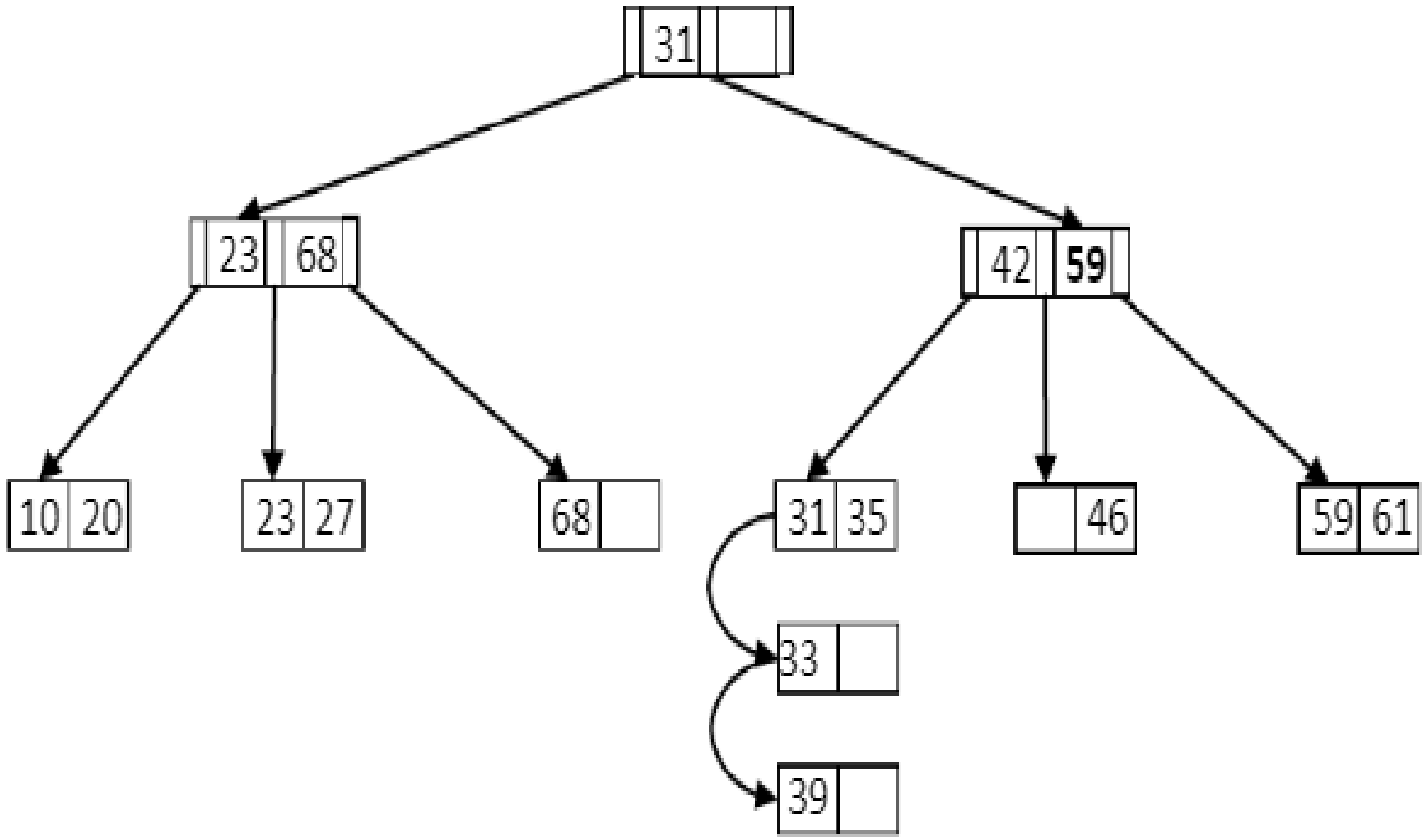
Example: Insert 10, 23, 31, 20, 68, 35, 42, 61, 27, 71, 46 and 59



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



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),

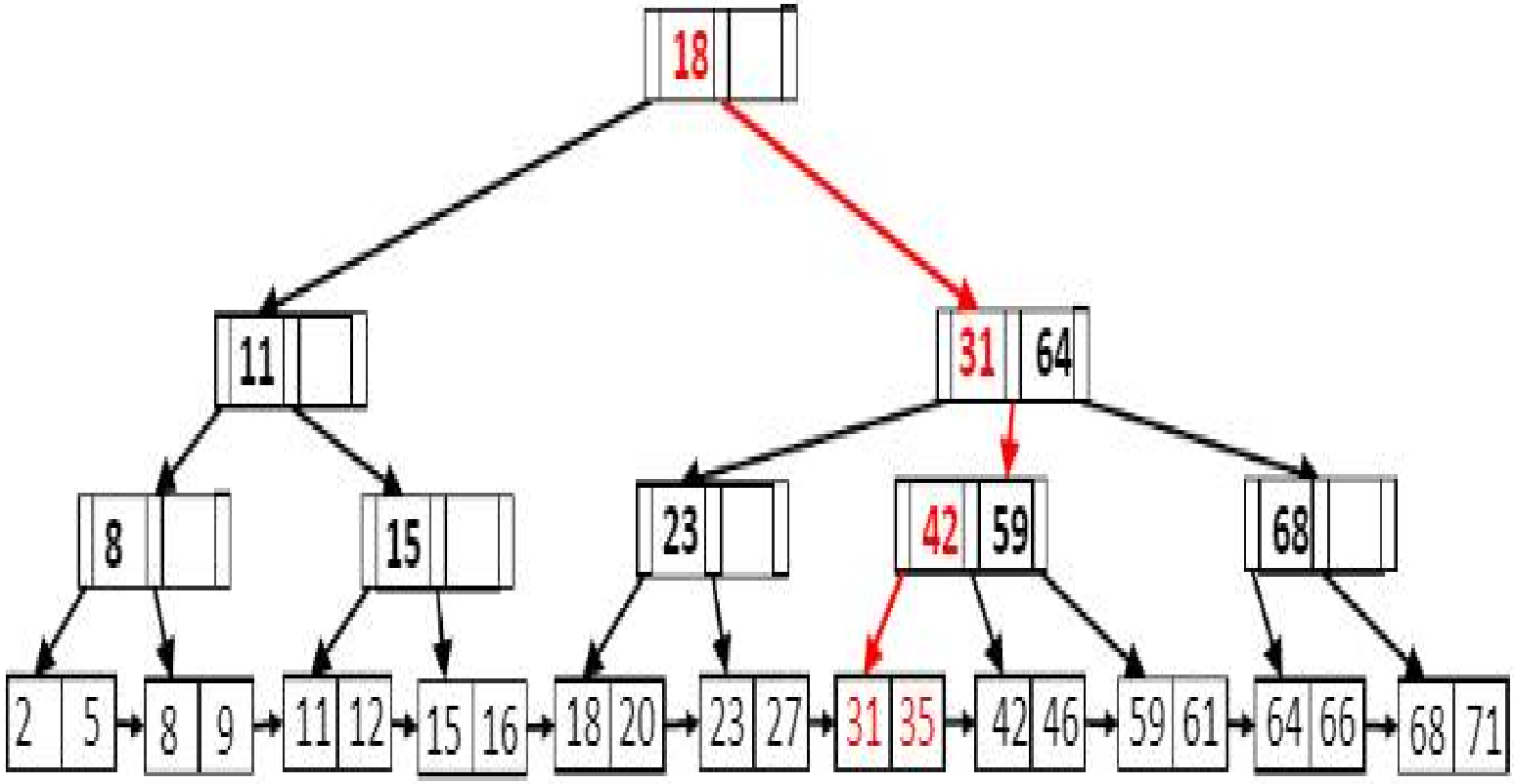
a. If all the values in the non-leaf node are greater than X, then move to its first child

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 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

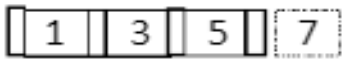
Initially

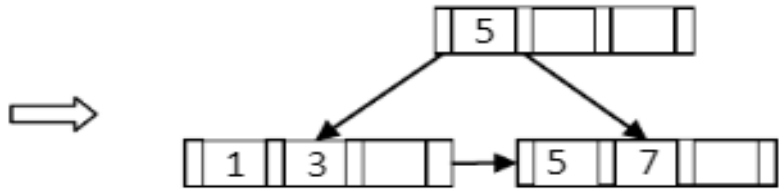
After inserting 1

After inserting 5

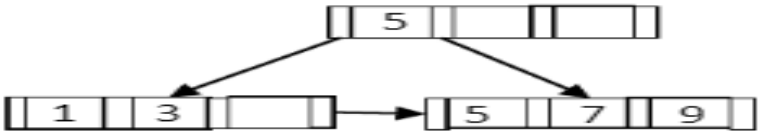
After inserting 3

After inserting 7

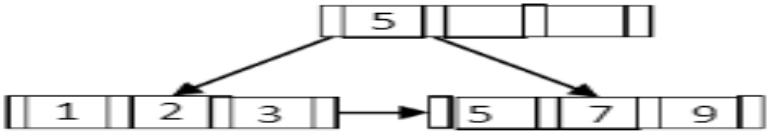




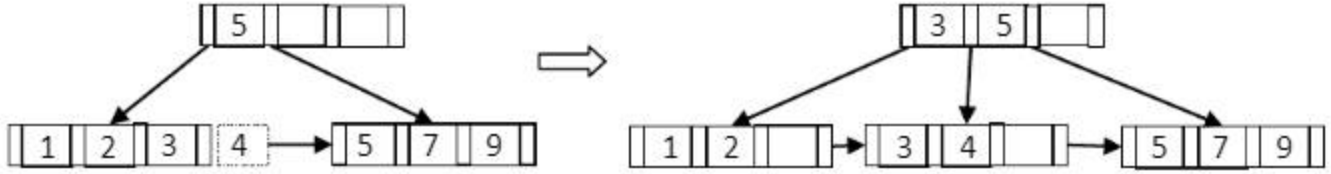
After inserting 9



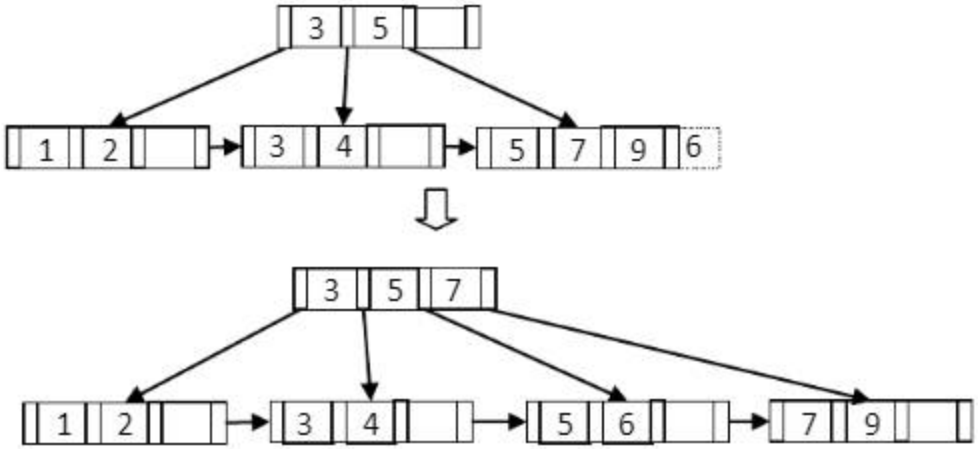
After inserting 2



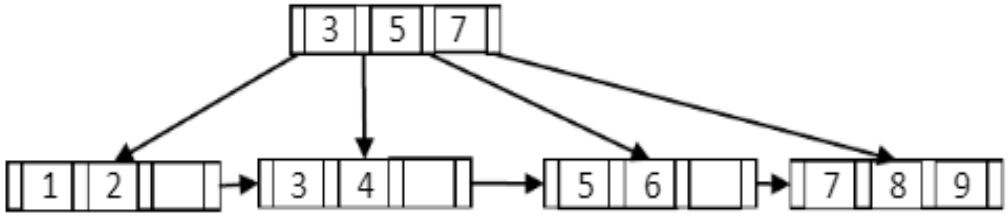
After inserting 4



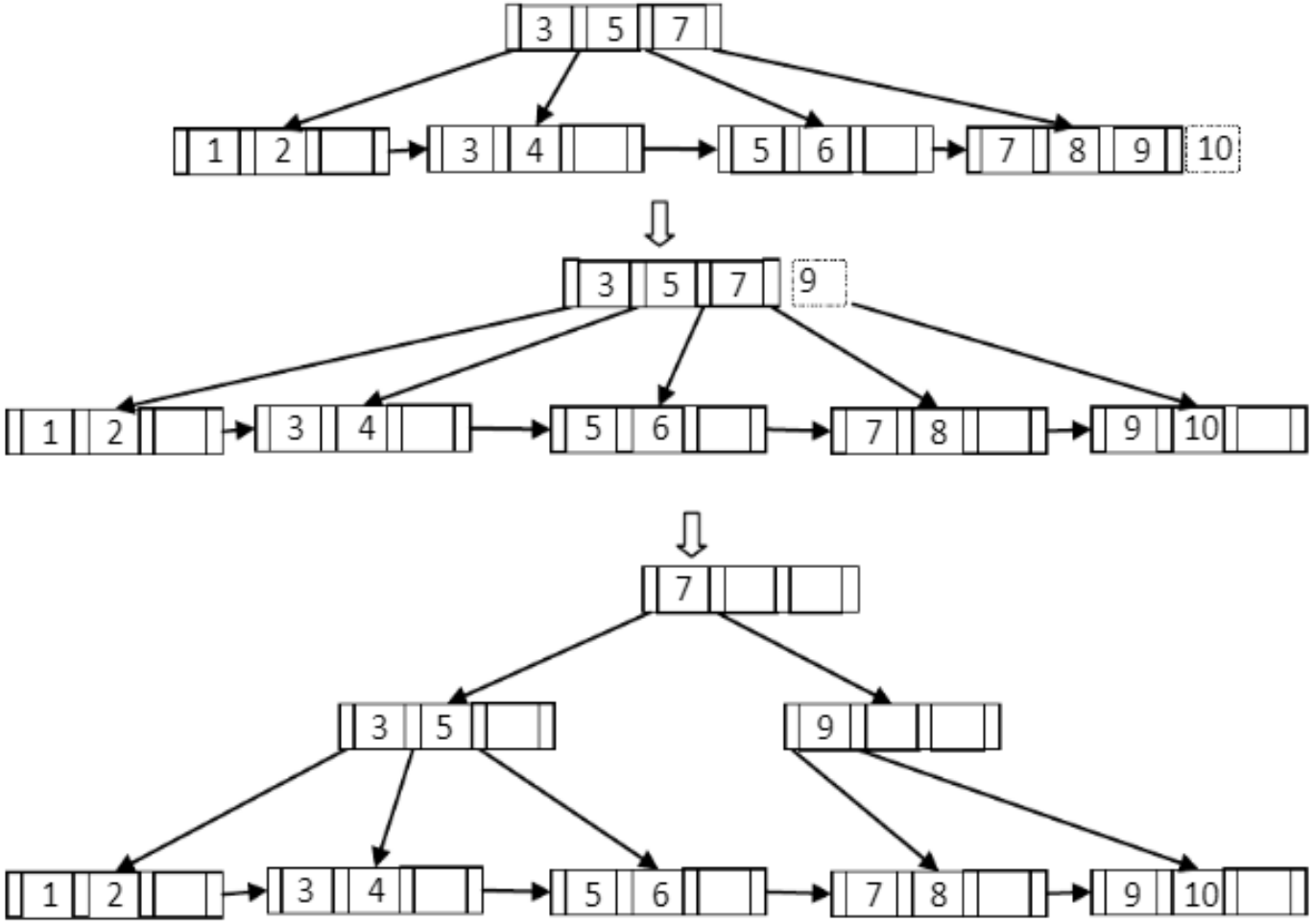
After inserting 6



After inserting 8



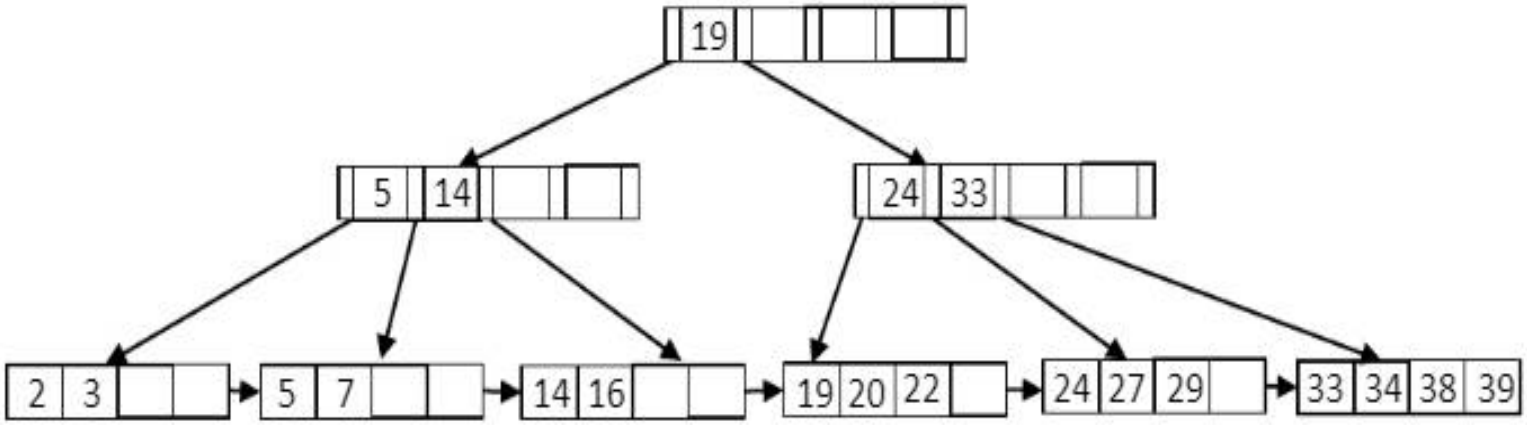
After inserting 10



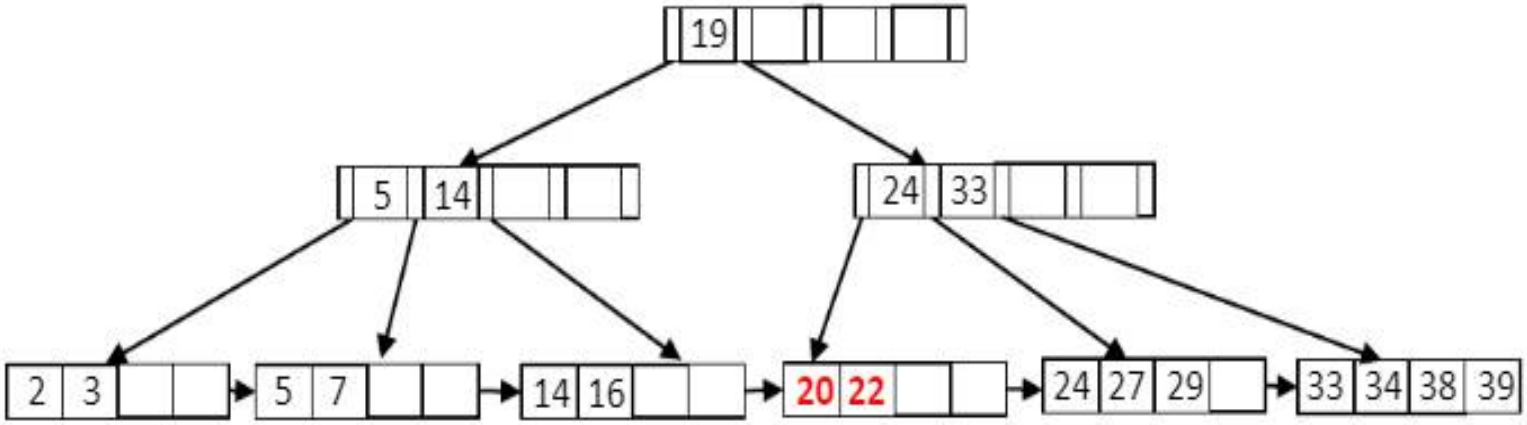
B+ Deletion

- 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).

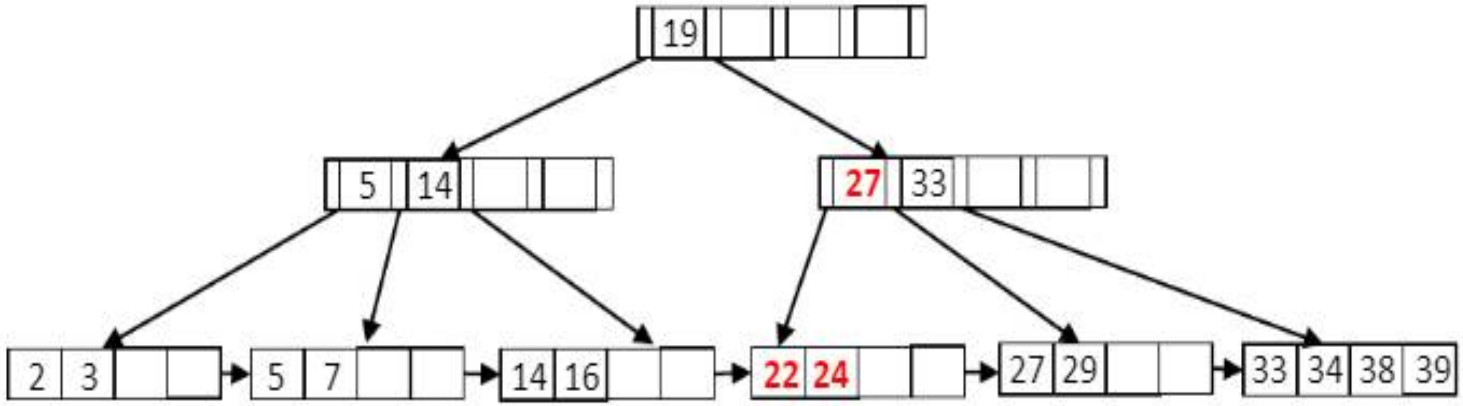
Example: Consider the given below tree and delete 19,



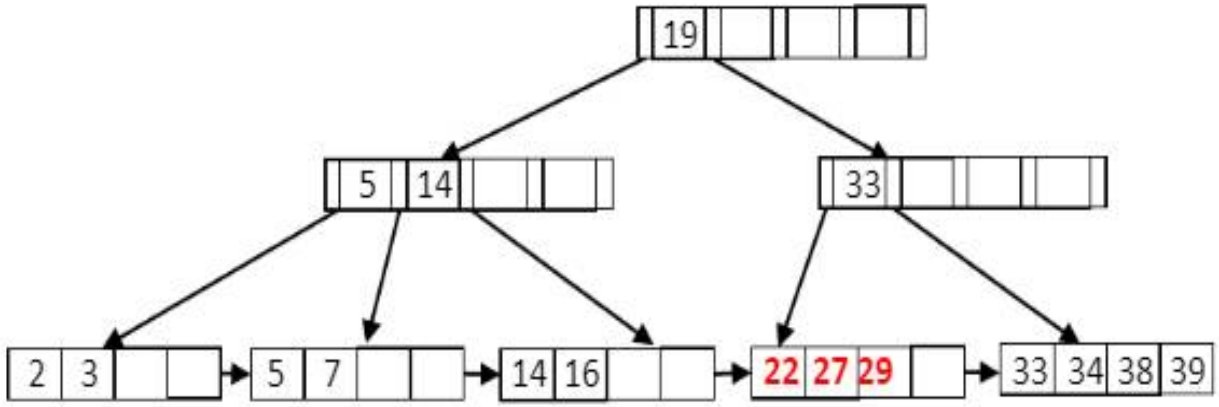
Delete 19 : Half full criteria is satisfied even after deleting 19, so just delete 19 from leaf node



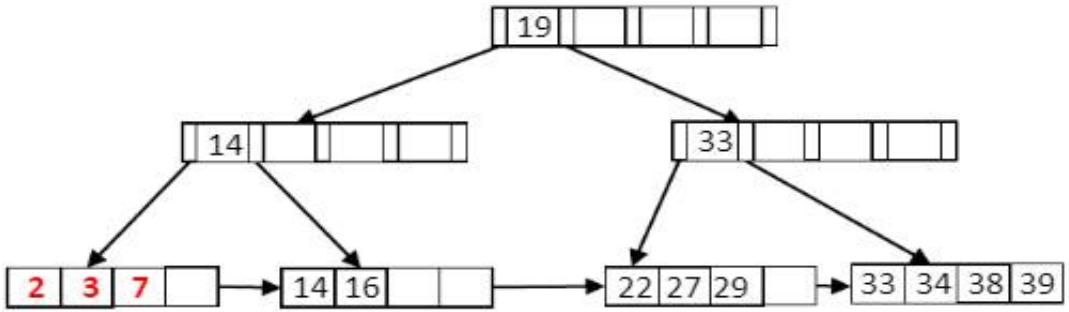
Delete 20: Half full criteria is not satisfied after deleting 20, so bring 24 from its right sibling and change key values in the internal nodes.



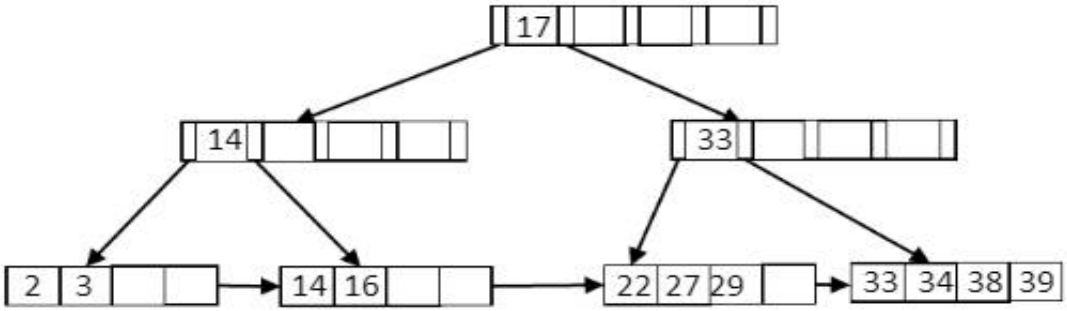
Delete 24: Half full criteria is not satisfied after deleting 24, bringing a value from its siblings also not possible. Therefore merge it with its right sibling and change key values in the internal nodes.



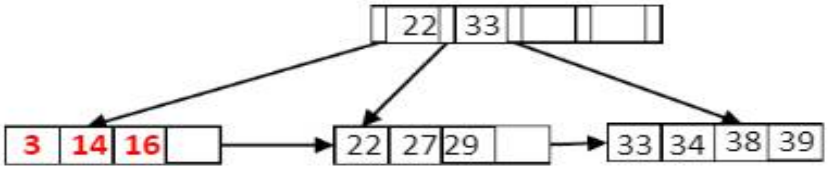
Delete 5: Half full criteria is not satisfied after deleting 5, bringing a value from its siblings also not possible. Therefore merge it with its left sibling (you can also merge with right) and change key values in the internal nodes.



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.



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.