## Chapter 3.2 B - Tree

Although One Level or Two Level Index can help speed up Query, normally in the Business System uses one more normal structure, which is called B - Tree, but the most normally used is called B + Tree.

* *B - Tree can keep adaptive index level with Database File automatically.*
* *Manage all used Storage Block and keep each Block between Half - Full and Full.*

This chapter would focus on B + Tree but not Tree.

### Chapter 3.2.1 Structure of B - Tree

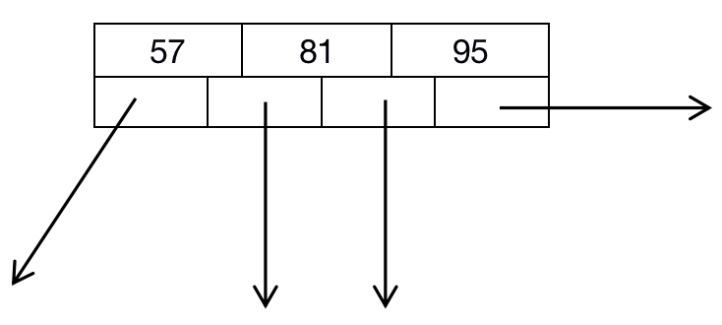
*Organization:*

B - Tree organizes all Storage Blocks into One Tree. This Tree is balanced, which means that all paths from Tree Root to Tree Leaf are the same. Normally, B - Tree has three levels: Root Level, Internal Level and Leaf Level, but also it can be random levels.

*Example:*

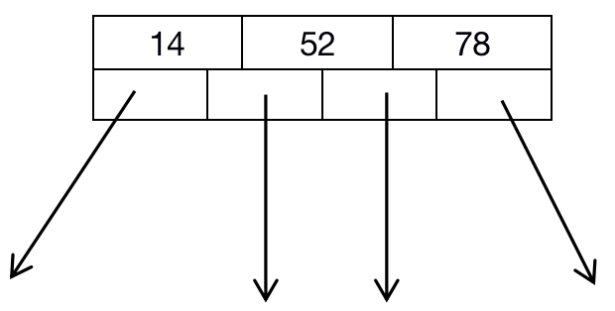
The typical *B - Tree Node:*

*Among B - Tree Nodes, there have three keys, which are 57, 81 and 95. The first three pointers point to the Tree Node which has the exact key value and the last pointer points to Next Node which has a bigger value. This is exact leaf situation, if this leaf node is the last one in the sequence, then the pointer equals to Null.*



The typical *B - Tree Internal Node:*

*Among B - Tree Internal Nodes, there have three keys, 14, 52, and 78. This Node have four pointers, through the first node with Key 14, we can reach all keys which are less than Key 14. Through the second node with Key 52, we can reach those nodes whose nodes are bigger than Key 14 and less than Key 52. Through the third node with Key 78, we can reach those nodes whose nodes are bigger than Key 52 and less than Key 78.*



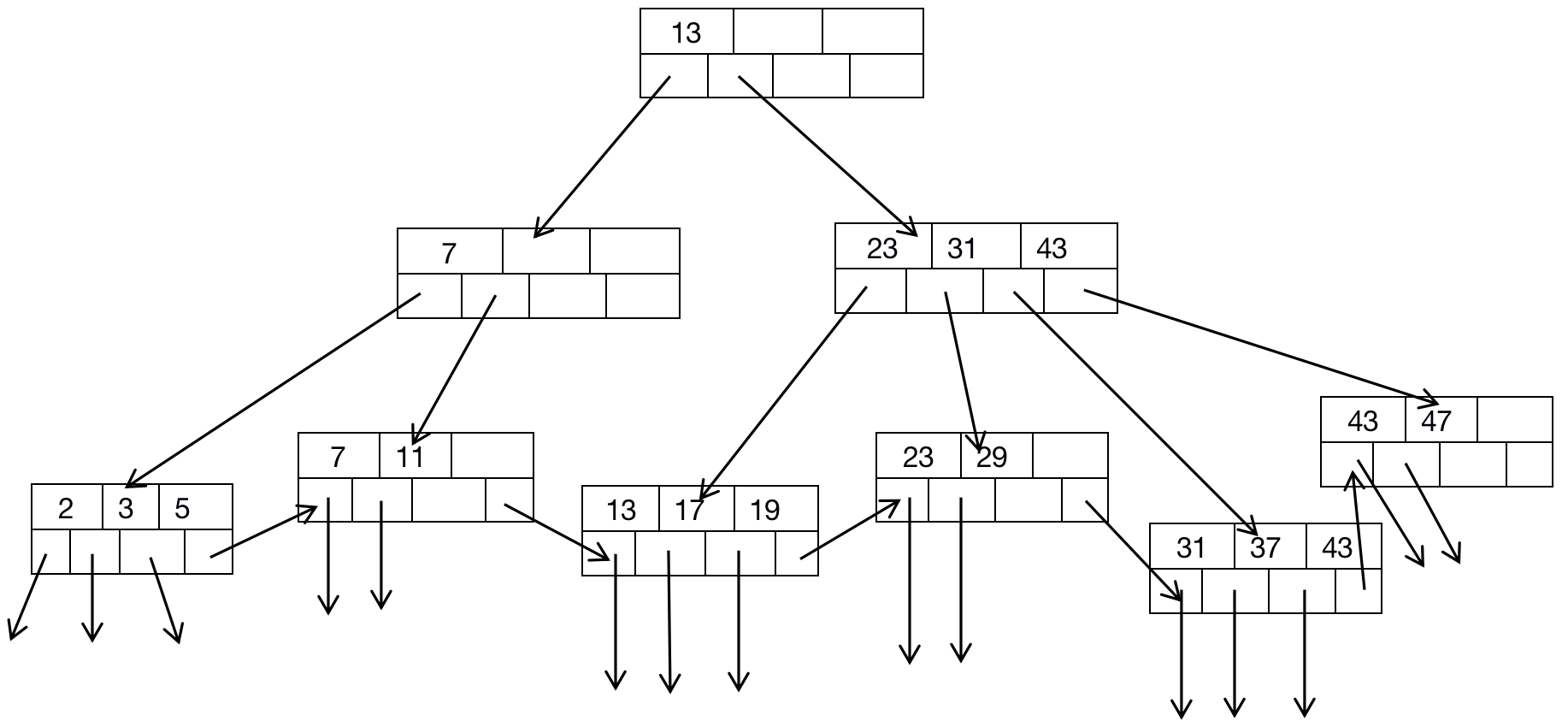
*Attention that, there has no need to fill all nodes with Keys and Pointers. When value of n equals to 3, then in the internal Node, there should have at least one key and two pointers.*

The typical *B - Tree:*

*Among Three Level B - Tree, n = 3. We assume that all keys among B - Tree belong to 2 to 47. Attention that, those values would exist once in Leaf for once. Each Tree Leaf Node has 2 - 3 Key - Value pair, and there also has the pointer which points to the next node. When we look from left to right, then they all have been sorted ascending.*

*There has only two nodes in the Tree Root, which is just the allowed pointer number. Although it is allowed at most 4 pointers, and key value in Root Node divides the tree into two parts with the key value which can be visited from first Tree Node and second Tree Node, which is to say, left child tree with Key Values which are less than key value 13 and the right child tree with Key Values which equals to or bigger than key value 13.*

*Attention that, there has Root Node with the four pointer values, which range from 23, 31, and 43. So the first part of the Tree would be Key Values which are less than 23, the second part of Tree would be Key Values which equal to or bigger than 23, less than 31, and the third part of Tree would be Key Values which equal to Key Value 43 or bigger than Key Value 43.*



*Principle:*

For each B - Tree Structure, there must have a parameter which is n, it decides the structure of all Storage Block. Each Storage Block would store n keys and n + 1 pointers. From some kind of meaning, the Storage Block of B - Tree also has one extra pointer which is used to point to the Next B - Tree Node. We need to make the value of n as big as possible.

*Example:*

Assume that we have the size of 4096 bytes of Storage Block, and the integer value occupies 4 bytes, the pointer occupies 8 bytes. As long as we do not need to consider the occupation size of Storage Block Head, then we hope to find the integer n which has the biggest value. 4 \* n + 8 \* (n + 1) <= 4096, then n takes 340.

*Rules of B - Tree:*

*Key in Leaf Node:*

* The keys in Leaf Node are all key copies of Data File, these keys are sorted, and distributed in all Leaf Nodes.

*Root Node:*

* There would be at least two pointers in the Root Node. All nodes point to the Tree Node in the next Level of B - Tree.

*Leaf Node:*

* In Leaf Node, the last node would points to the next Tree Node Storage Block, whose keys are all equal or bigger than those of Current Node Storage Block. In all other n pointers in Leaf Node, there would at least [ (n + 1) / 2 ] pointers to points to Data Records. Unused pointers would be seen as Null pointer and point to nowhere. If the ith pointer has been used, then it would points to the ith record.

*Internal Node:*

* In Internal Tree Node, all n + 1 pointers can be used to point to Storage Block of Tree Node in the Next Level of B - Tree. As the same, there would be at least [ (n + 1) / 2 ] pointers being used. If there are j pointers used, then also have j - 1 keys in the Storage Block, here, assume that these keys are K1, K2, K3, ..., Kj - 1. The First pointer points to those keys that are less than the Key K1. The Second pointer points to those keys that equals to or bigger than Key K1 but less than Key K2...At last, the Last pointer points to those keys that are bigger than Kj - 1.

*(Attention that, those keys are far less than K1 or much bigger than Kj - 1 can not be accessed by this block, but can be accessed by the same other blocks.)*

*(N + 1)th Leaf Node Pointer:*

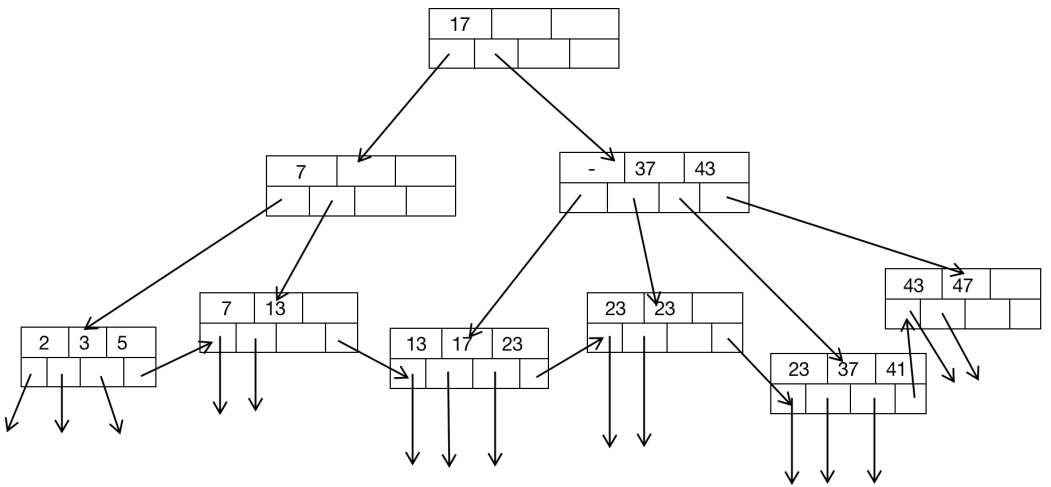
* All used keys and pointers are normally stay in start of Data Block, and the (n + 1)th Leaf Node Pointer is one exception, it points to the next Leaf Node.

### Chapter 3.2.2 Application of B - Tree

B - Tree is one of power tool which can be used to construct Index. Below are some useful instances:

1. *Query Key in B - Tree are Main Key in Data File, and Index are Dense Index, which is to say, the Leaf Node in the Data File records each Key - Pointer Pair. The Data File can be sorted according to Main Key, but it also can be not sorted according to Main Key.*
2. *Data File is sorted according to Main Key, and B + Tree is Sparse Index. Set up one Key - Data Pair in each block in Leaf Node.*
3. *Data File is sorted according to Non - Main Key, and this attribute is the Query Key of B + Tree. The Leaf Node would set up one Key - Data Pair for each attribute value in Data file.*

Another variant of B - Tree enables *Duplication Keys* exist in some Applications. Below is such a B - Tree:



*Definition:*

If we do allow the existence of Duplication Keys in B - Tree, then we need to modify the definition of Internal Node. Assume that Keys in one Internal Node is K1, K2, ..., Kn, then Ki would be the smallest new value which can be visited from ( i + 1 )th Sub - Tree. *‘New’ means that Key Value has not existed in the left of sub - Tree, but Key Value has existed for once in the current Sub - Tree.*

*(Attention that, in some situation, Ki can be null, but the corresponding pointer is still needed.)*

*Example:*

For B - Tree with no duplicates turns to with duplicates, we need to update Key Value of Root Node. To be more clear, we need to update the value of 13 to 17. Although 13 is the smallest key value in the left Tree,but it is not the newest value in this Sub - Tree, because it has appeared in the Left Tree.

We still need to do some updates of the second child node of the Root Node. The second node value equals to 37, since it is the new key value in the fifth Node from left to right. More interesting is that, the first Key Value equals to Null, because the forth Node has no new Key Value. If we want to find the key value 17, then we need to start from the first node in Second Part of B - Tree but not the second node. Or we can find the needed 23, we also need to start from the first node of Second Part of B - Tree.

*Attention:*

*Query 13:*

Query from the first Leaf Node but not the second Leaf Node when we want to query node with key value 13.

*Query value among 24 - 36:*

Query from the third Leaf Node, but when we can not find the required key value, then we do not need to proceed to the right and continue. For example, if Key Value 24 exists in the Leaf Node, then it would exist in the fifth Tree Node, and substitute the value of 24 as 37.

### Chapter 3.2.3 Query B - Tree

### Assume that there has no Duplication Key in Leaf Node and B - Tree is Dense Index, since each Query Key appears in Data File would appear in Leaf Node.

### Assume we have a B - Tree Index, and we want to find record with Query Key K. We need to find from Root to Leaf recursively, the Query Process is:

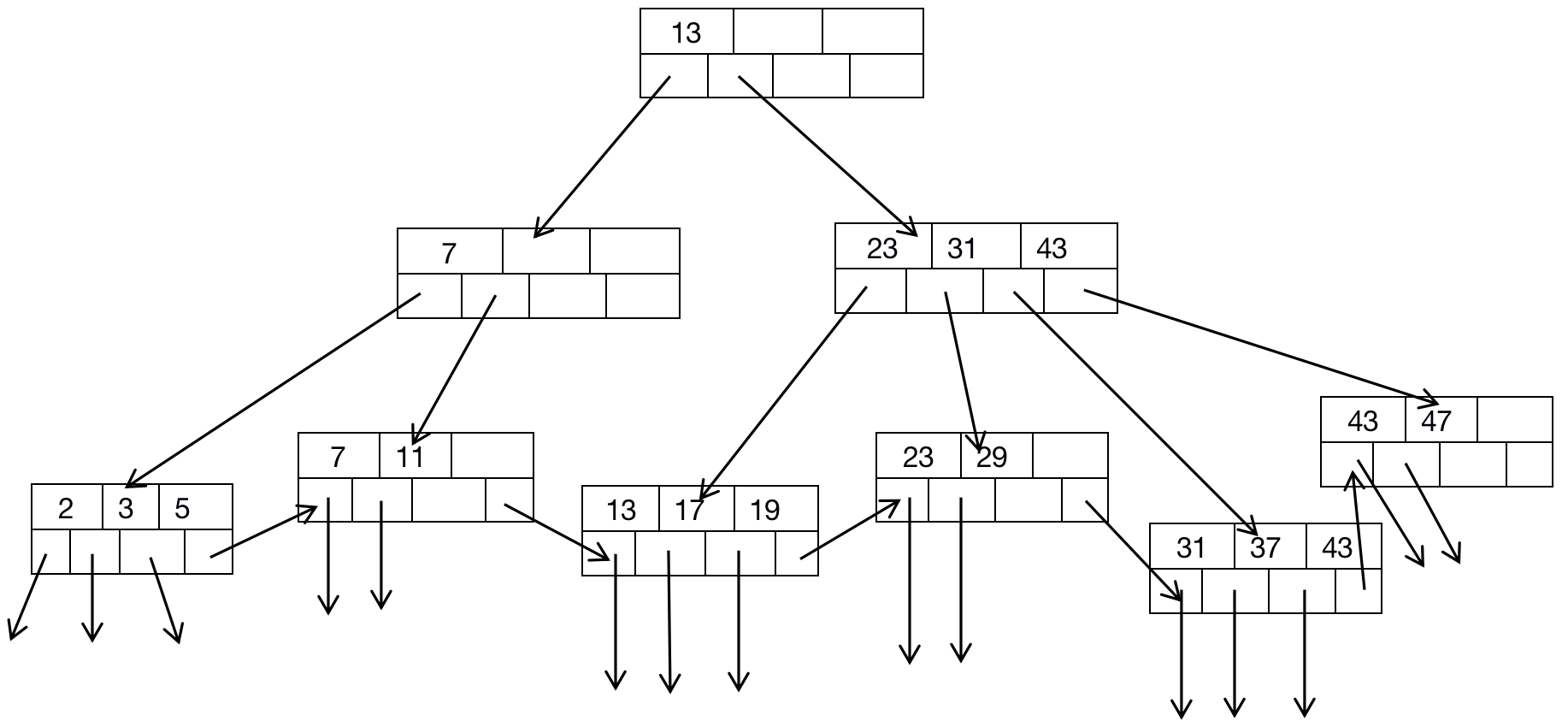
### *Basics:*

*If we stay in the Leaf Node, then we need to query from Key Value field. If the ith key equals to K, then we can get required value from ith pointer.*

### *Conclusion:*

### *If we stay in some Internal Node, and its Key Value are K1, k2, ..., Kn, we need to decide which route to go and query, which is to say, there has only one Leaf Node with Key Value K. If K is less than Ki, then it would be the first Node. If K1 <= K < K2, then it would be the second Node. We need to query the node recursively.*

### *Example:*

Assume we have one B - Tree, and we need to find the record with Key Value 40. 

We start from Root Node, among which there has one Key Value 13. Since 13 is less than 40, then we turn left and find the second B - Tree node with Key Value of 23, 31, and 43.

In this Node, we find that 31 <= 40 < 43, therefore we go to fifth Tree Node with Key Value 31, 37 and 43, among which there has not got the Key Value 40. So we can make sure that there has no Key Value 43 in all Leaf Node.

Attention that, if we want to find the node with Key Value 37, just as the steps above, we go to the fifth Tree Node and find the Key Value 37. So we can go further to query the Record with Key Value 37.

### Chapter 3.2.4 Range Query

### *Principle:*

### B - Tree is not only useful to Query the single Key Value but also useful to find the Key with Range. In general, Range Query would include one field to compare the Query Key with Single Value or Multiple Value, which would include ‘=’, ‘>’, ‘<’.

### *Example:*

### SELECT \* FROM R WHERE R.k > 40;

SELECT \* FROM R WHERE R.k >= AND R.k <= 25;

### *Rule:*

*If we want to look for all Key Values from B - Tree Node with Range [a, b], then we need to query the Key Value a. No matter whether it exists or not, we may need to reach Leaf Node with Key Value a or with Key Value which is bigger than a. Also, as long as there does not exist Key Value is less than Key Value b, then we need go and check the next Leaf Node.*

*( Attention that, if b equals to +8, then we go to Last Node of B - Tree. Otherwise, if a is -8, then we go to the first Node of B - Tree. )*

*Example:*

Assume that we have a B - Tree as below, and we need to find the Range Query with (10, 25). At first, we need to confirm Query Key 10, and we locate at the second Leaf Node, it’s first Key Value is less than 10, but the second Key Value is bigger than 10. Then we find through it and find the Record with Key Value 11.

There has no other Leaf Node in the Second Leaf Node, therefore, we need to query through the third Leaf Node, whose Key Value equals to 13, 17, and 19, which are all less than 25. But we still need to go to the next Leaf Node with Key Value 23 which satisfies the condition. But the next Key Value which equals to 29 and exceeds the Key Value 25, therefore, we stop here and finish our Query. So we find five Key Leaf Node with Key Values in the given Range (10, 25).

### Chapter 3.2.5 Insertion into B - Tree

When we consider how to insert one new node into B - Tree, then we can find that B - Tree is much better than Simple Multi - Level Index.

*Insertion Principle - Recursively Rule:*

* *Try to find space in adaptive Leaf Node for the New Key, if there has any space, then we can put Key there.*
* *If there has no such space in the Leaf Node, then we can split the Leaf Node into two and divide those Key Values into two New Leaf Node, which can make each new Leaf Node has half or more than half Key Values.*
* *The Node Split for the Level looks like to insert a new Leaf Node in higher Level. Therefore, we can use this insertion Strategy in the high level recursively: If there has extra space, we need to insert; Otherwise, we need to split Father Node and proceeds to higher Level of B - Tree.*
* *The exception situation is that, if we try to insert the Key into the Root Node and there has no space in the Root Node, then we need to split the Root Node and split into two Nodes. Also further create one newer Node and this new Root Node has two split Nodes as Sub - Node. Recall that, no matter how big n is, the Root Node is always allowed to have one Key and Two Sub - Nodes.*

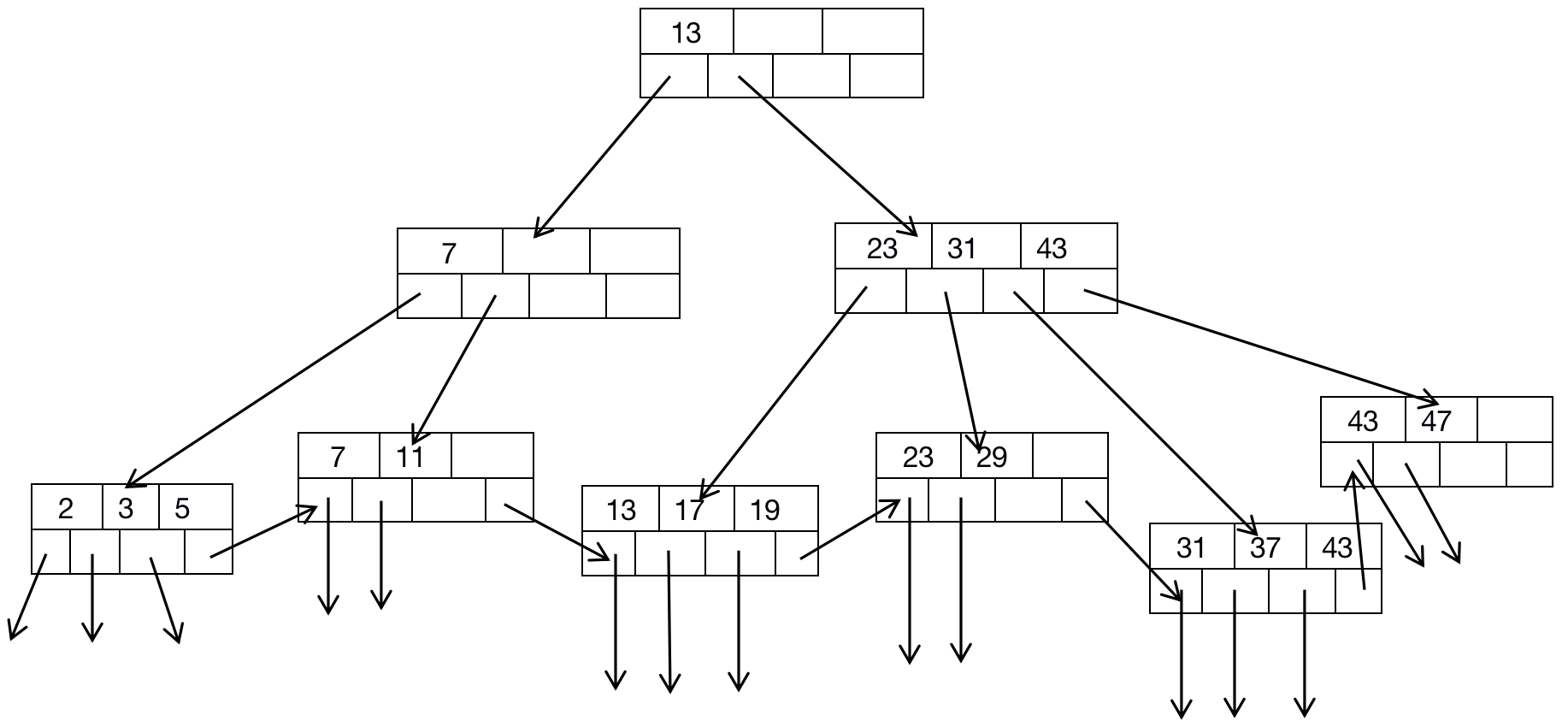
Assume that N is the Leaf Node with n Key Values, and we assume to insert the (n + 1)th key and its corresponding pointers. We create a new node M, which is the brother of Node N, which is in the right side of node N. So according to the Sorted Sequence, the [ (n + 1) / 2] Key - Pointers would stay in the Node N; While all other Key - Pointer would be moved to the Node M, attention that, Node M and N have enough Key - Pointer Pair, which means that at least [ ( n + 1) / 2 ] would stay in M and N.

*Assume N would be an Internal Node with n Keys and ( n + 1 ) Pointers, and the Split Node N from lower Level Node is assigned to (n + 2)th pointer.* We can execute following Steps:

1. *Create a new Node N, it would be the brother of Node M and close to the right of Node N.*
2. *Leave the front [ (n + 2) / 2] pointers into Node N according to Sorted Sequence, but leave the left [ (n + 2) / 2] pointers into Node M.*
3. *The front [ n / 2 ] Nodes has been kept into the Node N, but the left [ n / 2 ] Nodes have been moved to the Node M. Attention that, the intermediate Node would be stayed, it does not belong to the Node N or M. This kept Key K means the smallest Key of the first Sub - Node through Node M. Although Key Value K does not exist in the Node N and Node M, but it present the smallest Key Value that M can access, from such point, we can say that this Key Value connects with Node M. Therefor, Key K can be used to divide N and M by its Father Node.*

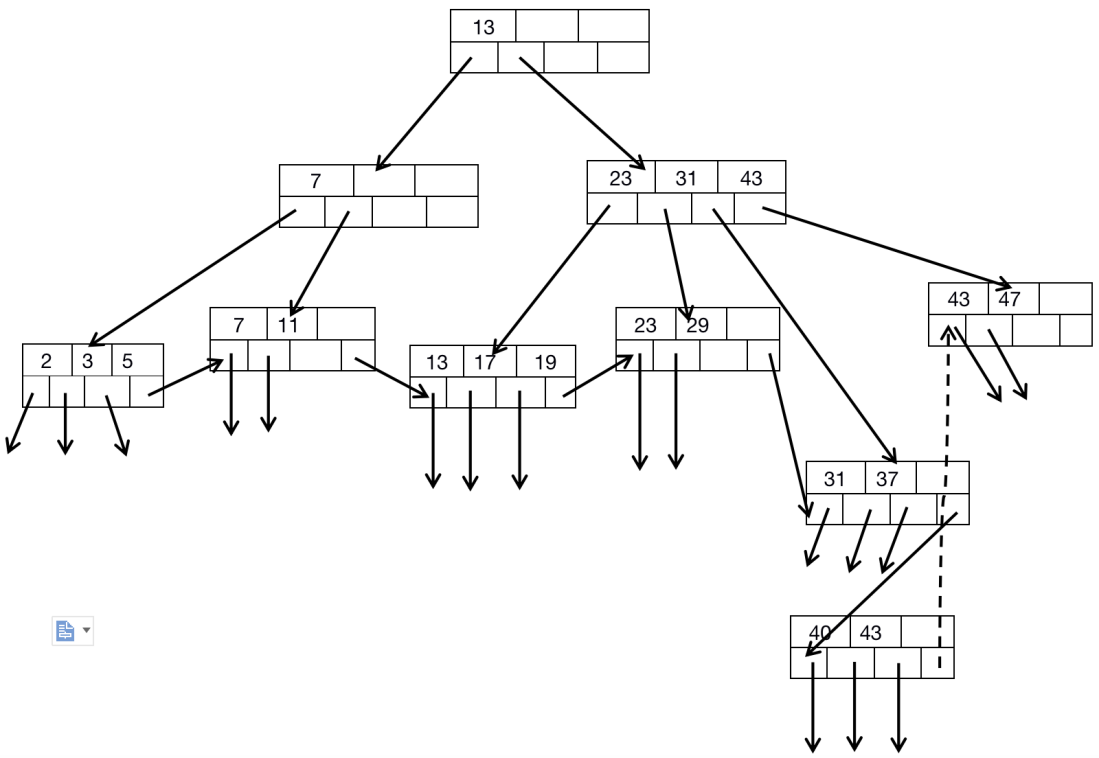
*Example:*

Assume we insert Key Value 40 into B - Tree, and the process to insert can be found according to the following steps:

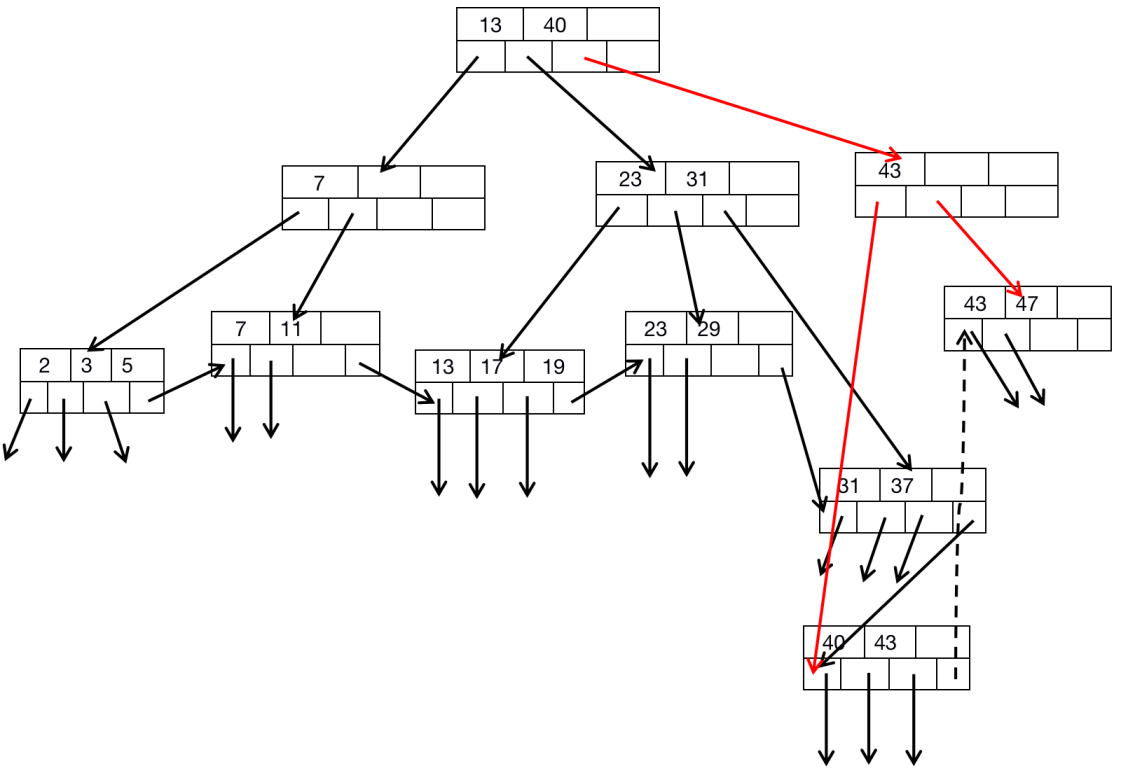


1. We find the fifth Node with Key Values (31, 37, 43) in B - Tree. After that, there would have four Nodes (31, 37, 40, 43), so we need to split these four leaf nodes. At first, we need to create one New node, and move two other nodes 40 and 43 into New Node.

*B - Tree after insertion:*



1. Although we display these Nodes as four lines, but for B - Tree, it still has only three lines, however, seven leaf pointers occupy the later two lines. These Leaf Nodes can be linked by the last node, and formed the final list from left to right.
2. Insert one pointer which is used to point to the new leaf node in its father Node, also we need to relate the pointer to Key 40. Since Key 40 is the smallest Key Value which can be accessed by new Leaf Node. However, we can tell from B - Tree and find that the father Node is full and it has no other space to store any other Key or Pointer. So it also needs to be split.
3. We need to find pointers of five Leaf Pointers and four smallest Key Values among four Leaf Nodes, which is to say, *Pointers P1, P2, P3, P4 and P5 all point to those Leaf Nodes, and the smallest Key Value are 13, 23, 31, 40 and 43, we need to use the Key Sequence 23, 31, 40, and 43 to isolate these pointers*. The first three pointers and two keys are kept in the new split Internal Node; and the last two pointers and one key are kept in a new Node. Left Key Value is 40 which represents the smallest Key accessed by the new Node.



Key Value 40 is used as the mark of smallest Key which can be visited by Third Node, and it is located in Root Node, and is used to distinguish from the Second Node and Third Node.

### Chapter 3.2.6 Deletion from B - Tree

*Introduction:*

If we want to delete the record with given value K, then we need to locate the record and its Key - Data pair in Leaf Node. The main process of Deletion is to find, just as introduce before, then we need to delete the record itself and delete from Key - Data Pair from B - Tree.

*Rule:*

* After Deletion, if the number of Key - Data still has the least Key - Pointer number, then we do not need to do anything.
* Otherwise, if the number of Key - Data is just smaller than the least Key - Pointer number, so after deletion, it violates the constraint number of Key - Pointer Pair. At that time, we need to do two things, the first thing is to delete Key - Pointer Pair from down to top recursively.

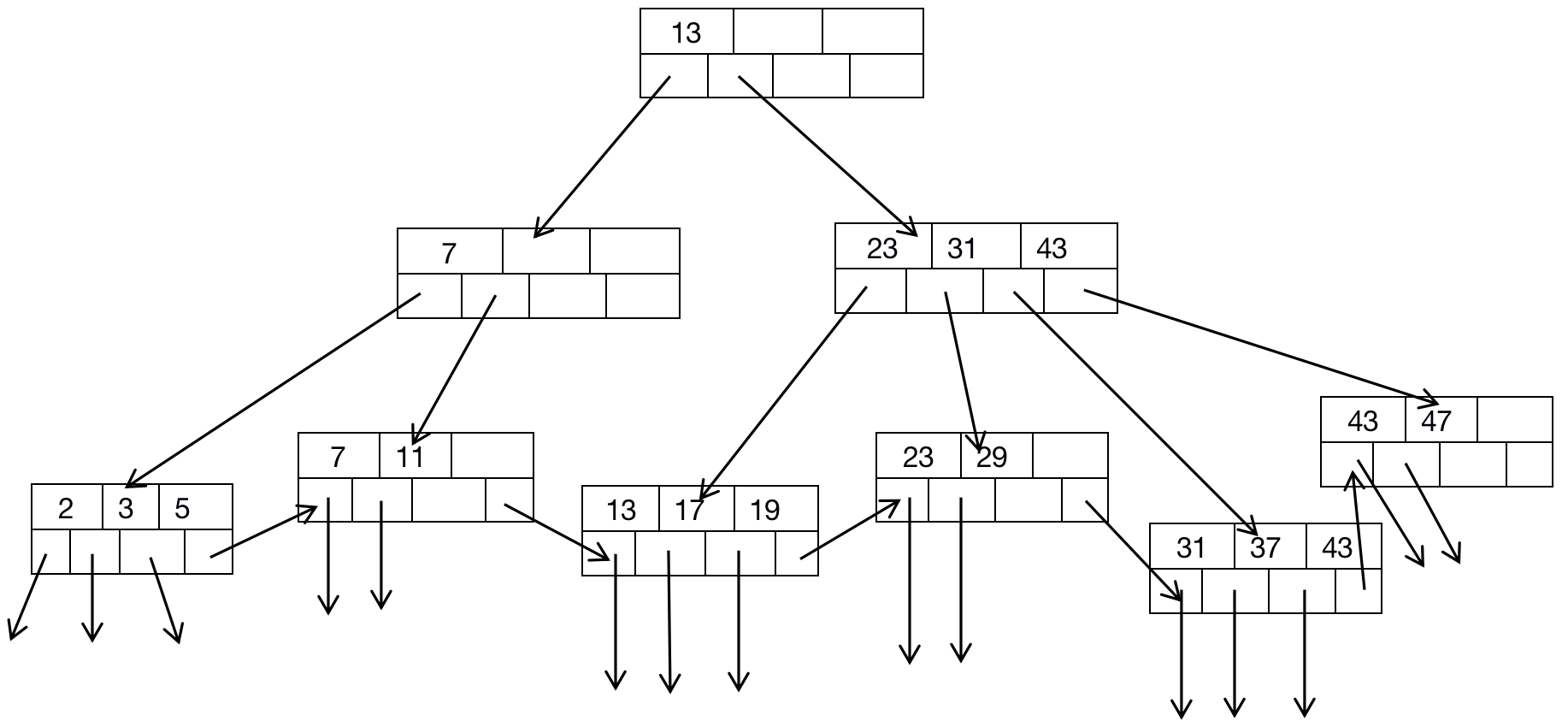
1. *If the number of Key and Pointer in Neighbor Brother Node of Node N have exceeded the least number, then one of Key - Pointer Pair can be moved into Node N to keep the sequence of Key - Pointer Pair. The Parent of Node N also needs to adjust such situation to reflect this kind of situation.*

*( For example, Right Neighbor Node M of Node N can provide one Key and Pointer, then the Key - Pointer pair which needs to be moved to Node N must have the smallest Key Value in Pointer M. Under such situation, we must need to increase Key Value in the Parent Father of Node N to reflect the new modified Node M. )*

1. *The difficult situation is that when the neighbor two brothers can not provide such Key - Pointer Pair to Pointer N. But, under such situation, Node N and its Right Neighbor Node M, the latter Key - Pointer Pair has the least number and the former Key - Pointer Pair is less than the least number. Therefore, their combination can not exceed the constraint Key - Pointer number of single Node. So when we combine these two nodes, actually we need to delete one of them and we need to adjust the Key Value of Parent Node, and delete one Key - Pointer of the Parent Node. If the number of Parent Node is satisfied the constraint, then we finish the deletion Operation, otherwise, we need to use Deletion Algorithm recursively.*

*Example:*

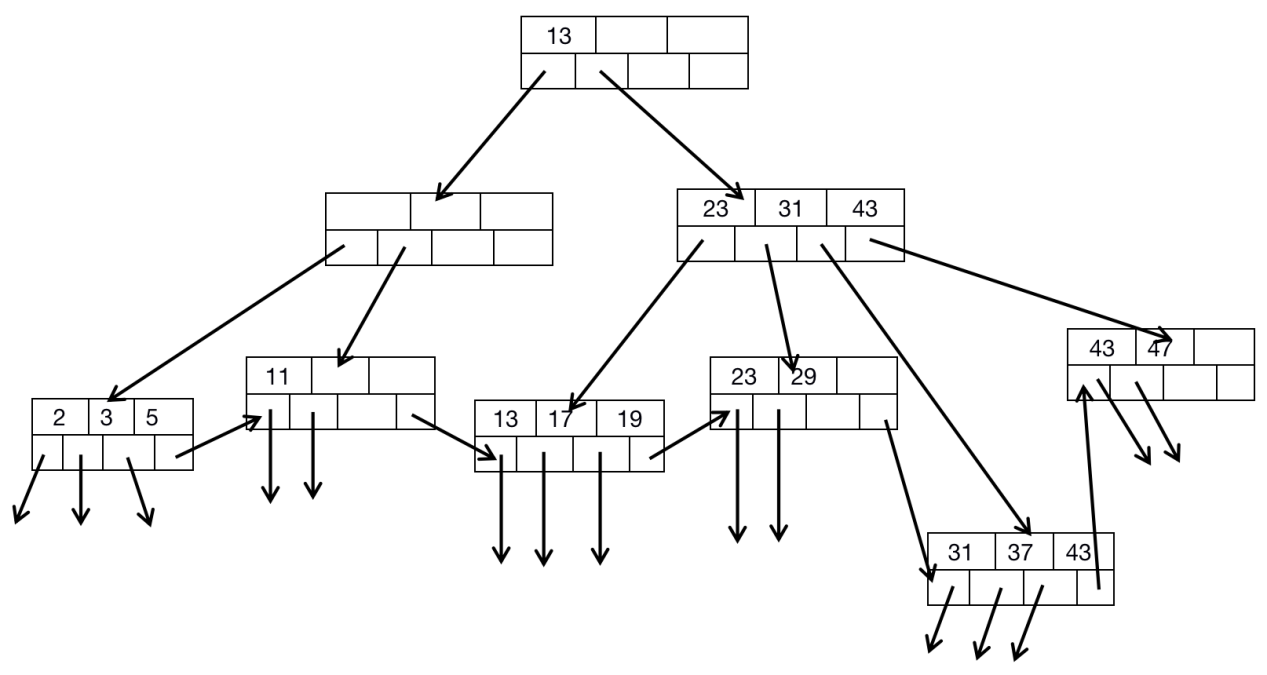
Start from the initial B - Tree, which means that starts before insert Node N into B - Tree.



* *Delete Data Node which has no influence on Structure of Father Node*

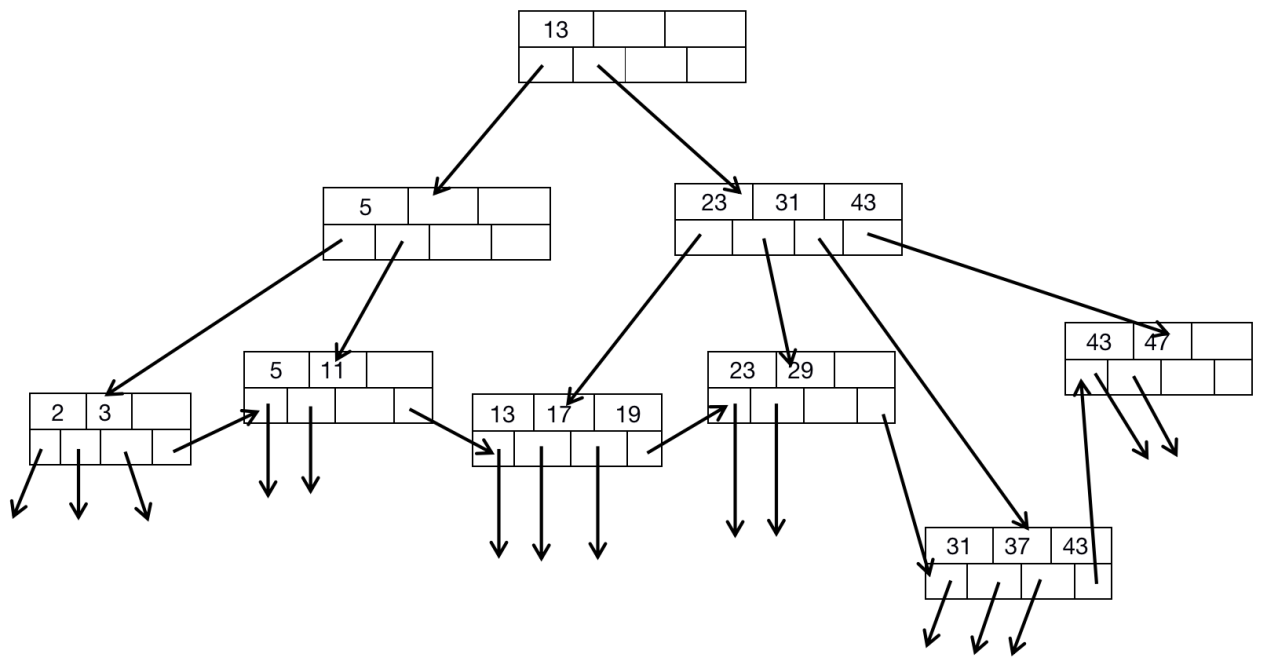
Assume that we delete Key Value 7 from B - Tree and we can tell from B - Tree above and Node Value 7 appears in the Second B - Tree Node. Then we need to delete this Node and corresponding Pointer and Data Value.

At that time, after delete Key Value 7, then the Second Leaf Node would has only one Data Node.



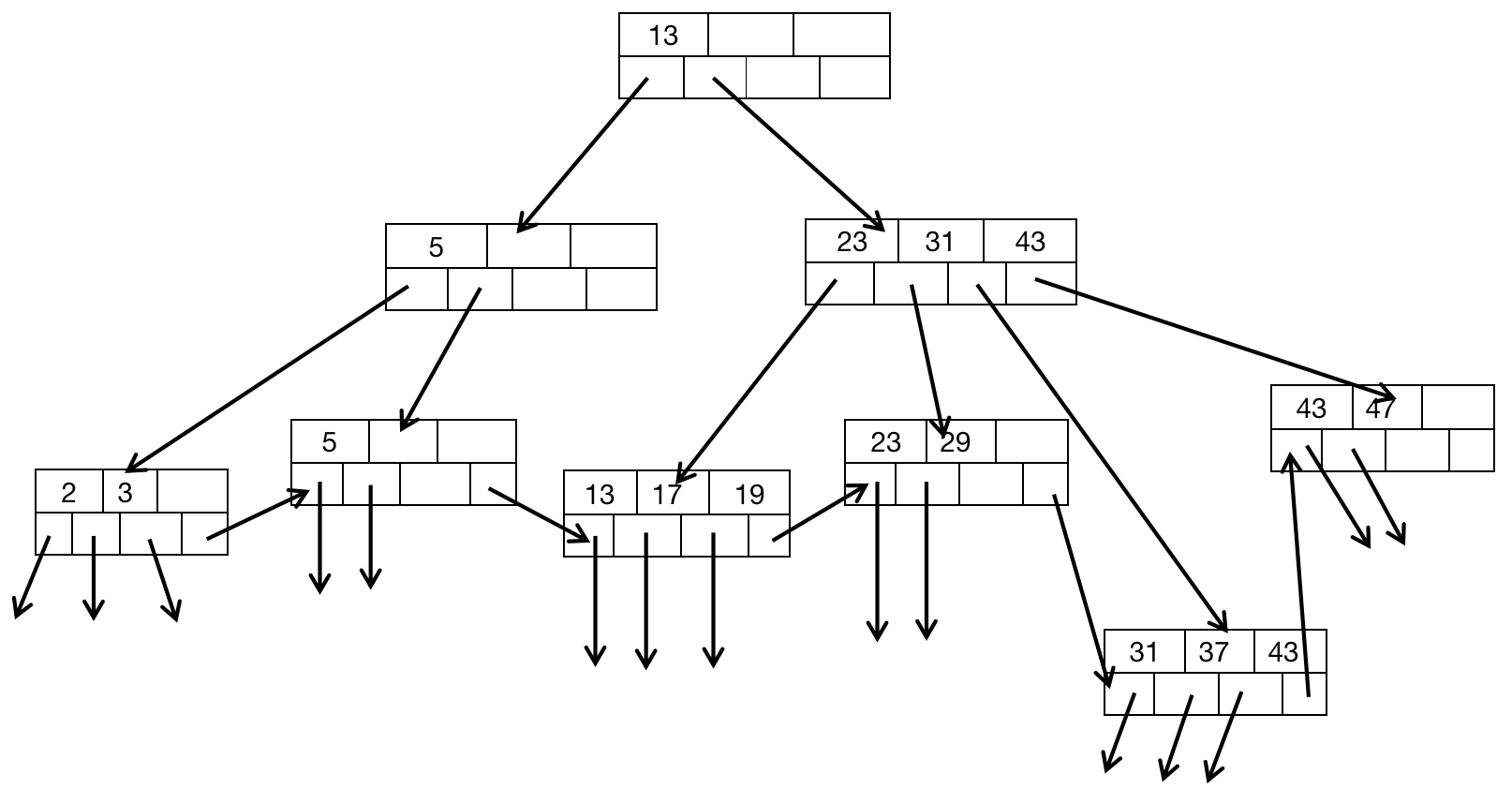
Now, in B - Tree Node, there will have not got any Data Value any longer. However, in the Second Node, there only has one Data Value which equals to 11, which has not satisfied condition which is required above. Luckily, in the First Node, there already have three Data Node which ranges from 2, 3, and 5 in the First Node.

We need to combine these two Data Nodes and divide them into two separate Data Node. Therefore, the first Left Node have two Nodes 2 and 3, while the second Node have another two Nodes 5 and 11. At this time, the least number of Second Node equals to 5, so the Data Value of its father Node equals to 5 this time. B - Tree is as below:

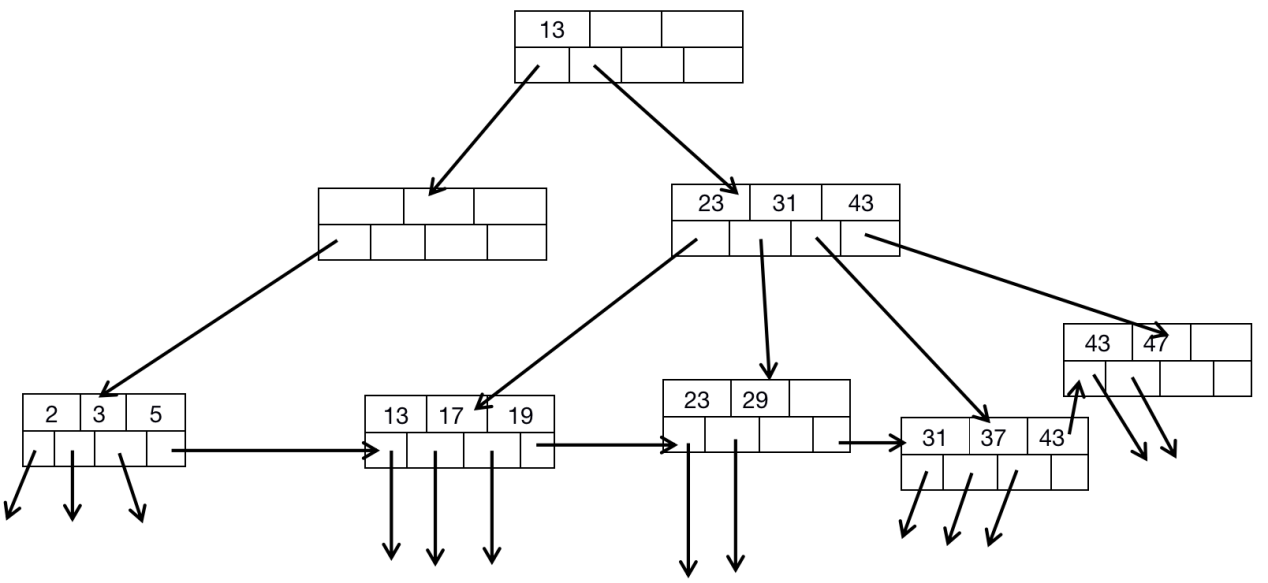


* *Delete Data Node which has influence on Structure of Father Node*

Assume that we need to delete Node with Data Value 11, below is the B - Tree after delete Node with Data Value 11.

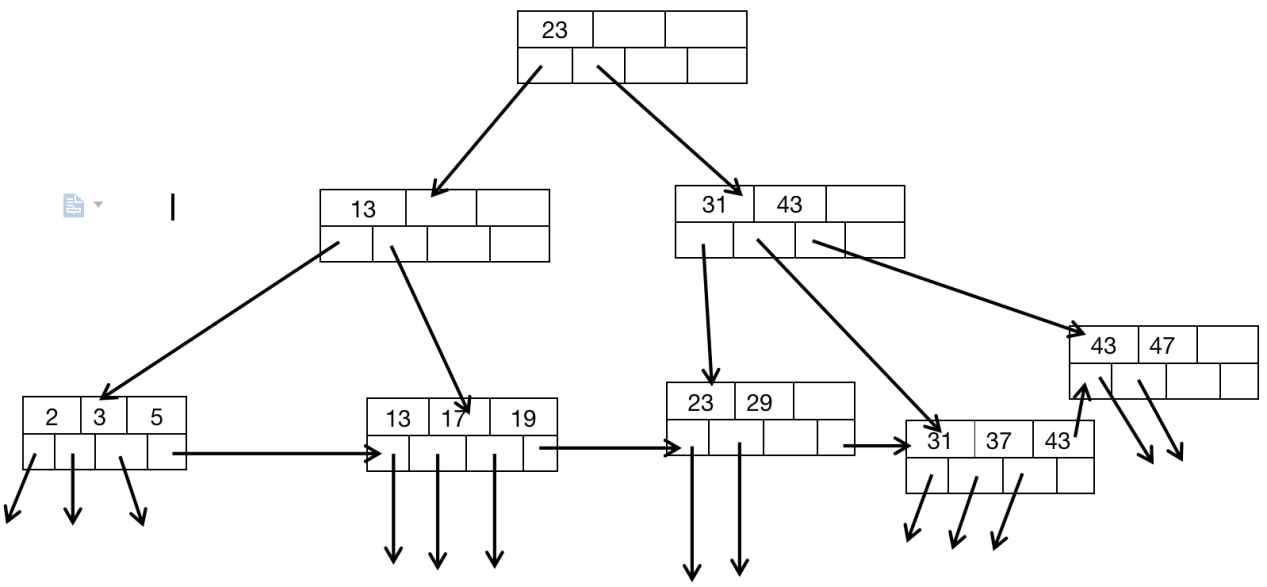


After deletion, we find that the Leaf Node has only one Data Value which equals to 5. Same as all steps above, we need to combine Node with Data Value 2 and 3 and Node with Data Value 5. After that, there would has only one B - Tree Node.



Therefore, we need to delete the Empty Node in Second Level, and we can see Node Change of its Father Node.

We need to move Node Data 13 downward. So the Leaf Node which includes 13, 17, and 19 need to move leftward and become the Leaf Node of the new created Internal Node.



*Attention that Data Value 23 means that the least Data Value can be accessed by Second Child of Root Node.*

### Chapter 3.2.7 Productivity of B - Tree

*Introduction:*

B - Tree can helps us to Realize Record Insertion, Deletion and Update, but each File Operation has only a few Disk I/O.

First of all, if the number of N is big, then there would be less chances to split and merge. Besides, when this situation happens, the changes would only be limited in Leaf Node, because only have the Leaf Node and its Father Node would be influenced. So, normally we can ignore Disk I/O.

*Normally, we can claim that it’s enough to assign Level of B - Tree 3, unless the Database is too big. So, we assign Level of B - Tree 3.*

*Example:*

Assume that Storage Size equals to 4096 bytes, and integer value occupies 4 bytes and pointers occupies 8 bytes. Then we can make sure the biggest value of n. Here, 4096 = 4 \* n + 8 \* ( n + 1 ) = 12 \* n + 1, then n = 340.

So, we can assume that the Root Node will have 255 Internal Nodes and 255 ^ 2 = 65025 Leaf Nodes. Among these Leaf Node, we can have 65023 ^ 3 = 1.66 \* 10 ^ 3 File which can be visited by three - Level B - Tree.

*Supplement:*

However, for each query, we can use B - Tree to realize it which is less than 3 times Disk I/O. *It would be the Best Choice to buffer B - Tree Root Node Block into Main Memory.* If under this solution, then each time query B - Tree with Level three would only need two times Disk Read Operation. *Actually, it’s reasonable to buffer the Second Level B - Tree into Main Memory.* So, Query on B - Tree would cost one time Disk I/O addition with the Disk I/O to proceed Data File.