Physical Database Design

Dr. Jyotismita Chaki

Indexing

- Indexes are used to speed up the retrieval of records in response to certain search conditions.
- The index structures are additional files on disk that provide secondary access paths, which provide alternative ways to access the records without affecting the physical placement of records in the primary data file on disk.
- They enable efficient access to records based on the indexing fields that are used to construct the index.
- Any field of the file can be used to create an index.
- The most prevalent types of indexes are based on
  - ordered files (single-level indexes) and
  - use tree data structures (multilevel indexes, B+-trees) to organize the index.

Ordered Indices

- The idea behind an ordered index is similar to that behind the index used in a textbook, which lists important terms at the end of the book in alphabetical order along with a list of page numbers where the term appears in the book.
- We can search the book index for a certain term in the textbook to find a list of addresses—page numbers in this case—and use these addresses to locate the specified pages first and then search for the term on each specified page.
- The alternative, if no other guidance is given, would be to sift slowly through the whole textbook word by word to find the term we are interested in; this corresponds to doing a *linear* search, which scans the whole file.
- Of course, most books do have additional information, such as chapter and section titles, which help us find a term without having to search through the whole book.
- However, the index is the only exact indication of the pages where each term occurs in the book.

Single Level Indexing

- For a file with a given record structure consisting of several fields (or attributes), an index access structure is usually defined on a **single field of a file**, called an **indexing field** (or **indexing attribute**).
- The index typically stores each value of the index field along with a list of pointers to all disk blocks that contain records with that field value.
- The values in the index are ordered so that we can do a binary search on the index.
- Types:
  - Primary index
  - Clustering index
  - Secondary index

Single Level Indexing: Primary index

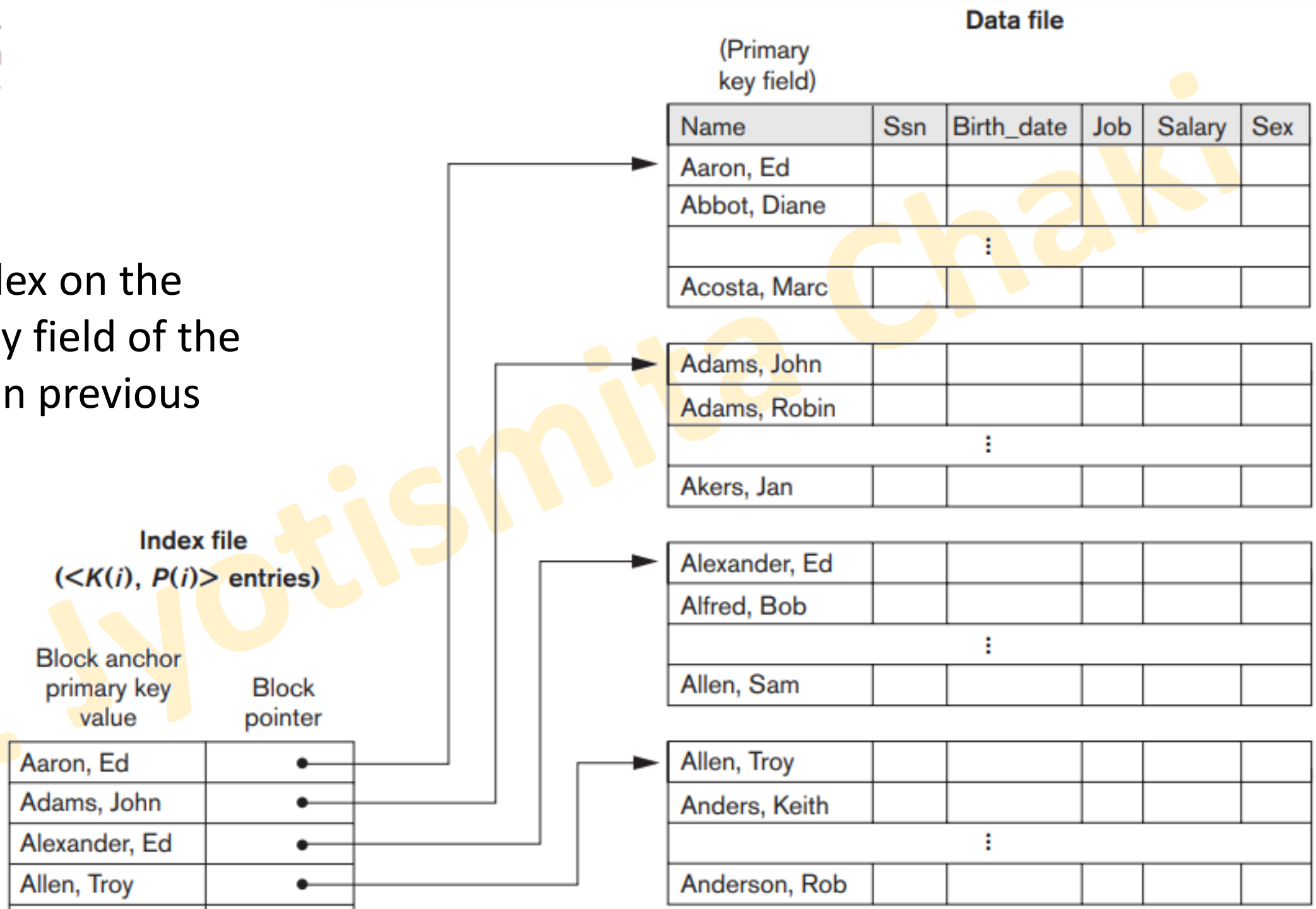
- A primary index is an ordered file whose records are of fixed length with two fields.
- It acts like an access structure to efficiently search for and access the data records in a data file.
- The first field is of the same data type as the ordering key field—called the primary key—of the data file, and the second field is a pointer to a disk block (a block address).
- There is one index entry (or index record) in the index file for each block in the data file.
- The two field values of index entry can be represented as <K(i), P(i)>.

Single Level Indexing: Primary index

Some blocks of an ordered (sequential) file of EMPLOYEE records with Name as the ordering key field.

	Name	Ssn	Birth_date	Job	Salary	Sex
Block 1	Aaron, Ed					
	Abbott, Diane					
	⋮					
	Acosta, Marc					
Block 2	Adams, John					
	Adams, Robin					
	⋮					
	Akers, Jan					
Block 3	Alexander, Ed					
	Alfred, Bob					
	⋮					
	Allen, Sam					

Primary index on the ordering key field of the file shown in previous slide



Single Level Indexing: Primary index

- To create a primary index on the ordered file shown, we use the Name field as primary key, because that is the ordering key field of the file (assuming that each value of Name is unique).
- Each entry in the index has a Name value and a pointer.
- The first three index entries are as follows:
  - <K(1) = (Aaron, Ed), P(1) = address of block 1>
  - <K(2) = (Adams, John), P(2) = address of block 2>
  - <K(3) = (Alexander, Ed), P(3) = address of block 3>
- The total number of entries in the index is the same as the number of disk blocks in the ordered data file.
- The first record in each block of the data file is called the anchor record of the block, or simply the block anchor.

Single Level Indexing: Primary index: Example

- Suppose that we have an ordered file with r = 300,000 records stored on a disk with block size B = 4,096 bytes.
- File records are of fixed size, with record length R = 100 bytes.
- The blocking factor for the file would be bfr = (B/R) = (4,096/100) = 40 records per block.
- The number of blocks needed for the file is b = (r/bfr) = (300,000/40) = 7,500 blocks.
- A binary search on the data file would need approximately $\log_2 b = (\log_2 7{,}500) = 13$ block accesses.

Single Level Indexing: Primary index: Example

- Now suppose that the ordering key field of the file is K = 9 bytes long, a block pointer is P = 6 bytes long, and we have constructed a primary index for the file.
- The size of each index entry is R_i = (9 + 6) = 15 bytes, so the blocking factor for the index is $bfr_i = (B/R_i) = (4{,}096/15) = 273$ entries per block.
- The total number of index entries r_i is equal to the number of blocks in the data file, which is 7,500. The number of index blocks is hence $b_i = (r_i/bfr_i) = (7{,}500/273) = 28$ blocks.
- To perform a binary search on the index file would need $(\log_2 b_i) = (\log_2 28) = 5$ block accesses.
- To search for a record using the index, we need one additional block access to the data file (to read the data block) for a total of 5 + 1 = 6 block accesses—an improvement over binary search on the data file, which required 13 disk block accesses.

Single Level Indexing: Primary index: Advantages and Limitations

- The index file for a primary index occupies a much smaller space than does the data file, for two reasons.
  - First, there are fewer index entries than there are records in the data file.
  - Second, each index entry is typically smaller in size than a data record because it has only two fields, both of which tend to be short in size; consequently, more index entries than data records can fit in one block.
- Therefore, a binary search on the index file requires fewer block accesses than a binary search on the data file.
- A major problem with a primary index—as with any ordered file—is insertion and deletion of records.
  - With a primary index, the problem is compounded because if we attempt to insert a record in its correct position in the data file, we must not only move records to make space for the new record but also change some index entries, since moving records will change the anchor records of some blocks.

Single Level Indexing: Clustering index

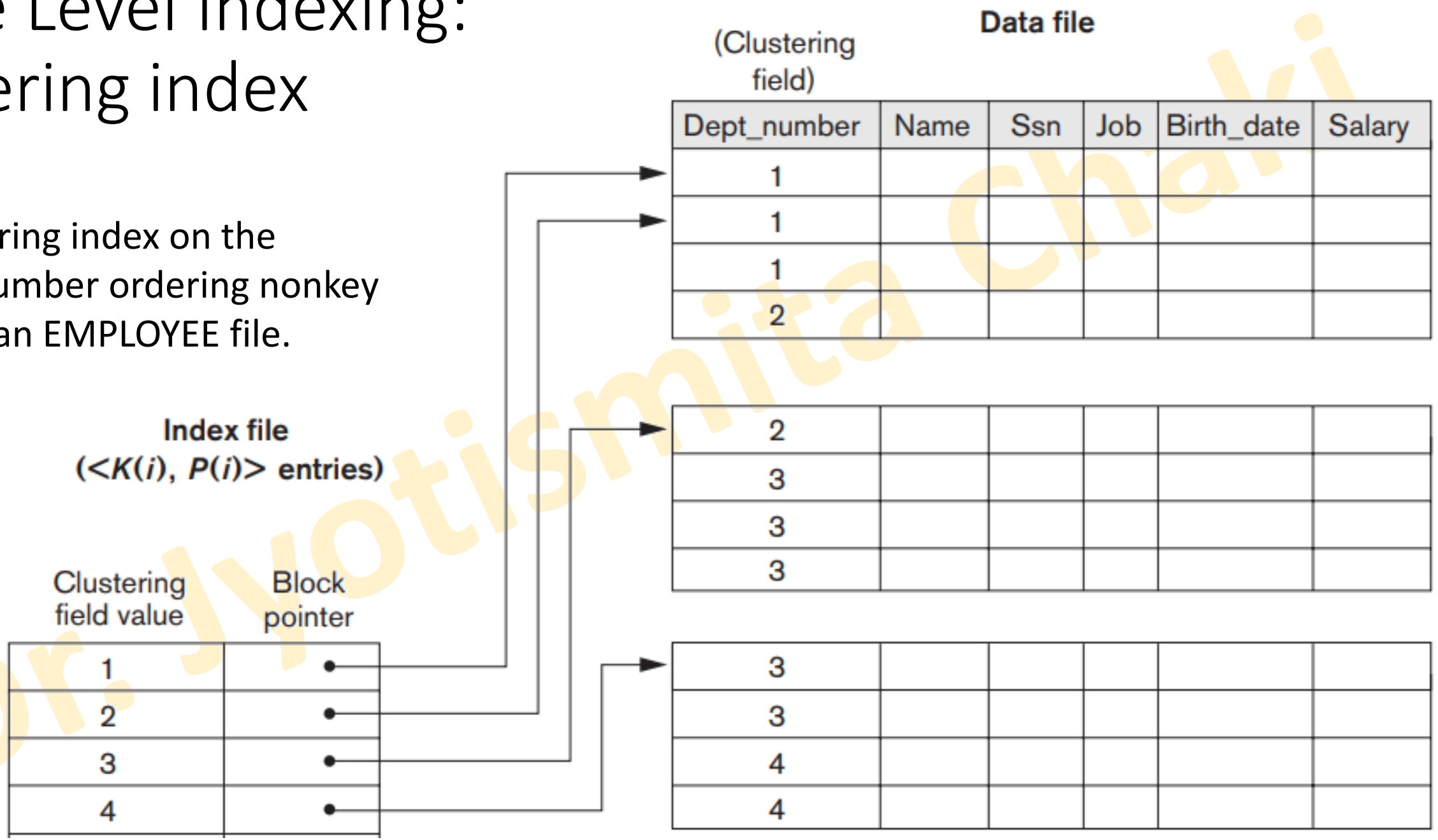
- If file records are physically ordered on a nonkey field—which does not have a distinct value for each record—that field is called the **clustering field** and the data file is called a **clustered file**.
- We can create a different type of index, called a **clustering index**, to speed up retrieval of all the records that have the same value for the clustering field.
- This *differs* from a primary index, which requires that the ordering field of the data file have a *distinct* value for each record.

Single Level Indexing: Clustering index

- A clustering index is an ordered file with two fields;
  - the first field is of the same type as the clustering field of the data file, and
  - the second field is a disk block pointer.
- There is one entry in the clustering index for each distinct value of the clustering field.
- It contains the value and a pointer to the first block in the data file that has a record with that value for its clustering field.

Single Level Indexing: Clustering index

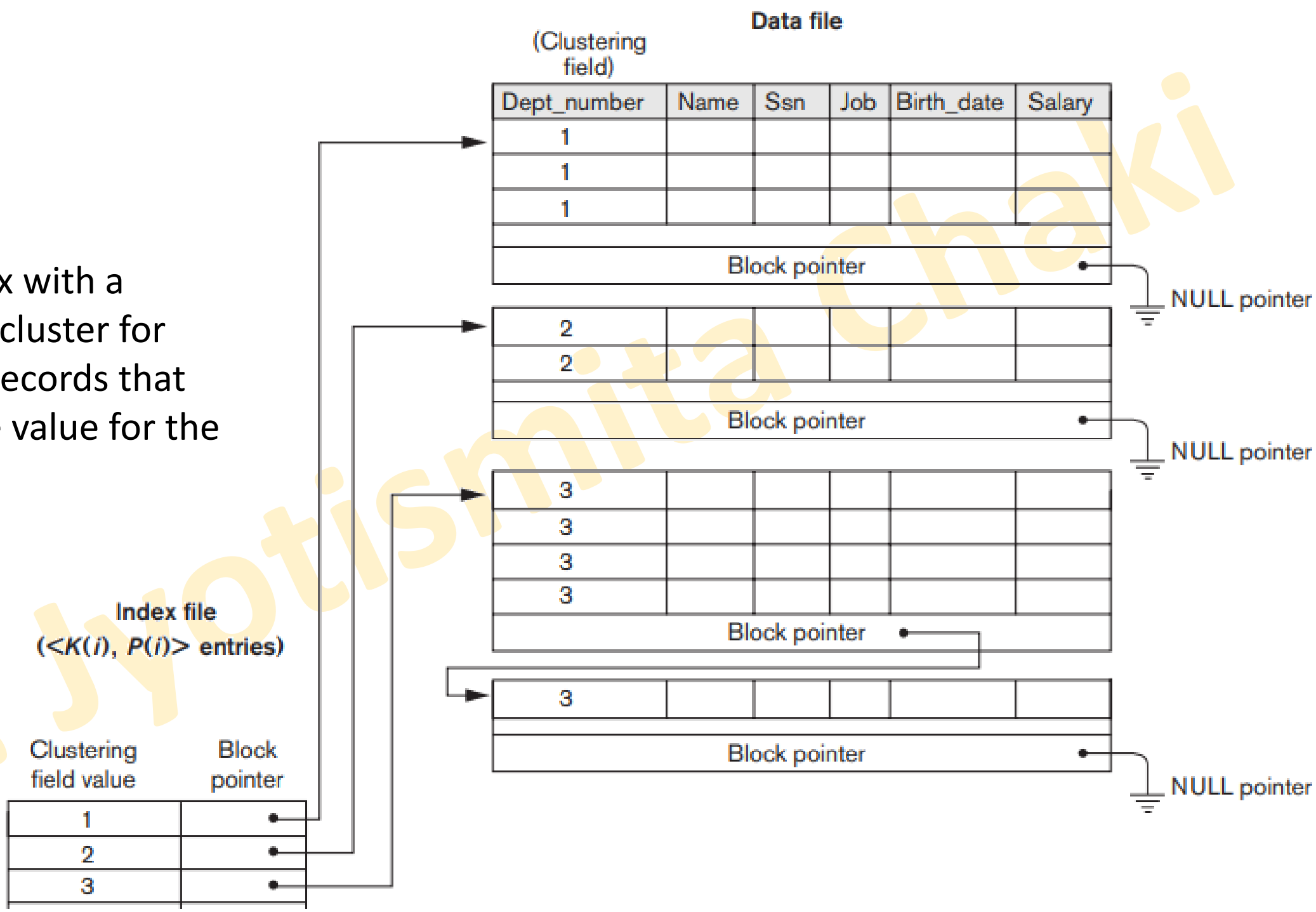
A clustering index on the Dept_number ordering nonkey field of an EMPLOYEE file.



Single Level Indexing: Clustering index

- In the above solution, the record insertion and deletion still cause problems because the data records are physically ordered.
- To alleviate the problem of insertion, it is common to reserve a whole block (or a cluster of contiguous blocks) for each value of the clustering field.
- All records with that value are placed in the block (or block cluster).
- This makes insertion and deletion relatively straightforward.

Clustering index with a separate block cluster for each group of records that share the same value for the clustering field.



Single Level Indexing: Clustering index

- Suppose that we consider the same ordered file with r = 300,000 records stored on a disk with block size B = 4,096 bytes. Imagine that it is ordered by the attribute Zipcode and there are 1,000 zip codes in the file (with an average 300 records per zip code, assuming even distribution across zip codes.)
- The index in this case has 1,000 index entries of 11 bytes each (5-byte Zipcode and 6-byte block pointer) with a blocking factor bfr_i = (B/Ri) = (4,096/11) = 372 index entries per block.
- The number of index blocks is hence $b_i = (r_i/bfr_i)$ = (1,000/372) = 3 blocks.
- To perform a binary search on the index file would need $(\log_2 b_i) = (\log_2 3)$ = 2 block accesses.

Single Level Indexing: Secondary index

- A **secondary index** provides a secondary means of accessing a data file for which some primary access already exists.
- The data file records could be ordered, unordered.
- The secondary index may be created on a field that is a candidate key and has a unique value in every record, or on a nonkey field with duplicate values.
- The index is an ordered file with two fields.
  - The first field is of the same data type as some non-ordering field of the data file that is an indexing field.
  - The second field is either a block pointer or a record pointer.

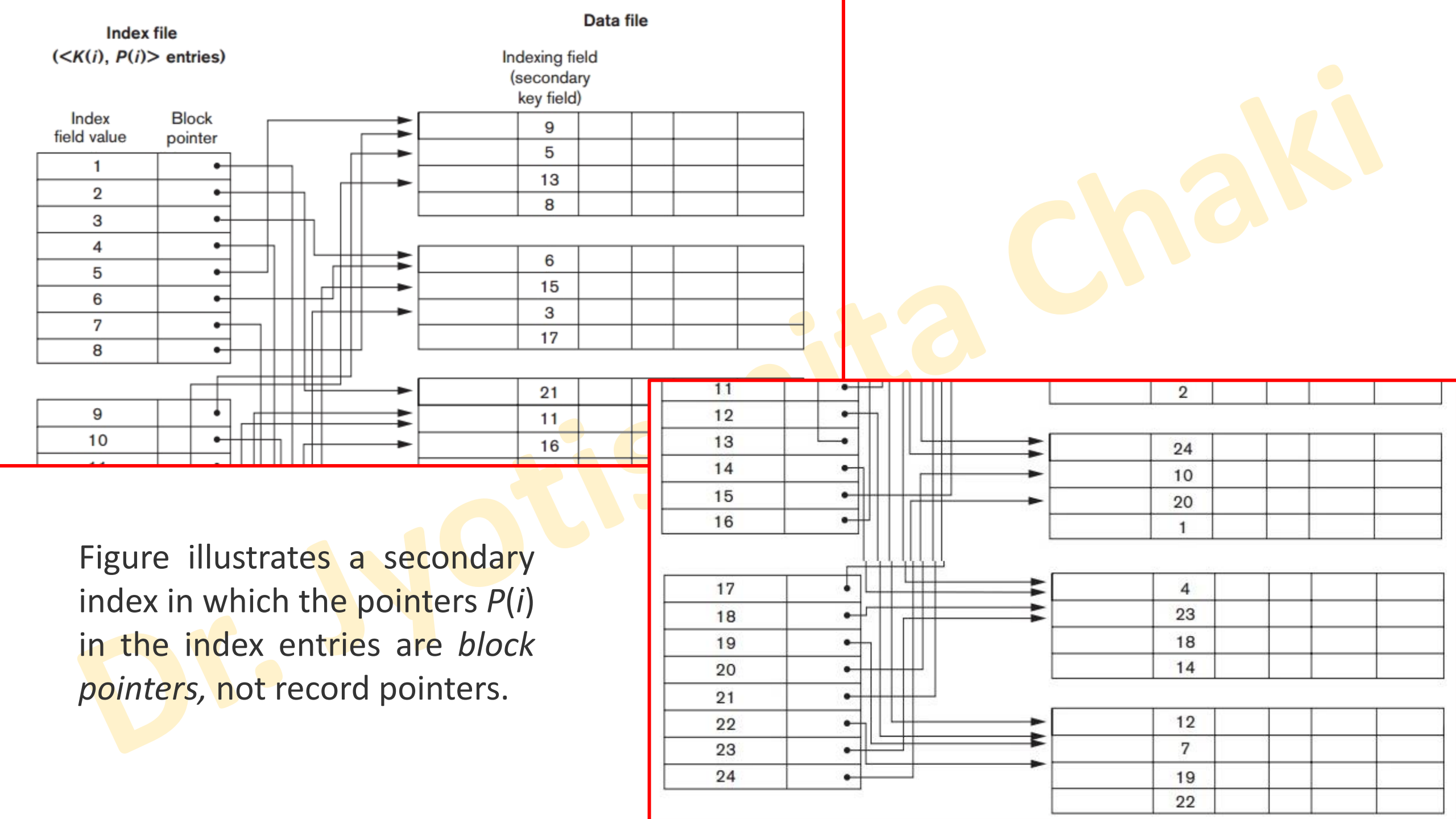
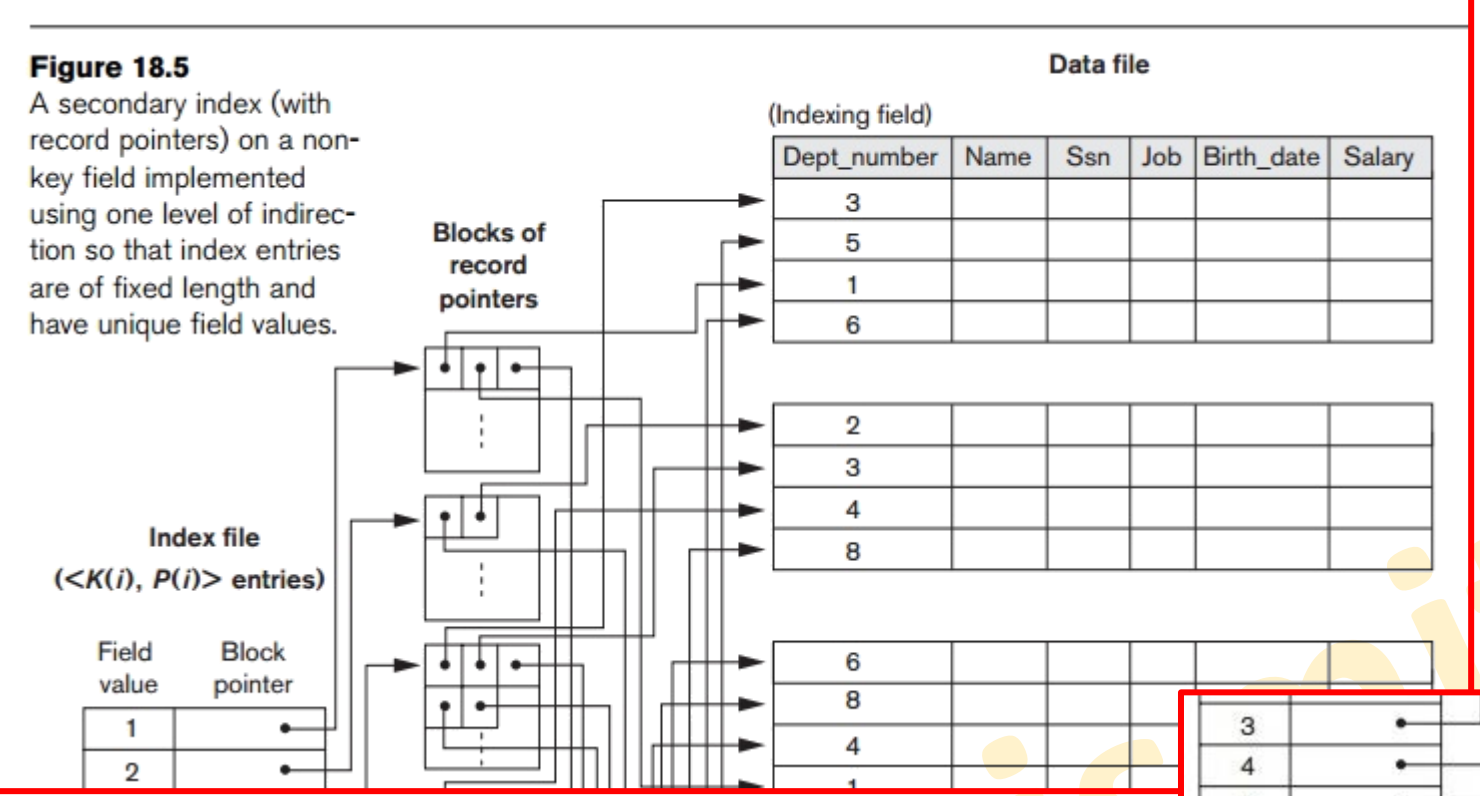
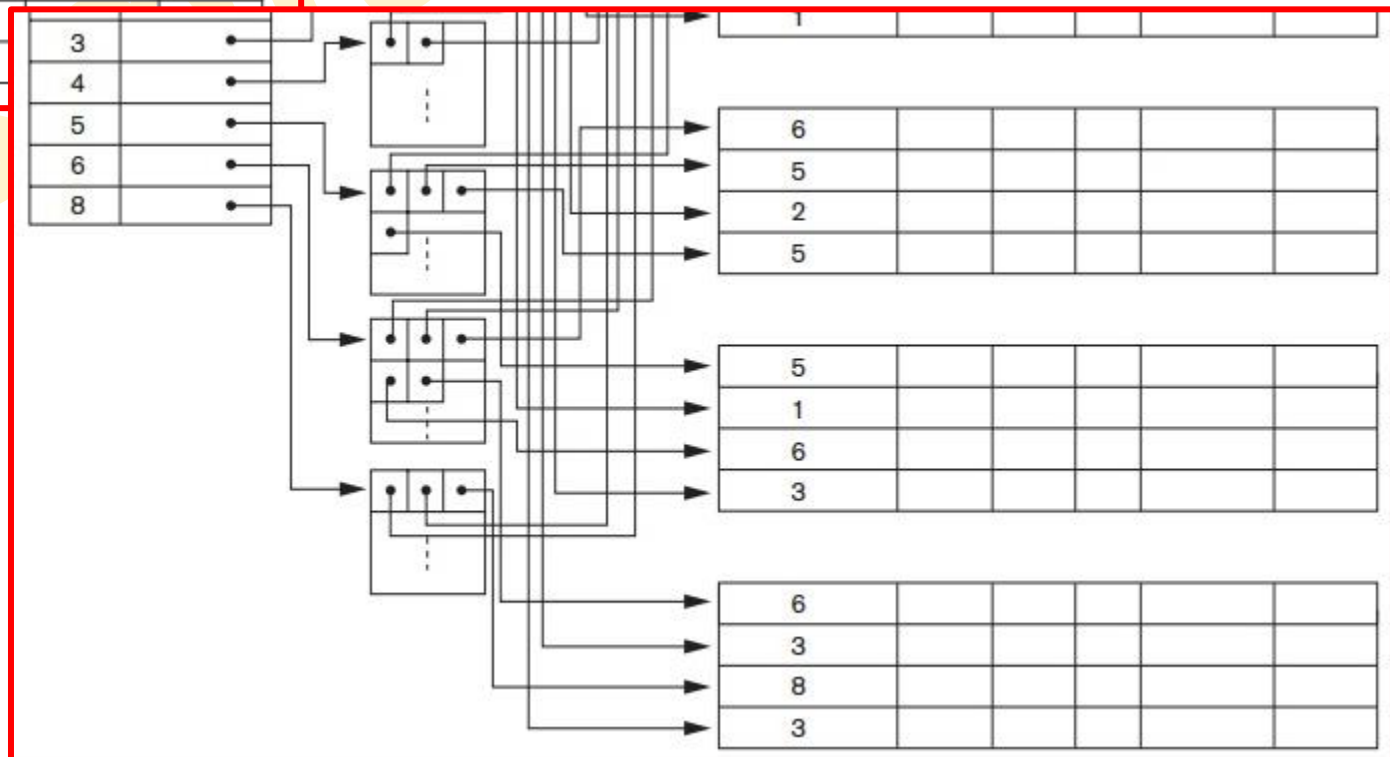


Figure illustrates a secondary index in which the pointers *P*(*i*) in the index entries are *block pointers,* not record pointers.

Figure 18.5
A secondary index (with record pointers) on a non-key field implemented using one level of indirection so that index entries are of fixed length and have unique field values.





Secondary index

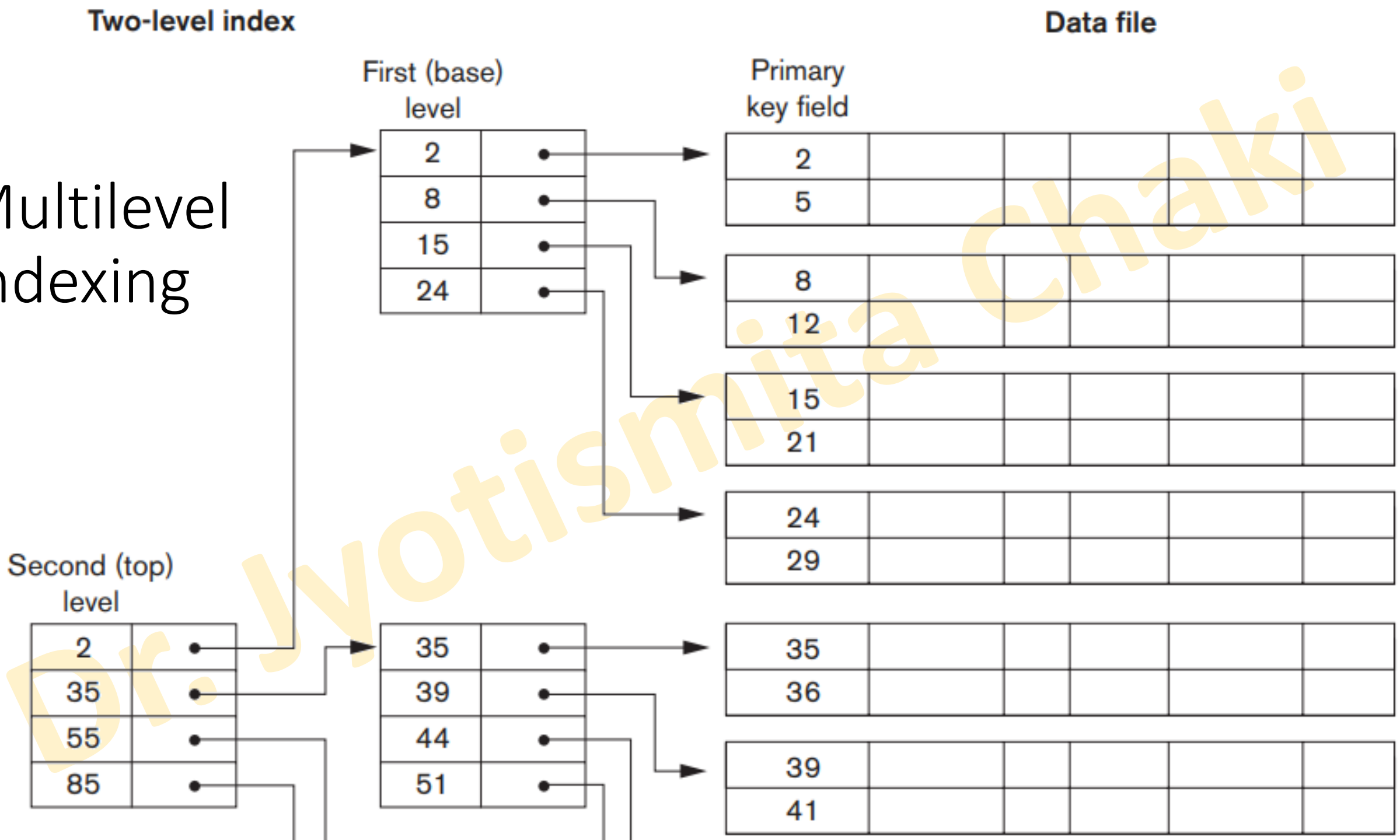
Multilevel Indexing

- Multilevel index is stored on the disk along with the actual database files.
- As the size of the database grows, so does the size of the indices.
- There is an immense need to keep the index records in the main memory so as to speed up the search operations.
- If single-level index is used, then a large size index cannot be kept in memory which leads to multiple disk accesses.
- Multi-level Index helps in breaking down the index into several smaller indices in order to make the outermost level so small that it can be saved in a single disk block, which can easily be accommodated anywhere in the main memory.

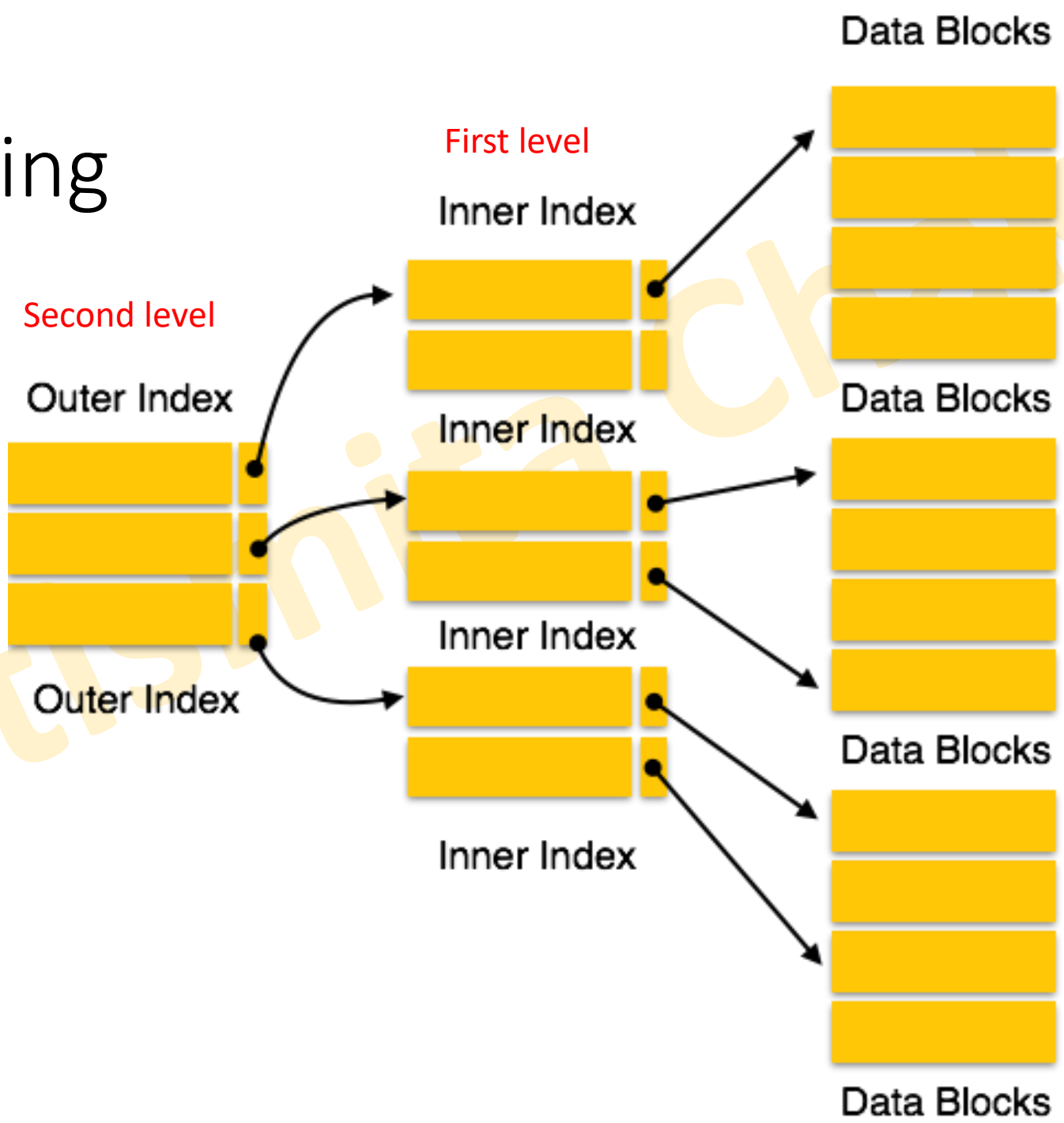
Multilevel Indexing

- The **first** (or **base**) **level** of a multilevel index is as an ordered file with a distinct value for each K(i).
- By considering the first-level index file as a sorted data file, we can create a primary index for the first level; this index to the first level is called the **second level** of the multilevel index.
- Because the second level is a primary index, we can use block anchors so that the second level has one entry for each block of the first level.
- The **third level**, which is a primary index for the second level, has an entry for each second-level block.
- We require a second level only if the first level needs more than one block of disk storage, and, similarly, we require a third level only if the second level needs more than one block.
- We can repeat the preceding process until all the entries of some index level *t* fit in a single block.
- This block at the t-th level is called the **top index** level.

Multilevel Indexing



Multilevel Indexing



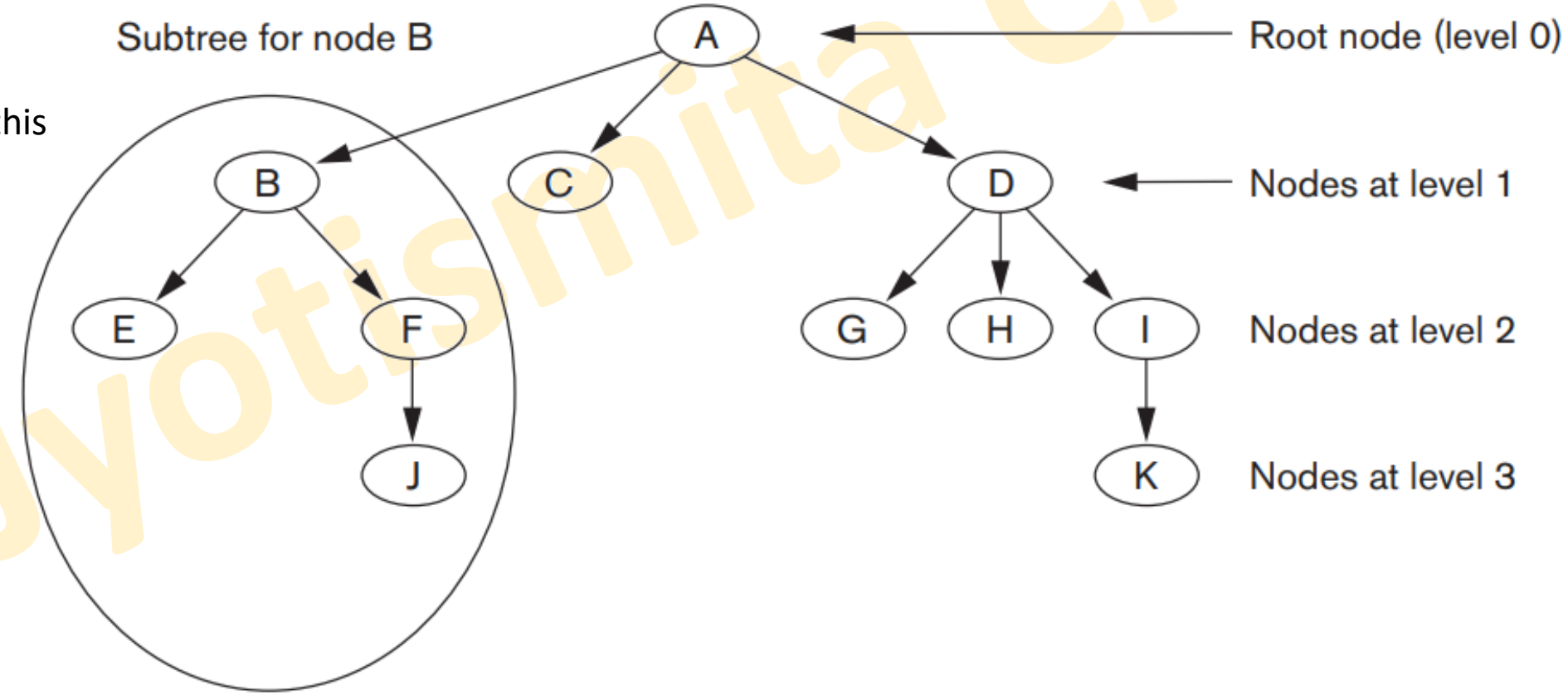
Dynamic Multilevel Indexing using B+ tree

- A **tree** is formed of **nodes**.
- Each node in the tree, except for a special node called the **root**, has one **parent node** and zero or more **child nodes**.
- The root node has no parent.
- A node that does not have any child nodes is called a **leaf** node; a nonleaf node is called an **internal** node.
- The level of a node is always one more than the level of its parent, with the level of the root node being *zero*.
- A subtree of a node consists of that node and all its descendant nodes—its child nodes, the child nodes of its child nodes, and so on.

Dynamic Multilevel Indexing using B+ tree

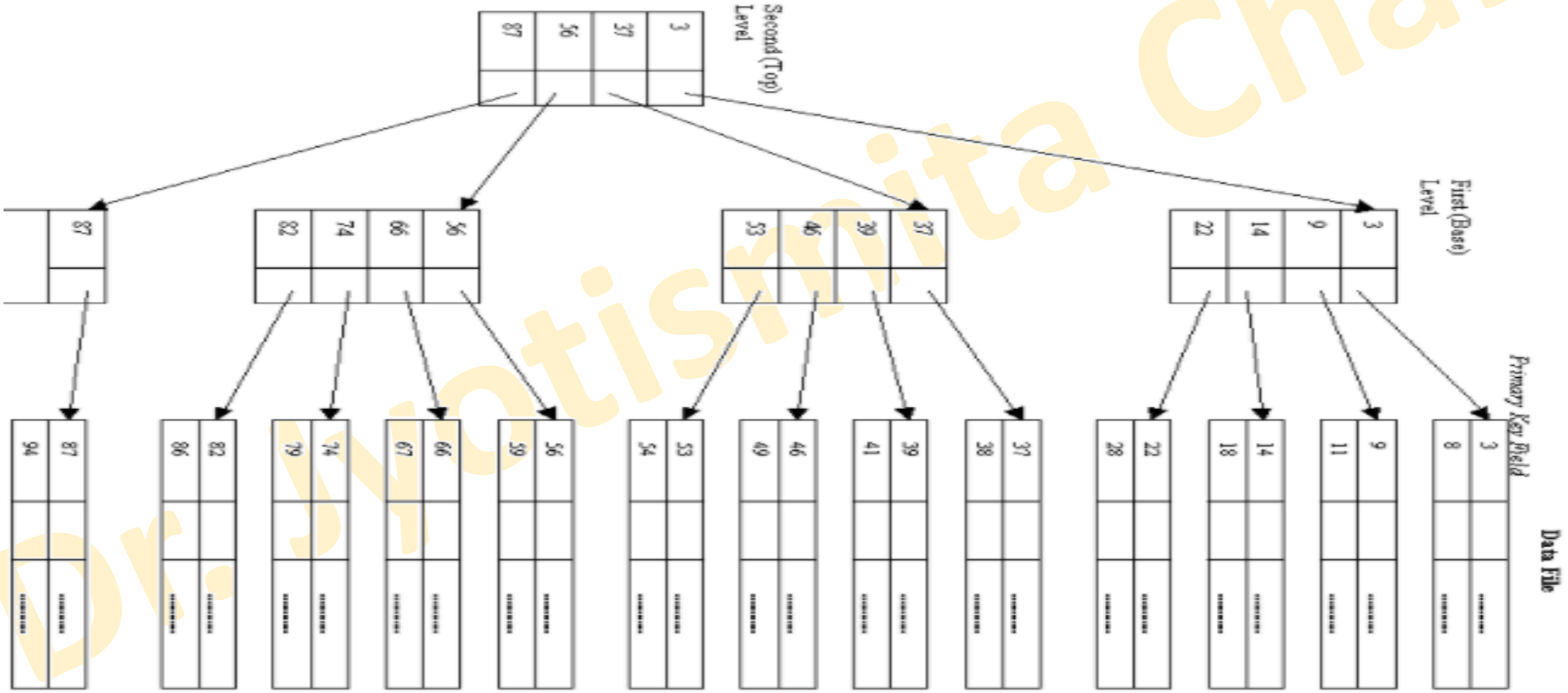
Since the leaf nodes are at different levels of the tree, this tree is called unbalanced.

A tree data structure that shows an unbalanced tree.



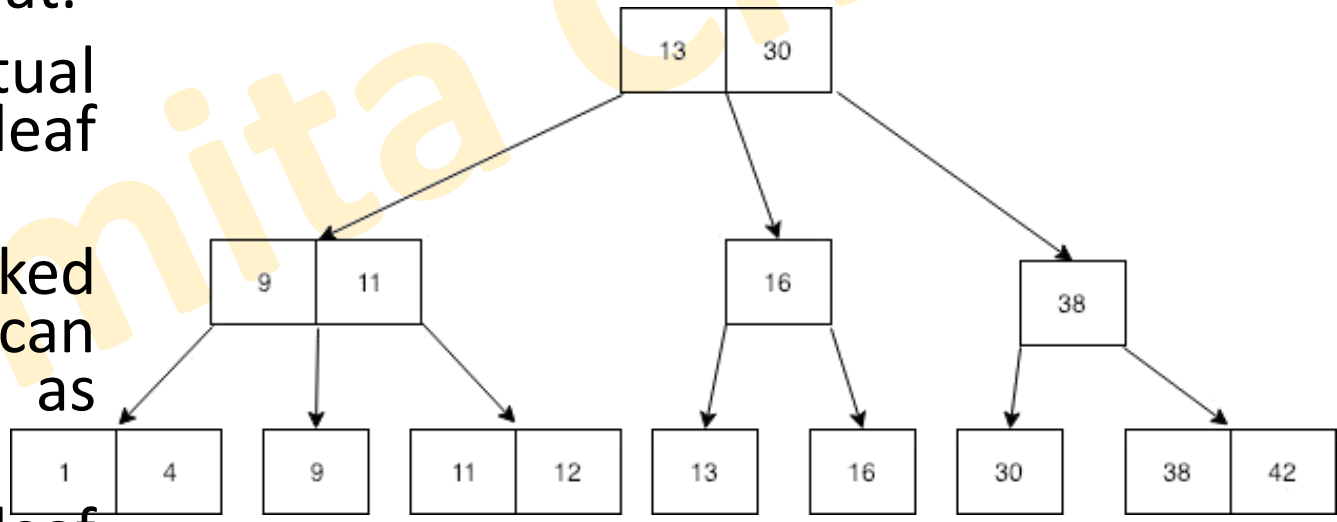
(Nodes E, J, C, G, H, and K are leaf nodes of the tree)

Dynamic Multilevel Indexing using B+ tree



Dynamic Multilevel Indexing using B+ tree

- The B+ tree is a balanced binary search tree. It follows a multi-level index format.
- In the B+ tree, leaf nodes denote actual data pointers. B+ tree ensures that all leaf nodes remain at the same height.
- In the B+ tree, the leaf nodes are linked using a link list. Therefore, a B+ tree can support random access as well as sequential access.
- B+tree is that it stores data on the leaf nodes.
- This means that all non-leaf node values are duplicated in leaf nodes again. Below is a sample B+tree.



13, 30, 9, 11, 16, and 38 non-leaf values are again repeated in leaf nodes.

Dynamic Multilevel Indexing using B+ tree

- Each internal node has at most m (order) tree pointers.
- Each internal node, except the root, has at least $\lceil (m/2) \rceil$ tree pointers.
- All leaf nodes are at the same level.
- Maximum no. of keys = m-1.
- Minimum no. of keys = $\lceil (m/2) \rceil - 1$

Dynamic Multilevel Indexing using B+ tree: Insertion

1. Even inserting at-least 1 entry into the leaf container does not make it full then add the record
2. Else, divide the node into more locations to fit more records.
   a) Assign a new leaf and transfer 50 percent of the node elements to a new placement in the tree
   b) The minimum key of the binary tree leaf and its new key address are associated with the top-level node.
   c) Divide the top-level node if it gets full of keys and addresses.
      i. Similarly, insert a key in the center of the top-level node in the hierarchy of the Tree.
   d) Continue to execute the above steps until a top-level node is found that does not need to be divided anymore.
3. Build a new top-level root node of 1 Key and 2 indicators.

Dynamic Multilevel Indexing using B+ tree: Insertion

CASE: MIN KEYS

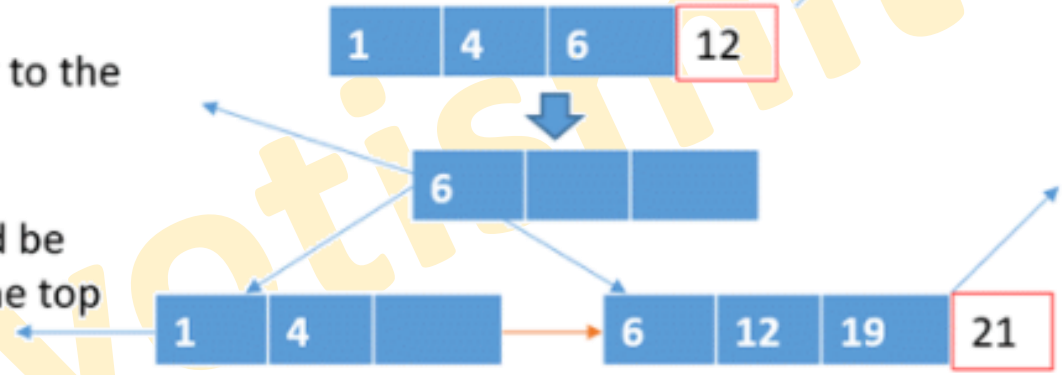
Order (m) = 4
Max children = 4
Min children = 2
Max Keys = 3
Min Keys = 1
Data: 1,4,6,12,19,21,31

Cannot add 10 here because max keys 3. Middle element can 4 or 6, we will take 6, split the node and make a right biased tree.

(6) is just a pointer to the leaf node.

All data should be present in this node and must be equal to or greater than top node (6)

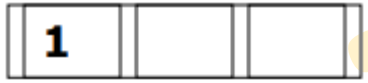
Data on left should be strictly less than the top node (6)



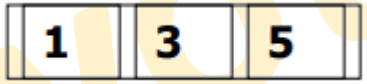
Leaf nodes connected with a link

Dynamic Multilevel Indexing using B+ tree: Insertion

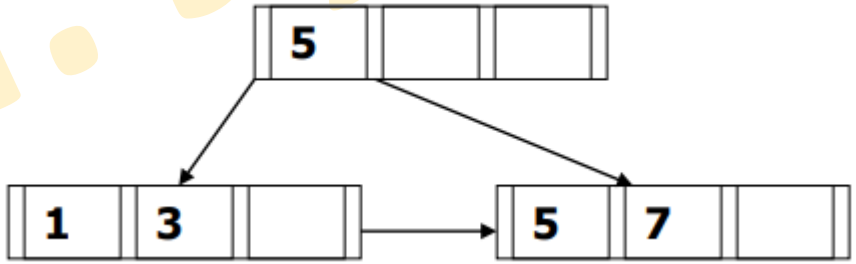
- Suppose each B+-tree node can hold up to 4 pointers and 3 keys.
- Insert 1, 3, 5, 7, 9, 2, 4, 6, 8, 10
- Step1: Insert 1



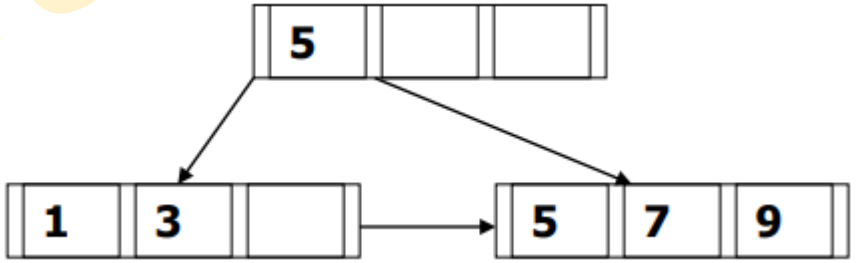
- Step2: Insert 3, 5



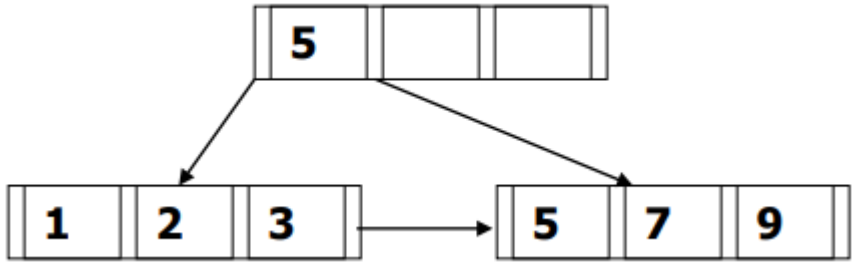
- Step3: Insert 7



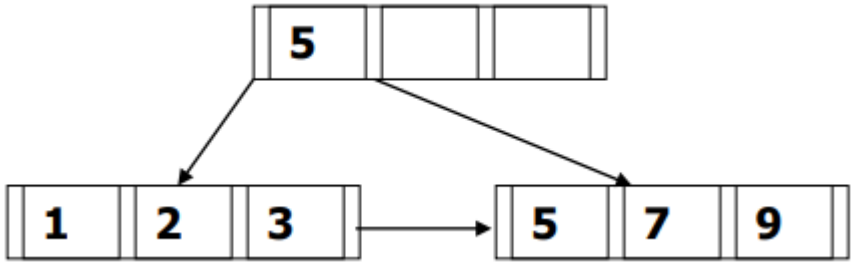
Step4: Insert 9



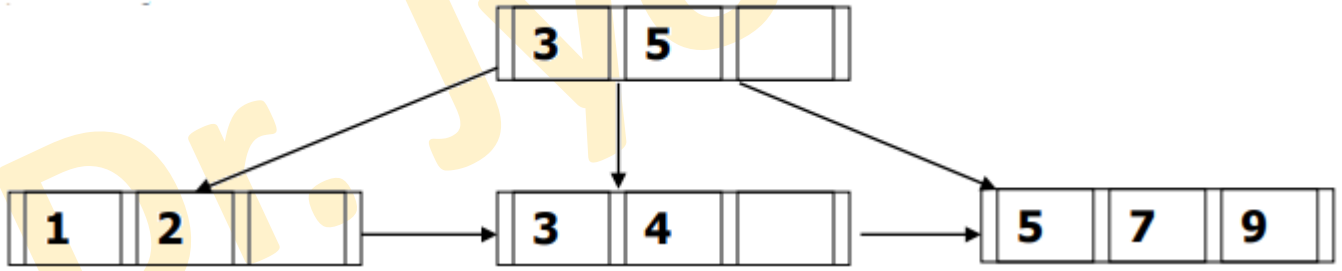
Step5: Insert 2



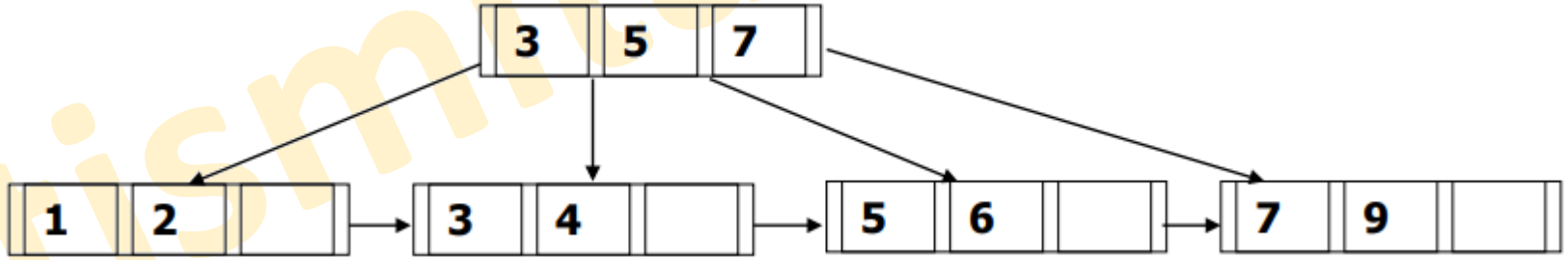
Dynamic Multilevel Indexing using B+ tree: Insertion



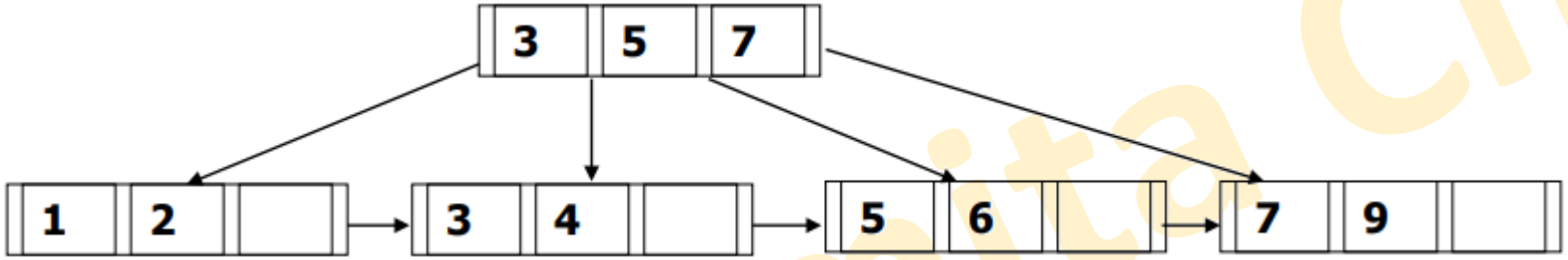
- Step 6: Insert 4



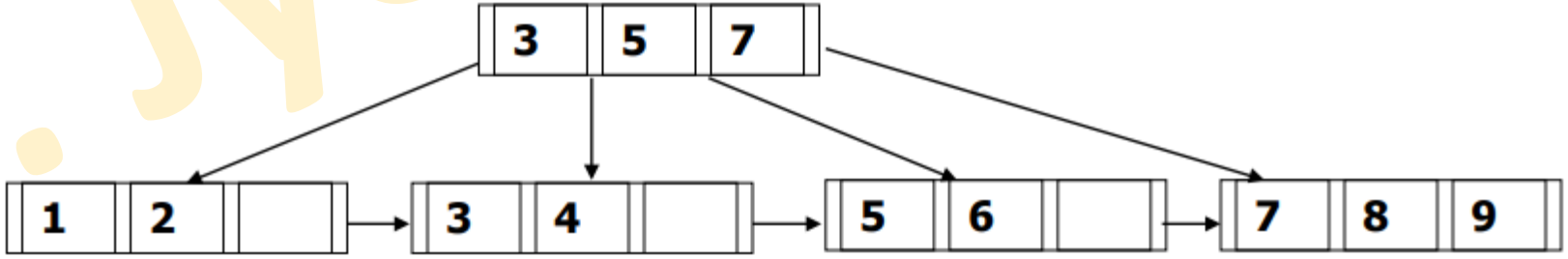
Step7: Insert 6



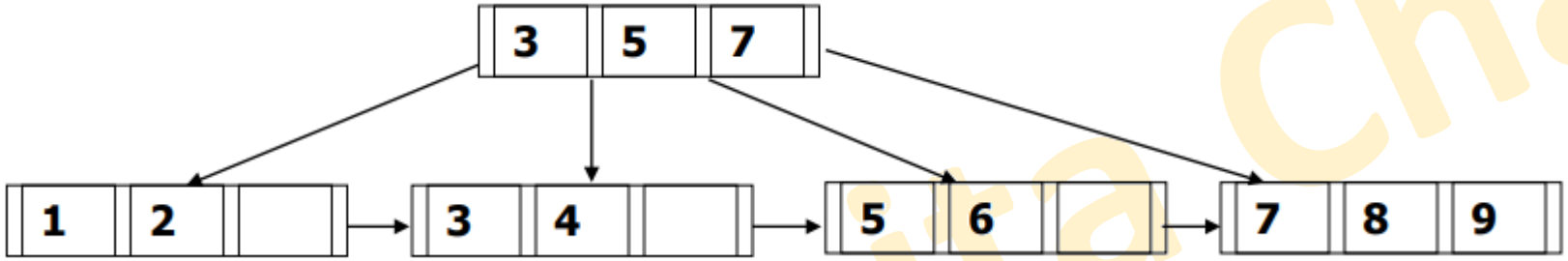
Dynamic Multilevel Indexing using B+ tree: Insertion



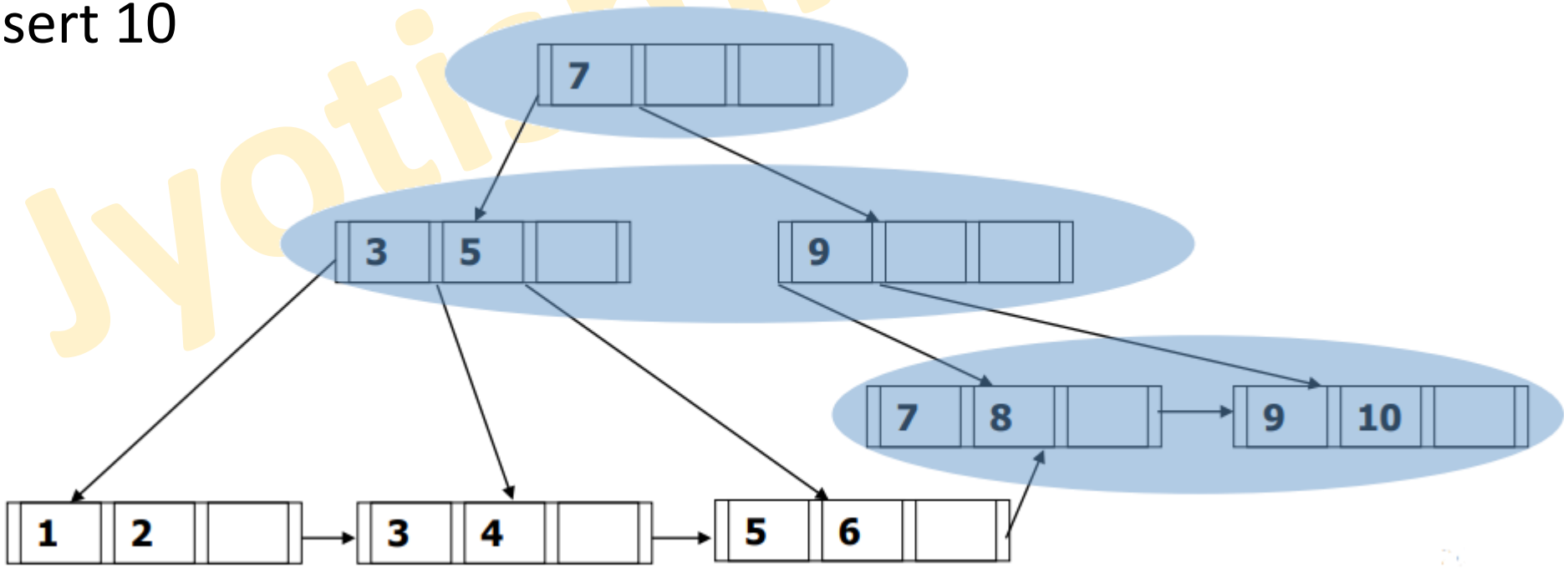
- Step 8: Insert 8



Dynamic Multilevel Indexing using B+ tree: Insertion

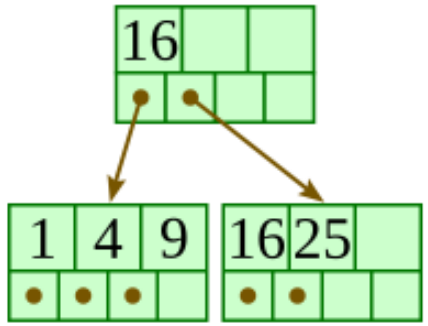


- Step9: Insert 10

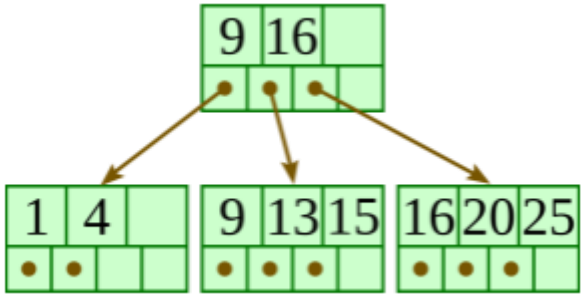


Dynamic Multilevel Indexing using B+ tree: Insertion

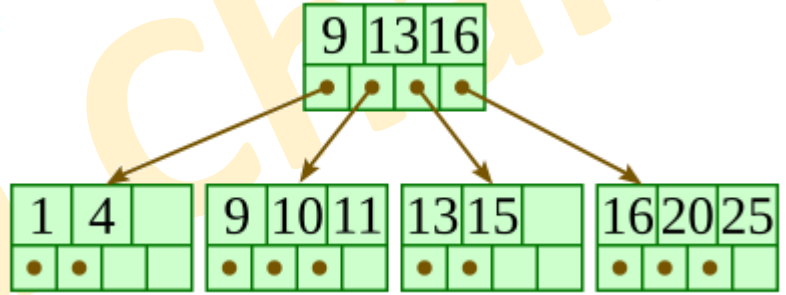
Initial:



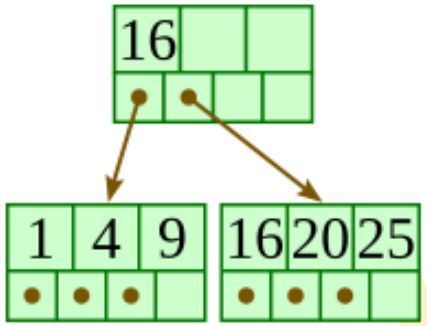
Insert 15:



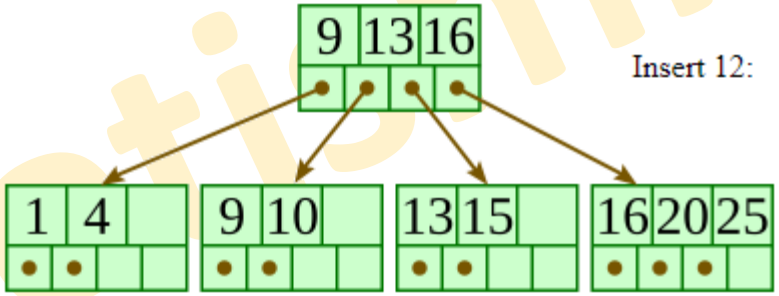
Insert 11:



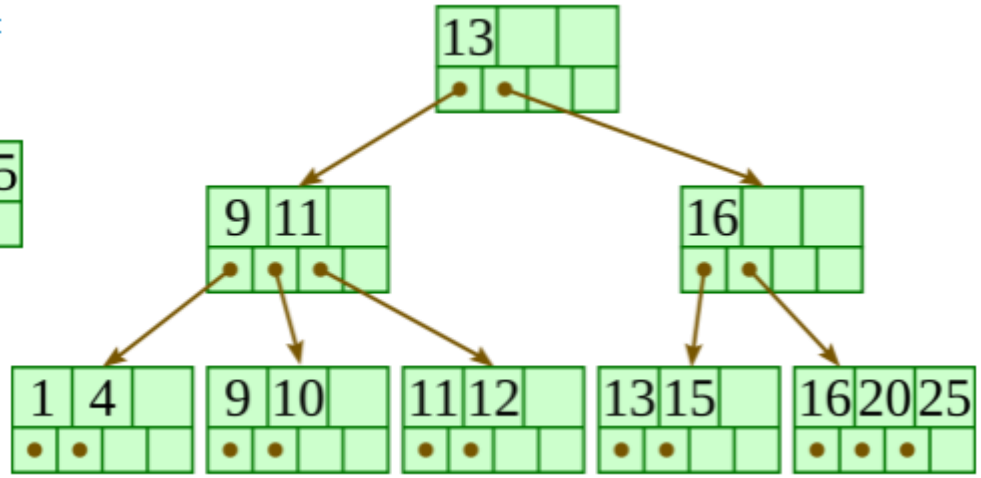
Insert 20:



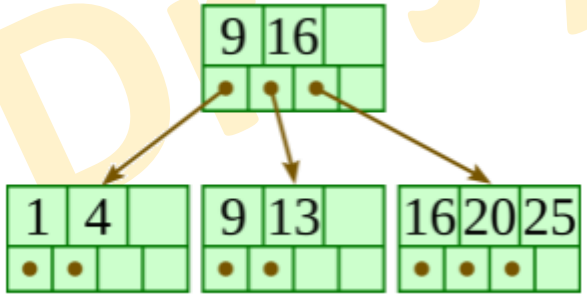
Insert 10:



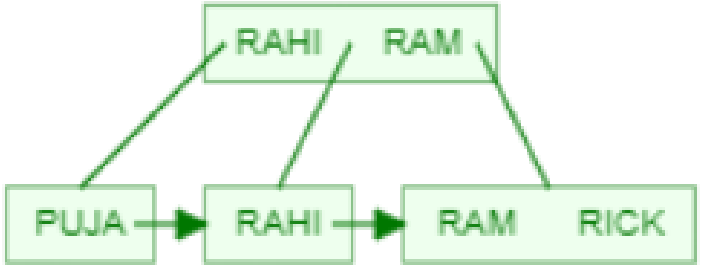
Insert 12:

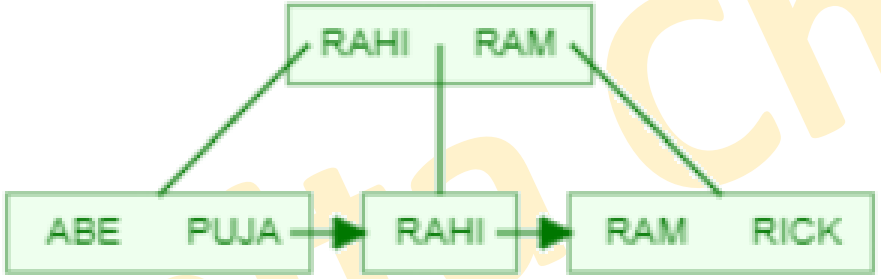


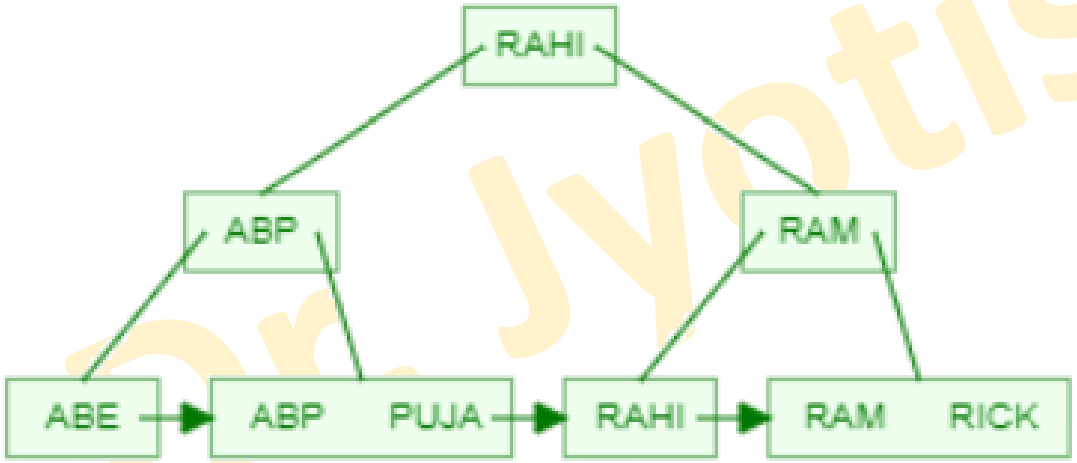
Insert 13:

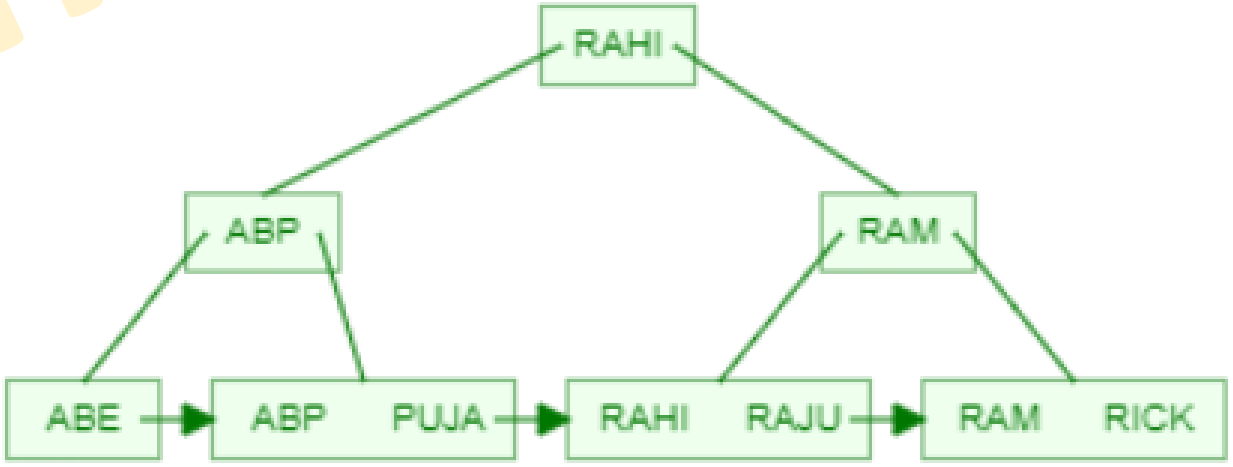


Dynamic Multilevel Indexing using B+ tree: Max degree = 3: Insert ABE, ABP, RAJU

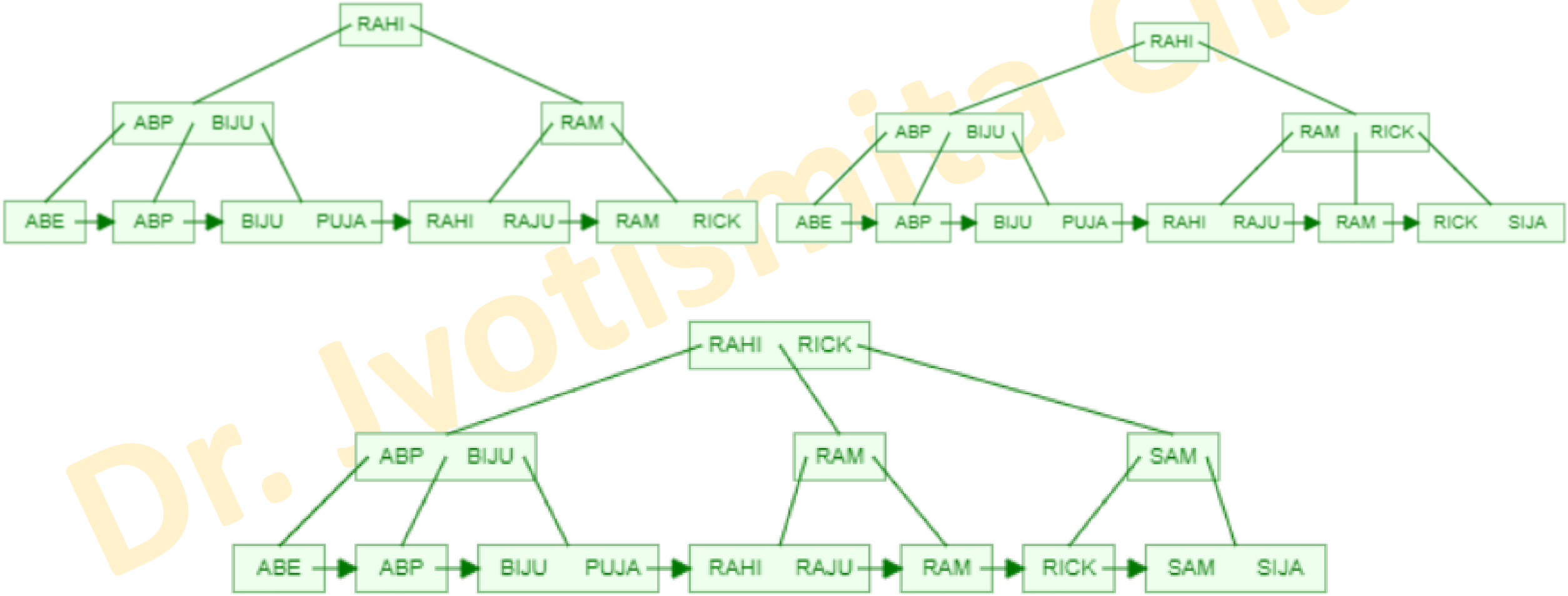




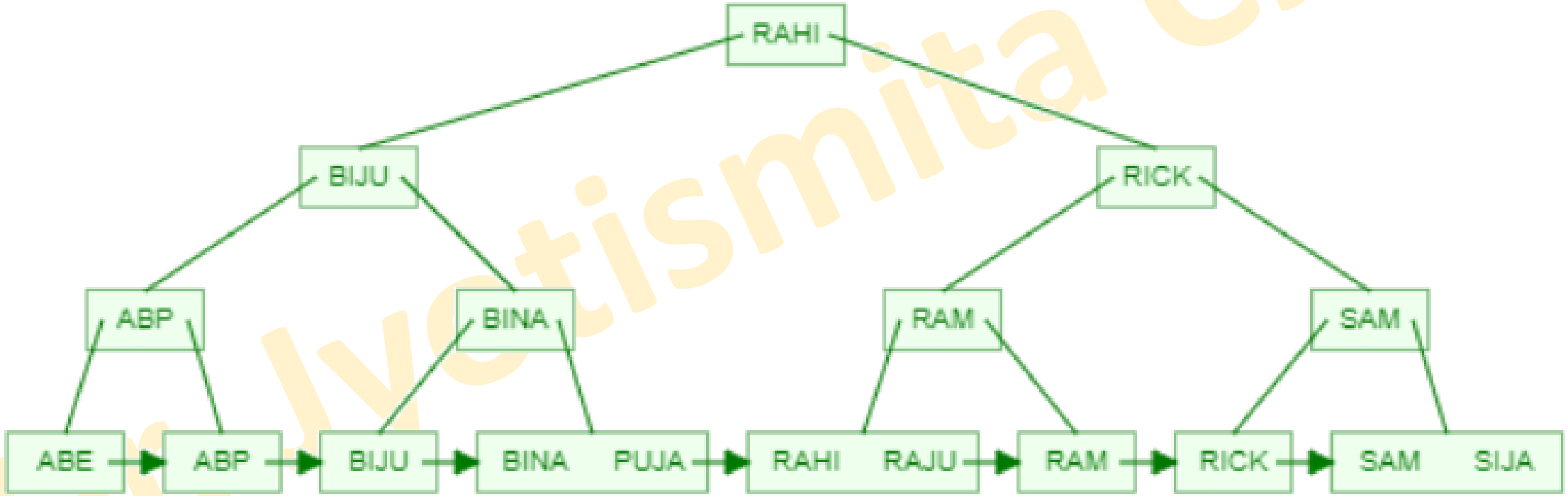




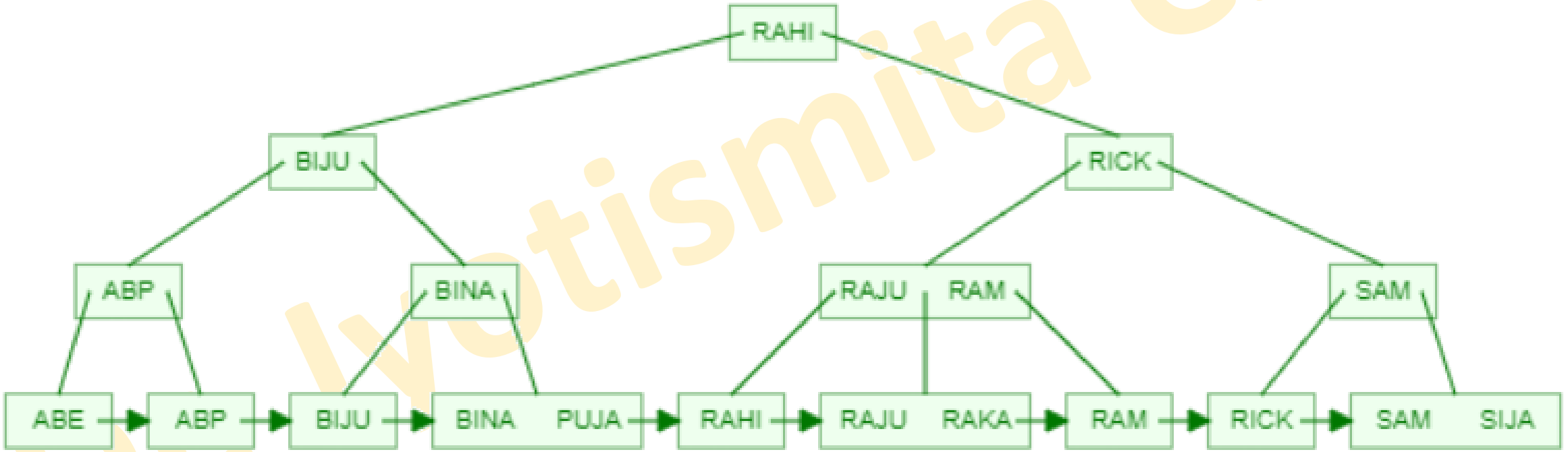
Dynamic Multilevel Indexing using B+ tree: Max degree = 3: Insert BUJU, SIJA, SAM



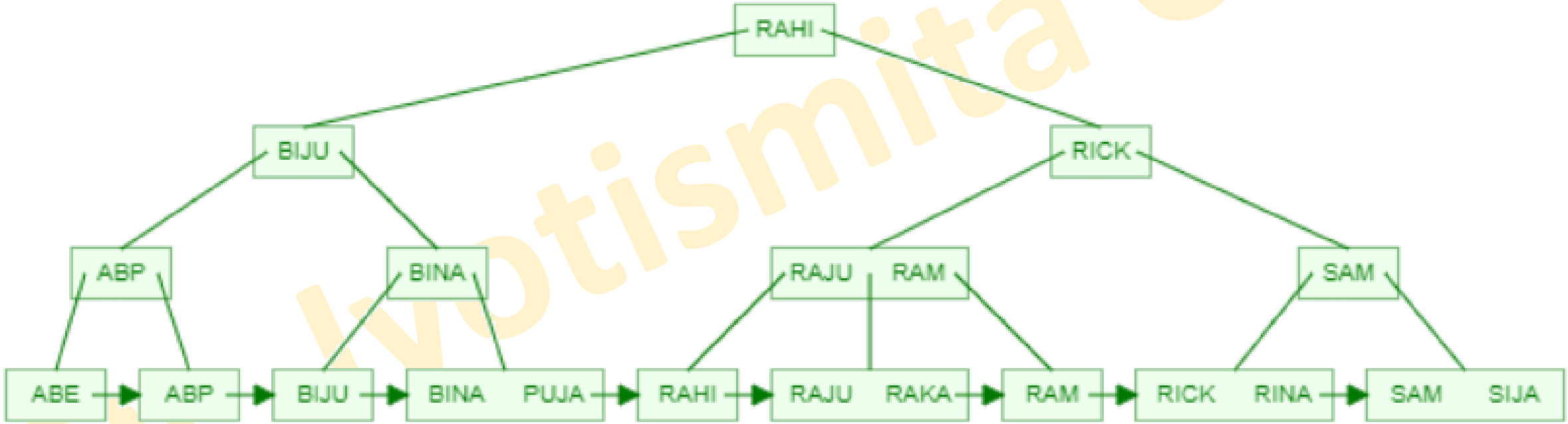
Dynamic Multilevel Indexing using B+ tree: Max degree = 3: Insert BINA



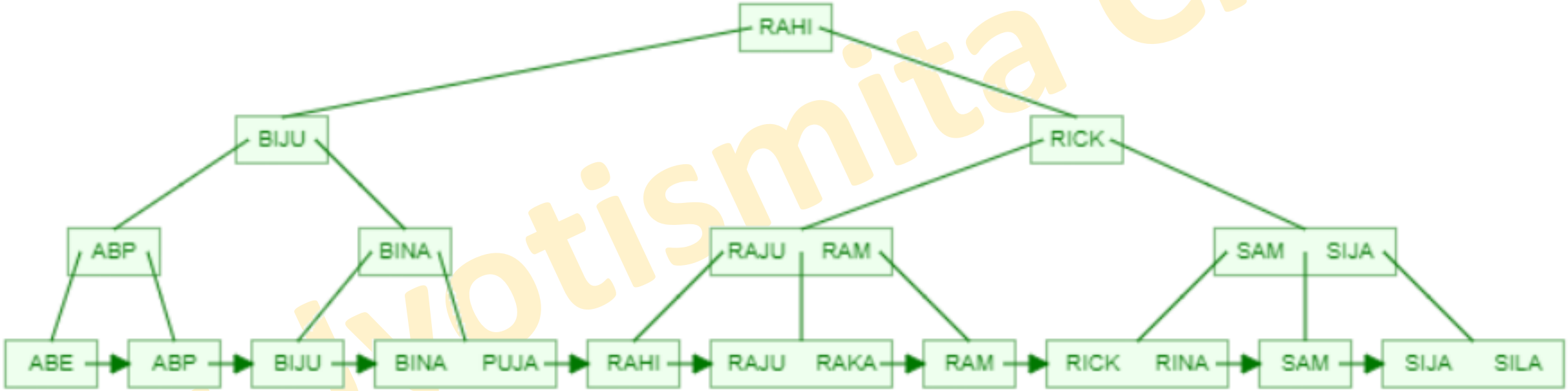
Dynamic Multilevel Indexing using B+ tree: Max degree = 3: Insert RAKA



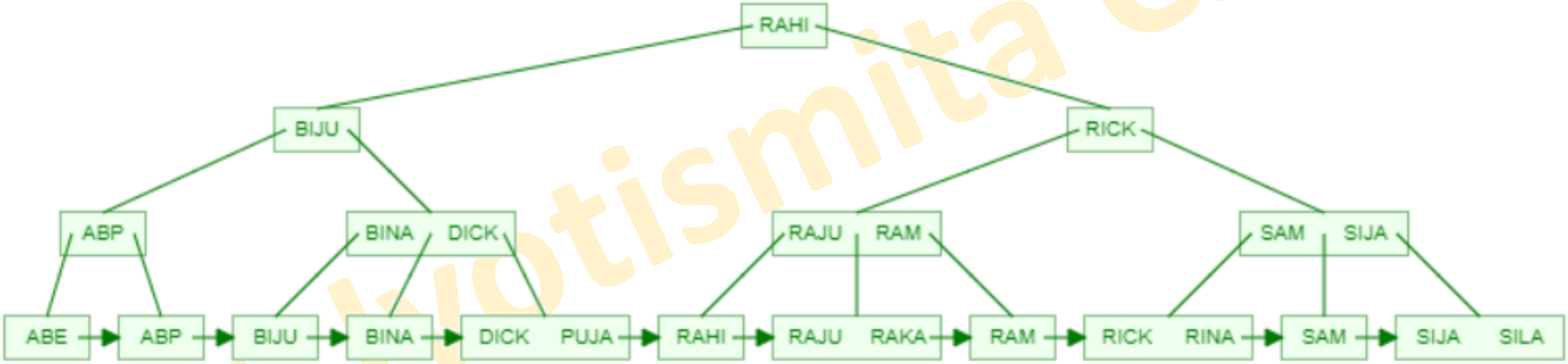
Dynamic Multilevel Indexing using B+ tree: Max degree = 3: Insert RINA



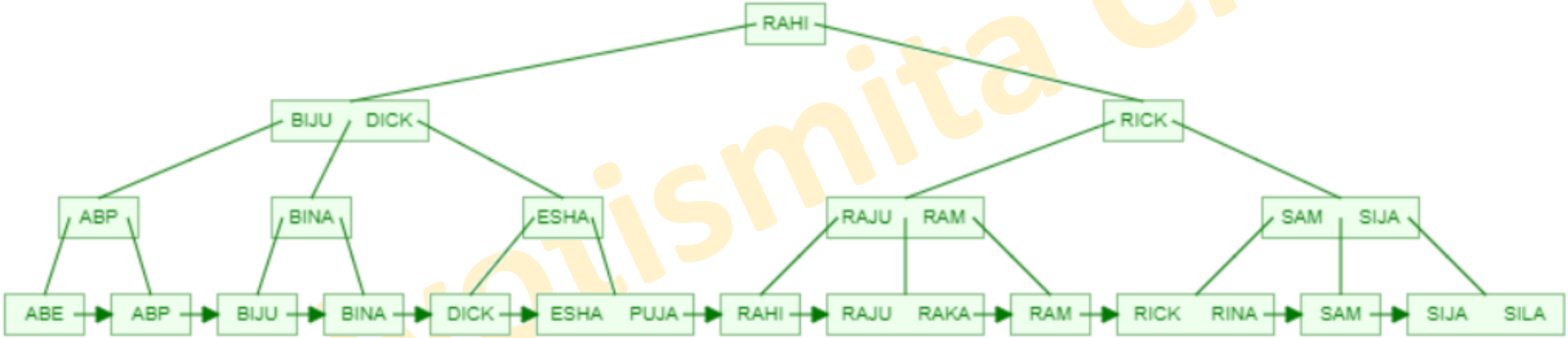
Dynamic Multilevel Indexing using B+ tree: Max degree = 3: Insert SILA



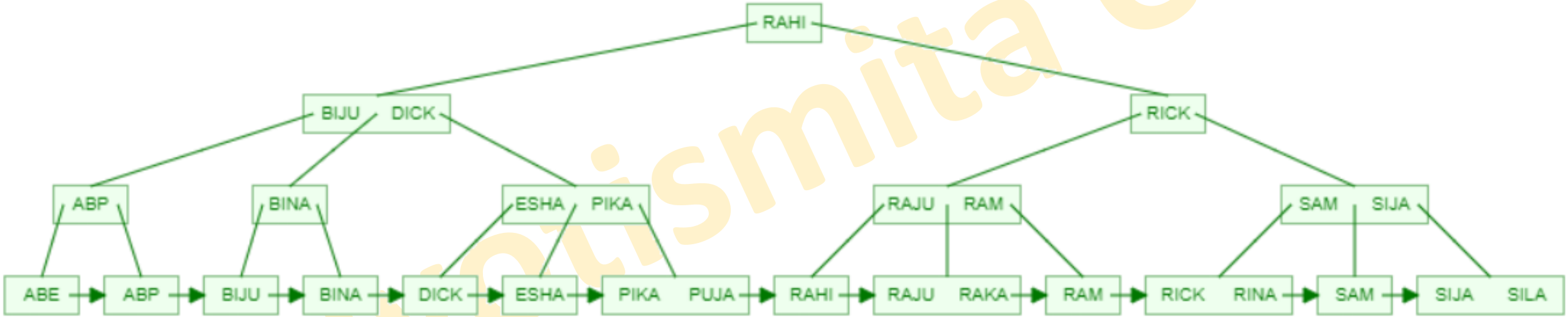
Dynamic Multilevel Indexing using B+ tree: Max degree = 3: Insert DICK



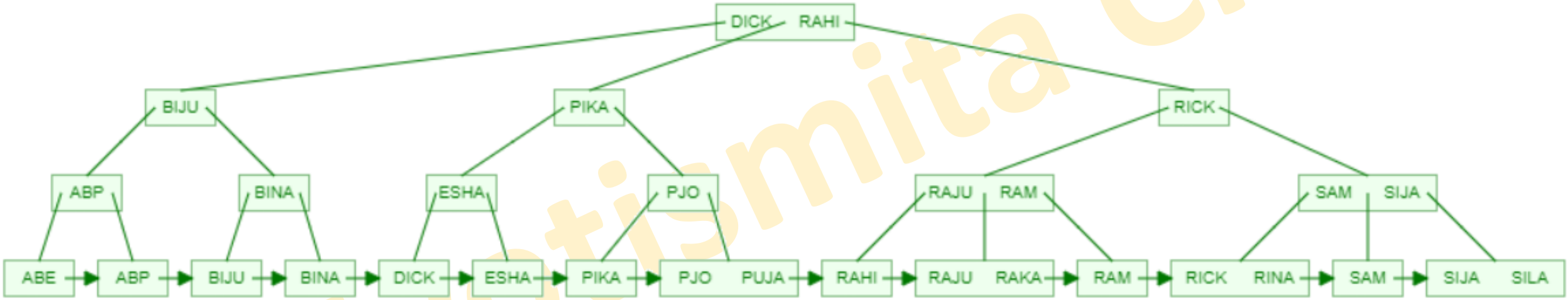
Dynamic Multilevel Indexing using B+ tree: Max degree = 3: Insert ESHA



Dynamic Multilevel Indexing using B+ tree: Max degree = 3: Insert PIKA



Dynamic Multilevel Indexing using B+ tree: Max degree = 3: Insert PJO



Dynamic Multilevel Indexing using B+ tree: Deletion

1. Start at the root and go up to leaf node containing the key K
2. Find the node n on the path from the root to the leaf node containing K
   A. If n is root, remove K
      a) if root has more than one key, done
      b) if root has only K
         i. if any of its child nodes can lend a node Borrow key from the child and adjust child links
         ii. Otherwise merge the children nodes. It will be a new root
      c) If n is an internal node, remove K
         i. If n has at least ceil(m/2) keys, done!
         ii. If n has less than ceil(m/2) keys, If a sibling can lend a key, Borrow key from the sibling and adjust keys in n and the parent node Adjust child links

Else

Merge n with its sibling Adjust child links

Dynamic Multilevel Indexing using B+ tree: Deletion (contd…)

d) If n is a leaf node, remove K

i. If n has at least ceil(M/2) elements, done! In case the smallest key is deleted, push up the next key

ii. If n has less than ceil(m/2) elements If the sibling can lend a key Borrow key from a sibling and adjust keys in n and its parent node

Else

Merge n and its sibling Adjust keys in the parent node

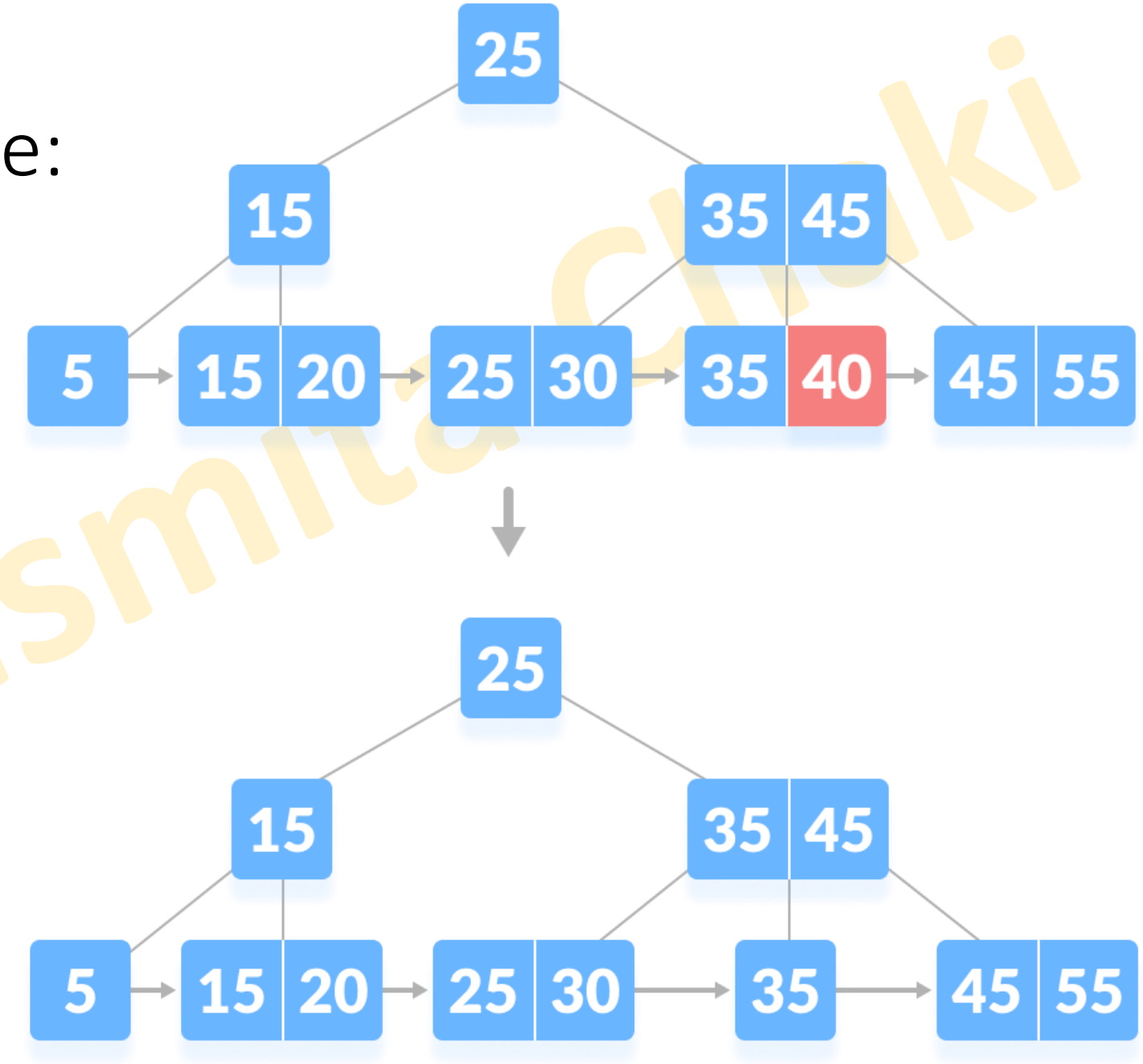
Dynamic Multilevel Indexing using B+ tree: Deletion

Before going through the deletion steps, we must know these facts about a B+ tree of degree **m**.

1. A node can have a maximum of m children. (i.e. 3)
2. A node can contain a maximum of m - 1 keys. (i.e. 2)
3. A node should have a minimum of ⌈m/2⌉ children. (i.e. 2)
4. A node (except root node) should contain a minimum of ⌈m/2⌉ - 1 keys. (i.e. 1)

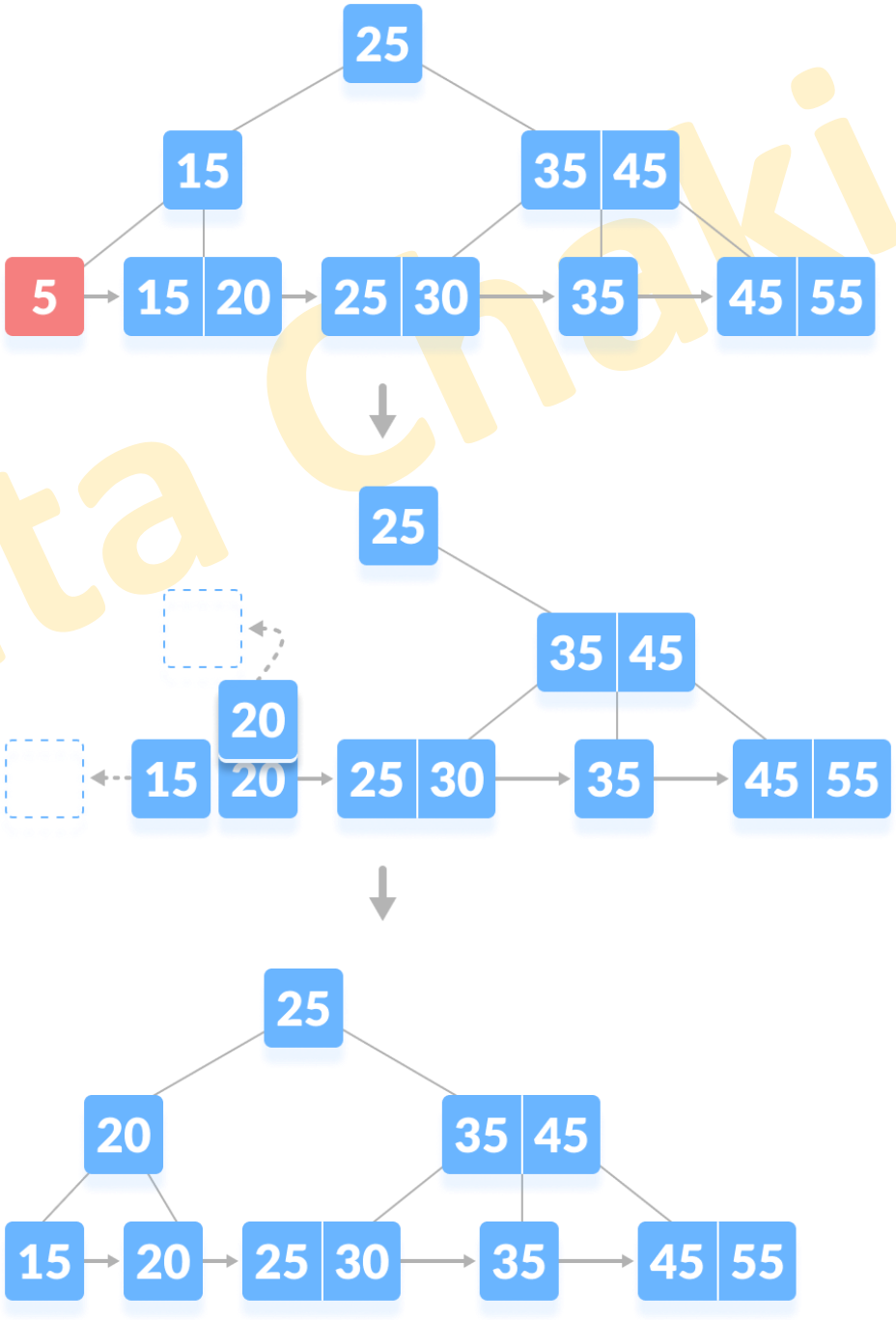
Dynamic Multilevel Indexing using B+ tree: Deletion

- **Case I:** The key to be deleted is present only at the leaf node not in the indexes (or internal nodes). There are two cases for it:
  1. There is more than the minimum number of keys in the node. Simply delete the key.



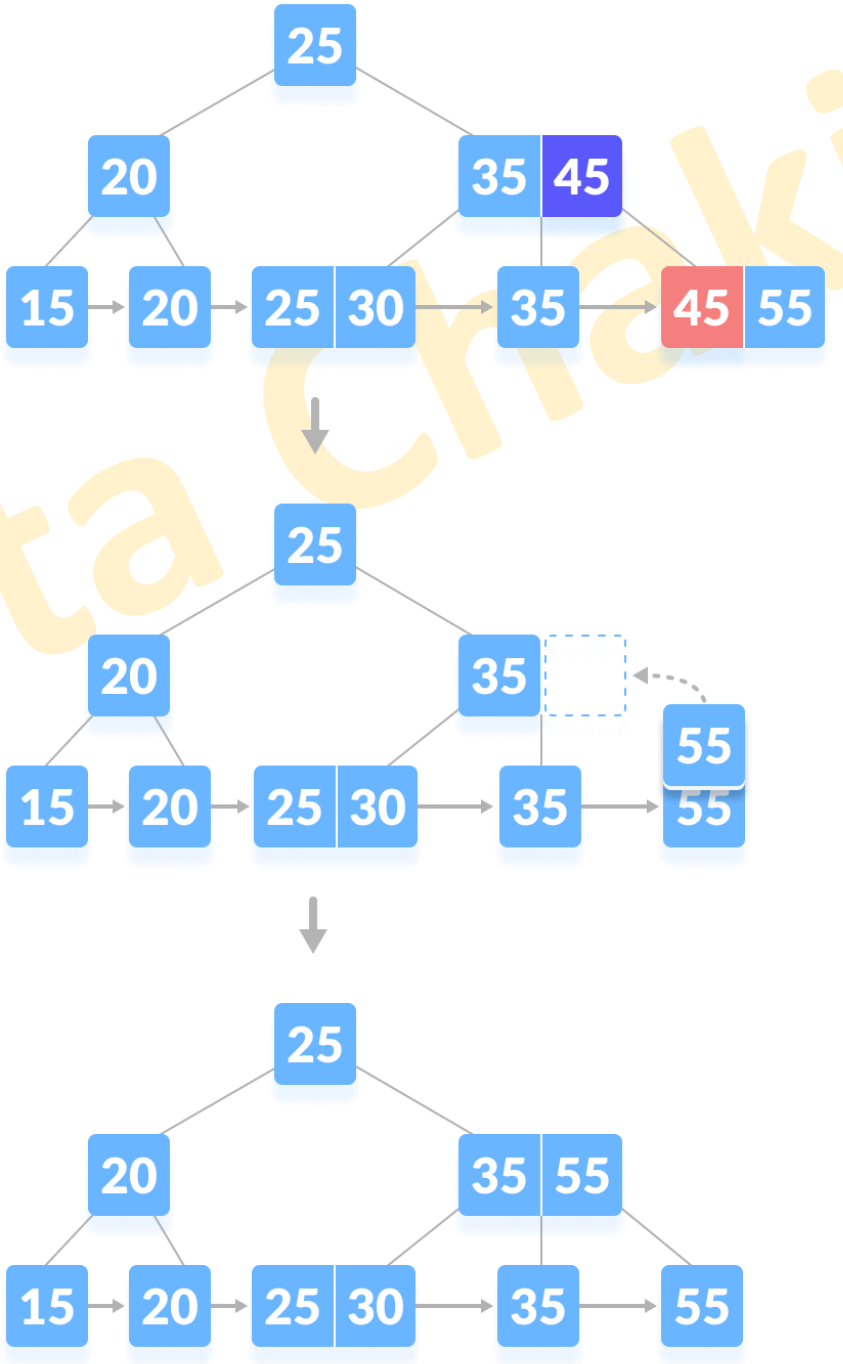
Dynamic Multilevel Indexing using B+ tree: Deletion

- **Case I:** The key to be deleted is present only at the leaf node not in the indexes (or internal nodes). There are two cases for it:
  2. There is an exact minimum number of keys in the node. Delete the key and borrow a key from the immediate sibling.



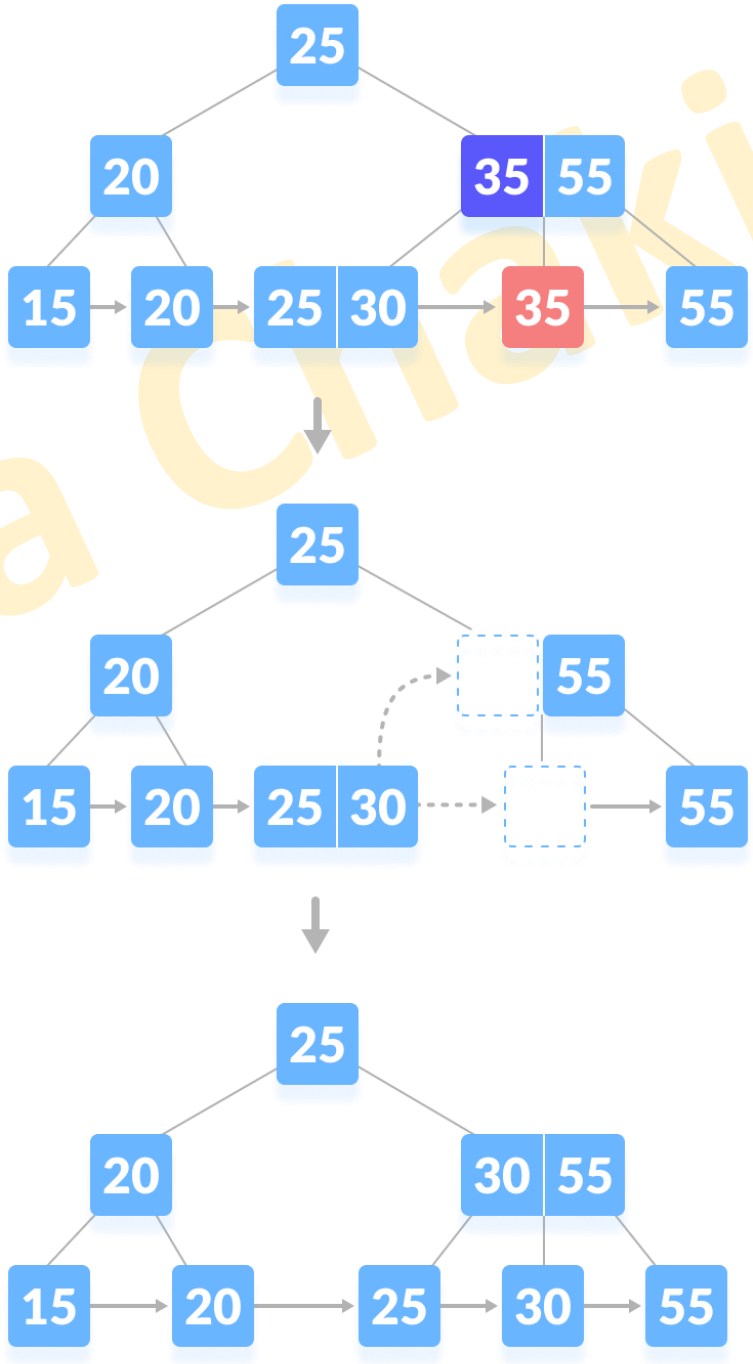
Dynamic Multilevel Indexing using B+ tree: Deletion

- **Case II:** The key to be deleted is present in the internal nodes as well. Then we have to remove them from the internal nodes as well. There are the following cases for this situation.
    1. If there is more than the minimum number of keys in the node, simply delete the key from the leaf node and delete the key from the internal node as well.
       Fill the empty space in the internal node with the inorder successor.



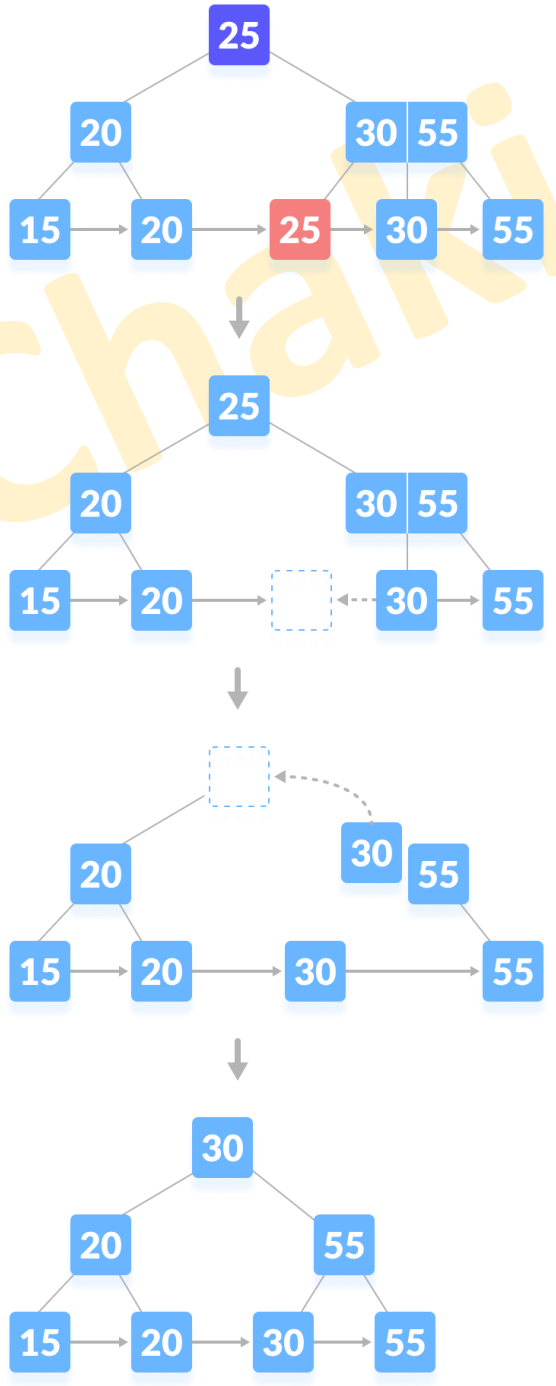
Dynamic Multilevel Indexing using B+ tree: Deletion

- **Case II:** The key to be deleted is present in the internal nodes as well. Then we have to remove them from the internal nodes as well. There are the following cases for this situation.
    2. If there is an exact minimum number of keys in the node, then delete the key and borrow a key from its immediate sibling (through the parent).
       Fill the empty space created in the index (internal node) with the borrowed key.

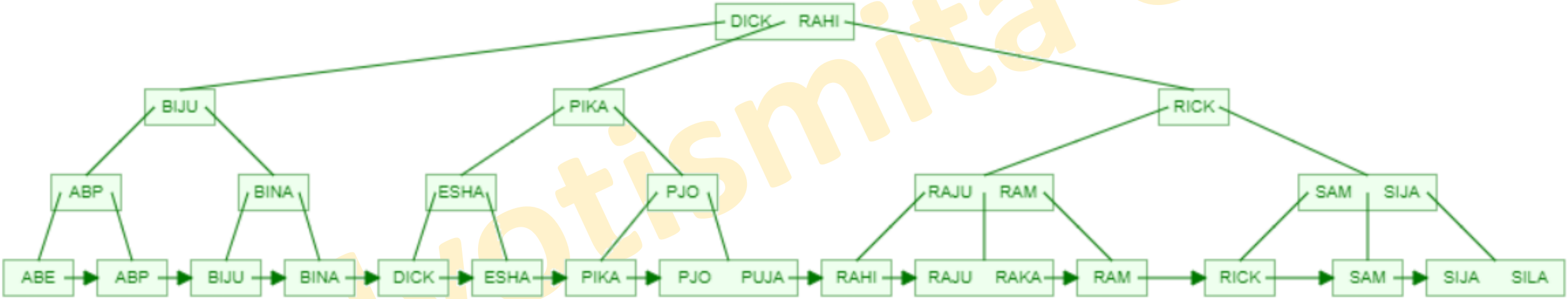


Dynamic Multilevel Indexing using B+ tree: Deletion

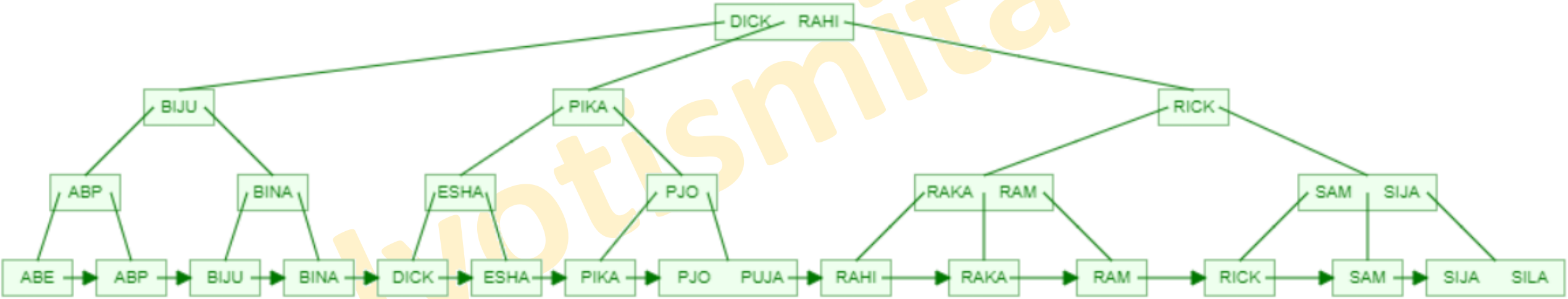
- **Case II:** The key to be deleted is present in the internal nodes as well. Then we have to remove them from the internal nodes as well. There are the following cases for this situation.
    3. This case is similar to Case II(1) but here, empty space is generated above the immediate parent node.
       After deleting the key, merge the empty space with its sibling.
       Fill the empty space in the grandparent node with the inorder successor.



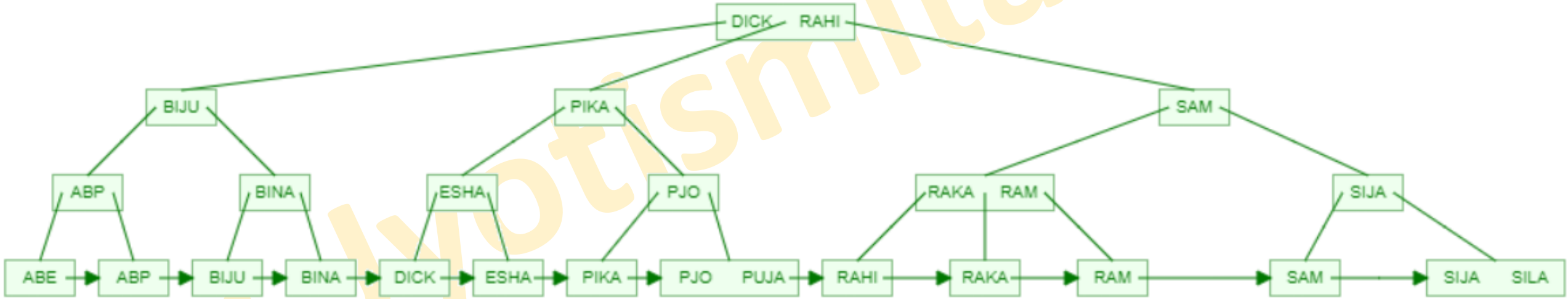
Dynamic Multilevel Indexing using B+ tree: Maximum degree = 3: Delete RINA



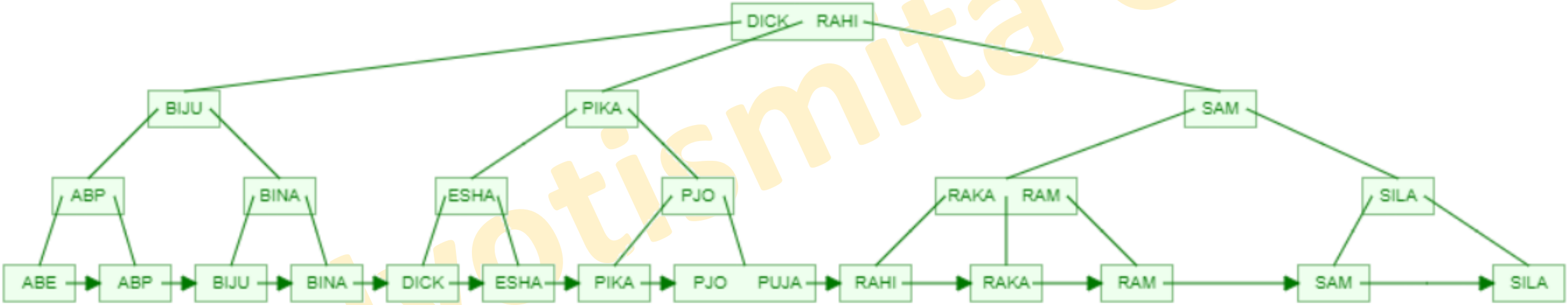
Dynamic Multilevel Indexing using B+ tree: Maximum degree = 3: Delete RAJU



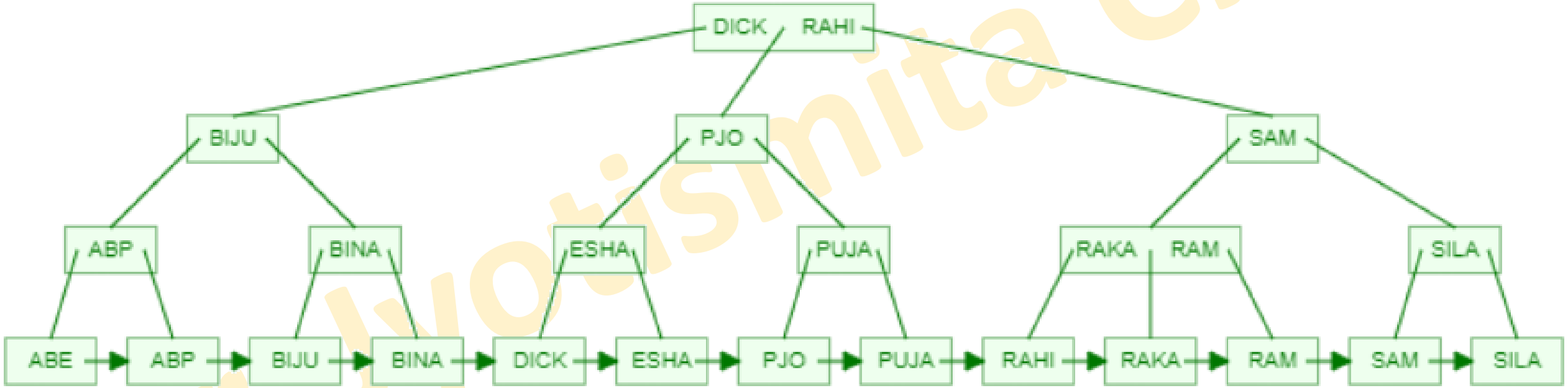
Dynamic Multilevel Indexing using B+ tree: Maximum degree = 3: Delete RICK



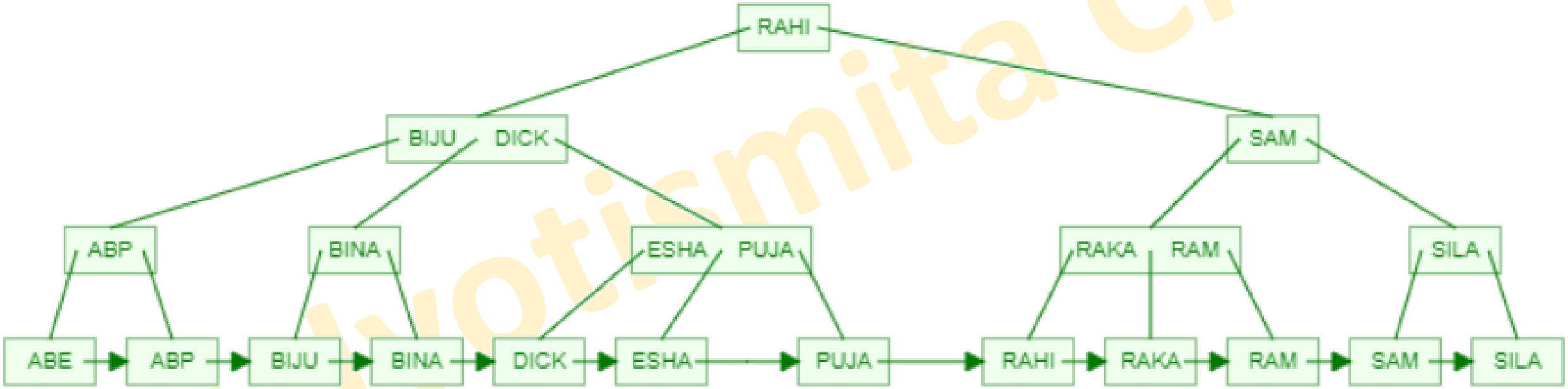
Dynamic Multilevel Indexing using B+ tree: Maximum degree = 3: Delete SIJA



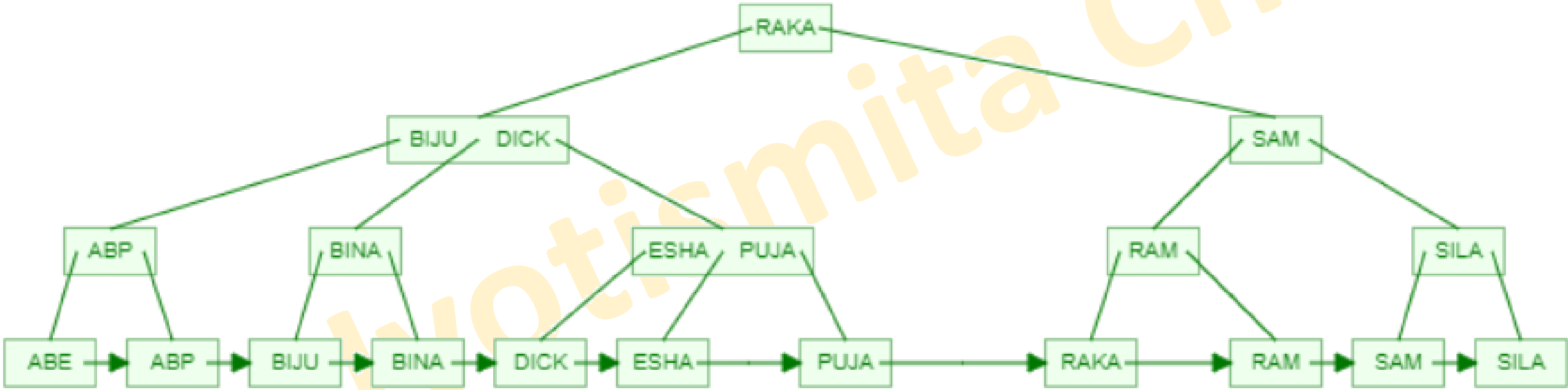
Dynamic Multilevel Indexing using B+ tree: Maximum degree = 3: Delete PIKA



Dynamic Multilevel Indexing using B+ tree: Maximum degree = 3: Delete PJO



Dynamic Multilevel Indexing using B+ tree: Maximum degree = 3: Delete RAHI



Dynamic Multilevel Indexing using B+ tree: Deletion: General Remarks

- When a node becomes underfull the algorithm will try to redistribute pointers from the neighbouring sibling either on the left or the right.
- If this is not possible, it should merge with one of them.
- The value held in an internal node or the root should always be the smallest value appearing in a leaf of the subtree pointed to by the pointer after the value.

Hashing

- Another type of primary file organization is based on **hashing**, which provides very fast access to records under certain search conditions.
- This organization is usually called a **hash file**.
- The search condition must be an equality condition on a single field, called the **hash field**.
- In most cases, the hash field is also a key field of the file, in which case it is called the **hash key**.
- The idea behind hashing is to provide a function *h*, called a **hash function** or **randomizing function**, which is applied to the hash field value of a record and yields the address of the disk block in which the record is stored.
- A search for the record within the block can be carried out in a main memory buffer.
- For most records, we need only a single-block access to retrieve that record.
- A hash file stores data in bucket format.
- Bucket is considered a unit of storage. A bucket typically stores one complete disk block, which in turn can store one or more records.

Internal hashing

- For internal files, hashing is typically implemented as a **hash table** through the use of an array of records.
- Suppose that the array index range is from 0 to M – 1, then we have M **slots** whose addresses correspond to the array indexes.
- We choose a hash function that transforms the hash field value into an integer between 0 and M – 1.
- One common hash function is the **h(K) = K mod M function**, which returns the remainder of an integer hash field value K after division by M; this value is then used for the record address.
- Noninteger hash field values can be transformed into integers before the mod function is applied.

	Name	Ssn	Job	Salary
0				
1				
2				
3				
	⋮			
M – 2				
M – 1				

Internal hashing

- For character strings, the numeric (ASCII) codes associated with characters can be used in the transformation.
- For a hash field whose data type is a string of 20 characters, following algorithm can be used to calculate the hash address.
- We assume that the code function returns the numeric code of a character and that we are given a hash field value K of type K.
  - temp ← 1;
    for i ← 1 to 20 do temp ← temp * code(K[i]) mod M ;
    hash_address ← temp mod M;

Internal hashing

- Other hashing functions can be used.
  - One technique, called **folding**, involves applying an arithmetic function such as **addition** or a **logical function** such as exclusive or to different portions of the hash field value to calculate the hash address
    - For example, with an address space from 0 to 999 to store 1,000 keys, a 6-digit key 235469 may be folded and stored at the address: (235+964) mod 1000 = 199.
  - Another technique involves picking some digits of the hash field value—for instance, the third, fifth, and eighth digits—to form the hash address
    - For example, storing 1,000 employees with Social Security numbers of 10 digits into a hash file with 1,000 positions would give the Social Security number 301-67-8923 a hash value of 172 by this hash function.
- The problem with most hashing functions is that they do not guarantee that distinct values will hash to distinct addresses

Internal hashing: Collision

- A **collision** occurs when the hash field value of a record that is being inserted hashes to an address that already contains a different record.
- Example
  - To put an item in the hash table, we compute its hash code (in this case, simply count the number of characters), then put the key and value in the arrays at the corresponding index.
  - For example, Cuba has a hash code (length) of 4. So we store Cuba in the 4th position in the keys array, and Havana in the 4th index of the values array etc.
  - But, what do we do if our dataset has a string which has more than 11 characters?
  - What if we have one another word with 5 characters, “India”, and try assigning it to an index using our hash function. Since the index 5 is already occupied, we have to make a call on what to do with it. This is called a **collision**.

Position	Keys array	Values array
1		
2		
3		
4	Cuba	Havana
5	Spain	Madrid

Internal hashing: Collision Resolution:

- In this situation, we must insert the new record in some other position, since its hash address is occupied.
- The process of finding another position is called collision resolution.
- There are numerous methods for **collision resolution**, including the following:
  - Open addressing
  - Chaining

Internal hashing: Collision Resolution: Open addressing

- Proceeding from the occupied position specified by the hash address, the program checks the subsequent positions in order until an unused (empty) position is found.
- Following algorithm may be used for this purpose (Linear probing).
  - i ← hash_address(K);
    a ← i;
    if location i is occupied
        then begin i ← (i + 1) mod M;
            while (i ≠ a) and location i is occupied
                do i ← (i + 1) mod M;
            if (i = a) then all positions are full
            else new_hash_address ← i;
        end;

Internal hashing: Collision Resolution: Open addressing: Example: Question

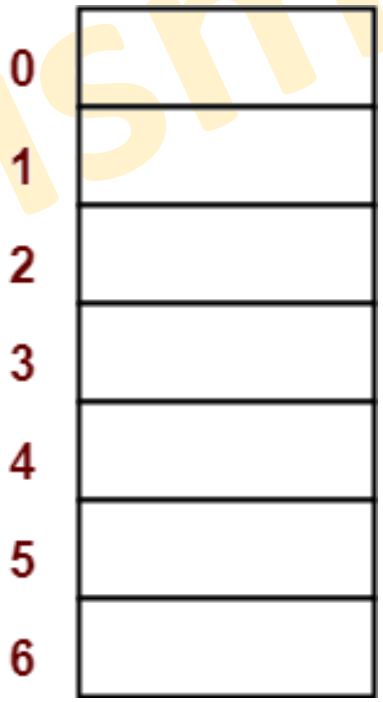
- Using the hash function ‘key mod 7’, insert the following sequence of keys in the hash table-

  50, 700, 76, 85, 92, 73 and 101

- Use linear probing open addressing to resolve the collision

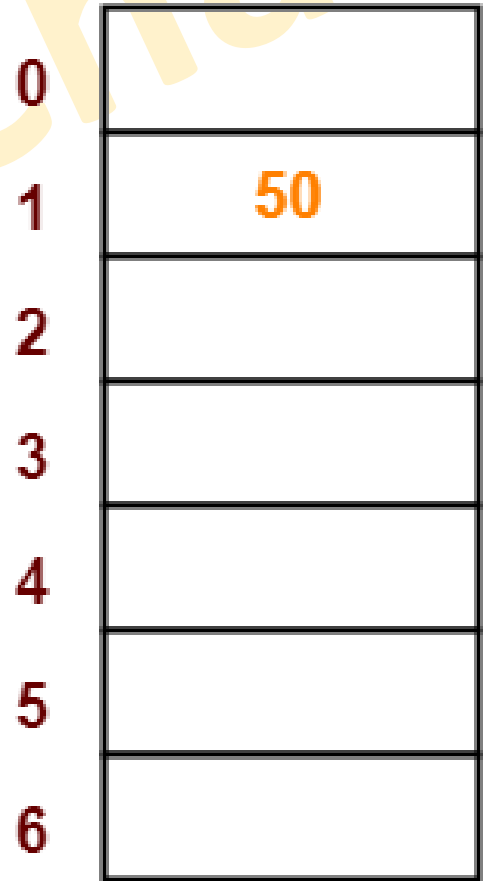
Internal hashing: Collision Resolution: Open addressing: Example: Solution: Step 1

- Draw an empty hash table.
- For the given hash function, the possible range of hash values is [0, 6].
- So, draw an empty hash table consisting of 7 buckets as-



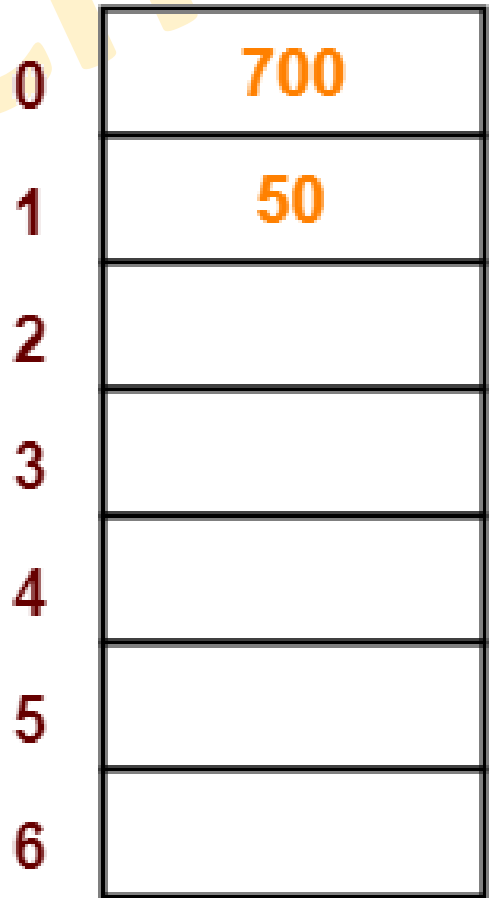
Internal hashing: Collision Resolution: Open addressing: Example: Solution: Step 2

- Insert the given keys in the hash table one by one.
- The first key to be inserted in the hash table = 50.
- Bucket of the hash table to which key 50 maps = 50 mod 7 = 1.
- So, key 50 will be inserted in bucket-1 of the hash table as-



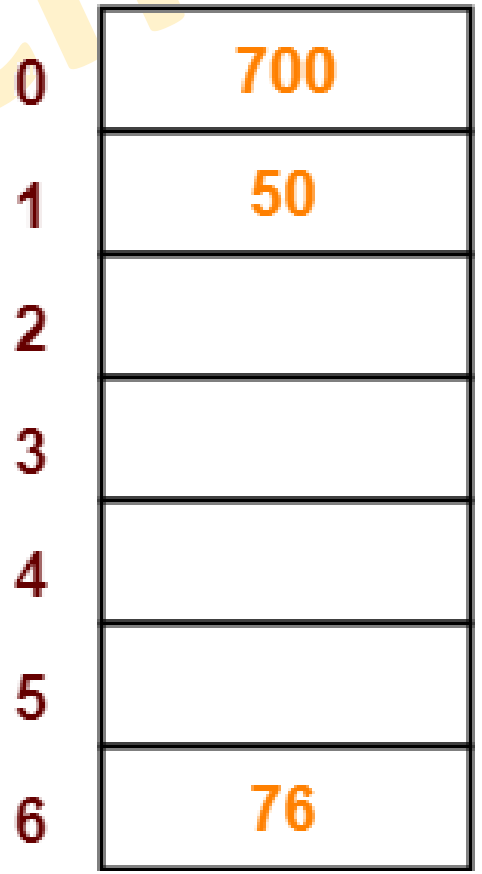
Internal hashing: Collision Resolution: Open addressing: Example: Solution: Step 3

- The next key to be inserted in the hash table = 700.
- Bucket of the hash table to which key 700 maps = 700 mod 7 = 0.
- So, key 700 will be inserted in bucket-0 of the hash table as-



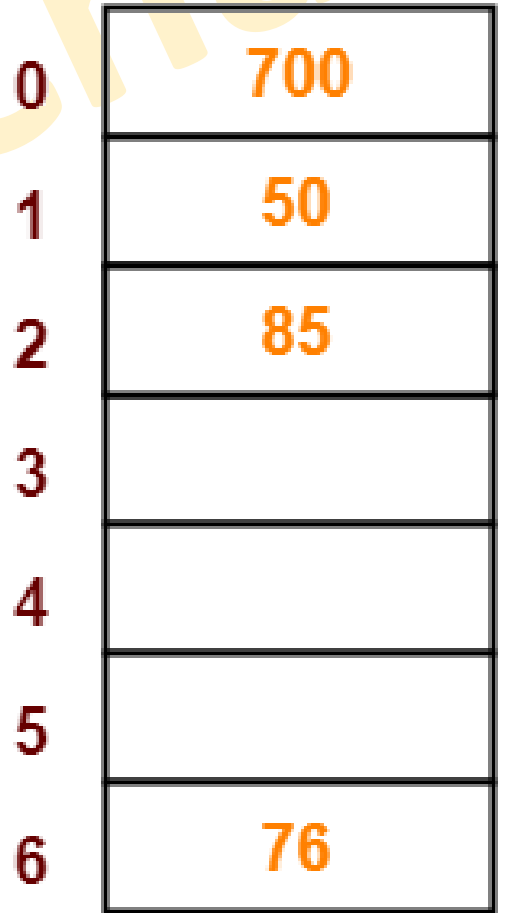
Internal hashing: Collision Resolution: Open addressing: Example: Solution: Step 4

- The next key to be inserted in the hash table = 76.
- Bucket of the hash table to which key 76 maps = 76 mod 7 = 6.
- So, key 76 will be inserted in bucket-6 of the hash table as-



Internal hashing: Collision Resolution: Open addressing: Example: Solution: Step 5

- The next key to be inserted in the hash table = 85.
- Bucket of the hash table to which key 85 maps = 85 mod 7 = 1.
- Since bucket-1 is already occupied, so collision occurs.
- To handle the collision, linear probing technique keeps probing linearly until an empty bucket is found.
- The first empty bucket is bucket-2.
- So, key 85 will be inserted in bucket-2 of the hash table as-



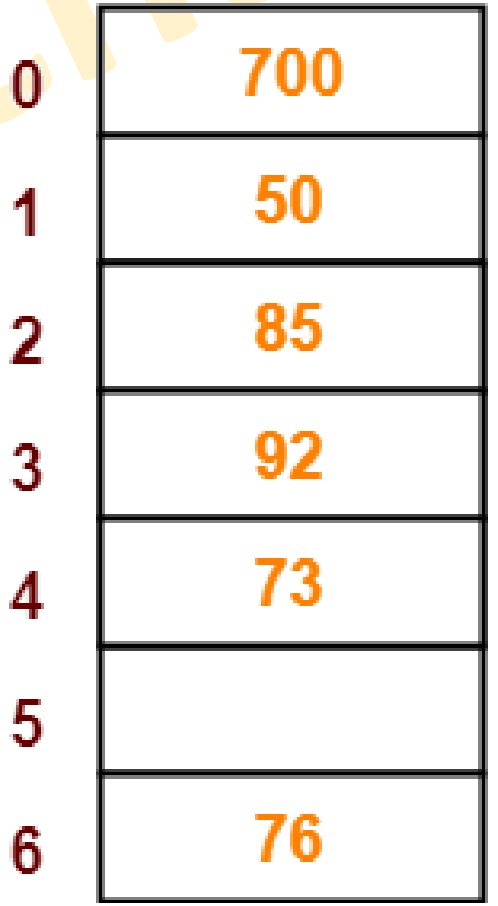
Internal hashing: Collision Resolution: Open addressing: Example: Solution: Step 6

- The next key to be inserted in the hash table = 92.
- Bucket of the hash table to which key 92 maps = 92 mod 7 = 1.
- Since bucket-1 is already occupied, so collision occurs.
- To handle the collision, linear probing technique keeps probing linearly until an empty bucket is found.
- The first empty bucket is bucket-3.
- So, key 92 will be inserted in bucket-3 of the hash table as-

0	700
1	50
2	85
3	92
4	
5	
6	76

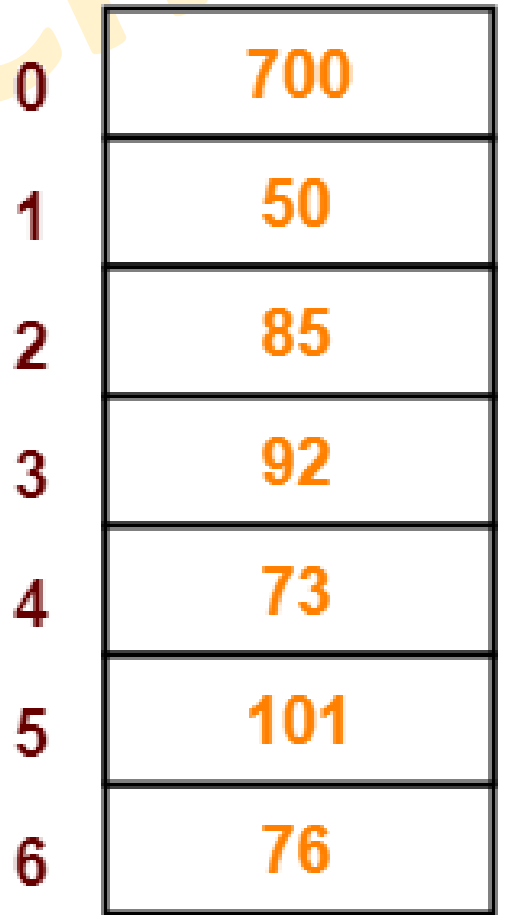
Internal hashing: Collision Resolution: Open addressing: Example: Solution: Step 7

- The next key to be inserted in the hash table = 73.
- Bucket of the hash table to which key 73 maps = 73 mod 7 = 3.
- Since bucket-3 is already occupied, so collision occurs.
- To handle the collision, linear probing technique keeps probing linearly until an empty bucket is found.
- The first empty bucket is bucket-4.
- So, key 73 will be inserted in bucket-4 of the hash table as-



Internal hashing: Collision Resolution: Open addressing: Example: Solution: Step 8

- The next key to be inserted in the hash table = 101.
- Bucket of the hash table to which key 101 maps = 101 mod 7 = 3.
- Since bucket-3 is already occupied, so collision occurs.
- To handle the collision, linear probing technique keeps probing linearly until an empty bucket is found.
- The first empty bucket is bucket-5.
- So, key 101 will be inserted in bucket-5 of the hash table as-



Internal hashing: Collision Resolution: Chaining

- For this method, various overflow locations are kept, usually by extending the array with a number of overflow positions.
- Additionally, a pointer field is added to each record location.
- A collision is resolved by placing the new record in an unused overflow location and setting the pointer of the occupied hash address location to the address of that overflow location.
- A linked list of overflow records for each hash address is thus maintained

Internal hashing: Collision Resolution: Chaining: Example: Question

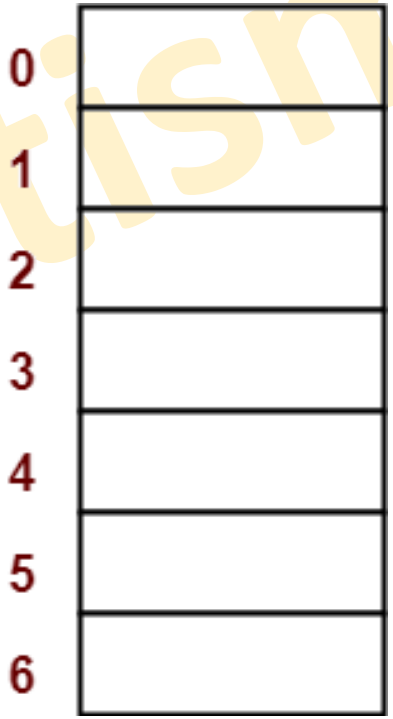
- Using the hash function ‘key mod 7’, insert the following sequence of keys in the hash table-

  50, 700, 76, 85, 92, 73 and 101

- Use separate chaining technique for collision resolution.

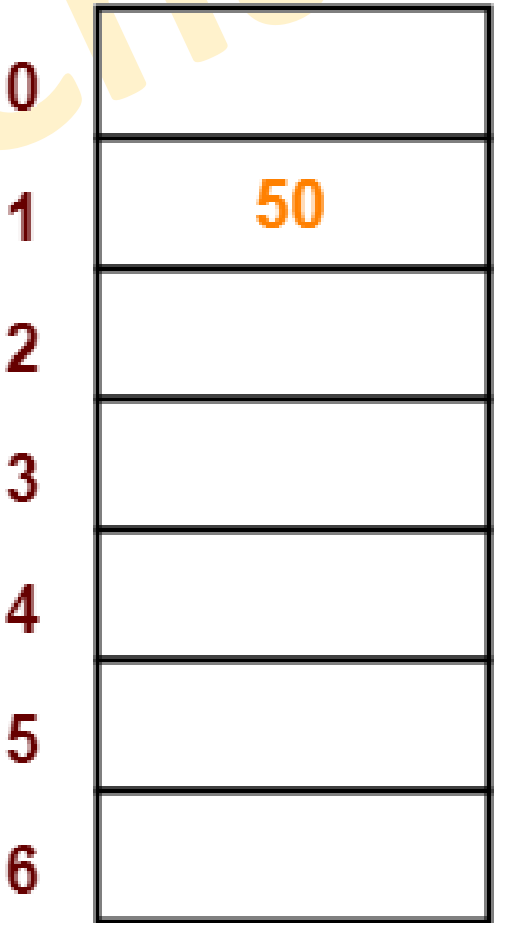
Internal hashing: Collision Resolution: Chaining: Example: Solution: Step 1

- Draw an empty hash table.
- For the given hash function, the possible range of hash values is [0, 6].
- So, draw an empty hash table consisting of 7 buckets as-



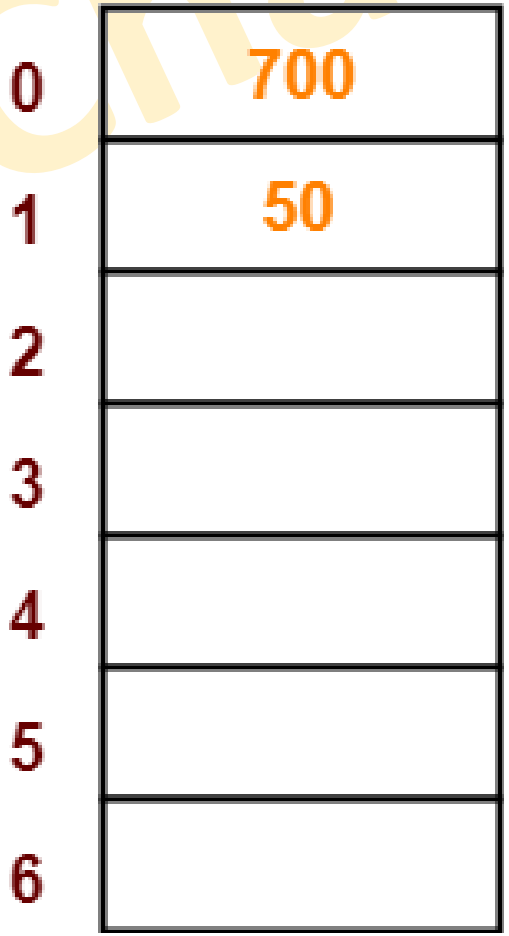
Internal hashing: Collision Resolution: Chaining: Example: Solution: Step 2

- Insert the given keys in the hash table one by one.
- The first key to be inserted in the hash table = 50.
- Bucket of the hash table to which key 50 maps = 50 mod 7 = 1.
- So, key 50 will be inserted in bucket-1 of the hash table as-



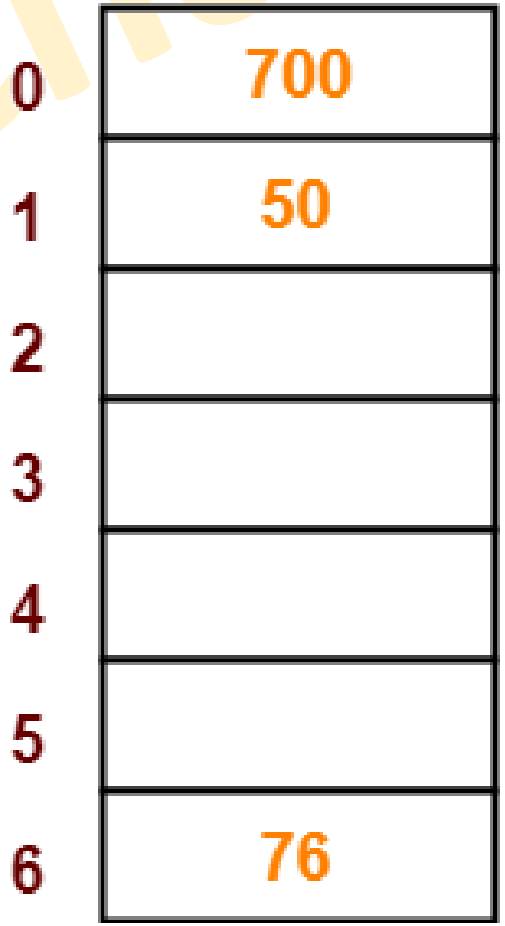
Internal hashing: Collision Resolution: Chaining: Example: Solution: Step 3

- The next key to be inserted in the hash table = 700.
- Bucket of the hash table to which key 700 maps = 700 mod 7 = 0.
- So, key 700 will be inserted in bucket-0 of the hash table as-



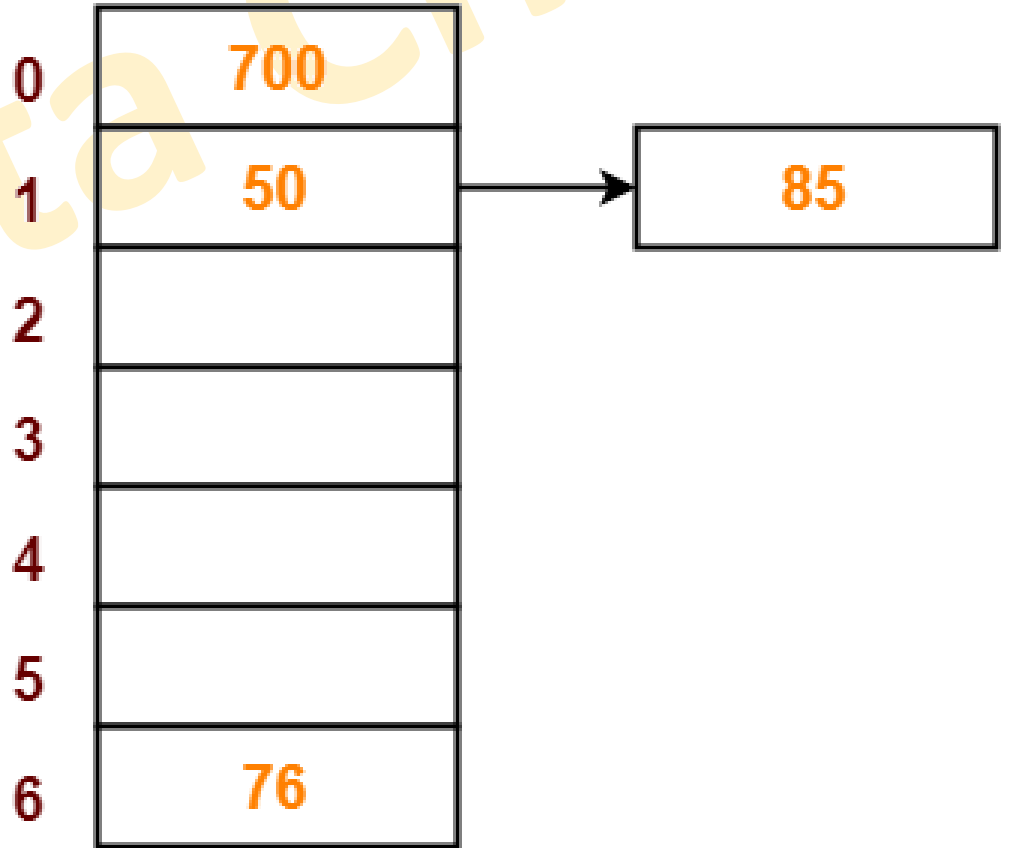
Internal hashing: Collision Resolution: Chaining: Example: Solution: Step 4

- The next key to be inserted in the hash table = 76.
- Bucket of the hash table to which key 76 maps = 76 mod 7 = 6.
- So, key 76 will be inserted in bucket-6 of the hash table as-



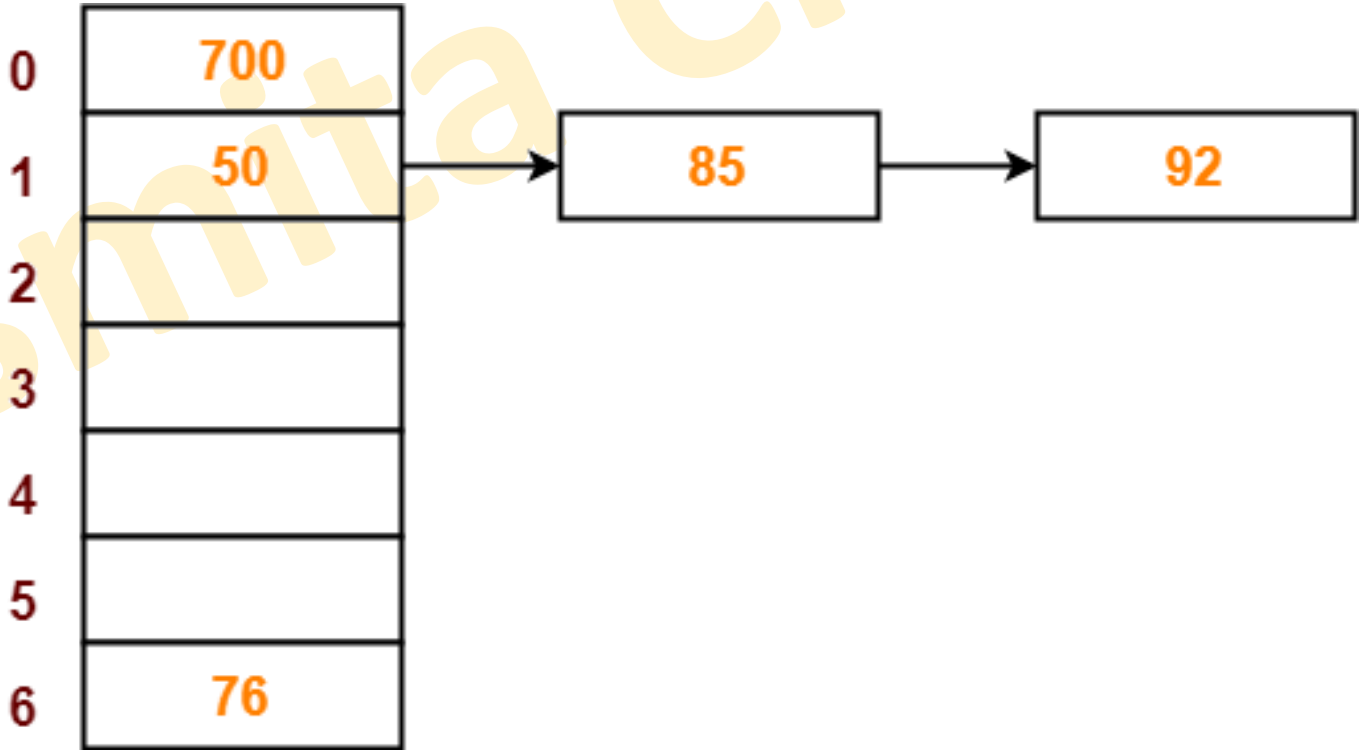
Internal hashing: Collision Resolution: Chaining: Example: Solution: Step 5

- The next key to be inserted in the hash table = 85.
- Bucket of the hash table to which key 85 maps = 85 mod 7 = 1.
- Since bucket-1 is already occupied, so collision occurs.
- Separate chaining handles the collision by creating a linked list to bucket-1.
- So, key 85 will be inserted in bucket-1 of the hash table as-



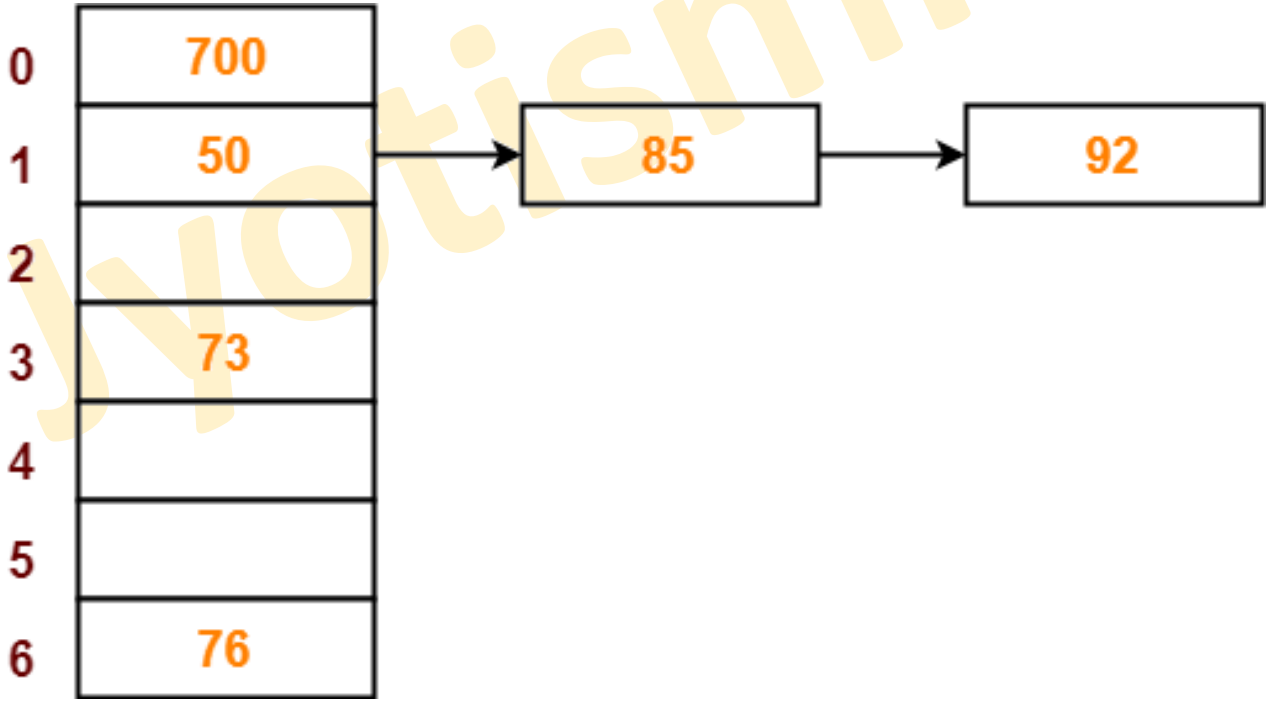
Internal hashing: Collision Resolution: Chaining: Example: Solution: Step 6

- The next key to be inserted in the hash table = 92.
- Bucket of the hash table to which key 92 maps = 92 mod 7 = 1.
- Since bucket-1 is already occupied, so collision occurs.
- Separate chaining handles the collision by creating a linked list to bucket-1.
- So, key 92 will be inserted in bucket-1 of the hash table as-



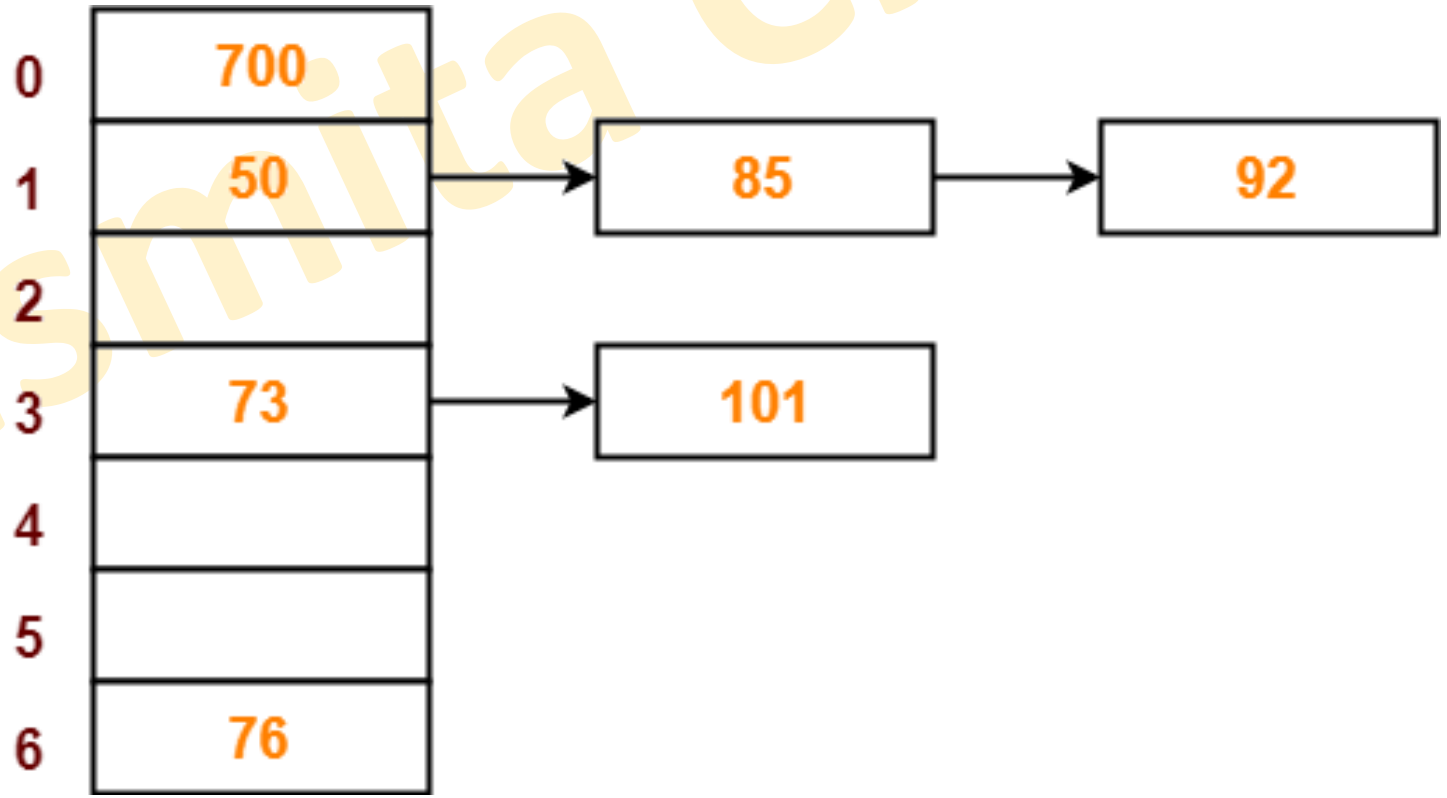
Internal hashing: Collision Resolution: Chaining: Example: Solution: Step 7

- The next key to be inserted in the hash table = 73.
- Bucket of the hash table to which key 73 maps = 73 mod 7 = 3.
- So, key 73 will be inserted in bucket-3 of the hash table as-



Internal hashing: Collision Resolution: Chaining: Example: Solution: Step 8

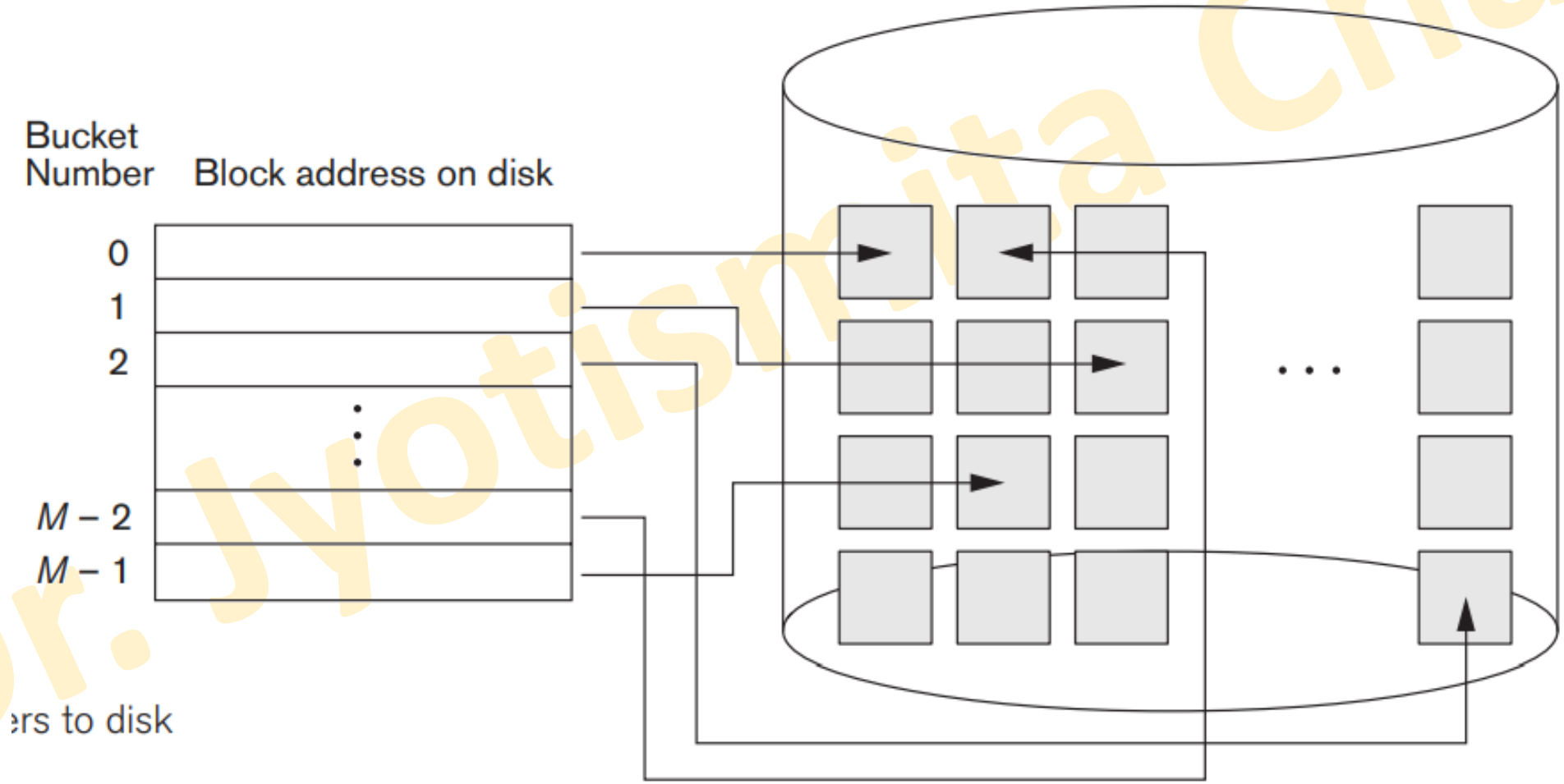
- The next key to be inserted in the hash table = 101.
- Bucket of the hash table to which key 101 maps = 101 mod 7 = 3.
- Since bucket-3 is already occupied, so collision occurs.
- Separate chaining handles the collision by creating a linked list to bucket-3.
- So, key 101 will be inserted in bucket-3 of the hash table as-



External Hashing

- Hashing for disk files is called **external hashing**.
- To suit the characteristics of disk storage, the target address space is made of **buckets**, each of which holds multiple records.
- A **bucket** is either one disk block or a cluster of contiguous disk blocks.
- The hashing function maps a **key** into a **relative bucket number** rather than assigning an absolute block address to the bucket.
- A table maintained in the file header converts the bucket number into the corresponding disk block address

External Hashing



External Hashing

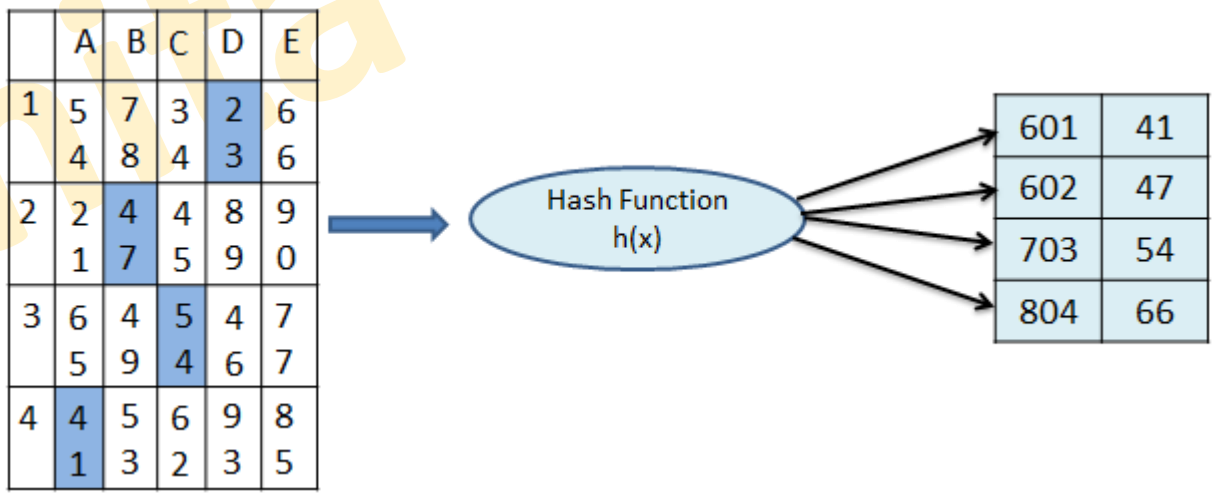
- The collision problem is less severe with buckets, because as many records as will fit in a bucket can hash to the same bucket without causing problems.
- However, we must make provisions for the case where a bucket is filled to capacity and a new record being inserted hashes to that bucket.
- We can use a variation of chaining in which a pointer is maintained in each bucket to a linked list of overflow records for the bucket.
- The pointers in the linked list should be **record pointers**, which include both a block address and a relative record position within the block.

Static Hashing

- The hashing scheme described so far is called **static hashing** because a fixed number of buckets M is allocated.
- In this type of hashing, the address of the resultant data bucket always will be the same.
- Let's take an example; if we want to generate the address for Product_ID = 76 using the mod (5) hash function, it always provides the result in the same bucket address as 3.
- In memory, the number of data buckets always remains the same or constant.

Static Hashing

- In the below image, we can see the records are placed in the A4, B2, C3, and D1 cells in the Hash table.
- These records are then passed through the Hash function h(x), in order to apply the Static Hashing technique.
- After the hash function is executed on the records in the table, the records are then placed in the proper addresses in the storage memory.
- These new addresses for the given example are 601, 602, 703, and 804, which are situated to be the outcome of static hashing process in the DBMS.



Static Hashing: Operations

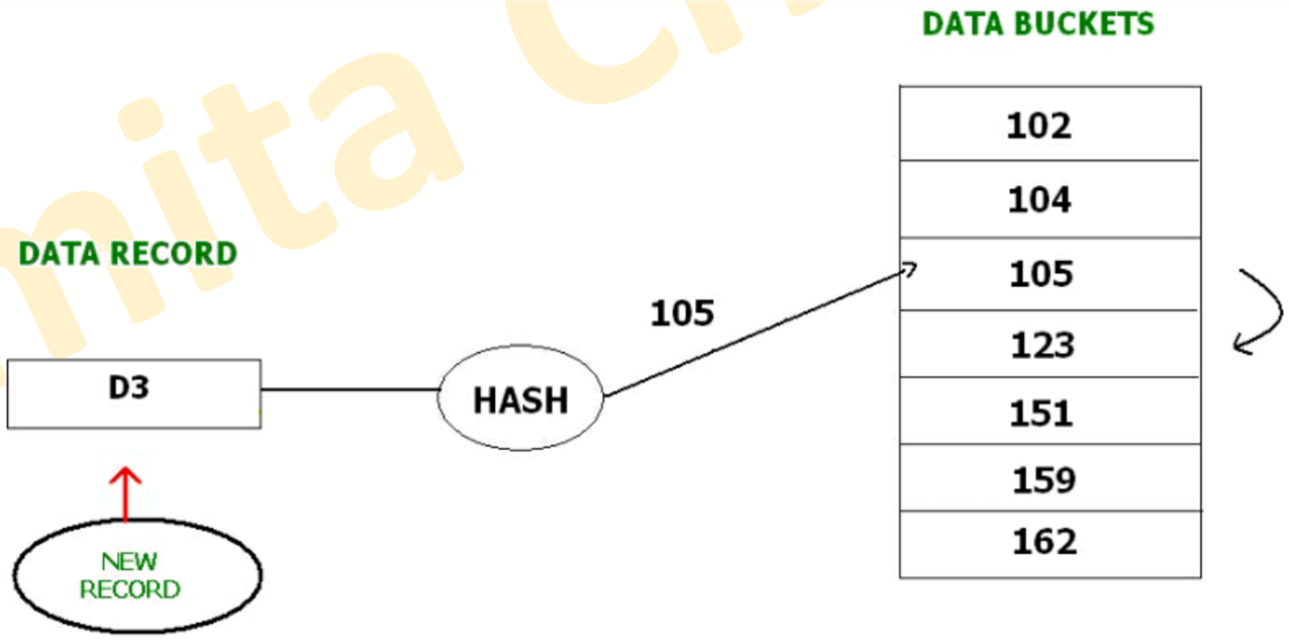
- **Inserting a record**: When a new record requires to be inserted into the table, you can generate an address for the new record using its hash key. When the address is generated, the record is automatically stored in that location.
- **Searching**: When you need to retrieve the record, the same hash 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.
- **Update a Record:** When we want to update the existing record in the table, then, by using the hash function firstly, we have to search the record, which is to be updated. Then the data record is updated automatically.

Static Hashing

- Now, If we want to insert some new records into the file But the data bucket address generated by the hash function is not empty or the data already exists in that address.
- This becomes a critical situation to handle.
- This situation in the static hashing is called **bucket overflow**.
- How will we insert data in this case?
  - There are several methods provided to overcome this situation. Some commonly used methods are:
    - Open Hashing
    - Closed Hashing

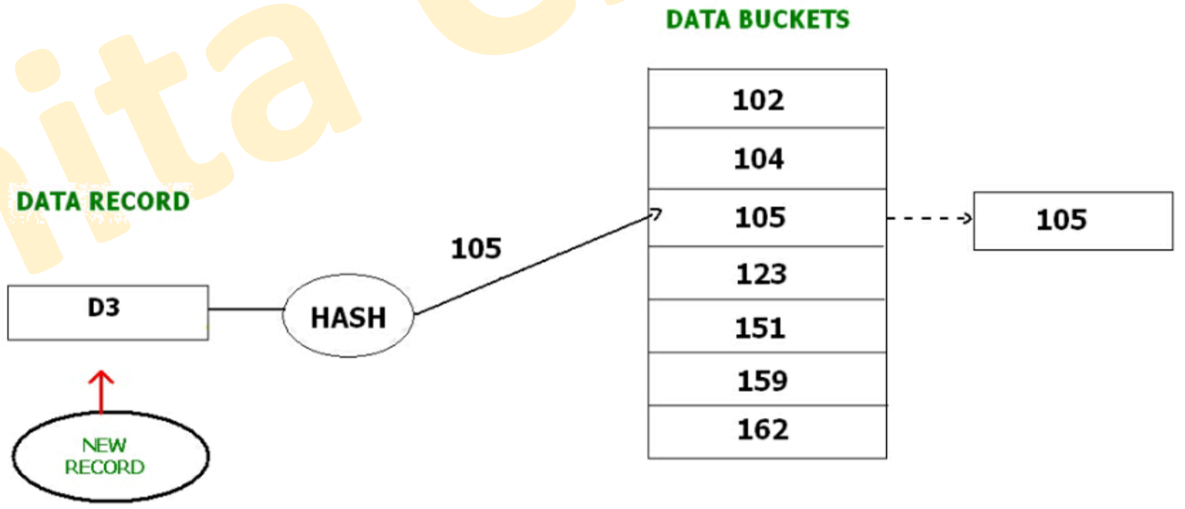
Static Hashing: Resolve Bucket Overflow

- **Open Hashing / Linear Probing**
  - In this method of hashing, the next free data block is selected for entering the new record, rather than overwriting the data to the previous block. This mechanism or method is also known as linear probing.
  - Example of open hashing: For example, D3 is a new record which needs to be inserted , the hash function generates address as 105. But it is already full. So the system searches next available data bucket, 123 and assigns D3 to it.



Static Hashing: Resolve Bucket Overflow

- **Closed Hashing / Overflow Chaining**
  - In this method of hashing, when the data bucket is filled with data, then the new bucket is selected for the same hash result and is linked with the previously filled bucket. This mechanism is called as Overflow-Chaining.
  - Example: For example, we have to insert a new record D3 into the tables. The static hash function generates the data bucket address as 105. But this bucket is full to store the new data. In this case is a new data bucket is added at the end of 105 data bucket and is linked to it. Then new record D3 is inserted into the new bucket.
  - It can add any number of new data buckets, when it is full.



Dynamic Hashing

- A major drawback of the static hashing scheme just discussed is that the hash address space is fixed. Hence, it is difficult to expand or shrink the file dynamically.
- In Dynamic hashing, data buckets grows or shrinks (added or removed dynamically) as the records increases or decreases.
- Key terms:
  - **Directories:** These containers store pointers to buckets. Each directory is given a unique id which may change each time when expansion takes place.
  - **Buckets:** They store the hashed keys. Directories point to buckets. A bucket may contain more than one pointers to it if its local depth is less than the global depth.

Dynamic Hashing

- Key terms:
  - **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.
  - **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: Workflow

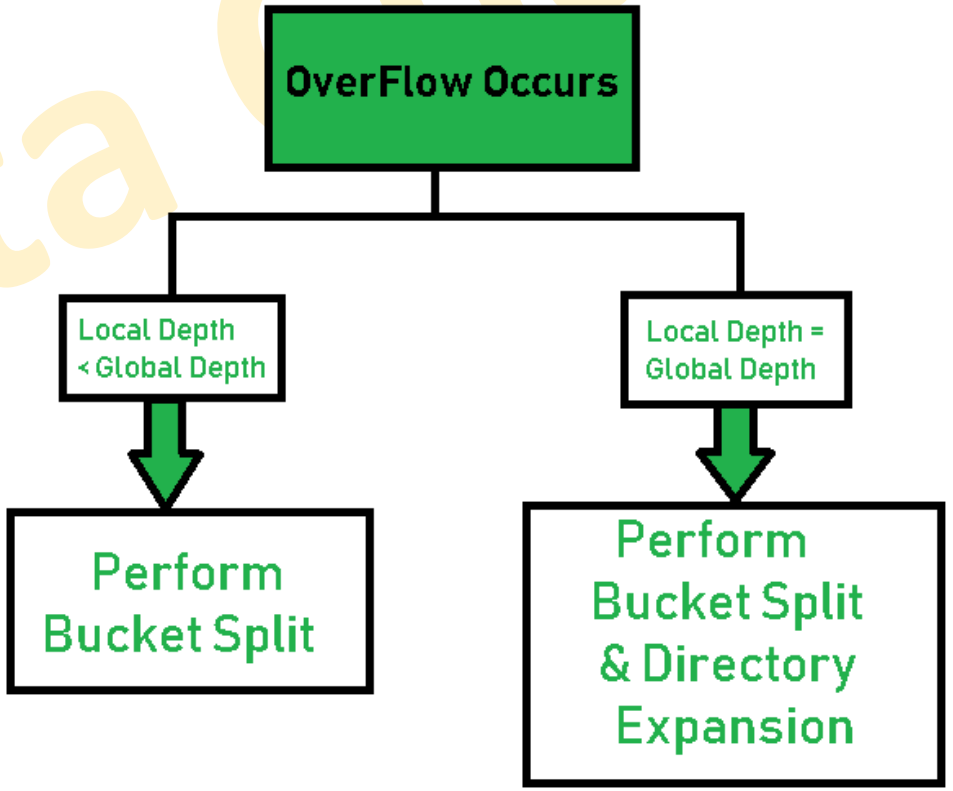
- **Step 1 – Analyze Data Elements:** Data elements may exist in various forms eg. Integer, String, Float, etc.. Currently, let us consider data elements of type integer. eg: 49.
- **Step 2 – Convert into binary format:** Convert the data element in Binary form. For string elements, consider the ASCII equivalent integer of the starting character and then convert the integer into binary form. Since we have 49 as our data element, its binary form is 110001.
- **Step 3 – Check Global Depth of the directory.** Suppose the global depth of the Hash-directory is 3.

Dynamic Hashing: Workflow

- **Step 4 – Identify the Directory:** Consider the 'Global-Depth' number of LSBs in the binary number and match it to the directory id. Eg. The binary obtained is: 110001 and the global-depth is 3. So, the hash function will return 3 LSBs of 110**001** viz. 001.
- **Step 5 – Navigation:** Now, navigate to the bucket pointed by the directory with directory-id 001.
- **Step 6 – Insertion and Overflow Check:** Insert the element and check if the bucket overflows. If an overflow is encountered, go to **step 7** followed by **Step 8**, otherwise, go to **step 9**.

Dynamic Hashing: Workflow

- **Step 7 – Tackling Over Flow Condition during Data Insertion:** Many times, while inserting data in the buckets, it might happen that the Bucket overflows. In such cases, we need to follow an appropriate procedure to avoid mishandling of data. First, Check if the local depth is less than or equal to the global depth. Then choose one of the cases below.
  - **Case1:** If the local depth of the overflowing Bucket is equal to the global depth, then Directory Expansion, as well as Bucket Split, needs to be performed. Then increment the global depth and the local depth value by 1. And, assign appropriate pointers.
    Directory expansion will double the number of directories present in the hash structure.
  - **Case2:** In case the local depth is less than the global depth, then only Bucket Split takes place. Then increment only the local depth value by 1. And, assign appropriate pointers.



Dynamic Hashing: Workflow

- **Step 8 – Rehashing of Split Bucket Elements:** The Elements present in the overflowing bucket that is split are rehashed w.r.t the new global depth of the directory.
- **Step 9 –** The element is successfully hashed.

Dynamic Hashing: Insertion Example

- Now, let us consider a example of hashing the following elements: **16,4,6,22,24,10,31,7,9,20,26.**

Solution

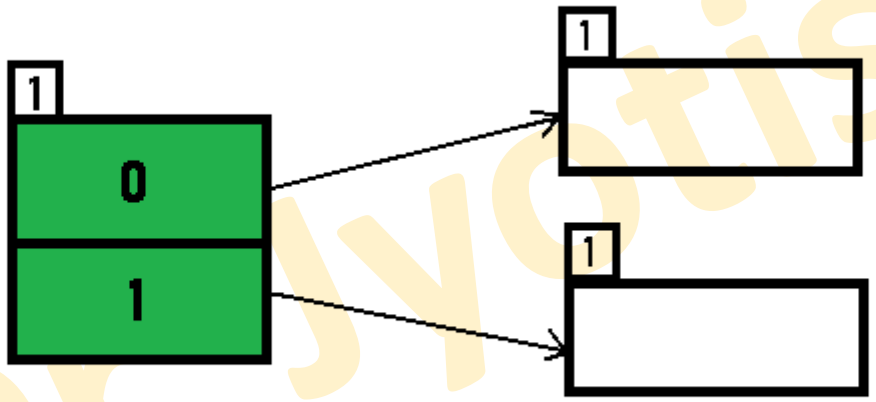
- **Bucket Size:** 3 (Assume)
- **Hash Function:** Suppose the global depth is X. Then the Hash Function returns X LSBs.

Dynamic Hashing: Insertion Example

- 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- 11010

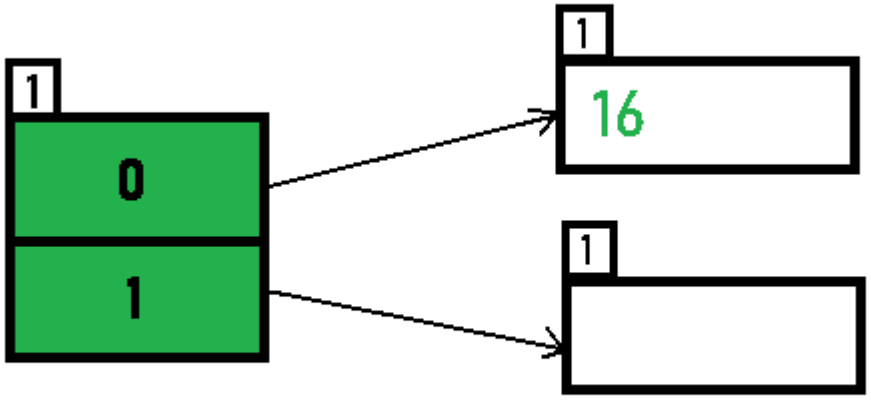
Dynamic Hashing: Insertion Example

- Initially, the global-depth and local-depth is always 1. Thus, the hashing frame looks like this:



- **Inserting 16:**

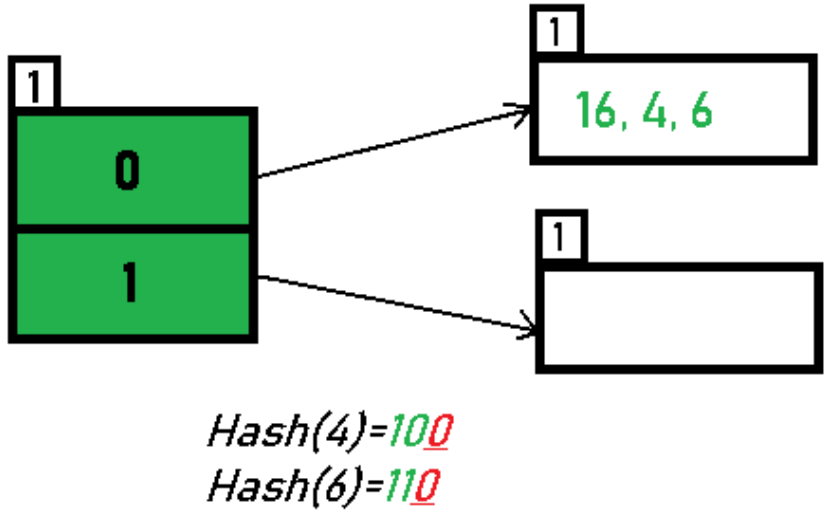
The binary format of 16 is 10000 and global-depth is 1. The hash function returns 1 LSB of 10000 which is 0. Hence, 16 is mapped to the directory with id=0.



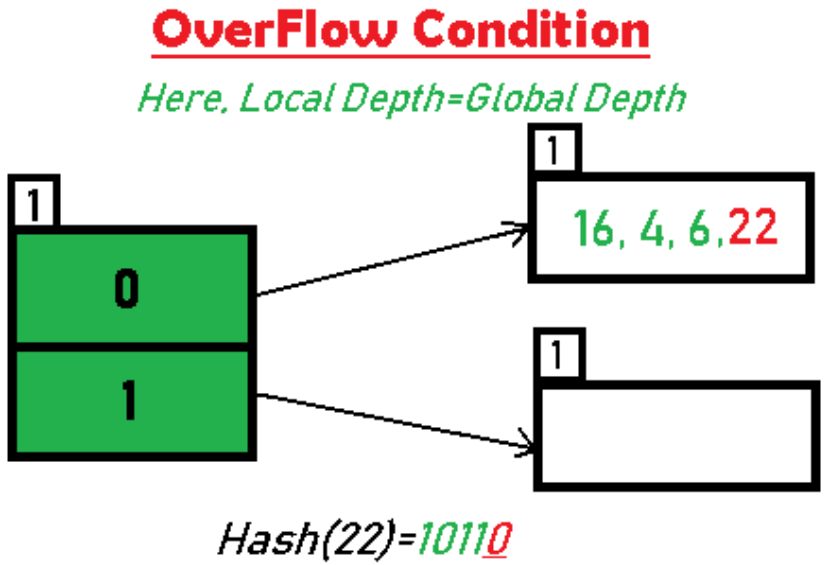
Hash(16)= 10000

Dynamic Hashing: Insertion Example

Inserting 4 and 6:
Both 4(10**0**) and 6(11**0**)have 0 in their LSB. Hence, they are hashed as follows:

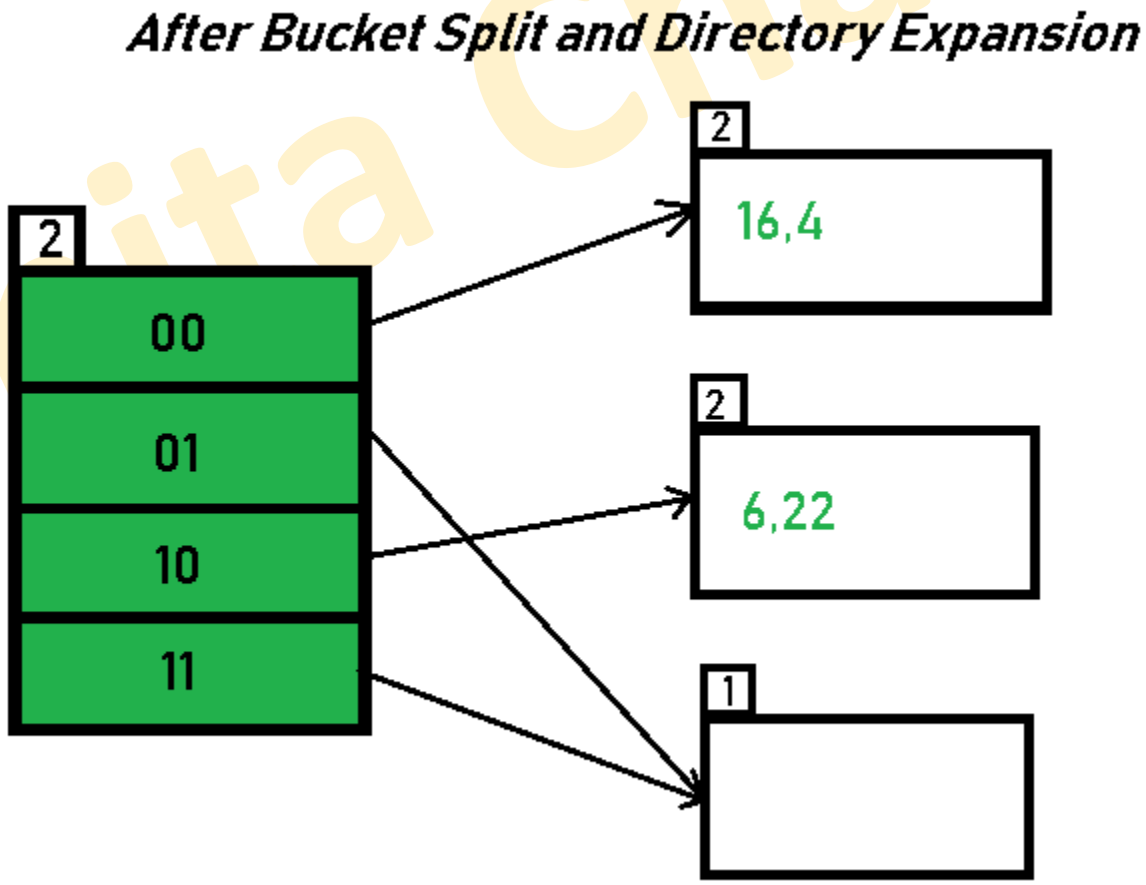


Inserting 22: The binary form of 22 is 1011**0**. Its LSB is 0. The bucket pointed by directory 0 is already full. Hence, Over Flow occurs.



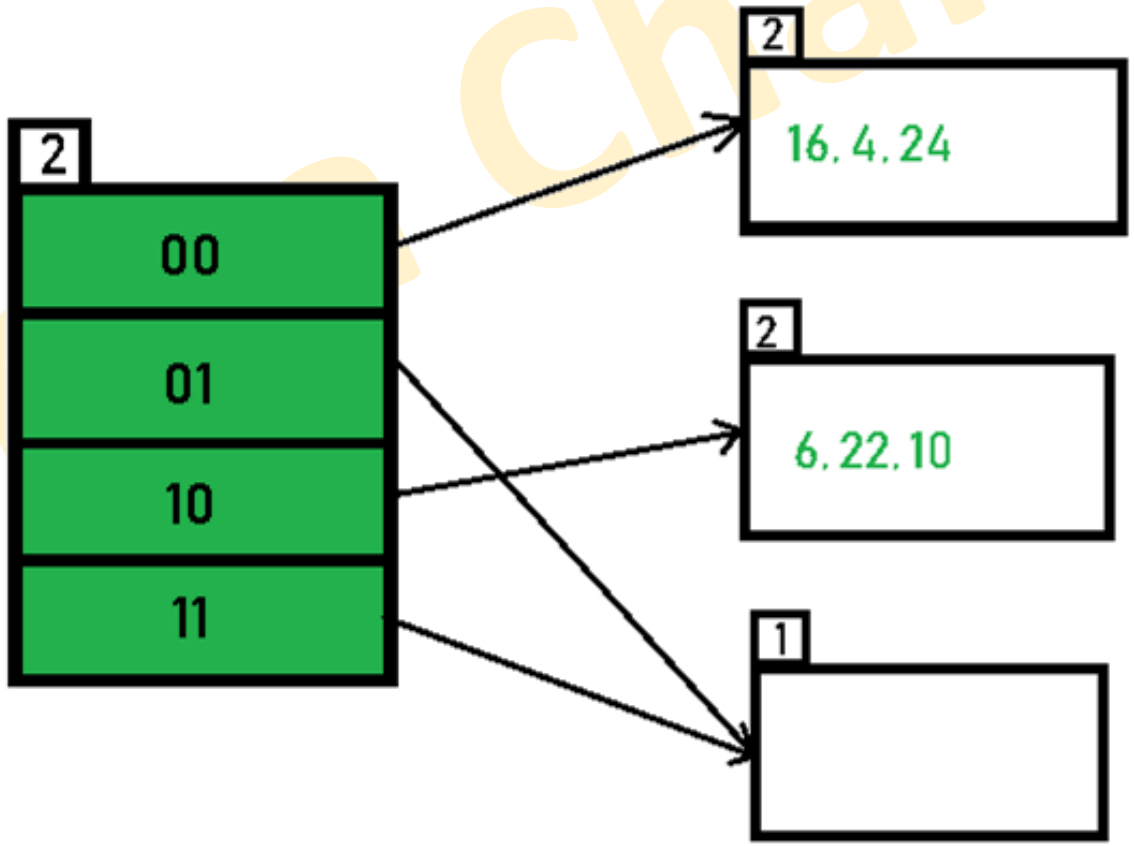
Dynamic Hashing: Insertion Example

- As directed by **Step 7-Case 1**, 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 (100**00**), 4(1**00**), 6(1**10**), 22(101**10**)]
- *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.*



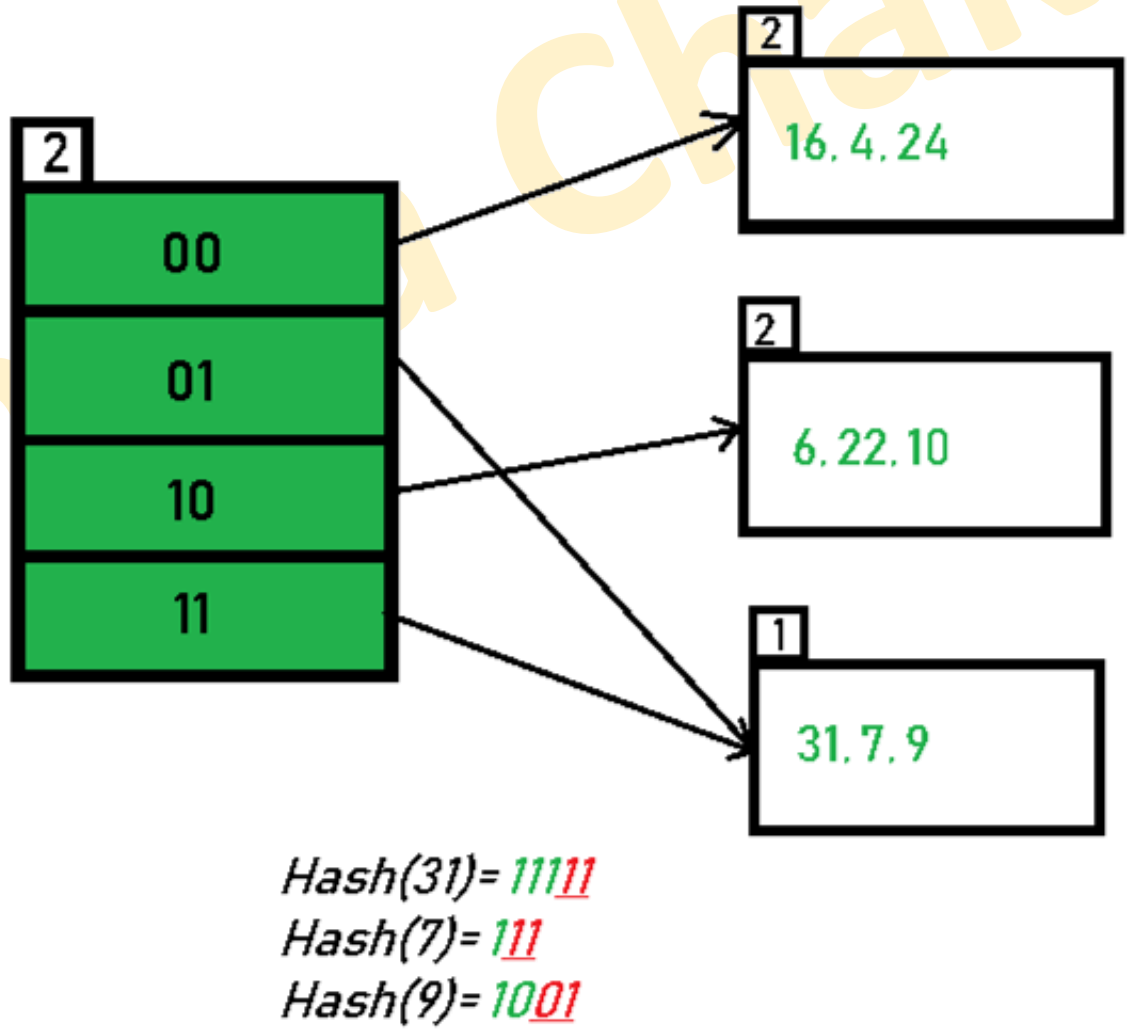
Dynamic Hashing: Insertion Example

- **Inserting 24 and 10:** 24(110**00**) and 10 (10**10**) can be hashed based on directories with id 00 and 10. Here, we encounter no overflow condition.



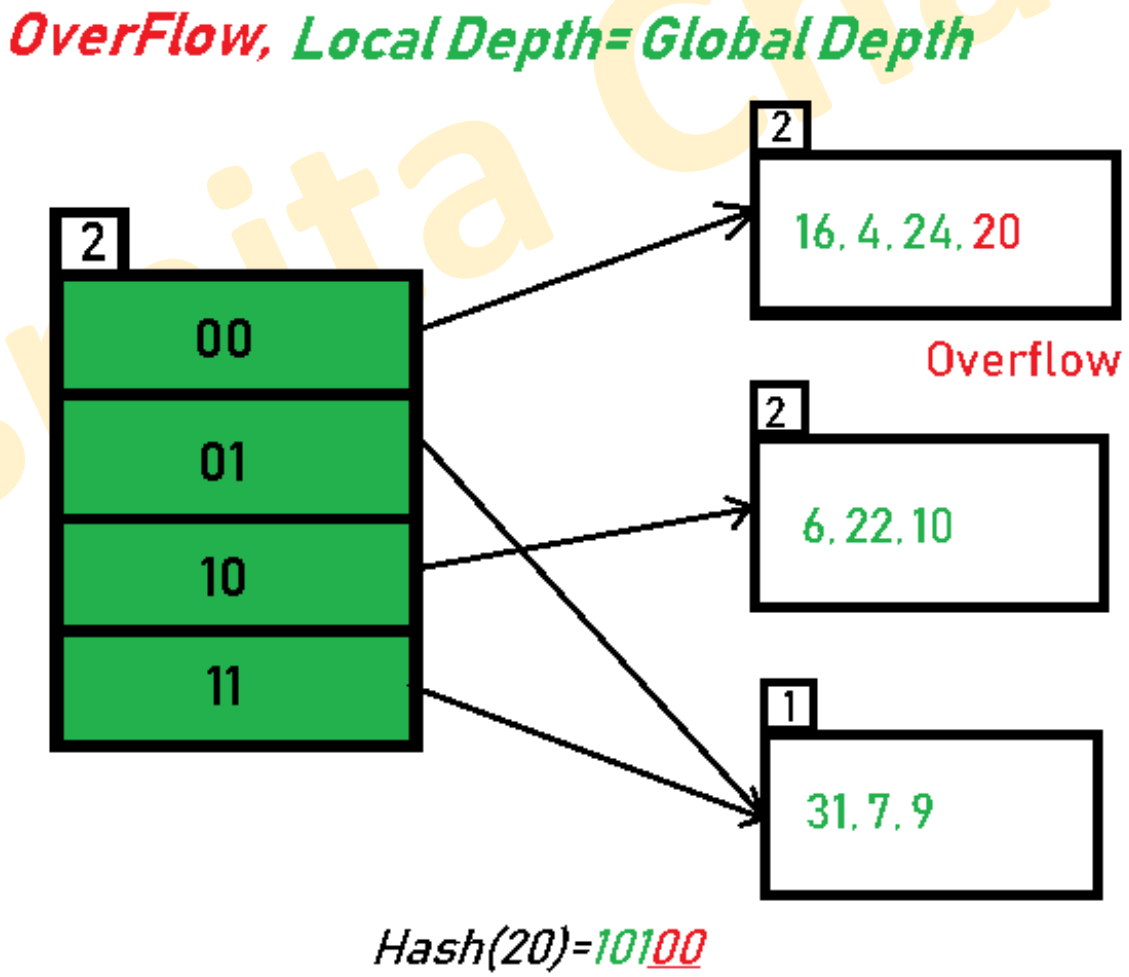
Dynamic Hashing: Insertion Example

- **Inserting 31,7,9:** All of these elements[31(111**11**), 7(1**11**), 9(10**01**)] 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.



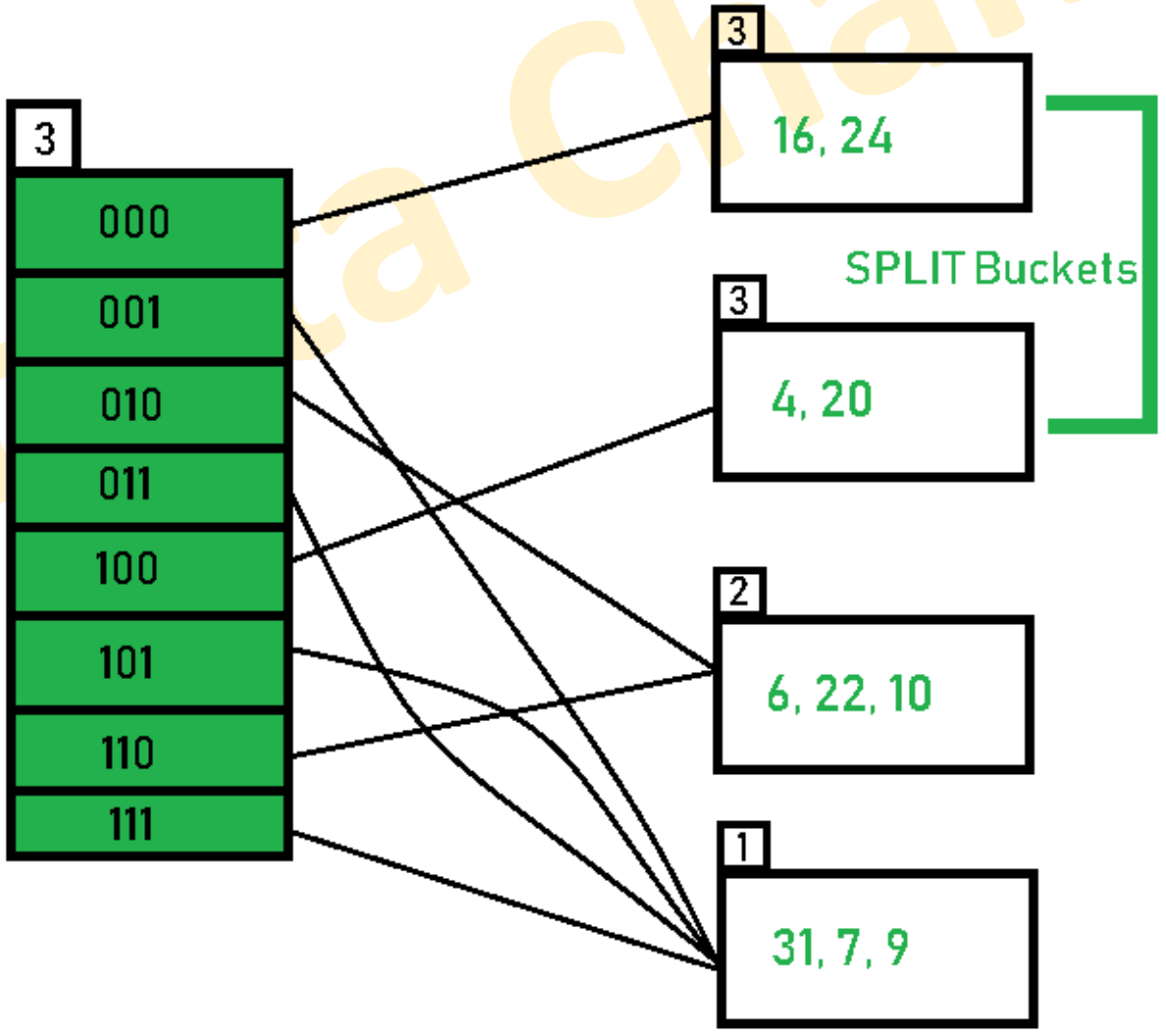
Dynamic Hashing: Insertion Example

- **Inserting 20:** Insertion of data element 20 (101**00**) will again cause the overflow problem.



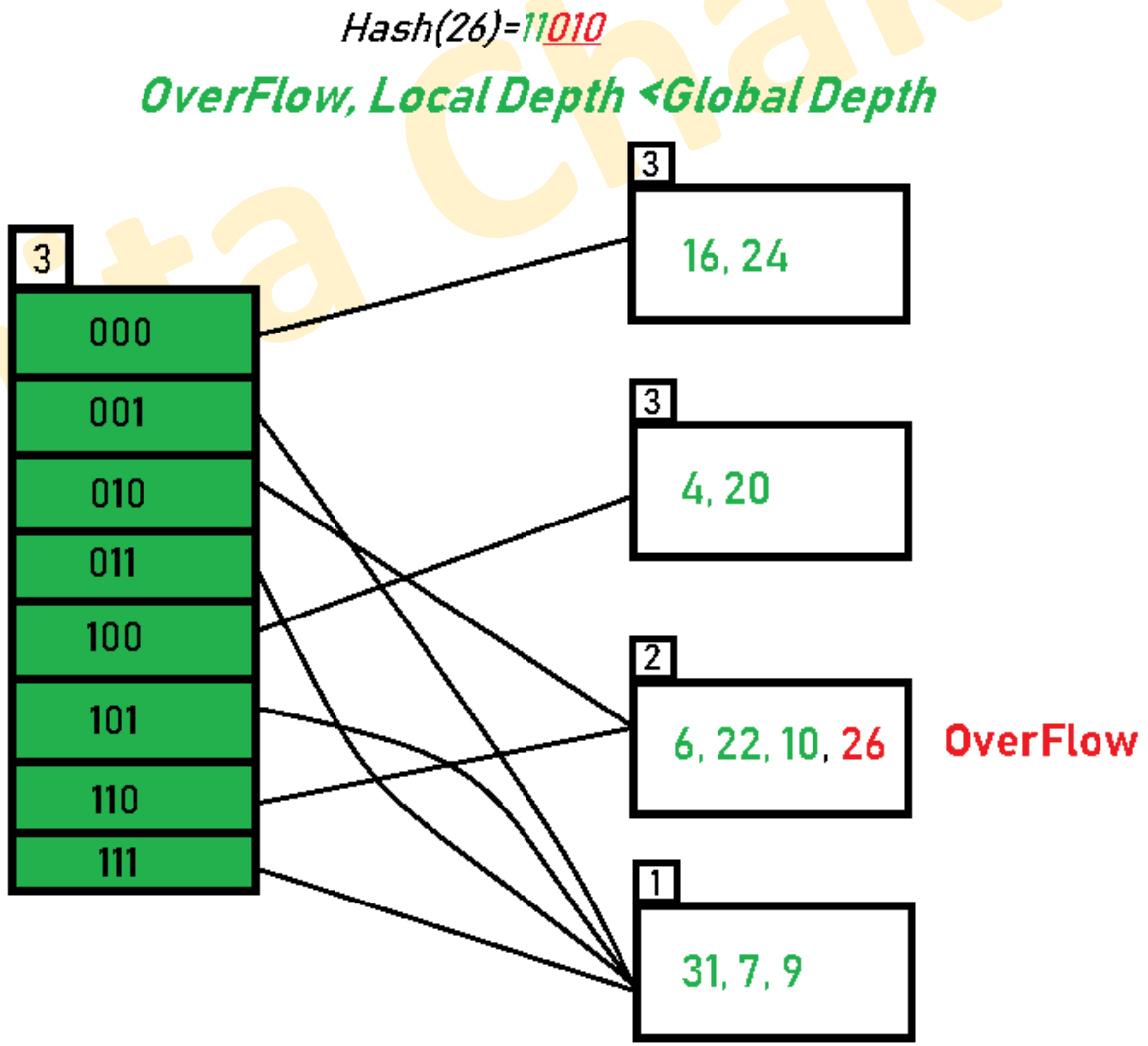
Dynamic Hashing: Insertion Example

- 20 is inserted in bucket pointed out by 00. As directed by **Step 7-Case 1**, since the **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:



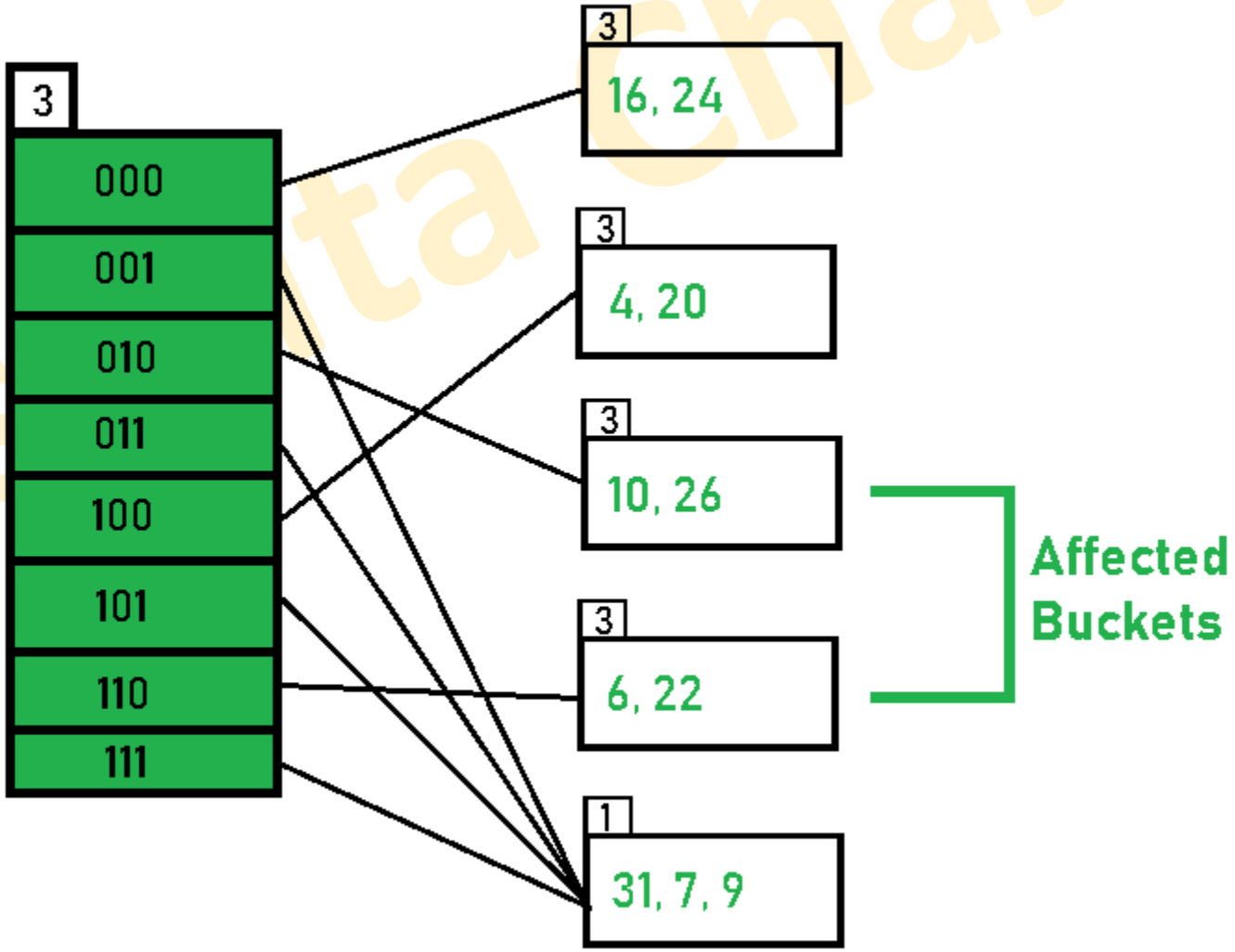
Dynamic Hashing: Insertion Example

- **Inserting 26:** Global depth is 3. Hence, 3 LSBs of 26(11**010**) are considered. Therefore 26 best fits in the bucket pointed out by directory 010.



Dynamic Hashing: Insertion Example

- The bucket overflows, and, as directed by **Step 7-Case 2,** since the **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:

Advantages

1. Data retrieval is less expensive (in terms of computing).
2. No problem of Data-loss since the storage capacity increases dynamically.
3. With dynamic changes in hashing function, associated old values are rehashed w.r.t the new hash function.

Limitations

1. The directory size may increase significantly if several records are hashed on the same directory while keeping the record distribution non-uniform.
2. Size of every bucket is fixed.
3. Memory is wasted in pointers when the global depth and local depth difference becomes drastic.
4. This method is complicated to code.