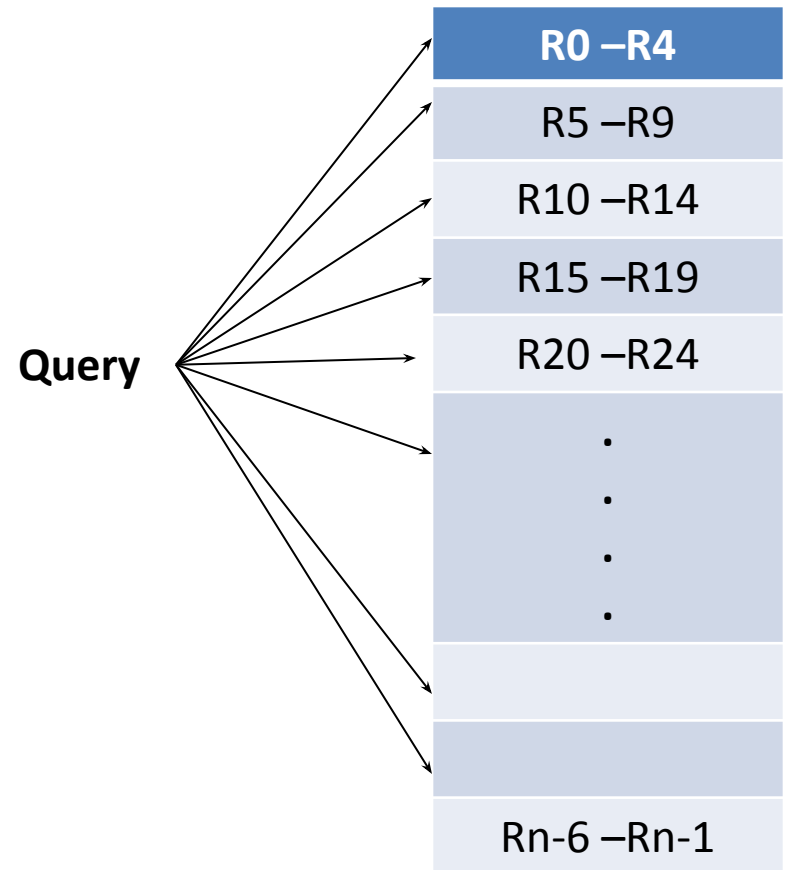
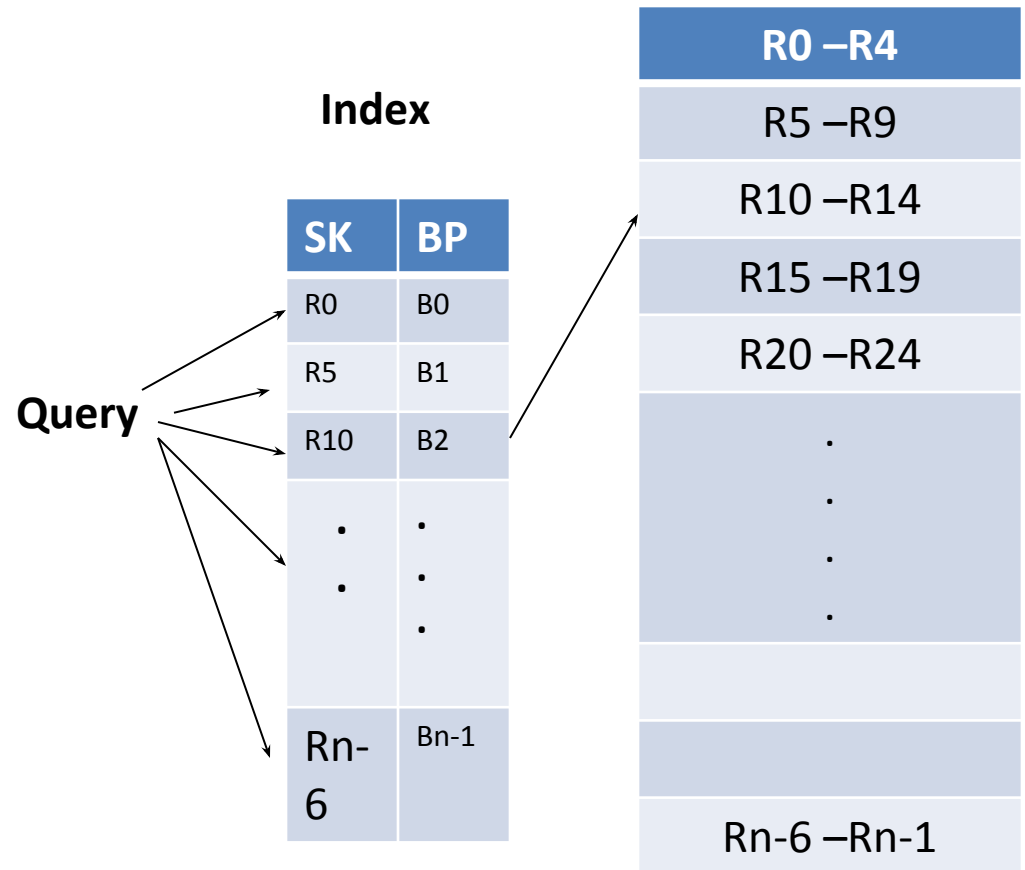


# Indexing & Hashing

# Indexing



Database file stored on disk



Database file stored on disk

# Indexing in DBMS

- Indexing is used to optimize the performance of a database by minimizing the number of disk accesses required when a query is processed.
- The index is a type of data structure. It is used to locate and access the data in a database table quickly.
- Index files are typically much smaller than the original file
- Two basic kinds of indices:
  - **Ordered indices:** search keys are stored in sorted order
  - **Hash indices:** search keys are distributed uniformly across “buckets” using a “hash function”.

# Index Evaluation Metrics

- Access types supported efficiently.
- Access time
- Insertion time
- Deletion time
- Space overhead

## Index structure:

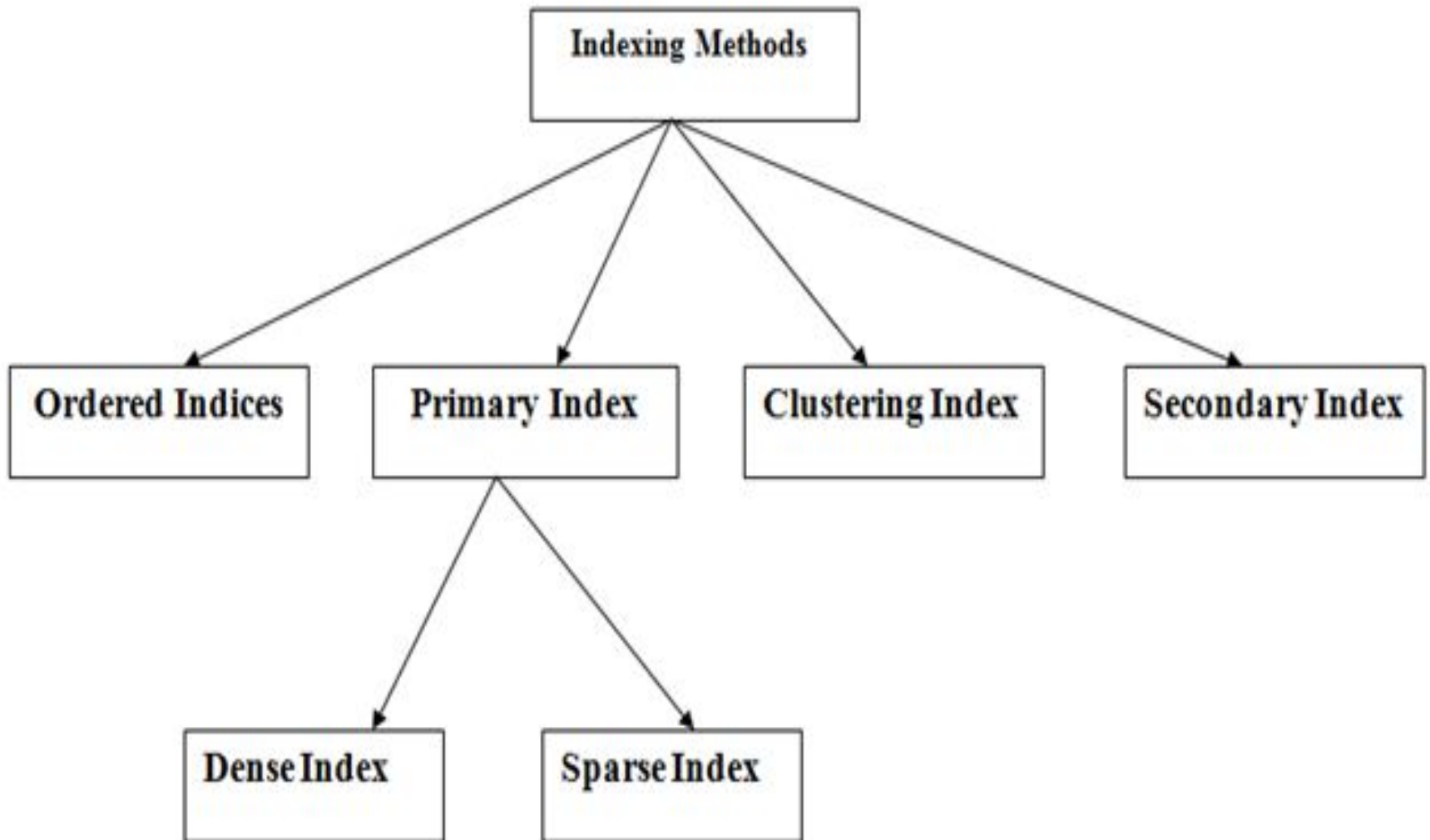
- Indexes can be created using some database columns.

Search key	Data Reference
------------	-------------------

The first column of the database is the search key. .

- Often search key is primary key or candidate key of the table but not always. The search key should be a field on which most query can come.
- The second column of the database is the data reference. It contains a set of pointers holding the address of the disk block where the value of the particular key can be found.

# Indexing methods



# Ordered indices

The indices are usually sorted to make searching faster. The indices which are sorted are known as ordered indices.

- Ordered indices are sorted on search key.
- On ordered index binary search can be apply otherwise linear search has to apply

# Primary Index

- If the index is created on the basis of the primary key of the table, then it is known as primary indexing.
  - Main file is sorted
  - Primary key is used as search key
- As primary keys are stored in sorted order, the performance of the searching operation is quite efficient.
- The primary index can be classified into two types: Dense index and Sparse index.

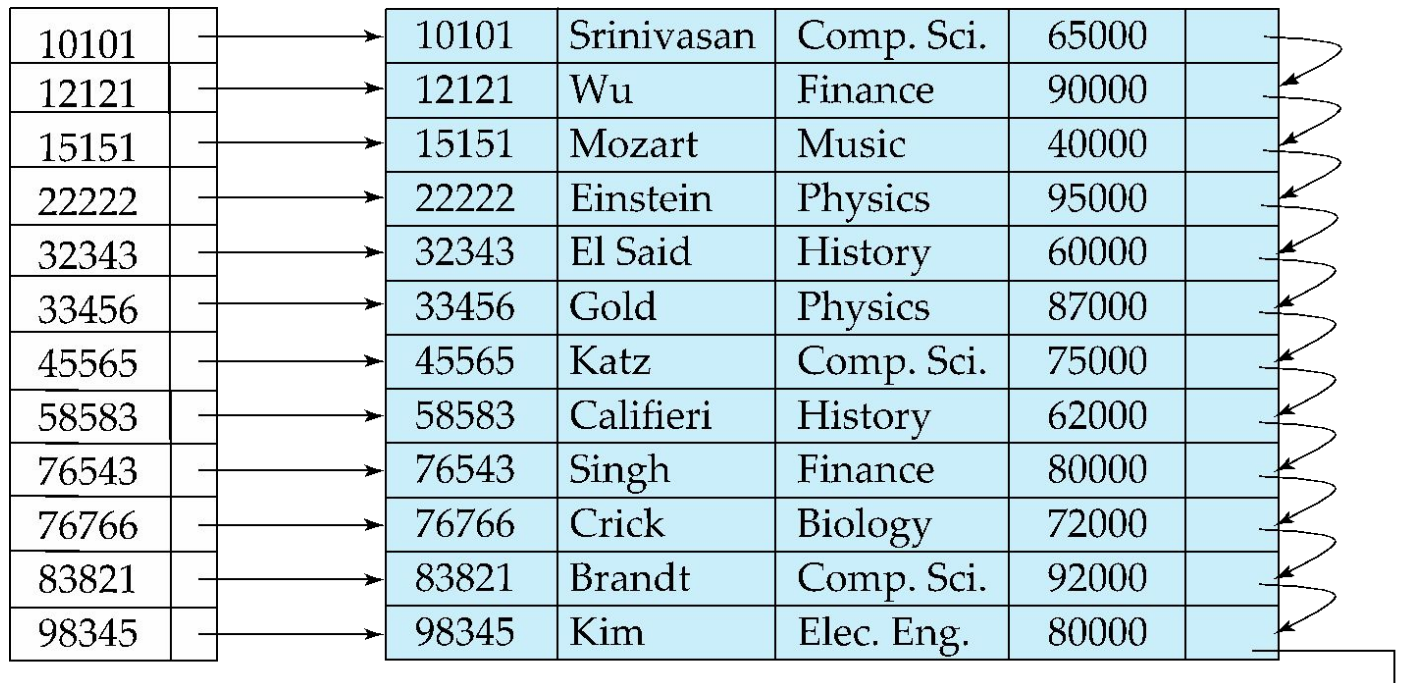


# Dense index

- The dense index contains an index record for every search key value in the data file. It makes searching faster.
- In this, the number of records in the index table is same as the number of records in the main file.
- It needs more space to store index record itself. The index records have the search key and a pointer to the actual record on the disk.

- index on *ID* attribute of *instructor* relation

10101		→	10101	Srinivasan	Comp. Sci.	65000	
12121		→	12121	Wu	Finance	90000	
15151		→	15151	Mozart	Music	40000	
22222		→	22222	Einstein	Physics	95000	
32343		→	32343	El Said	History	60000	
33456		→	33456	Gold	Physics	87000	
45565		→	45565	Katz	Comp. Sci.	75000	
58583		→	58583	Califieri	History	62000	
76543		→	76543	Singh	Finance	80000	
76766		→	76766	Crick	Biology	72000	
83821		→	83821	Brandt	Comp. Sci.	92000	
98345		→	98345	Kim	Elec. Eng.	80000	



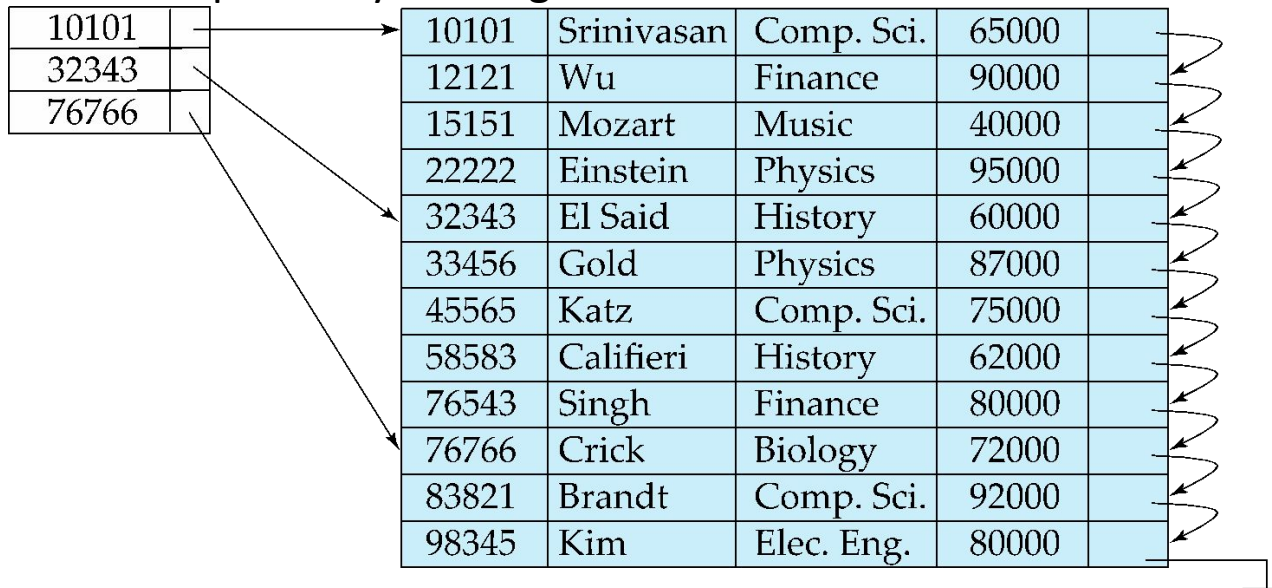
- Dense index on *dept\_name*, with *instructor* file sorted on *dept\_name*

Biology		76766	Crick	Biology	72000	
Comp. Sci.		10101	Srinivasan	Comp. Sci.	65000	
Elec. Eng.		45565	Katz	Comp. Sci.	75000	
Finance		83821	Brandt	Comp. Sci.	92000	
History		98345	Kim	Elec. Eng.	80000	
Music		12121	Wu	Finance	90000	
Physics		76543	Singh	Finance	80000	
		32343	El Said	History	60000	
		58583	Califieri	History	62000	
		15151	Mozart	Music	40000	
		22222	Einstein	Physics	95000	
		33465	Gold	Physics	87000	

The diagram illustrates a dense index on the *dept\_name* column. The index table (left) contains department names. The main table (right) contains instructor records sorted by department. Arrows indicate the mapping from the index to the main table rows.

# Sparse index

- In the data file, index record appears only for a few items. Each item points to a block
  - Applicable when records are sequentially ordered on search-key
  - In this, the number of records in the index table is equal to number of blocks acquired by the main file.
- To locate a record with search-key value  $K$  we:
  - Find index record with largest search-key value  $< K$
  - Search file sequentially starting at the record to which the index record points



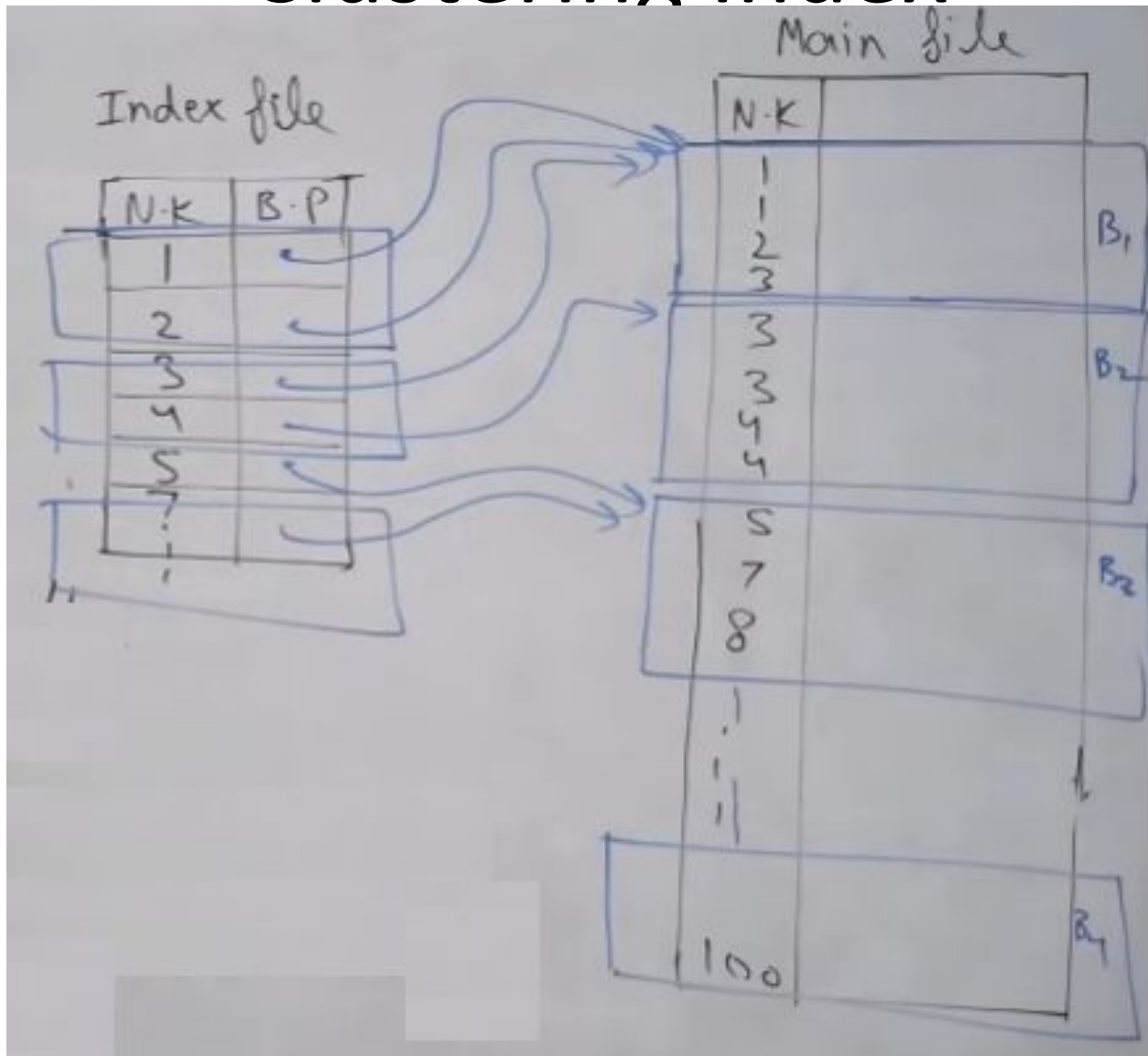
# Clustering Index

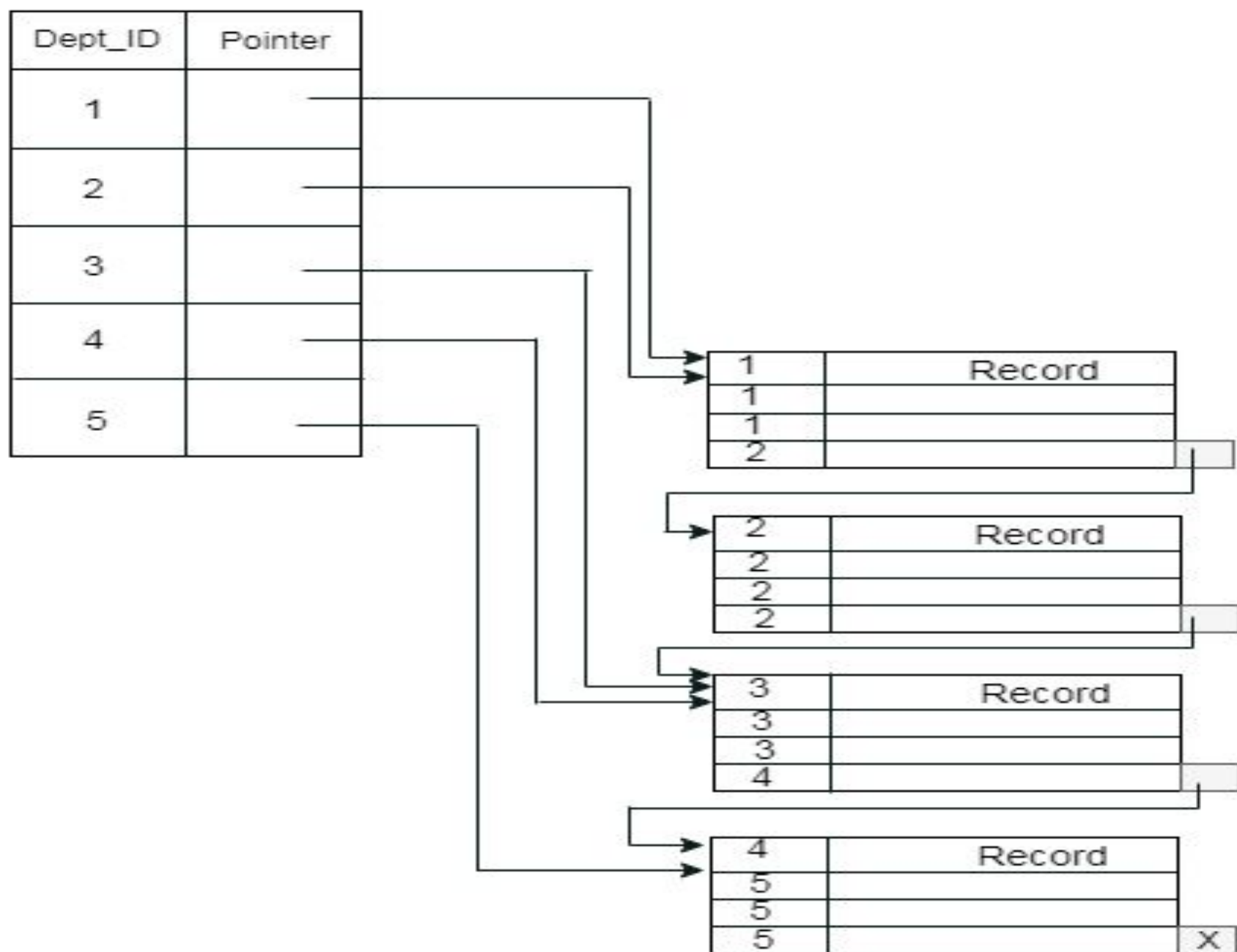
- A clustered index can be defined as an ordered data file. Sometimes the index is created on non-primary key columns which may not be unique for each record.
- In this, the number of records in the index table is equal to number of unique values in the field of main file to which we want to make search key

# Clustering Index

- Main file is sorted  
(on some non-key attribute)
- there will be one entry  
for each unique value of the  
non-key attribute
- if number of Block acquired by Index file  
is  $n$ , then Block access required will be  
 $\geq \log_2 n + 1$

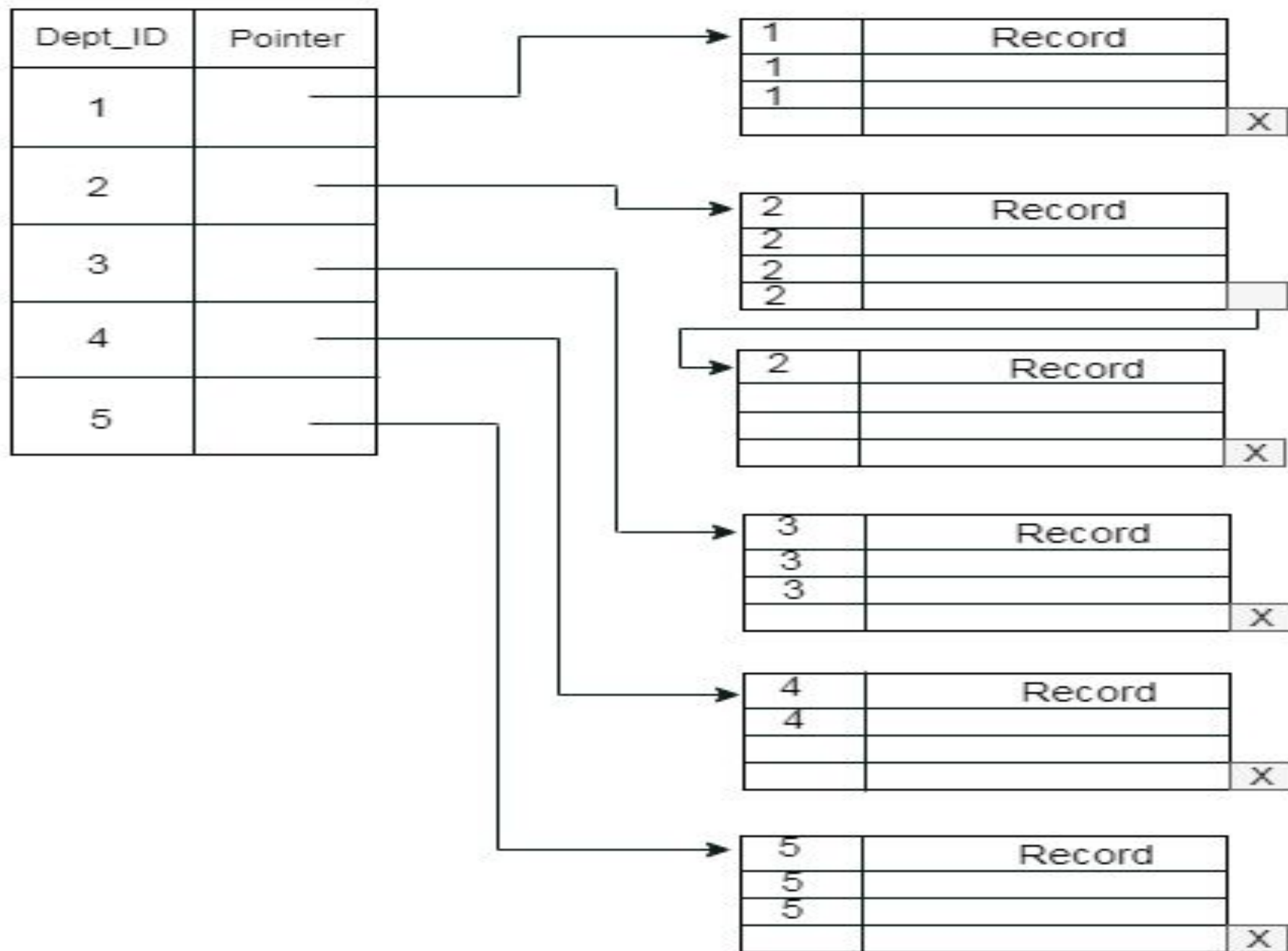
# Clustering Index







- The previous schema is little confusing because one disk block is shared by records which belong to the different cluster. If we use separate disk block for separate clusters, then it is called better technique.



# Secondary indexing

- Main file is unsorted
  - Sparse indexing not possible
- Can be on key or non-key attribute.
- Secondary indexing because primary indexing is already done.
- In this, the number of records in the index table is same as the number of records in the main file.
- No of block access  $\lceil \log_2 n \rceil + 1$

Index file

S.K	B.P
1	→
2	→
3	→
...	→
92	→
100	→

Main file

S.K		
1		
27		
3		
92		
52		
...		
8-10-50		

# Multilevel Index

- If index does not fit in memory, access becomes expensive.
- Solution: treat index kept on disk as a sequential file and construct a sparse index on it.
  - outer index – a sparse index of the basic index
  - inner index – the basic index file
- If even outer index is too large to fit in main memory, yet another level of index can be created, and so on.
- Indices at all levels must be updated on insertion or deletion from the file.

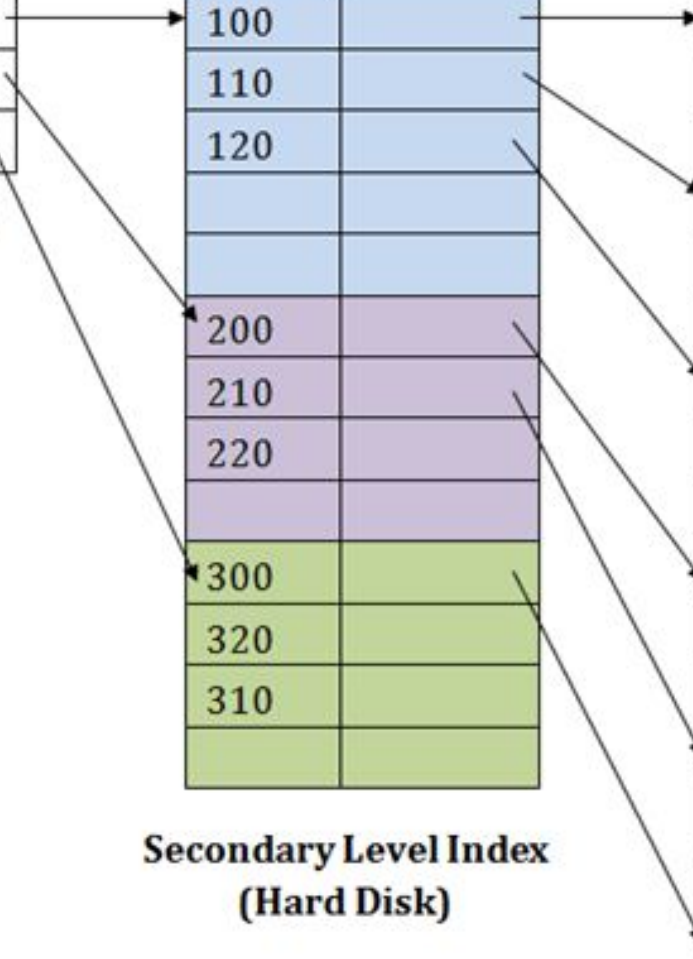
Roll	Pointer
100	
200	
300	

**Primary Level Index  
(RAM)**

Roll	Pointer
100	
110	
120	
200	
210	
220	
300	
320	
310	

**Secondary Level Index  
(Hard Disk)**

Data bock in Memory	
100	
101	
- - -	- - - - -
110	
111	
110	- - - - -
120	
121	
- - -	
200	
201	
- - -	- - - - -
210	
211	
- - -	- - - - -
300	
- - -	- - - - -



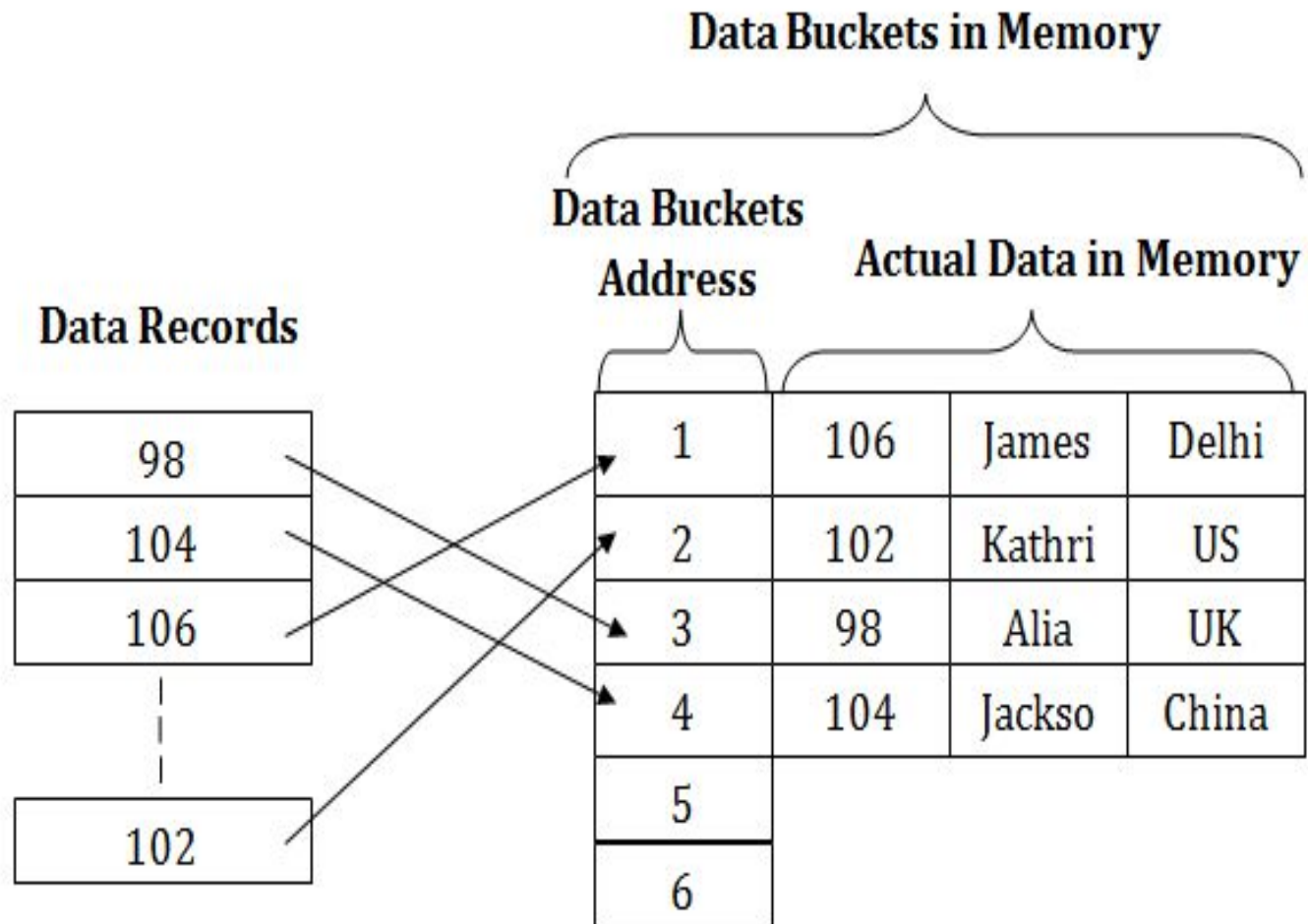
# Hashing

- Hashing technique is used to calculate the direct location of a data record on the disk without using index structure.
- In this technique, data is stored at the data blocks whose address is generated by using the hashing function.
- The memory location where these records are stored is known as data bucket or data blocks.

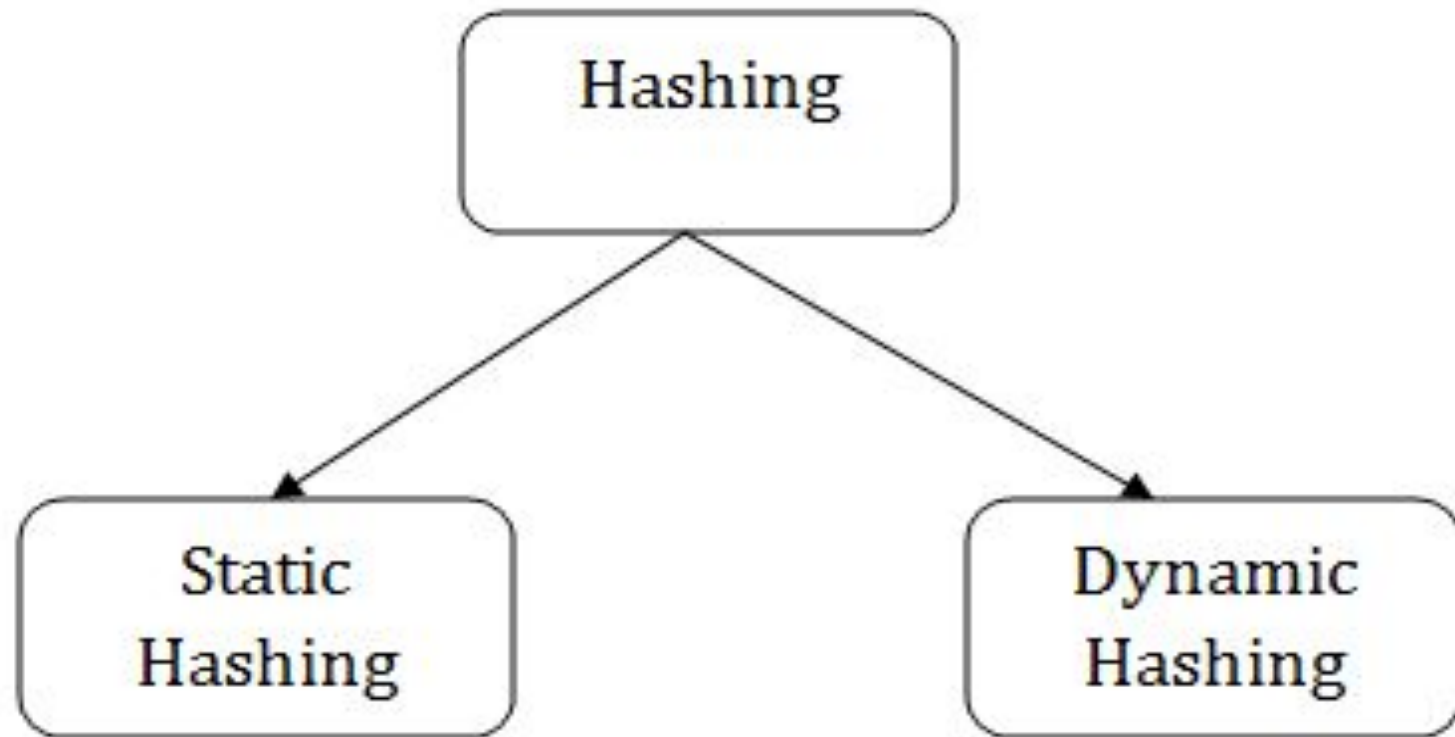
- In this, a hash function can choose any of the column value (table attribute) to generate the address.
- Most of the time, the hash function uses the primary key to generate the address of the data block.
- A hash function is a simple mathematical function to any complex mathematical function.
- Suppose we have mod (5) hash function to determine the address of the data block.



- In this case, it applies mod (5) hash function on the primary keys and generates 3, 1, 4 and 2 respectively, and records are stored in those data block addresses.



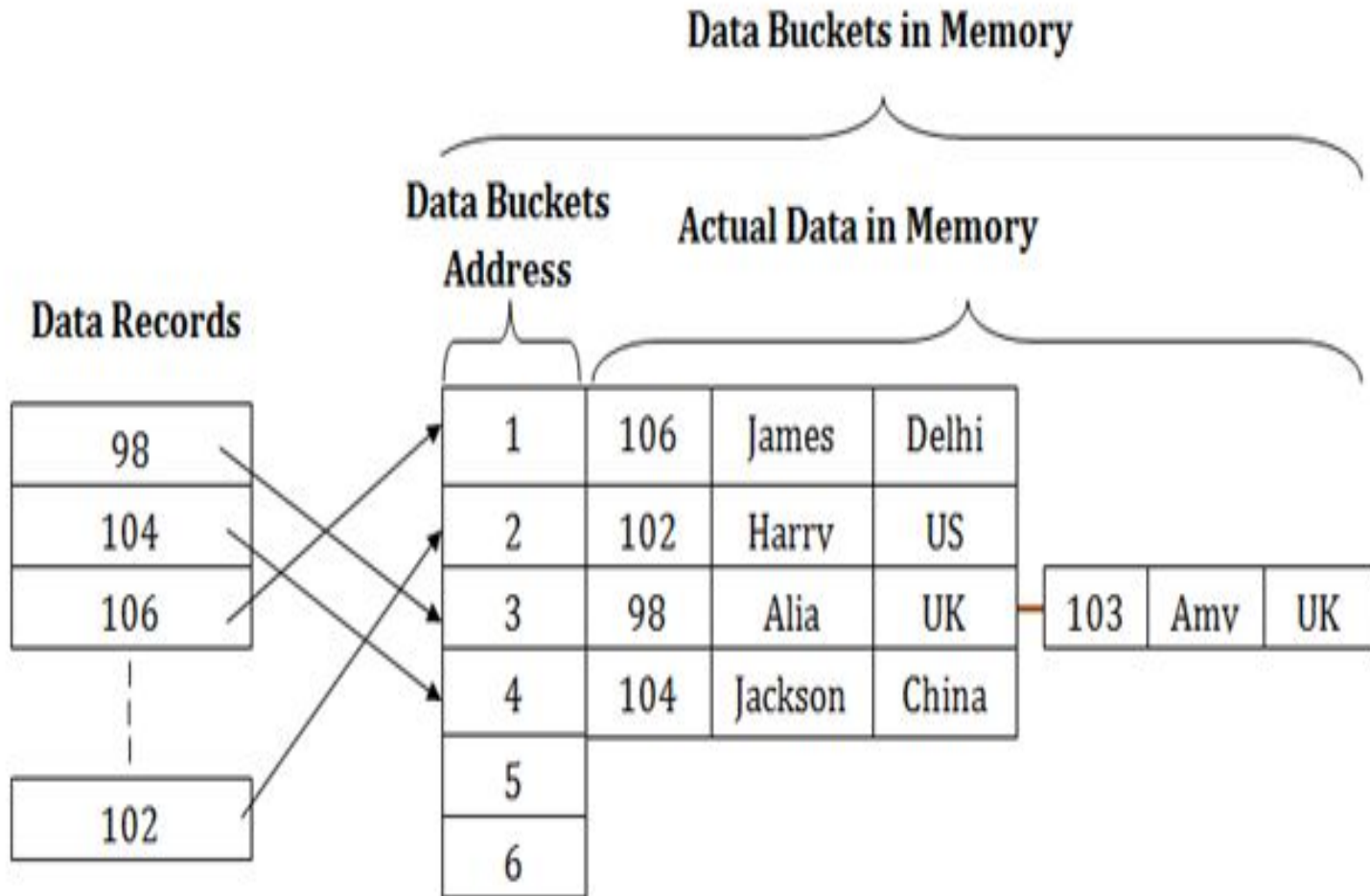
# Types of Hashing:



# Static Hashing

- In static hashing, the resultant data bucket address will always be the same.
- That means if we generate an address for EMP\_ID = 103 using the hash function  $\text{mod } 5$  then it will always result in same bucket address 3. Here, there will be no change in the bucket address.
- Hence in this static hashing, the number of data buckets in memory remains constant throughout. In this example, we will have five data buckets in the memory used to store the data.

# Bucket Overflow



# Operations of Static Hashing

## 1. Searching a record

When a record needs to be searched, then the same hash function retrieves the address of the bucket where the data is stored.

## 2. Insert a Record

When a new record is inserted into the table, then we will generate an address for a new record based on the hash key and record is stored in that location.

### 3. Delete a Record

- To delete a record, we will first fetch the record which is supposed to be deleted. Then we will delete the records for that address in memory.

### 4. Update a Record

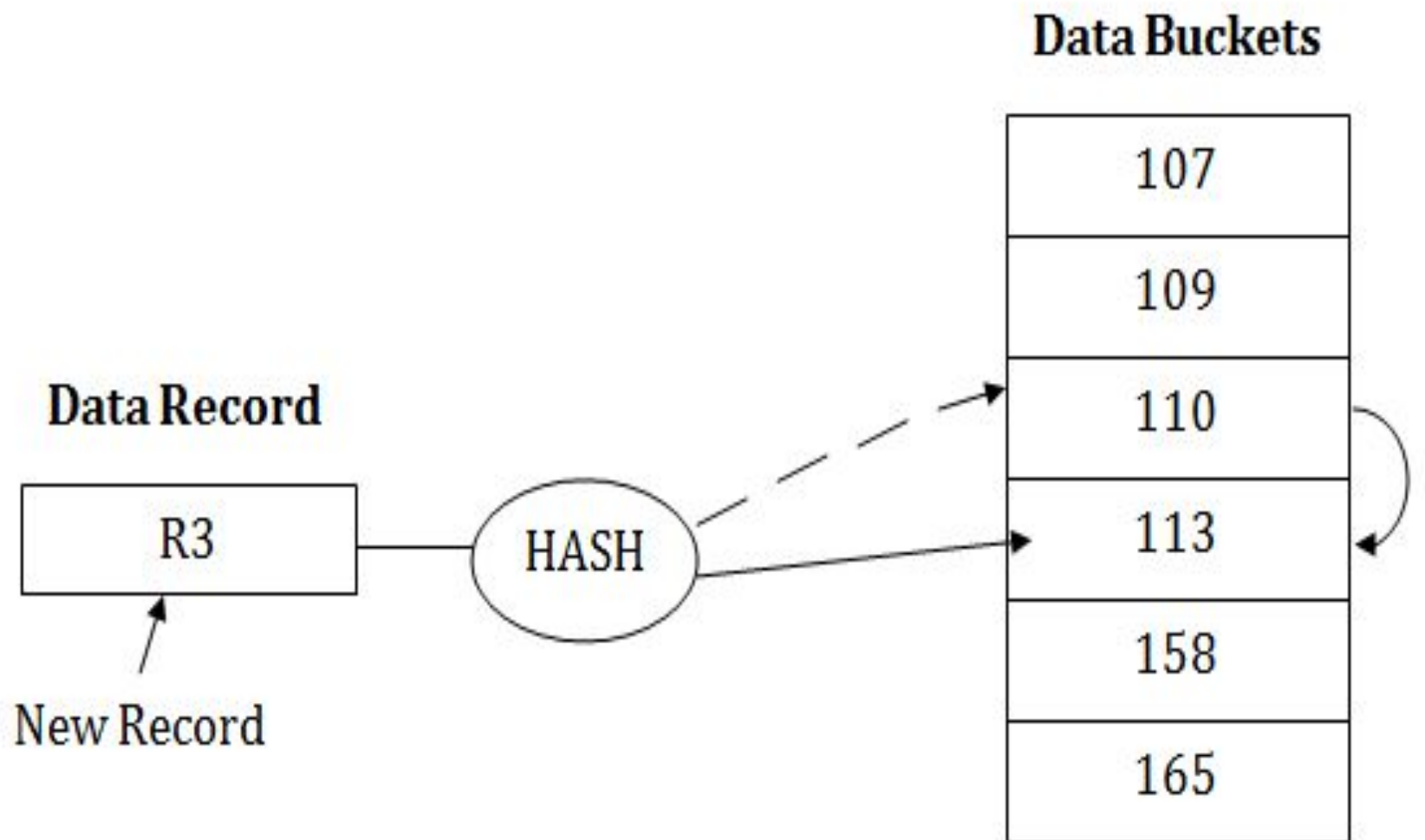
- To update a record, we will first search it using a hash function, and then the data record is updated.

- If we want to insert some new record into the file but the address of a data bucket generated by the hash function is not empty, or data already exists in that address.
- This situation in the static hashing is known as **bucket overflow**. This is a critical situation in this method.
- To overcome this situation, there are various methods. Some commonly used methods are as follows:

# 1. Open Hashing

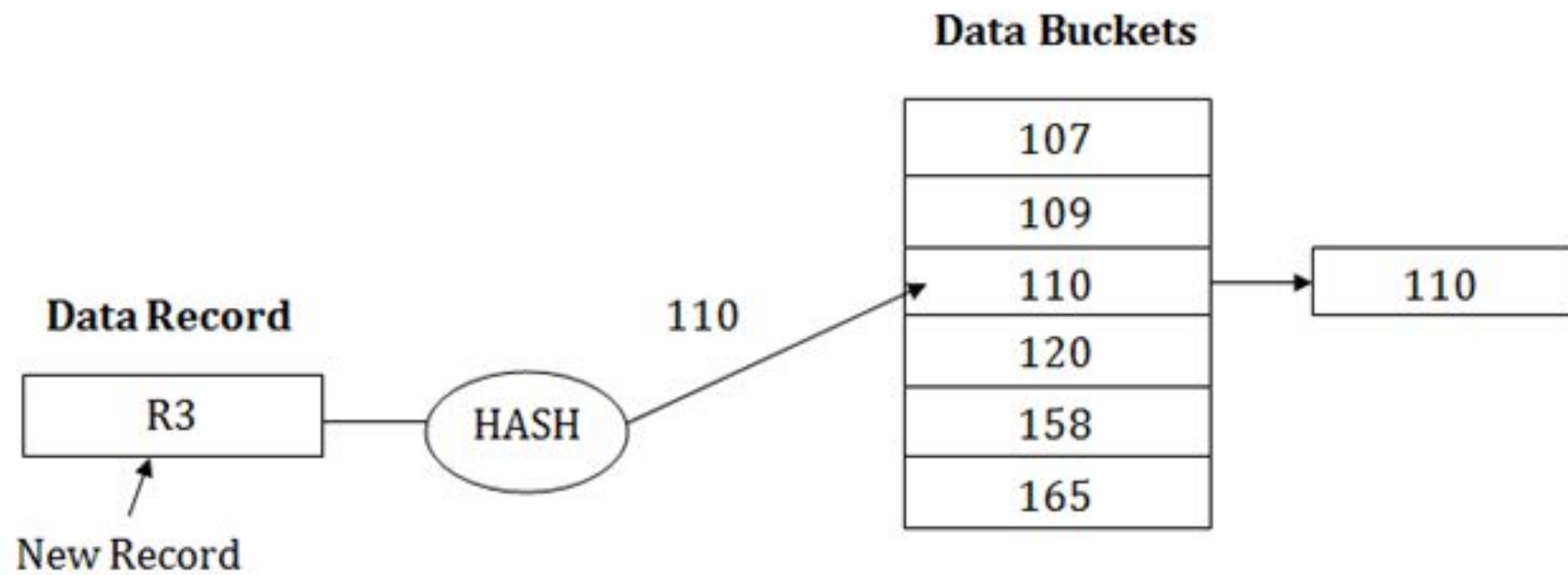
- When a hash function generates an address at which data is already stored, then the next bucket will be allocated to it. This mechanism is called as **Linear Probing**.
- **For example:** suppose R3 is a new address which needs to be inserted, the hash function generates address as 112 for R3. But the generated address is already full. So the system searches next available data bucket, 113 and assigns R3 to it.





## 2. Close Hashing

- When buckets are full, then a new data bucket is allocated for the same hash result and is linked after the previous one. This mechanism is known as **Overflow chaining**.
- **For example:** Suppose R3 is a new address which needs to be inserted into the table, the hash function generates address as 110 for it. But this bucket is full to store the new data. In this case, a new bucket is inserted at the end of 110 buckets and is linked to it.



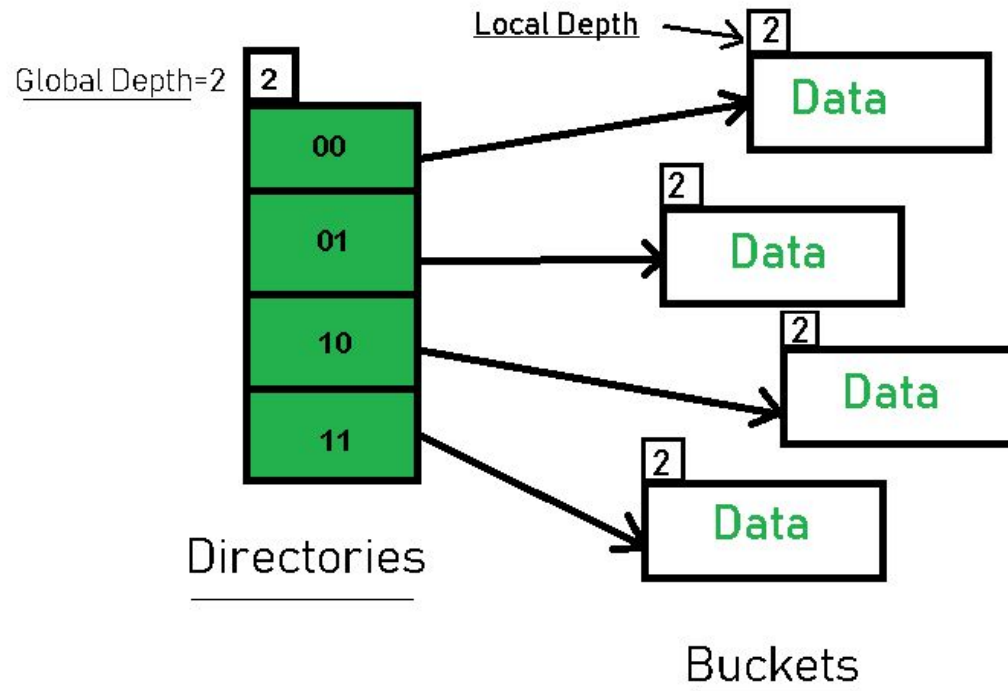
# Dynamic Hashing

- 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.

# Dynamic Hashing

- **Main features of Extendible Hashing:**
- **Directories:** The directories store addresses of the buckets in pointers. An id is assigned to each directory which may change each time when Directory Expansion takes place.
- **Buckets:** The buckets are used to hash the actual data.

# Dynamic Hashing



**Extensible Hashing**

# Dynamic Hashing

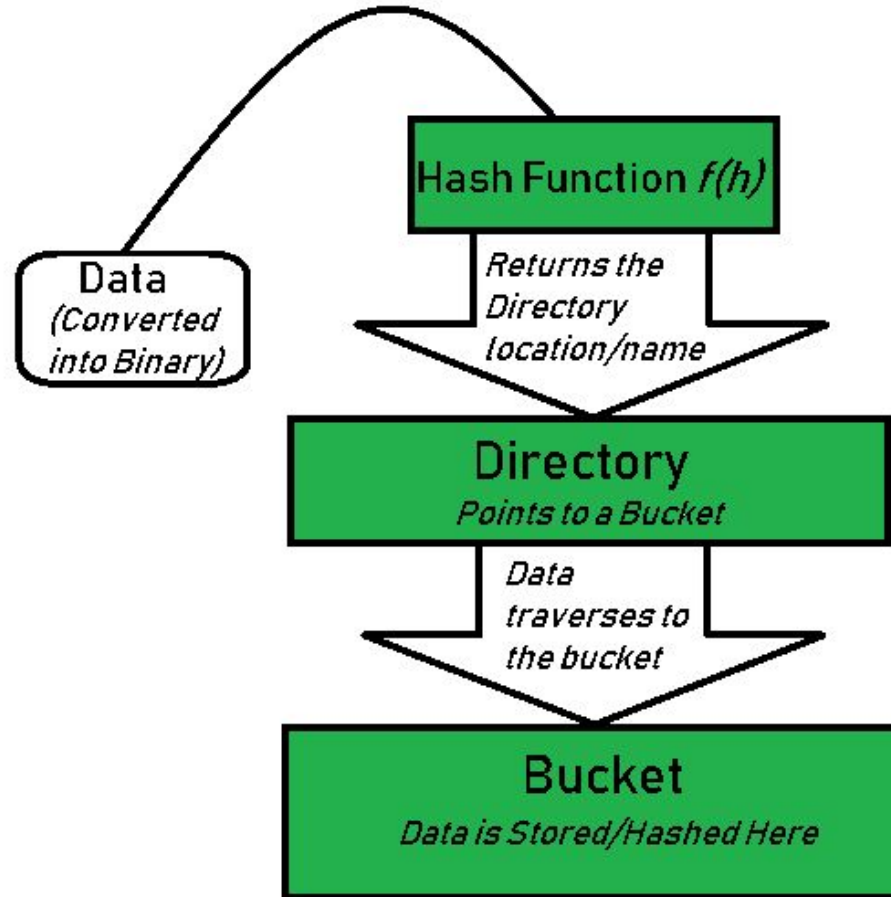
- **Global Depth:** It is associated with the Directories. They denote the number of bits which are used by the hash function to categorize the keys. Global Depth = Number of bits in directory id.
- **Local Depth:** It is the same as that of Global Depth except for the fact that Local Depth is associated with the buckets and not the directories. Local depth in accordance with the global depth is used to decide the action that to be performed in case an overflow occurs. Local Depth is always less than or equal to the Global Depth.

# Dynamic Hashing

- **Bucket Splitting:** When the number of elements in a bucket exceeds a particular size, then the bucket is split into two parts.
- **Directory Expansion:** Directory Expansion Takes place when a bucket overflows. Directory Expansion is performed when the local depth of the overflowing bucket is equal to the global depth.



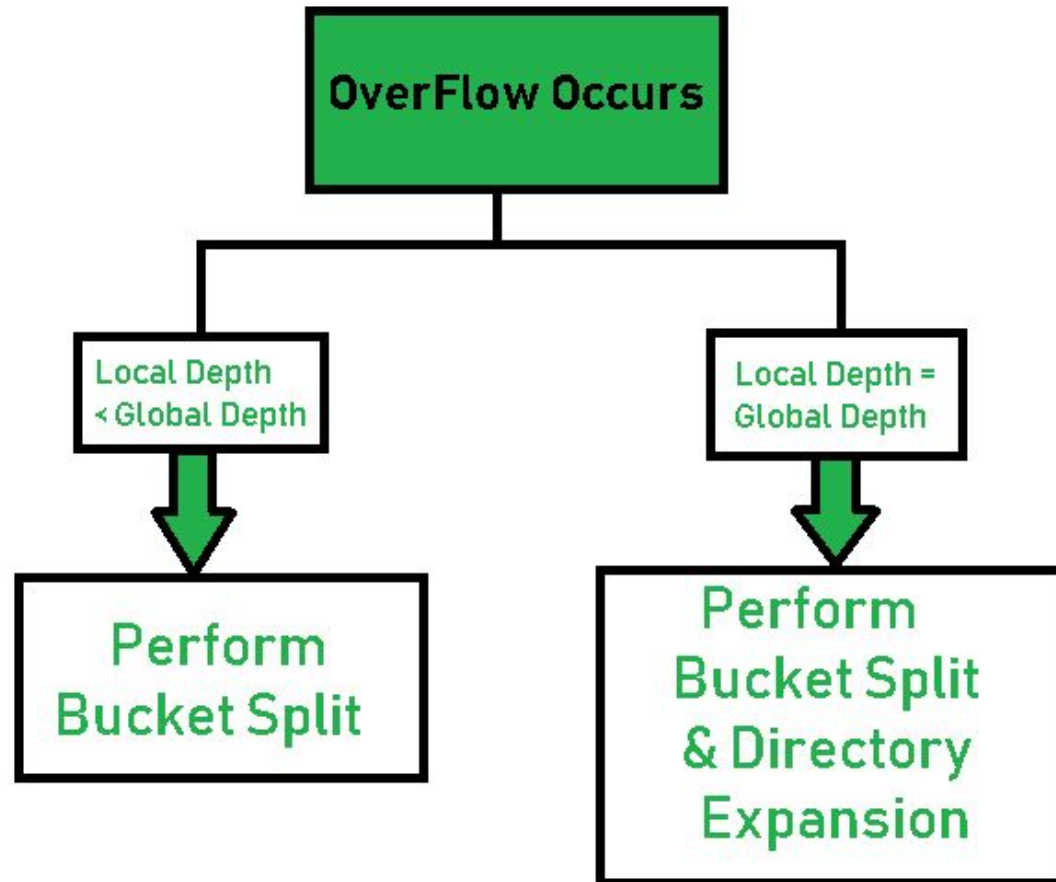
# Dynamic Hashing



# Dynamic Hashing

1. Convert into binary format.
2. Identify the Directory id using global depth of directory
  - I. If global depth is 3, directory id will be the represented by 3 LSBs of binary format
3. Now, navigate to the bucket pointed by the directory-id.
4. Check for overflow

# Dynamic Hashing



# Dynamic Hashing

5. **Rehashing of Split Bucket Elements**

6. The element is successfully hashed.

let us consider a example of dynamic hashing:

**16,4,6,22,24,10,31,7,9,20,26.**

- **Bucket Size: 3** (Assume)
- Initially, the global-depth and local-depth is always 1.

# Solution

- First, calculate the binary forms of each of the given numbers.

16- 10000

4- 00100

6- 00110

22- 10110

24- 11000

10- 01010

31- 11111

7- 00111

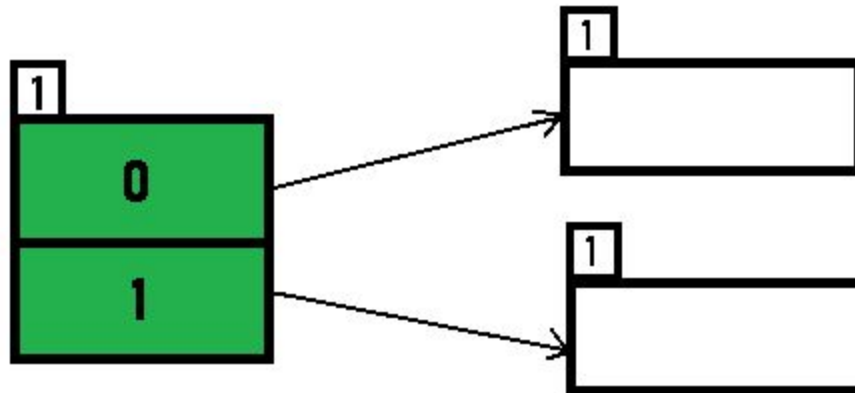
9- 01001

20- 10100

26- 01101

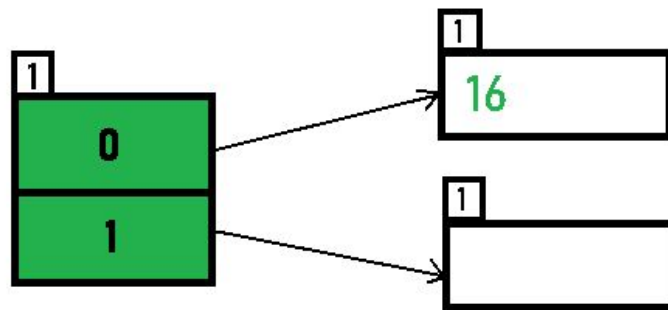
# Solution

- Initially, the hashing frame looks like this:



# Solution

- **Inserting 16:** The binary format of 16 is 10000 and global-depth is 1. The hash function returns 1 LSB of 10000 which is 0. Hence, 16 is mapped to the directory with id=0.



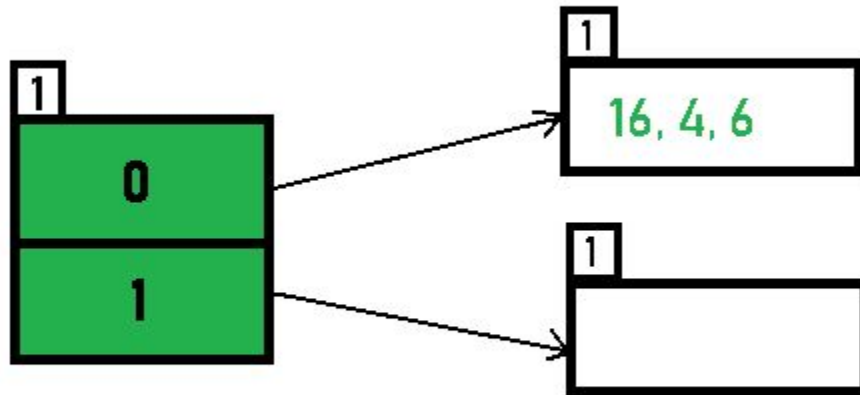
*Hash(16) = 10000*

# Solution

- **Inserting 4:** The binary format of 4 is 00100 and global-depth is 1. The hash function returns 1 LSB of 00100 which is 0. Hence, 4 is mapped to the directory with id=0. No overflow as bucket size is 3.
- **Inserting 6:** (00110) mapped to the directory with id=0



# Solution



$Hash(4) = 100$

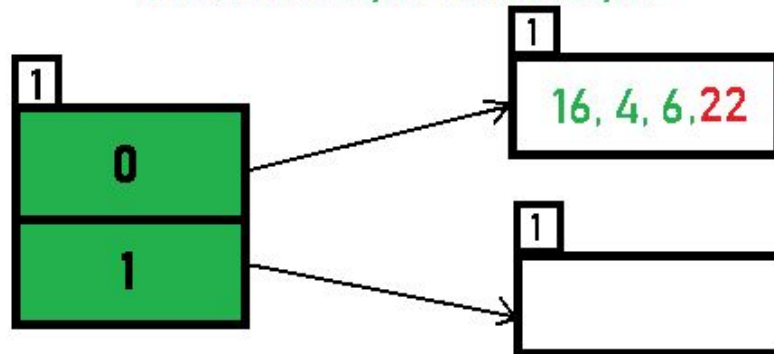
$Hash(6) = 110$

# Solution

- **Inserting 22:** The binary form of 22 is 10110. Its LSB is 0. The bucket pointed by directory 0 is already full. Hence, Over Flow occurs.

## OverFlow Condition

*Here, Local Depth=Global Depth*



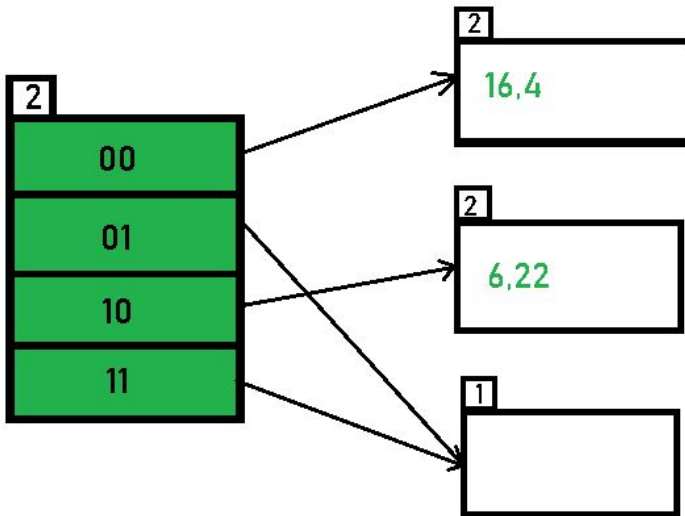
*Hash(22)=10110*

# Solution

- Since Local Depth = Global Depth, the bucket splits and directory expansion takes place.
- Also, rehashing of numbers present in the overflowing bucket takes place after the split.
- And, since the global depth is incremented by 1, now, the global depth is 2. Hence, 16, 4, 6, 22 are now rehashed w.r.t 2 LSBs.  
**16(10000), 4(100), 6(110), 22(10110) ]**

# Solution

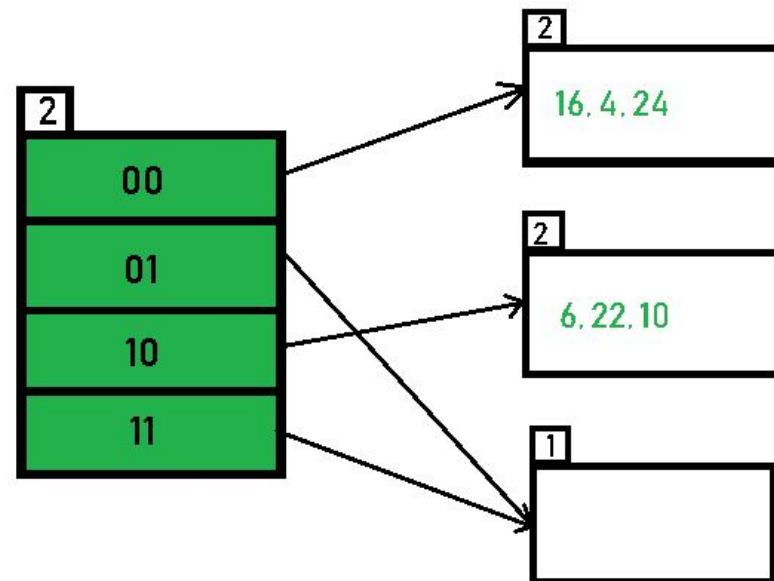
*After Bucket Split and Directory Expansion*



- › *\*Notice that the bucket which was underflow has remained untouched.*
- › *But, since the number of directories has doubled, we now have 2 directories 01 and 11 pointing to the same bucket.*
- › *This is because the local-depth of the bucket has remained 1.*
- › *And, any bucket having a local depth less than the global depth is pointed-to by more than one directories.*

# Solution

- **Inserting 24 and 10:** 24(11000) and 10 (1010) can be hashed based on directories with id 00 and 10. Here, we encounter no overflow condition.

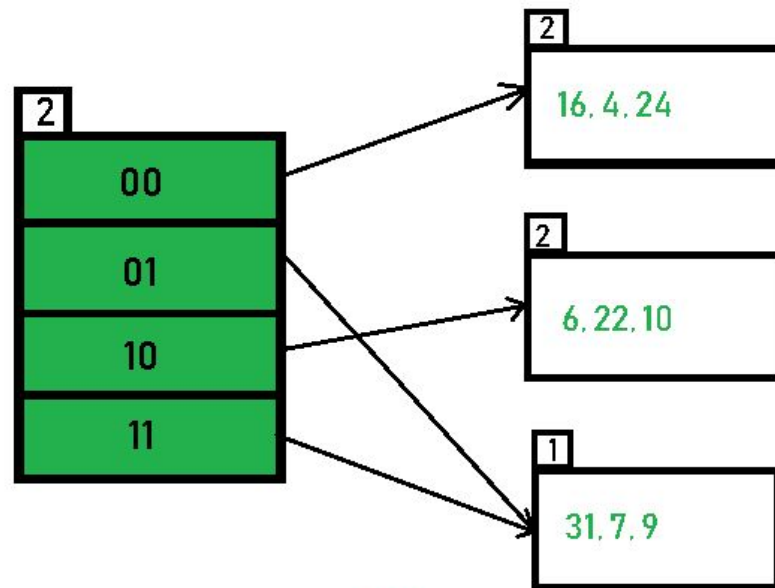


*Hash(24) = 11000*

*Hash(10) = 1010*

# Solution

- **Inserting 31,7,9:** All of these elements[ 31(11111), 7(00111), 9(01001) ] have either 01 or 11 in their LSBs. Hence, they are mapped on the bucket pointed out by 01 and 11. We do not encounter any overflow condition here.



*Hash(31) = 11111*

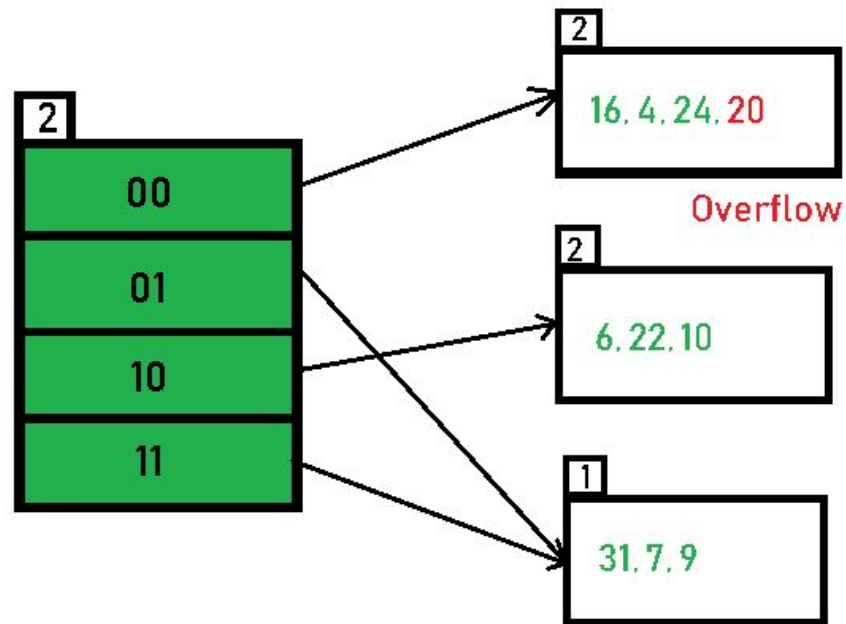
*Hash(7) = 111*

*Hash(9) = 1001*

# Solution

- **Inserting 20:** Insertion of data element 20 (10100) will again cause the overflow problem.

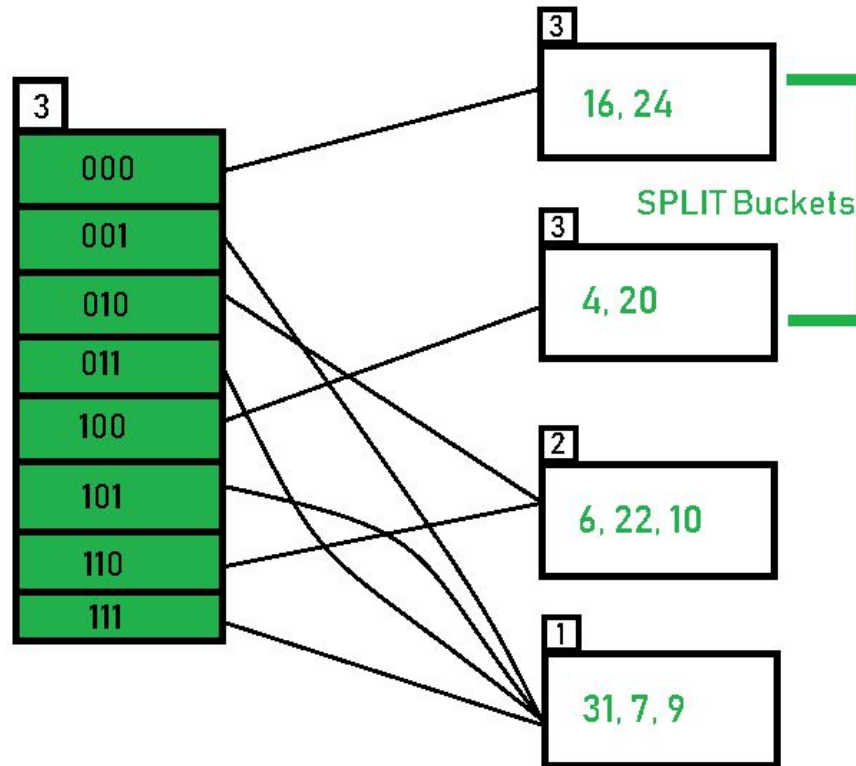
*Overflow, Local Depth= Global Depth*



*Hash(20)=10100*

# Solution

- **local depth of the bucket = global-depth**, directory expansion (doubling) takes place along with bucket splitting. Elements present in overflowing bucket are rehashed with the new global depth. Now, the new Hash table looks like this:



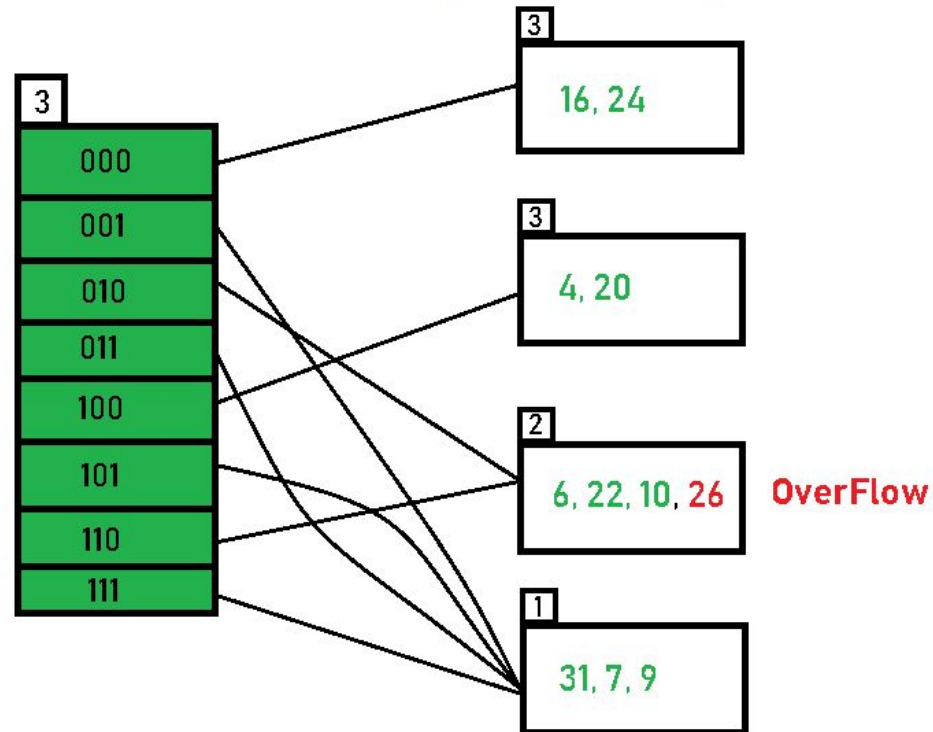


# Solution

- **Inserting 26:** Global depth is 3. Hence, 3 LSBs of 26(11010) are considered. Therefore 26 best fits in the bucket pointed out by directory 010

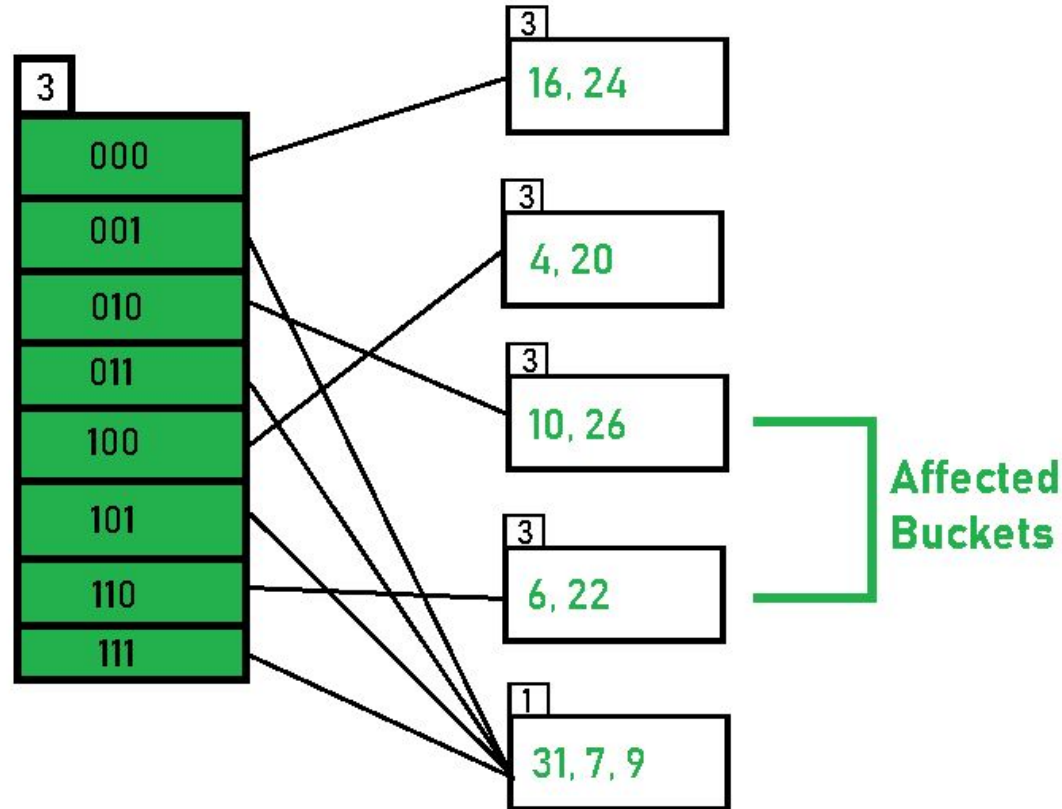
*Hash(26)=11010*

*OverFlow, Local Depth < Global Depth*



# Solution

- **local depth of bucket < Global depth ( $2 < 3$ )**, directories are not doubled but, only the bucket is split and elements are rehashed. Finally, the output of hashing the given list of numbers is obtained.



# Dynamic Hashing: 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 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.

# Limitations of Dynamic Hashing

- The directory size may increase significantly if several records are hashed on the same directory while keeping the record distribution non-uniform.
- Memory is wasted in pointers when the global depth and local depth difference becomes drastic.

- RAID
  - Self study from books and notes
- B and B++ Trees
  - (<https://www.youtube.com/watch?v=aZjYr87r1b8>)
  - Insertion
  - Deletion

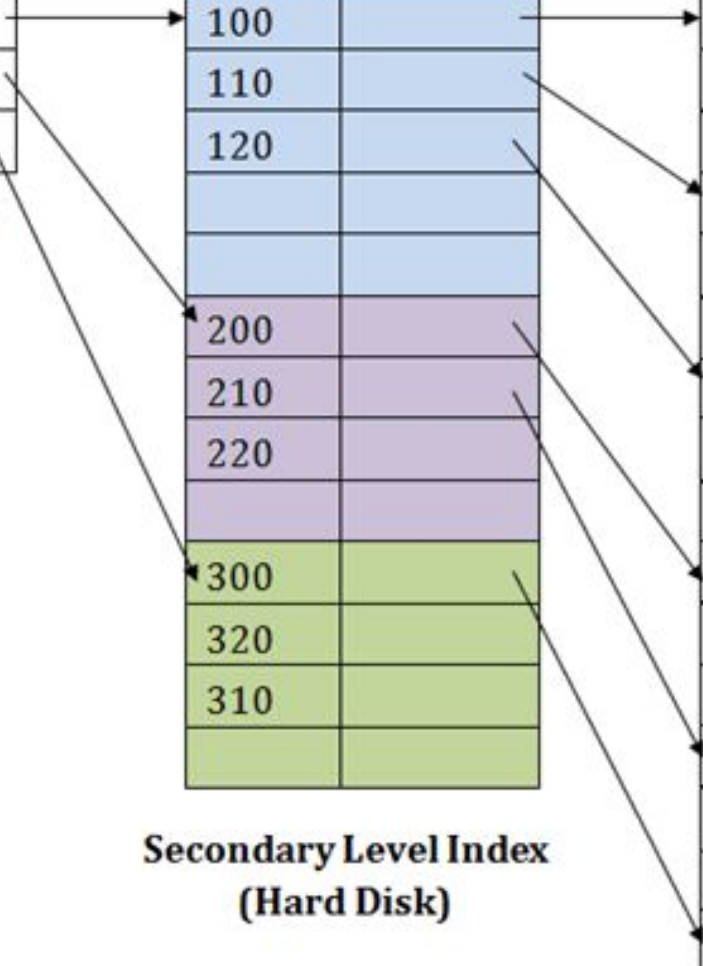
Roll	Pointer
100	
200	
300	

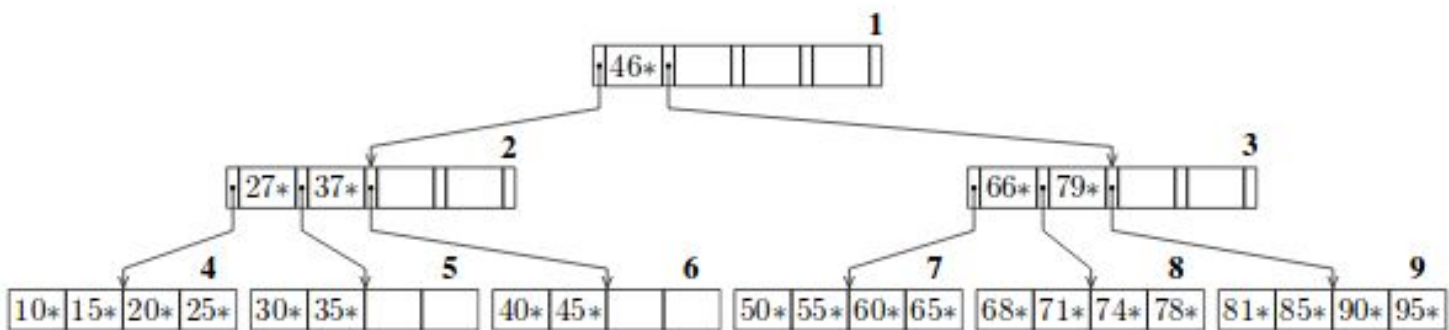
**Primary Level Index  
(RAM)**

Roll	Pointer
100	
110	
120	
200	
210	
220	
300	
320	
310	

**Secondary Level Index  
(Hard Disk)**

Data bock in Memory	
100	
101	
- - -	- - - - -
110	
111	
110	- - - - -
120	
121	
- - -	
200	
201	
- - -	- - - - -
210	
211	
- - -	- - - - -
300	
- - -	- - - - -





**Figure 5.8:** Example of a  $B$ -tree with  $m = 5$