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

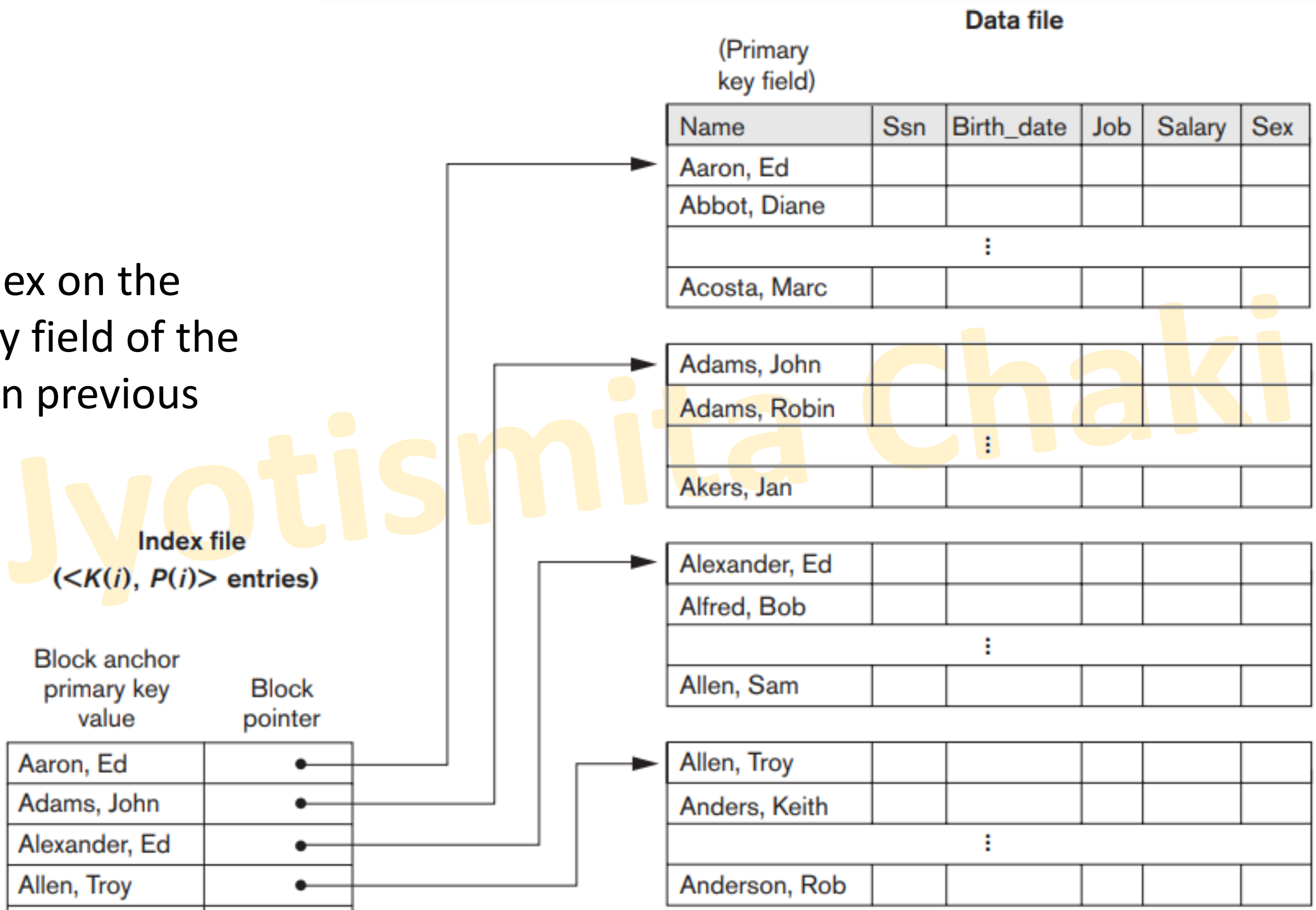
- A primary index is an ordered file whose records are of fixed length with two fields.
- It acts like an access structure to efficiently search for and access the data records in a data file.
- The first field is of the same data type as the ordering key field—called the primary key—of the data file, 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

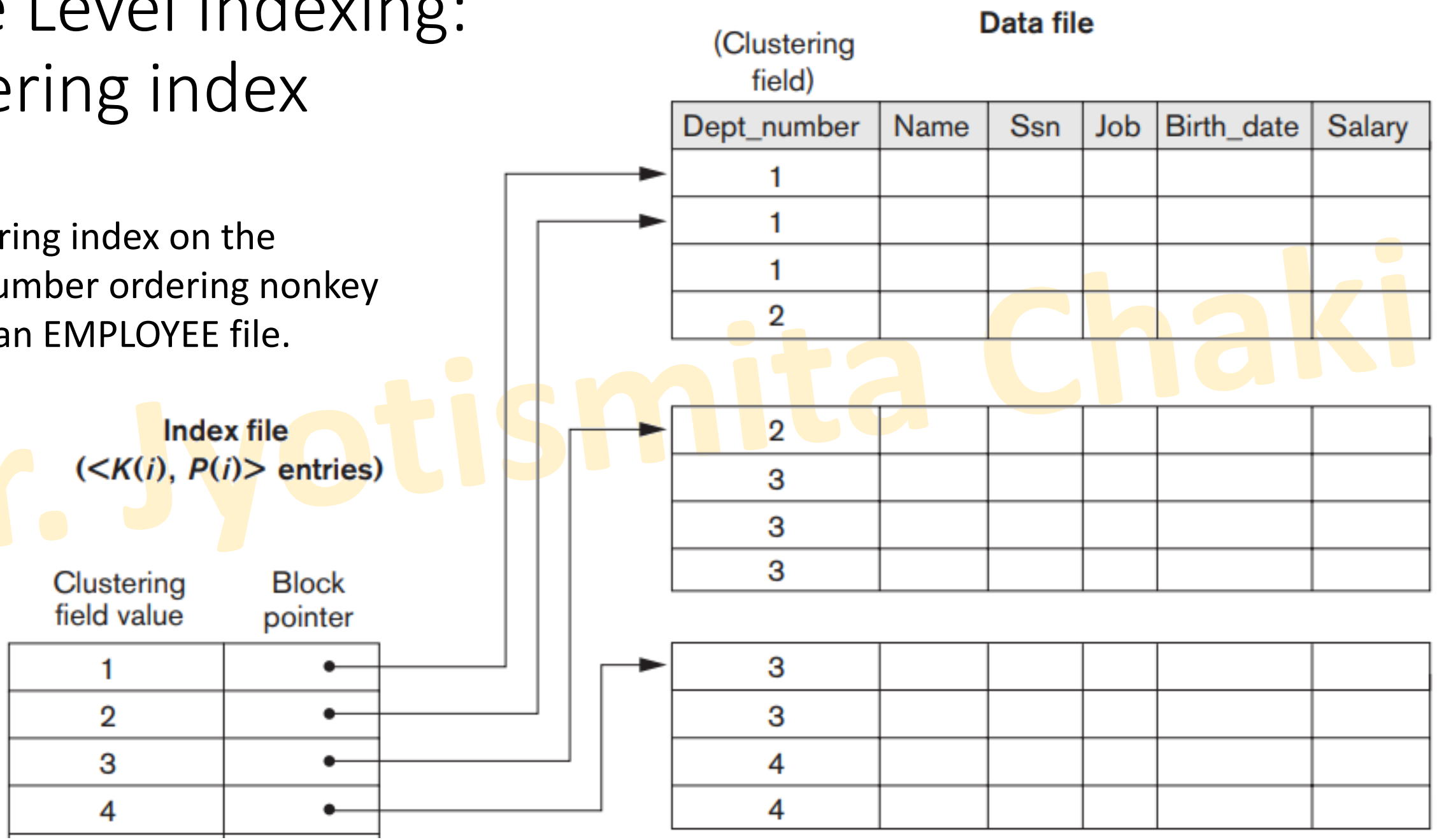
- If file records are physically ordered on a nonkey field—which does not have a distinct value for each record—that field is called the **clustering field** and the data file is called a **clustered file**.
- We can create a different type of index, called a **clustering index**, to speed up retrieval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

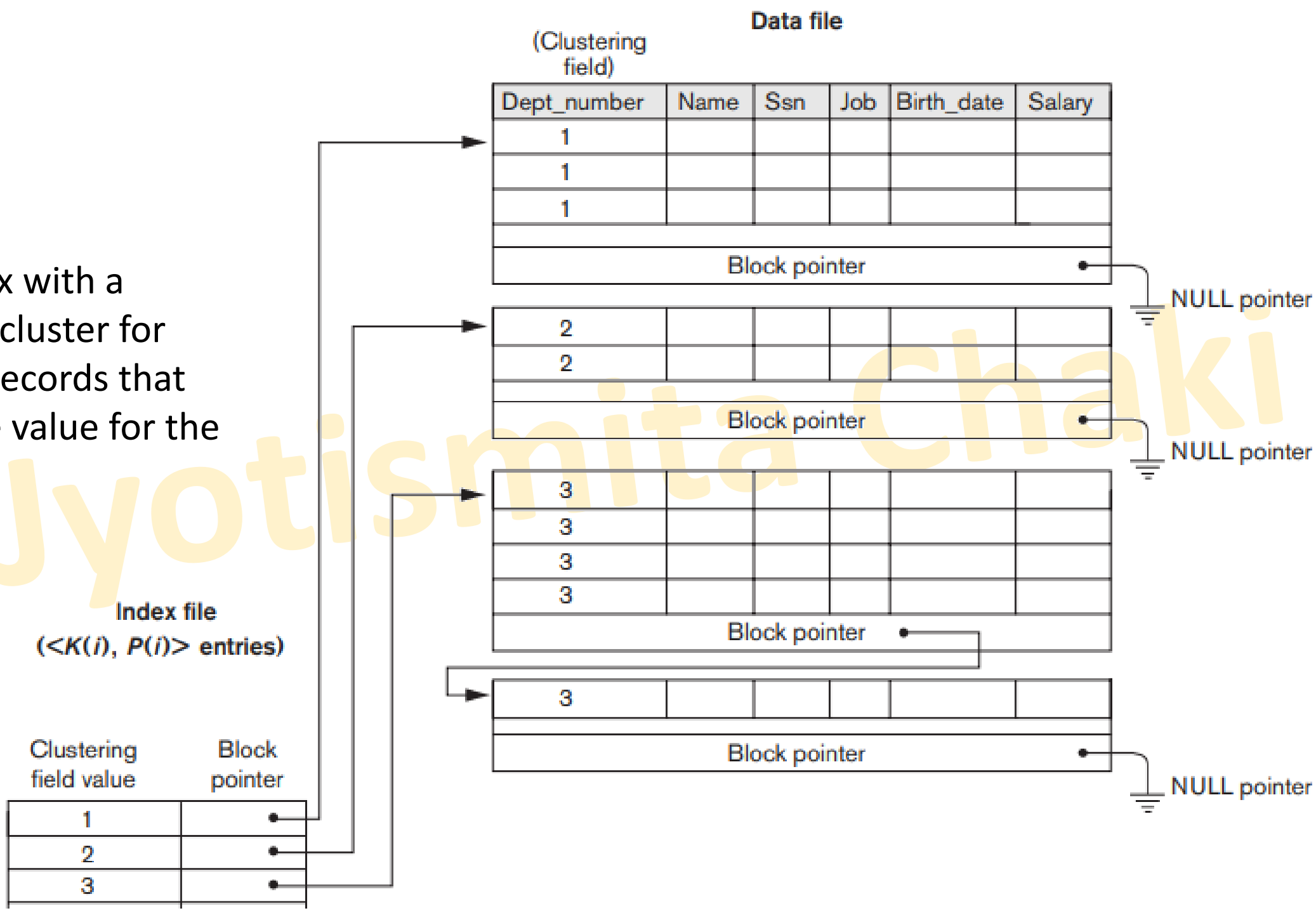
A clustering index on the Dept_number ordering nonkey field of an EMPLOYEE file.



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.

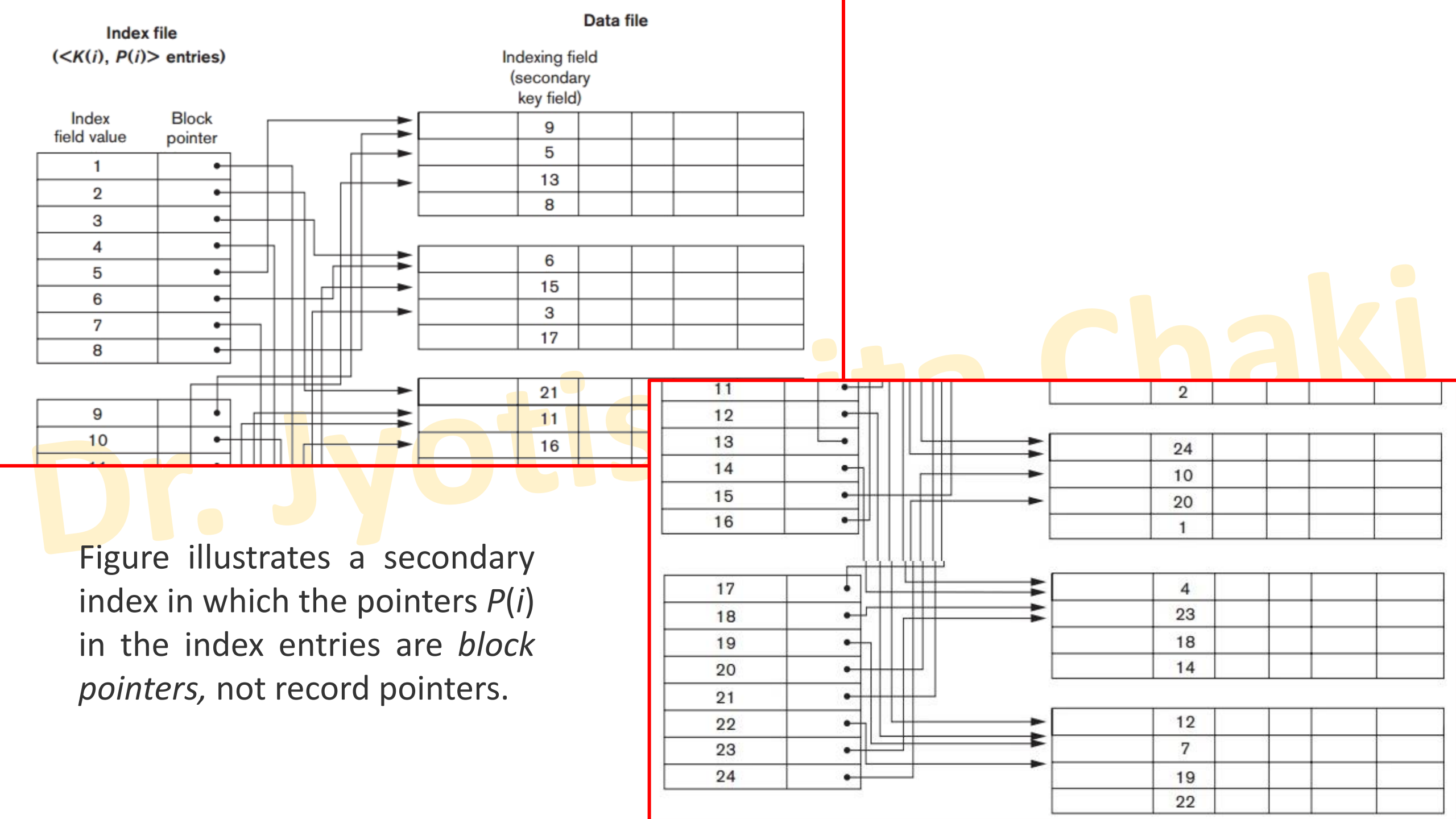
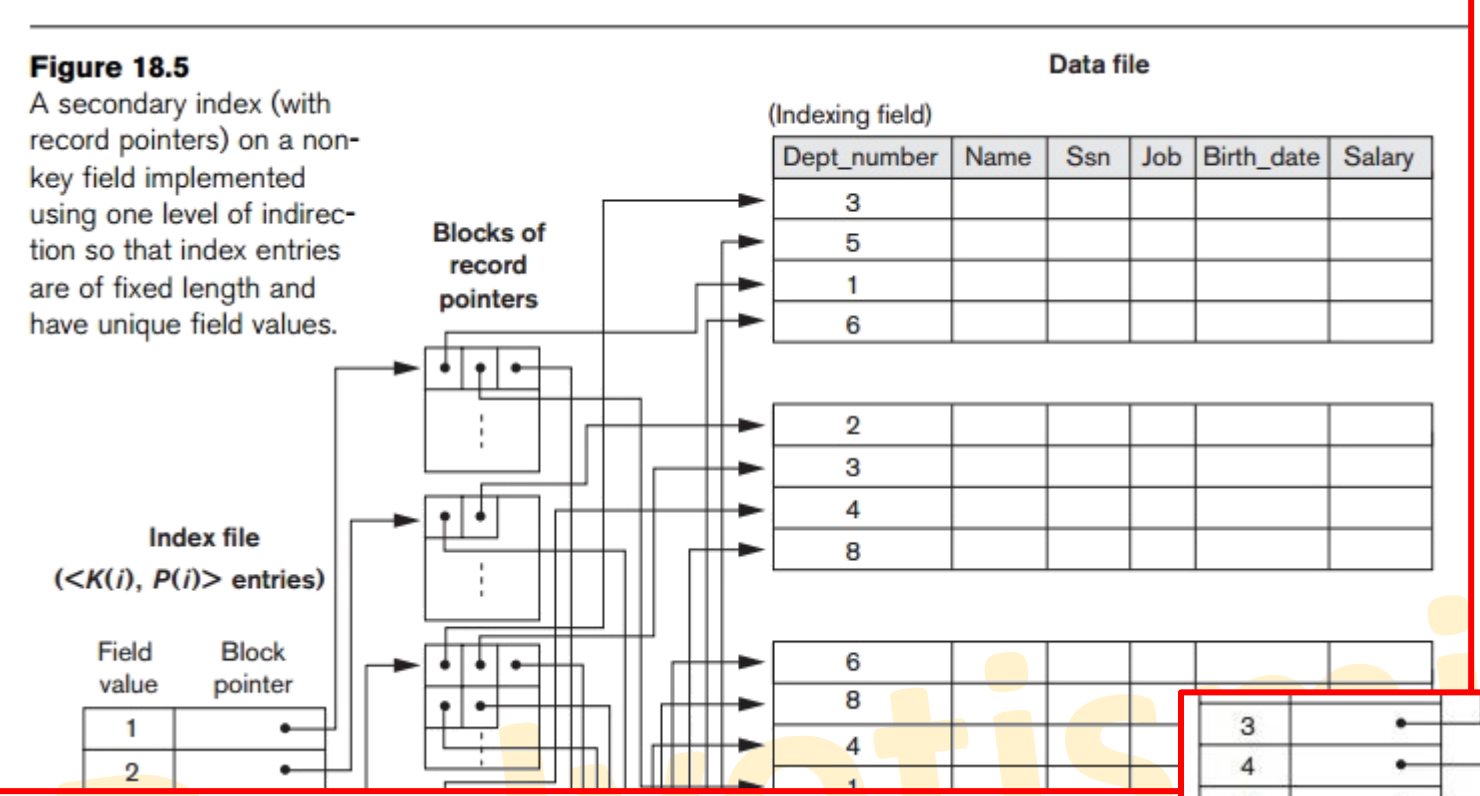
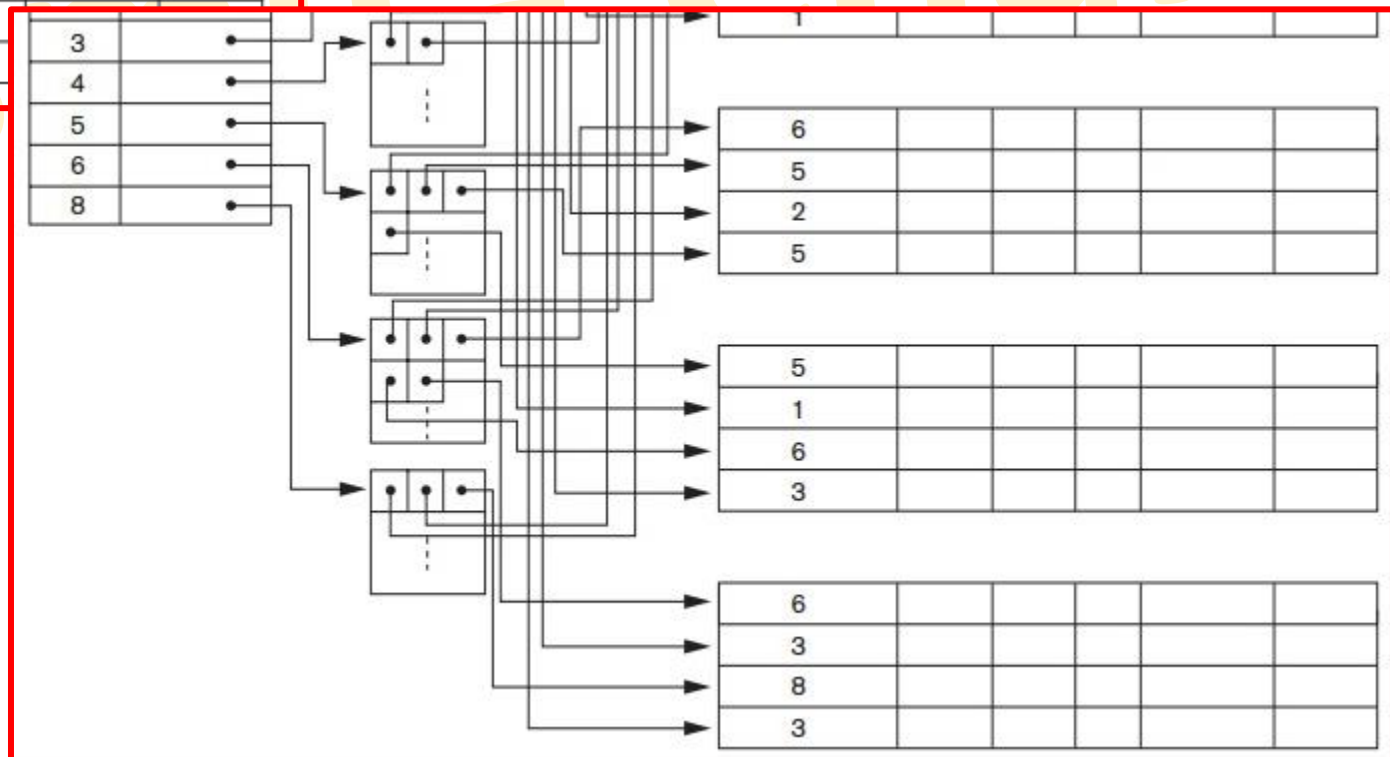


Figure illustrates a secondary index in which the pointers *P*(*i*) in the index entries are *block pointers,* not record pointers.

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

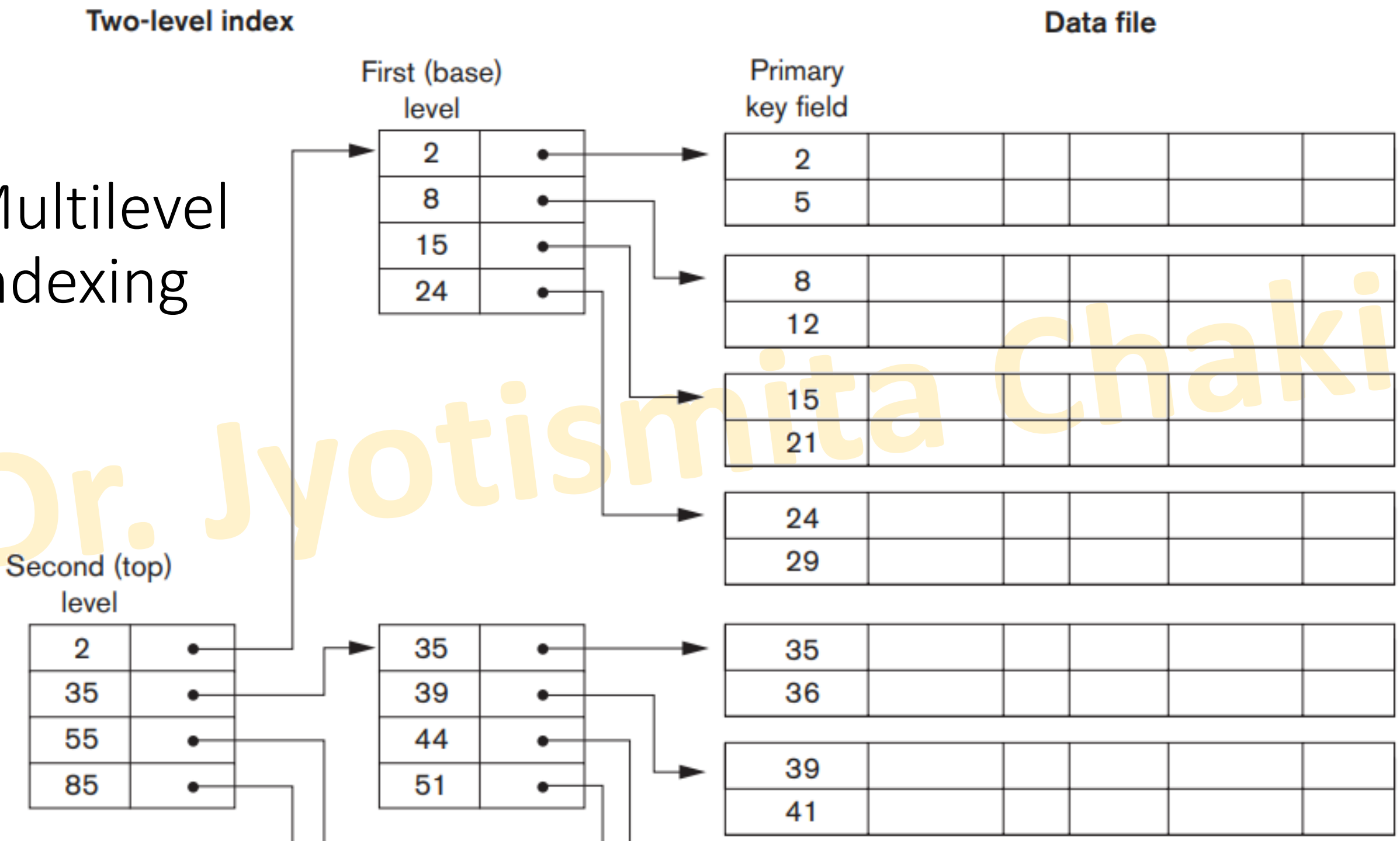
Multilevel Indexing

- 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.

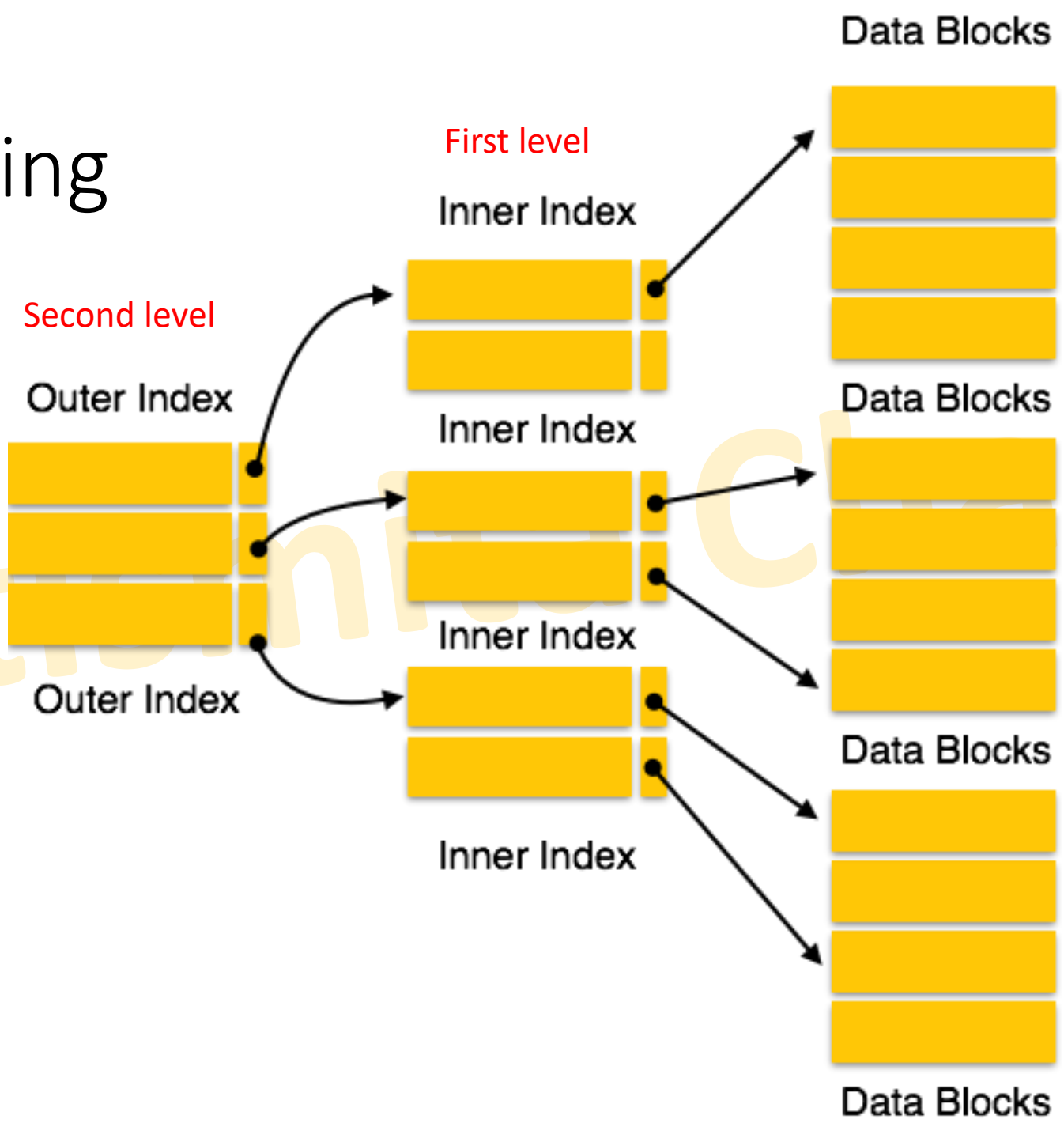
Multilevel Indexing

- 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



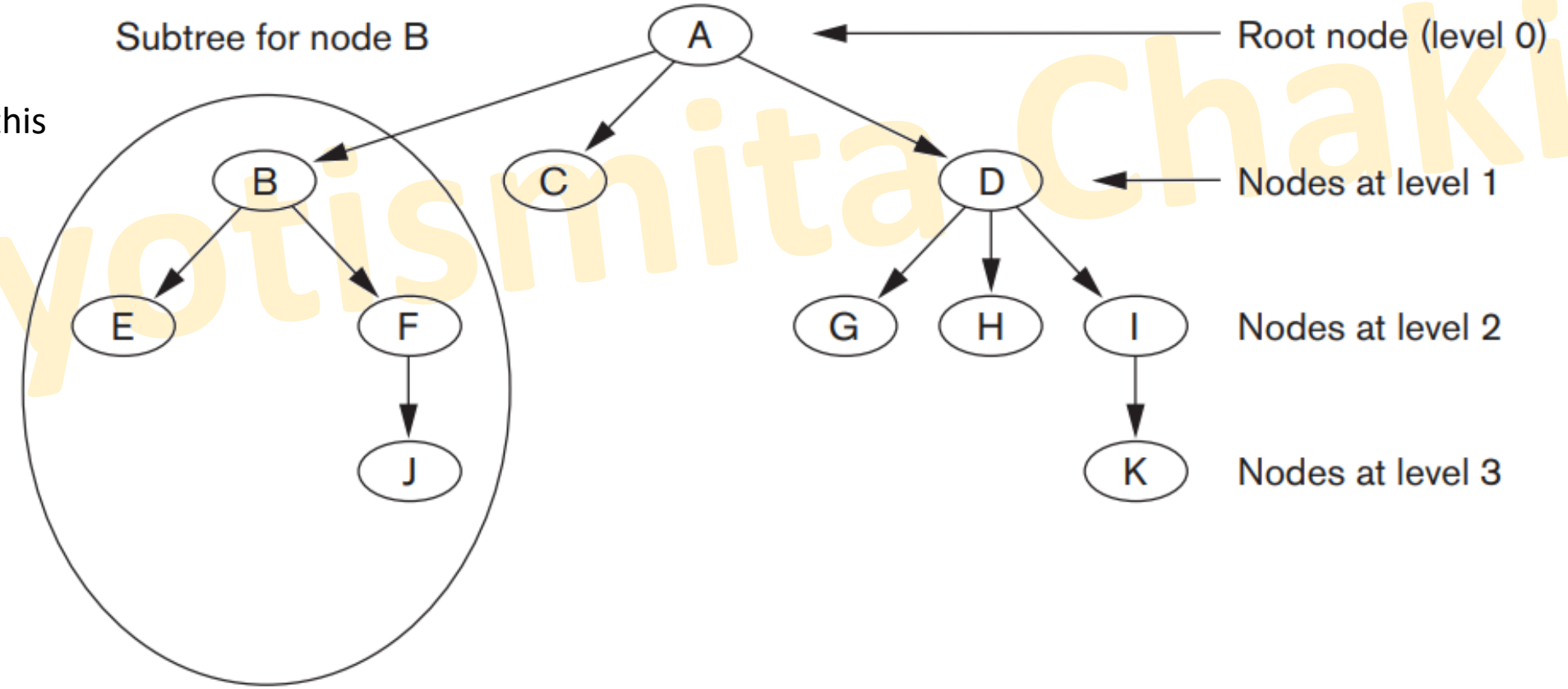
Dynamic Multilevel Indexing using B+ tree

- 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

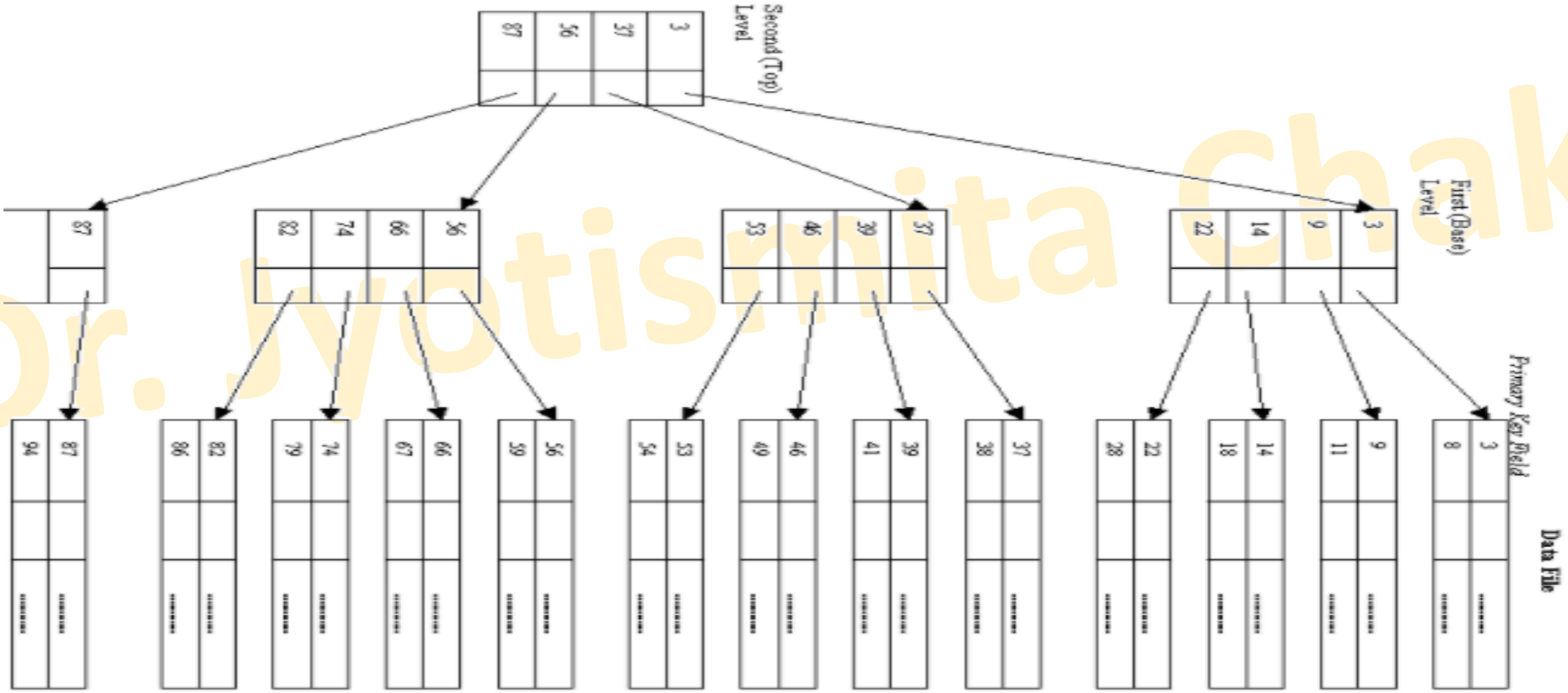
Since the leaf nodes are at different levels of the tree, this tree is called unbalanced.

A tree data structure that shows an unbalanced tree.



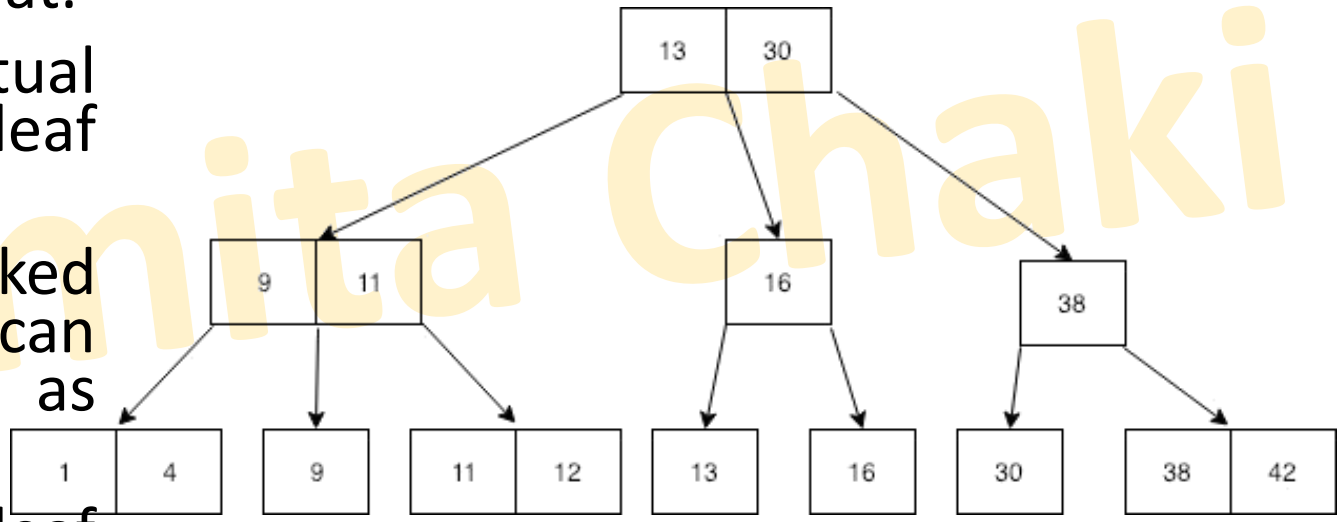
(Nodes E, J, C, G, H, and K are leaf nodes of the tree)

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.

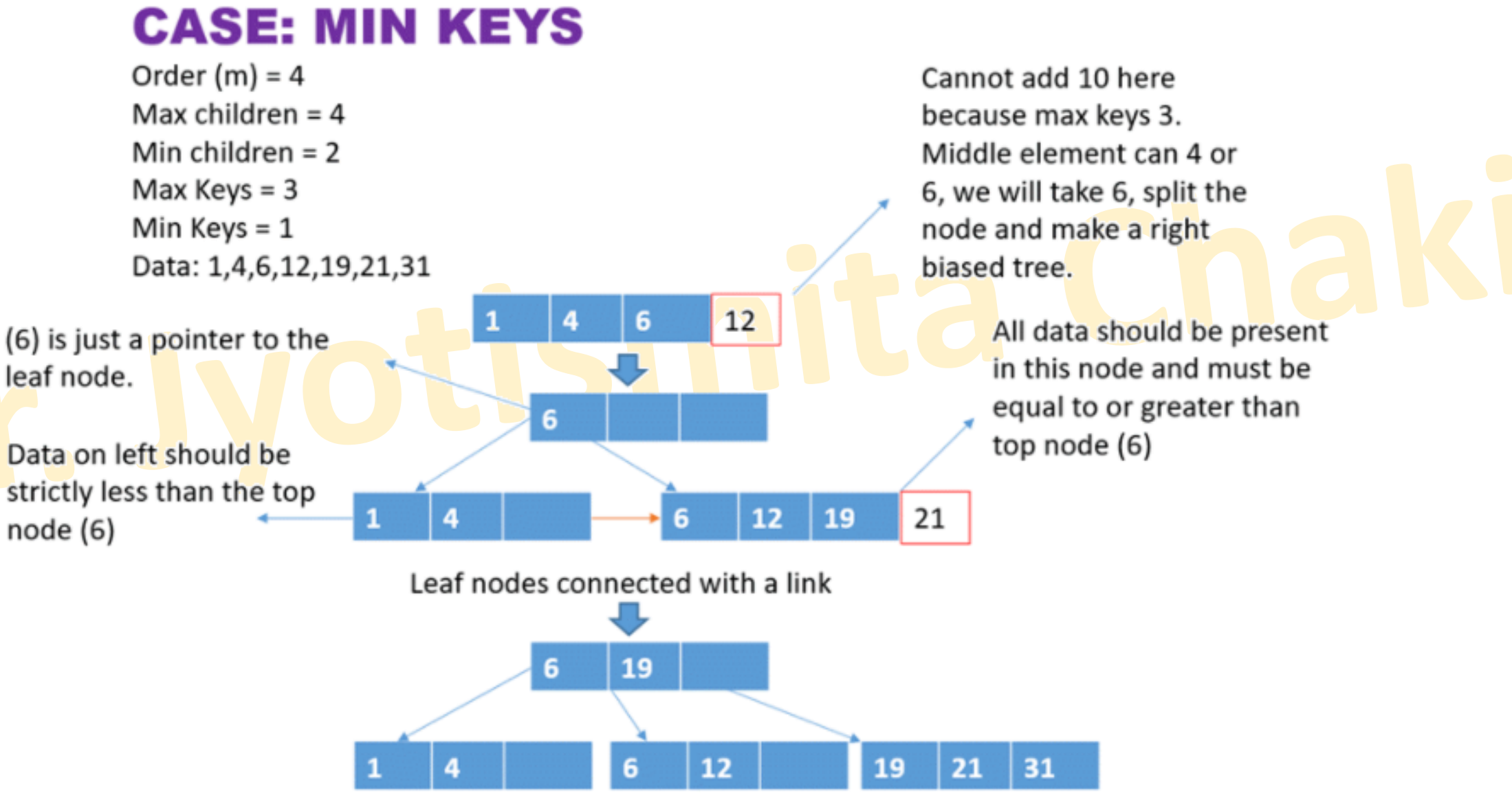
Dynamic Multilevel Indexing using B+ tree

- 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.

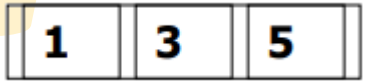
Dynamic Multilevel Indexing using B+ tree: Insertion



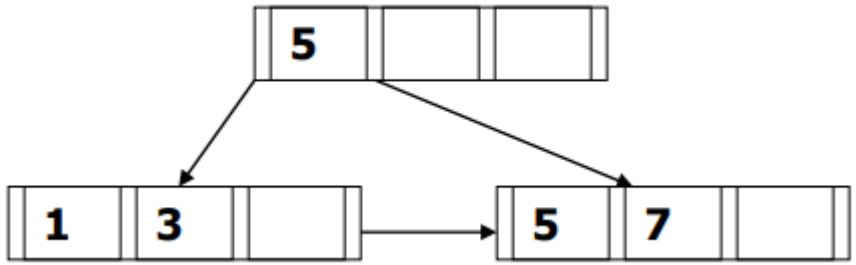
Dynamic Multilevel Indexing using B+ tree: Insertion

- 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

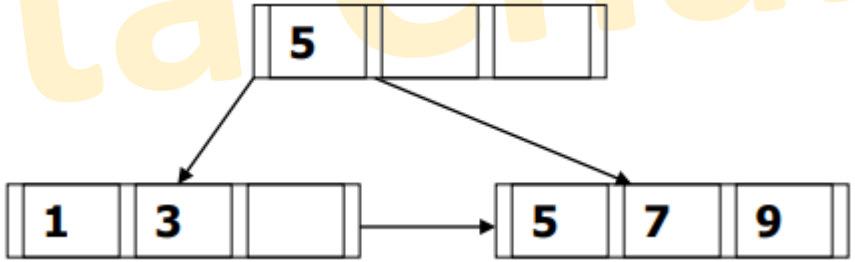
- Step2: Insert 3, 5



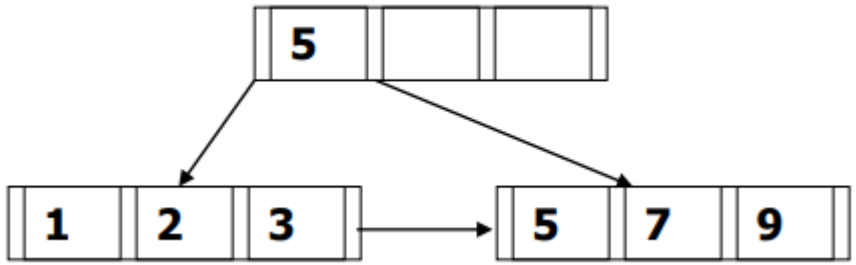
- Step3: Insert 7



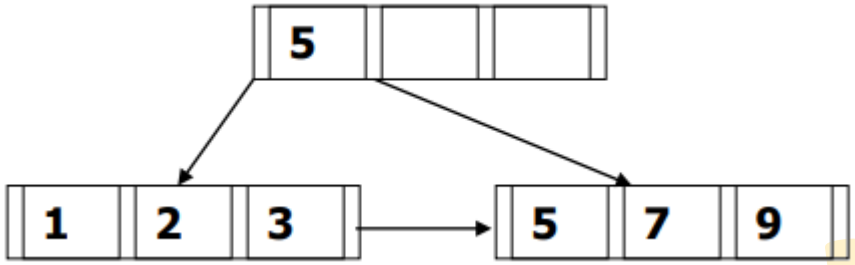
Step4: Insert 9



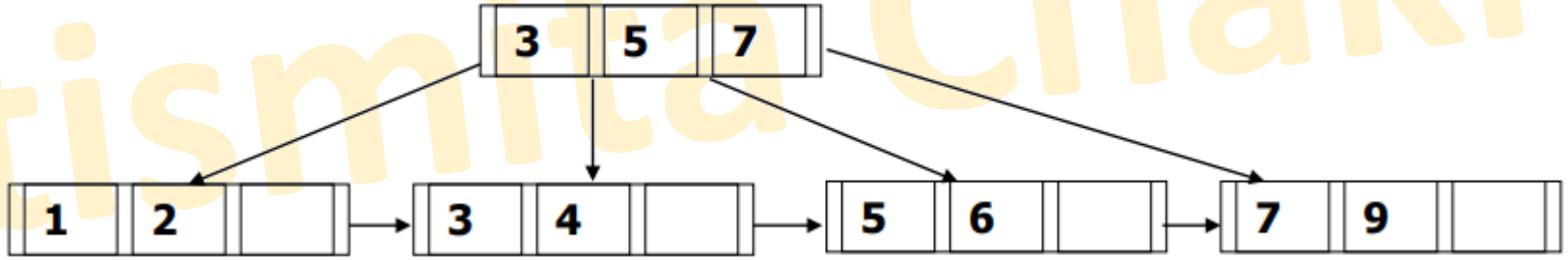
Step5: Insert 2



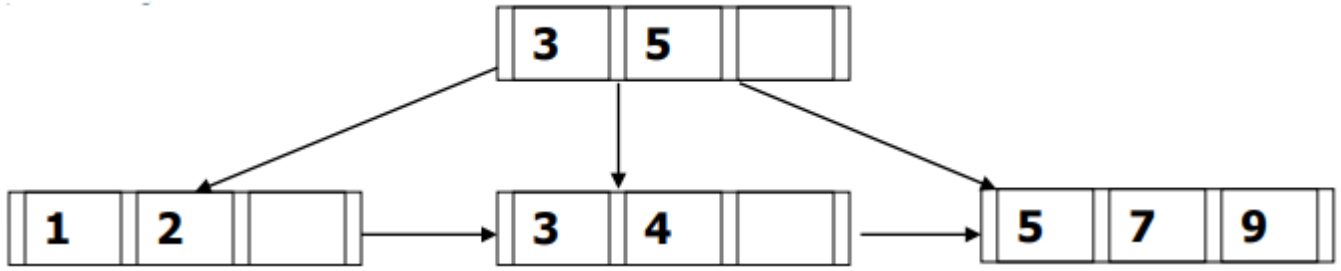
Dynamic Multilevel Indexing using B+ tree: Insertion



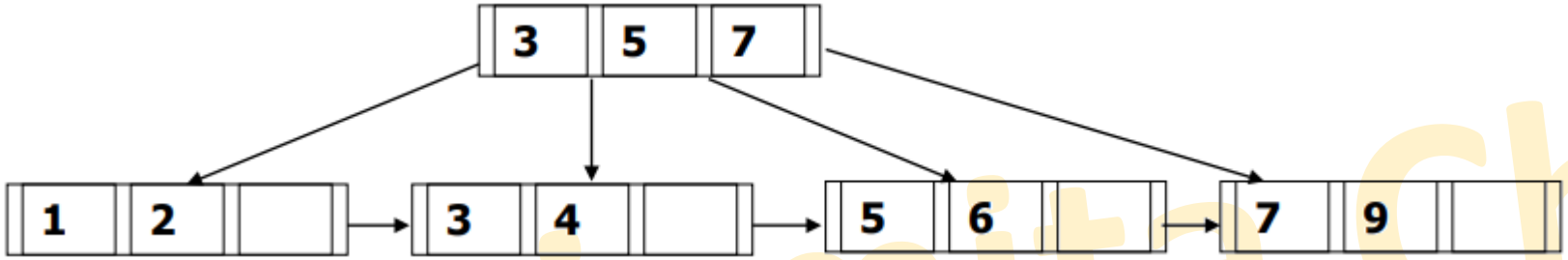
Step7: Insert 6



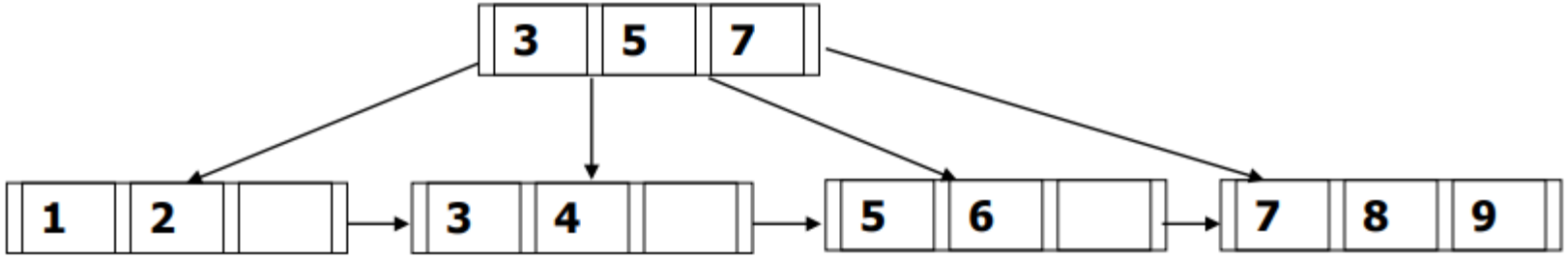
- Step 6: Insert 4



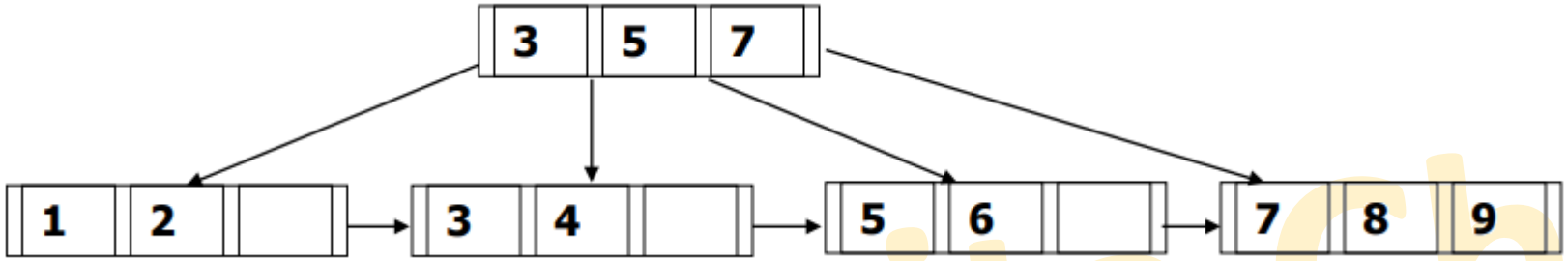
Dynamic Multilevel Indexing using B+ tree: Insertion



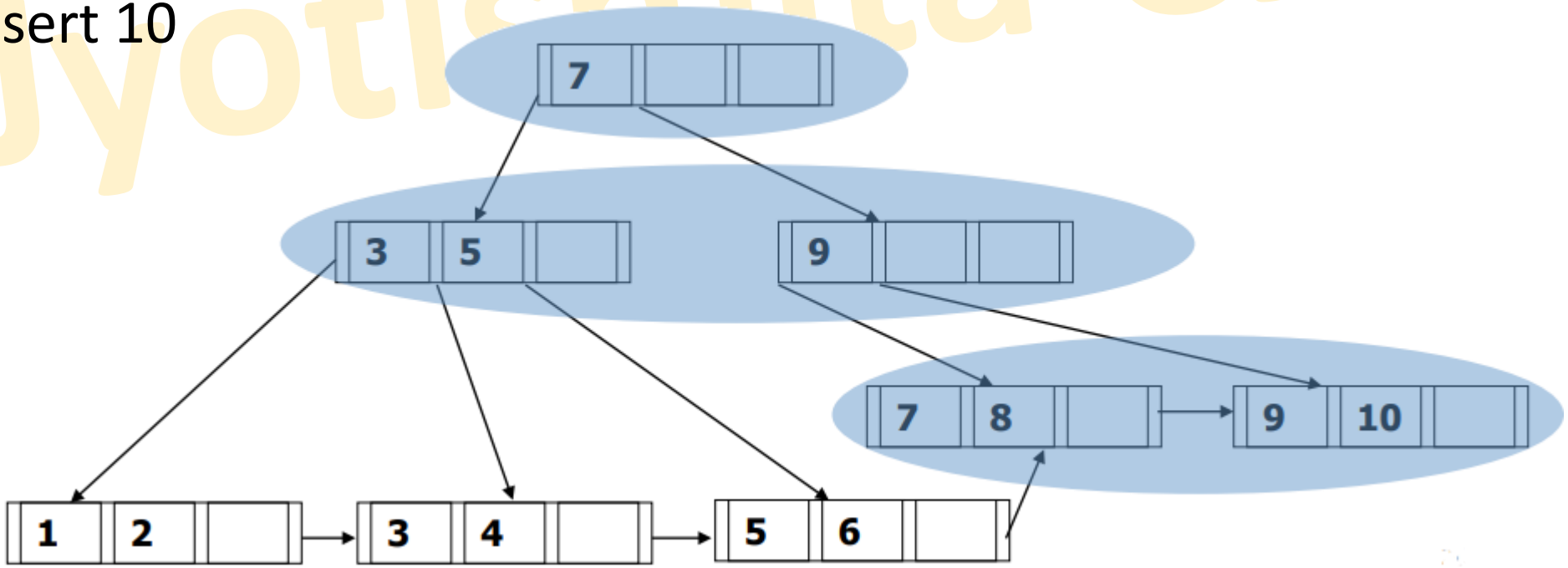
- Step 8: Insert 8



Dynamic Multilevel Indexing using B+ tree: Insertion

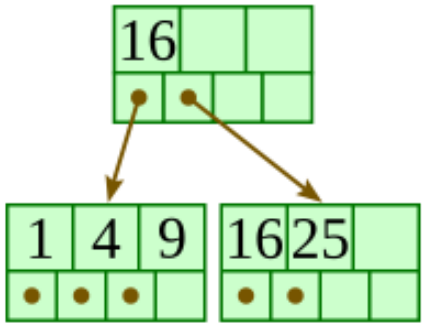


- Step9: Insert 10

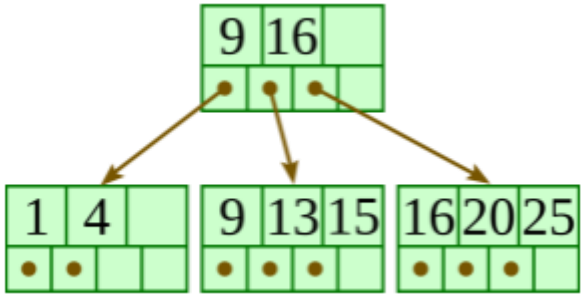


Dynamic Multilevel Indexing using B+ tree: Insertion

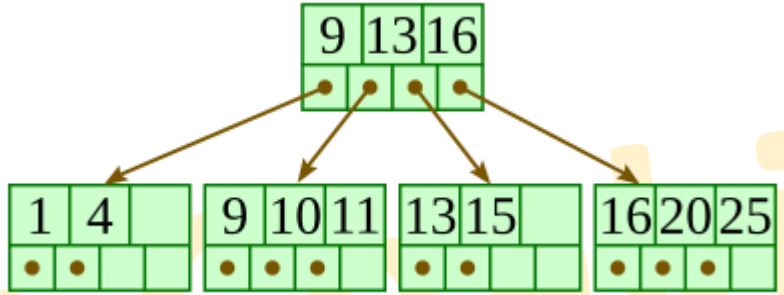
Initial:



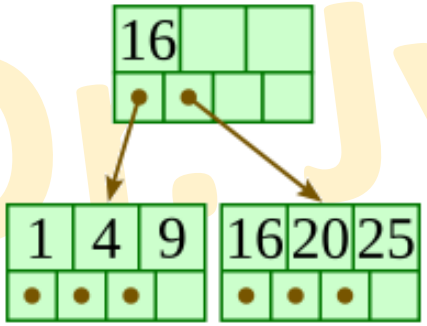
Insert 15:



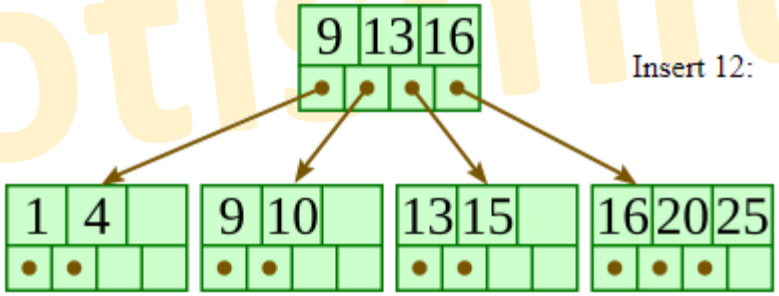
Insert 11:



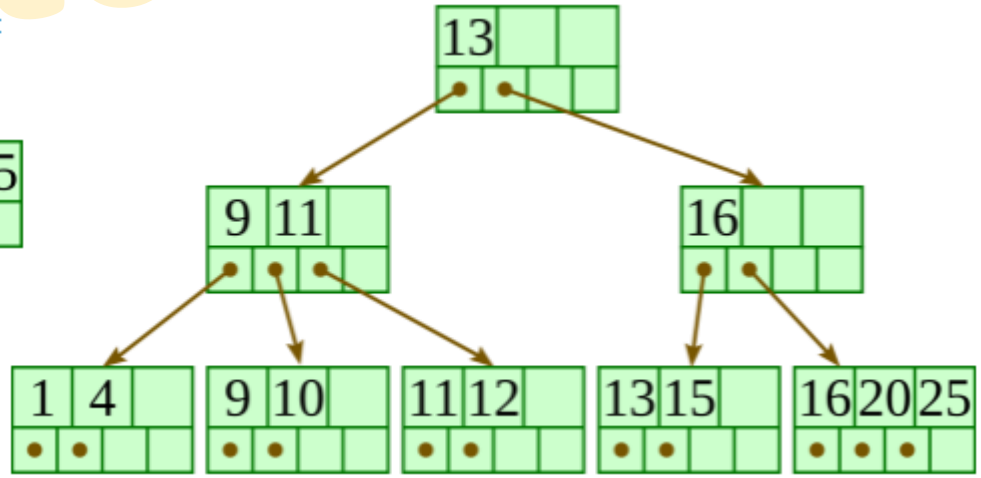
Insert 20:



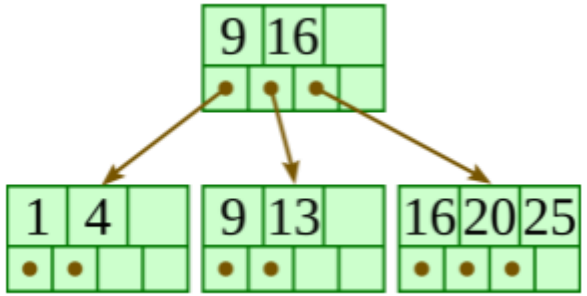
Insert 10:



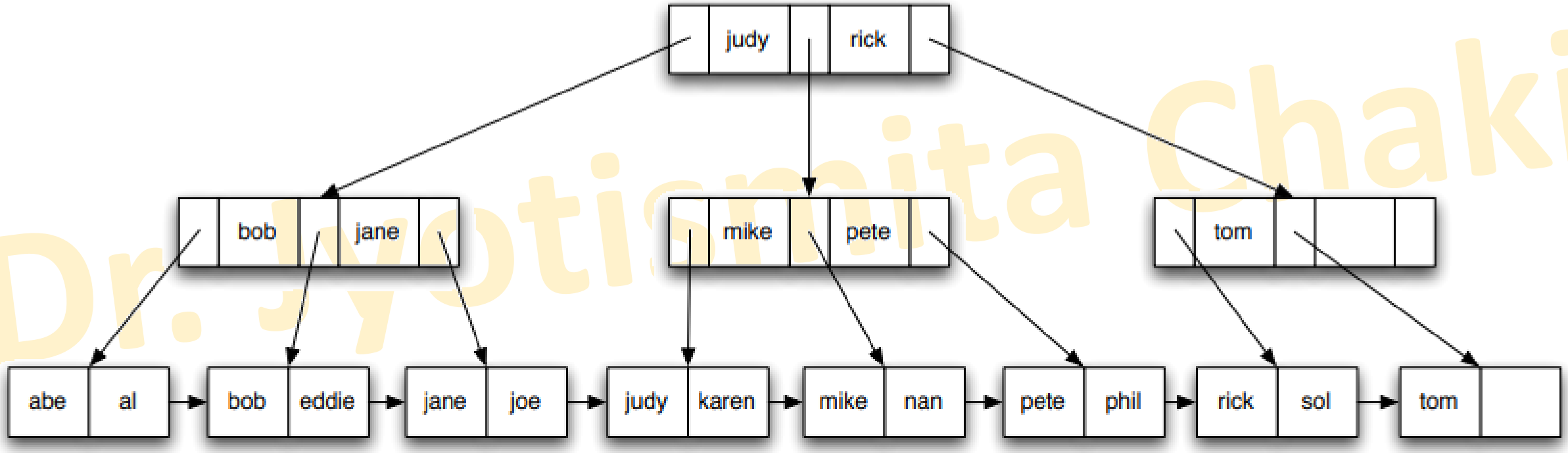
Insert 12:



Insert 13:

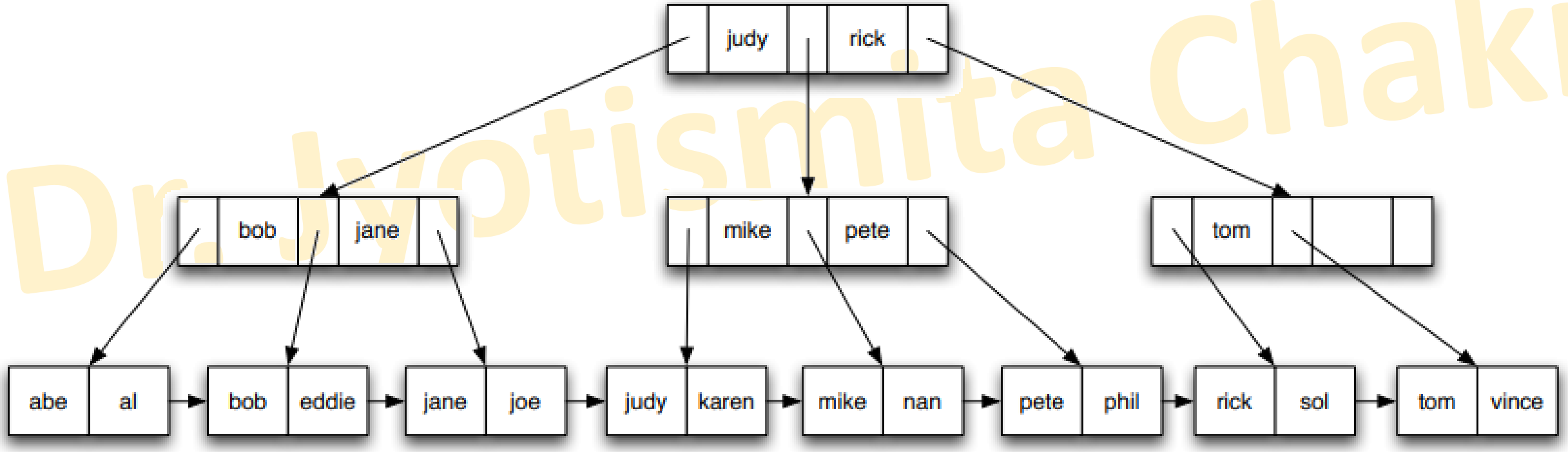


Dynamic Multilevel Indexing using B+ tree: Insertion



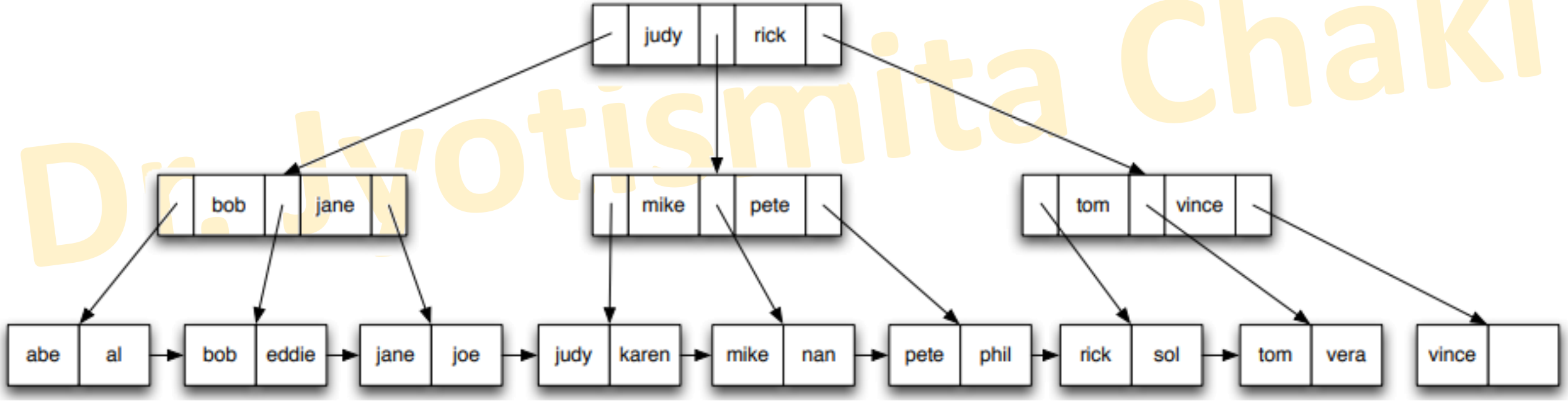
Dynamic Multilevel Indexing using B+ tree: Insertion

- Inserting Vince



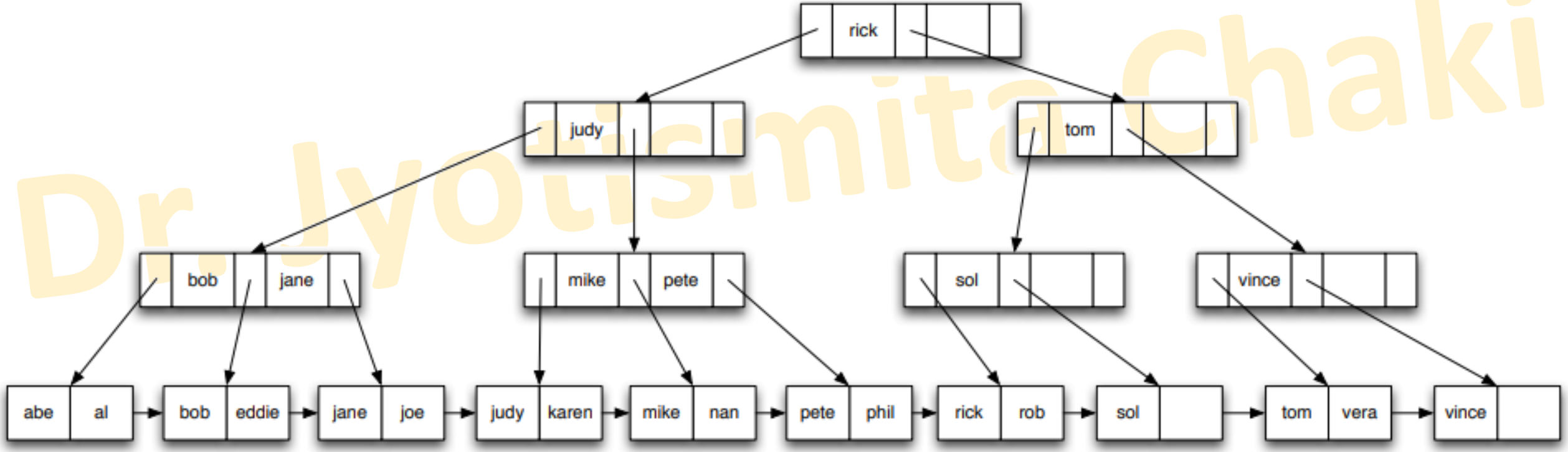
Dynamic Multilevel Indexing using B+ tree: Insertion

- Inserting Vera



Dynamic Multilevel Indexing using B+ tree: Insertion

- Inserting rob



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

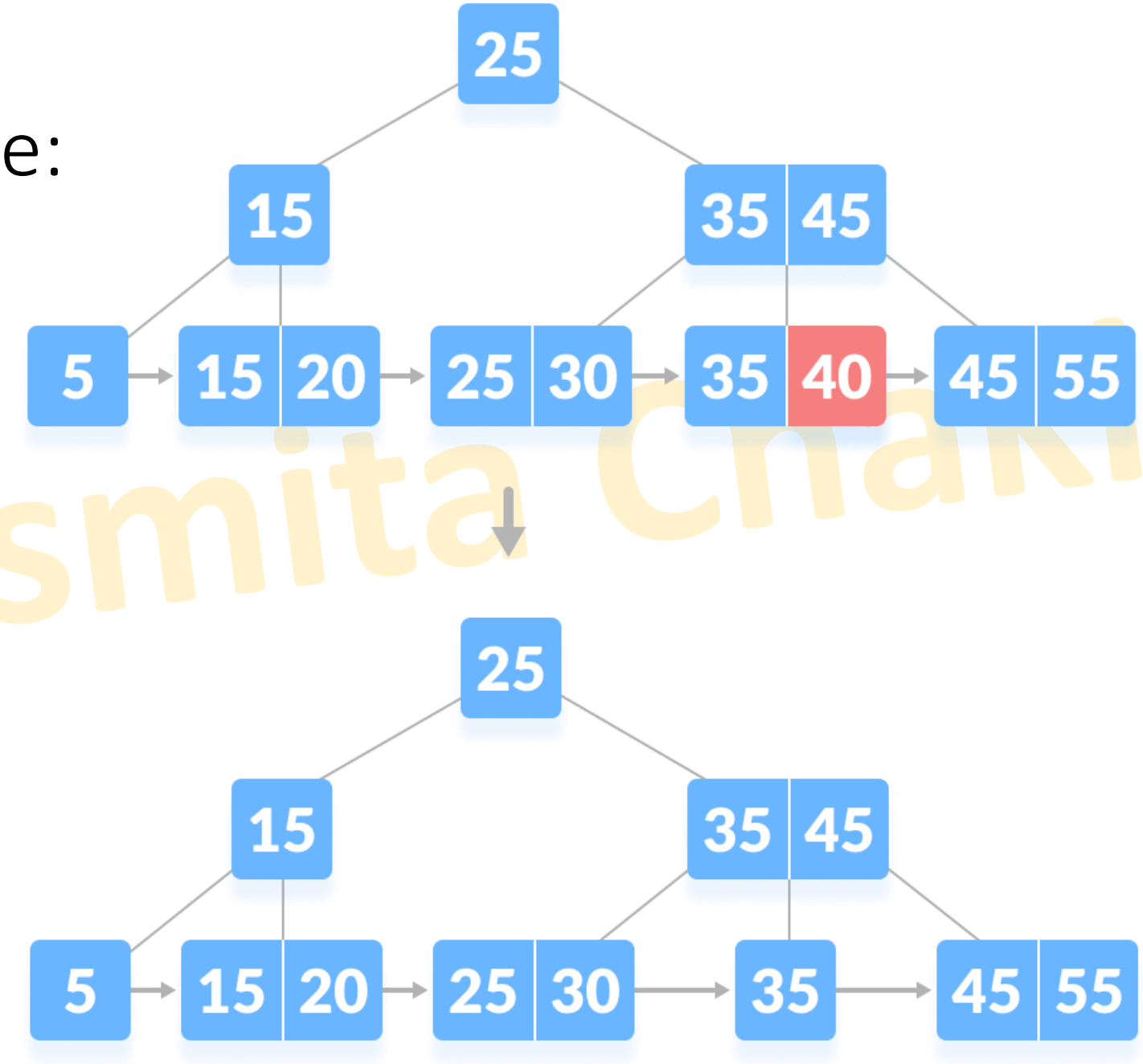
Dynamic Multilevel Indexing using B+ tree: Deletion

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)

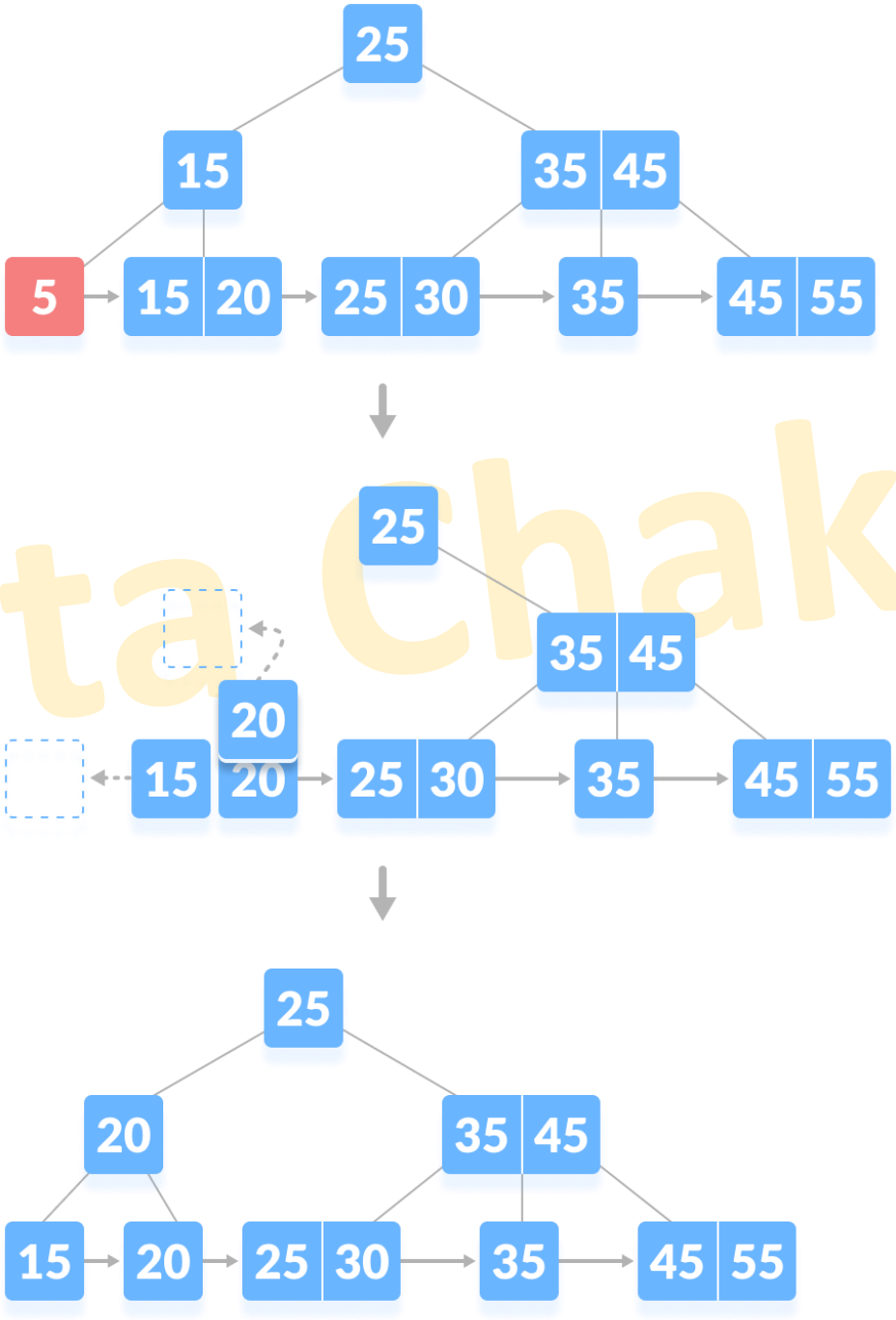
Dynamic Multilevel Indexing using B+ tree: Deletion

- **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.



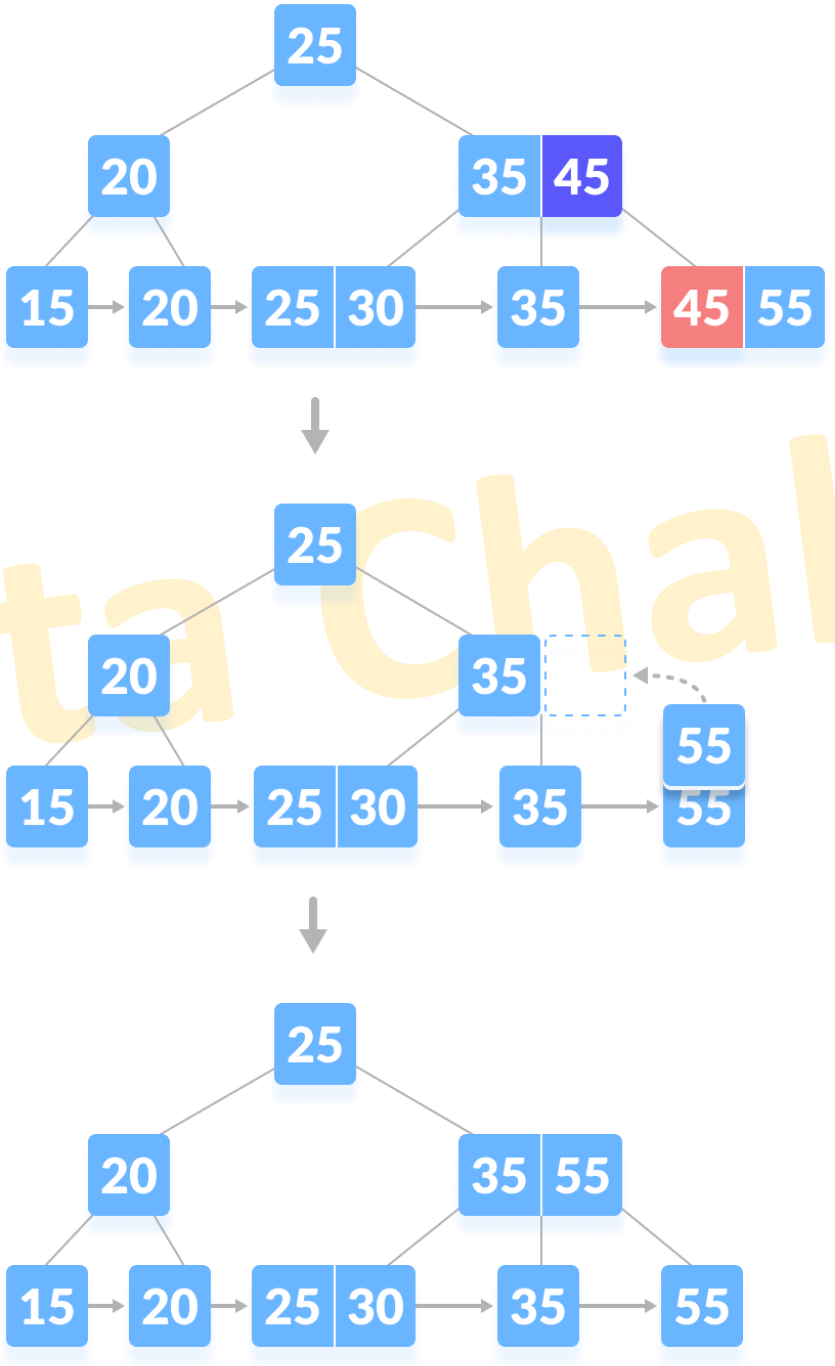
Dynamic Multilevel Indexing using B+ tree: Deletion

- **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.



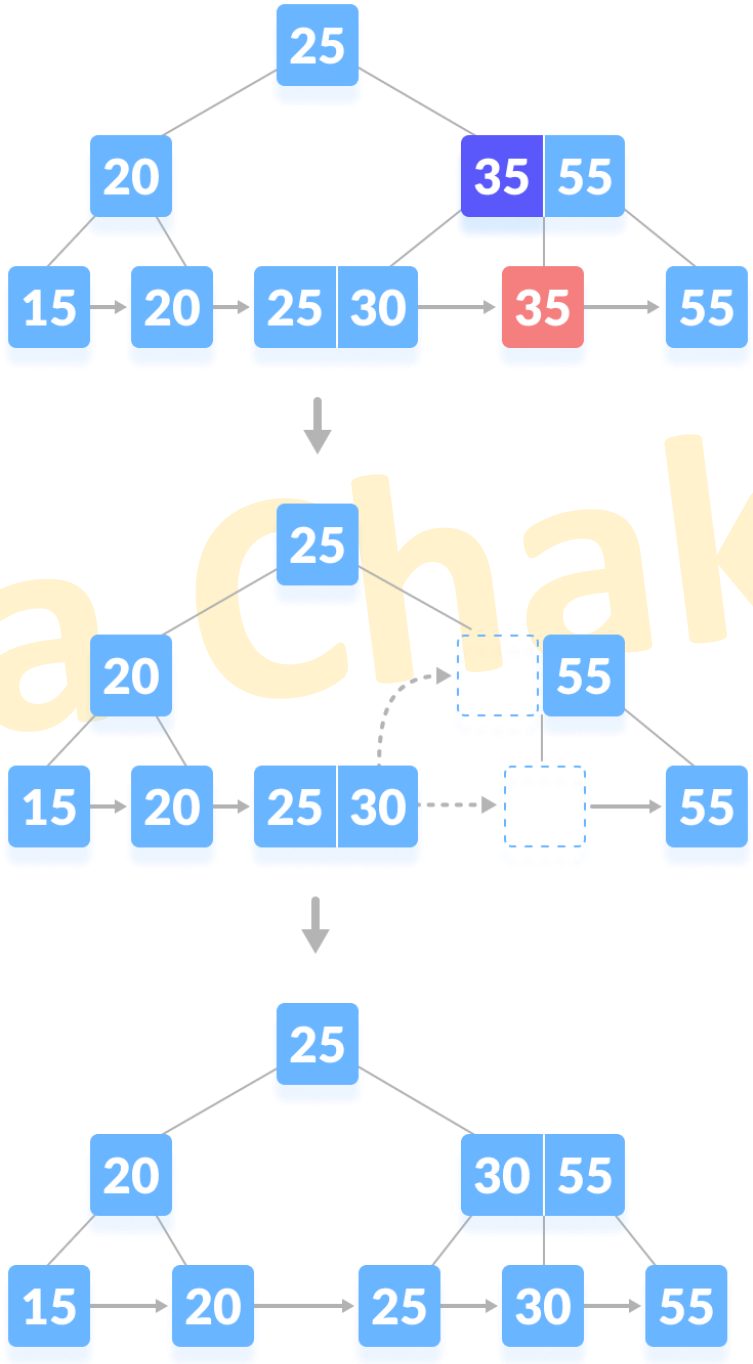
Dynamic Multilevel Indexing using B+ tree: Deletion

- **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.



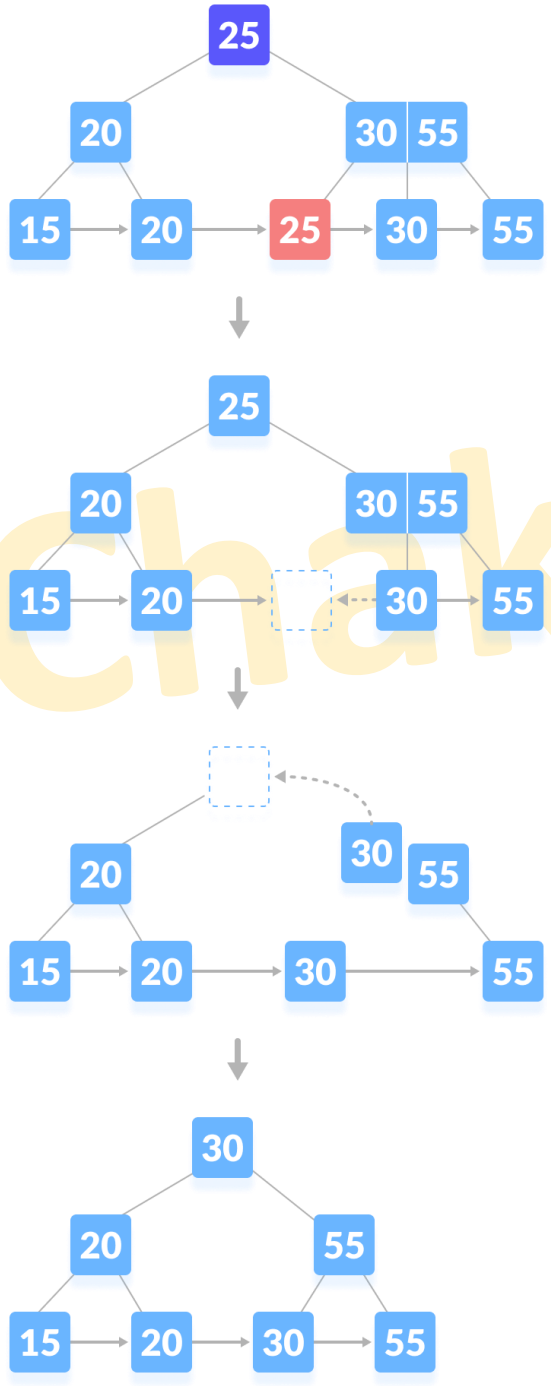
Dynamic Multilevel Indexing using B+ tree: Deletion

- **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.



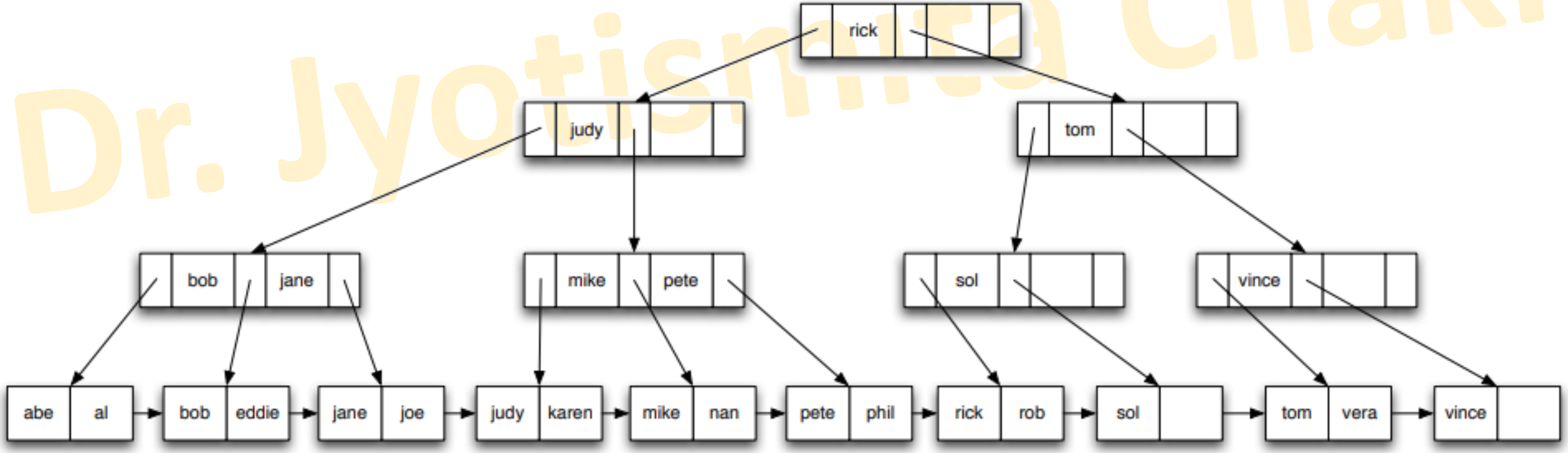
Dynamic Multilevel Indexing using B+ tree: Deletion

- **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.



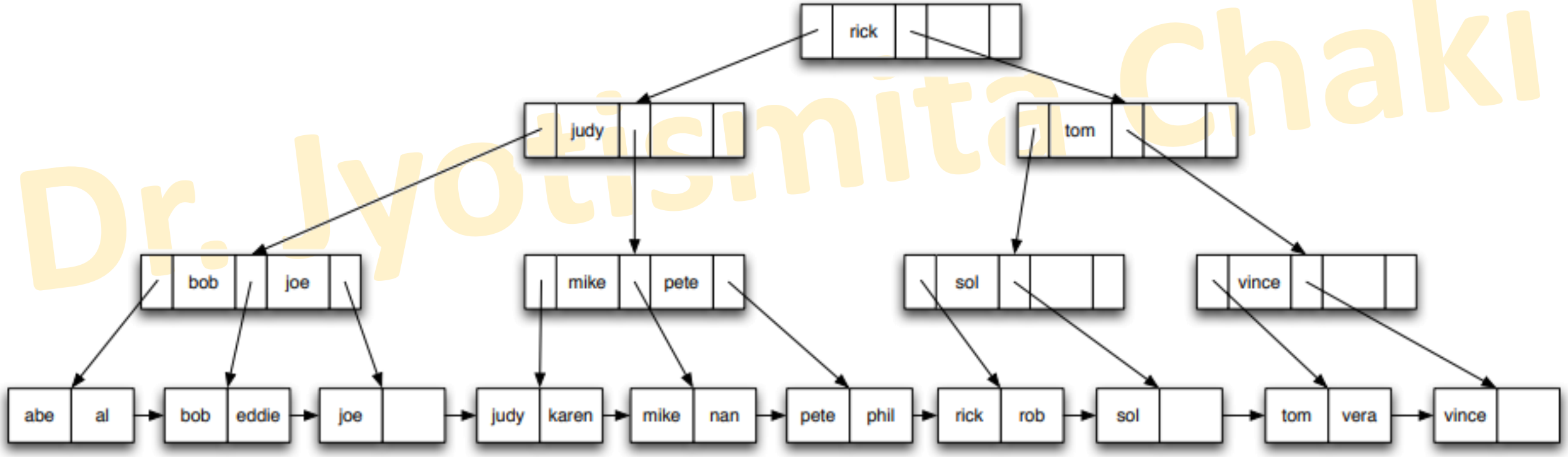
Dynamic Multilevel Indexing using B+ tree: Deletion

- Leafs must contain between 1 and 2 values
- Internal nodes must contain between 2 and 3 pointers
- Root must have between 2 and 3 pointers - Tree must be balanced, i.e., all paths from root to a leaf must be of same length



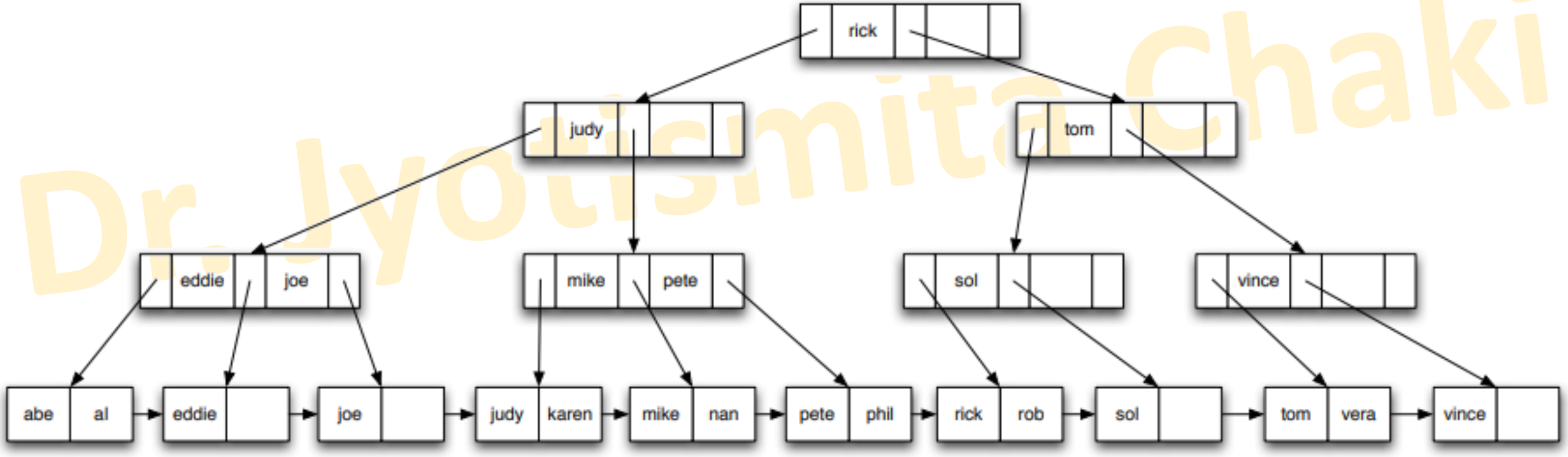
Dynamic Multilevel Indexing using B+ tree: Deletion

- Deleting jane



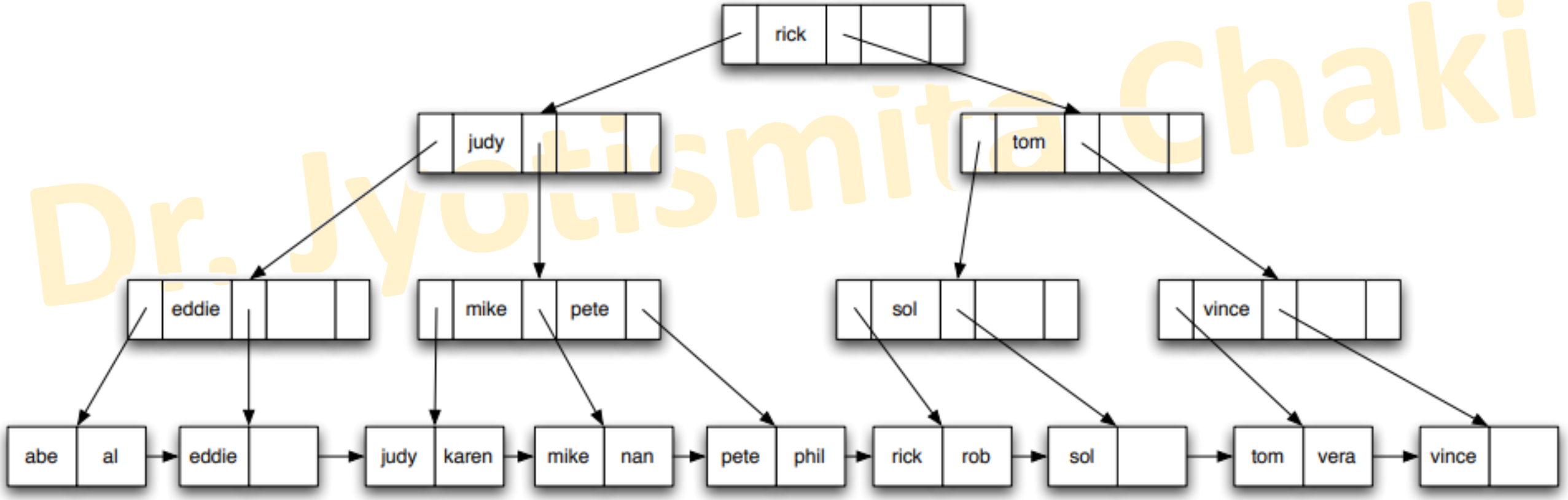
Dynamic Multilevel Indexing using B+ tree: Deletion

- Deleting bob



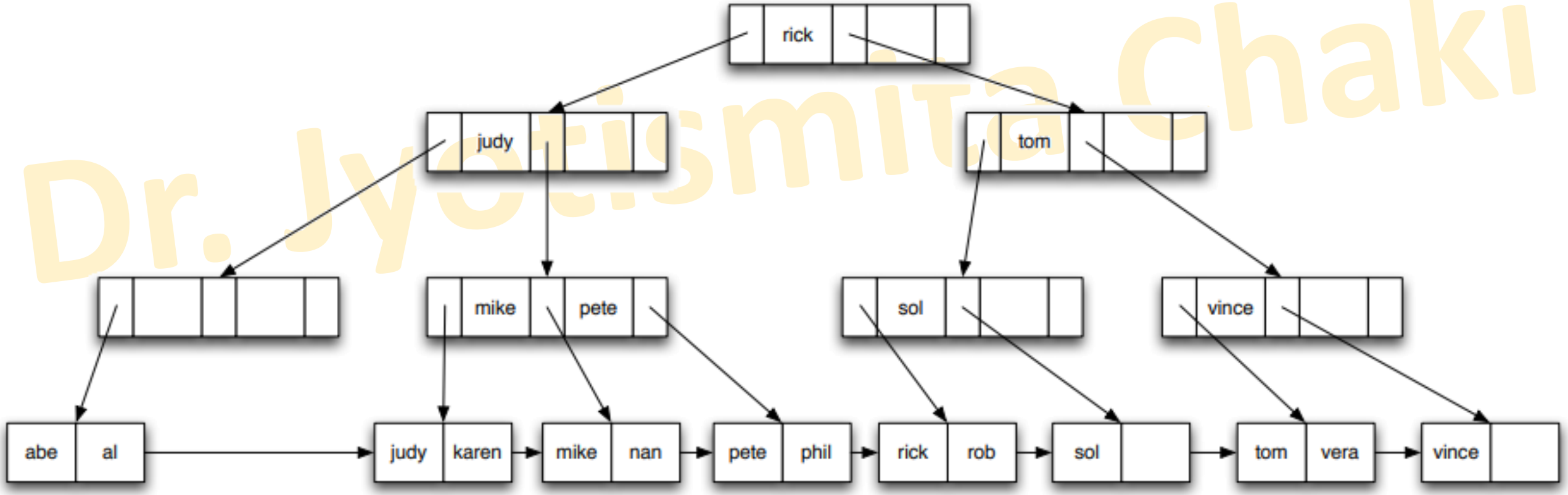
Dynamic Multilevel Indexing using B+ tree: Deletion

- Deleting joe



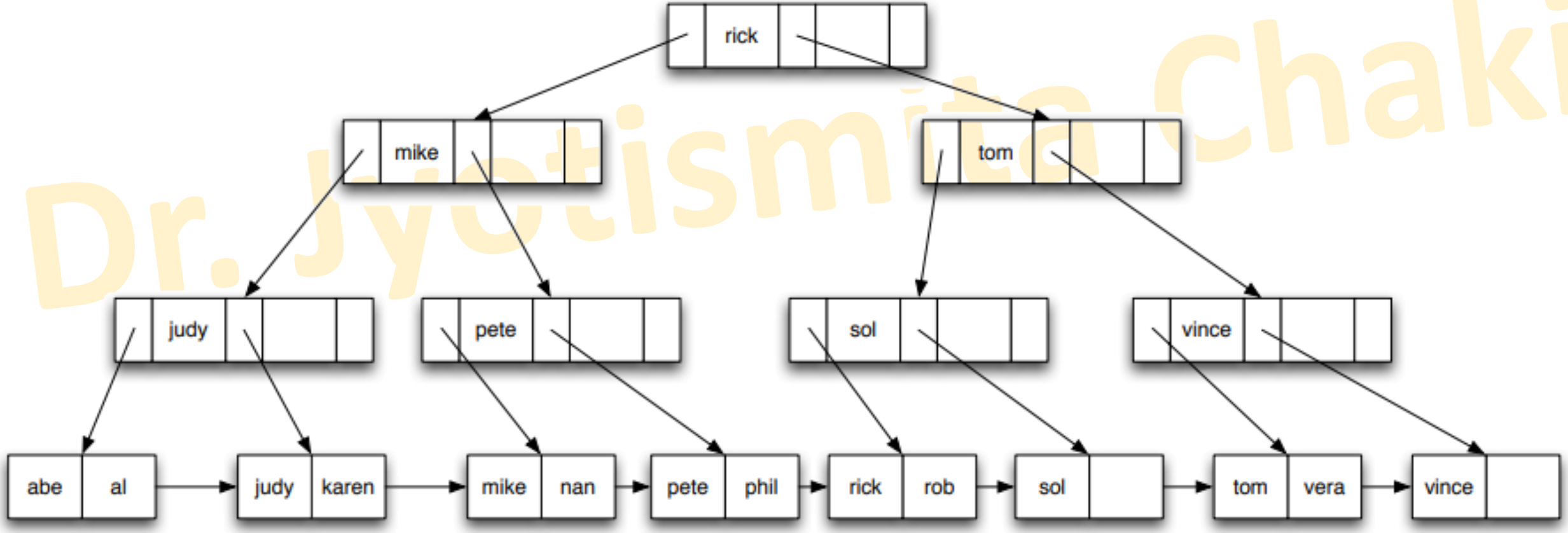
Dynamic Multilevel Indexing using B+ tree: Deletion

- Deleting eddie



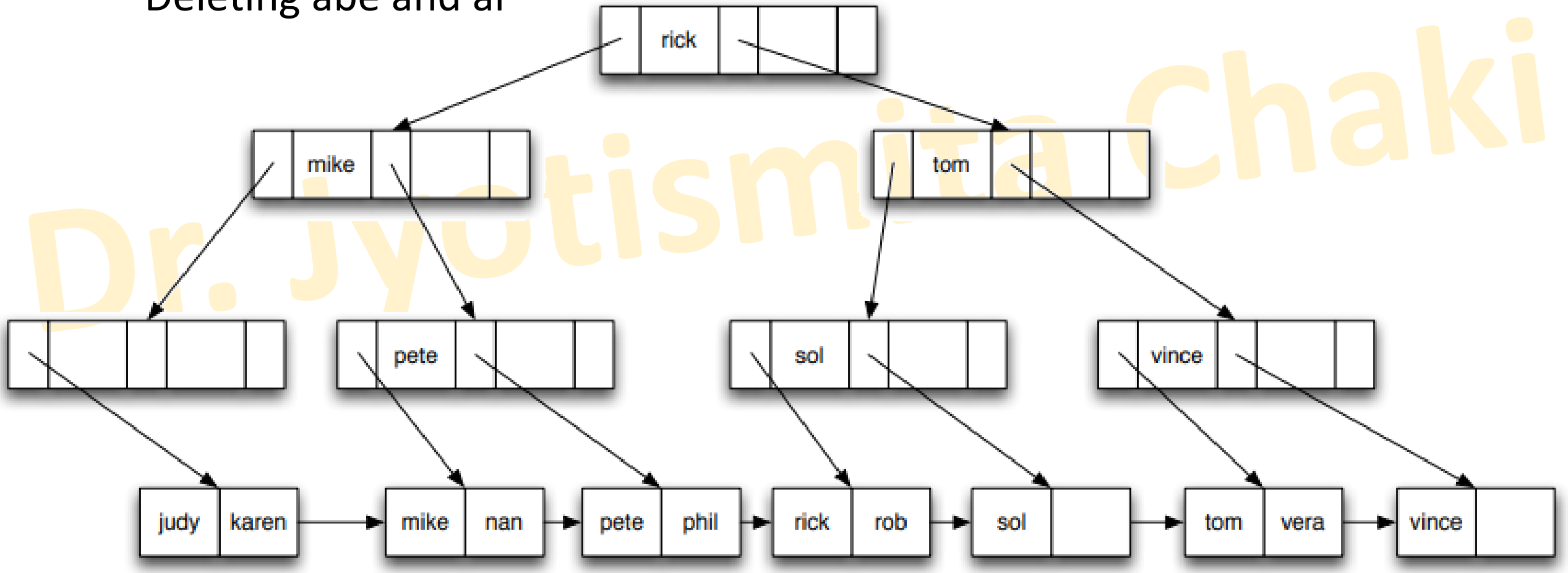
Dynamic Multilevel Indexing using B+ tree: Deletion

- Deleting eddie



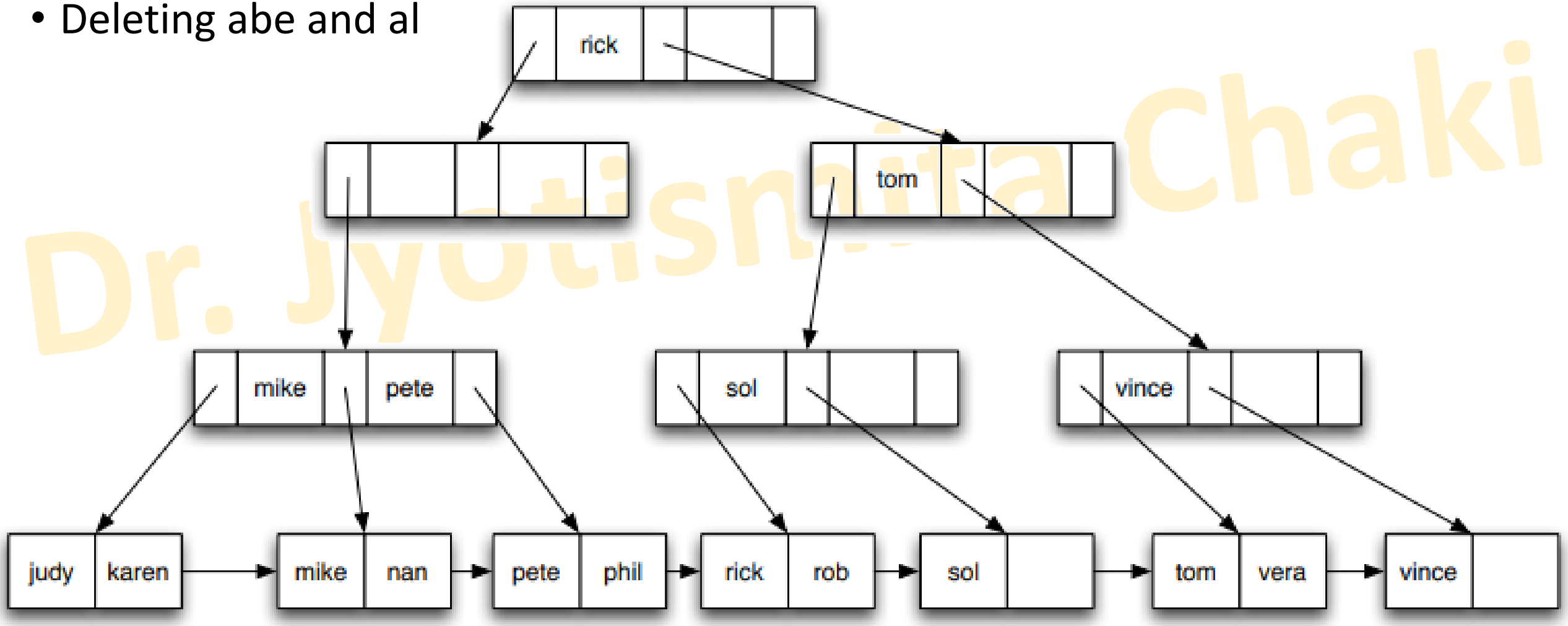
Dynamic Multilevel Indexing using B+ tree: Deletion

- Deleting abe and al



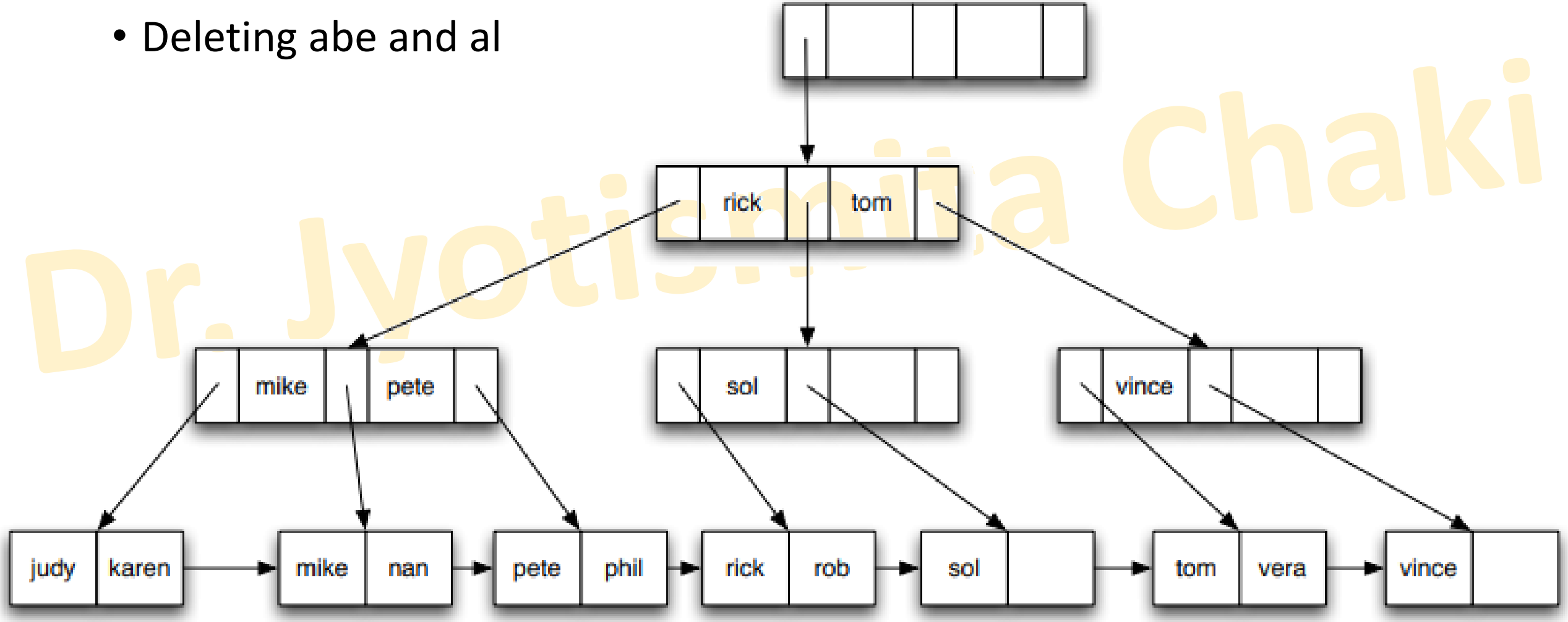
Dynamic Multilevel Indexing using B+ tree: Deletion

- Deleting abe and al



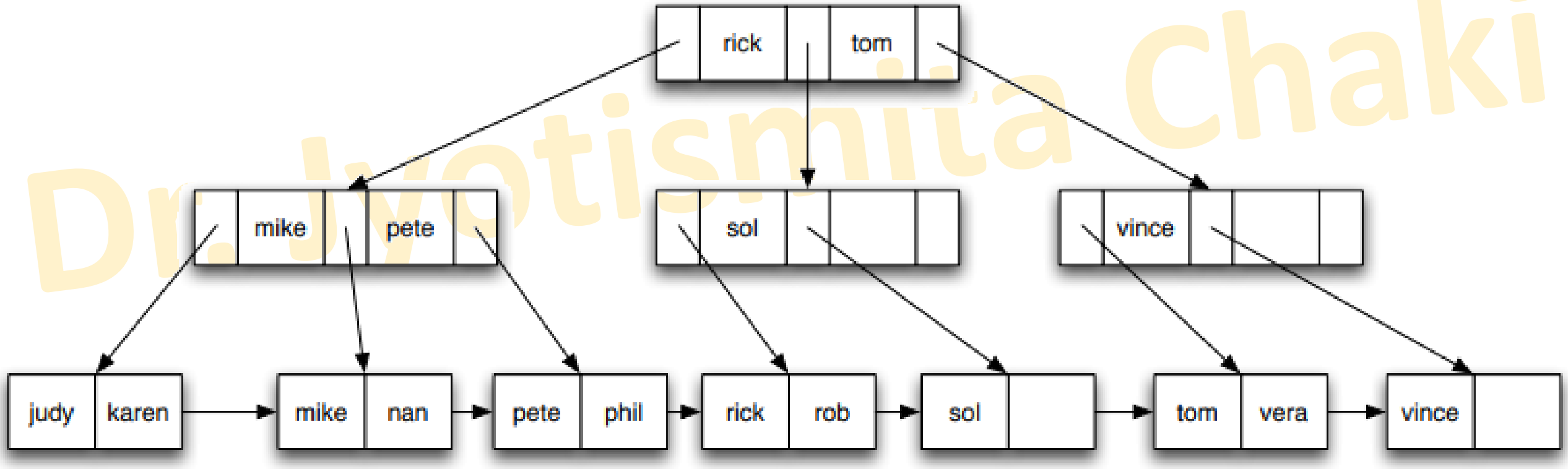
Dynamic Multilevel Indexing using B+ tree: Deletion

- Deleting abe and al



Dynamic Multilevel Indexing using B+ tree: Deletion

- Deleting abe and al



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.