Table of Contents

[A. Introduction about B-Trees: 1](#_Toc106086254)

[*1. Definition of B-Tree:* 4](#_Toc106086255)

[*2. Application and Advantage for B-tree:* 6](#_Toc106086256)

[B. Operation on B-Tree data structure: 7](#_Toc106086257)

[1. Constructing a B-Tree: 7](#_Toc106086258)

[*2. Insert a Node into a tree* 9](#_Toc106086259)

[*3. Searching for a node:* 16](#_Toc106086260)

[*4. Delete a node in B-tree:* 21](#_Toc106086261)

[§ SUMMERIZE ABOUT COMPLEXITY: 33](#_Toc106086262)

# **A. Introduction about B-Trees:**

Linked lists have great advantages of flexibility over the contiguous representation of data structures, but they have one weak feature: They are sequential lists; that is, they are arranged so that it is necessary to move through them only one position at a time. Therefore, we can consider trees or graphs as a data structure, using the methods of pointers and linked lists for their implementation. Data structures organized as trees or graphs can prove valuable for a range of applications, especially for problems of information retrieval.

At some moments, we have been drawing trees or graphs to illustrate the behavior of algorithms look like this:



In data structures for programming, we already learned about some data structures such as: binary tree and 2-3 tree, etc. ; which are very useful for programming.





Therefore, we can come to the generalized form of these data structures, which is called: M-Way tree.



Extending more on the M-Way tree, we will get a more special type of data structure: M-Way search tree.



Going deeper on the M-way search tree we get a more specific type of data structure : B-Tree.



***So, let’s check out how B-Tree was invented!***

B-trees were invented by Rudolf Bayer and Edward M. McCreight while working at Boeing Research Labs, for the purpose of efficiently managing index pages for large random-access files. The basic assumption was that indexes would be so voluminous that only small chunks of the tree could fit in main memory. Bayer and McCreight's paper, Organization, and maintenance of large ordered indices, was first circulated in July 1970 and later published in Acta Informatica.

Bayer and McCreight never explained what, if anything, the B stands for: Boeing, balanced, broad, bushy, and Bayer have been suggested. McCreight has said that "the more you think about what the B in B-trees means, the better you understand B-trees."

## ***1. Definition of B-Tree:***

To get a specific definition for B-Tree, first, we should know about M-way trees.

* An **M-way(multi-way) tree** is a tree that has the following properties:
  + Each node in the tree can have at most **m** children.
  + Nodes in the tree have at most **(m-1)** key fields and pointers(references) to the children.



The above image is a 3-way tree, where each node has at most (3-1) = 2 keys and 3 children.

* An **M-way search tree** is a more constrained **M-way tree**, which has more property:
  + Each node in the tree can associate with m children and **m-1** key fields.
  + The keys in any node of the tree are arranged in a sorted order (ascending).
  + The keys in the first **K** children are less than the **K**th key of this node.
  + The keys in the last **(m-K)** children are higher than the **K**th key.



**§** Therefore, a B-tree is a special case of M-way search tree, and we got a new definition:

A B-tree is an extension of an M-way search tree. Besides having all the properties of an M-way search tree, it has some properties of its own, these mainly are:

* All leaves of B-tree are at the same level.
* A node in B-tree of order m can have at most m-1 keys and m children.
* Root node must have at least two nodes.
* Every node except the root node and the leaf node contain at least m/2 children.



§**Note:**

* 2-3 trees and binary search trees that we learned before can be B-trees.
* If n ≥ 1, then for any n-key B-tree of height h and minimum degree t ≥ 2, h ≥

## ***2. Application and Advantage for B-tree:***

* Advantage:
  + The need for B-tree arose with the rise in the need for lesser time in accessing the physical storage media like a hard disk. The secondary storage devices are slower with a larger capacity. There was a need for such types of data structures that minimize the disk accesses.
  + Other data structures such as a binary search tree, AVL tree, red-black tree, e.t.c. can store only one key in one node. If you have to store a large number of keys, then the height of such trees becomes very large and the access time increases.
  + However, B-tree can store many keys in a single node and can have multiple child nodes. This decreases the height significantly allowing faster disk accesses.
* Application :
  + databases and file systems
  + to store blocks of data (secondary storage media)
  + multilevel indexing

# **B. Operation on B-Tree data structure:**

**1. Constructing a B-Tree:**

There are 3 main operations on B-Tree, that is: Insert a node in a B-Tree, Search for a node and Delete a node. Let have a look at them.

Before go deeper into these operation, first, we need to define B-tree in coding. Let’s have a look at the definition of B-Tree and then the below picture: ( Assume that keys in each node are integer, other data structure we can do almost the same).

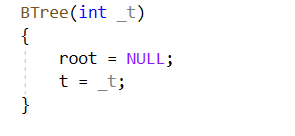
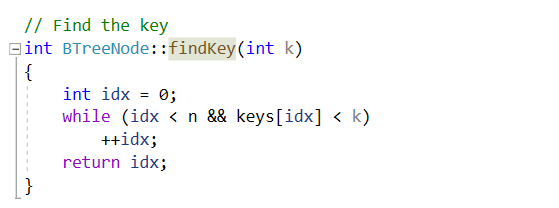


In this structure, we have 2 integer variable is sizeNums and degree, sizeNums will tell us how many keys in that node, degree will tell us how many children for that node. We also have an array of integer numbers for the keys, and an array of pointers for children of that node. But, it is not clear for the property of a node. So, we will add 1 more variable to check if that node is a leaf or not. Then, we will get:



® Note: **t** is minimum degree of B-tree and **n** is number of children.

For convenience coding, we have some small function and constructor:

* Constructor:   
    
  This function defines the number of keys and set root to null to initialize the tree.  
    
  This function creates a B-tree node with t1 is number of keys and a Boolean variable leaf1 to define if it is a leaf or not. It’s also creating a set of keys and children.
* Find key in a node:  
    
  This function return the key index of a number you want to find.

Now, let’s look at some operations for B-Tree.

## ***2. Insert a Node into a tree***

Before go to the insertion operation, we need some assistant function:

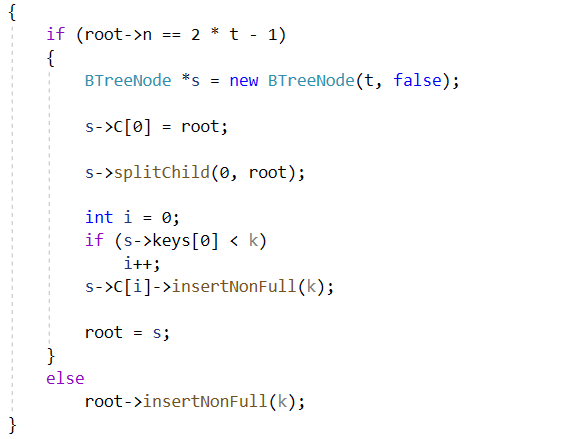
* ***Split children function:***  
  In this function, before inserting a value, should we know that a child node named y is already full, we have to split it into two. We also have ***i*** as the index of child node in array. During the process***,*** the median key of the split child node will be pushed into the current parent node.
* This function is used in 2 cases:  
  1. When we the root node is full and is made into a child of a newly created node.  
  2. When we know which child node will have the new key, and it is full, so we have to split it before inserting the key into that node.
* Insert a node whose root was not full:  
    
  This function is used to insert a key into a non-full node (it might be already non-full or it had been split beforehand).For a non-leaf node, we also have to check whether the corresponding child node is full, if yes then we have to split it before inserting to that node.

So, with these function, first let look at the main operation for insertion and implementation:

* If the tree is empty, allocate a root node and insert the key.
* Update the allowed number of keys in the node.
* Search the appropriate node for insertion.
* If the node is full, follow the steps below.
* Insert the elements in increasing order.
* Now, there are elements greater than its limit. So, split at the median.
* Push the median key upwards and make the left keys as a left child and the right keys as a right child.
* If the node is not full, follow the steps below.
* Insert the node in increasing order.

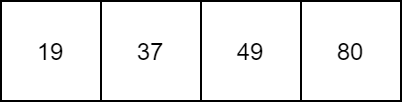
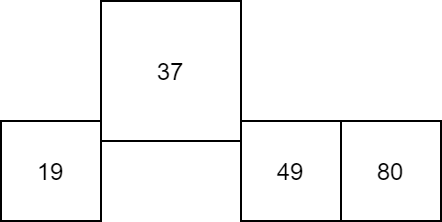
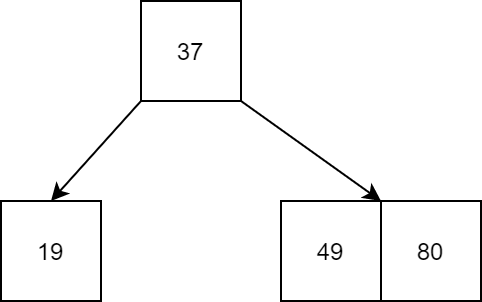
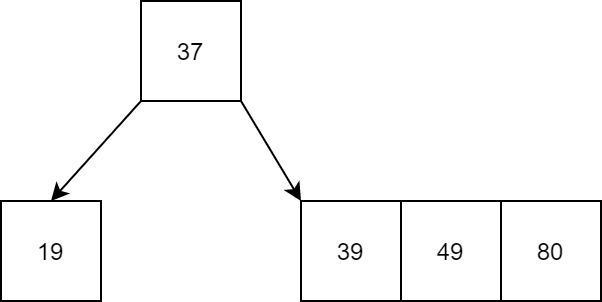
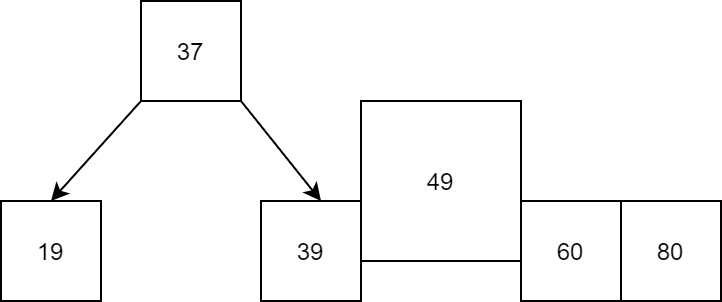
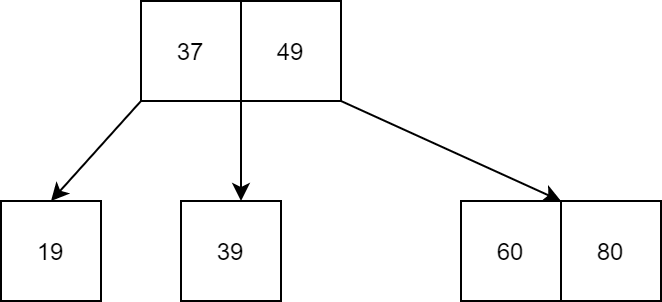
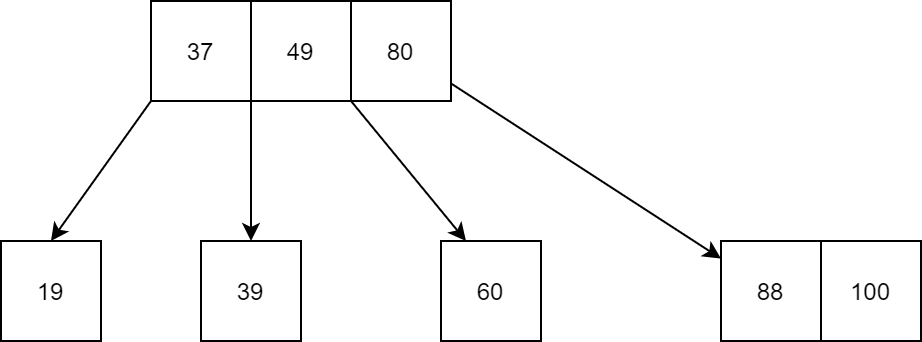


Now, look at the code, to insert a new integer k into the B-tree, first we check root is a null or not!

* If root is a null pointer:  
    
  Then, create a new tree by constructor for root, with t we already set as the order for the tree before, then, set the first key to k, then set number of key to 1.
* If B-Tree already existed:  
  

+ Check if the root is full or not by check if ( root->n == 2 \* t -1). If it is true, then we must create another new node for updating root, split the old root into two roots and make them become children of the new root, which is the root has 1 key of the old root so that it’s still keep the same property of B-tree. If it’s not full, then use the function of insert B-tree that its root isn’t full.

***Now, let check out the visualization of the insertion of b-tree has order 4 in this set of numbers: 37, 49, 80 ,19, 39, 60, 88 and 100.***

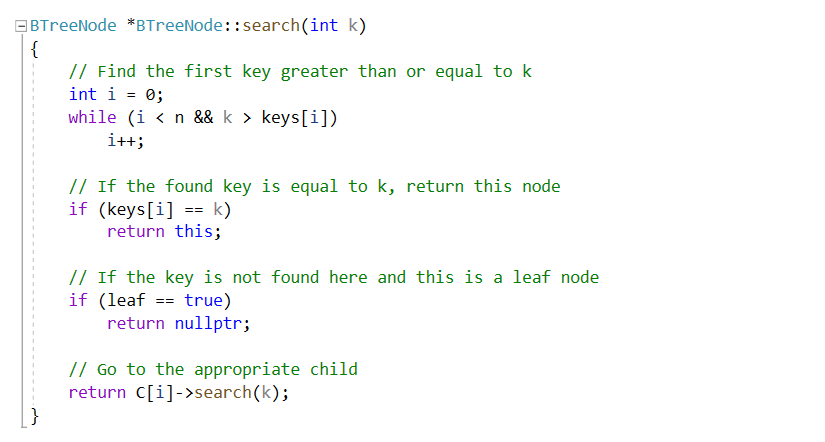
* 37 , 49 and 80 are quite simple, just remember to order them. 
* Add 19: In order, 4 numbers will be: 19 -> 37 -> 49 -> 80 , choose the median number : 37; split other in to 2 children, and their parent will be node with key 37  
    
    
    
    
    
    
    
  
* 39 : add as a key along with 49 and 80.  
  
* 60: same operation when doing with 19, at now, we push 60 upward and split the 39,49,80 node into 2 nodes:  
    
    
    
    
  88 and 100:   
  
* Let check the complexity for a B-tree:
  + Time complexity: O()
  + Space complexity: O(n)

## ***3. Searching for a node:***

This look a bit easier than insertion, however, we should not ignore it at once. Therefore, let take a look of the algorithm:

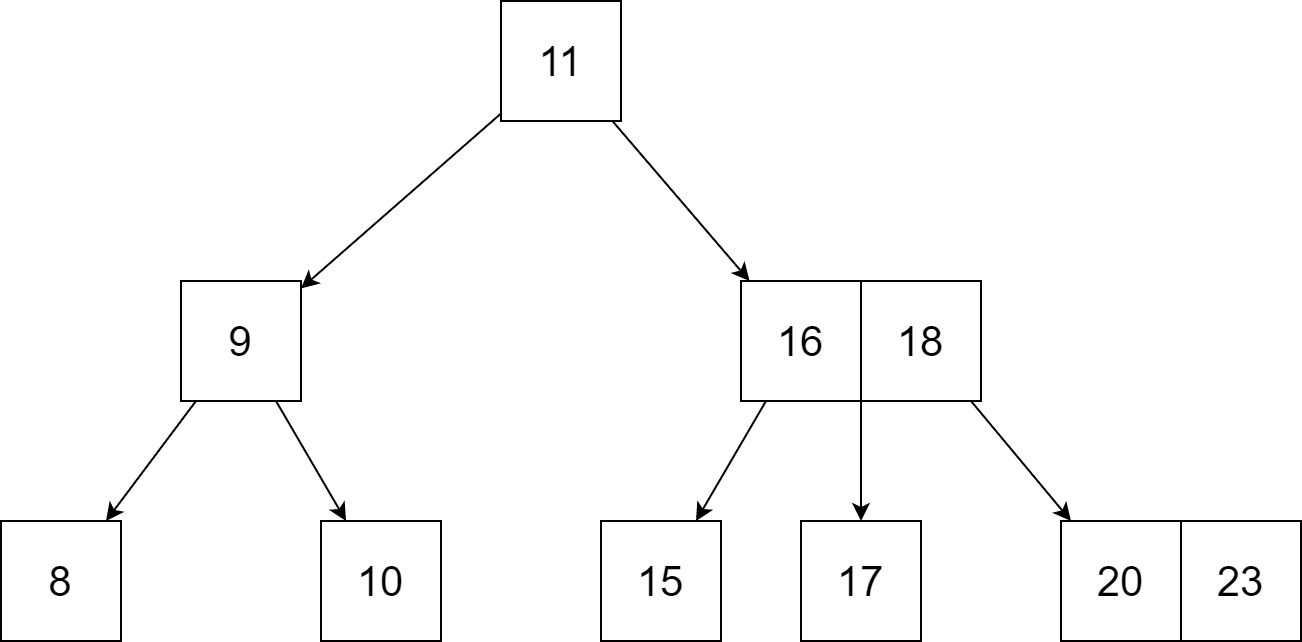
* + Starting from the root node, compare k with the first key of the node.
  + If **k = the first key of the node**, return the node and the index.
  + If **k.leaf = true**, return **NULL** (i.e. not found).
  + If **k < the first key of the root node**, search the left child of this key recursively.
  + If there is more than one key in the current node and **k > the first key**, compare k with the next key in the node.
  + If **k < next key**, search the left child of this key (ie. K lies in between the first and the second keys).
  + Else, search the right child of the key.
  + Repeat steps 1 to 4 until the leaf is reached.

Now, let have a look at the implementation:

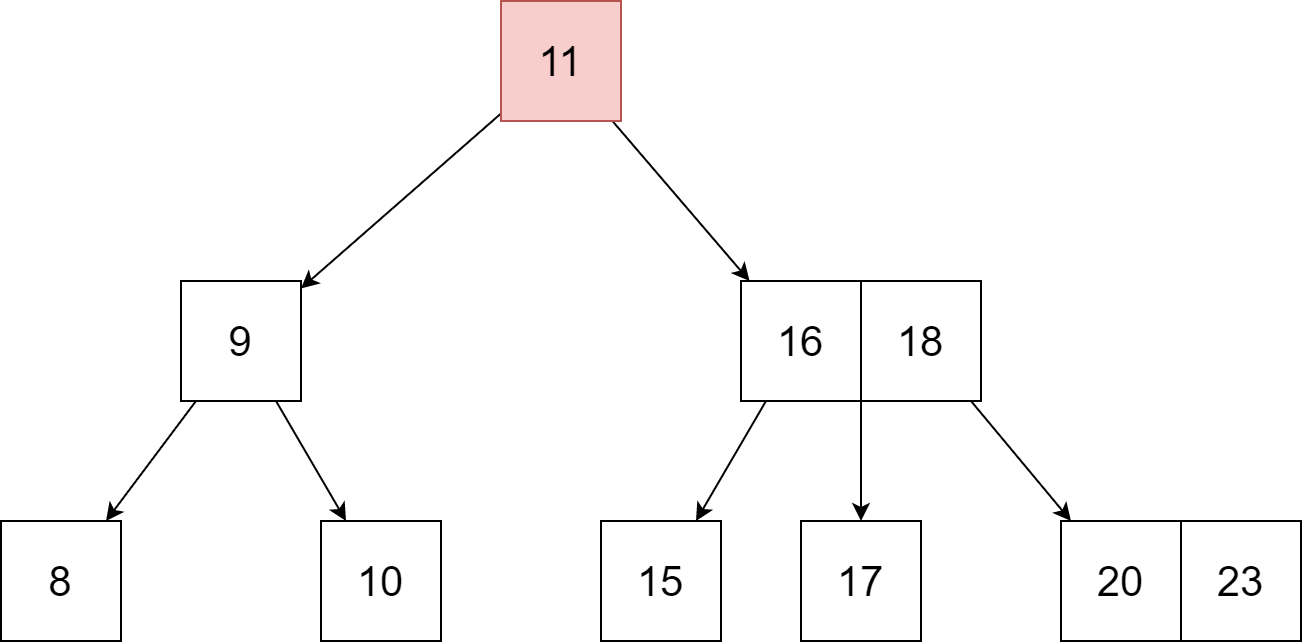
  
To understand the code, we will go to each section, according to the algorithm:

* 1. Find the first key in that node greater or equal to k.  
     At that moment, the pointer is between two value, one greater and one smaller than k or the first or the last pointer.
  2. If at the key we stop, key is equal to k. So this is the node we need to return and stop the function.
  3. But if the key is not found at here and it is a leaf node, so , there is no more node to search, so the node we need to return do not exist .
  4. And in the last case, according to a), we will do the recursion to find the node at children k.

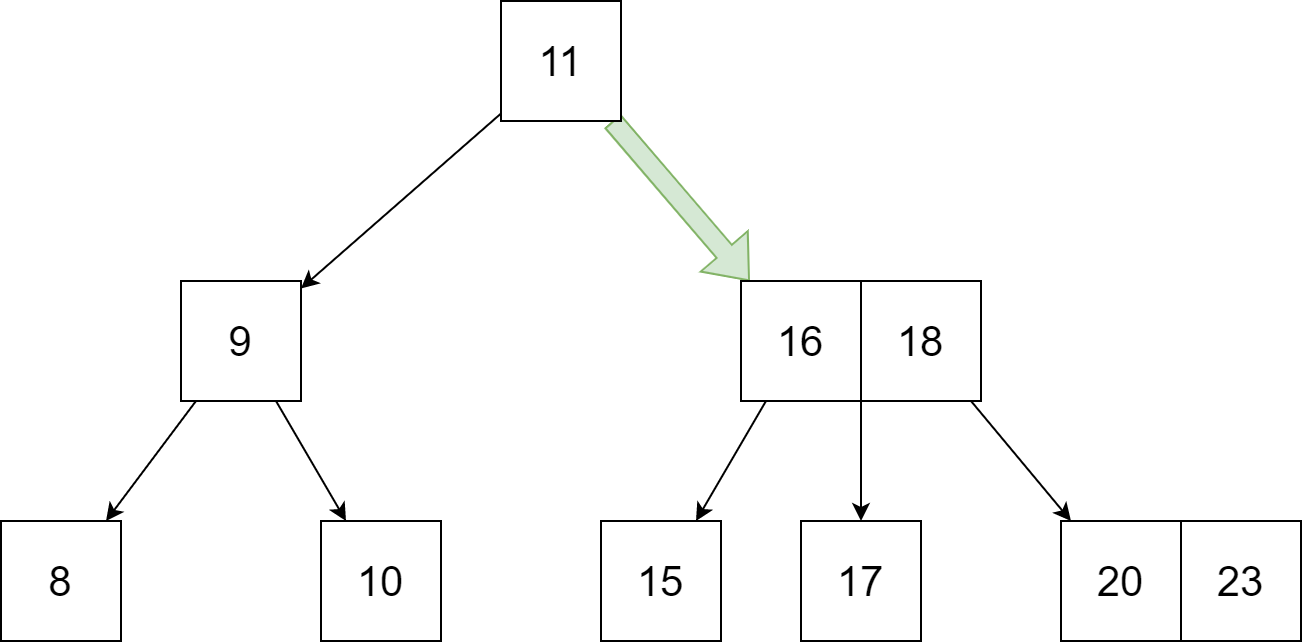
Now, let have a look at this example: Given a B-tree as below:

  
And we need to search 17:

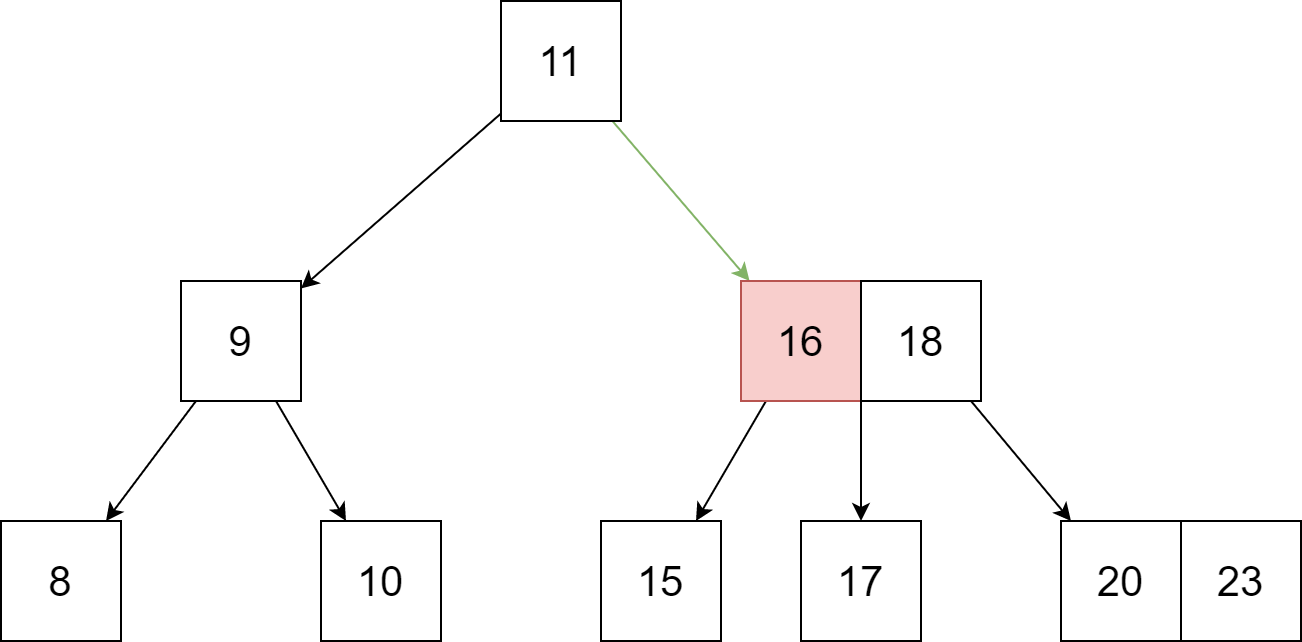
* + 1. k is not found in the root so, compare it with the root key.



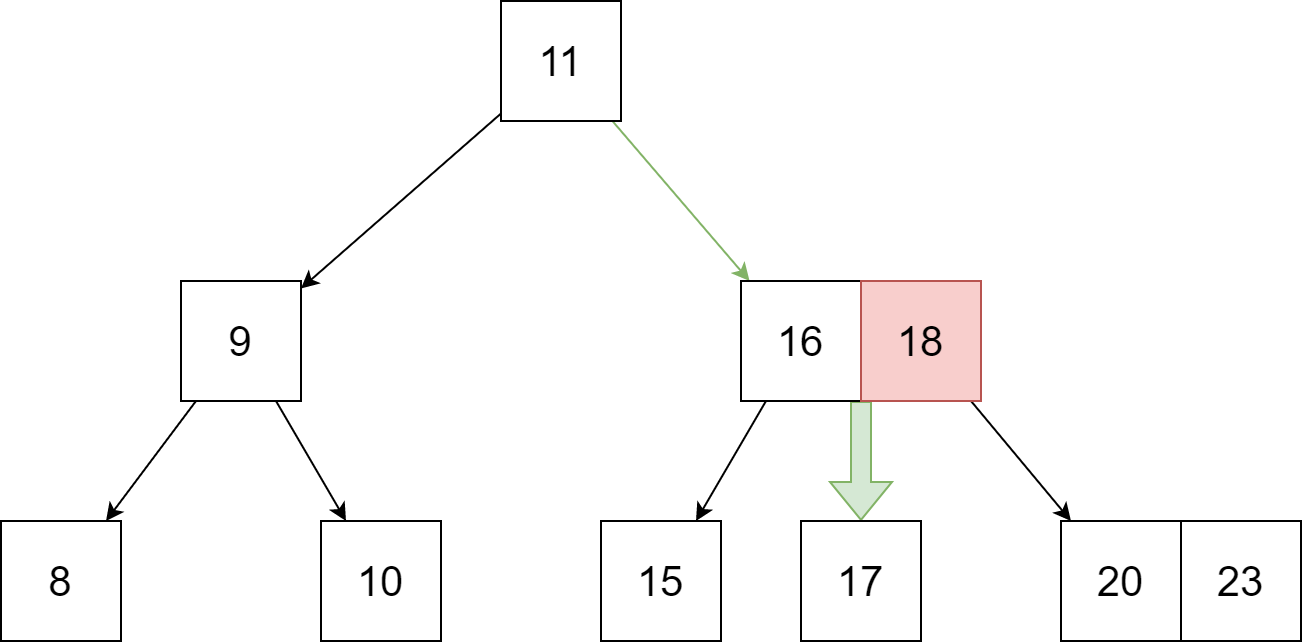
* + 1. Since k > 11, go to the right child of the root node.



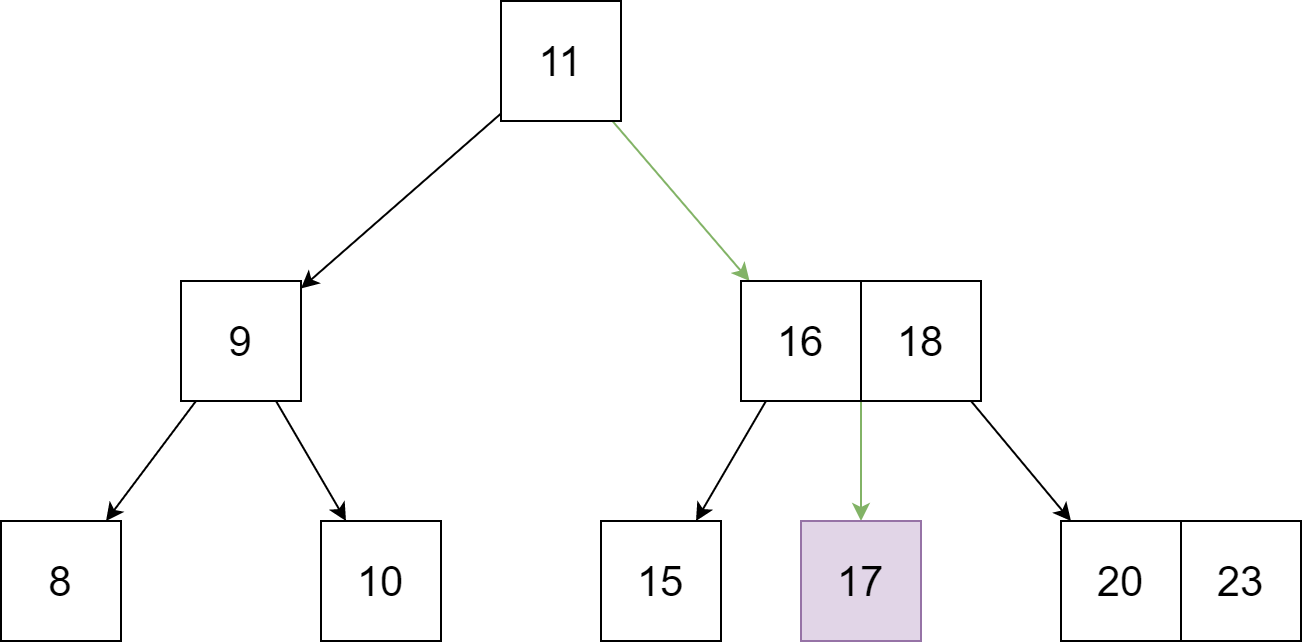
* + 1. Compare k with 16. Since k > 16, compare k with the next key 18.



* + 1. Since k < 18, k lies between 16 and 18. Search in the right child of 16 or the left child of 18.



* + 1. k=17 is found.



* Calculate the complexity:
* Worst case Time complexity: O(log n)
* Average case Time complexity: O(log n)
* Best case Time complexity: O(log n)
* Average case Space complexity: O(n)
* Worst case Space complexity: O(n)

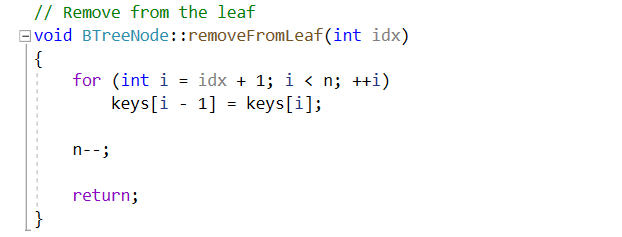
## ***4. Delete a node in B-tree:***

* This is the most tricky part in the operation of B-tree, so before go to the operation and algorithm, let have a look again at the property of B-tree of degree m:

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

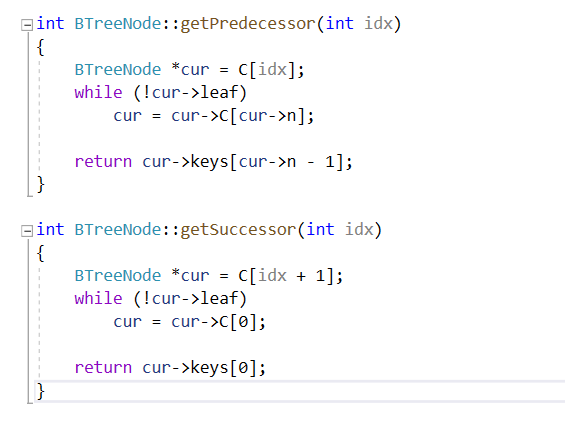
To remove a node in B-tree but still maintain the property, we also need some additional function to support.

**3.1. Remove a key in leaf node.**

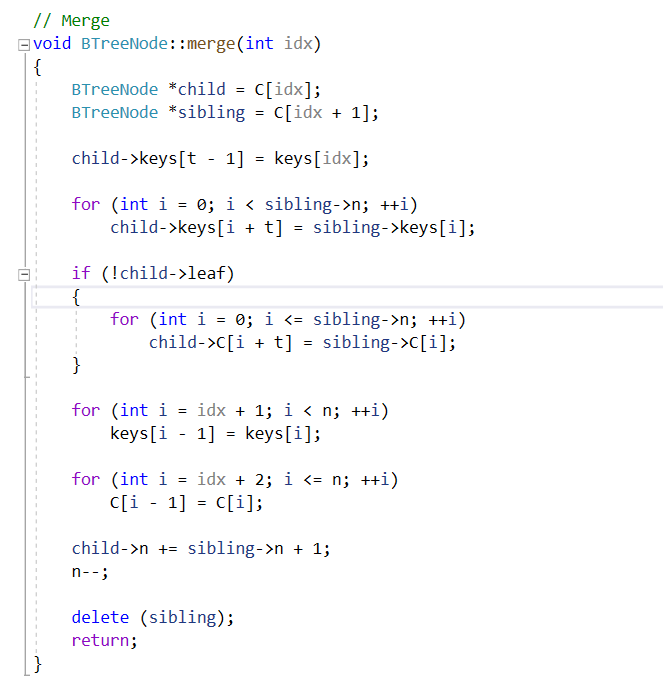


This function will remove an idxth key in a leaf node and arrange other numbers.

**3.2 Replace a node by a successor node or a predecessor node.**

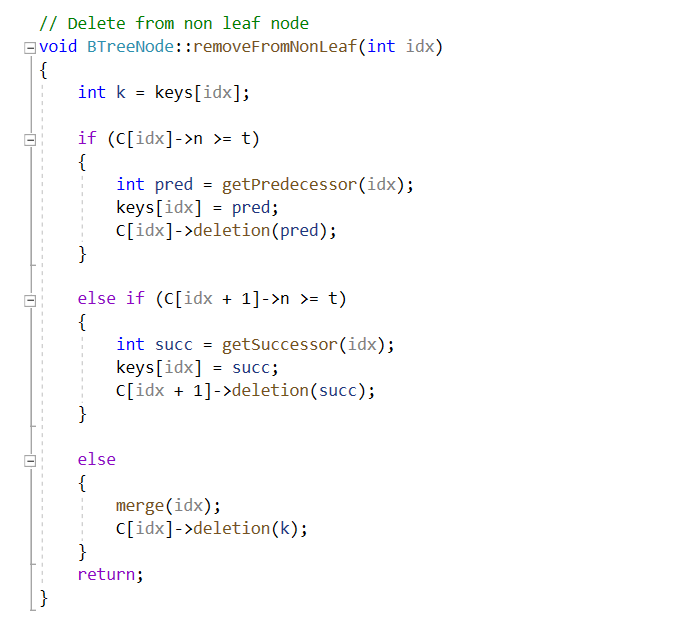
****

**3.3. Merge 2 children into 1 children of a node:**



This function will merge 2 idxth and (idx+1)th children into 1 child.

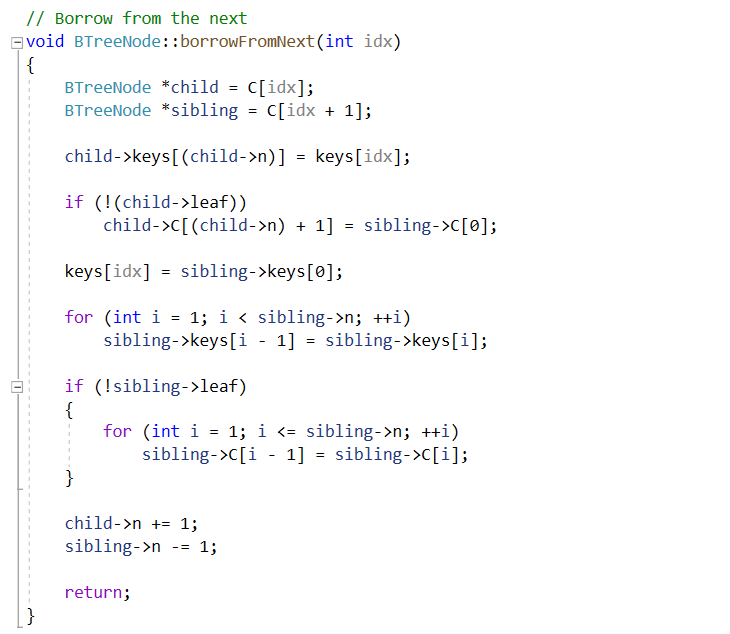
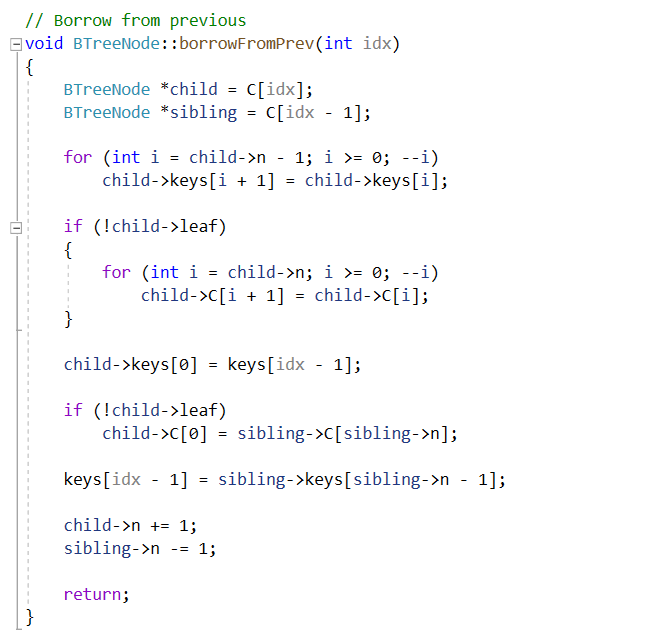
**3.4. Remove a key in an internal node**



There are 3 main cases for this operation:

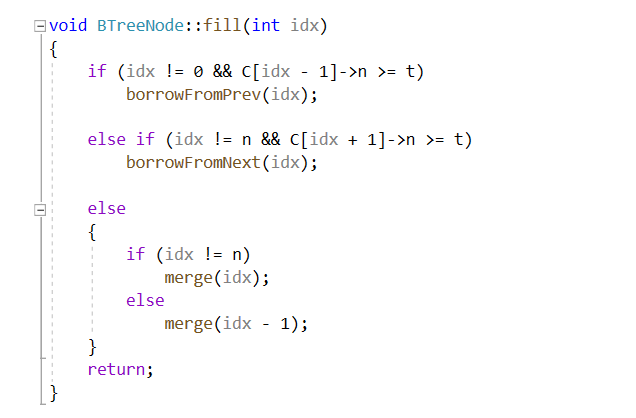
* The internal node, which is deleted, is replaced by an inorder predecessor if the left child has more than the minimum number of keys. ( first *if* block)
* The internal node, which is deleted, is replaced by an inorder successor if the right child has more than the minimum number of keys.(the *else if* block)
* If either child has exactly a minimum number of keys then, merge the left and the right children. ( the *else* block)

**3.5 Borrow a node from the previous and next:**

****

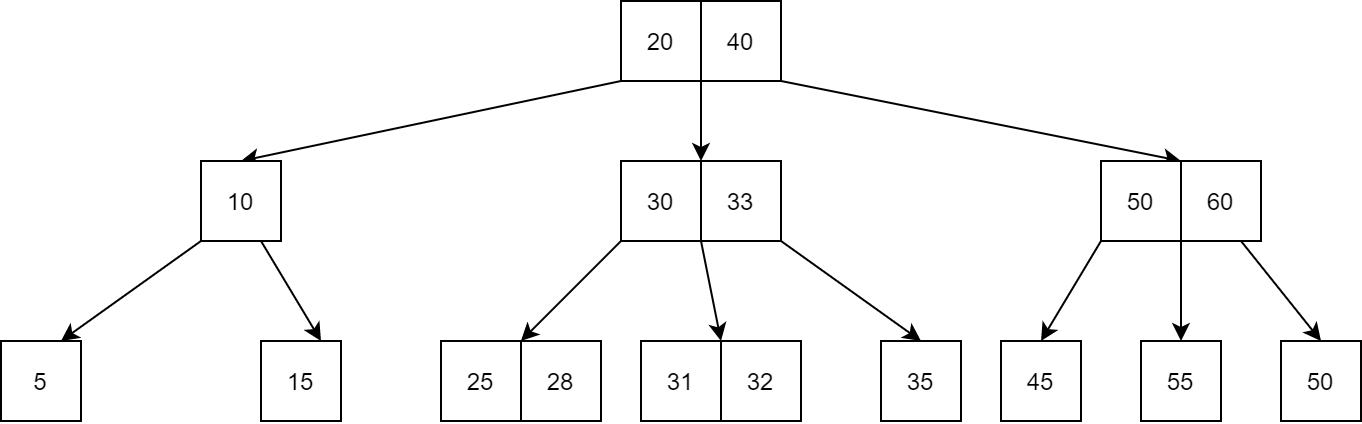
These 2 functions help us borrow the node from their children, the next children or the previous child of a key (for shorter, we can understand it as a left child or right child of a key) and keep the property of a B-tree after delete a node in a special case in the next part of the report.

**3.6 Fill a node after delete:**

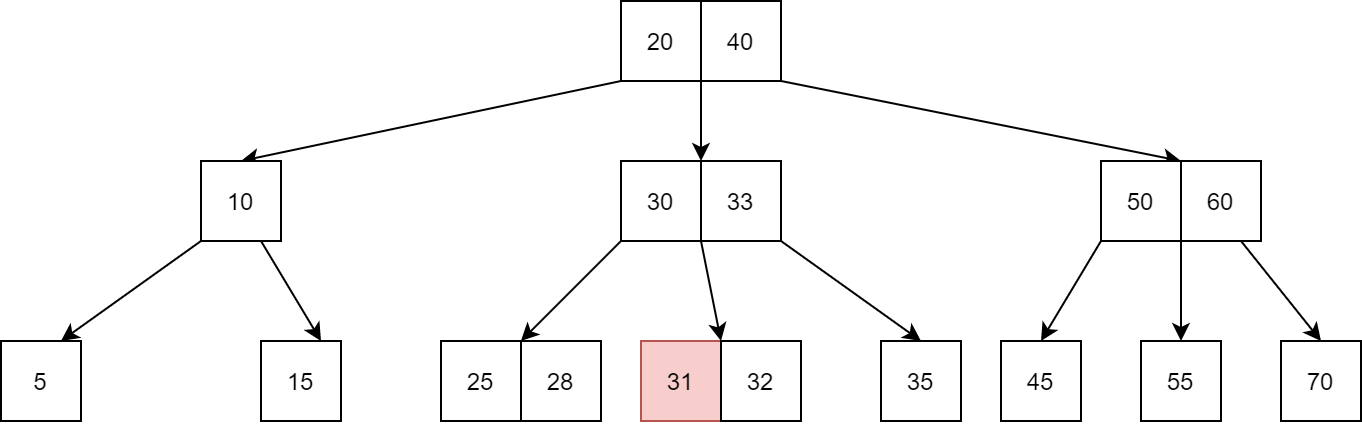
****

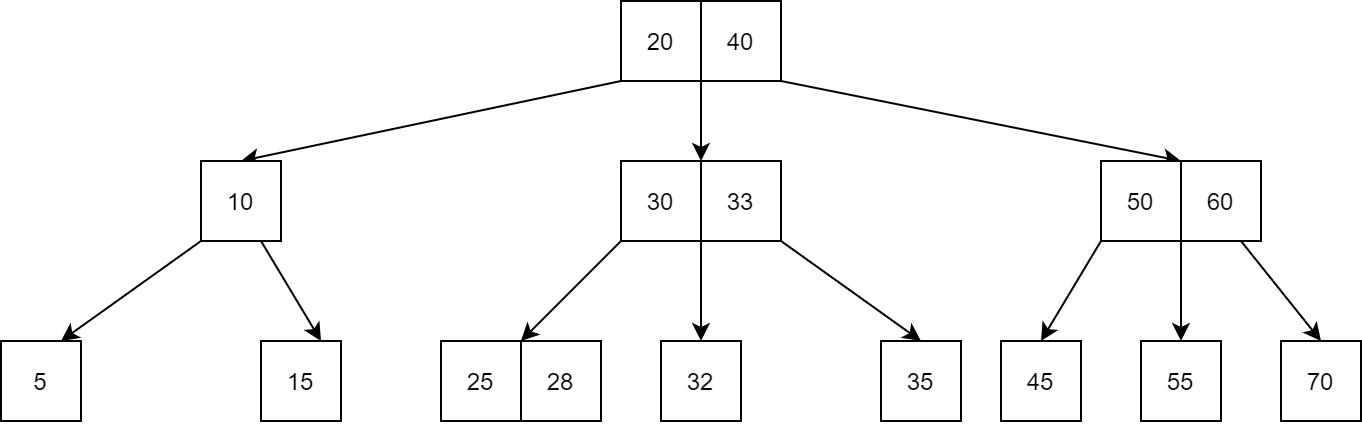
After delete a node, there may be some empty space in tree and we need to fill it to make the tree look “fully”. The main idea of this function is borrow a spare key from the next or previous children or merge 2 children into 1.

* So after these supporting functions, now we go to the main algorithm and operation of delete a number in B-tree, along with example to illustrate:
  + **Case 1**: The key does not exist in the B-tree.
    - It’s easy, no work for that case.
  + **Case 2**: The key in the leaf node. Let consider this B-tree for all of case in **Case 2** and **Case 3**.



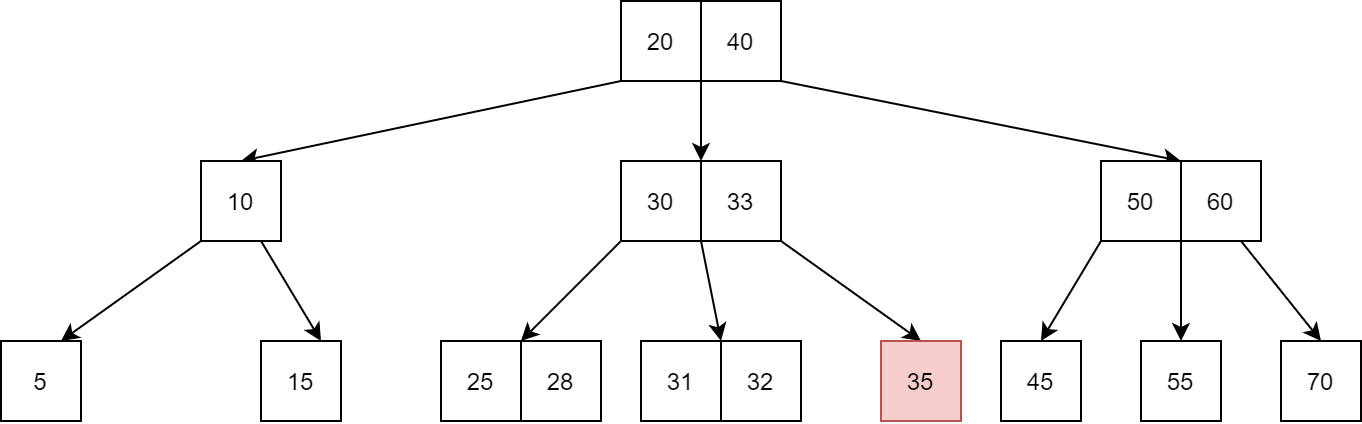
* The deletion of the key does not violate the property of the minimum number of keys a node should hold. In example: delete key **31**



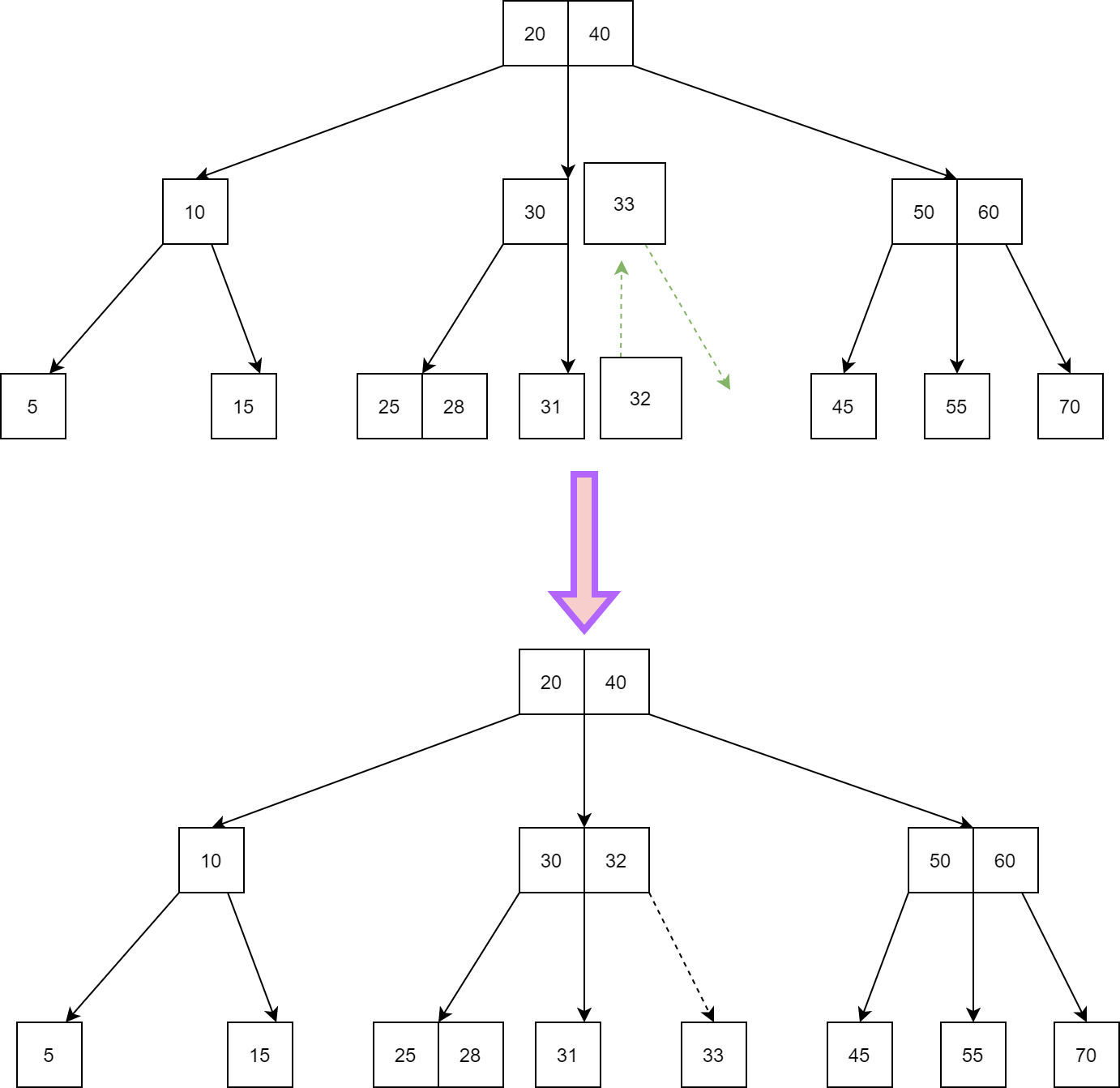
Therefore, we just remove that key, other property still maintain so no more work to do.  
  


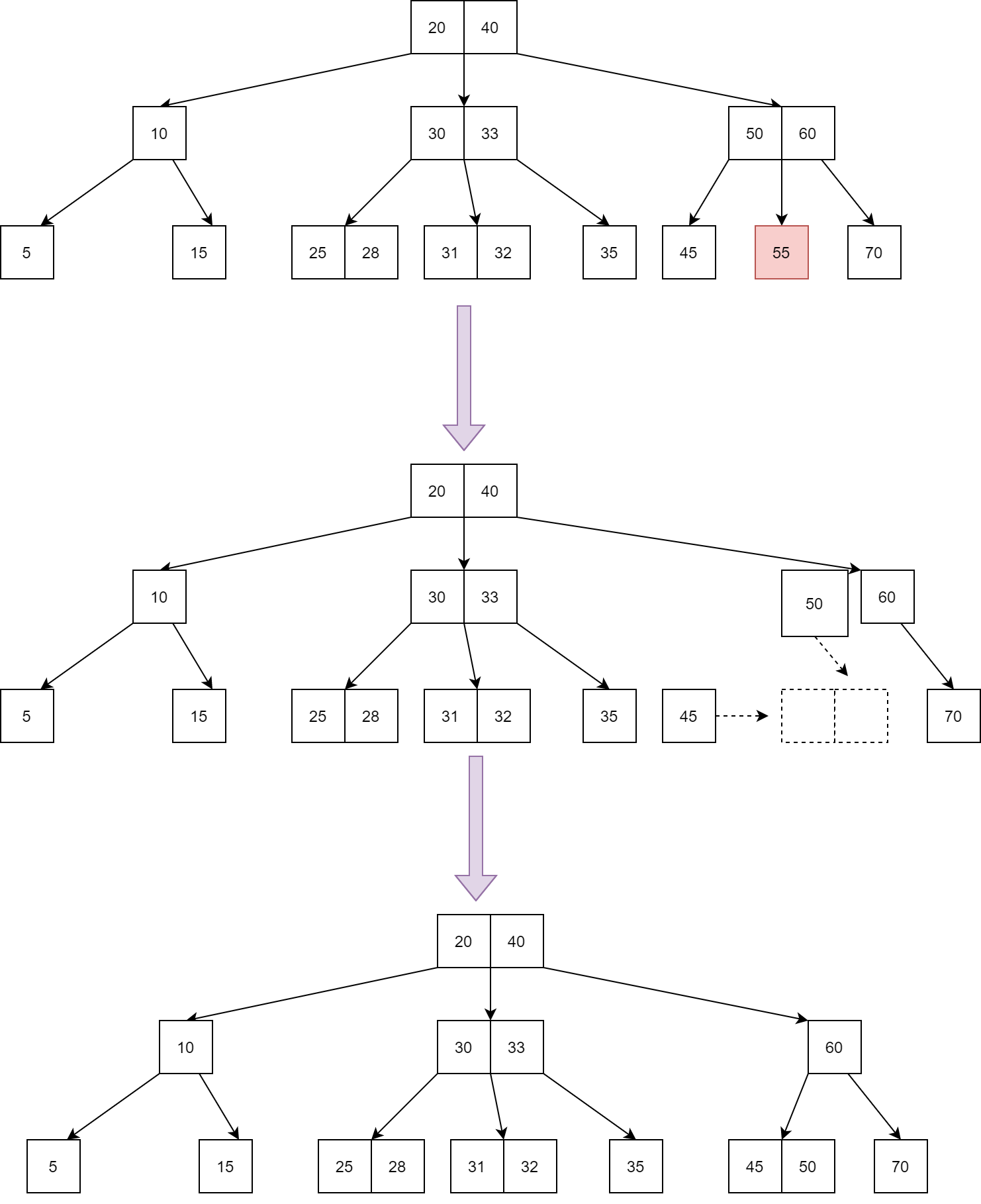
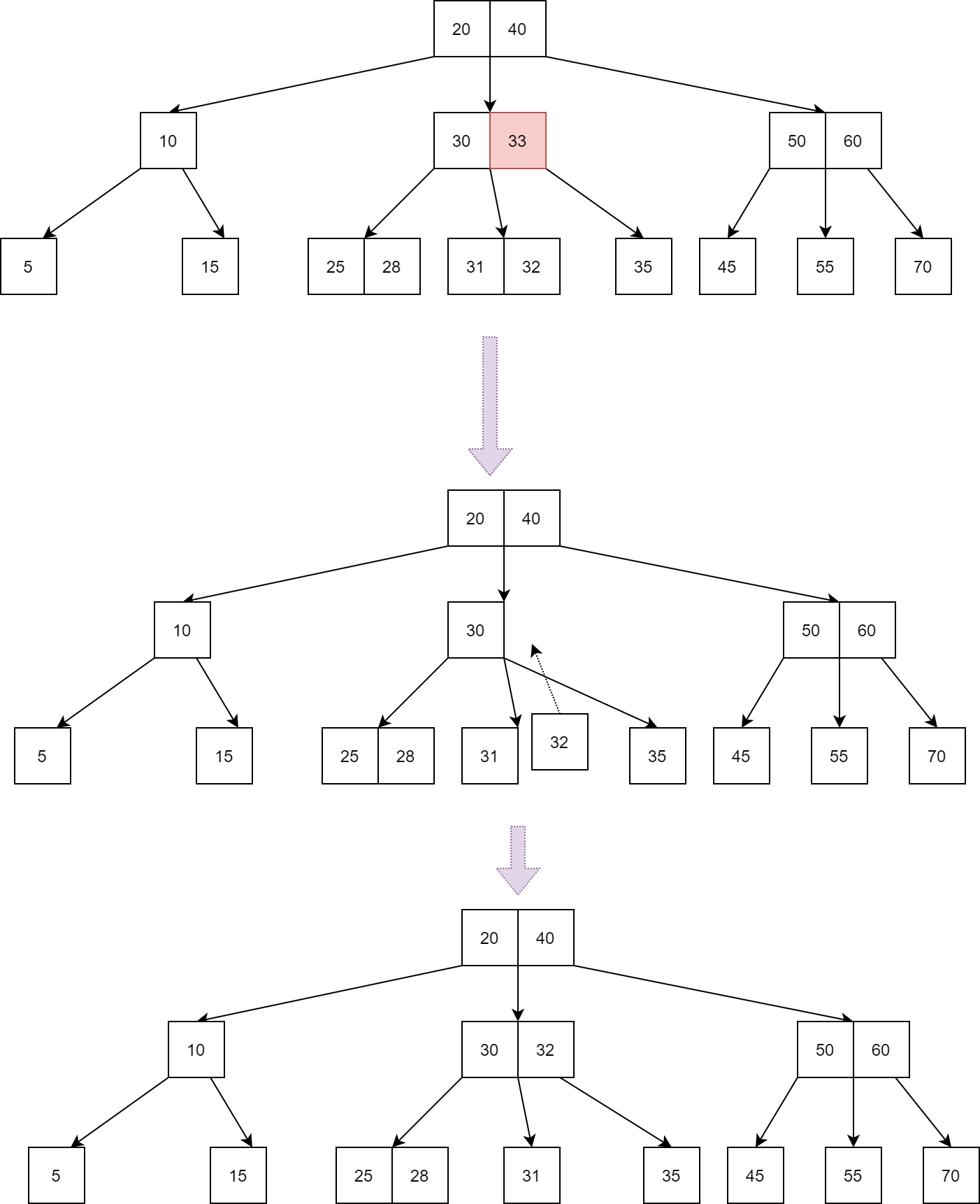
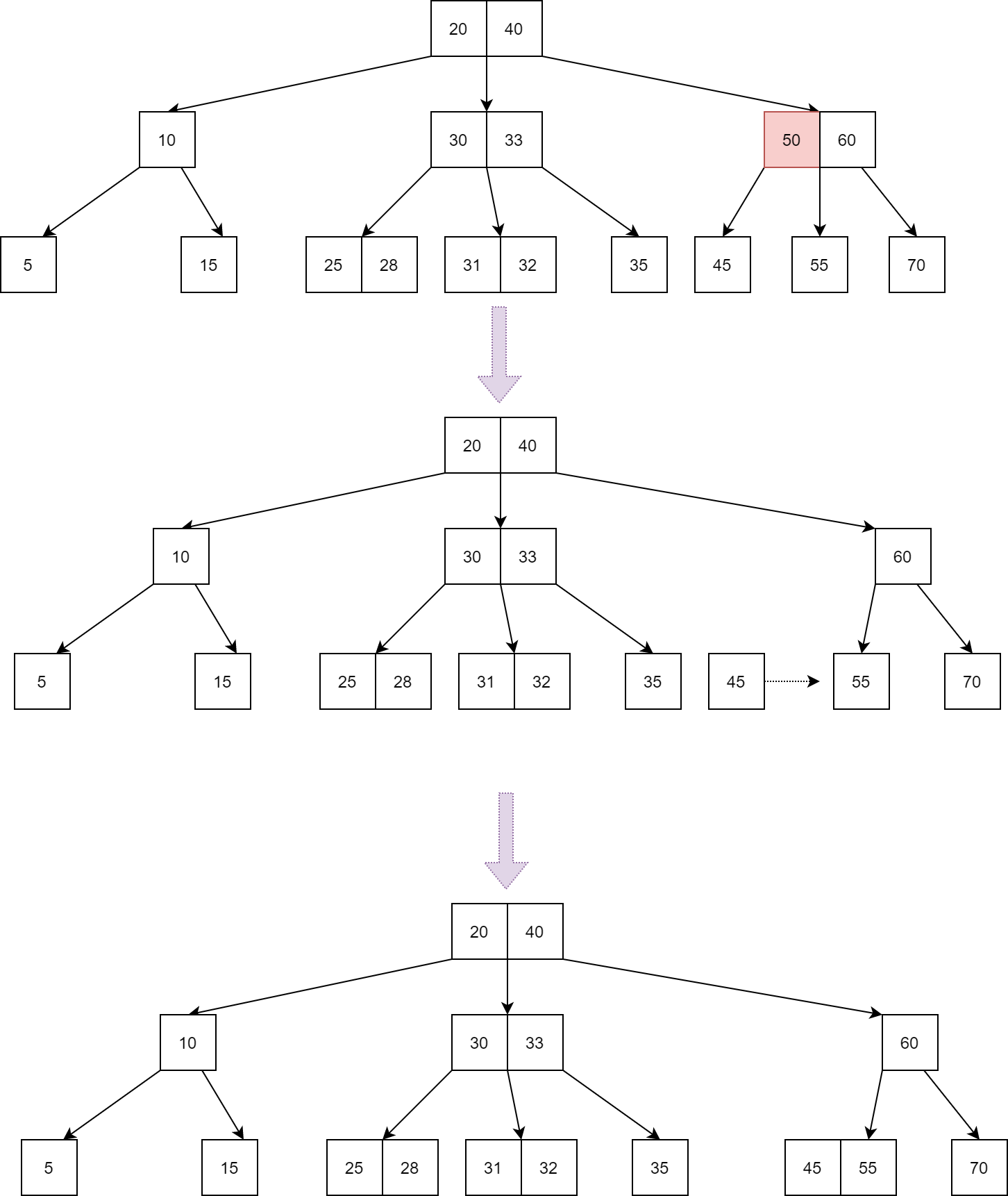
* The deletion of the key violates the property of the minimum number of keys a node should hold. In this case, we borrow a key from its immediate neighboring sibling node in the order of left to right .
  + First, visit the immediate left sibling. If the left sibling node has more than a minimum number of keys, then borrow a key from this node.
  + Else, check to borrow from the immediate right sibling node.

**Example**: delete key **35**.

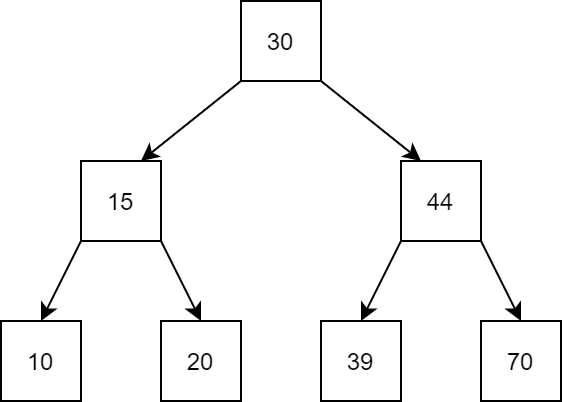
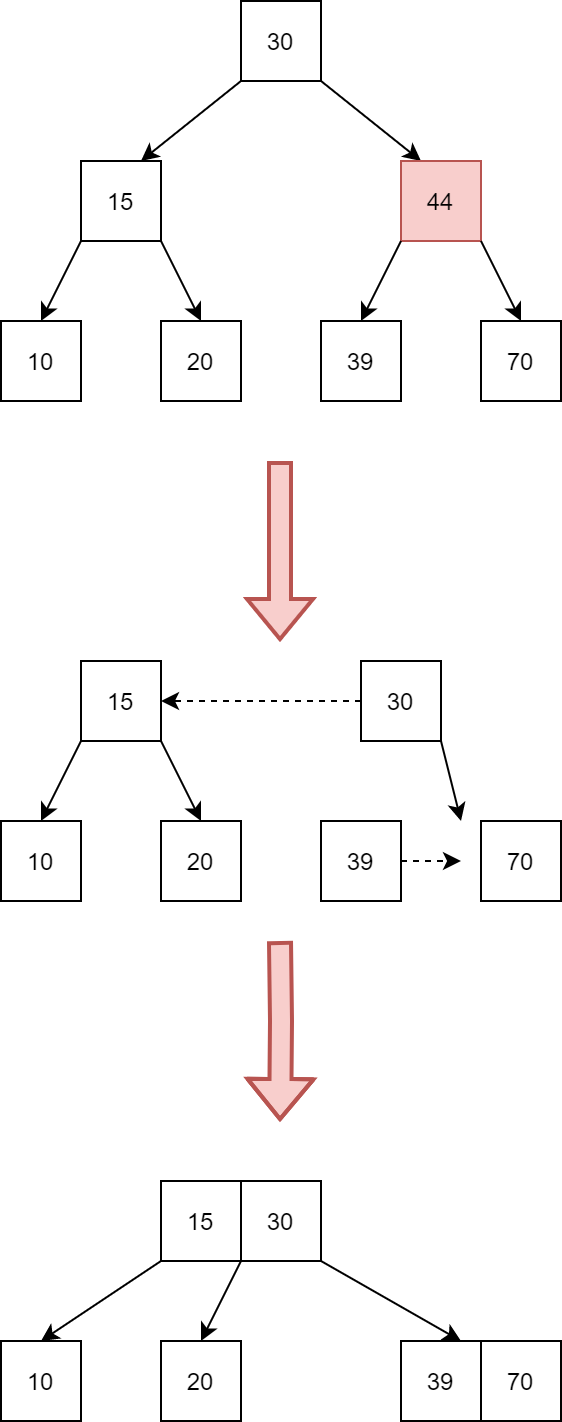


After delete 35:

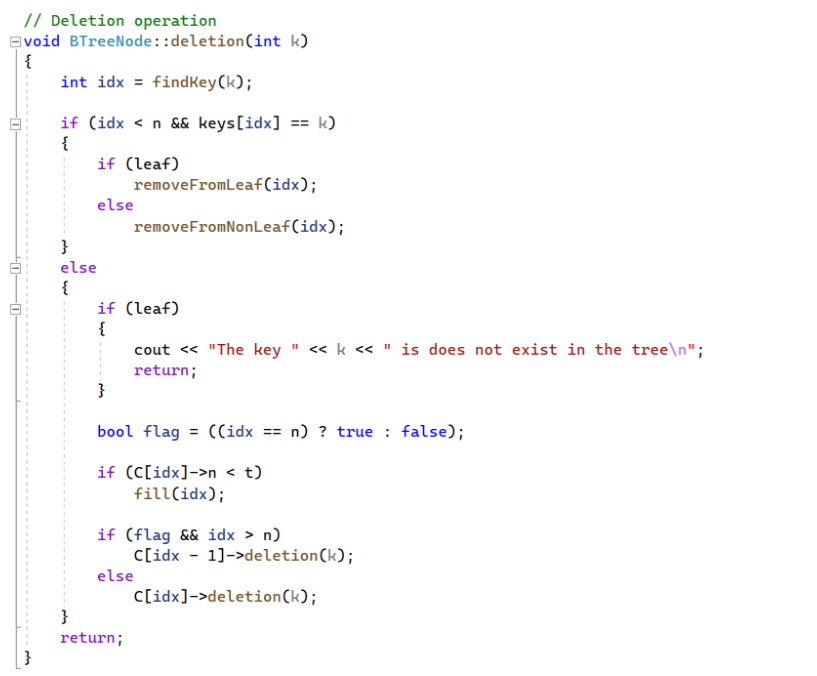


* If both the immediate sibling nodes already have a minimum number of keys, then merge the node with either the left sibling node or the right sibling node. This merging is done through the parent node.
  + **Example:** delete key **55.** 
* Case 3: Delete a key in an internal node:
* The internal node, which is deleted, is replaced by an inorder predecessor or successor if the left child has more than the minimum number of keys.  
  In example: Delete 33.  
  
* If either child has exactly a minimum number of keys then, merge the left and the right children.  
  For example: Delete 50  
  
* Case 4:   
  In this case, the height of the tree shrinks. If the target key lies in an internal node, and the deletion of the key leads to a fewer number of keys in the node (i.e. less than the minimum required), then look for the inorder predecessor and the inorder successor. If both the children contain a minimum number of keys then, borrowing cannot take place. This leads to **Case 3** i.e. merging the children.

Again, look for the sibling to borrow a key. But, if the sibling also has only a minimum number of keys then, merge the node with the sibling along with the parent. Arrange the children accordingly (increasing order).

Let have a look at this tree as an example:  
  
Now, we delete 44:  


So, after reaching all case in our algorithm and operation, let’s look at the implementation:



This function includes all cases above. Now, let check the complexity:

* Best case Time complexity: O(log n)
* Average case Space complexity: O(n)
* Worst case Space complexity: O(n)

**§ SUMMERIZE ABOUT COMPLEXITY:**

1. **Insert:**
   1. Time complexity: O()
   2. Space complexity: O(n)
2. **Search:**
   1. Worst case Time complexity: O(log n)
   2. Average case Time complexity: O(log n)
   3. Best case Time complexity: O(log n)
   4. Average case Space complexity: O(n)
   5. Worst case Space complexity: O(n)
3. **Delete:**
   1. Best case Time complexity: O(log n)
   2. Average case Space complexity: O(n)
   3. Worst case Space complexity: O(n)

After all, we hope all of us will have a better view at one of the very useful data structures in coding and improve on the algorithm when working with other different kinds of data structure.

***Reference link:***

<https://www.programiz.com/dsa/b-tree>

<https://www.javatpoint.com/b-tree>

<https://en.wikipedia.org/wiki/B-tree>