B TREE in Data Structure: Search, Insert, Delete Operation Example

What is a B Tree?

B Tree is a self-balancing data structure based on a specific set of rules for searching, inserting, and deleting the data in a faster and memory efficient way. In order to achieve this, the following rules are followed to create a B Tree.

A B-Tree is a special kind of tree in a data structure. In 1972, this method was first introduced by McCreight, and Bayer named it Height Balanced m-way Search Tree. It helps you to preserves data sorted and allowed various operations like Insertion, searching, and deletion in less time.

In this B-Tree tutorial, you will learn:

Rules for B-Tree

Here, are important rules for creating B_Tree

- All leaves will be created at the same level.

- B-Tree is determined by a number of degree, which is also called "order" (specified by an external actor, like a programmer), referred to as

  ```
  m
  ```

  onwards. The value of

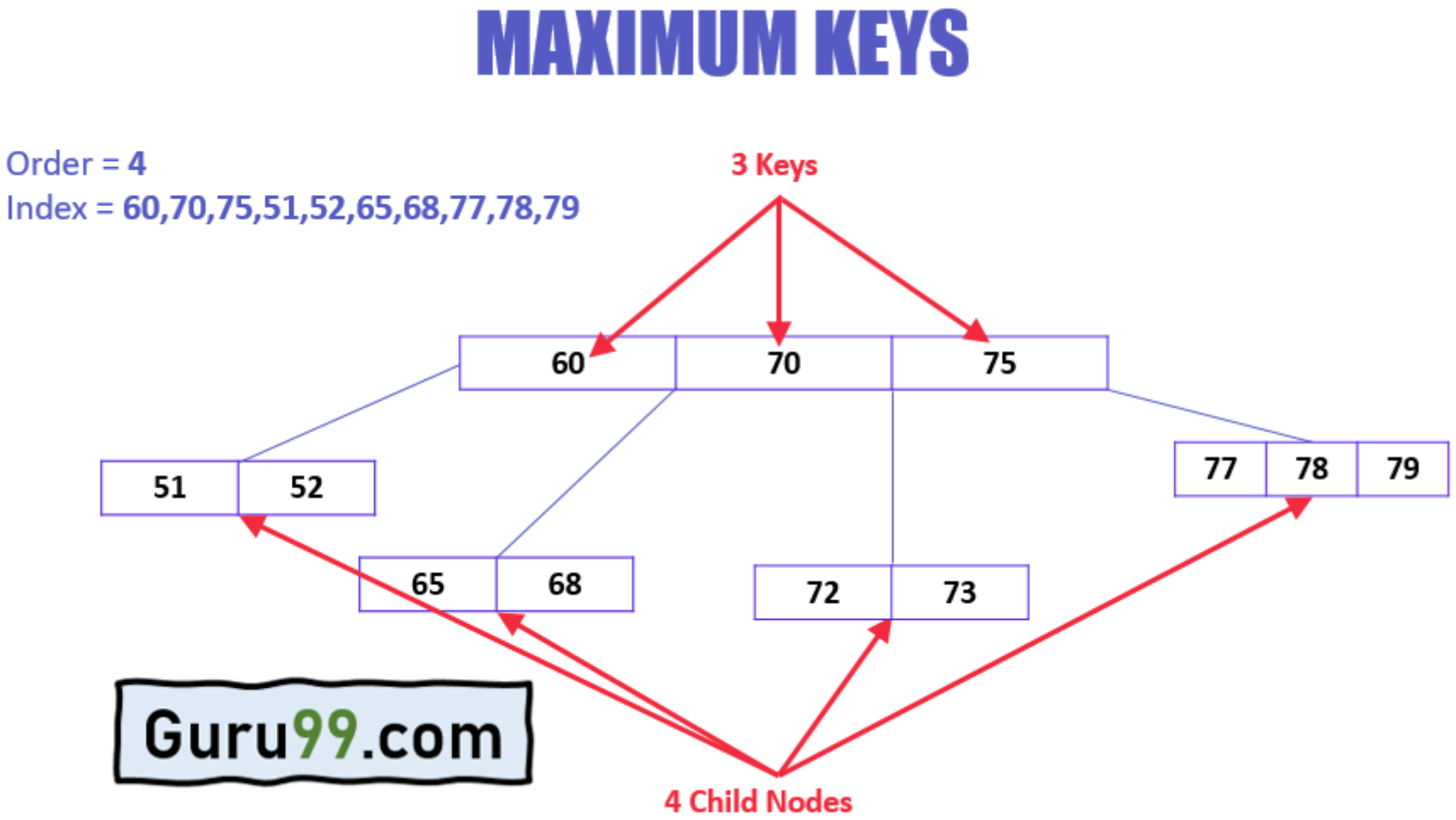
  ```
  m
  ```

  depends upon the block size on the disk on which data is primarily located.
- The left subtree of the node will have lesser values than the right side of the subtree. This means that the nodes are also sorted in ascending order from left to right.
- The maximum number of child nodes, a root node as well as its child nodes can contain are calculated by this formula:

  ```
  m – 1
  ```

  For example:

  ```
  m = 4
  max keys: 4 – 1 = 3
  ```



(/images/1/120319_1206_BTREEinData1.png)

- Every node, except root, must contain minimum keys of

```
[m/2]-1
```

For example:

```
m = 4
min keys: 4/2-1 = 1
```

- The maximum number of child nodes a node can have is equal to its degree, which is

```
m
```

- The minimum children a node can have is half of the order, which is m/2 (the ceiling value is taken).
- All the keys in a node are sorted in increasing order.

Why use B-Tree

Here, are reasons of using B-Tree

- Reduces the number of reads made on the disk
- B Trees can be easily optimized to adjust its size (that is the number of child nodes) according to the disk size
- It is a specially designed technique for handling a bulky amount of data.
- It is a useful algorithm for databases and file systems.
- A good choice to opt when it comes to reading and writing large blocks of data

History of B Tree

- Data is stored on the disk in blocks, this data, when brought into main memory (or RAM) is called data structure.
- In-case of huge data, searching one record in the disk requires reading the entire disk; this increases time and main memory consumption due to high disk access frequency and data size.
- To overcome this, index tables are created that saves the record reference of the records based on the blocks they reside in. This drastically reduces the time and memory consumption.
- Since we have huge data, we can create multi-level index tables.

- Multi-level index can be designed by using B Tree for keeping the data sorted in a self-balancing fashion.

Search Operation

The search operation is the simplest operation on B Tree.

The following algorithm is applied:

- Let the key (the value) to be searched by "k".
- Start searching from the root and recursively traverse down.
- If k is lesser than the root value, search left subtree, if k is greater than the root value, search the right subtree.
- If the node has the found k, simply return the node.
- If the k is not found in the node, traverse down to the child with a greater key.
- If k is not found in the tree, we return NULL.

Insert Operation

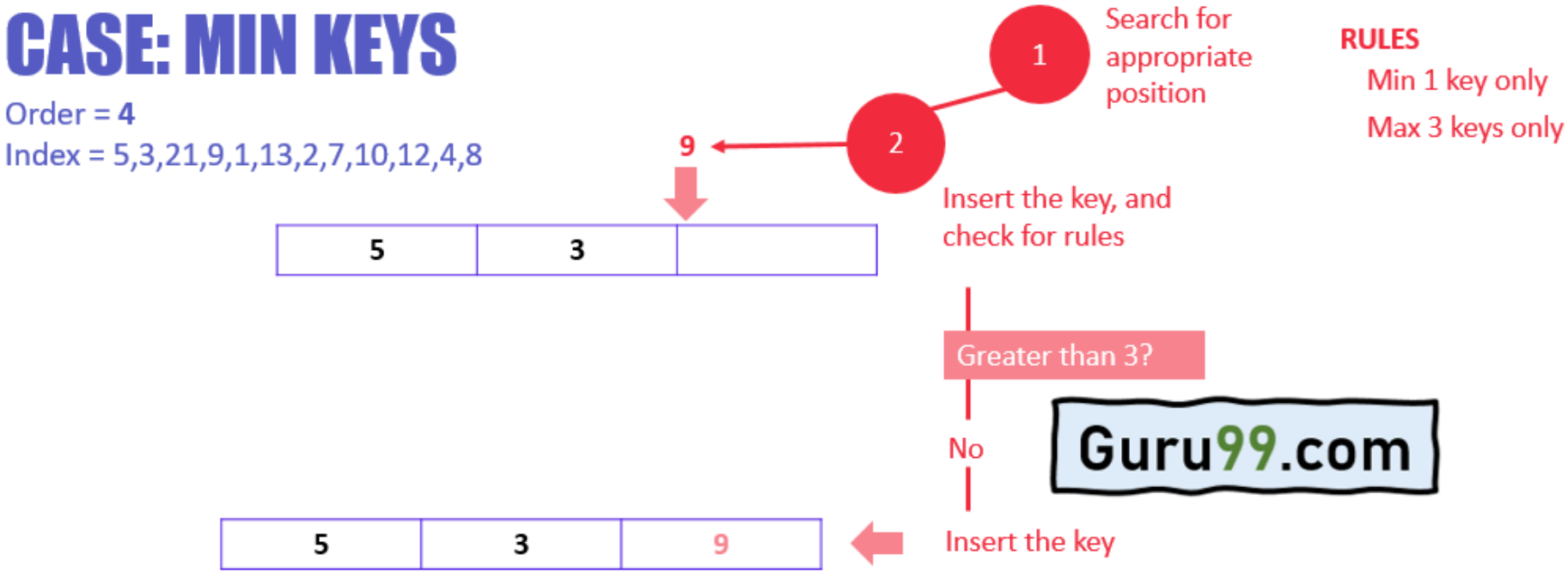
Since B Tree is a self-balancing tree, you cannot force insert a key into just any node.

The following algorithm applies:

- Run the search operation and find the appropriate place of insertion.
- Insert the new key at the proper location, but if the node has a maximum number of keys already:
- The node, along with a newly inserted key, will split from the middle element.
- The middle element will become the parent for the other two child nodes.
- The nodes must re-arrange keys in ascending order.

TIP The following is not true about the insertion algorithm:

Since the node is full, therefore it will split, and then a new value will be inserted

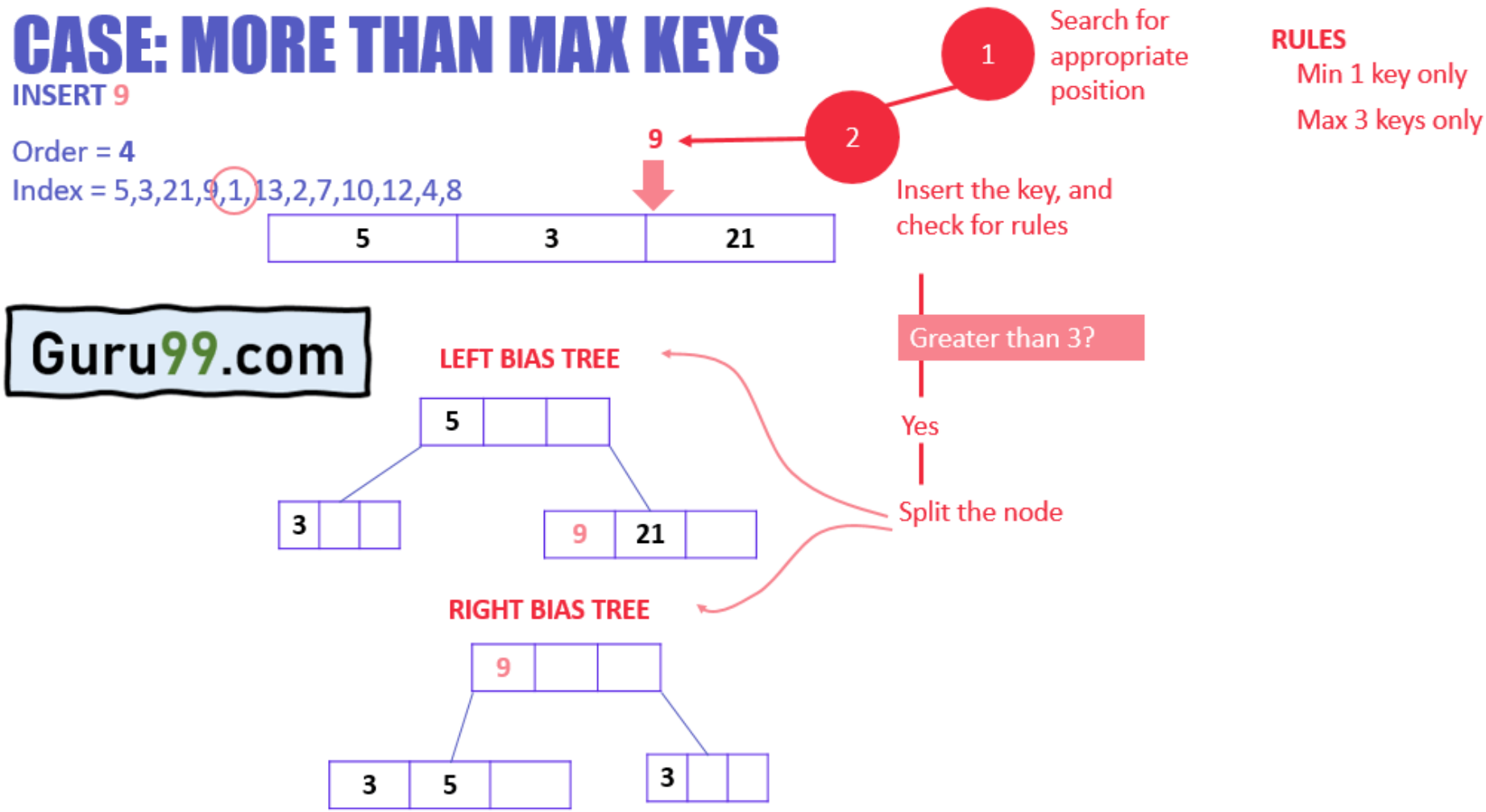


(/images/1/120319_1206_BTREEinData2.png)

In the above example:

- Search the appropriate position in the node for the key
- Insert the key in the target node, and check for rules
- After insertion, does the node have more than equal to a minimum number of keys, which is 1? In this case, yes, it does. Check the next rule.

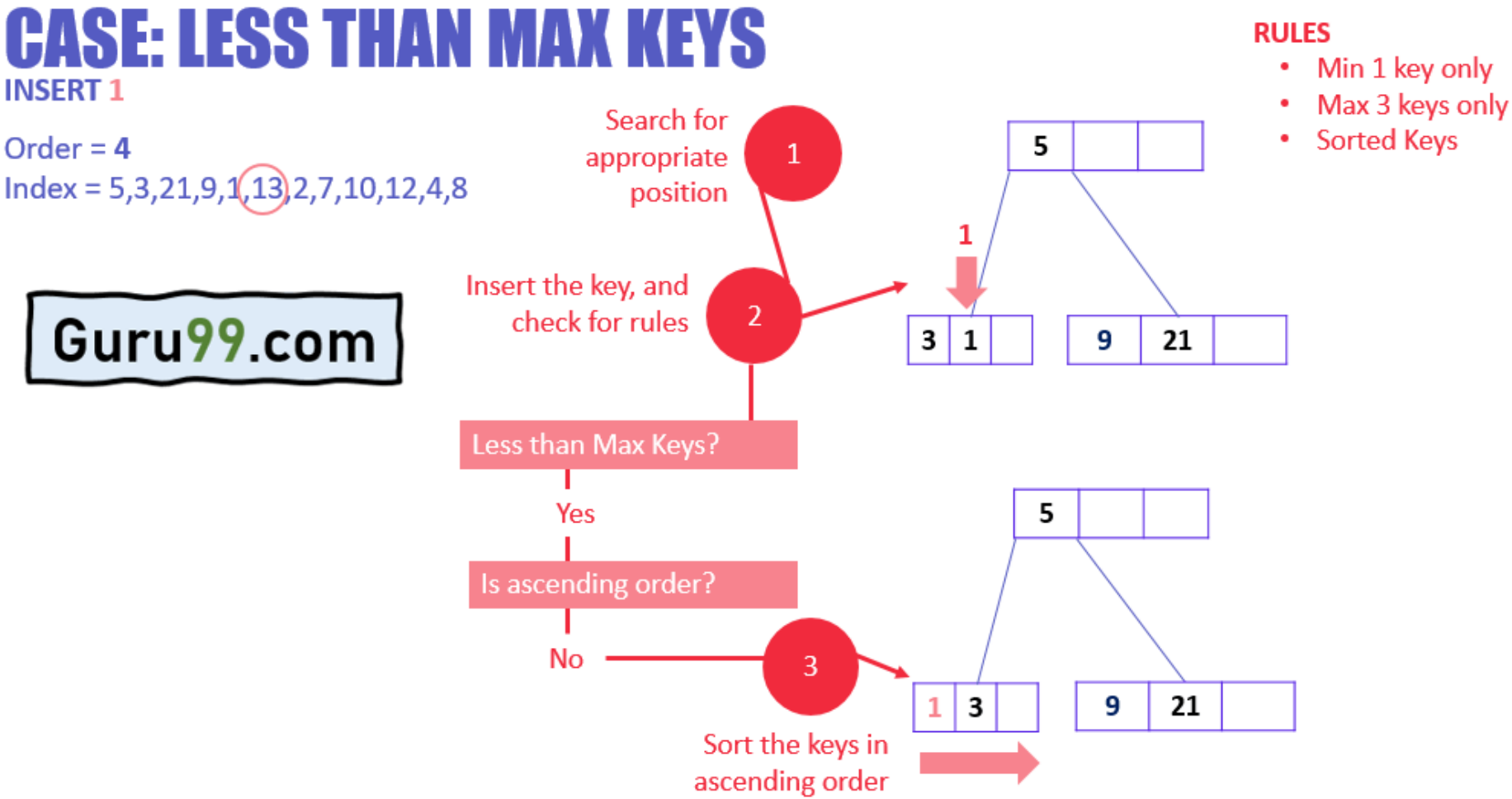
- After insertion, does the node have more than a maximum number of keys, which is 3? In this case, no, it does not. This means that the B Tree is not violating any rules, and the insertion is complete.



(/images/1/120319_1206_BTREEinData3.png)

In the above example:

- The node has reached the max number of keys
- The node will split, and the middle key will become the root node of the rest two nodes.
- In case of even number of keys, the middle node will be selected by left bias or right bias.

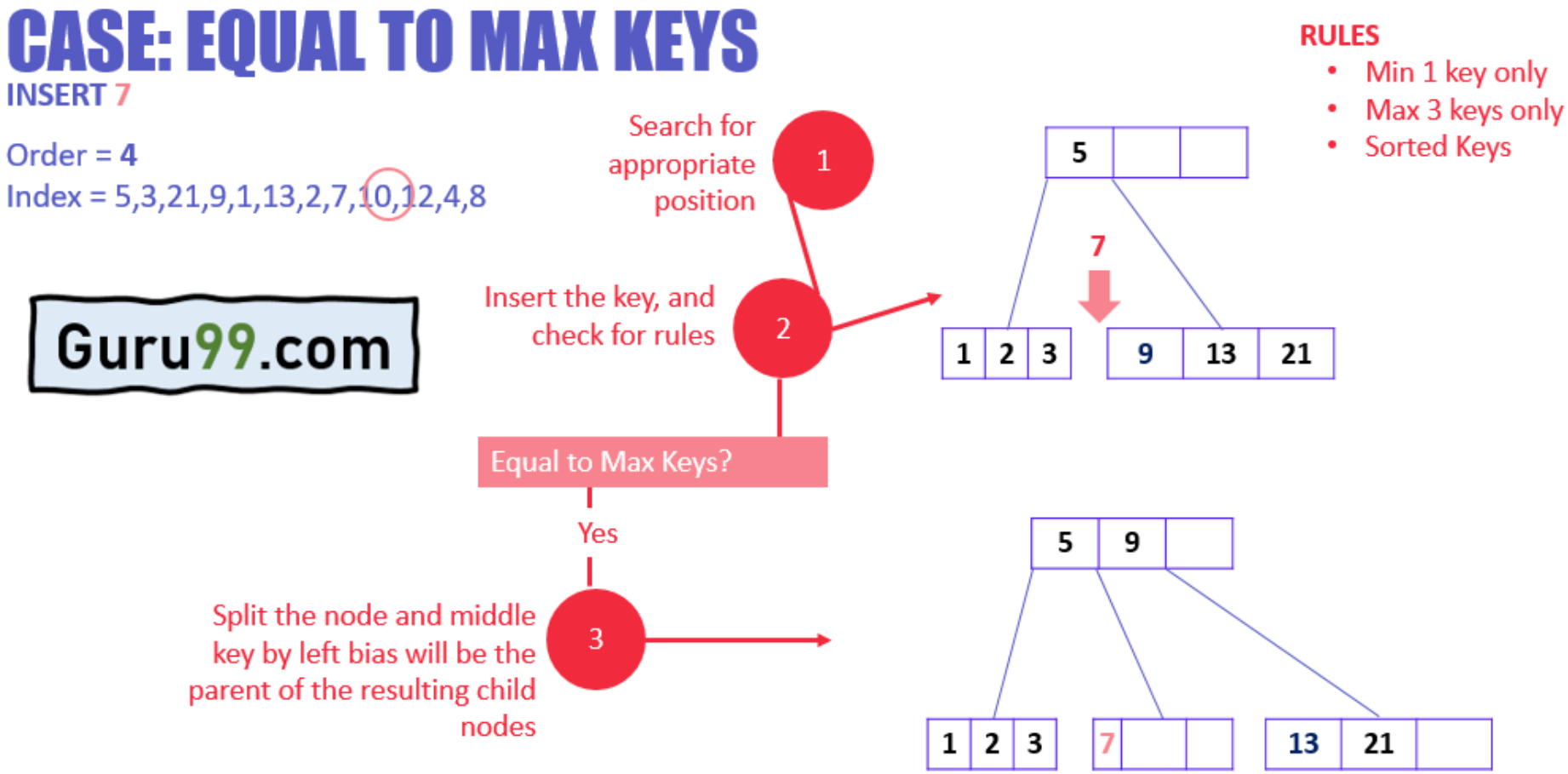


(/images/1/120319_1206_BTREEinData4.png)

In the above example:

- The node has less than max keys
- 1 is inserted next to 3, but the ascending order rule is violated
- In order to fix this, the keys are sorted

Similarly, 13 and 2 can be inserted easily in the node as they fulfill less than max keys rule for the nodes.

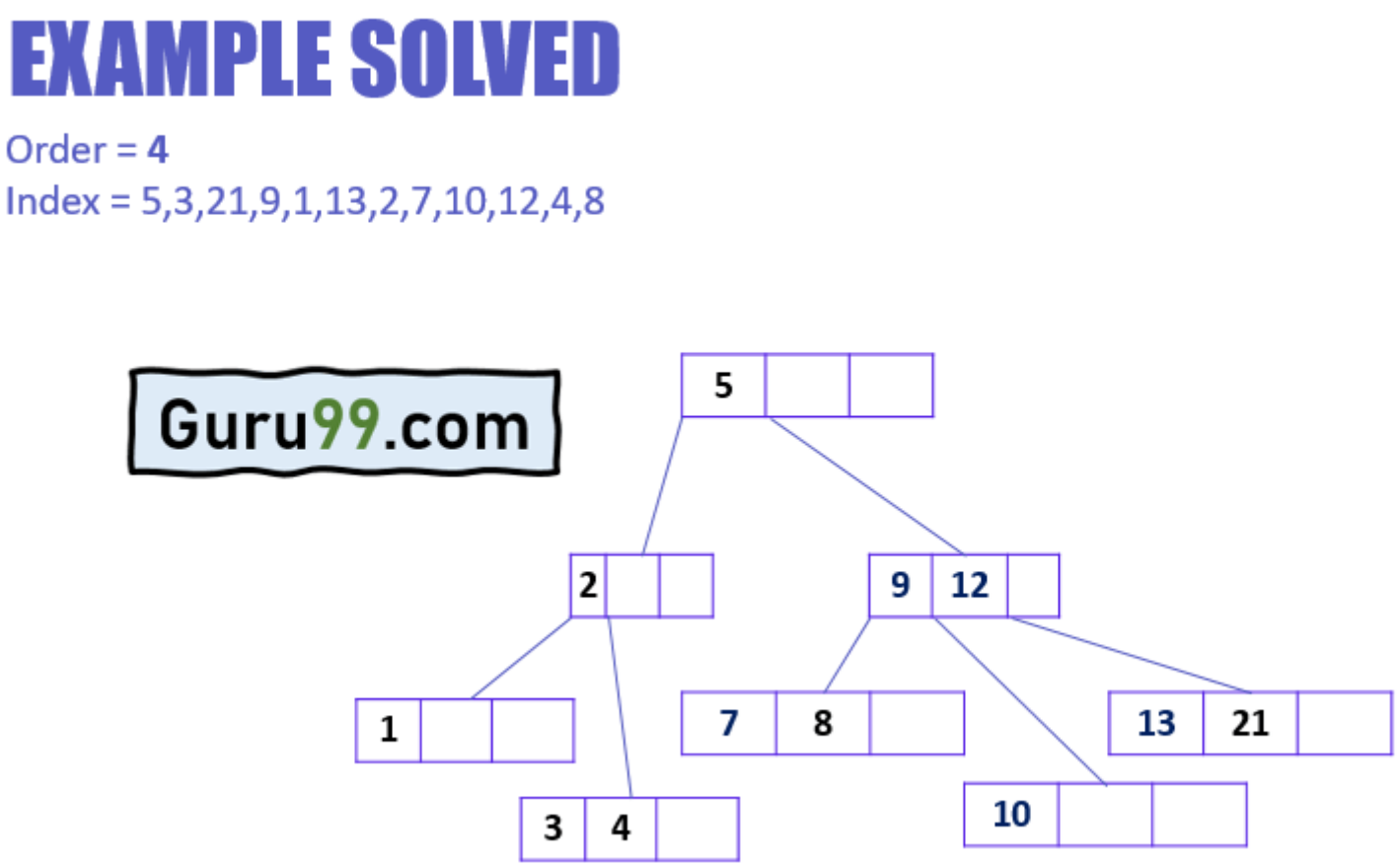


(/images/1/120319_1206_BTREEinData5.png)

In the above example:

- The node has keys equal to max keys.
- The key is inserted to the target node, but it violates the rule of max keys.
- The target node is split, and the middle key by left bias is now the parent of the new child nodes.
- The new nodes are arranged in ascending order.

Similarly, based on the above rules and cases, the rest of the values can be inserted easily in the B Tree.



(/images/1/120319_1206_BTREEinData6.png)

Delete Operation

The delete operation has more rules than insert and search operations.

The following algorithm applies:

- Run the search operation and find the target key in the nodes
- Three conditions applied based on the location of the target key, as explained in the following sections

If the target key is in the leaf node

- Target is in the leaf node, more than min keys.
  - Deleting this will not violate the property of B Tree
- Target is in leaf node, it has min key nodes
  - Deleting this will violate the property of B Tree
  - Target node can borrow key from immediate left node, or immediate right node (sibling)
  - The sibling will say **yes** if it has more than minimum number of keys
  - The key will be borrowed from the parent node, the max value will be transferred to a parent, the max value of the parent node will be transferred to the target node, and remove the target value
- Target is in the leaf node, but no siblings have more than min number of keys
  - Search for key
  - Merge with siblings and the minimum of parent nodes
  - Total keys will be now more than min
  - The target key will be replaced with the minimum of a parent node

If the target key is in an internal node

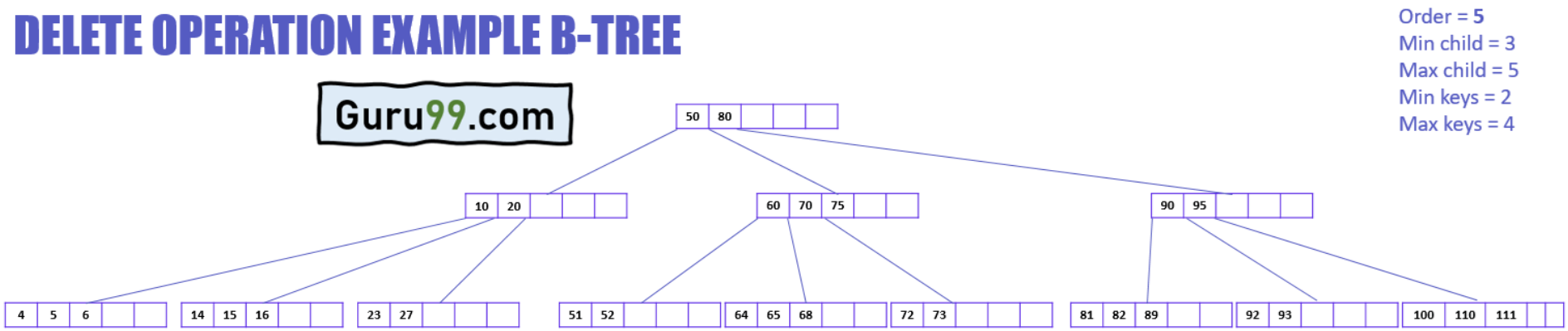
- Either choose, in- order predecessor or in-order successor
- In case the of in-order predecessor, the maximum key from its left subtree will be selected
- In case of in-order successor, the minimum key from its right subtree will be selected
- If the target key's in-order predecessor has more than the min keys, only then it can replace the target key with the max of the in-order predecessor
- If the target key's in-order predecessor does not have more than min keys, look for in-order successor's minimum key.
- If the target key's in-order predecessor and successor both have less than min keys, then merge the predecessor and successor.

If the target key is in a root node

- Replace with the maximum element of the in-order predecessor subtree

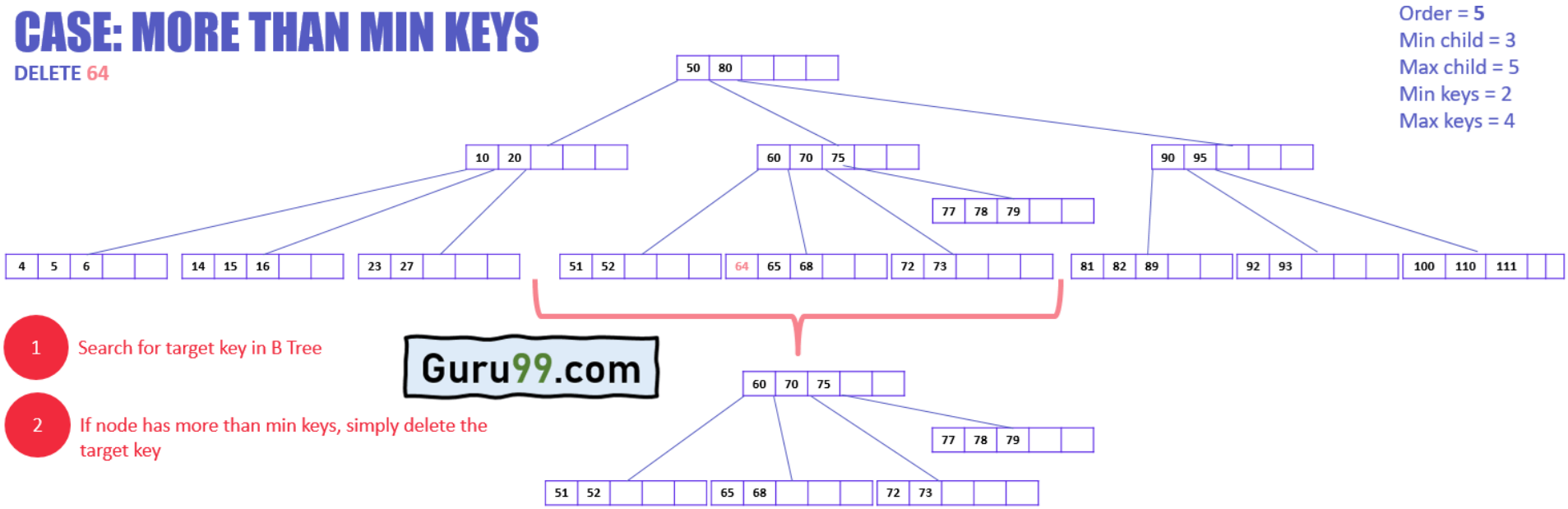
- If, after deletion, the target has less than min keys, then the target node will borrow max value from its sibling via sibling's parent.
- The max value of the parent will be taken by a target, but with the nodes of the max value of the sibling.

Now, let's understand the delete operation with an example.



(/images/1/120319_1206_BTREEinData7.png)

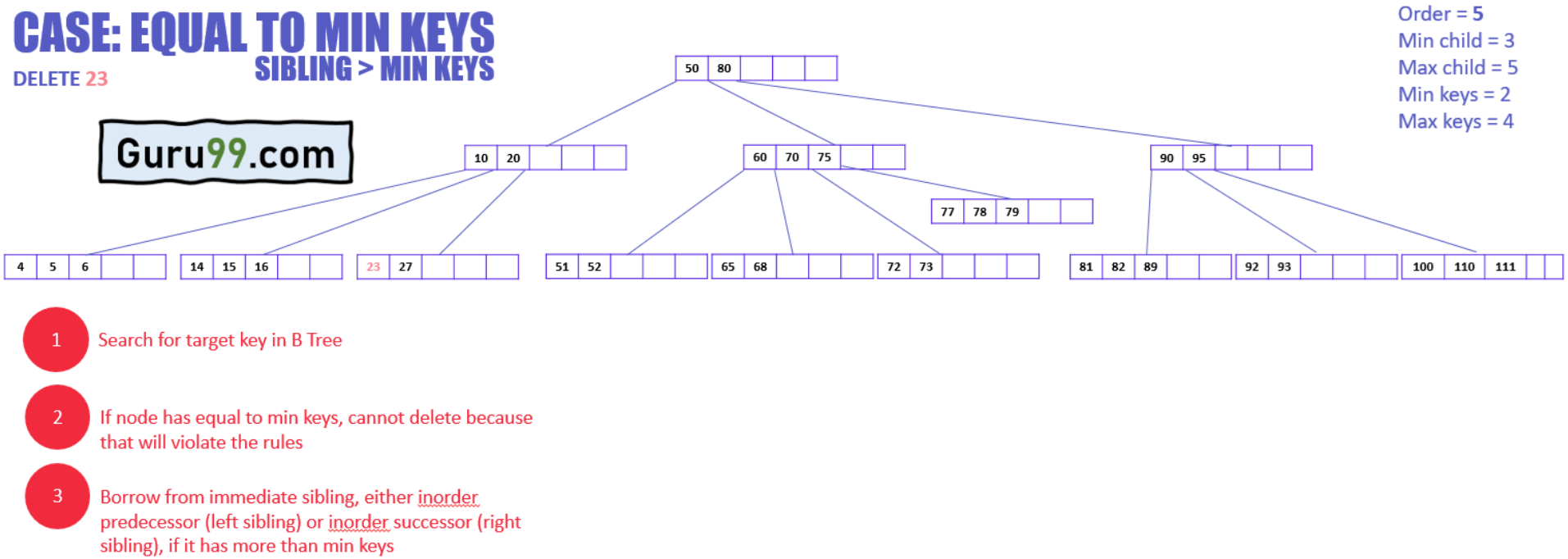
The above diagram displays different cases of delete operation in a B-Tree. This B-Tree is of order 5, which means that the minimum number of child nodes any node can have is 3, and the maximum number of child nodes any node can have is 5. Whereas the minimum and a maximum number of keys any node can have are 2 and 4, respectively.



(/images/1/120319_1206_BTREEinData8.png)

In the above example:

- The target node has the target key to delete
- The target node has keys more than minimum keys
- Simply delete the key

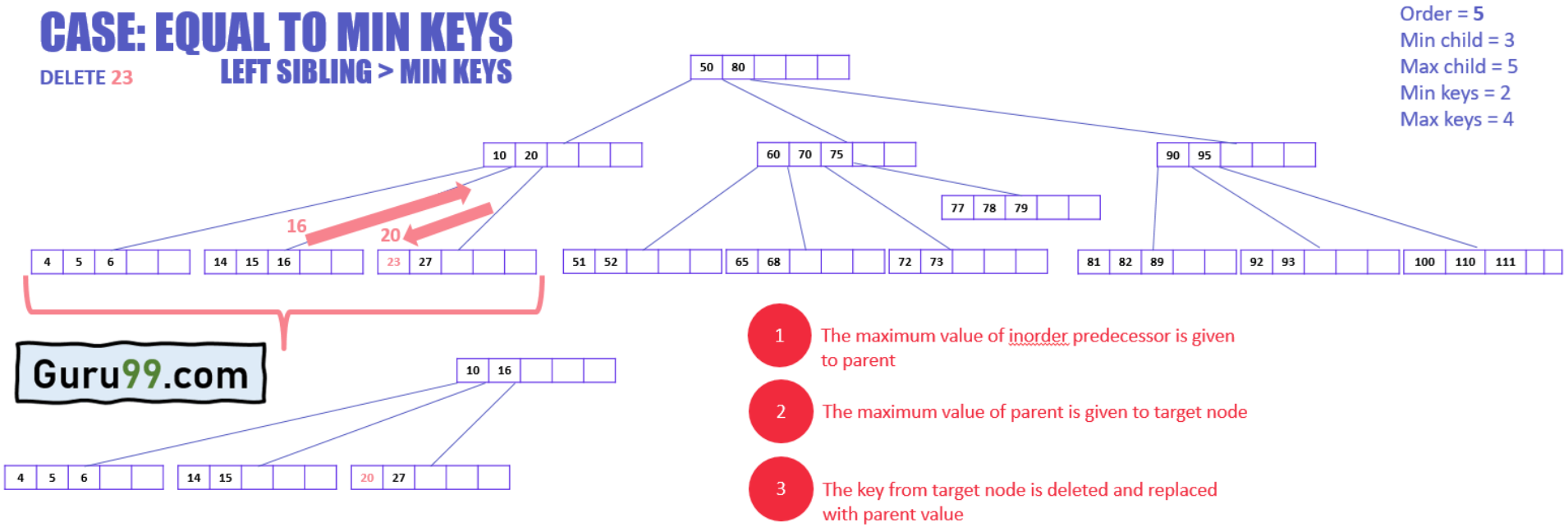


(/images/1/120319_1206_BTREEinData9.png)

In the above example:

- The target node has keys equal to minimum keys, so cannot delete it directly as it will violate the conditions

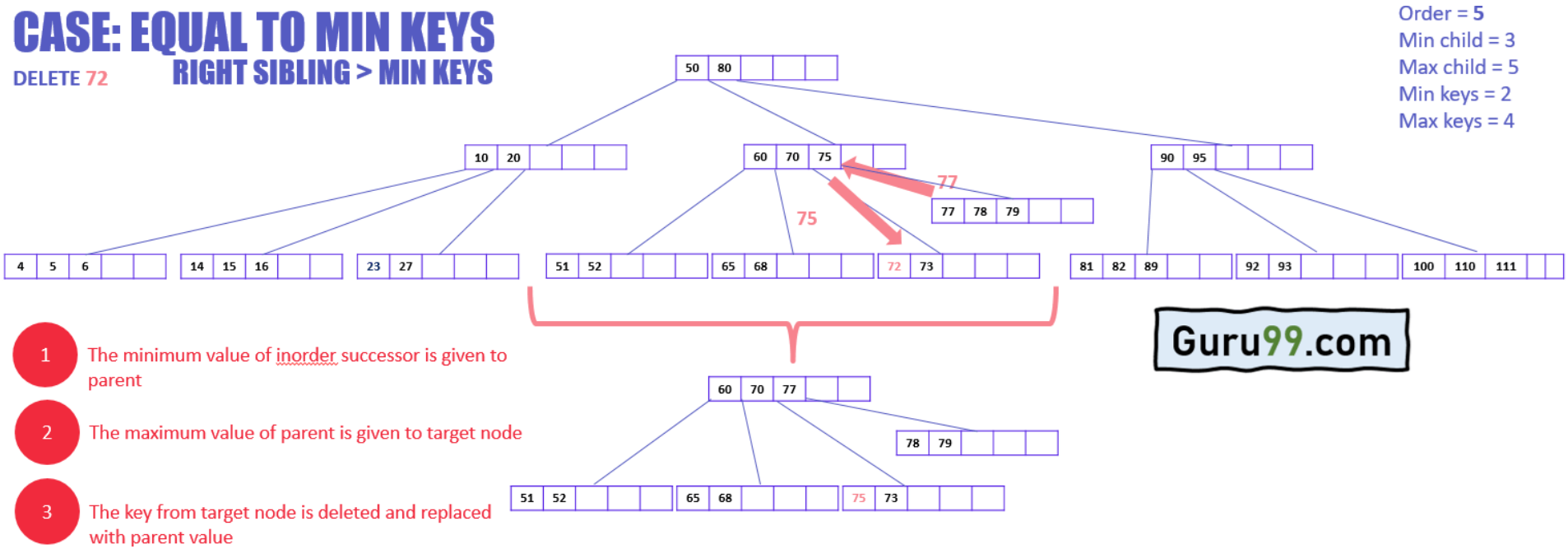
Now, the following diagram explains how to delete this key:



(/images/1/120319_1206_BTREEinData10.png)

- The target node will borrow a key from immediate sibling, in this case, in-order predecessor (left sibling) because it does not have any in-order successor (right sibling)
- The maximum value of the inorder predecessor will be transferred to the parent, and the parent will transfer the maximum value to the target node (see the diagram below)

The following example illustrates how to delete a key that needs a value from its in-order successor.

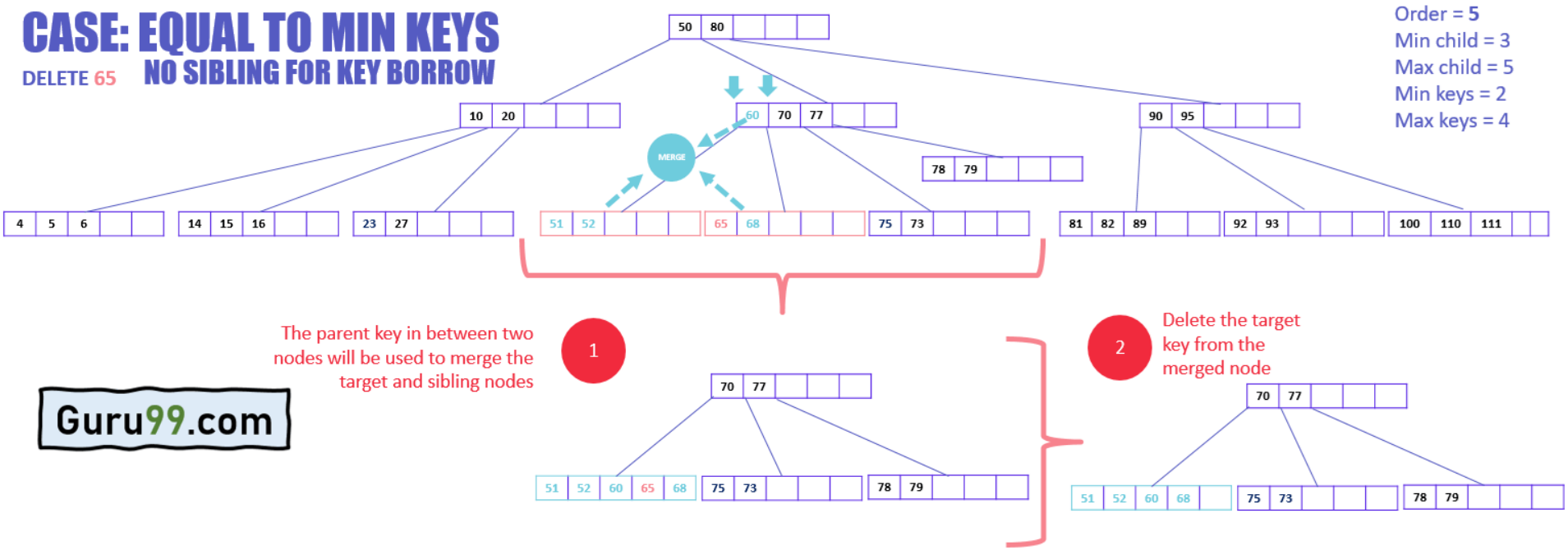


(/images/1/120319_1206_BTREEinData11.png)

- The target node will borrow a key from immediate sibling, in this case, in-order successor (right sibling) because it's in-order predecessor (left sibling) has keys equal to minimum keys.
- The minimum value of the in-order successor will be transferred to the parent, and the parent will transfer the maximum value to the target node.

In the example below, the target node does not have any sibling that can give its key to the target node. Therefore, merging is required.

See the procedure of deleting such a key:



(/images/1/120319_1206_BTREEinData12.png)

- Merge the target node with any of its immediate siblings along with parent key
  - The key from the parent node is selected that siblings in between the two merging nodes
- Delete the target key from the merged node

Delete Operation Pseudo Code

```
private int removeBiggestElement()
{
    if (root has no child)
        remove and return the last element
    else {
        answer = subset[childCount-1].removeBiggestElement()
        if (subset[childCount-1].dataCount < MINIMUM)
            fixShort (childCount-1)
        return answer
    }
}
```

Output:

The biggest element is deleted from the B-Tree.

Summary:

- B Tree is a self-balancing data structure for better search, insertion, and deletion of data from the disk.
- B Tree is regulated by the degree specified
- B Tree keys and nodes are arranged in ascending order.
- The search operation of B Tree is the simplest one, which always starts from the root and starts checking if the target key is greater or lesser than the node value.
- The insert operation of B Tree is rather detailed, which first finds an appropriate position of insertion for the target key, inserts it, evaluates the validity of B Tree against different cases, and then restructure the B Tree nodes accordingly.
- The delete operation of B Tree first searches for the target key to be deleted, deletes it, evaluates the validity based on several cases like minimum and maximum keys of the target node, siblings, and parent.

Prev (/array-data-structure.html) Report a Bug Next (/introduction-b-plus-tree.html)

(/jenkins-interview-questions.html)

(/download-install-jenkins.html)

SDLC

(/mean-stack-developer.html)

(/mean-stack-developer.html)

What is Mean Stack Developer? Skills, Salary, Growth

(/mean-stack-developer.html)

COURSE

(/pentaho-tutorial.html)

(/pentaho-tutorial.html)

Pentaho Tutorial: Learn Data Integration Reports

(/pentaho-tutorial.html)

R PROGRAMMING

(/r-programming-tutorial-pdf.html)

(/r-programming-tutorial-pdf.html)

R Programming Tutorial PDF

(/r-programming-tutorial-pdf.html)

Design & Algorithms Tutorial

1) Greedy Algorithm (/greedy-algorithm.html)

2) Circular Linked List (/circular-linked-list.html)

3) Array in Data Structures (/array-data-structure.html)

4) B TREE in Data Structure (/b-tree-example.html)

5) B+ TREE (/introduction-b-plus-tree.html)

6) BFS Algorithm (/breadth-first-search-bfs-graph-example.html)

7) Binary Search Tree (/binary-search-tree-data-structure.html)

8) Binary Search Algorithm (/binary-search.html)

9) BFS Vs DFS (/difference-between-bfs-and-dfs.html)

(https://www.facebook.com/guru99com/)
(https://twitter.com/guru99com)
(https://www.linkedin.com/company/guru99/)
(https://www.youtube.com/channel/UC19i1XD6k88KqHlET8atqFQ)
(https://forms.aweber.com/form/46/724807646.htm)

About

About Us (/about-us.html)
Advertise with Us (/advertise-us.html)
Write For Us (/become-an-instructor.html)
Contact Us (/contact-us.html)

Career Suggestion

SAP Career Suggestion Tool (/best-sap-module.html)
Software Testing as a Career (/software-testing-career-complete-guide.html)

Interesting

eBook (/ebook-pdf.html)
Blog (/blog/)
Quiz (/tests.html)
SAP eBook (/sap-ebook-pdf.html)

Execute online

Execute Java Online (/try-java-editor.html)
Execute Javascript (/execute-javascript-online.html)
Execute HTML (/execute-html-online.html)
Execute Python (/execute-python-online.html)