UNIT V

Indexing and Multiway Tree

Introduction to Indexing

- A file is a collection of records, each record having one or more fields.
- The fields used to distinguish among the records are known as keys.
- File organization describes the way in which records are stored in a file.
- File organization is concerned with representing data records on an external storage media.
- The choice of such a representation depends on the environment where the fi le is to operate, forexample, real-time, batched, simple query, one key, or multiple keys. When there is only one key, the records may be stored on this key and stored sequentially either on a tape or a disk.
- In these cases, the fi le organization breaks down into two more aspects:
  - Directory for the collection of indices
  - File organization for the physical organization of records
- A directory is a collection of indices.
- It may contain one index for every key or only one index for some of the keys.
- If an index contains an entry for every record, then it is called a dense index.
- If an index contains an entry for only some of the records, then it is non_x0002_dense index.

- The index is a collection of pairs of the form (key value, address). For example, consider the sample data for employee file.
- Suppose P1, P2, P3, P4, P5 are the disk addresses where these records are stored.
- An index is too large and has to be maintained on the external storage.
- **Cylinder-surface Indexing -** This is the simplest type of index organization.

Record	Emp. no.	Name	Occupation	Disk address
A	100	Saurabh	Developer	P1
B	500	Abolee	Project head	P2
C	300	Anagha	Developer	P3
D	200	Abhijeet	Project head	P4
E	400	Devnarayanan	Developer	P5

- **Cylinder-surface Indexing -** This is the simplest type of index organization.

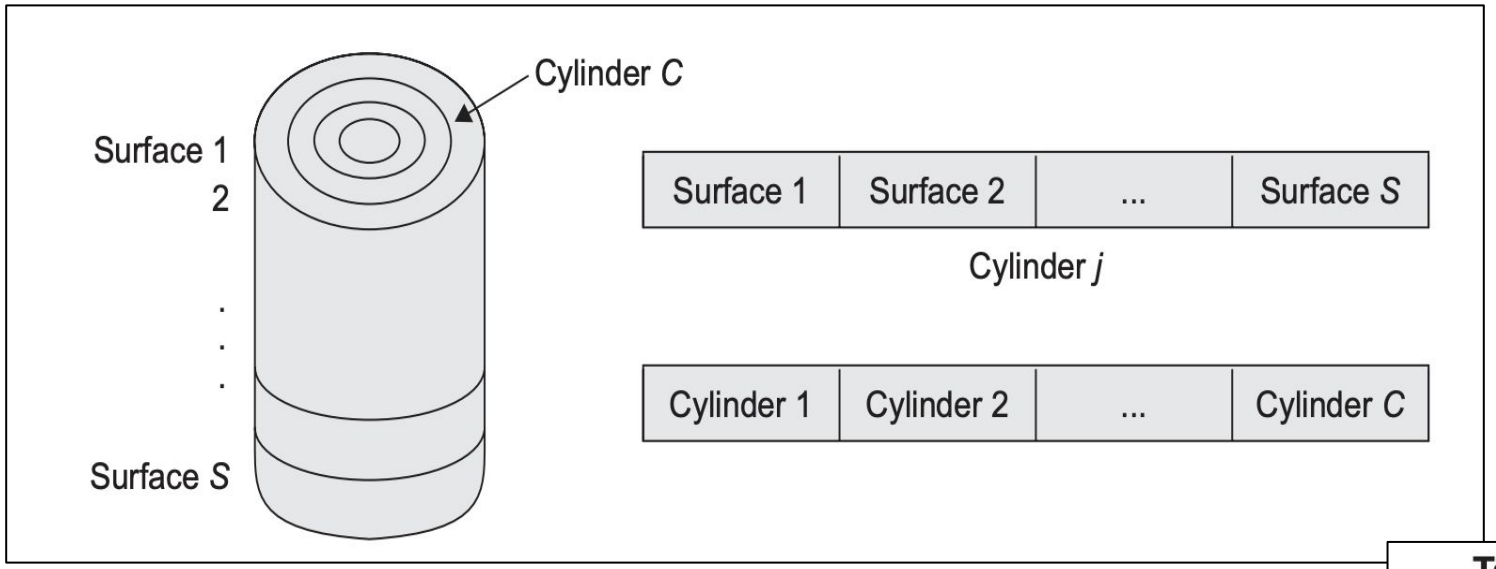
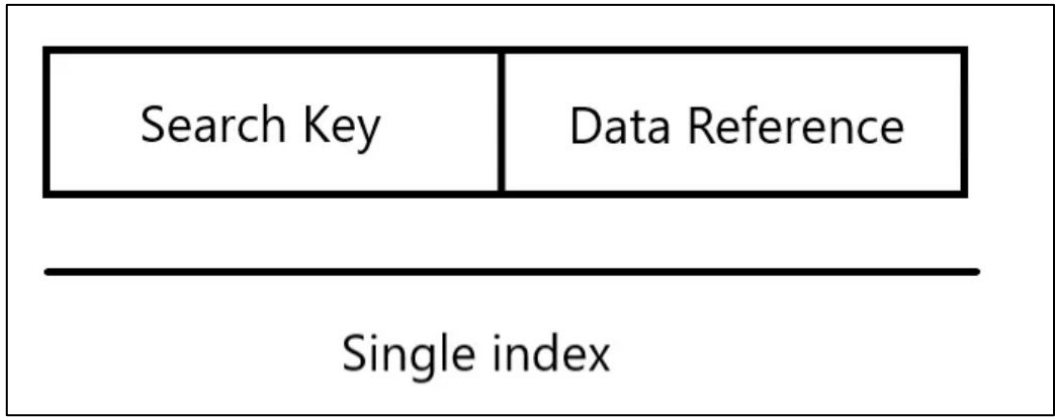


Table 13.2 Employee records cylinder-surface indexing

Emp. no.	Emp. name	Cylinder	Surface
1	Abolee	1	1
2	Anand	1	1
3	Amit	1	2
4	Amol	1	2
5	Rohit	2	1
6	Santosh	2	1
7	Saurabh	2	2
8	Shila	2	2

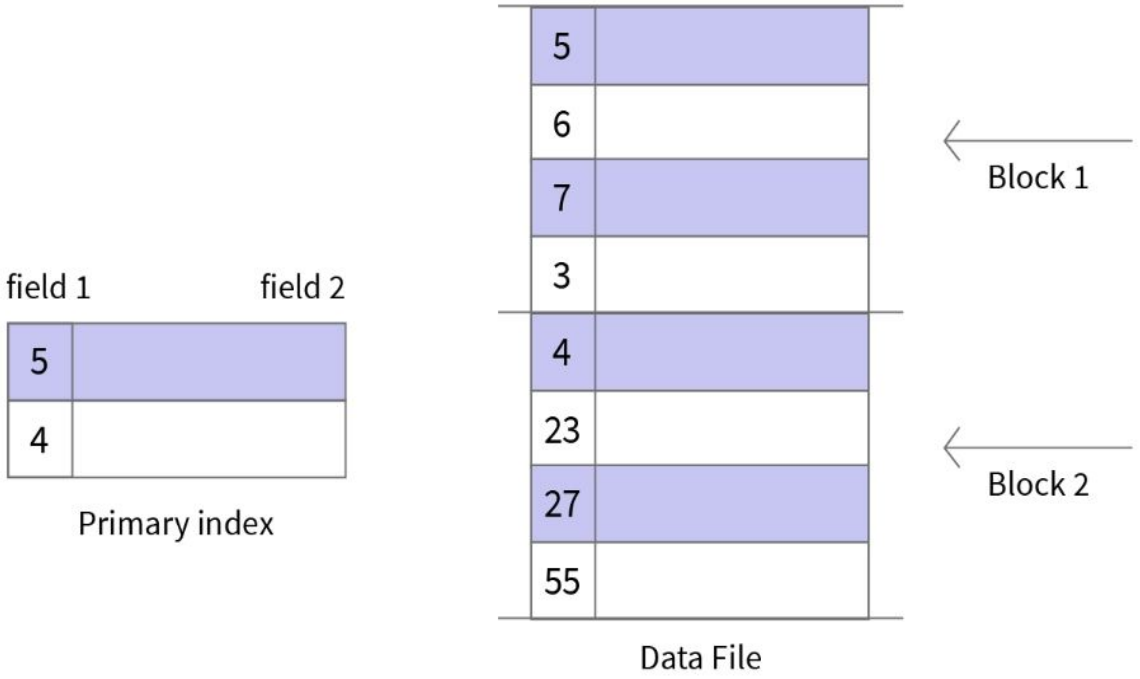
Indexing and Multiway Trees- Indexing

- Indexing is a data structure technique that helps to speed up data retrieval.
- Indexing minimizes the number of disk accesses required when a query is processed.
- Indexes are created as a combination of the two columns.
- **First column** is the Search key. It contains a copy of the primary key or candidate key of the table. The values of this column may be sorted or not. But if the values are sorted, the corresponding data can be accessed easily.
- **Second column** is the Data reference or Pointer. It contains the address of the disk block where we can find the corresponding key value.



Indexing Techniques

- **Primary Indexing -**
  - The indexing or the index table created using Primary keys is known as Primary Indexing.
  - It is defined on ordered data. As the index is comprised of primary keys, they are unique, not null, and possess one to one relationship with the data blocks.

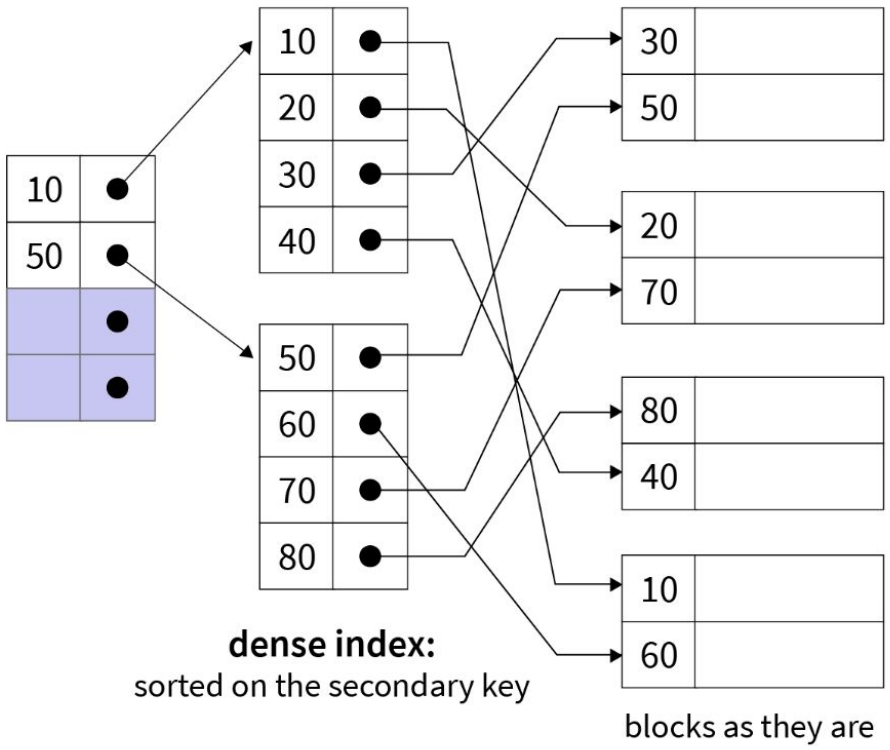


Characteristics of Primary Indexing:

1. Search Keys are unique.
2. Search Keys are in sorted order.
3. Search Keys cannot be null as it points to a block of data.
4. Fast and Efficient Searching.

Secondary Indexing (Non clustered Indexing)

- It is a two-level indexing technique used to reduce the mapping size of the primary index.
- The secondary index points to a certain location where the data is to be found but the actual data is not sorted like in the primary indexing.
- Secondary Indexing is also known as non-clustered Indexing.

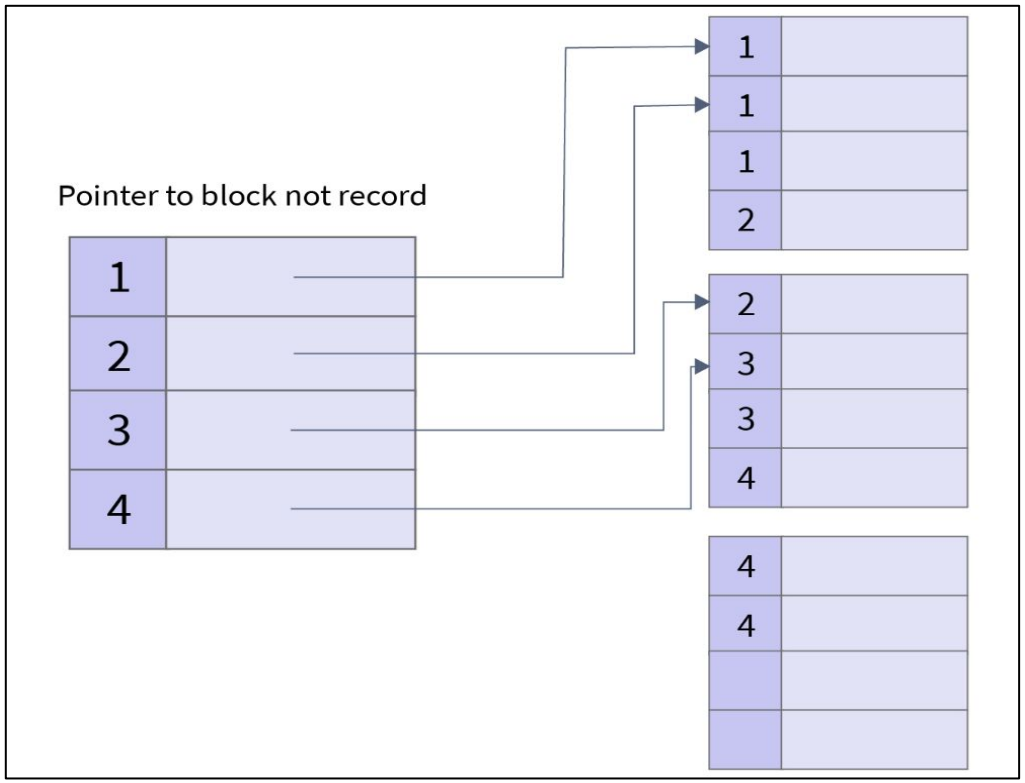


Characteristics of Secondary Indexing:

1. Search Keys are Candidate Keys.
2. Search Keys are sorted but actual data may or may not be sorted.
3. Requires more time than primary indexing.
4. Search Keys cannot be null.
5. Faster than clustered indexing but slower than primary indexing.

Cluster Indexing:

- Clustered Indexing is used when there are multiple related records found at one place.
- It is defined on ordered data.
- The important thing to note here is that the index table of clustered indexing is created using non-key values which may or may not be unique.
- To achieve faster retrieval, we group columns having similar characteristics.
- The indexes are created using these groups and this process is known as Clustering Index.

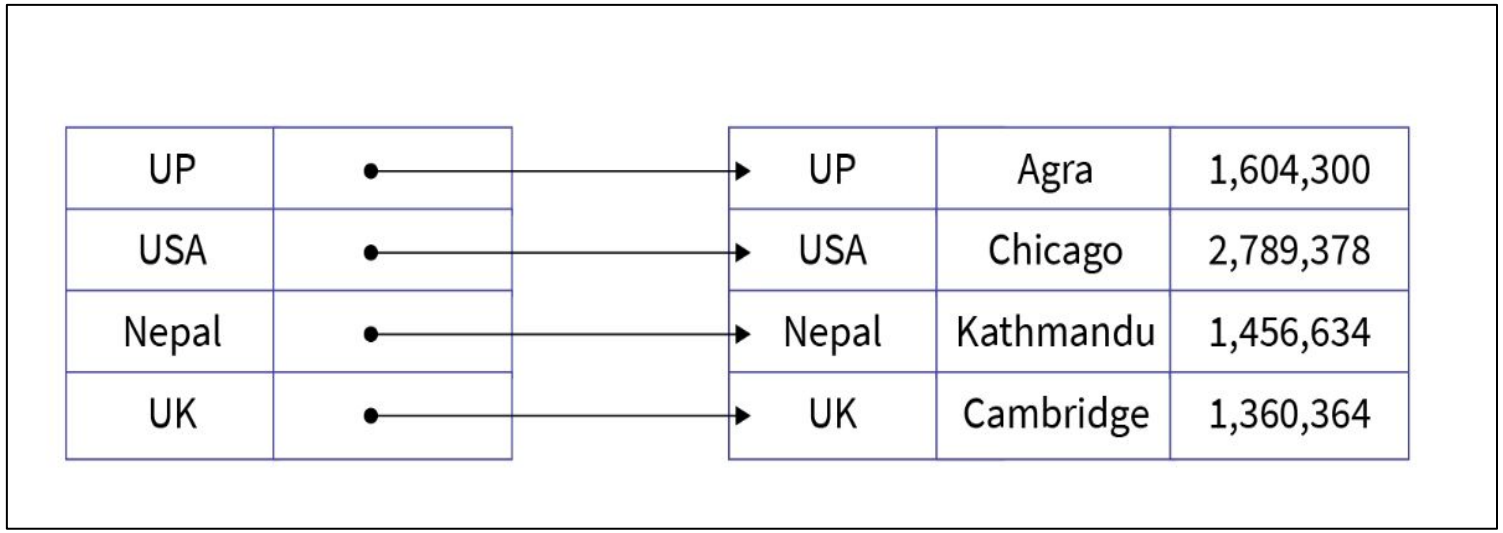


Characteristics of Clustered Indexing:

1. Search Keys are non-key values.
2. Search Keys are sorted.
3. Search Keys cannot be null.
4. Search Keys may or may not be unique.
5. Requires extra work to create indexing.

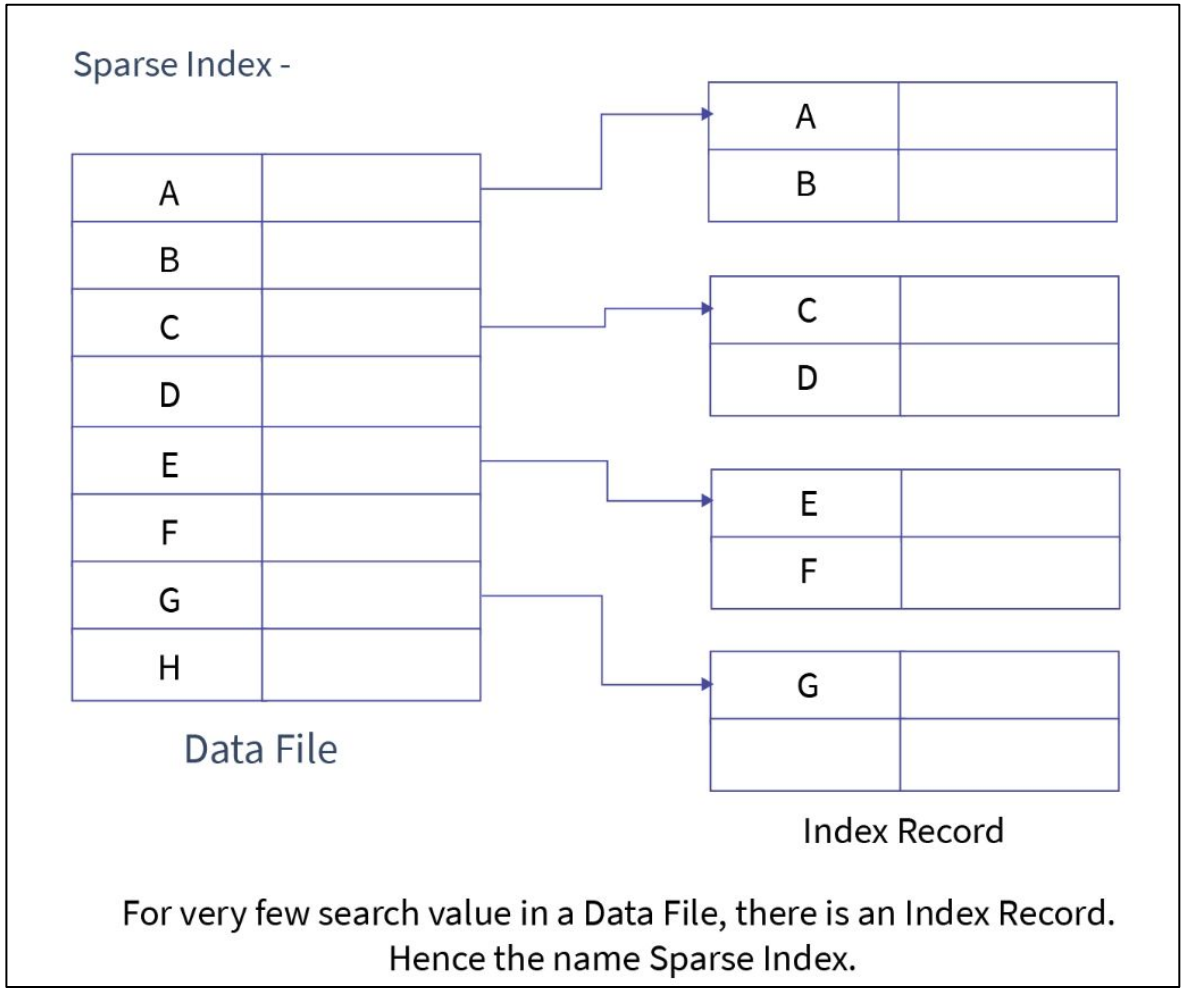
Dense Indexing

- In dense indexing, the index table contains records for every search key value of the database.
- This makes searching faster but requires a lot more space.
- It is like primary indexing but contains a record for every search key.



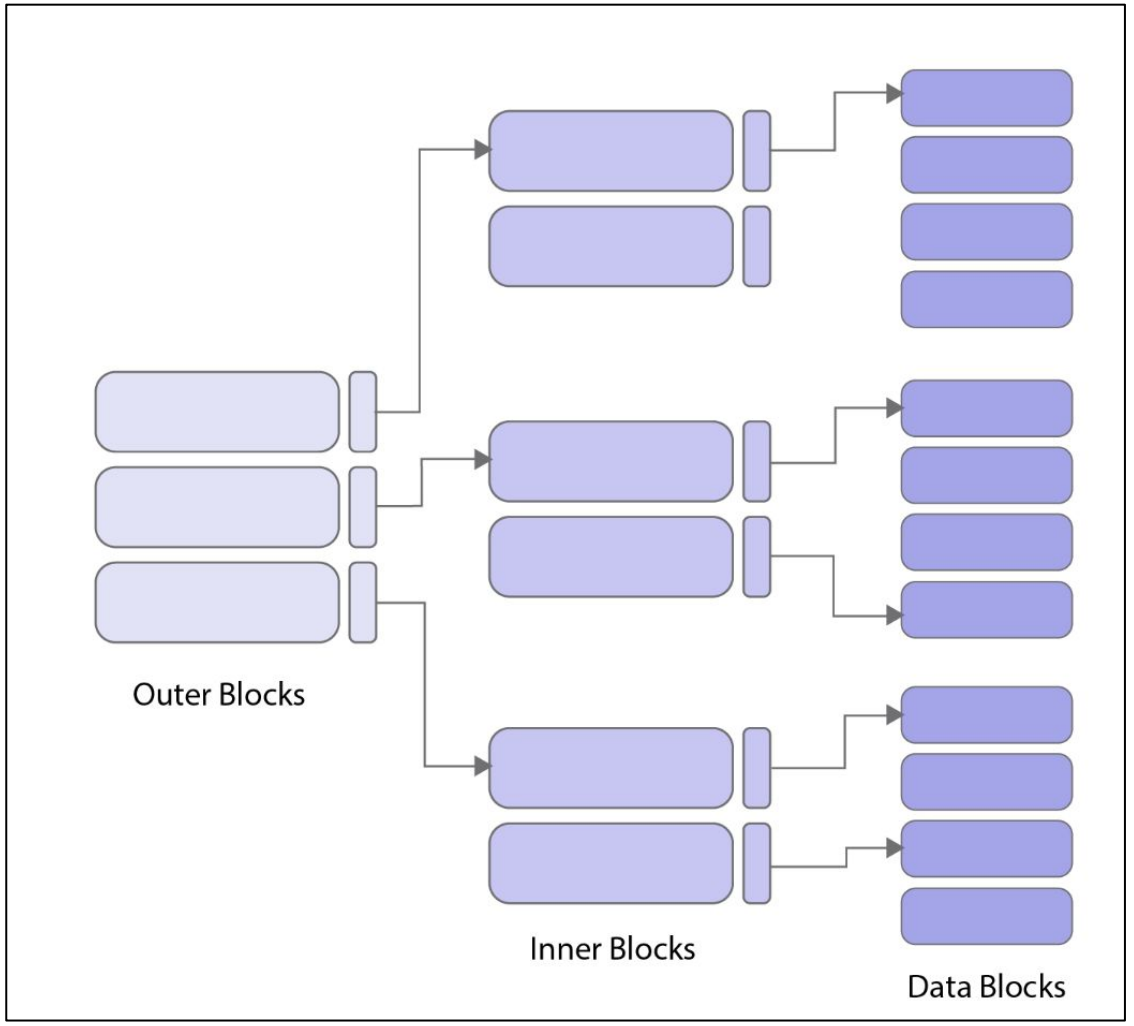
Sparse Indexing

- Sparse indexing consumes lesser space than dense indexing, but it is a bit slower as well.
- We do not include a search key for every record despite that we store a Search key that points to a block.
- The pointed block further contains a group of data. Sometimes we have to perform double searching this makes sparse indexing a bit slower.



Multi-Level Indexing

- Since the index table is stored in the main memory, single-level indexing for a huge amount of data requires a lot of memory space.
- Hence, multilevel indexing was introduced in which we divide the main data block into smaller blocks.
- This makes the outer block of the index table small enough to be stored in the main memory.
- We use the B+ Tree data structure for multilevel indexing.



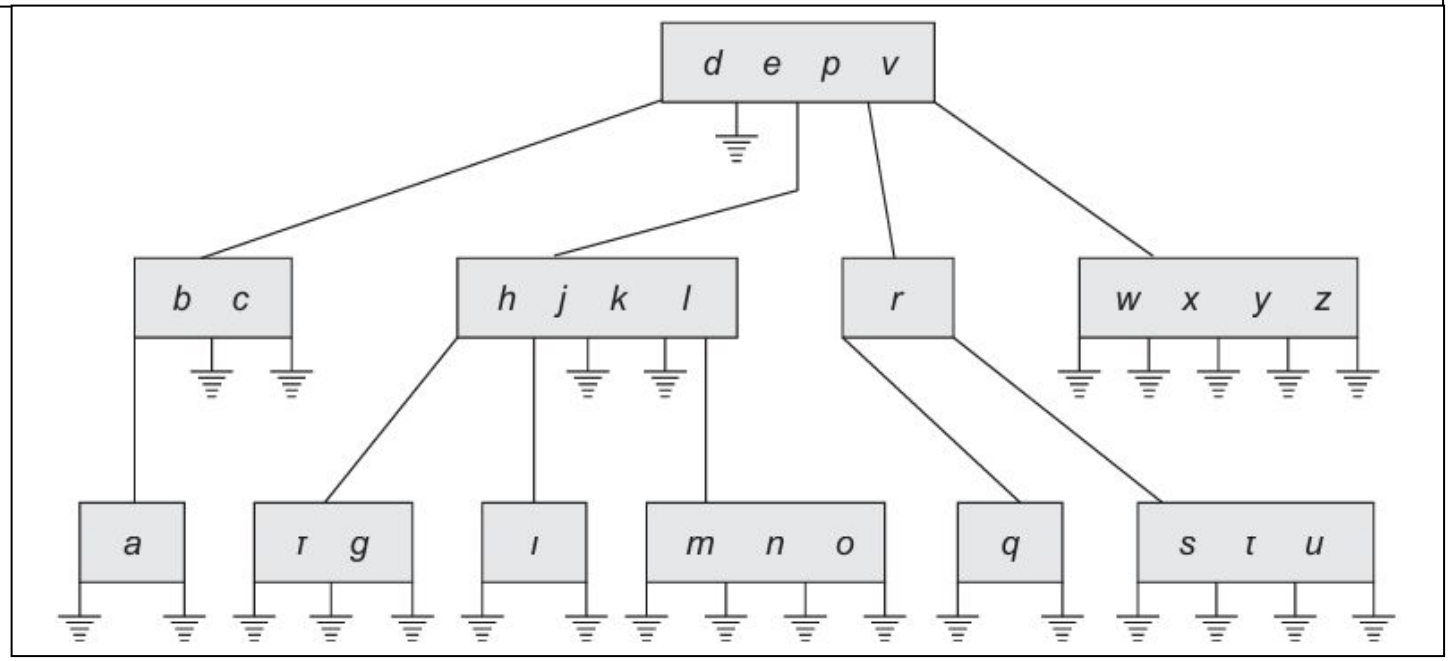
Advantages of Indexing

1. Indexing helps in faster query results or quick data retrieval.
2. Indexing helps in faster sorting and grouping of records
3. Some Indexing uses sorted and unique keys which helps to retrieve sorted queries even faster.
4. Index tables are smaller in size so require lesser memory.
5. As Index tables are smaller in size, they are stored in the main memory.
6. Since CPU speed and secondary memory speed have a large difference, the CPU uses this main memory index table to bridge the gap of speeds.
7. Indexing helps in better CPU utilization and better performance.

Multiway search trees

- Multi-way search trees are a generalized version of binary trees that allow for efficient data searching and sorting.
- A multiway search tree is a tree of **order m,** where each node has **utmost m children.** Here m is an integer.
- If **k ≤ m** is the number of children, then the node contains exactly **k - 1** keys, which partition all the keys in the subtrees into **k** subsets.
- If some of these subsets are empty, then the corresponding children in the tree are empty.

- **Figure shows a 5-way search tree.**
- We always want to construct a multiway search tree that will minimize file accesses.
- So the height of the tree should be as small as possible, **for example, B-tree and B+ tree.**

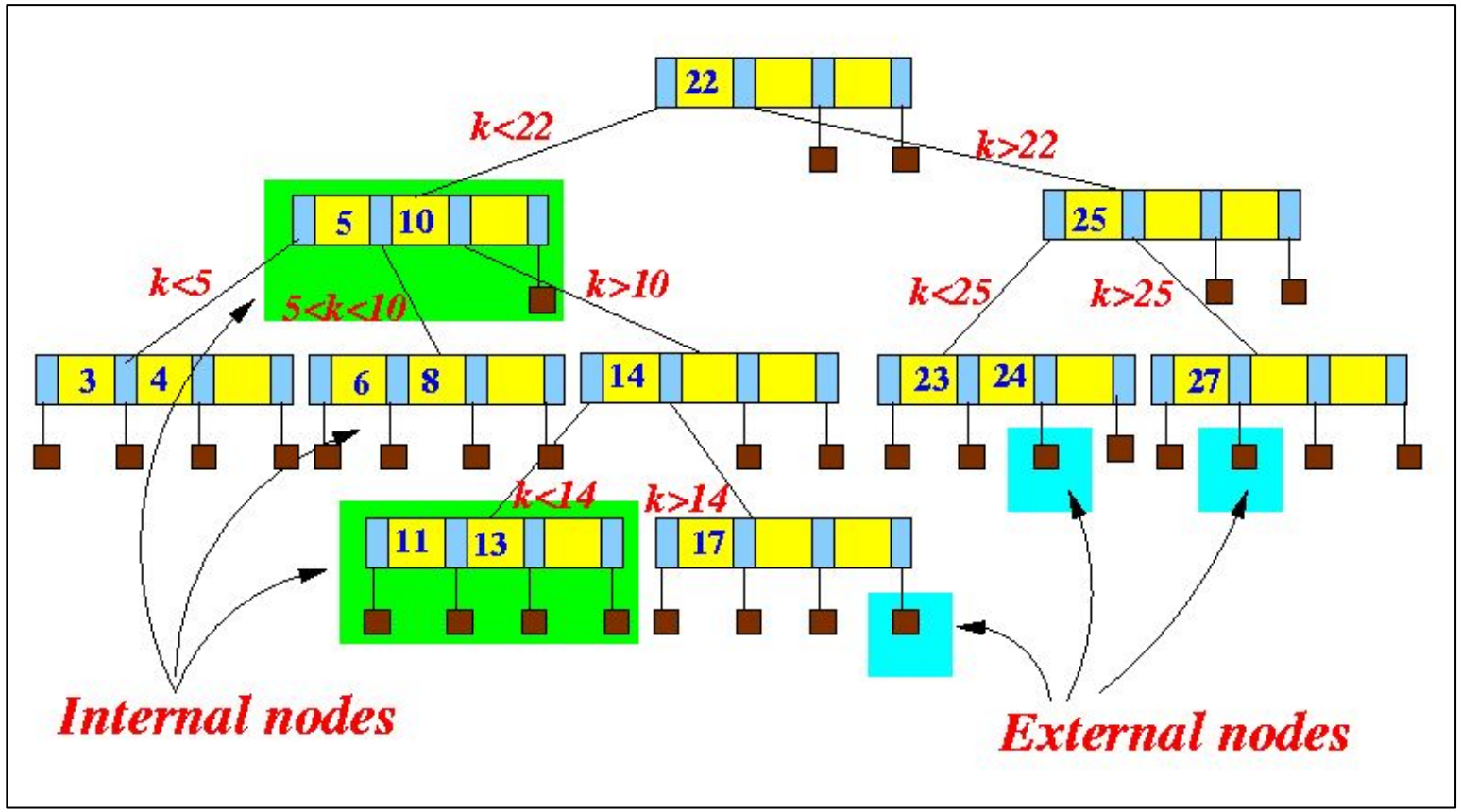


Multiway search trees

It contains the following characteristics:

1. Nodes may carry multiple keys.
2. The tree maintains all leaves at the same level
3. Each node has 0 .. m subtrees.All subtrees are m-way trees.
4. A node with k < m subtrees, contains k-1 keys.
5. The key values of the first subtree are all less than the key value.
6. The data entries are ordered.
7. Due to their structure, M-way trees are mainly used in external searching, i.e. in situations where data is to be retreived from secondary storage like disk files.

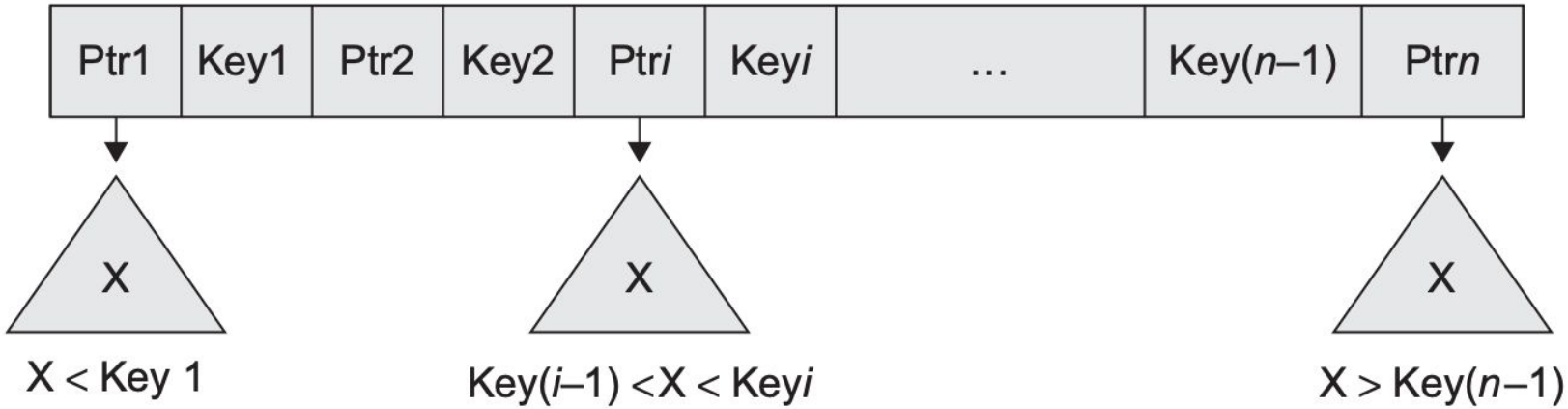
Example: a 4-way tree



B-tree

- A B-tree is a balanced multiway tree.
- A B Tree (height Balanced m-way search Tree) is a special type of M-way tree which balances itself.
- A node of the tree contains many records or keys of records and pointers to children.
- To reduce disk access, the following points are applicable:
    1. Height is kept minimum.
    2. All leaves are kept at the same level.
    3. All nodes other than leaves must have at least minimum number of children.

Node structure of B-tree



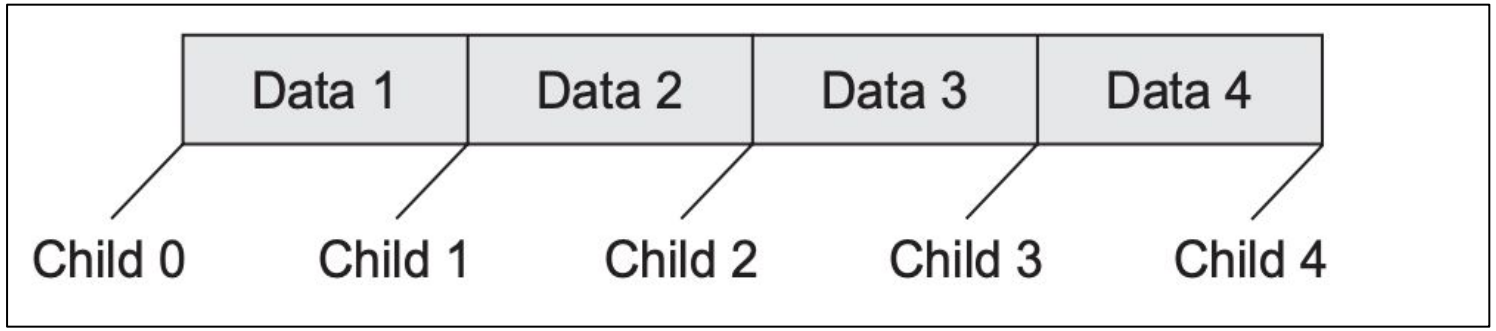
B-tree Definition -

A B-tree of order m is a multiway tree with the following properties:

1. The number of keys in each internal node is one less than the number of its non-empty children, and these keys partition the keys in the children in the fashion of the search tree.
2. All leaves are on the same level.
3. All internal nodes except the root have **utmost(max) m non-empty children and at least (min) [m/2] non-empty children.**
4. The root is either a leaf node, or it has from two to m children.
5. A leaf node contains no more than m - 1 keys.

We will describe a B-tree of order 5 using a C++ structure. The declaration of B-tree node is given in Fig.

Node structure of 5-way B-tree



- The maximum number of items in a B-tree of order m and height h is shown in Table

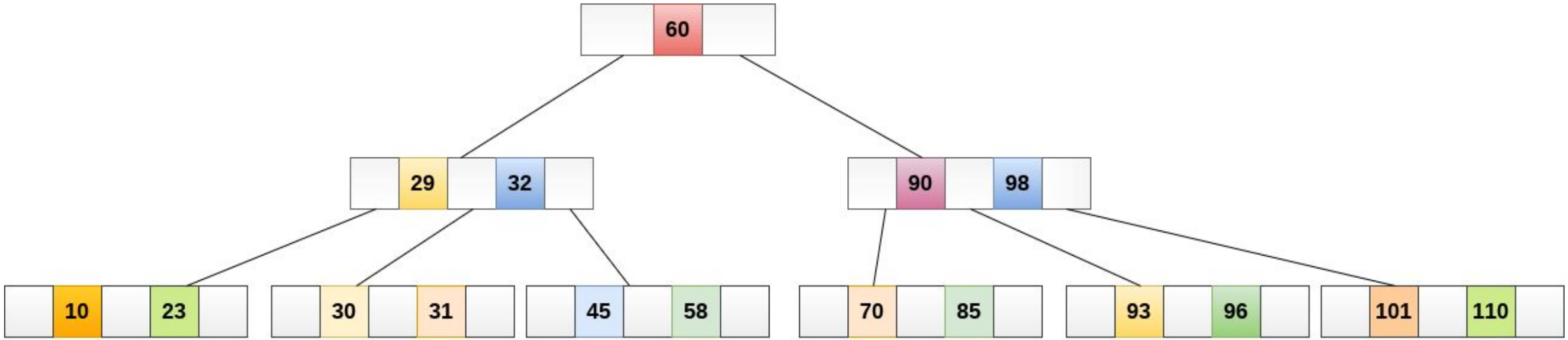
Level	Number of keys
Root	$m - 1$
Level 1	$m(m - 1)$
Level 2	$m^2(m - 1)$
⋮	
Level h	$m^h(m - 1)$

So, the total number of items is

$$(1 + m + m^2 + m^3 + \ldots + m^h)(m - 1) = [(m^{h+1} - 1)/ (m - 1)] (m - 1) = m^{h+1} - 1 \qquad (13.1)$$

When $m = 5$ and $h = 2$, Eq. (13.1) gives $5^3 - 1 = 124$.

B tree of order 3 is shown in the following image.



Search Operation in B-Tree

- The search operation in B-Tree is similar to the search operation in Binary Search Tree.
- In a Binary search tree, the search process starts from the root node and we make a 2-way decision every time (we go to either left subtree or right subtree).
- In B-Tree also search process starts from the root node but here we make an n-way decision every time. Where 'n' is the total number of children the node has.
- In a B-Tree, the search operation is performed with O(log n) time complexity.
- The search operation is performed as follows...
    - **Step 1 -** Read the search element from the user.
    - **Step 2 -** Compare the search element with first key value of root node in the tree.
    - **Step 3 -** If both are matched, then display "Given node is found!!!" and terminate the function
    - **Step 4 -** If both are not matched, then check whether search element is smaller or larger than that key value.
    - **Step 5 -** If search element is smaller, then continue the search process in left subtree.
    - **Step 6 -** If search element is larger, then compare the search element with next key value in the same node and repeate steps 3, 4, 5 and 6 until we find the exact match or until the search element is compared with last key value in the leaf node.
    - **Step 7 -** If the last key value in the leaf node is also not matched then display "Element is not found" and terminate the function.

Insert Operation on B Tree in Data Structure

- **Binary search trees grow at their leaves, but the B-trees grow at the root.**
- The general method of insertion is as follows:
  1. First, the new key is searched in the tree. If the new key is not found, then the search terminates at a leaf.
  2. Attempt to insert the new key into a leaf.
  3. If the leaf node is not full, then the new key is added to it and the insertion is finished.
  4. If the leaf node is full, then it splits into two nodes on the same level, except that the median key is sent up the tree to be inserted into the parent node.
  5. If this would result in the parent becoming too big, split the parent into two, promoting the middle key.
  6. This strategy might have to be repeated all the way to the top.
  7. If necessary, the root is split into two and the middle key is promoted to a new root, making the tree one level higher.

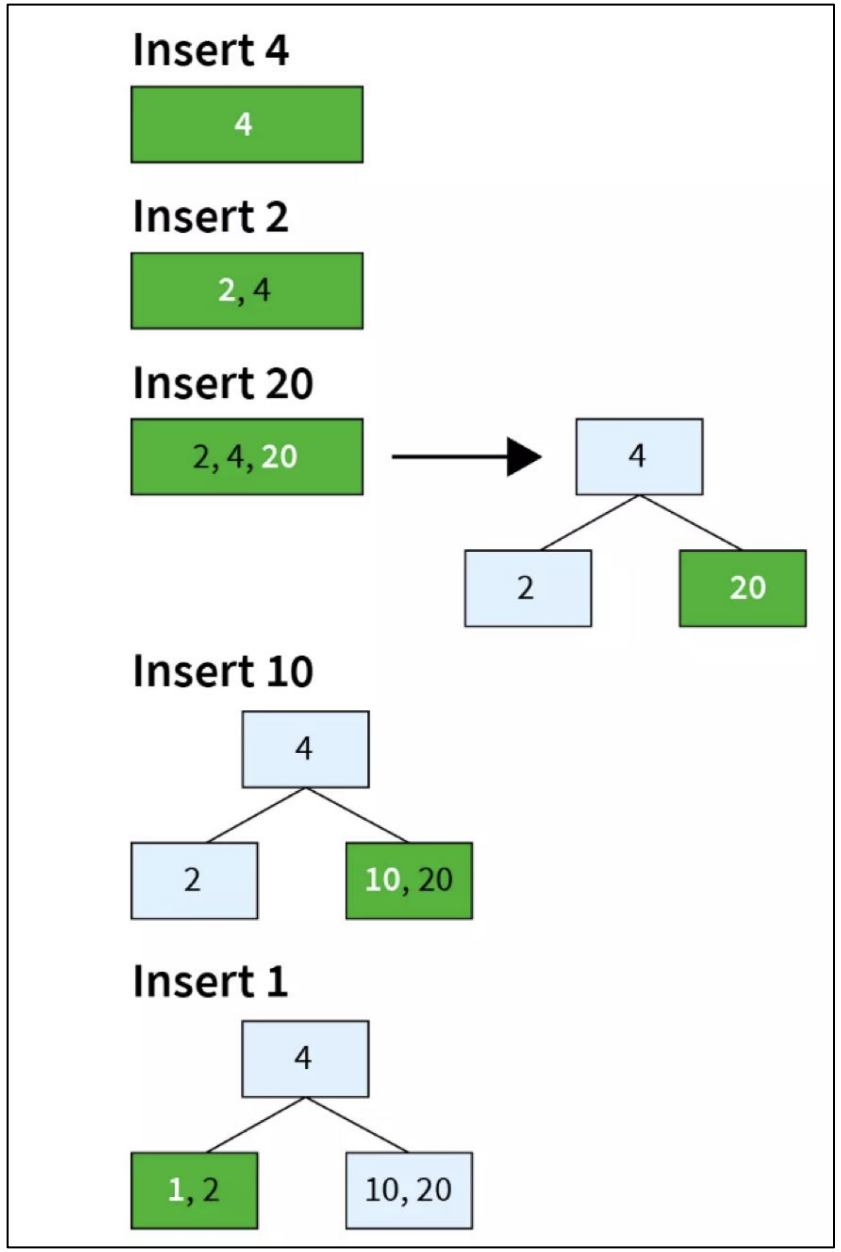
Steps to insert an element in B Tree in Data Structure

1. Calculate the maximum number of keys in the node based on the order of the B tree.
2. If the tree is empty, a root node is allocated and the key is inserted and acts as the root node.
3. Search the appropriate node for insertion.`
4. If the node is full:
    1. Insert the elements in increasing order.
    2. If the elements are greater than the maximum number of keys, split at the median.
    3. Push the median key upwards and split the left and right keys as left and right child respectively.
5. If the node is not full, insert the node in increasing order.

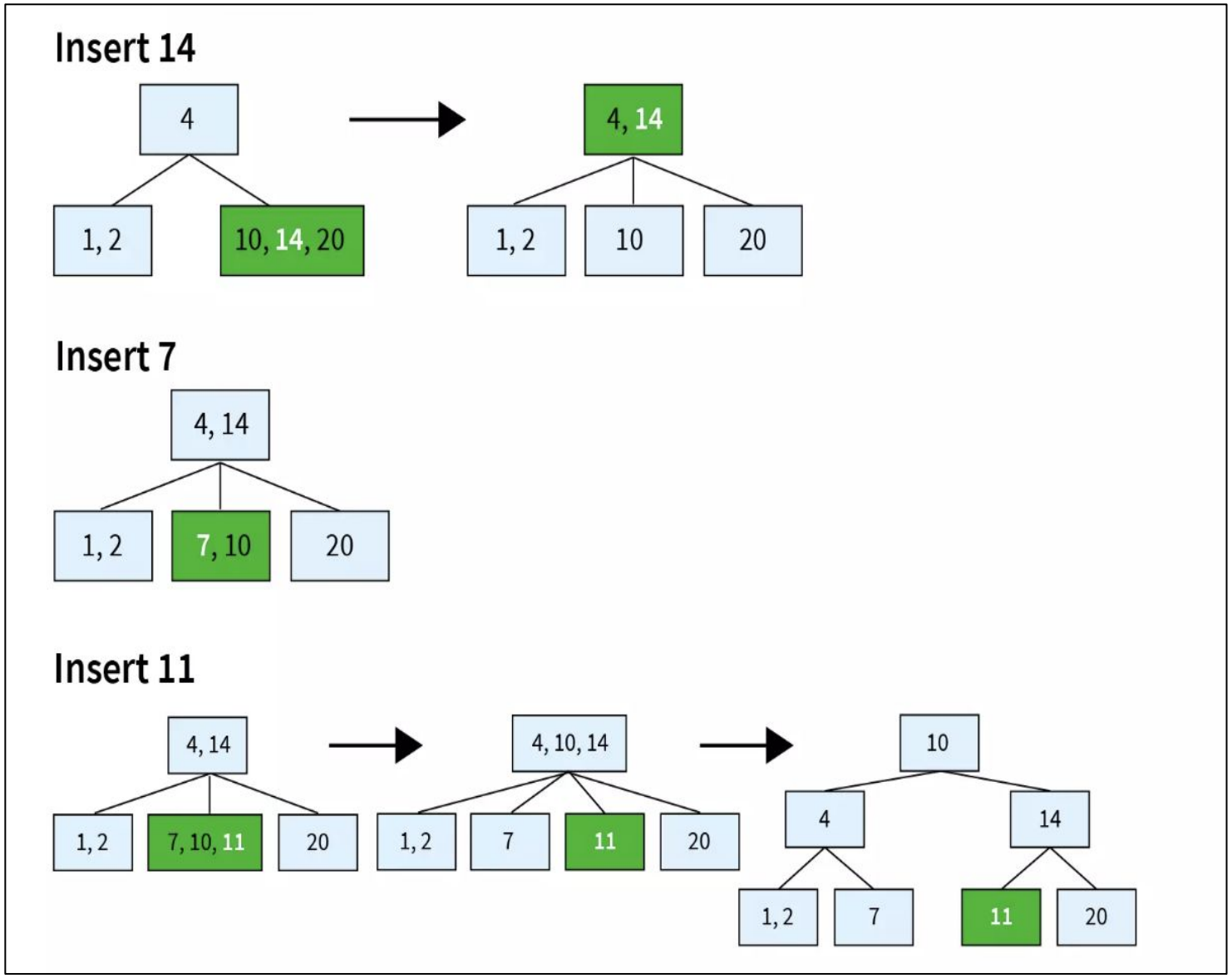
Suppose we have to create a B tree of order 3. The elements to be inserted are 4, 2, 20, 10, 1, 14, 7, 11, 3, 8.

Since m=3, max number of keys for a node = m-1 = 2.

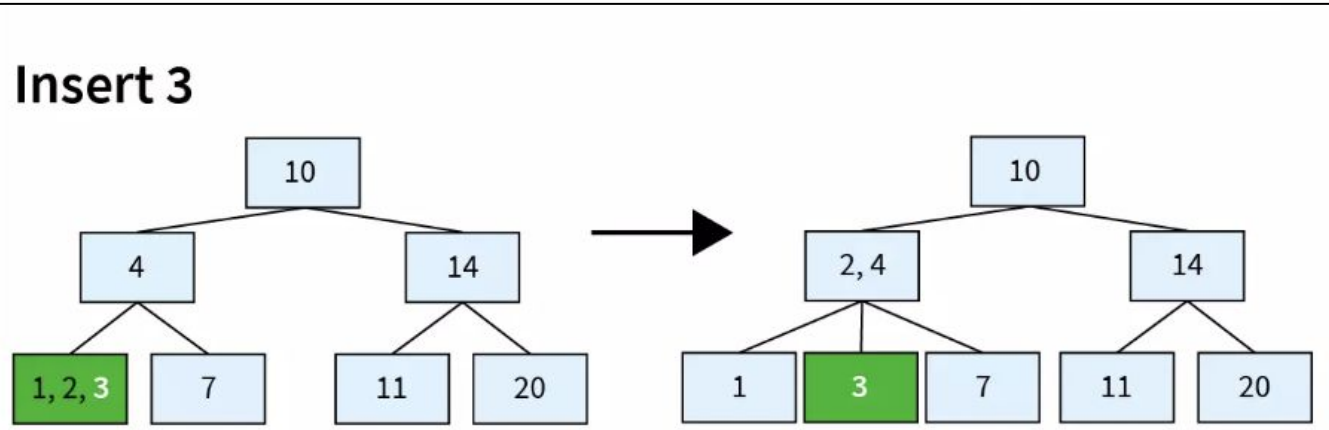
- **Insert 4**
- **Since 2<4**, insert 2 to the left of 4 in the same node.
- **Since 20>4,** insert 20 to the right of 4 in the same node. As we now, maximum number of keys in the node are 2, one of these keys will have to be moved to a node above to split it. 4 being the middle element will move up and 2 and 20 will be its left and right nodes respectively.
- **10>4 and 10<20** and thus, 10 will be inserted as a key in the node that contains 20 as a key.
- **Since 1<2,** it will be inserted as a key in the node that contains 2 as a key.

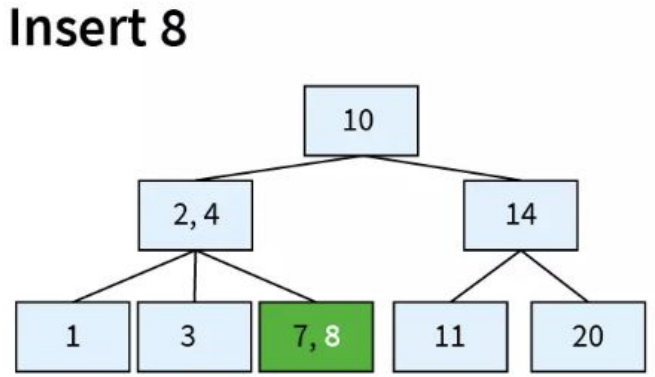


- **14>10 and 14<20.** Since the number of keys in that node exceeds the maximum number of keys, the node will split after the middle key moves upto the node in the above line. Thus, 14 gets added to the right of 4 in the node that contains 4, and 10 and 20 are split as 2 separate nodes.
- **Since 7<10**, it gets inserted to the left in the node that contains 10 as a key.
- **11<14 and 11>10**. Thus, 11 should get added to the right of the node that contains 7 and 10. However, since the maximum number of keys in the tree are 2, a split should take place. Thus, the middle element 10 moves to the above node and 7 and 11 split as separate nodes. The above node now contains 4, 10 and 14. Since the count of keys exceeds the maximum key count, there would be a split there. Now, 10 is the root node with 4 and 14 as its children.



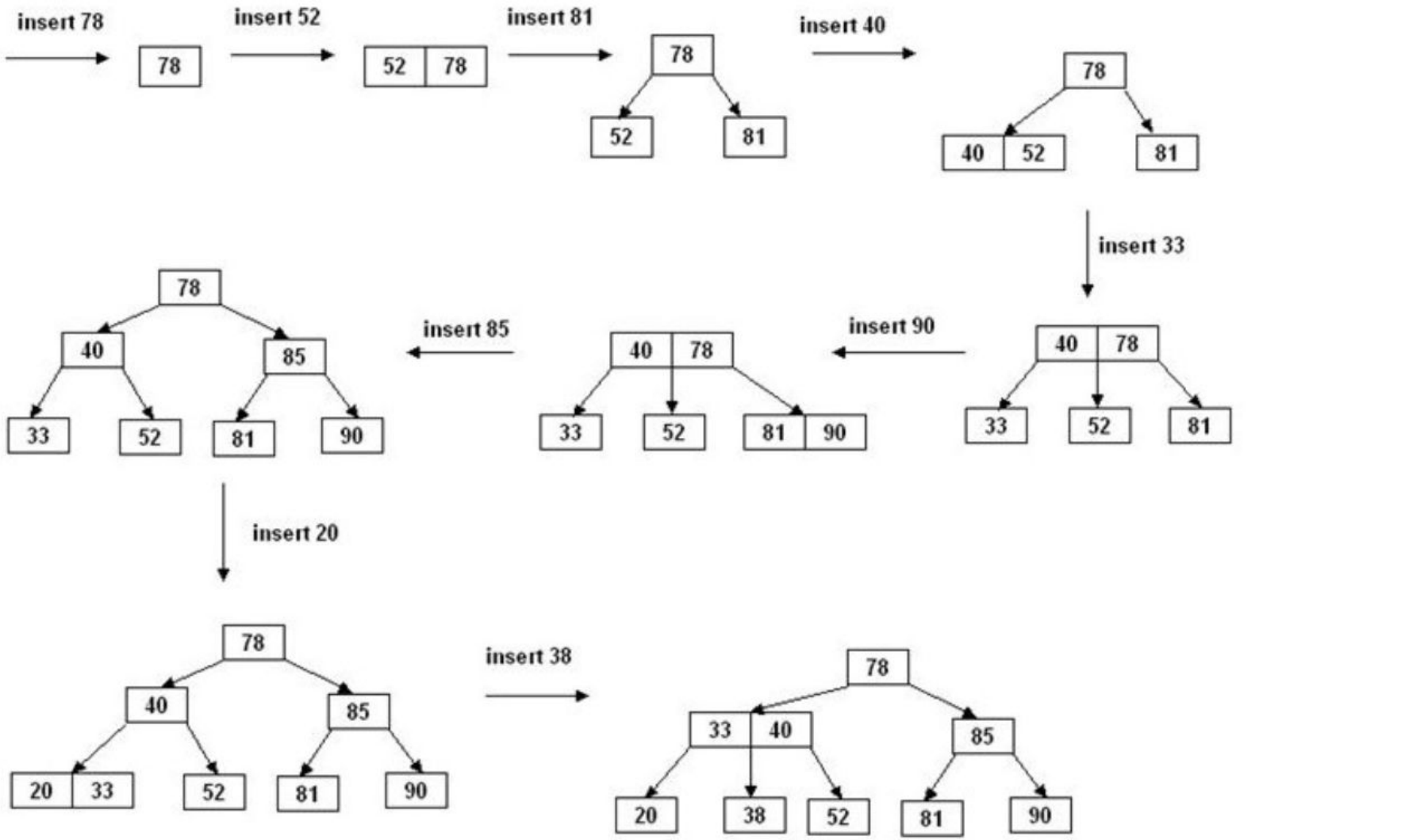
- **Since 3<4 and 3>2**,it gets inserted to the right of the node containing 1, 2. This node exceeds the maximum count of keys in a node, leading to a split. 2 is added to the upper node beside 4.
- **Since 8>7 and 8<10**, it gets added to the left of the node that contains 7 as a key.





The time complexity for insertion in a B Tree is dependent on the number of nodes and thus, **O(log n).**

- Example: Insert the keys 78, 52, 81, 40, 33, 90, 85, 20, and 38 in this order in an initially empty B-tree of order 3



Insertion

1) Initialize x as root.

2) While x is not leaf, do following

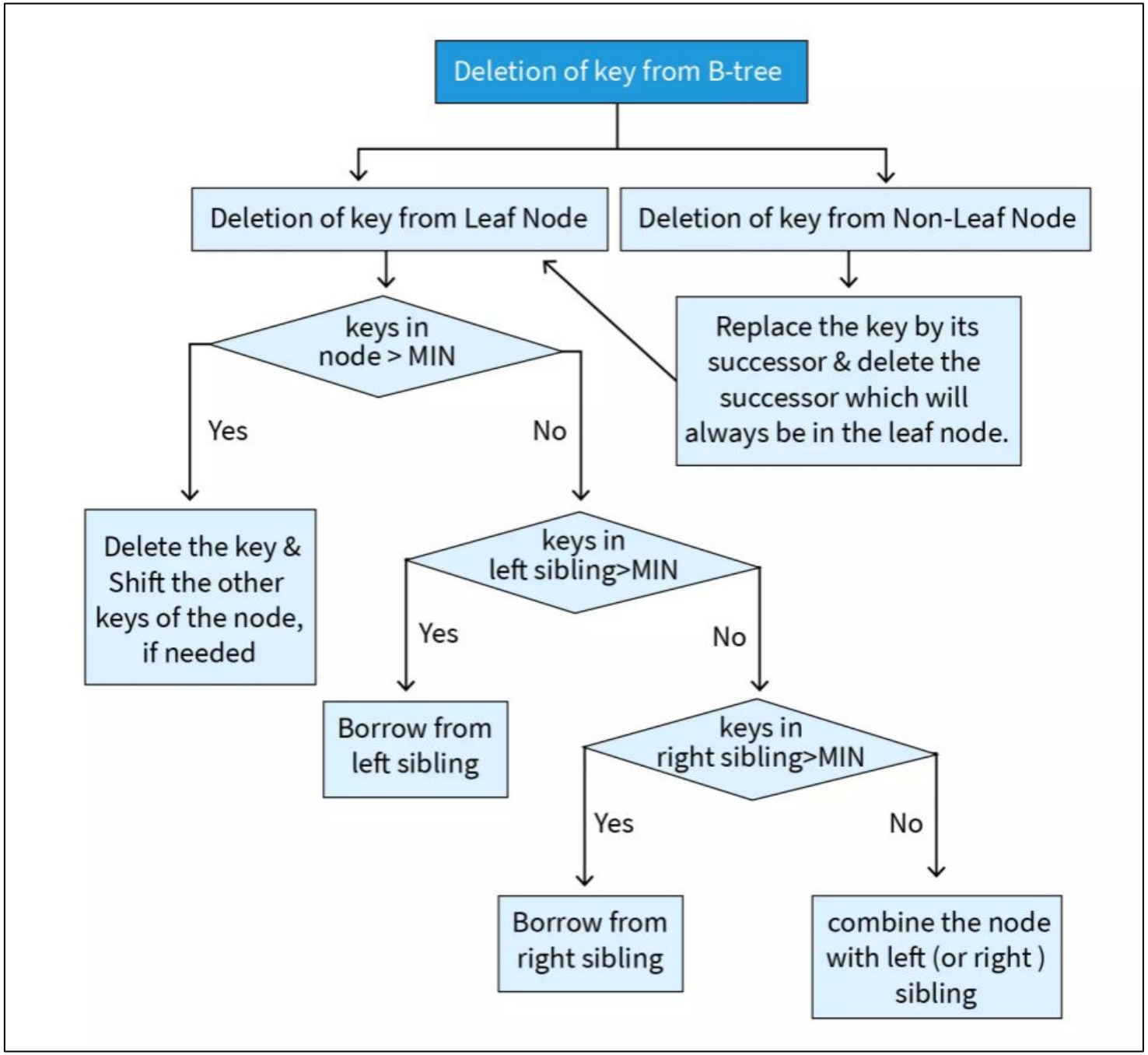
    1. Find the child of x that is going to be traversed next. Let the child be y.
    2. If y is not full, change x to point to y.
    3. If y is full, split it and change x to point to one of the two parts of y. If k is smaller than mid key in y, then set x as the first part of y. Else second part of y. When we split y, we move a key from y to its parent x.

3) The loop in step 2 stops when x is leaf. x must have space for 1 extra key as we have been splitting all nodes in advance. So simply insert k to x.

Deletion Operation on B Tree in Data Structure

- During insertion, we had to ensure that the number of keys in the node doesn't cross a maximum.
- Similarly, during deletion, we need to ensure that the number of keys in the node after deletion doesn't go below the minimum number of keys that a node can hold.
- Thus, in case of deletion, a combination process takes place instead of a split.
- Steps to delete an element in B Tree in Data Structure
- Deletion can be studied using 2 cases:
    1. Deletion from a leaf node.
    2. Deletion from a non-leaf node.

- The 2 cases can be represented by the following flowchart:



Case 1 - Deletion from Leaf Node

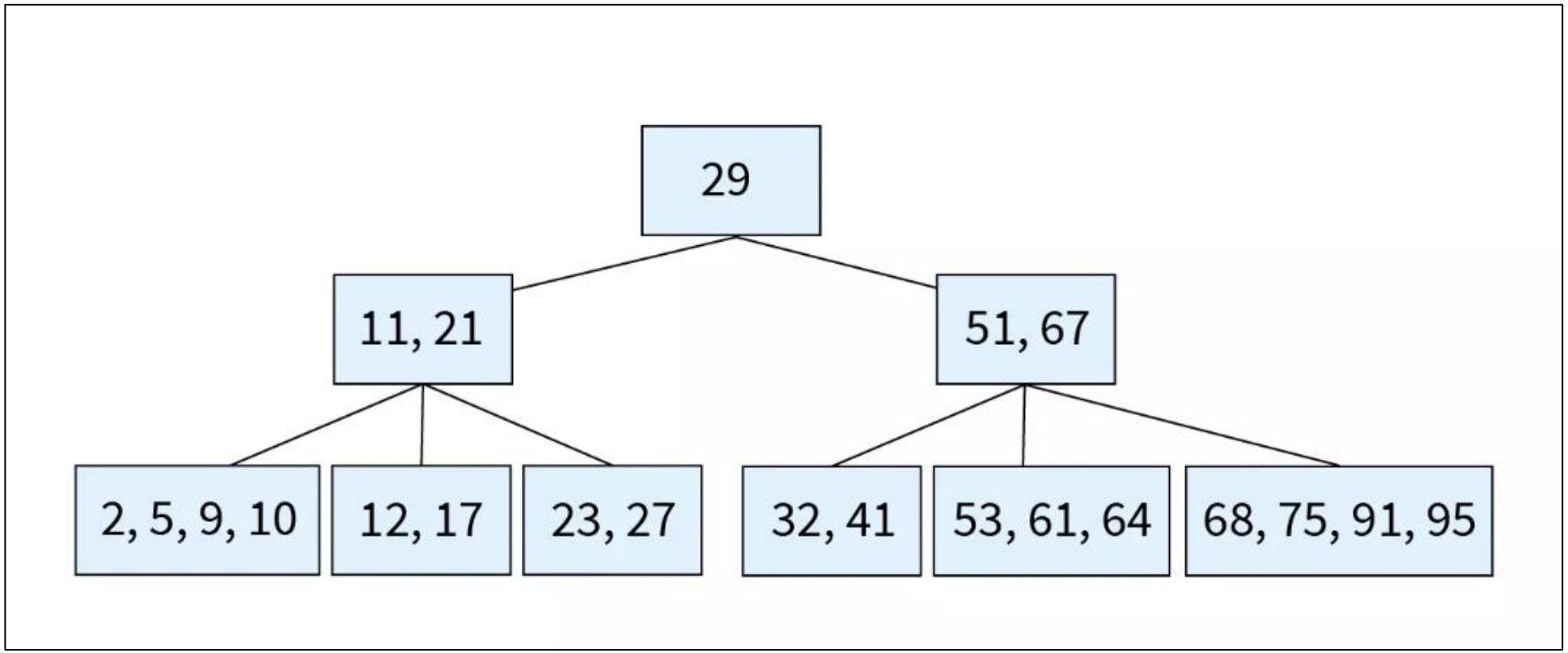
1.1 If the node has more than MIN keys - Deletion of key does not violate any property and thus the key can be deleted easily by shifting other keys of the node, if required.

1.2 If the node has less than MIN keys - This kind of deletion violates a property of B tree. In case the keys in the left sibling are greater than MIN, keys are borrowed from there. If the keys in right sibling are greater than MIN, then keys are borrowed from there. If either of these do not hold true, then a combination of node takes place with either of the siblings.

Case 2 - Deletion from Non-Leaf Node

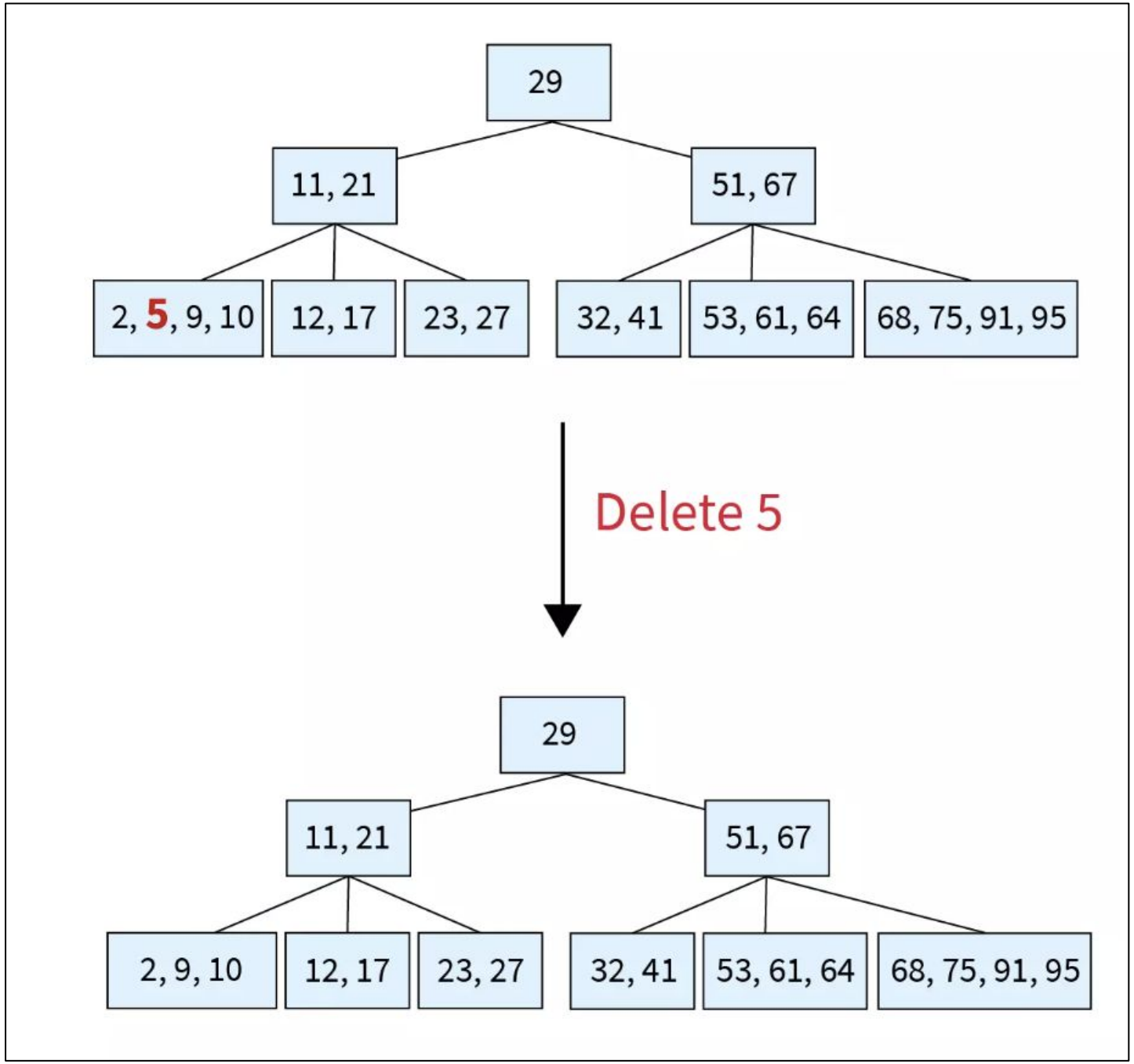
2.1 In this case, the successor key (smallest key in the right subtree) is copied at the place of the key to be deleted and then the successor is deleted. This case further reduces to Case 1, i.e. deletion from a leaf node.

- Let's understand deletion using an example. Consider the following B tree of order 5 with the nodes 5, 12, 32 and 53 to be deleted in the given order.
- Since the order of the B tree is 5, the minimum and maximum number of keys in a node are 2 and 4 respectively.



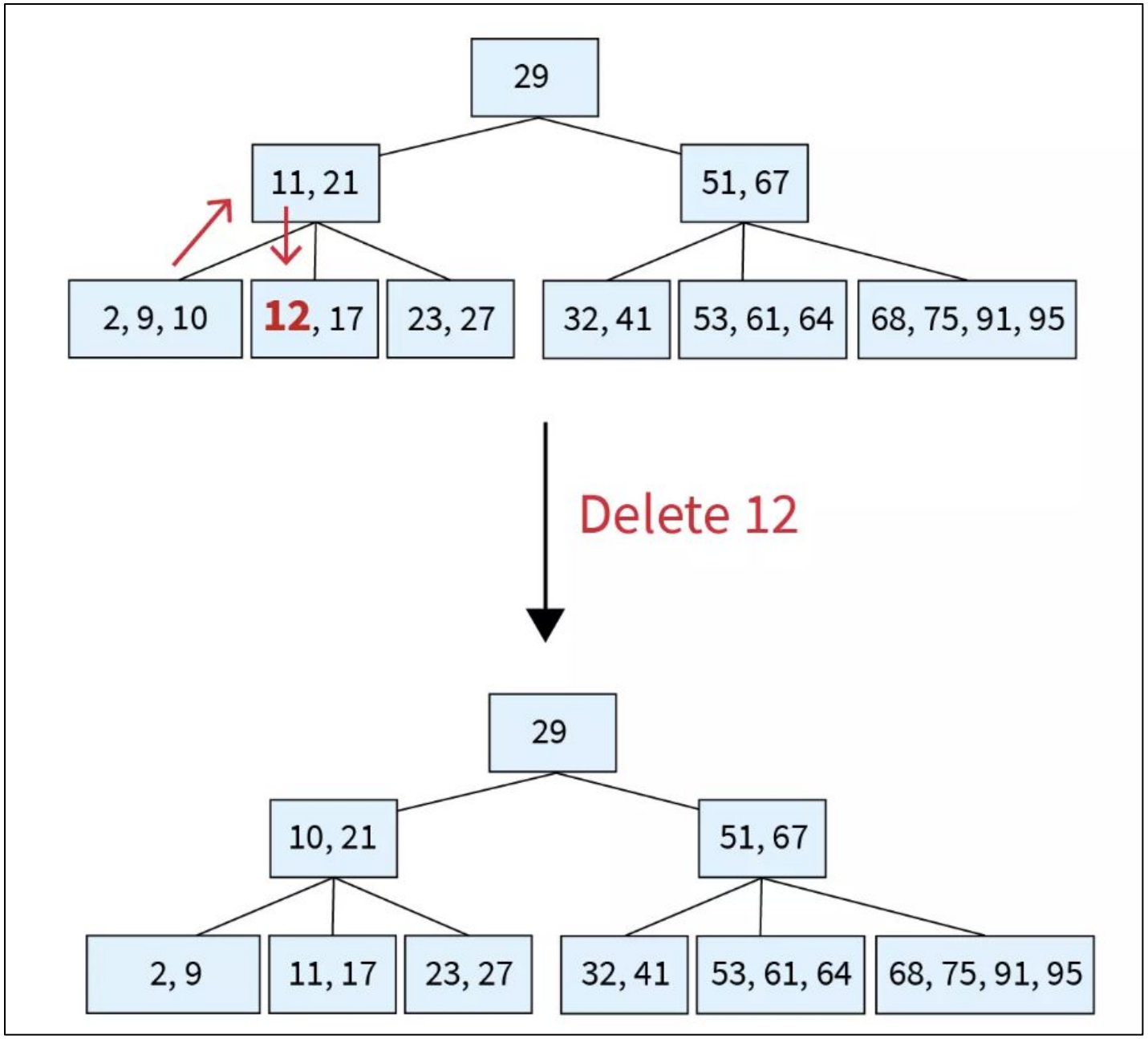
Step 1 - Deleting 5

Since 5 is a key in a leaf node with keys>MIN, this would be Case 1.1. A simple deletion with key shift would be done. Keys 9 and 10 are shifted left to fill the gap created by the deletion of 5.



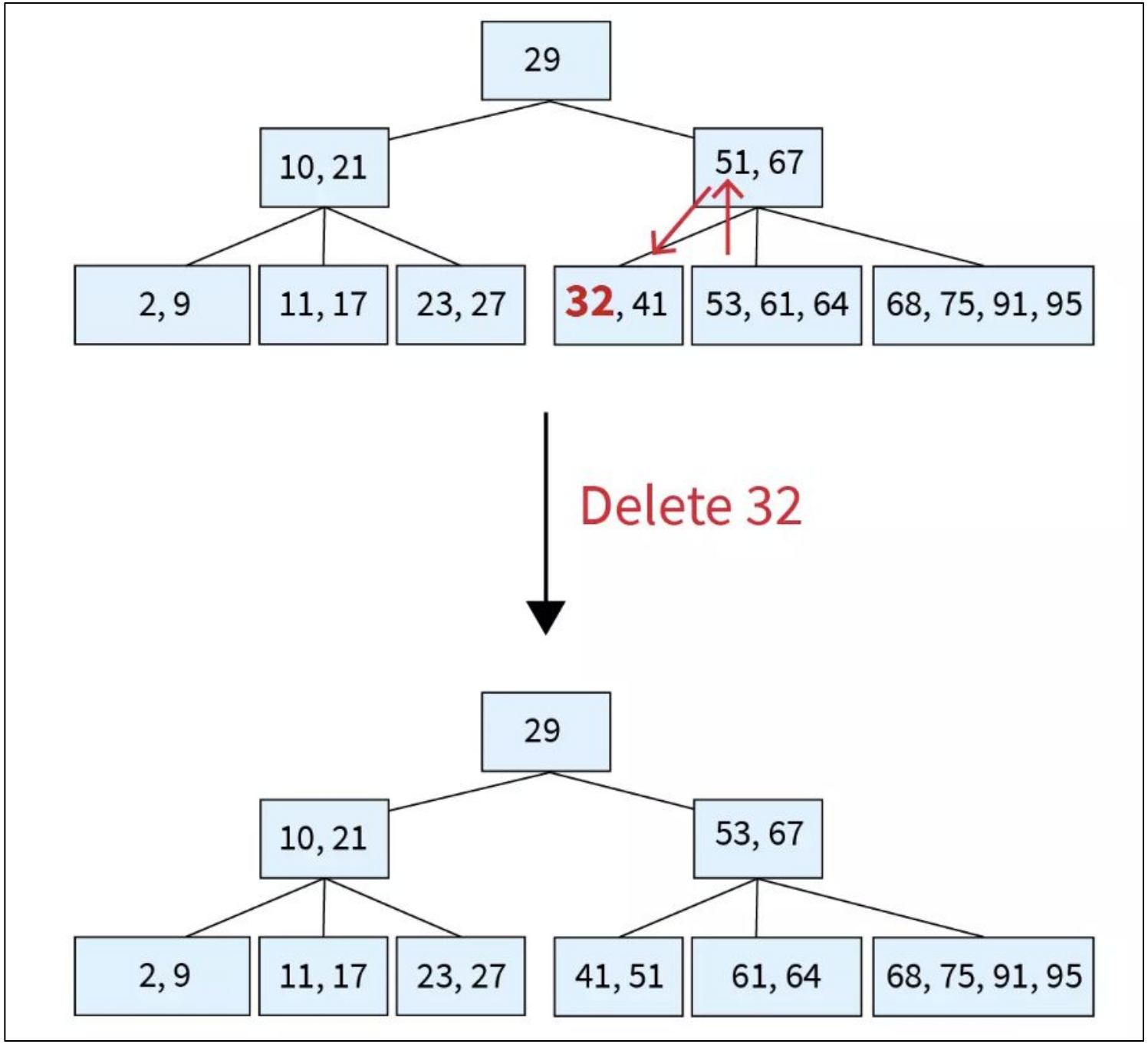
Step 2 - Deleting 12

- Here, key 12 is to be deleted from node [12,17]. Since this node has only MIN keys, we will try to borrow from its left sibling [2,9,10] which has more than MIN keys. The parent of these nodes [11,21] contains the separator key 11. So, the last key of left sibling (10) is moved to the place of the separator key and the separator key is moved to the underflow node (the node where deletion took place). The resulting tree after deletion can be found as follows:



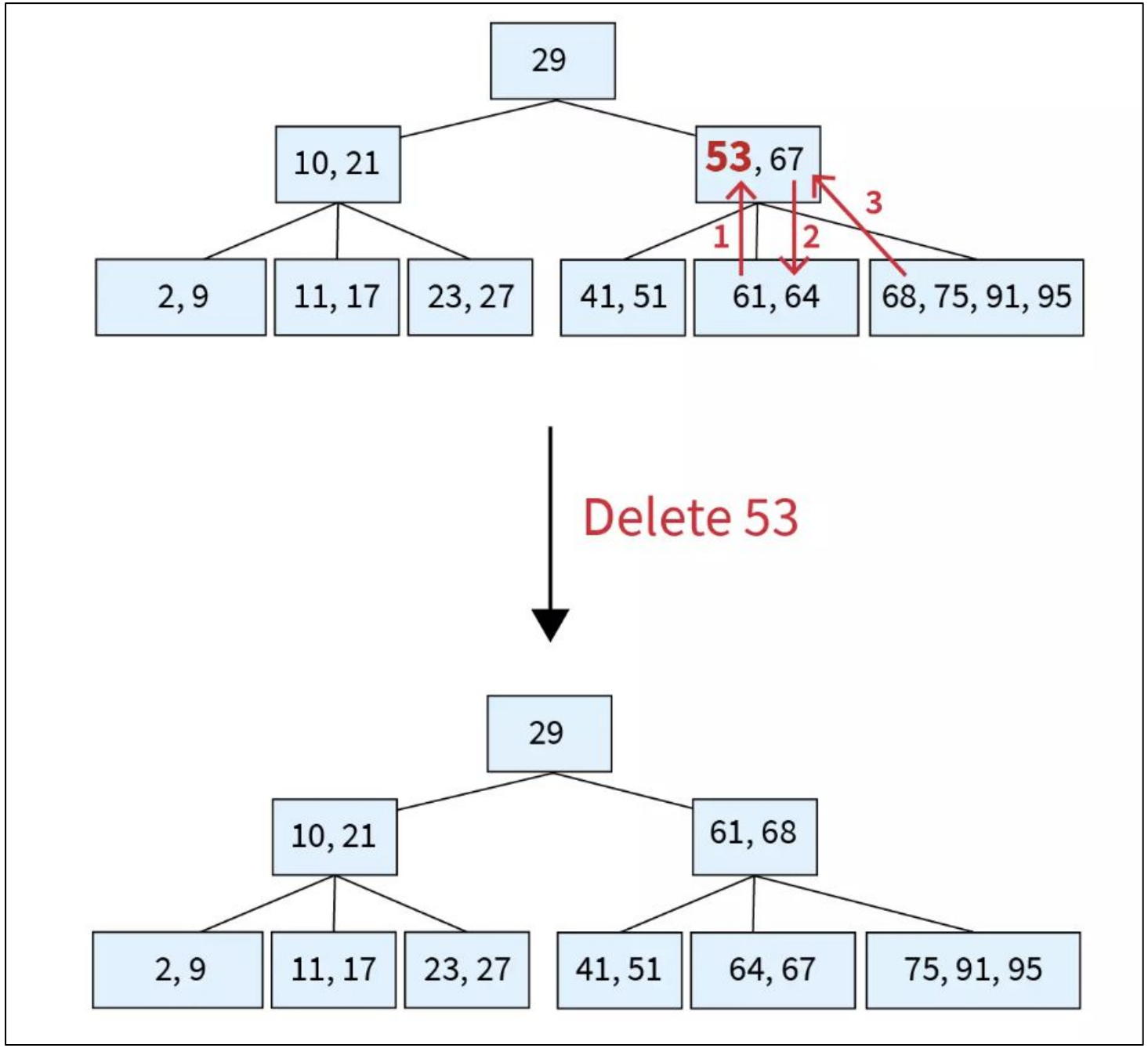
Step 3 - Deleting 32

- Here, key 32 is to be deleted from node [32, 41]. Since this node has only MIN keys and does not have a left sibling, we will try to borrow from its right sibling [53, 61, 64] which has more than MIN keys. The parent of these nodes [51, 67] contains the separator key 51. So, the first key of right sibling (53) is moved to the place of the separator key and the separator key is moved to the underflow node (the node where deletion took place). The resulting tree after deletion can be found as follows:

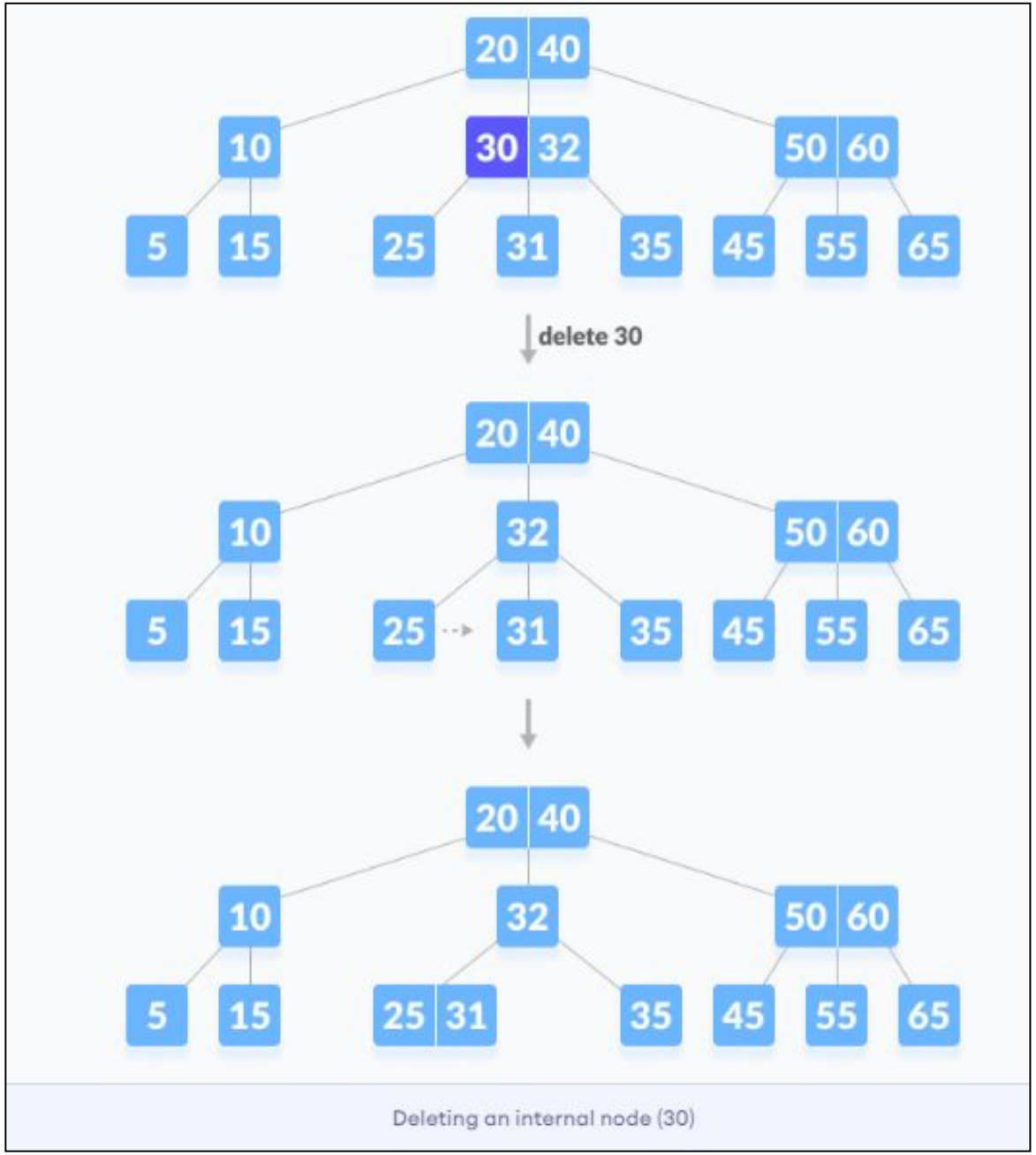


Step 4 - Deleting 53

Here key 53 is to be deleted from node [53, 67] which is a non-leaf node. In such a case, the successor key (61) will be copied in place of 53 and now the task reduces to deletion of 61 from the leaf node. Since this node would have less than MIN keys, we check for the left sibling. Since the left sibling has only MIN keys, we move to the right sibling. The leftmost key of the right sibling (68) moves to the parent node and replaces the separator (67) while the separator shifts to the underflow node making it [64,67]. The resulting tree after deletion can be found as follows:



- The internal node, which is deleted, is replaced by an inorder successor if the right child has more than the minimum number of keys.
- If either child has exactly a minimum number of keys then, merge the left and the right children.
- After merging if the parent node has less than the minimum number of keys then, look for the siblings as in Case I.



Deleting an internal node (30)

B-tree Insertion Algorithm

Procedure Insert(key):

If the root is empty:

1. Create a new root node.
2. Add the key to the root.

Otherwise:

- If the root is full (contains ORDER - 1 keys):
  1. Create a new node.
  2. Set the new node as the parent of the current root.
  3. Split the current root and move the median key to the new parent.
  4. Set the root to the new parent.
- Recursively insert the key into the B-tree starting from the root using the InsertNonFull function.

Procedure InsertNonFull(node, key):

- Find the appropriate position to insert the key in the node's keys array.
- If the node is a leaf:
    - Insert the key into the appropriate position in the node's keys array.
- If the node is not a leaf:
    - Find the appropriate child node to descend into.
    - If the child node is full (contains ORDER - 1 keys):
        - Split the child node.
        - Decide which child of the current node to descend into based on the value of the key.
    - Recursively call InsertNonFull on the selected child node with the key.

Procedure Split(parent, childIndex):

- Create a new node (newNode).
- Copy the upper half of the keys from the child node to the newNode.
- If the child node is not a leaf:
    - Copy the corresponding children of the child node to the newNode.
- Insert the median key from the child node into the parent's keys array at the appropriate position.
- Insert the newNode into the parent's children array at the appropriate position.

B-tree Deletion Algorithm

Procedure Remove(key):

- If the root is empty, print "Tree is empty" and return.
- Call the Remove function starting from the root with the given key.
- If the root becomes empty after deletion, adjust the root.

Procedure Remove(node, key):

- Search for the key in the node:
  - If found:
    - If the node is a leaf, remove the key from the node's keys array.
    - If the node is not a leaf:
      - Find the predecessor of the key from the left subtree (or successor from the right subtree).
      - Replace the key with its predecessor (or successor).
      - Recursively call Remove on the child node from which the predecessor (or successor) was obtained.
  - If not found:
    - If the node is a leaf, print "Key not found" and return.
    - If the node is not a leaf:
      - If necessary, borrow a key from a sibling or merge with a sibling.
      - Recursively call Remove on the appropriate child node.

Procedure BorrowFromLeft(node, index):

- Get the child node (child) at index from the parent node.
- Get the left sibling node (sibling) of the child node from the parent node.
- Move the rightmost key from the sibling node to the left of the child node.
- If the child node is not a leaf, adjust the corresponding child pointers.

Procedure BorrowFromRight(node, index):

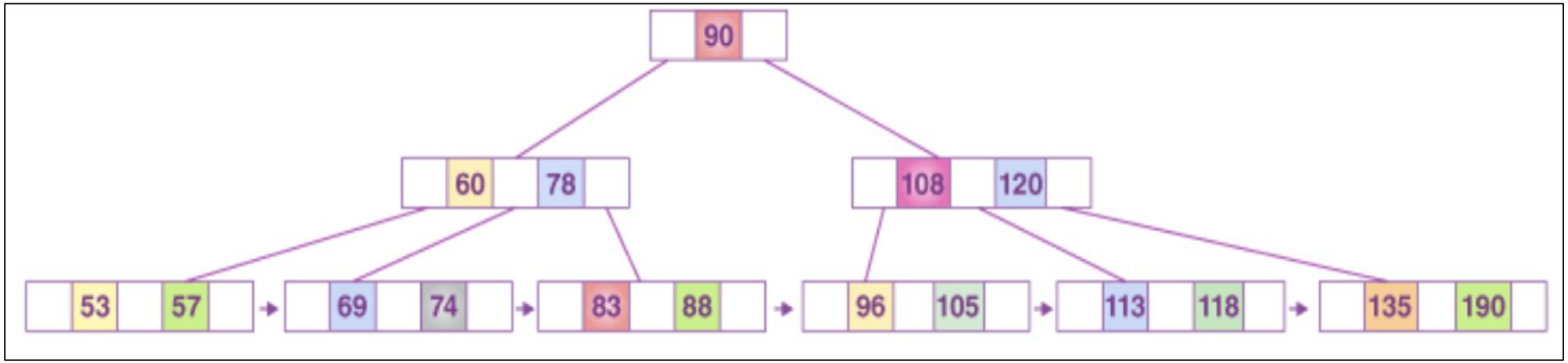
- Get the child node (child) at index from the parent node.
- Get the right sibling node (sibling) of the child node from the parent node.
- Move the leftmost key from the sibling node to the right of the child node.
- If the child node is not a leaf, adjust the corresponding child pointers.

Procedure Merge(node, index):

- Get the child node (child) at index from the parent node.
- Get the sibling node (sibling) next to the child node.
- Merge the keys and children of the sibling node into the child node.
- Remove the key from the parent node.
- Remove the sibling node from the parent node and free its memory.

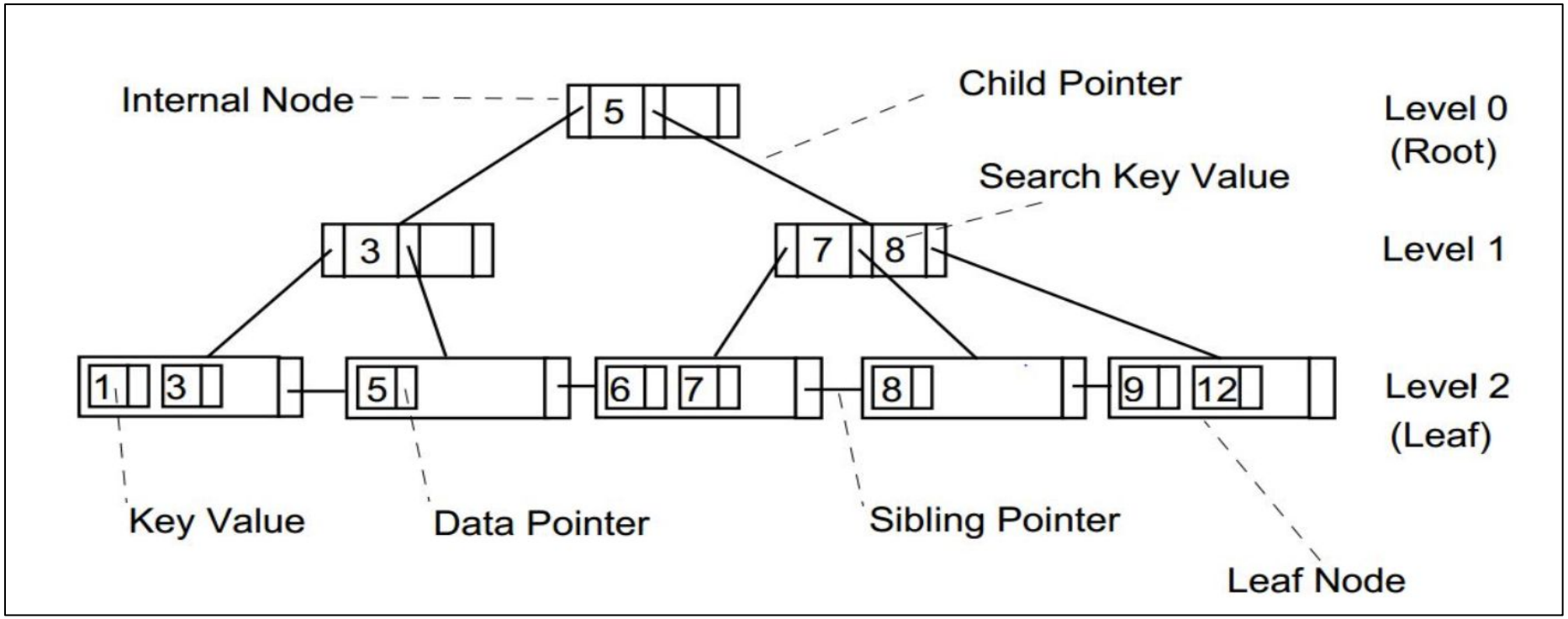
B+Tree

- A B+ Tree is simply a balanced binary search tree, in which all data is stored in the leaf nodes, while the internal nodes store just the indices.
- B+ tree is an extension of the B tree.
- **The difference in B+ tree and B tree is that in B tree the keys and records can be stored as internal as well as leaf nodes whereas in B+ trees, the records are stored as leaf nodes and the keys are stored only in internal nodes.**
- Each leaf is at the same height and all leaf nodes have links to the other leaf nodes.
- The root node always has a minimum of two children.



Properties of B+ Trees

1. All data is stored in the leaf nodes, while the internal nodes store just the indices.
2. Each leaf is at the same height.
3. All leaf nodes have links to the other leaf nodes.
4. The root node has a minimum of two children.
5. Each node except root can have a maximum of m children and a minimum of m/2 children.
6. Each node can contain a maximum of m-1 keys and a minimum of ⌈m/2⌉ - 1 keys.



Difference Between B Tree and B+ Tree

B Tree	B+ Tree
Data is stored in leaf as well as internal nodes	Data is stored only in leaf nodes.
Operations such as searching, insertion and deletion are comparatively slower.	Operations such as searching, insertion and deletion are comparatively faster.
No redundant search keys are present.	Redundant keys may be present.
Leaf nodes are not linked together.	Leaf nodes are linked together as a linked list.
They are not advantageous as compared to B+ trees, and hence, they aren't used in DBMS.	They are advantageous as compared to B trees, and hence, because of their efficiency, they find their applications in DBMS.

Insertion Operation on B+ Tree

Case 1: The leaf node is not full

- Insert the key into the leaf node in increasing order if the leaf isn't full.

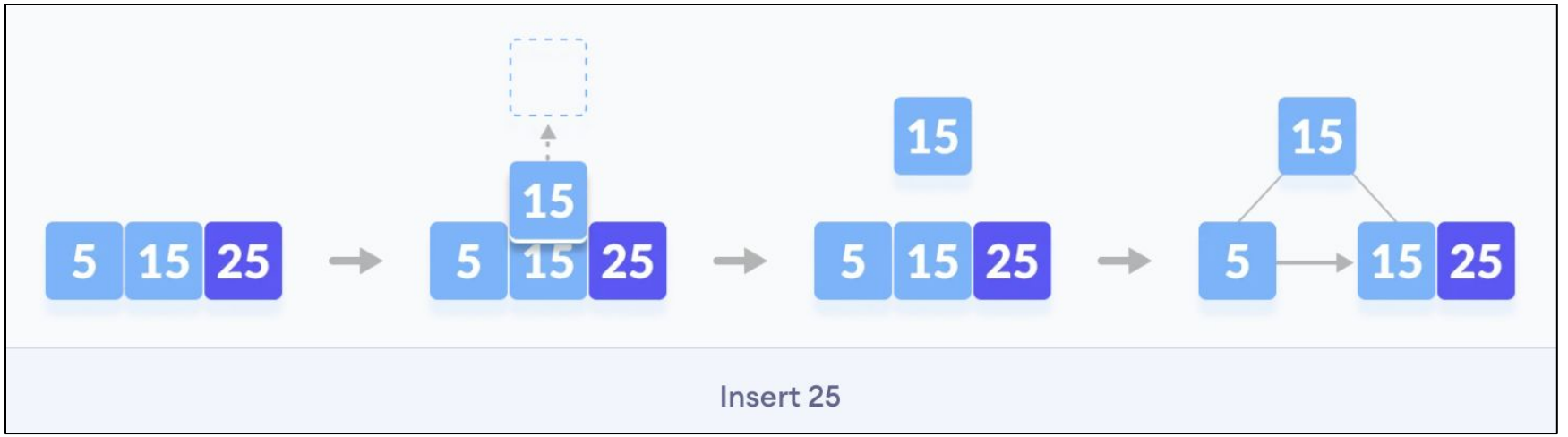
Case 2: The leaf node is full

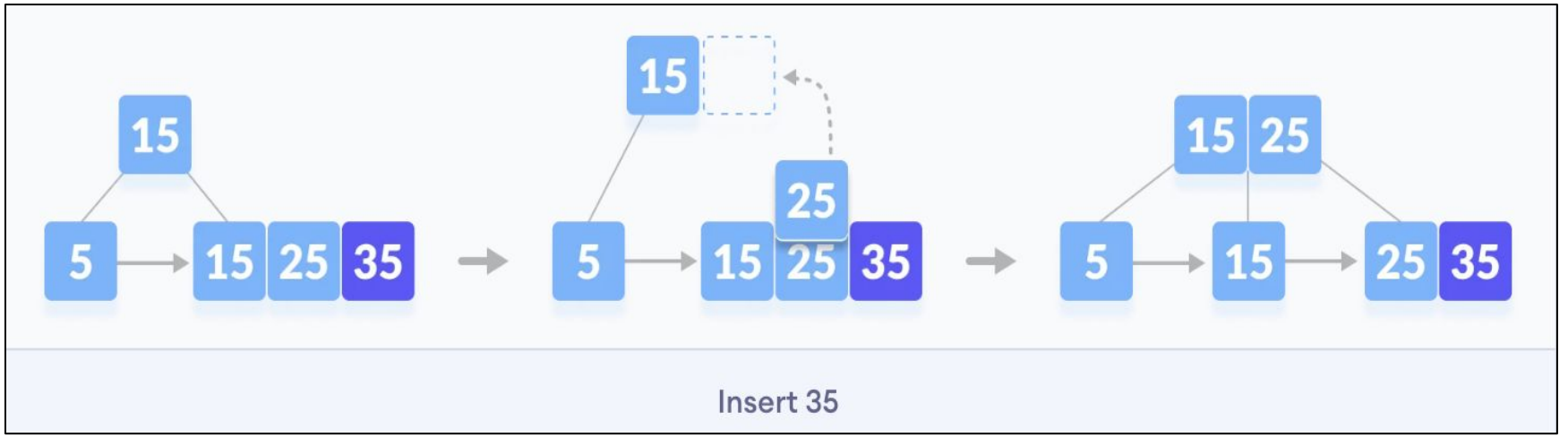
- **Step 1:** Insert the new node as a leaf node, in the increasing order. Now since the leaf node was already full, some balancing would be needed.
- **Step 2:** Break the node at m/2th position.
- **Step 3:** Add the m/2nd key to the parent node.
- **Step 4:** In case the parent node is already full, just repeat the steps 2 and 3.

Insertion Example

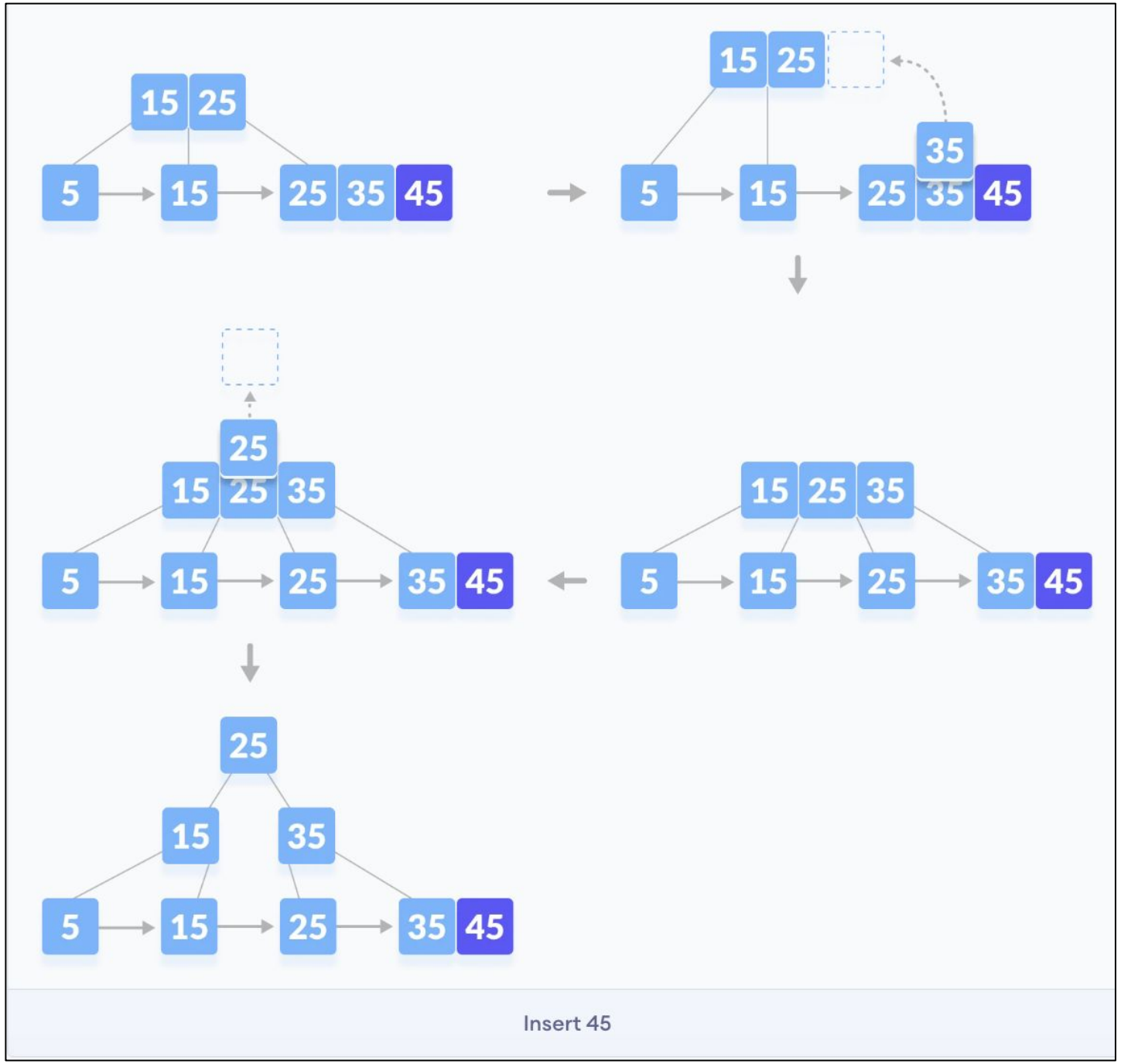
The elements to be inserted are 5,15, 25, 35, 45.

- Insert 5.
- Insert 15.
- Insert 25.
- Insert 35





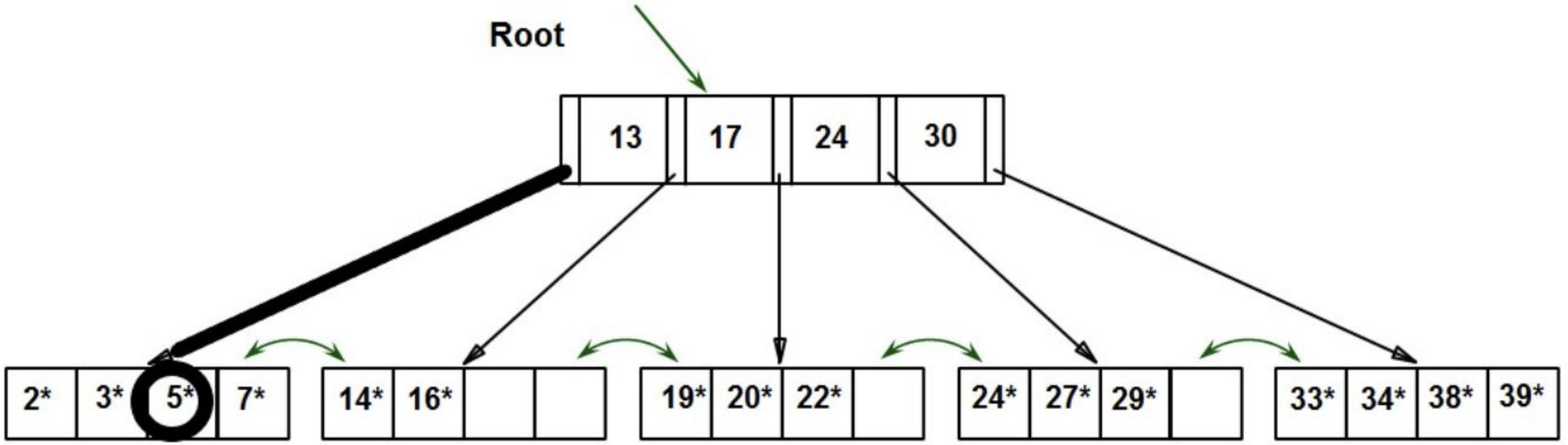
Insert 45



Insert 45

Search Operation on B+ Tree

- Search begins at root, and key comparisons direct it to a leaf
- Search for 5* ➤ Traverse along the left pointer to the element with the value 13 in the root node.



Deletion from a B+ Tree

- Deleting an element on a B+ tree consists of three main events:
  - **searching** the node where the key to be deleted exists,
  - **deleting** the key and balancing the tree if required.
  - **Underflow** is a situation when there is less number of keys in a node than the minimum number of keys it should hold.

Deletion Operation

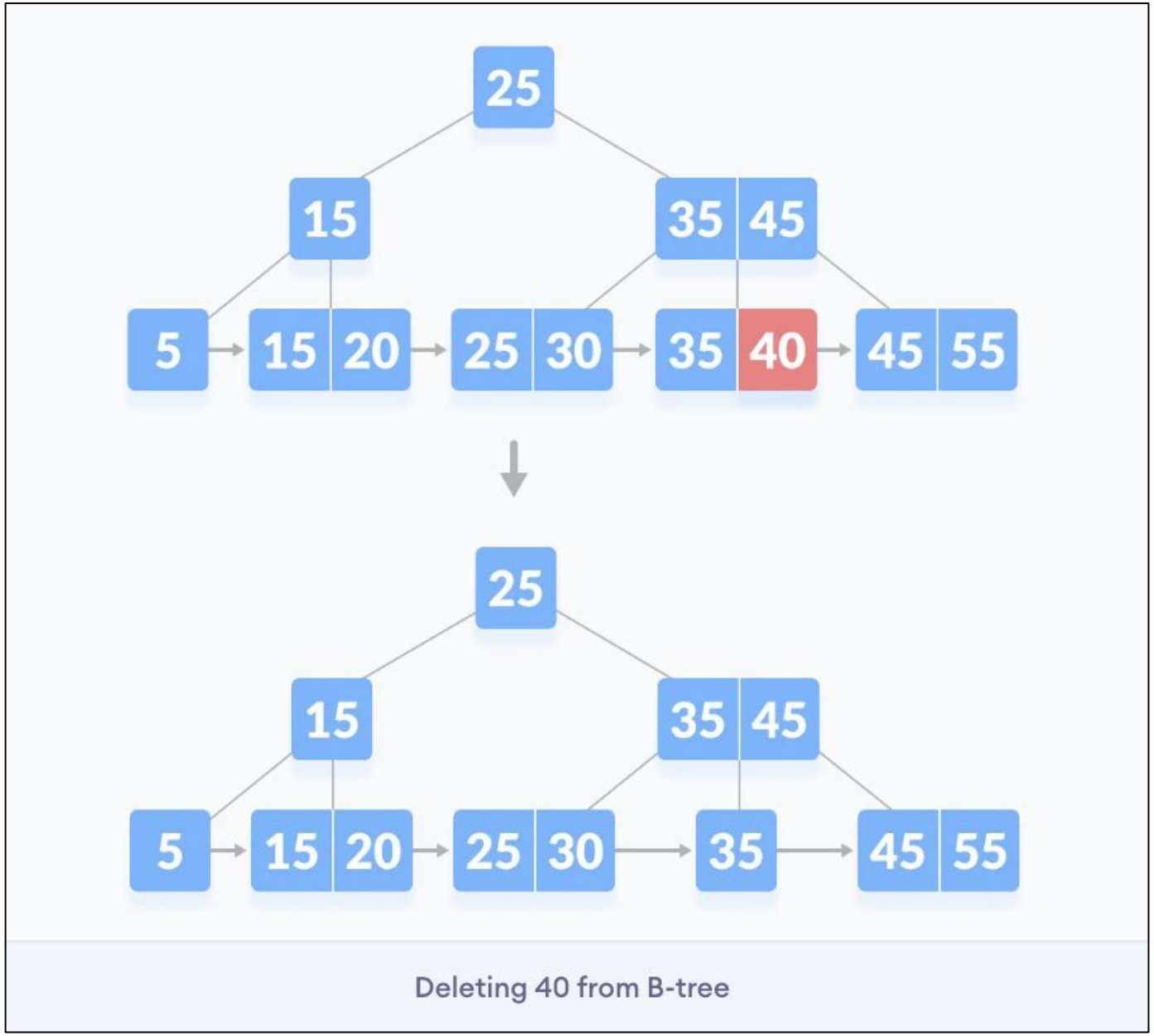
Before going through the steps below, one must know these facts about a B+ tree of degree m.

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

Example - Deletion from a B+ Tree

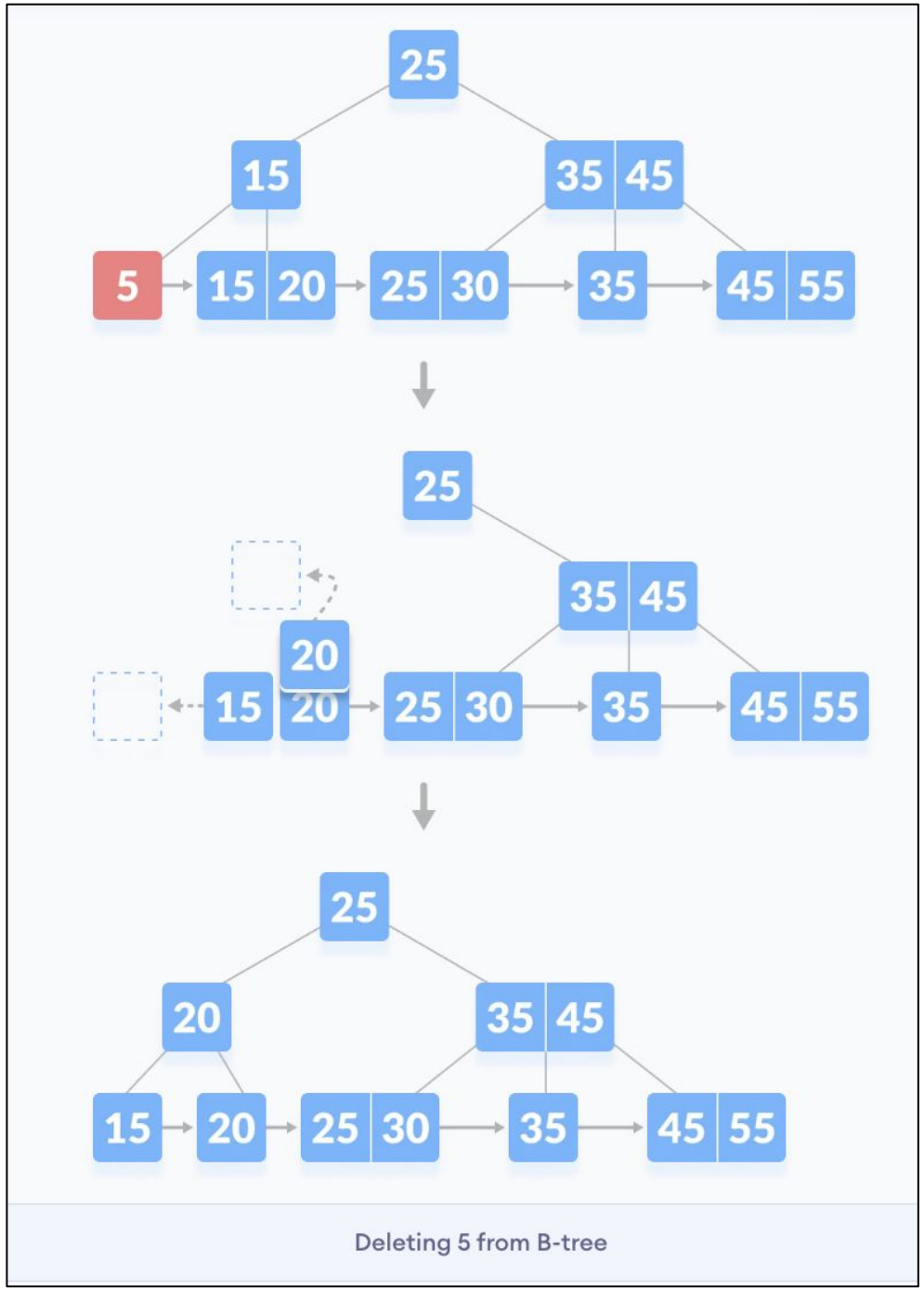
- While deleting a key, we have to take care of the keys present in the internal nodes (i.e. indexes) as well because the values are redundant in a B+ tree. Search the key to be deleted then follow the following steps.
- **Case I - The key to be deleted is present only at the leaf node not in the indexes** (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.



Deleting 40 from B-tree

2 . There is an exact minimum number of keys in the node. Delete the key and borrow a key from the immediate sibling. Add the median key of the sibling node to the parent.

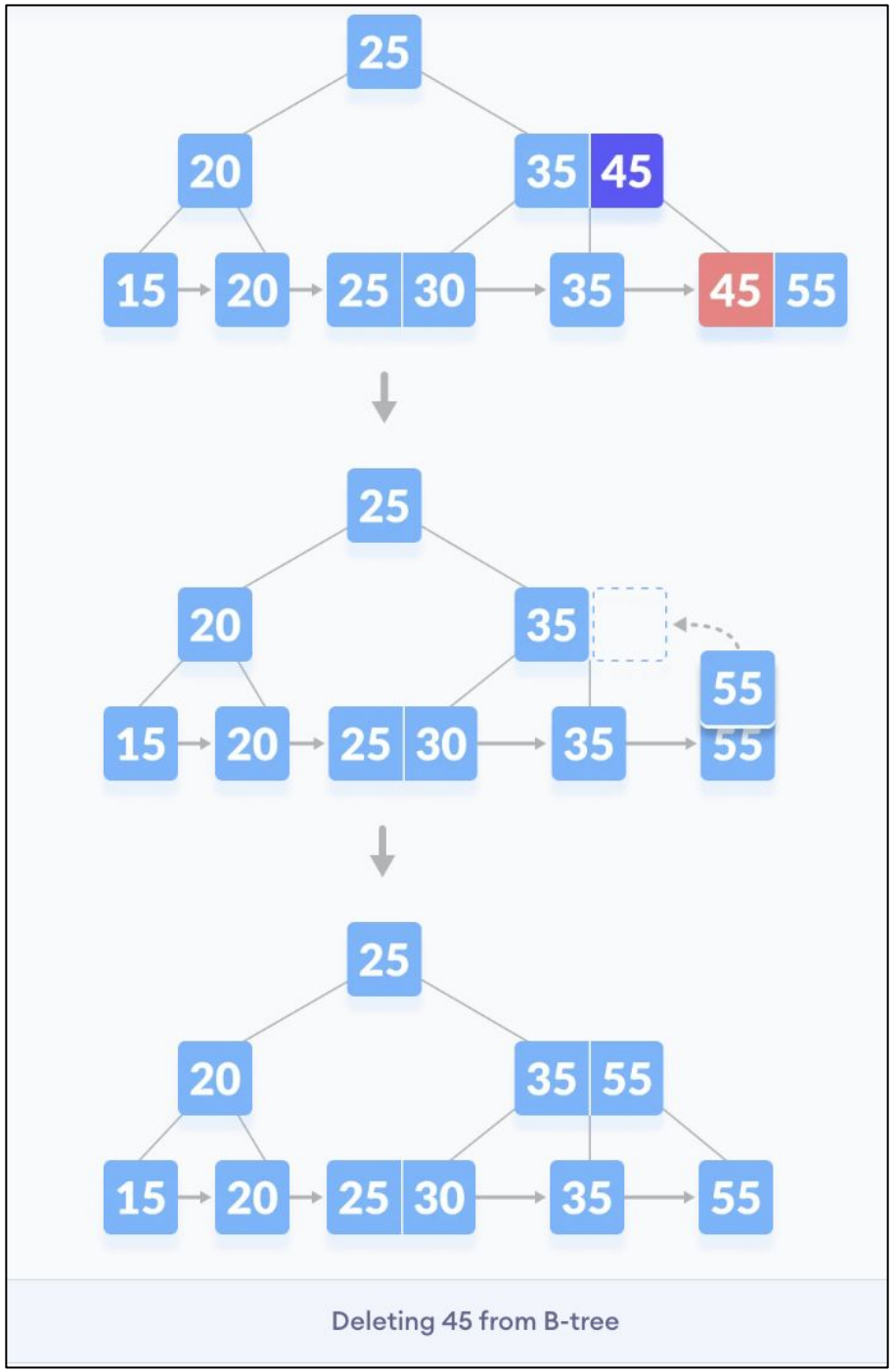


Deleting 5 from B-tree

Case II

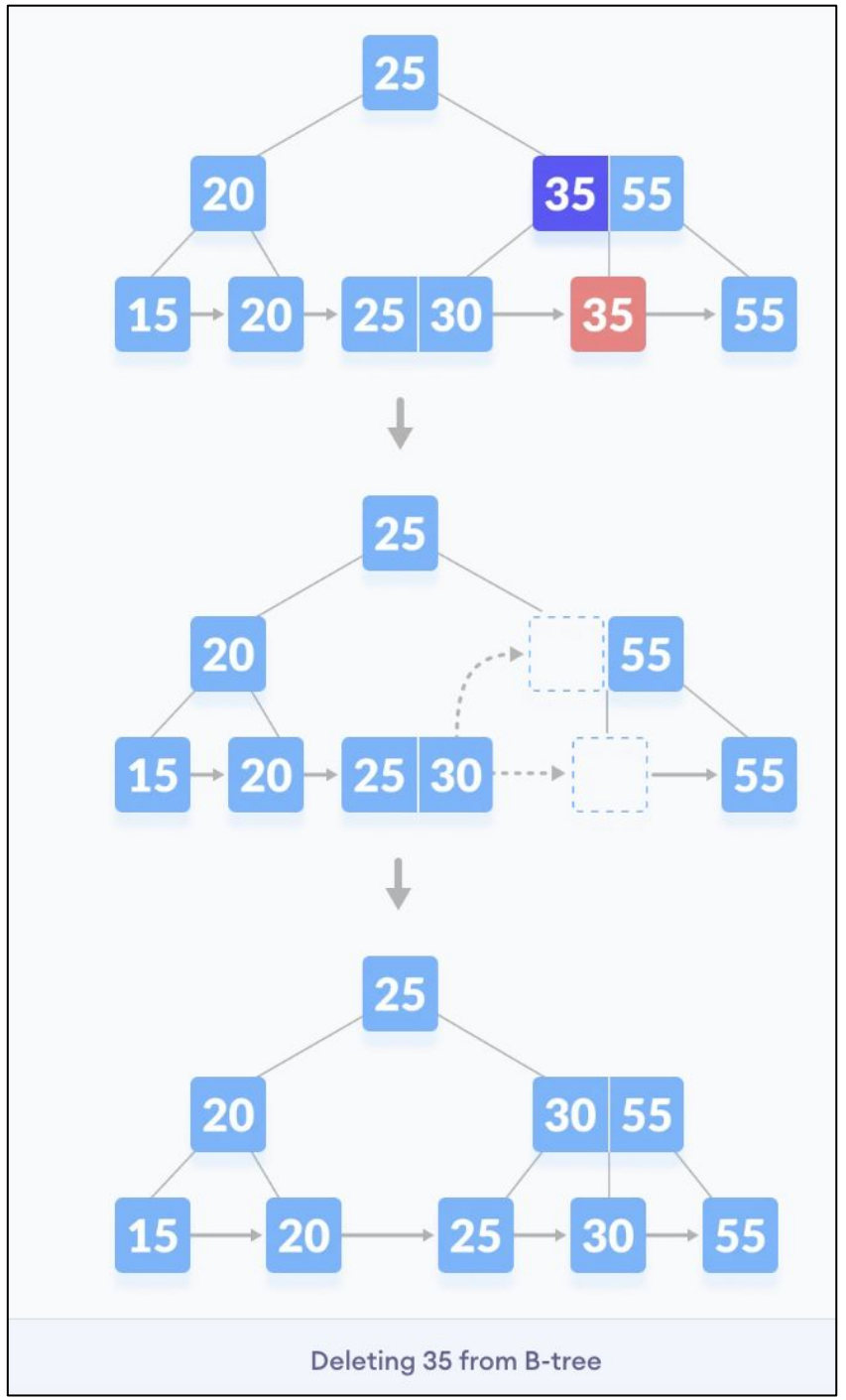
- The key to be deleted is present in the internal nodes as well. Then we have to remove them from the internal nodes as well. There are the following cases for this situation.

  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.



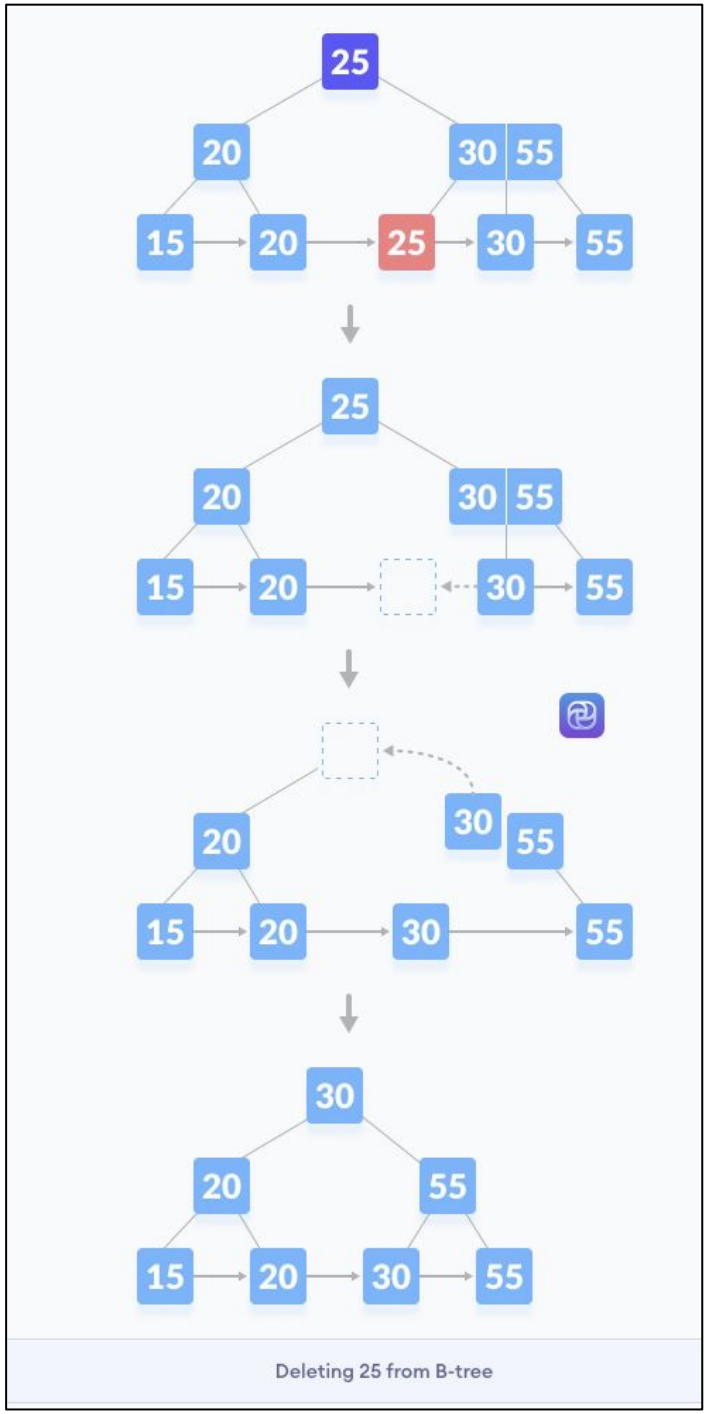
Deleting 45 from B-tree

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.



Deleting 35 from B-tree

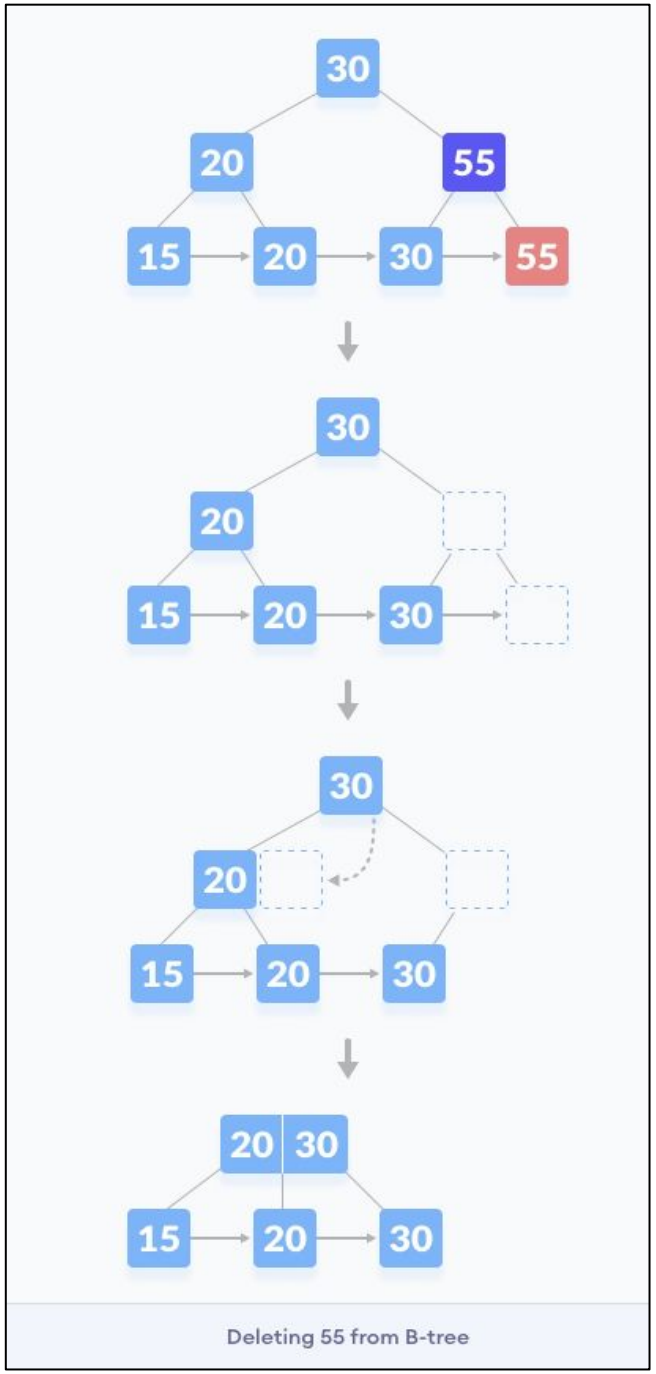
3 . This case is similar to **Case II(1)** but here, empty space is generated above the immediate parent node.After deleting the key, merge the empty space with its sibling.Fill the empty space in the grandparent node with the inorder successor.



Deleting 25 from B-tree

Case III -

- In this case, the height of the tree gets shrinked. It is a little complicated.Deleting 55 from the tree below leads to this condition. It can be understood in the illustrations below.



Deleting 55 from B-tree

Algorithm - Basic operations associated with B+ Tree

Searching a node in a B+ Tree:

1. Perform a binary search on the records in the current node.
2. If a record with the search key is found, then return that record.
3. If the current node is a leaf node and the key is not found, then report an unsuccessful search.
4. Otherwise, follow the proper branch and repeat the process.

Insertion of node in a B+ Tree:

5. Allocate new leaf and move half the buckets elements to the new bucket.
6. Insert the new leaf's smallest key and address into the parent.
7. If the parent is full, split it too.
8. Add the middle key to the parent node.
9. Repeat until a parent is found that need not split.
10. If the root splits, create a new root which has one key and two pointers. (That is, the value that gets pushed to the new root gets removed from the original node)

Algorithm - Deletion of a node in a B+ Tree:

- Descend to the leaf where the key exists.
- Remove the required key and associated reference from the node.
- If the node still has enough keys and references to satisfy the invariants, stop.
- If the node has too few keys to satisfy the invariants, but its next oldest or next youngest sibling at the same level has more than necessary, distribute the keys between this node and the neighbor. Repair the keys in the level above to represent that these nodes now have a different “split point” between them; this involves simply changing a key in the levels above, without deletion or insertion.
- If the node has too few keys to satisfy the invariant, and the next oldest or next youngest sibling is at the minimum for the invariant, then merge the node with its sibling; if the node is a non-leaf, we will need to incorporate the “split key” from the parent into our merging.
- In either case, we will need to repeat the removal algorithm on the parent node to remove the “split key” that previously separated these merged nodes — unless the parent is the root and we are removing the final key from the root, in which case the merged node becomes the new root (and the tree has become one level shorter than before).

Pseudocode - B + Tree

Searching

1) Apply Binary search on records.

2) If record with the search key is found

return required record

Else if current node is leaf node and key not found

print Element not Found

Insertion

1) If the bucket is not full (at most b 1 entries after the insertion), add the record.

2) Otherwise, split the bucket.

1) Allocate new leaf and move half the buckets elements to the new bucket.

2) Insert the new leafs smallest key and address into the parent.

3) If the parent is full, split it too.

1) Add the middle key to the parent node.

4) Repeat until a parent is found that need not split.

3) If the root splits, create a new root which has one key and two pointers.(That is, the value that gets pushed to the new root gets removed from the original node)

Pseudocode - 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 mode than one keys, done
      b. if root has only K
         i) if any of its child node can lend a node
            Borrow key from the child and adjust child links
         ii) Otherwise merge the children nodes it will be new root
      c. If n is a internal node, remove K
         i) If n has at lease 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
      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

Advantages of B+ Tree

1. Height of the B+ tree is always balanced and is comparitively lesser than B tree.
2. It takes equal number of disk accesses to fetch records.
3. Keys are used for indexing.
4. Because the data is only stored on the leaf nodes, search queries are faster.
5. Data stored in a B+ tree can be accesssed both sequentially and directly.

Applications of B+ Tree

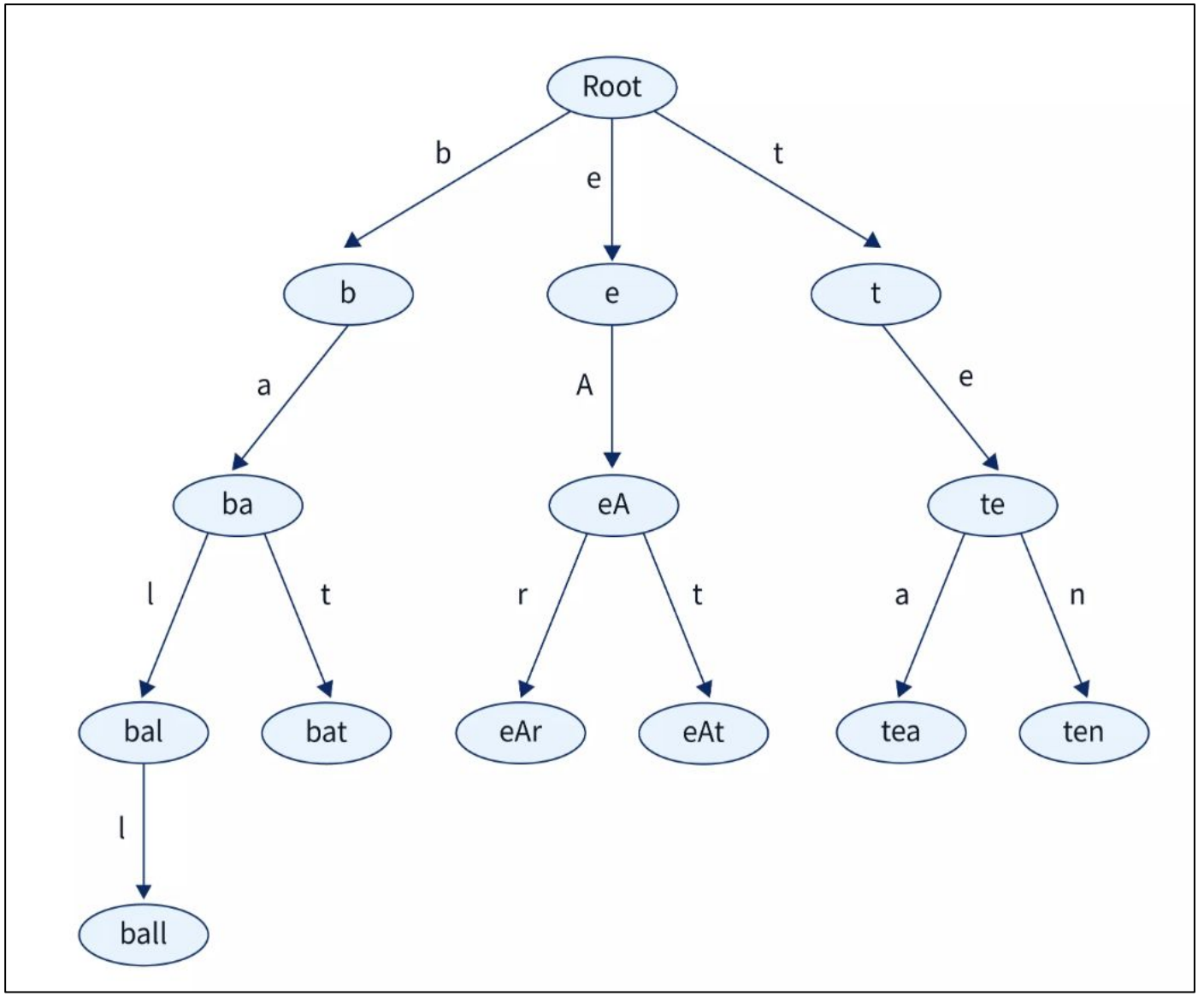
6. B+ trees in DBMS plays a useful role by supporting equality and range search.
7. It also facilitates database indexing in DBMS.
8. Another advantage is multilevel indexing.

Trie Tree

- Trie is a tree like data structure used to store collection of strings.
- Trie data structure is an advanced data structure used for storing and searching strings efficiently.
- Trie is an efficient information storage and retrieval data structure.
- Trie comes from the word reTRIEval which means to find or get something back.
- Dictionaries can be implemented efficiently using a Trie data structure and Tries are also used for the autocomplete features that we see in the search engines.
- Trie data structure is faster than binary search trees and hash tables for storing and retrieving data.
- Trie data structure is also known as a Prefix Tree or a Digital Tree. We can do prefix-based searching easily with the help of a Trie.

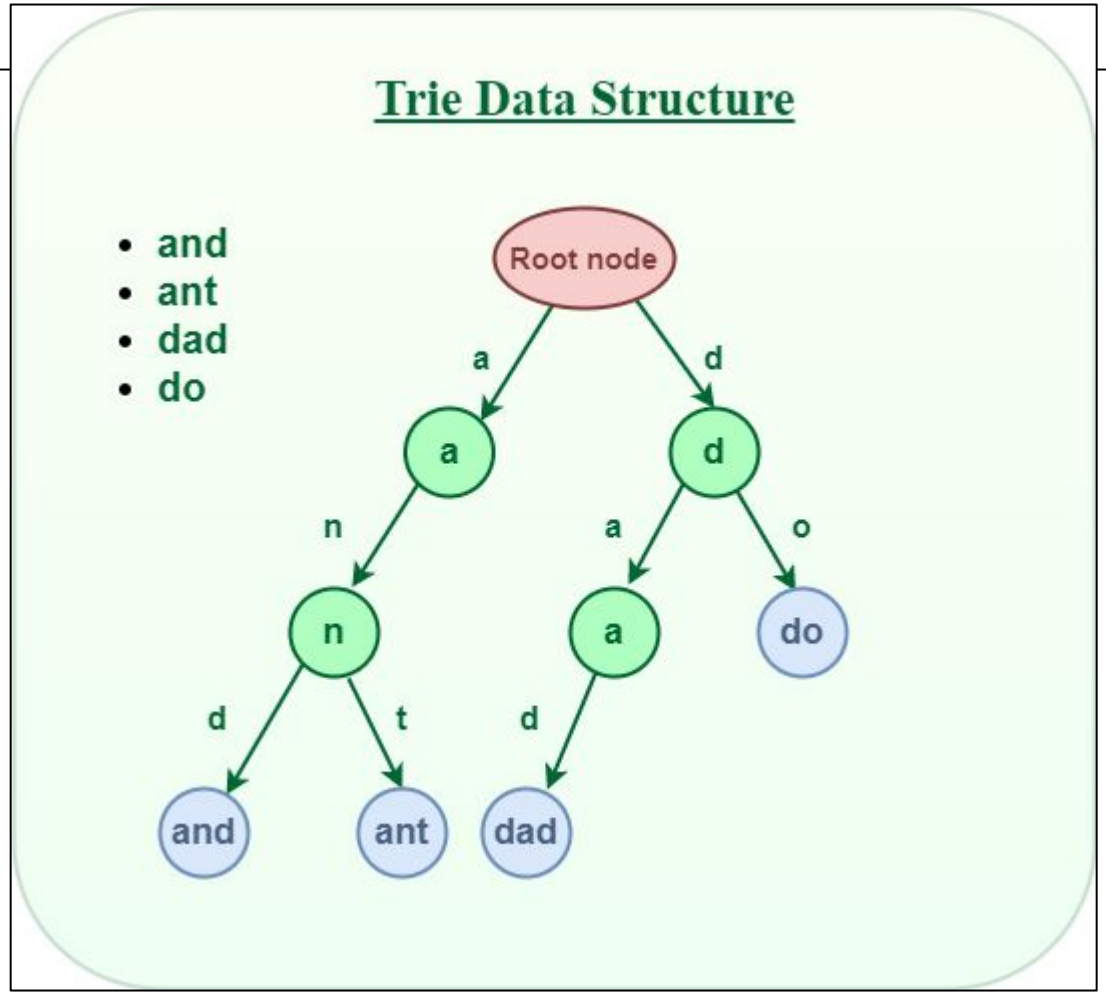
Properties of a Trie Data Structure

- The structure of a trie is like that of a tree.
- Each trie consists of a root node.
- The root node branches into various child nodes having multiple edges.
- Each TrieNode consists of an array of pointers where every index of the array represents a character.
- Each node of a trie represents a string and each edge represents a character.
- The root node is an empty string.
- Each node except the root node is a string having characters along the path from the root to that node.



Structure of Trie node:

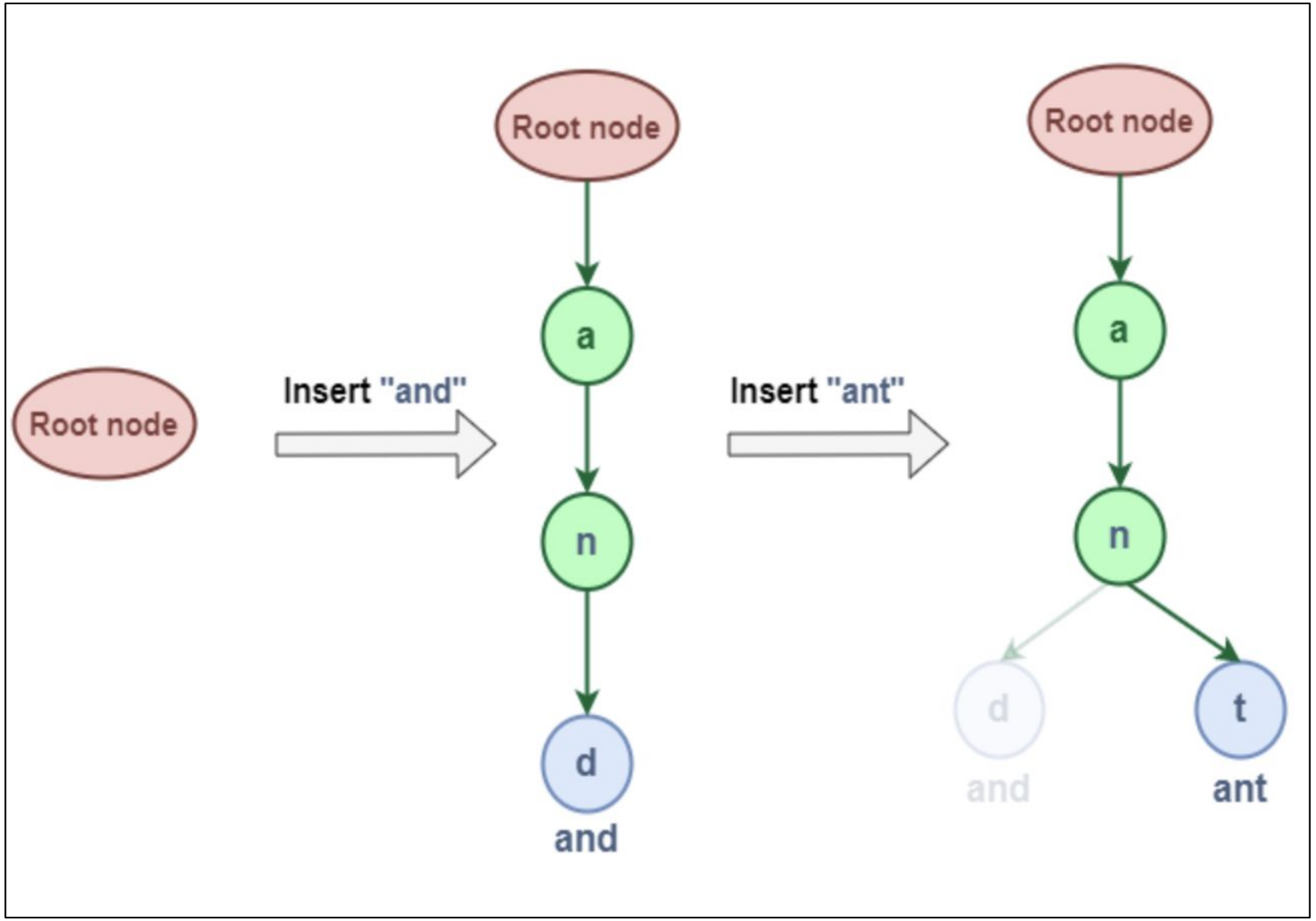
- Every node of Trie consists of multiple branches. Each branch represents a possible character of keys. Mark the last node of every key as the end of the word node. A Trie node field **isEndOfWord** is used to distinguish the node as the end of the word node.



Insert Operation in Trie:

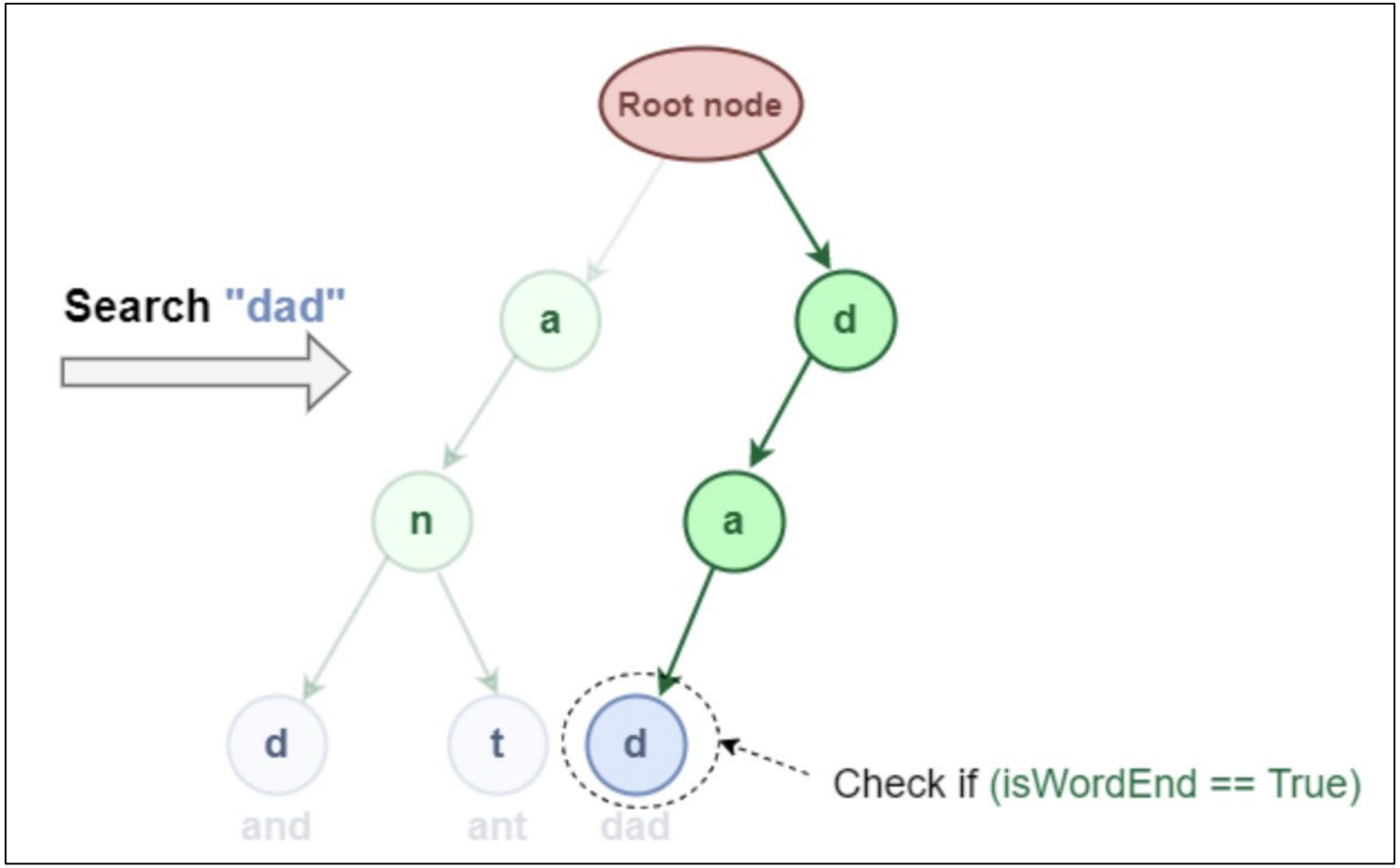
Inserting a key into Trie is a simple approach.

1. Every character of the input key is inserted as an individual Trie node. Note that the children is an array of pointers (or references) to next-level trie nodes.
2. The key character acts as an index to the array children.
3. If the input key is new or an extension of the existing key, construct non-existing nodes of the key, and mark the end of the word for the last node.
4. If the input key is a prefix of the existing key in Trie, Simply mark the last node of the key as the end of a word.



Search Operation in Trie:

- Searching for a key is similar to the insert operation.
- However, It only compares the characters and moves down.
- The search can terminate due to the end of a string or lack of key in the trie.
    1. In the former case, if the **isEndofWord** field of the last node is true, then the key exists in the trie.
    2. In the second case, the search terminates without examining all the characters of the key, since the key is not present in the trie.



Advantages of tries

1. In tries the keys are searched using common prefixes. Hence it is faster. The lookup of keys depends upon the height in case of binary search tree.

2. Tries take less space when they contain a large number of short strings. As nodes are shared between the keys.

3. Tries help with longest prefix matching, when we want to find the key.

Applications of tries

1. Tries has an ability to insert, delete or search for the entries. Hence they are used in building dictionaries such as entries for telephone numbers, English words.

2. Tries are also used in spell-checking softwares.

Program of B-tree implementation

```
#include <iostream>
#include <vector>
#include <algorithm>
template<typename T, int ORDER>
class BTree {
public:
    struct Node {
        std::vector<T> keys;
        std::vector<Node*> children;
        bool isLeaf;

        Node() : keys(ORDER - 1), children(ORDER), isLeaf(true) {}
    };
 Node* root;
```

```
void split(Node* parent, int childIndex) {
        Node* child = parent->children[childIndex];
        Node* newNode = new Node();
        newNode->isLeaf = child->isLeaf;

        parent->keys.insert(parent->keys.begin() + childIndex, child->keys[ORDER/2]);
        parent->children.insert(parent->children.begin() + childIndex + 1, newNode);

        newNode->keys.assign(child->keys.begin() + ORDER/2 + 1, child->keys.end());
        child->keys.resize(ORDER/2);

        if (!child->isLeaf) {
            newNode->children.assign(child->children.begin() + ORDER/2 + 1, child->children.end());
            child->children.resize(ORDER/2 + 1);
        }
    }
```

```
void insertNonFull(Node* node, const T& key)
  {
        int i = node->keys.size() - 1;
        if (node->isLeaf) {
            node->keys.push_back(T()); // Make space for the new key
            while (i >= 0 && key < node->keys[i]) {
                node->keys[i + 1] = node->keys[i];
                i--;
            }
            node->keys[i + 1] = key;
        } else {
            while (i >= 0 && key < node->keys[i]) {
                i--;
            }
            i++;
            if (node->children[i]->keys.size() == ORDER - 1) {
                split(node, i);
                if (key > node->keys[i])
                    i++;
            }
            insertNonFull(node->children[i], key);
        }
    }
```

```
bool search(Node* node, const T& key)
{
        int i = 0;
        while (i < node->keys.size() && key > node->keys[i])
       {
            i++;
        }
        if (i < node->keys.size() && key == node->keys[i])
     {
            return true;
        }
        if (node->isLeaf)
      {
            return false;
        }
        return search(node->children[i], key);
    }
```

```
void remove(Node* node, const T& key) {
        int i = 0;
        while (i < node->keys.size() && key > node->keys[i]) {
            i++;
        }
        if (i < node->keys.size() && key == node->keys[i]) {
            // Key found in this node
            if (node->isLeaf) {
                node->keys.erase(node->keys.begin() + i);
            } else {
                // Replace key with predecessor or successor
                // Here, I'm just taking the predecessor for simplicity
                T predecessor = findPredecessor(node->children[i]);
                node->keys[i] = predecessor;
                remove(node->children[i], predecessor);
            }
        }
```

```
else {
            if (node->isLeaf) {
                std::cerr << "Key not found\n";
                return;
            }
            bool isLastChild = (i == node->keys.size());
            if (node->children[i]->keys.size() < ORDER/2) {
                // Borrow from left or right sibling or merge
                if (i > 0 && node->children[i-1]->keys.size() >= ORDER/2) {
                    // Borrow from left sibling
                    borrowFromLeft(node, i);
                } else if (i < node->keys.size() && node->children[i+1]->keys.size() >= ORDER/2) {
                    // Borrow from right sibling
                    borrowFromRight(node, i);
                } else {
                    // Merge with a sibling
                    if (isLastChild) {
                        merge(node, i - 1);
                        i--;
                    } else {
                        merge(node, i);
                    }
                }
            }
            remove(node->children[i], key);
        }
    }
```

```
T findPredecessor(Node* node)
{
        while (!node->isLeaf) {
            node = node->children.back();
        }
        return node->keys.back();
    }

void borrowFromLeft(Node* node, int index)
 {
        Node* child = node->children[index];
        Node* sibling = node->children[index - 1];

        child->keys.insert(child->keys.begin(), node->keys[index - 1]);
        if (!child->isLeaf) {
            child->children.insert(child->children.begin(), sibling->children.back());
            sibling->children.pop_back();
        }
        node->keys[index - 1] = sibling->keys.back();
        sibling->keys.pop_back();
    }
```

```
void borrowFromRight(Node* node, int index)
{
        Node* child = node->children[index];
        Node* sibling = node->children[index + 1];

        child->keys.push_back(node->keys[index]);
        if (!child->isLeaf) {
            child->children.push_back(sibling->children.front());
            sibling->children.erase(sibling->children.begin());
        }
        node->keys[index] = sibling->keys.front();
        sibling->keys.erase(sibling->keys.begin());
    }
```

```
void merge(Node* node, int index)
{
        Node* child = node->children[index];
        Node* sibling = node->children[index + 1];

        child->keys.push_back(node->keys[index]);
        child->keys.insert(child->keys.end(), sibling->keys.begin(), sibling->keys.end());
        if (!child->isLeaf) {
        child->children.insert(child->children.end(),sibling->children.begin(),sibling->children.end());
        }
        node->keys.erase(node->keys.begin() + index);
        node->children.erase(node->children.begin() + index + 1);
        delete sibling;
    }
```

```
BTree() : root(nullptr) {}

void insert(const T& key)
   {
        if (!root)
     {
            root = new Node();
            root->keys.push_back(key);
        }
    else
         {
            if (root->keys.size() == ORDER - 1)
            {
                Node* newNode = new Node();
                newNode->isLeaf = false;
                newNode->children[0] = root;
                split(newNode, 0);
                root = newNode;
            }
            insertNonFull(root, key);
        }
    }
```

```
bool search(const T& key)
  {
        if (!root) return false;
        return search(root, key);
    }
void remove(const T& key)
  {
        if (!root) {
            std::cerr << "Tree is empty\n";
            return;
        }
        remove(root, key);
        if (root->keys.empty()) {
            Node* tmp = root;
            if (root->isLeaf) {
                root = nullptr;
            } else {
                root = root->children[0];
            }
            delete tmp;
        }
 }
```

```
void print()
 {
        if (!root) {
            std::cout << "Tree is empty\n";
            return;
        }
        print(root);
    }

void print(Node* node)
 {
        for (int i = 0; i < node->keys.size(); ++i) {
            if (!node->isLeaf) {
                print(node->children[i]);
            }
            std::cout << node->keys[i] << " ";
        }
        if (!node->isLeaf) {
            print(node->children.back());
        }
    }
};
```

```
int main()
{
    BTree<int, 3> tree;

    tree.insert(10);
    tree.insert(20);
    tree.insert(5);
    tree.insert(6);
    tree.insert(12);
    tree.insert(30);
    tree.insert(7);
    tree.insert(17);

    std::cout << "B-tree after insertion: ";
    tree.print();
    std::cout << std::endl;

    tree.remove(20);
    tree.remove(30);

    std::cout << "B-tree after deletion: ";
    tree.print();
    std::cout << std::endl;

    return 0;
}
```