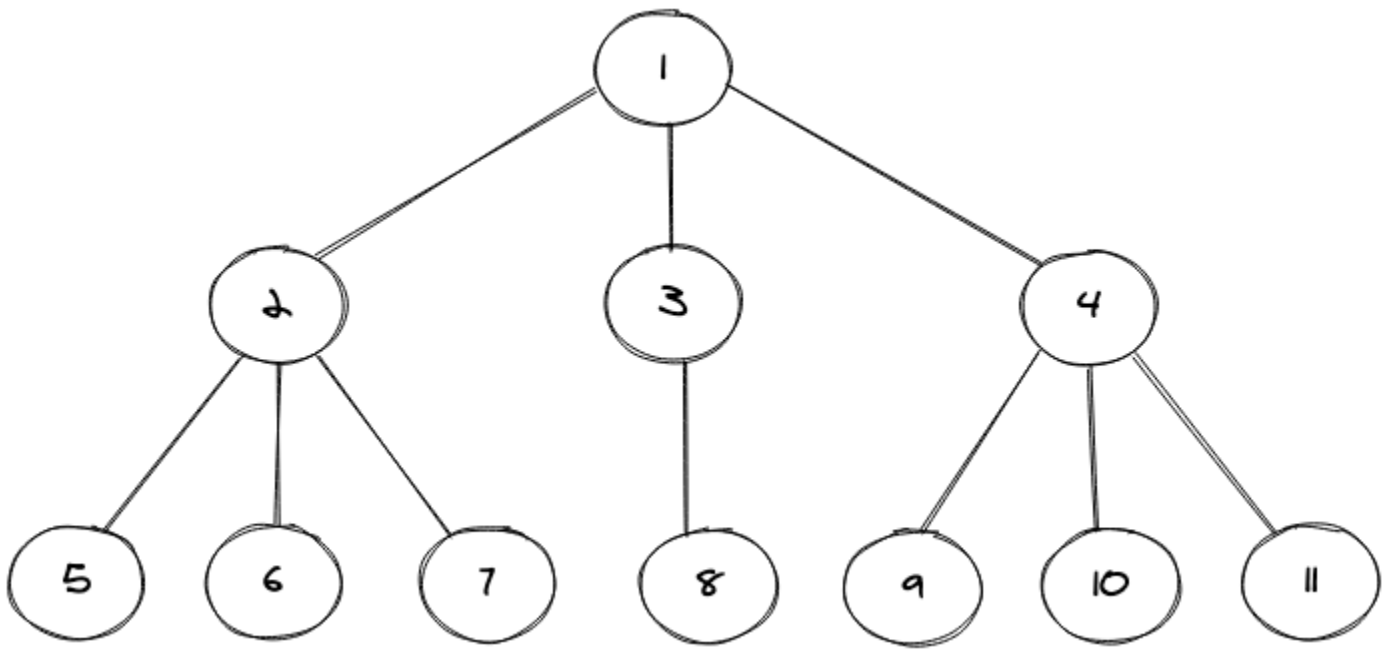
N-ary Tree Data Structure

The N-ary tree is a tree that allows us to have `n` number of children of a particular node, hence the name **N-ary**, making it **slightly complex than the very common binary trees** that allow us to have at most 2 children of a particular node.

A pictorial representation of an N-ary tree is shown below:



In the N-ary tree shown above, we can notice that there are **11 nodes** in total and **some nodes have three children** and some have had one only. In the case of a binary tree, it was easier to store these children nodes as we can assign two nodes (i.e. left and right) to a particular node to mark its children, but here it's not that simple. To store the children nodes of any tree node we make use of another data structure, mainly vector in C++ and LinkedList in Java.

Implementation of N-ary Tree

The first thing that we do when we are dealing with non-linear data structures is to create our own structure (constructors in Java) for them. Like in the case of a binary tree, we make use of a class `TreeNode` and inside that class, we create our constructors and have our class-level variables.

Consider the code snippet below:

```
public static class TreeNode{

        int val;

        List<TreeNode> children = new LinkedList<>();


        TreeNode(int data){

            val = data;

        }


        TreeNode(int data,List<TreeNode> child){

            val = data;

            children = child;
```

```
        }

}
```

Copy

In the above code snippet, we have a class named `TreeNode` which in turn contains two constructors with the same name, but they are overloaded (same method names but with different parameters) in nature. We also have two identifiers, one of them is the val that stores the value of any particular node, and then we have a List to store the children nodes of any node of the tree.

The above snippet contains the basic structure of our tree, and now are only left with making one and then later we will see how to print the tree using level order traversal. To build a tree we will make use of the constructors that we have defined in the above classes.

Consider the code snippet below:

```
public static void main(String[] args) {

        // creating an exact replica of the above pictorial N-ary Tree

        TreeNode root = new TreeNode(1);

        root.children.add(new TreeNode(2));

        root.children.add(new TreeNode(3));

        root.children.add(new TreeNode(4));

        root.children.get(0).children.add(new TreeNode(5));

        root.children.get(0).children.add(new TreeNode(6));

        root.children.get(0).children.add(new TreeNode(7));

        root.children.get(1).children.add(new TreeNode(8));

        root.children.get(2).children.add(new TreeNode(9));

        root.children.get(2).children.add(new TreeNode(10));

        root.children.get(2).children.add(new TreeNode(11));

        printNAryTree(root);

}
```

Copy

First thing first, we created the **root node of our N-ary Tree**, then we have to assign some children to this root node, we do this by making use of the dot(.) operator and accessing the children property of the root node and then adding different children to the root nodes by using the add method provided by the `List` interface. Once we have added all the children of the root node, it is time to add children of each of the new level nodes, we do that by first accessing that node by using the get method provided by the list interface and then adding the respective children nodes to that node.

Lastly, we print this N-ary Tree, we did it by calling the `printNAryTree` method.

Now, since printing a tree is not as simple as looping through a collection of items, we have different techniques (algorithms precisely) at our disposal. These are mainly:

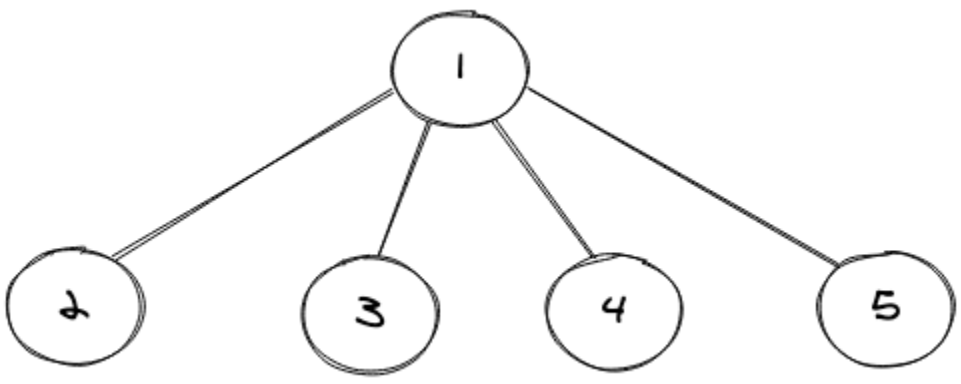
- **Inorder Traversal**
- **Preorder Traversal**
- **PostOrder Traversal**
- **Level Order Traversal**

For the sake of this tutorial, we will use the Level Order Traversal approach, as it is easier to understand, provided if you have seen how it works on a binary tree before.

Level Order Traversal (Printing the N-ary Tree)

The Level Order traversal of any tree takes into the fact that we want to print the nodes at a root level first, then move on to the next level and keep repeating this process until we are at the last level. We make use of the Queue data structure to store the nodes at a particular level.

Consider a simple N-ary Tree shown below:



Level Order Traversal for the above tree will look like this:

```
1

2 3 4 5
```

Copy

Consider the code snippet below:

```
private static void printNAryTree(TreeNode root){

        if(root == null) return;

        Queue<TreeNode> queue = new LinkedList<>();

        queue.offer(root);

        while(!queue.isEmpty()) {

            int len = queue.size();

            for(int i=0;i<len;i++) { // so that we can reach each level

                TreeNode node = queue.poll();

                System.out.print(node.val + " ");

                for (TreeNode item : node.children) { // for-Each loop
to iterate over all childrens

                    queue.offer(item);
```

```
                }

            }

            System.out.println();

        }

}
```

Copy

The entire code looks something like this:

```
import java.util.LinkedList;

import java.util.List;

import java.util.Queue;


public class NAryTree {

    public static class TreeNode{

        int val;

        List<TreeNode> children = new LinkedList<>();


        TreeNode(int data){

            val = data;

        }


        TreeNode(int data,List<TreeNode> child){

            val = data;

            children = child;

        }

    }


    private static void printNAryTree(TreeNode root){

        if(root == null) return;

        Queue<TreeNode> queue = new LinkedList<>();

        queue.offer(root);

        while(!queue.isEmpty()) {
```

```
            int len = queue.size();
            for(int i=0;i<len;i++) {
                TreeNode node = queue.poll();
                assert node != null;
                System.out.print(node.val + " ");
                for (TreeNode item : node.children) {
                    queue.offer(item);
                }
            }
            System.out.println();
        }
    }

    public static void main(String[] args) {
        TreeNode root = new TreeNode(1);
        root.children.add(new TreeNode(2));
        root.children.add(new TreeNode(3));
        root.children.add(new TreeNode(4));
        root.children.get(0).children.add(new TreeNode(5));
        root.children.get(0).children.add(new TreeNode(6));
        root.children.get(0).children.add(new TreeNode(7));
        root.children.get(1).children.add(new TreeNode(8));
        root.children.get(2).children.add(new TreeNode(9));
        root.children.get(2).children.add(new TreeNode(10));
        root.children.get(2).children.add(new TreeNode(11));
        printNAryTree(root);
    }
}
```

Copy

The output of the above code is:

```
1

2 3 4

5 6 7 8 9 10 11
```

We can compare this output to the pictorial representation of the N-ary tree we had in the starting, each level node contains the same values.

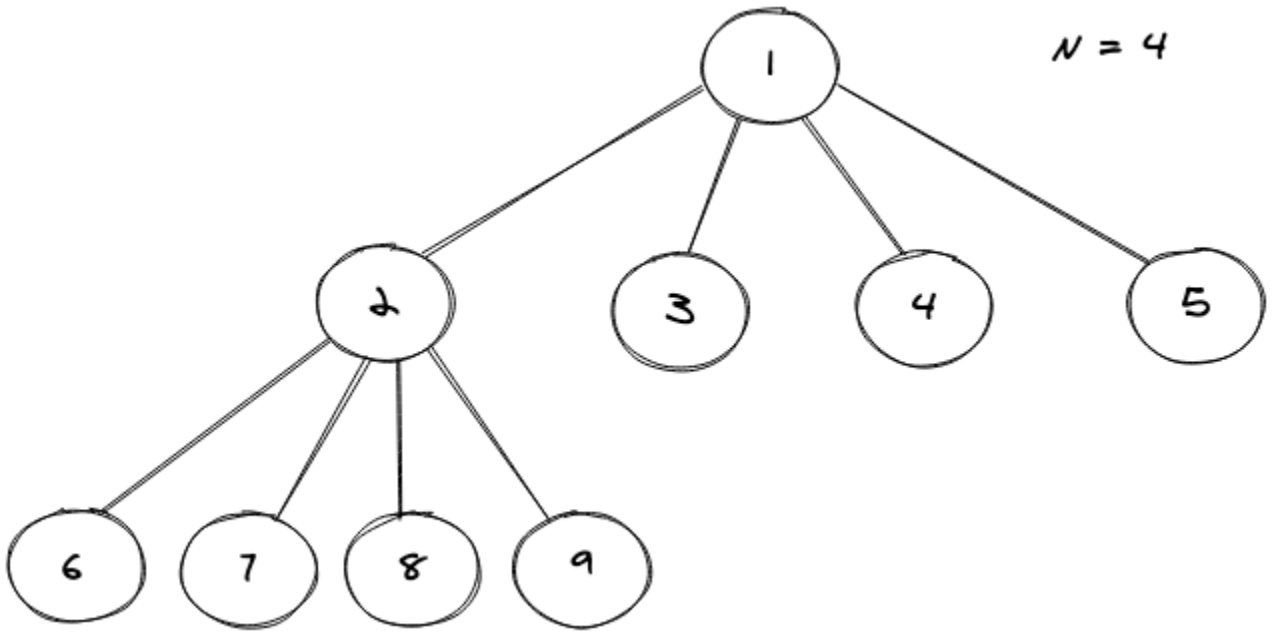
Types of N-ary Tree

Following are the types of N-ary tree:

1. Full N-ary Tree

A Full N-ary Tree is an N-ary tree that allows each node to have either 0 or N children.

Consider the pictorial representation of a full N-ary tree shown below:

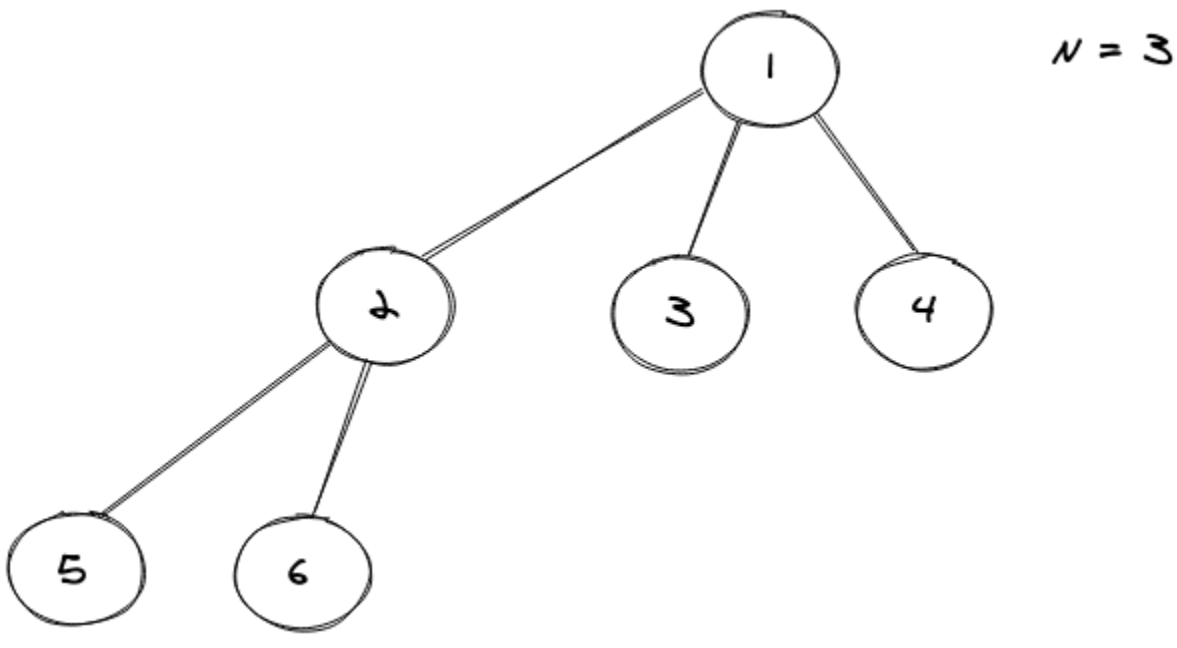


Notice that all the nodes of the above N-ary have either 4 children or 0, hence satisfying the property.

2. Complete N-ary Tree

A complete N-ary tree is an N-ary tree in which the nodes at each level of the tree should be complete (should have exactly **N children**) except the last level nodes and if the last level nodes aren't complete, the nodes must be "as left as possible".

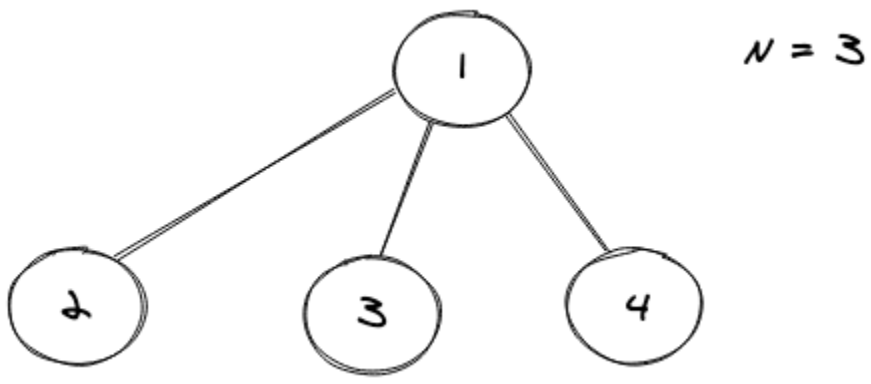
Consider the pictorial representation of a complete N-ary tree shown below:



3. Perfect N-ary Tree

A perfect N-ary tree is a full N-ary tree but the level of the leaf nodes must be the same.

Consider the pictorial representation of a perfect N-ary tree shown below:



Conclusions

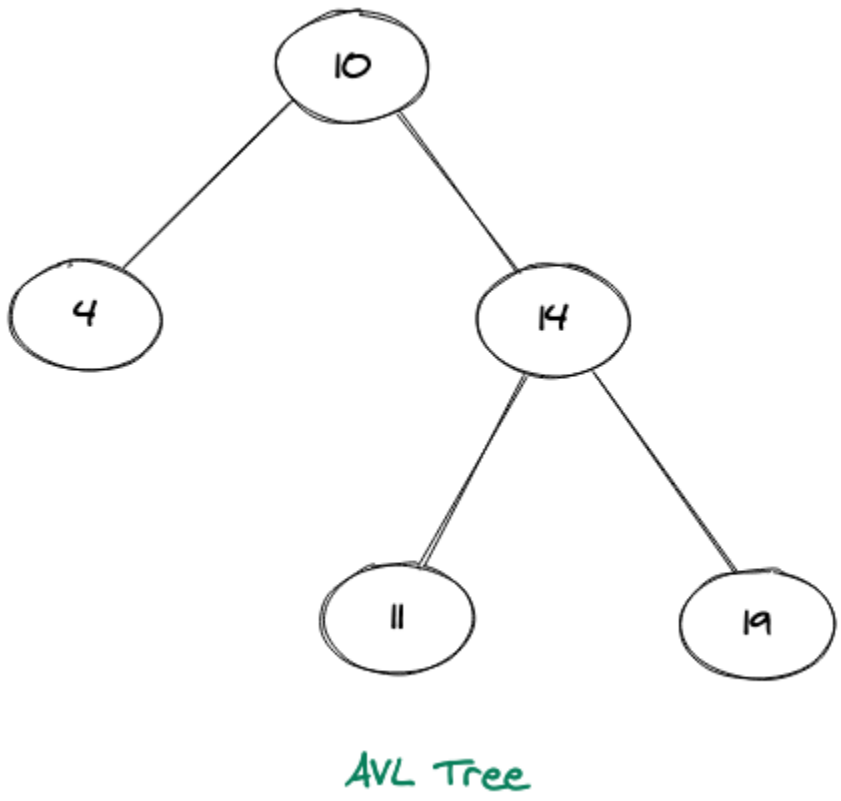
- We learned what an N-ary tree is.
- We also learned how to implement an N-ary tree in Java(via level order traversal).
- Then we learned what different types of N-ary trees are there in total.

AVL Tree Data Structure

An AVL tree is another special tree that has several properties that makes it special. These are:

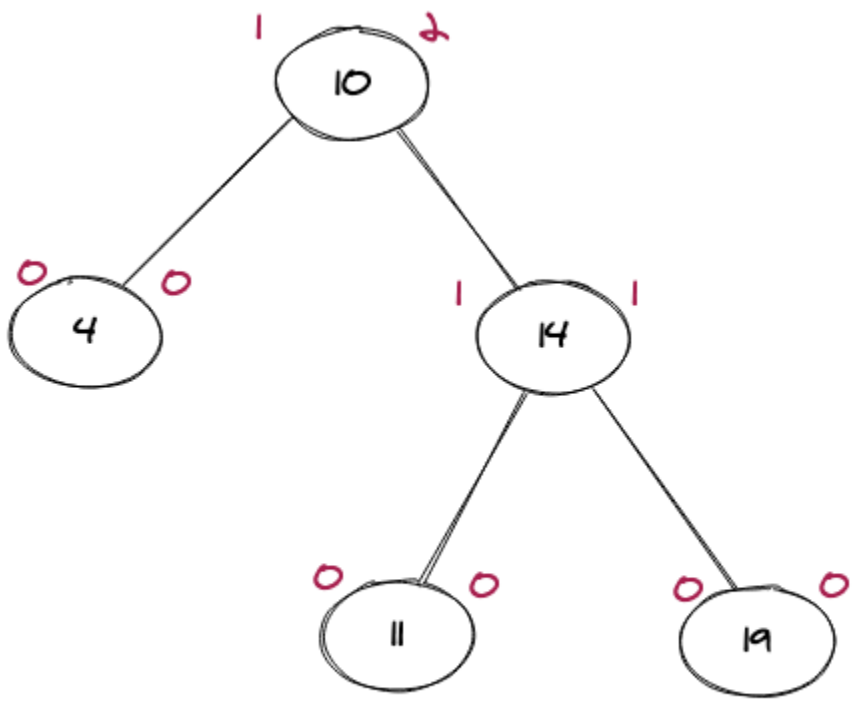
- It is a BST that is balanced in nature.
- It is self-balanced.
- It is not perfectly balanced.
- Every sub-tree is also an AVL Tree.

A pictorial representation of an AVL Tree is shown below:



AVL Tree

When explained, we can say that it is a Binary Search Tree that is height-balanced in nature. **Height balance** means that the difference between the left subtree height and the right subtree height for each node cannot be greater than 1(i.e **height(left) - height(right) <= 1**).



AVL Tree with Balance Factor

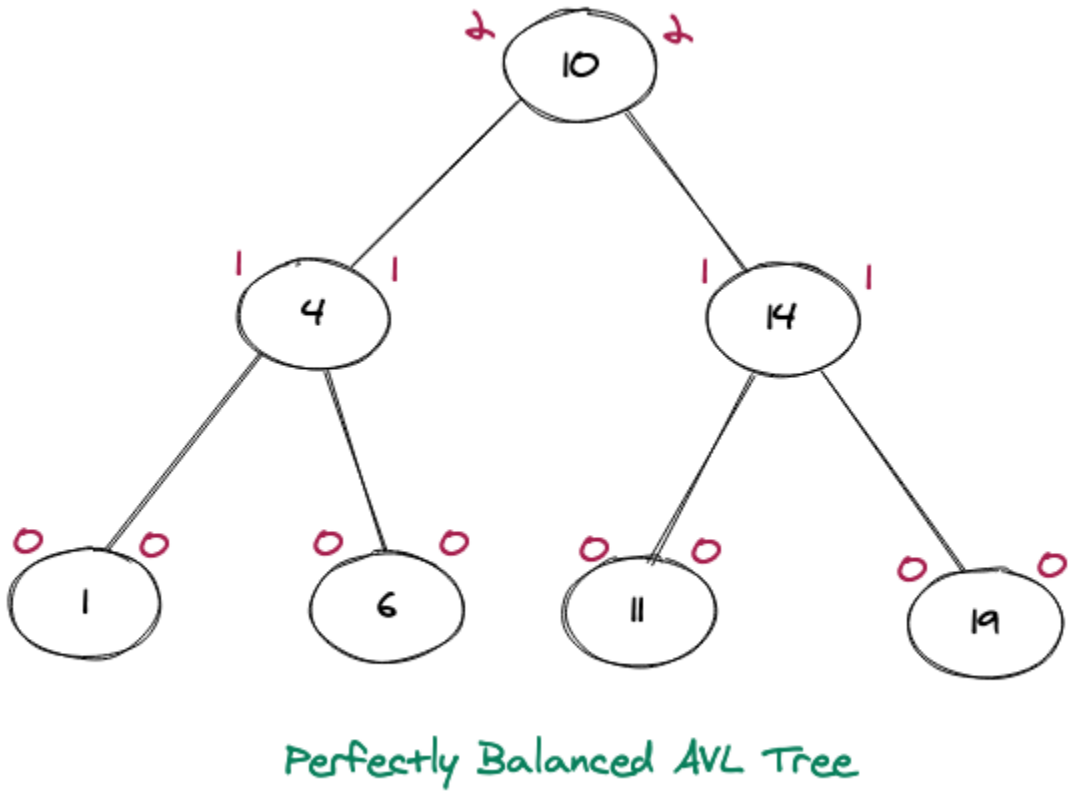
Notice that each node in the above BST contains two values, the left value denotes the height of the left subtree of that node and the right value denotes the height of the right subtree of that node, and it can be seen that at not a single node we have a value difference that is greater than 1, hence making it a balanced AVL Tree.

A **self-balanced tree** means that when we try to insert any element in this BST and if it violates the balancing factor of any node, then it dynamically rotates itself accordingly to make itself self-

balanced. It is also well known that AVL trees were the first well known dynamically balanced trees that were proposed.

It is not perfectly balanced, where the perfect balance implies the fact that the **height between the right subtree and the left subtree** of each node is equal to **0**.

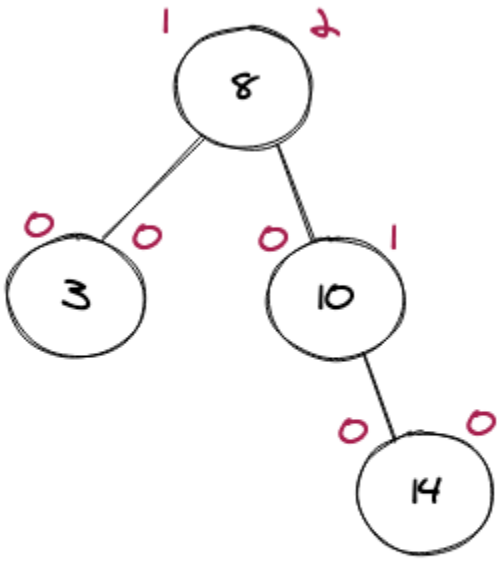
A pictorial representation of a Perfectly Balanced AVL Tree is shown below:



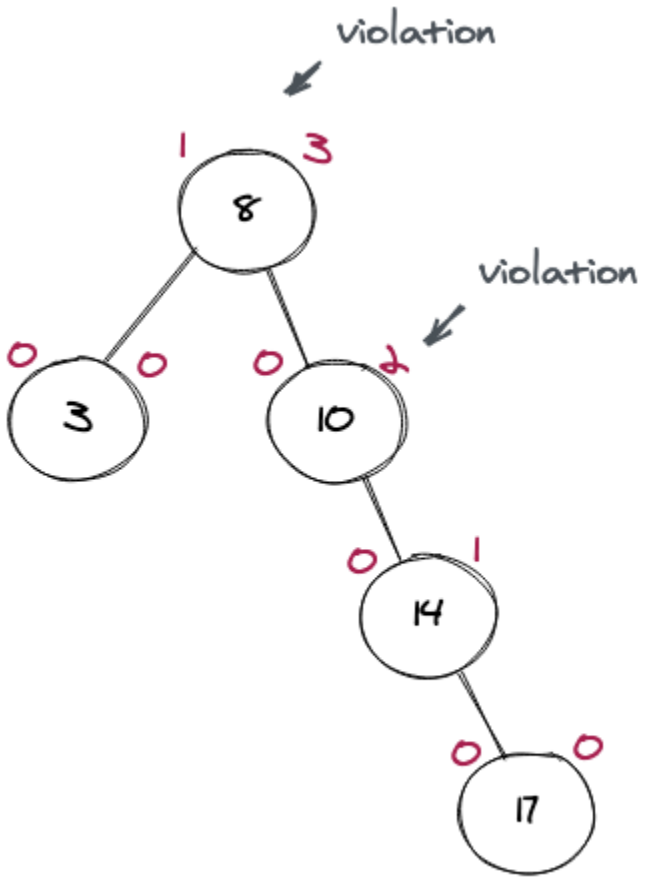
Perfectly Balanced AVL Tree

All the nodes of the above AVL Tree have the balance factor equal to 0(**height(left) - height(right) = 0**) making it a perfect AVL tree. It should also be noted that a perfect BST is the same as a perfect AVL tree. If we take a closer look at all the pictorial representations above, we can clearly see that each subtree of the AVL tree is also an AVL tree.

Now let's take a look at two more pictorial representations, where one of them is an AVL tree and one isn't.



AVL Tree



NOT an AVL Tree

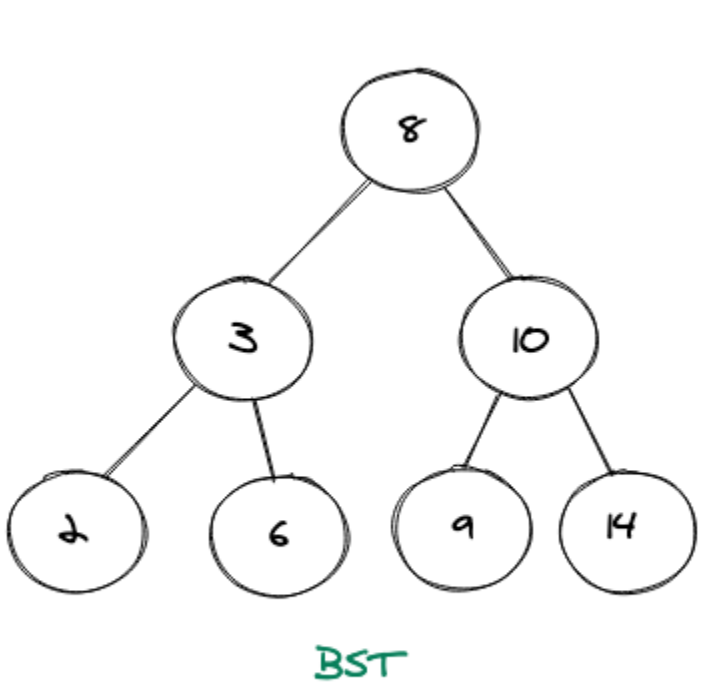
Though both the above trees are BST, only the left BST in the above representation is an AVL one as it height-balanced. In right BST, we have two violation nodes (i.e 8 and 10) where the balance

factor is more than 2 in both the cases. It should be noted that this tree can be rotated in a certain manner to make it an AVL tree.

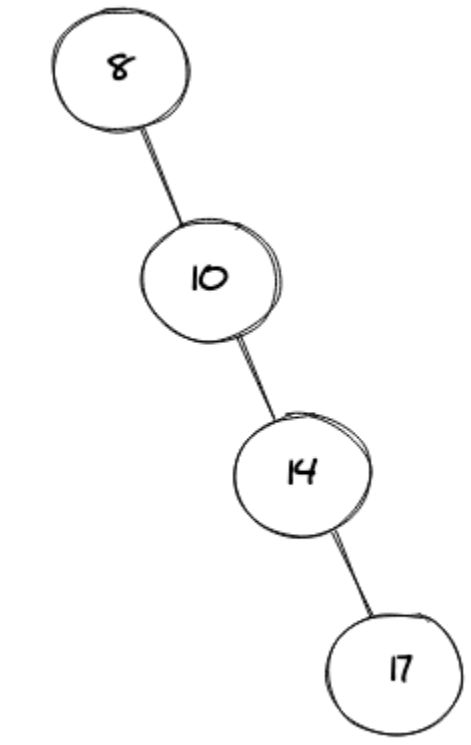
Why AVL Tree?

When we already had a similar data structure(i.e. BST's), why would we need a complex version of it?. The answer lies in the fact that BST has some limitations in certain scenarios that make some operations like searching, inserting costly (as costly as $O(n)$).

Consider the pictorial representation of two BST's below:



BST



Right Skewed BST

The height of the left BST is $O(\log n)$ and the height of the right BST is $O(n)$, thus the search operation time complexity for the left BST is $O(\log n)$ and for the right-skewed is $O(n)$ which is its worst case too. Hence, if we have such a skewed tree then there's no benefit of using a BST, as it is just like a linked list when it comes to time and space complexity.

That's why we needed a balanced BST's so that all the basic operations guarantee a time complexity of O(logN).

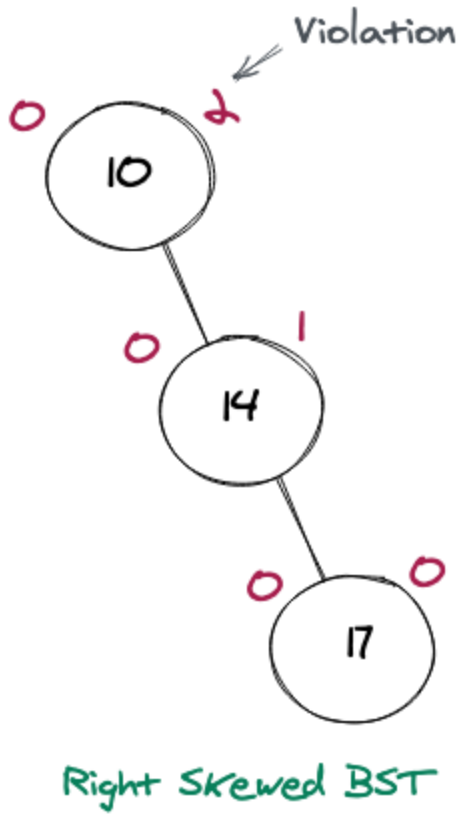
AVL Tree Rotations

If the AVL tree encounters any **violation of the balance factor** then it tries to do some rotations to make the tree balanced again. There are four rotations in total, we will look at each one of them. There mainly are:

- Left Rotation
- Right Rotation
- Left - Right Rotation
- Right - Left Rotation

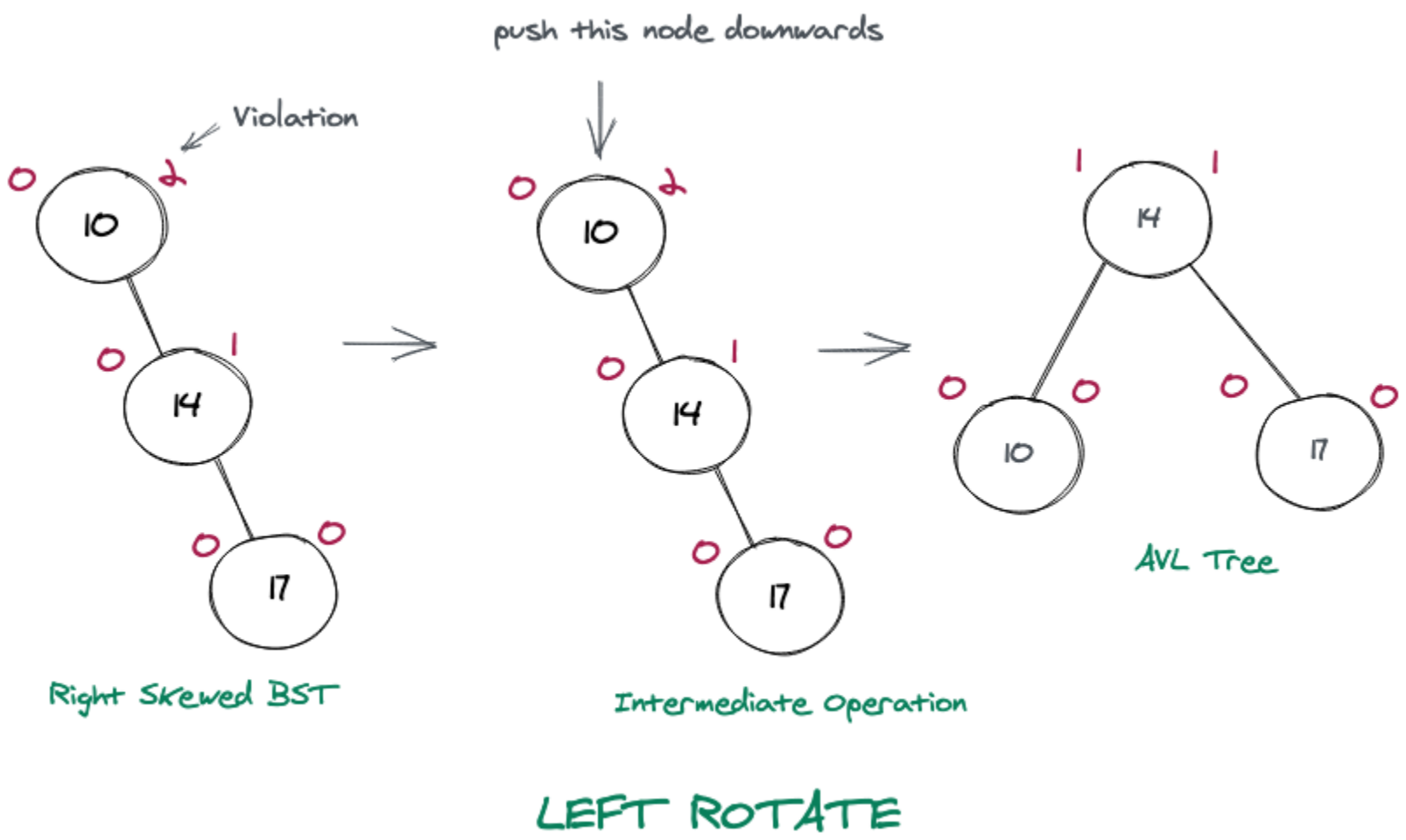
1. Left Rotation

This rotation is performed when the violation occurs in the left subtree of the left child of the violating node. Consider the pictorial representation of a BST below that is not balanced, in order to make it so, we will do some rotations.



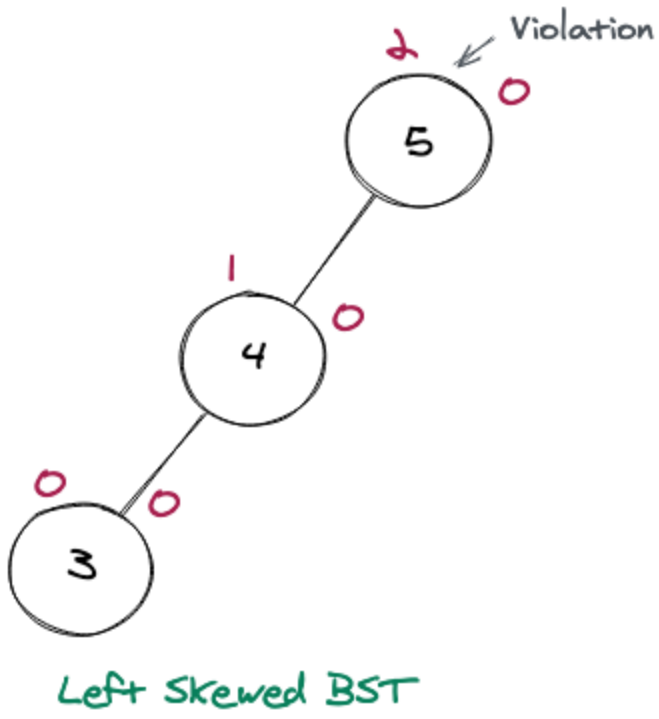
Clearly this is not an AVL tree, but consider the node where the violation occurs, we take this node as a pivot to do our rotations. The tree is heavy on the right and the node that causes the issue is 17 as if we remove that one, we would have a balanced BST (i.e. AVL). Since it is heavily skewed on the right, we will do a left-rotation to make it balanced.

Consider the pictorial representation of making the above right skewed tree into an AVL tree.



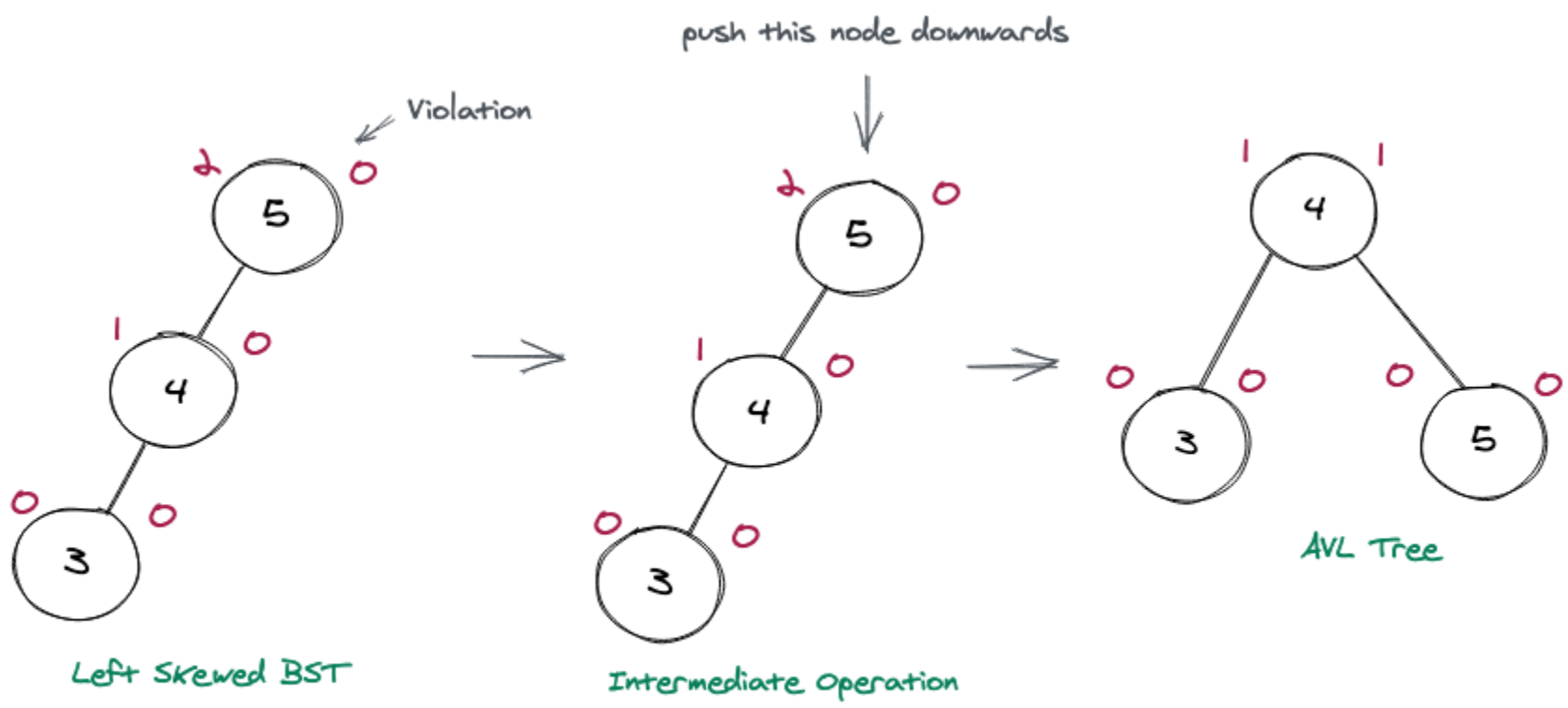
2. Right Rotation

This rotation is performed when the violation occurs in the right subtree of the right child of the violating node. Consider the pictorial representation of a BST below that is not balanced, in order to make it so, we will do some rotations.



Clearly this is not an AVL tree, but consider the node where the violation occurs, we take this node as a pivot to do our rotations. The tree is heavy on the left and the node that causes the issue is 3 as if we remove that one, we would have a balanced BST (i.e. AVL). Since it is heavily skewed on the left, we will do a right-rotation to make it balanced.

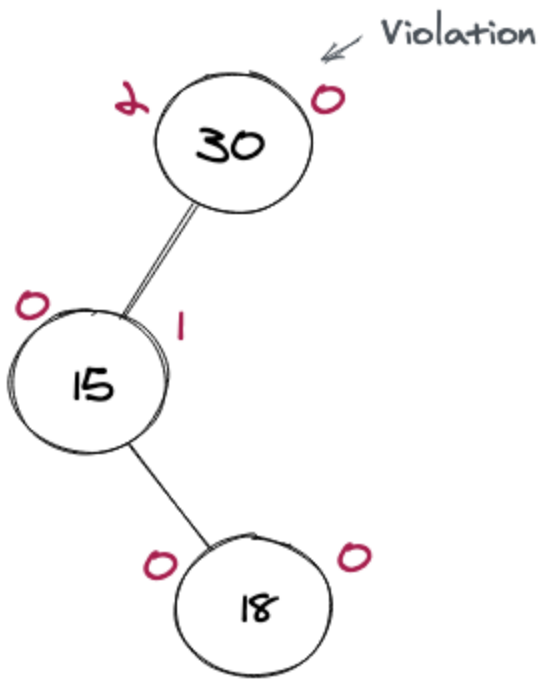
Consider the pictorial representation of making the above left-skewed tree into an AVL tree.



RIGHT ROTATE

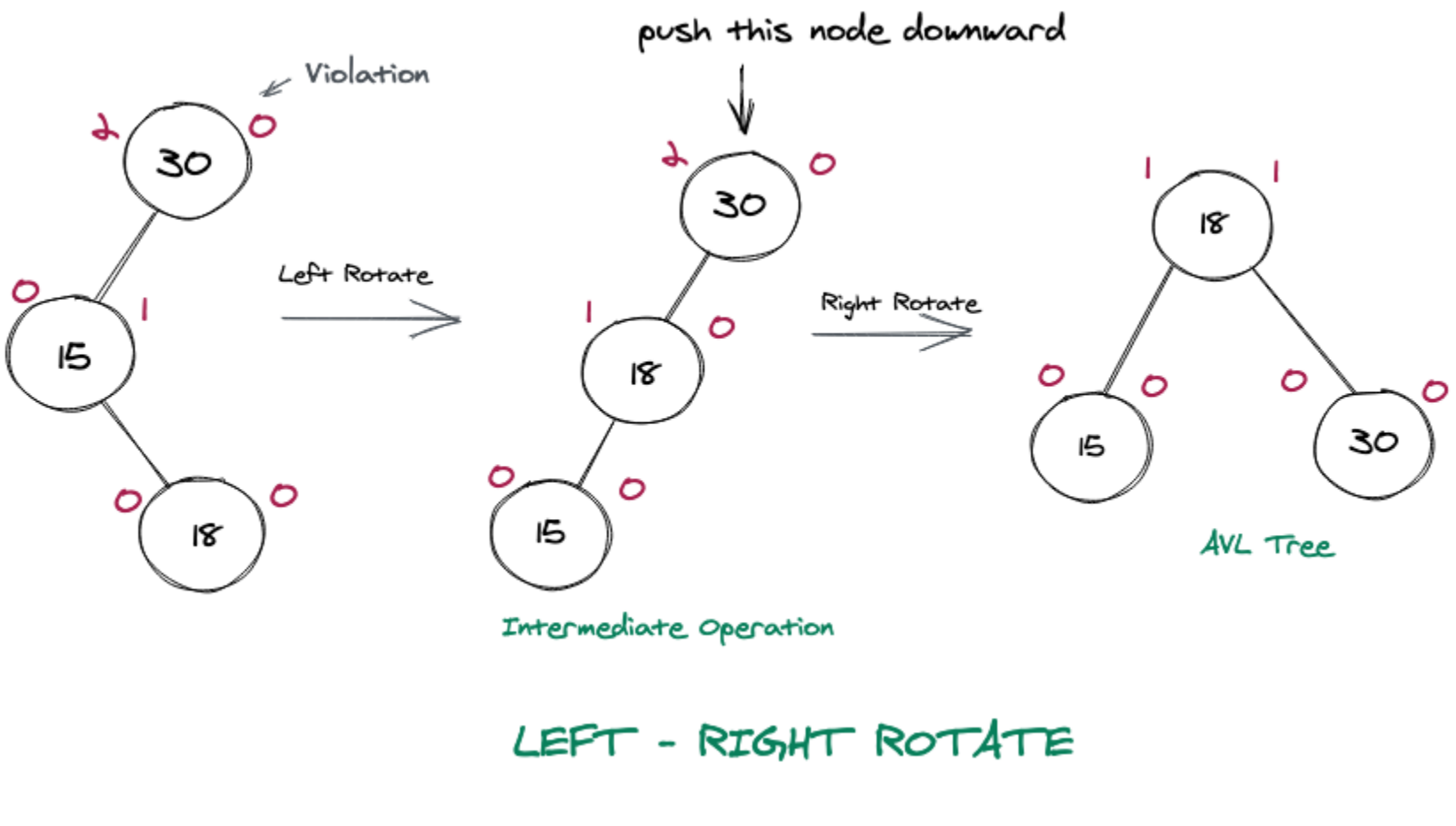
3. Left - Right Rotation

This rotation is performed when the violation occurs in the right subtree of the left child of the violating node. Consider the pictorial representation of a BST below that is not balanced, in order to make it so, we will do some rotations.



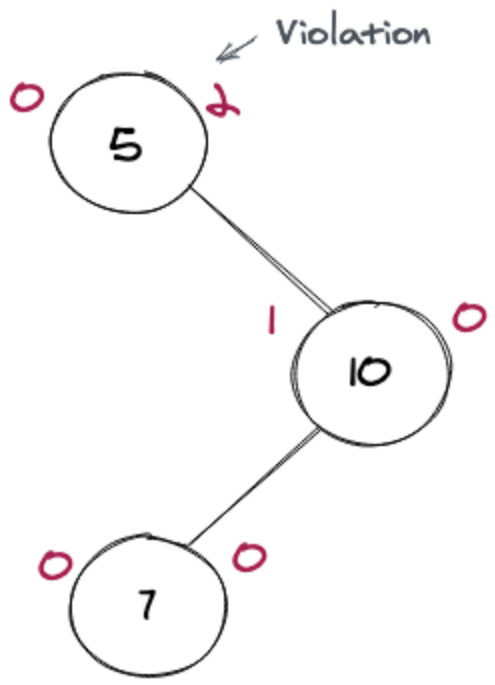
Clearly this is not an AVL tree, but consider the node where the violation occurs, we take this node as a pivot to do our rotations. The node that causes the issue is 18 as if we remove that one, we would have a balanced BST (i.e. AVL). Since the issue is because of the right child of the left subtree, we will do a Left-Right rotation to make it balanced.

Consider the pictorial representation of making the above tree into an AVL tree.



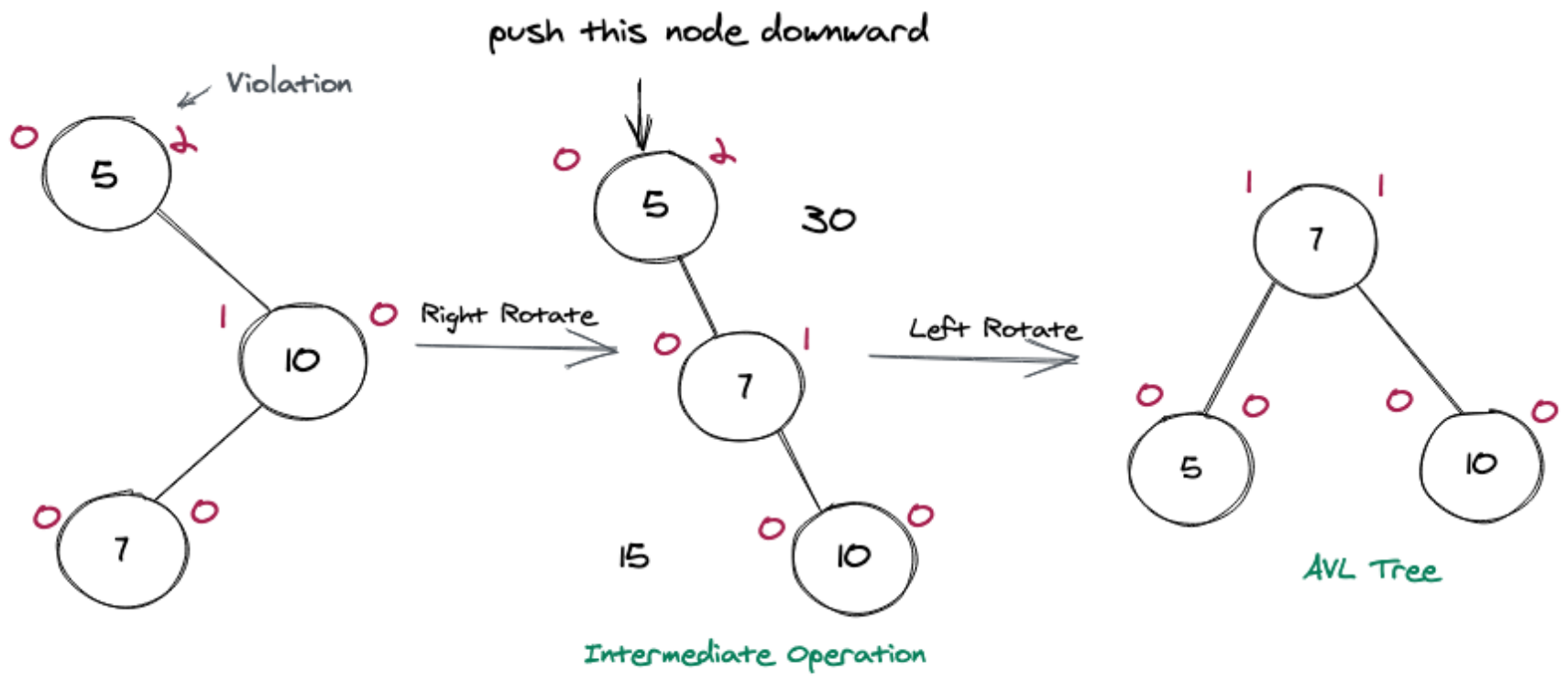
4. Right - Left Rotation

This rotation is performed when the violation occurs in the left subtree of the right child of the violating node. Consider the pictorial representation of a BST below that is not balanced, in order to make it so, we will do some rotations.



Clearly this is not an AVL tree, but consider the node where the violation occurs, we take this node as a pivot to do our rotations. The node that causes the issue is 7 as if we remove that one, we would have a balanced BST (i.e. AVL). Since the issue is because of the left child of the right subtree, we will do a Right-Left rotation to make it balanced.

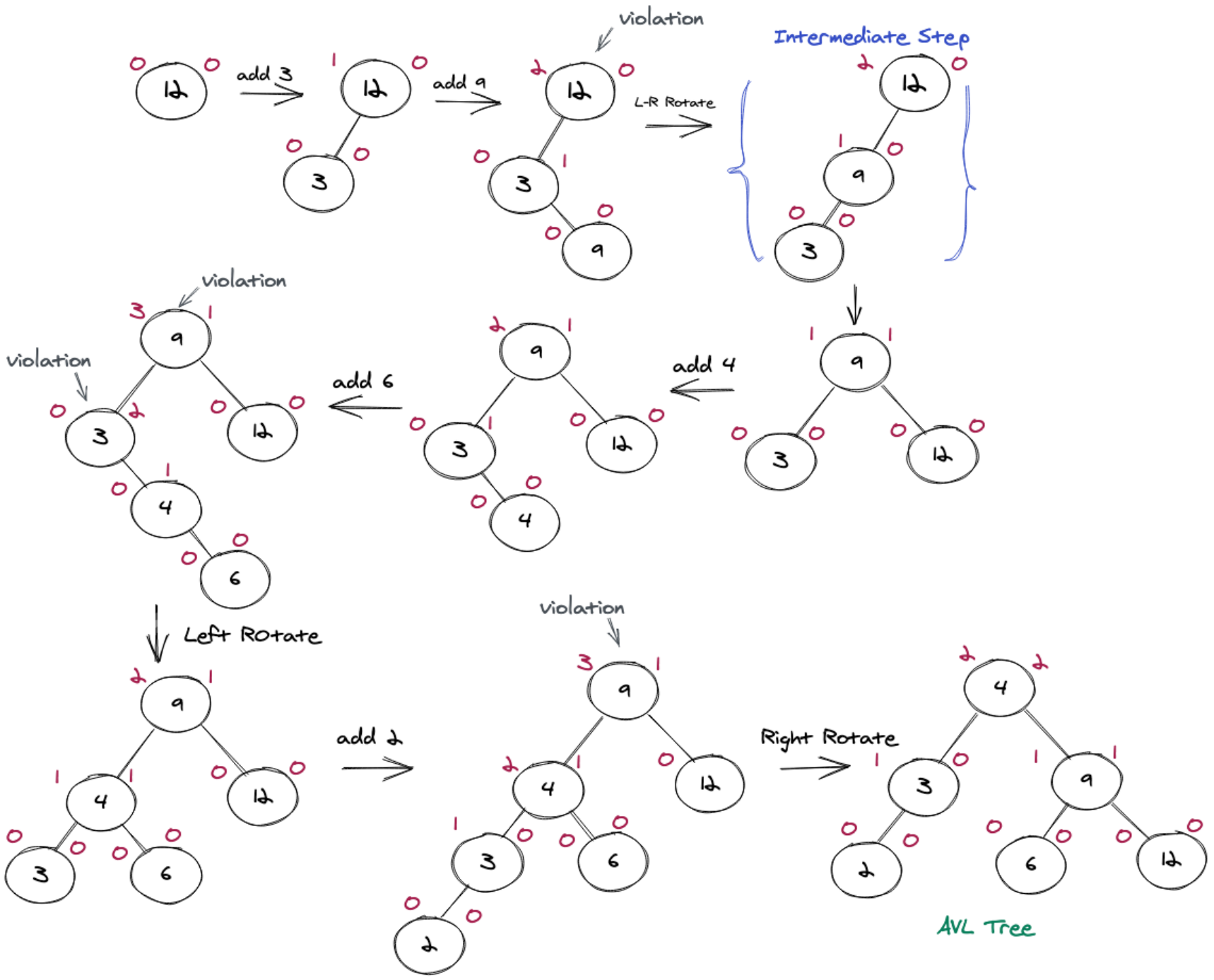
Consider the pictorial representation of making the above tree into an AVL tree.



AVL Tree Insertion

Let's take one array set of elements and build an AVL tree of these elements. Let, $\text{nums} = [12, 3, 9, 4, 6, 2]$

The complete step by step process of making an AVL tree (with rotations) is shown below.



INSERTING, IN AVL TREE

Key Points:

- An AVL tree with N number of nodes can have a minimum height of floor(logN) base 2.
- The height of an AVL tree with N number of nodes cannot exceed 1.44(logN) base 2.
- The maximum number of nodes in an AVL tree with height H can be : 2^H+1 - 1
- Minimum number of nodes with height h of an AVL tree can be represented as : N(h) = N(h-1) + N(h-2) + 1 for n>2 where N(0) = 1 and N(1) = 2.

Conclusions

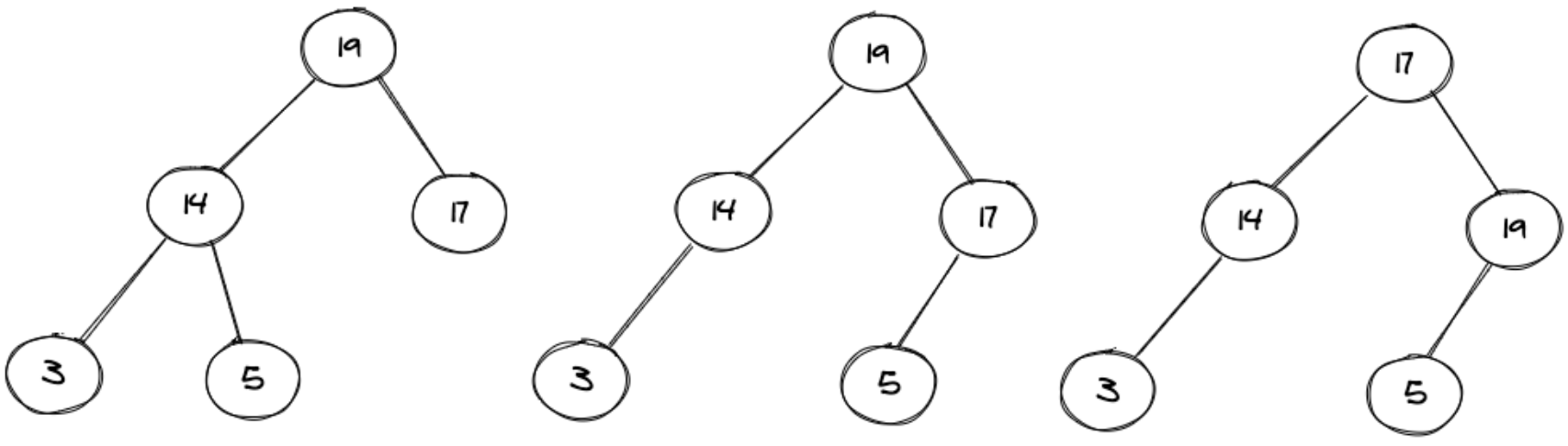
- We learned what an AVL tree is and why we need it.
- Then we learned the different type of Rotations that are possible in AVL tree to make it a balanced one.
- Followed by the rotations, we also did an insertion example in an AVL Tree.
- Lastly, we talked about different key points that we should remember with AVL

Heap Data Structure

A Heap is a special type of tree that follows two properties. These properties are :

- All leaves must be at `h` or `h-1` levels for some `h > 0`(complete binary tree property).
- The value of the node must be `>=` (or `<=`) the values of its children nodes, known as the heap property.

Consider the pictorial representation shown below:



In the pictures shown above, the leftmost tree denotes a heap (Max Heap) and the two tree to its right aren't heap as the middle tree violates the first heap property(not a complete binary tree) and the last tree from the left violates the second heap property(17 < 19).

Types of Heap

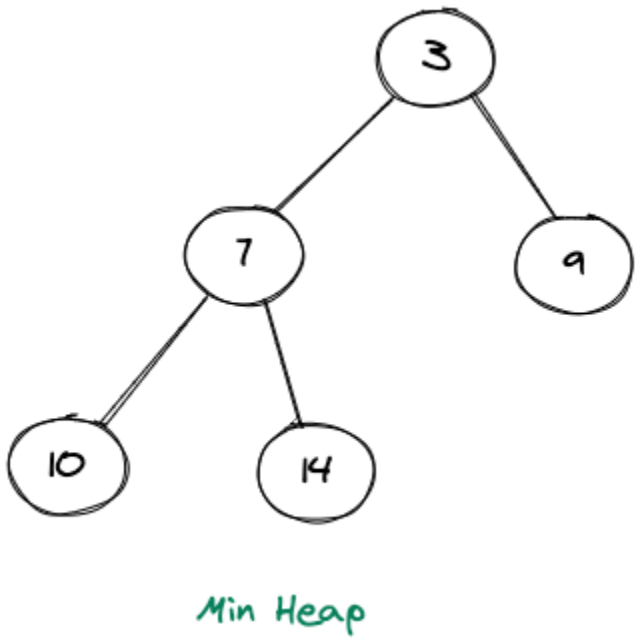
If we consider the properties of a heap, then we can have two types of heaps. These mainly are:

- Min Heap
- Max Heap

Min Heap

In this heap, the value of a node must be less than or equal to the values of its children nodes.

Consider the pictorial representation of a Min Heap below:

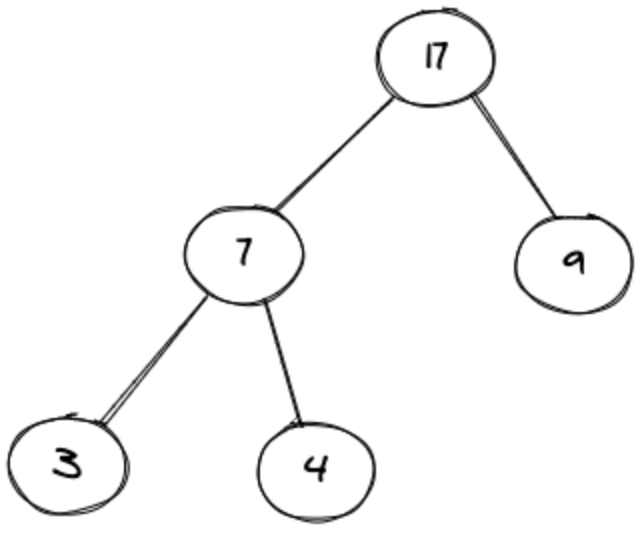


It can be clearly seen that the value of any node in the above heap is always less than the value of its children nodes.

Max Heap

In this heap, the value of a node must be greater than or equal to the values of its children nodes.

Consider the pictorial representation of a Max Heap below:



Max Heap

It can be clearly seen that the value of any node in the above heap is always greater than the value of its children nodes.

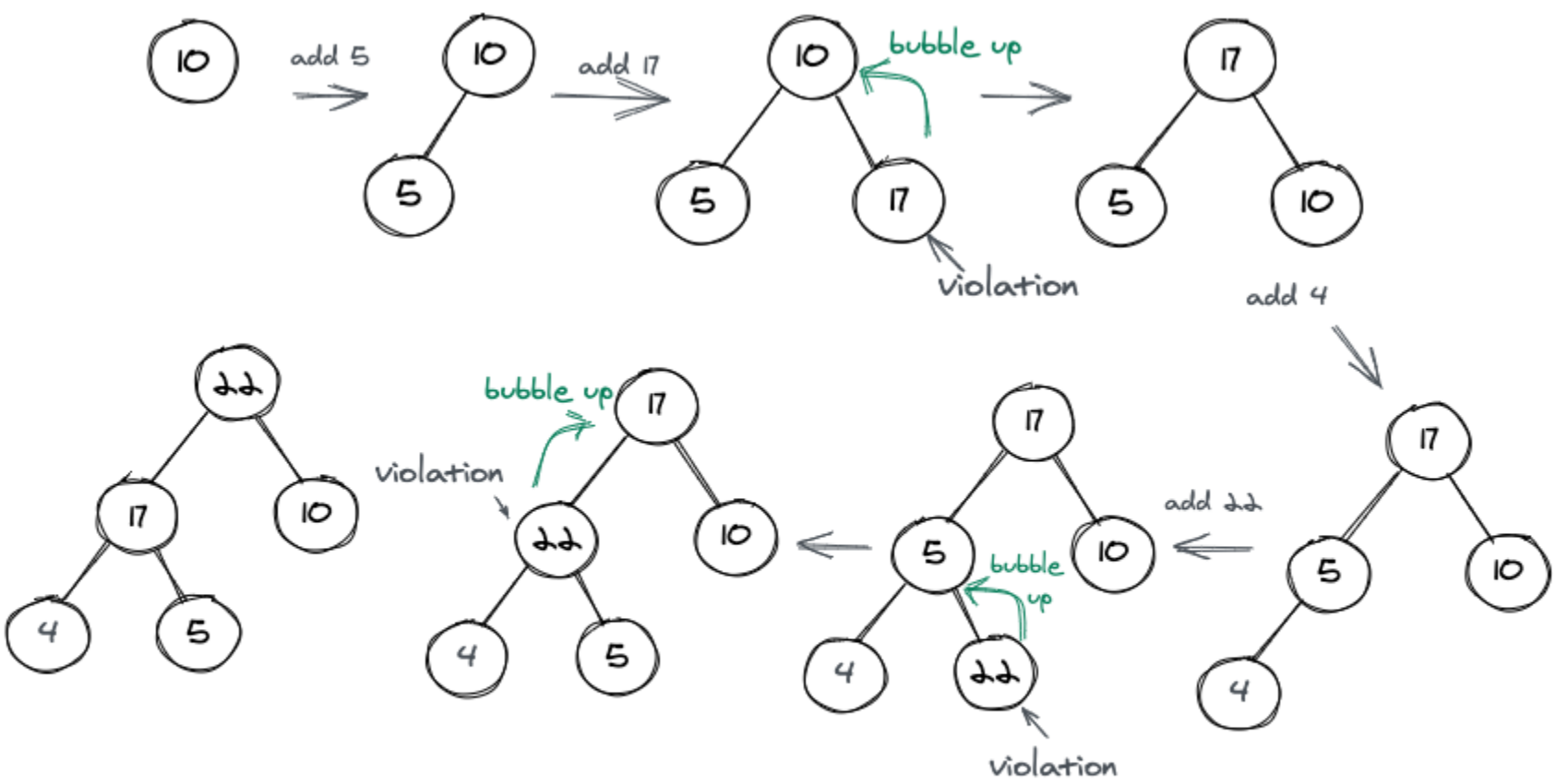
Building a Heap

Let's look at some operations like inserting an element in a heap or deleting an element from a heap.

1. Inserting an Element :

- First increase the heap size by 1.
- Then insert the new element at the available position(leftmost position ? Last level).
- Heapify the element from the bottom to the top(bubble up).

Building a heap includes adding elements into the heap. Let us consider an array of elements, namely nums = [10,5,17,4,22]. We want to make a Max Heap out of these elements, and the way we do that is shown in the pictorial representation below.

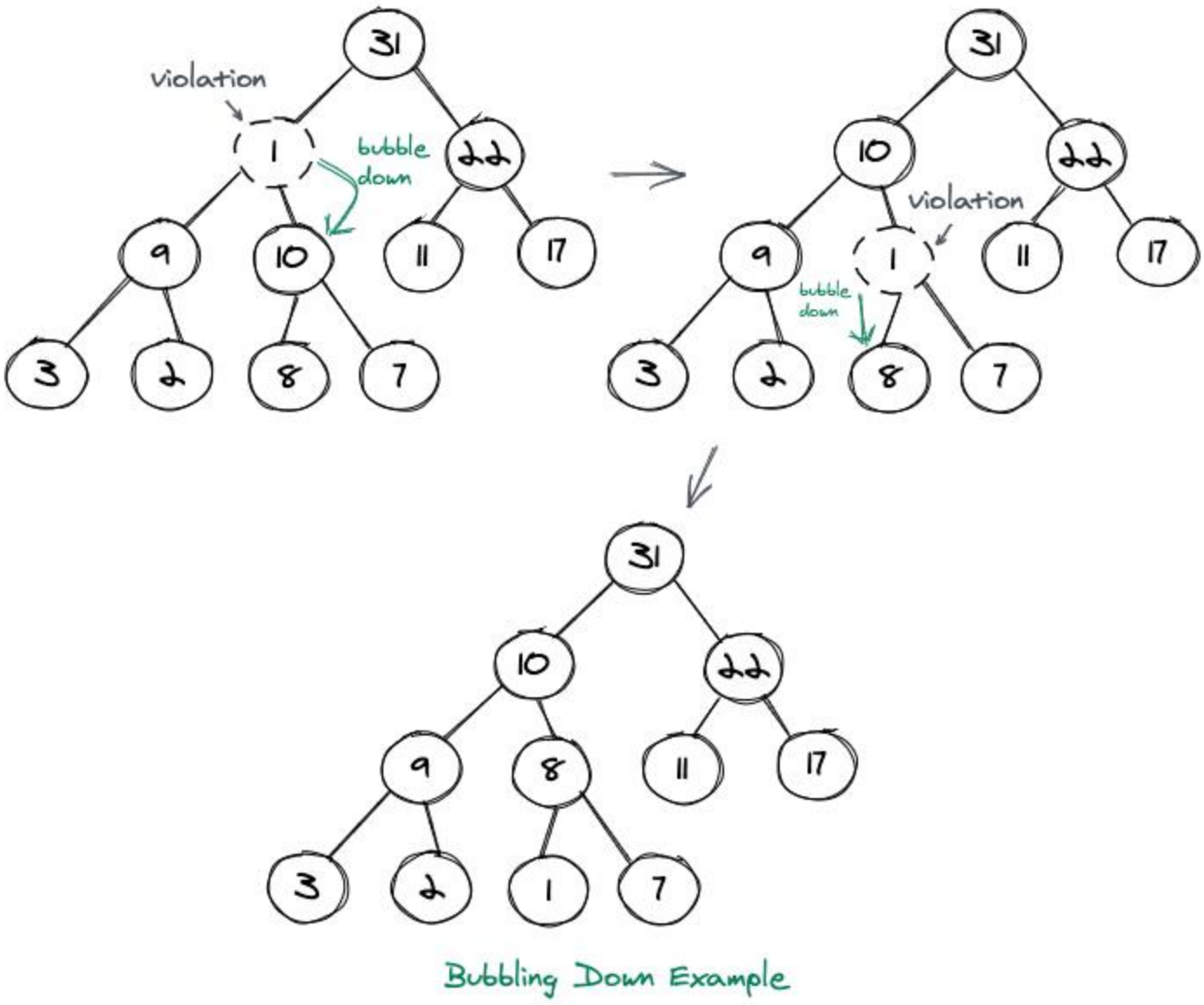


Building a Heap

Whenever we add an element while building a heap, it is quite possible that it will violate one of the properties. We take care of the first heap property by simply filling each level with maximum nodes and then when we move on to the next level, we fill the left child first and then the right child. But it is still possible that we have a violation regarding the second heap property, in that case we simply bubble up the current element that we have inserted and try to find the right position of it in the heap. Bubbling up includes swapping the current element with its parent till the heap is a max heap. In case of a min heap, we do the same procedure while adding an element to the heap.

The above representation shows three violations in total(17, 22, 22) and in all these cases we have basically swapped the current node with the parent node, hence bubbling up. It can also be noted that this process of bubbling up is also known as sift up.

Now let us look at another example, where we bubble down. Consider the pictorial representation of a tree(not heap) shown below:



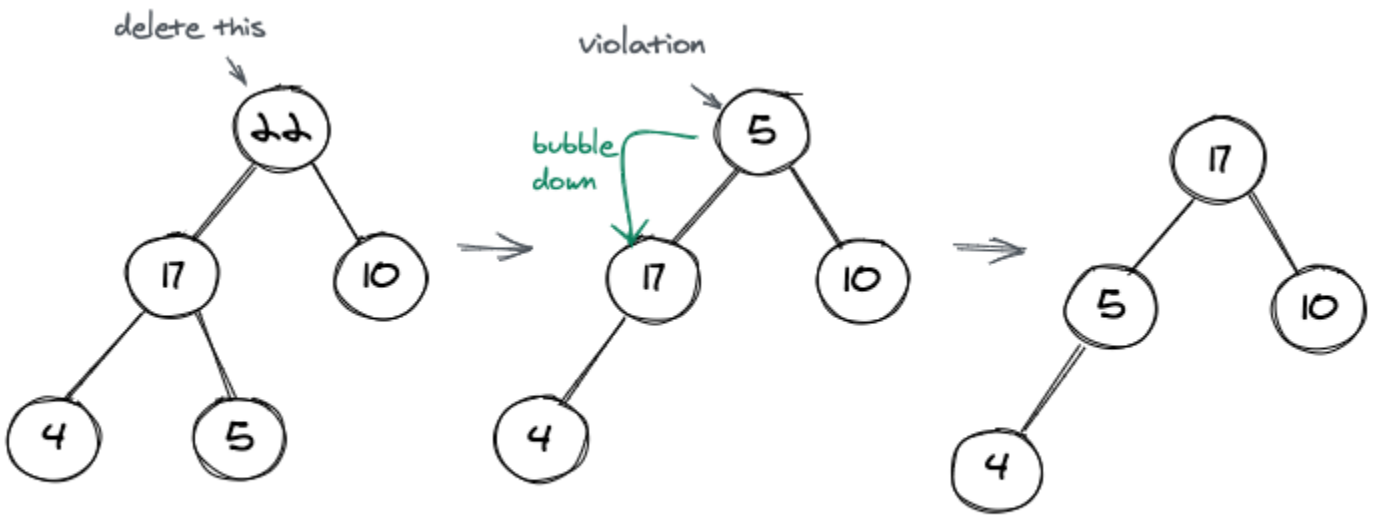
Bubbling Down Example

2. Deleting an Element :

- Copy the first element of the heap(root) into some variable
- Place the last element of the heap in the root's position
- Bubble Down to make it a valid heap

Whenever we are deleting an element, we simply delete the root element and replace it with the last element of the heap(the rightmost child of last level). We do that as we simply want to maintain the first property, as if we take any other element out of the heap we won't have a valid complete binary tree. We then place this node at the root, and it might be possible that this node will not satisfy the second property of the heap, and hence we bubble down to make it a valid heap. It can also be noted that this process of bubbling down is also known as sift down.

Consider the pictorial representation shown below:



Deletion in Heap

Applications of Binary Heaps

- Binary heaps are used in a famous sorting algorithm known as Heap sort.
- Binary heaps are also the main reason of implementing priority queues, as because of them the several priority queue operations like add(), remove() etc gets a time complexity of O(n).
- They are also the most preferred choice for solving Kth smallest / Kth Largest element questions.

Heap: Time Complexity Analysis

Let's see the time complexity for various operations of heap.

1. Inserting an Element:

Inserting an element in heap includes inserting it at the leaf level and then bubbling it up if it is somehow violating any property of the heap. We know that the heap is a complete binary tree, and the height of a complete binary tree is ($\log N$) where N represents the number of elements in the tree. So, if we consider the worst case scenario where we might have to swap this newly inserted node to the very top, we will have 1 swap at each level of the tree, hence we will require log N swaps. Hence, the worst case time complexity of Inserting an element in a binary heap is: O(log N)

2. Deleting an Element:

Deleting an element from a heap includes removing the root node and then swapping it with the last node of the last level, and then if this new root node violates any heap property, we need to swap it with the child node, until the tree is a valid binary heap. Since, in worst case scenarios we might have to swap this new root node with node at the lower levels to the very bottom(leaf level), which in turn means the height of the tree, the time complexity of deleting the node from the binary heap thus in turn is: $O(\log N)$.

3. Get Min/Max Element:

Getting the max(or min) element in a binary heap is simply a constant time operation, as we know that if it is a min heap the minimum will be the root node, and similarly in case of a max heap the maximum element will also be the root node. So, time complexity of extracting Min/Max is: $O(1)$.

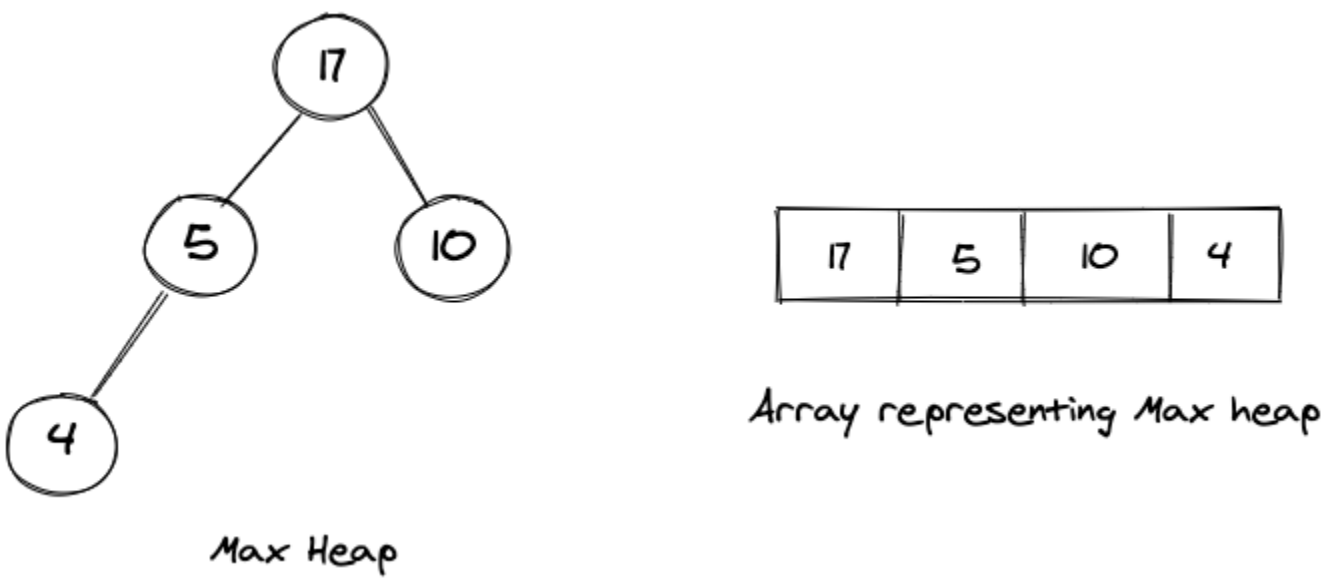
Conclusions

- We learned what a heap is, followed by the explanation of what min/max heaps are.
- Then we learned how to do insertion into a heap, followed by deletion of an element from the heap.
- Then we talked about the applications of binary heaps.
- Finally, we discussed the time complexity of heaps.

Implementing Heaps

In the previous article we learned all about heaps. It is time to implement them. Arrays are a preferred choice for implementing heaps because of the heap's complete binary tree property, we can be assured that no wastage of array locations will be there.

Let us consider a binary heap shown below, this heap can be represented by an array also shown in the pictorial representation below.



Implementing Heap

Let's see how we can implement a Heap data structure.

1. Structure of Heap

We need an array to store the elements of the heap, besides that we will also have a variable that represents the size of the heap. Consider the code snippet below:

```
private int[] items = new int[10];

private int size;
```

Copy

The initial capacity can be dynamic also, but here I have chosen a fixed length of 10 to represent the same.

2. Inserting into a Heap

When we want to insert an item into a heap, we will simply insert it into the array and then do a `bubbleUp()` operation if it doesn't satisfy any of the heap properties. The code will look something like this:

```
private void insert(int item){

        if(isFull()){

            throw new IllegalStateException();

        }

        items[size++] = item;

        bubbleUp();
```

```
}
```

Copy

A case might arrive where the heap is full, and then if we try to insert any more items into it, it should return an Exception. The `isFull()` method looks like this:

```
private boolean isFull(){ return size == items.length; }
```

Copy

After this `isFull()` call returns false, we move ahead and insert the item into our heap, and increase the size of the array as well. Now, the most important part is to correct the position of this element that we have inserted in the heap. We do this using `bubbleUp()` method. The code of `bubbleUp()` method is shown below:

```
private void bubbleUp(){

        int index = size - 1;

        while(index > 0 && items[index] > items[parent(index)]){

            swap(index,parent(index));

            index = parent(index);

        }

}
```

Copy

In the above method we are taking the index of the element we just inserted and then swapping it with the parent element in case this element is bigger than the parent element, also we are updating the index so that we don't pick the wrong element. The `swap()` method looks like this:

```
private void swap(int left,int right){

        int temp = items[left];

        items[left] = items[right];

        items[right] = temp;

}
```

Copy

It should also be noted that we are calling a parent method in our `bubbleUp()` method, and that parent method returns us the parent of a current index. The code of this method looks like this:

```
private int parent(int index){ return (index - 1)/2; }
```

Copy

The above methods are all we need to insert an element into a binary heap.

3. Removing from a Heap

Removing an element from a heap is a bit tricky process as when compared to inserting into a heap, as in this we need to `bubbleDown()` and this process of bubbling down involves checking the

parent element about its validity, also methods to insert into the tree as left as possible to maintain the complete binary tree property.

The `remove()` method looks like this:

```
private void remove(){

        if(isEmpty()){

            throw new IllegalStateException();

        }

        items[0] = items[--size];

        bubbleDown();

}
```

Copy

The first check is to make sure that we are not trying to remove anything from an empty heap. Then we insert the element at the start, and then `bubbleDown()` the element to it's correct position. The `bubbleDown()` method looks like this:

```
private void bubbleDown(){

        int index = 0;

        while(index <= size && !isValidParent(index)){

            int largerChildIndex = largerChildIndex(index);

            swap(index,largerChildIndex);

            index = largerChildIndex;

        }

}
```

Copy

We are checking that the index we have should be less than the size and also making sure that the parent node should not be a valid (should be less than the child values and other checks). The `largerChildIndex()` method return us the children item with which we will replace this current item while bubbling downwards. The code for `largerChildIndex()` looks like this:

```
private int largerChildIndex(int index){

        if(!hasLeftChild(index)) return index;

        if(!hasRightChild(index)) return leftChildIndex(index);

        return (leftChild(index) > rightChild(index)) ?
leftChildIndex(index) : rightChildIndex(index);

}
```

Copy

The code for isValidParent() method is shown below:

```
private boolean isValidParent(int index){

        if(!hasLeftChild(index)) return true;

        var isValid = items[index] >= leftChild(index);

        if(hasRightChild(index)){

            isValid &= items[index] >= rightChild(index);

        }

        return isValid;

}
```

Copy

In the above code snippet we are just making sure that the parent is valid, and also if the left child is null then we return true. The leftChild(), rightChild(), hasLeftChild(), hasRightChild() methods looks like this:

```
private int leftChild(int index){

        return items[leftChildIndex(index)];

}



private int rightChild(int index){

        return items[rightChildIndex(index)];

}



private boolean hasLeftChild(int index){

        return leftChildIndex(index) <= size;

}



private boolean hasRightChild(int index){

        return rightChildIndex(index) <= size;

}
```

Copy

The above method calls the methods to get the left and right children of a binary heap element, these are:

```
private int leftChildIndex(int index){

        return 2*index + 1;

}



private int rightChildIndex(int index){

        return 2*index + 2;

}
```

Copy

Now we are done with all the methods we need to insert or delete an element from the heap. The complete code is given below:

```
public class Heap {


    private int[] items = new int[10];

    private int size;


    private void insert(int item){

        if(isFull()){

            throw new IllegalStateException();

        }

        items[size++] = item;

        bubbleUp();

    }


    private void remove(){

        if(isEmpty()){

            throw new IllegalStateException();

        }

        items[0] = items[--size];

        bubbleDown();

    }
```

```
    private int largerChildIndex(int index){

        if(!hasLeftChild(index)) return index;

        if(!hasRightChild(index)) return leftChildIndex(index);

        return (leftChild(index) > rightChild(index)) ?
leftChildIndex(index) : rightChildIndex(index);

    }



    private boolean isValidParent(int index){

        if(!hasLeftChild(index)) return true;

        var isValid = items[index] >= leftChild(index);

        if(hasRightChild(index)){

            isValid &= items[index] >= rightChild(index);

        }

        return isValid;

    }


    private int leftChild(int index){

        return items[leftChildIndex(index)];

    }


    private int rightChild(int index){

        return items[rightChildIndex(index)];

    }


    private int leftChildIndex(int index){

        return 2*index + 1;

    }


    private int rightChildIndex(int index){

        return 2*index + 2;

    }
```

```
    private boolean hasLeftChild(int index){
        return leftChildIndex(index) <= size;
    }

    private boolean hasRightChild(int index){
        return rightChildIndex(index) <= size;
    }

    private boolean isFull(){
        return size == items.length;
    }

    private boolean isEmpty(){
        return size == 0;
    }

    private void bubbleUp(){
        int index = size - 1;
        while(index > 0 && items[index] > items[parent(index)]){
            swap(index,parent(index));
            index = parent(index);
        }
    }

    private void bubbleDown(){
        int index = 0;
        while(index <= size && !isValidParent(index)){
            int largerChildIndex = largerChildIndex(index);
            swap(index,largerChildIndex);
```

```
            index = largerChildIndex;

        }

    }


    private int parent(int index){

        return (index - 1)/2;

    }


    private void swap(int left,int right){

        int temp = items[left];

        items[left] = items[right];

        items[right] = temp;

    }


    private int getMax(){

        return items[0];

    }


    public static void main(String[] args) {

        Heap heap = new Heap();

        heap.insert(10);

         heap.insert(5);

         heap.insert(17);

         heap.insert(4);

         heap.insert(22);

        /// heap.remove();

         System.out.println("Done!");

    }

}
```

Copy

Conclusion

- We learned how to implement heaps, mainly the **bubbleUp()** and **bubbleDown()** methods.

Trie Data Structure - Explained with Examples

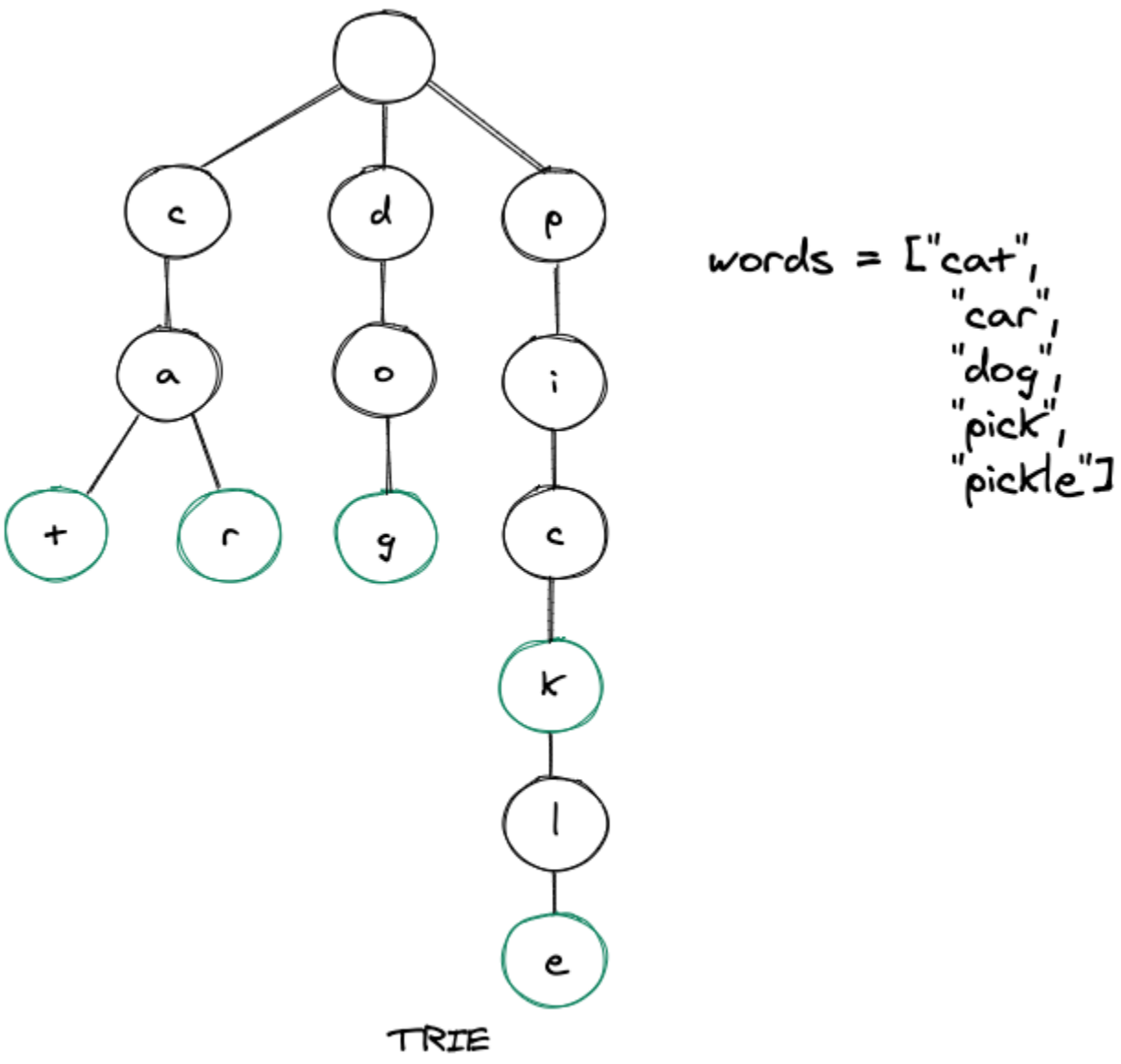
A **Trie is an advanced data structure** that is sometimes also known as **prefix tree or digital tree**. It is a tree that stores the data in an ordered and efficient way. We generally **use trie's to store strings**. Each node of a trie can have as many as 26 references (pointers).

Each node of a trie consists of two things:

- A character
- A boolean value is used to implement whether this character represents the end of the word.

Tries in general are used to store English characters, hence each character can have 26 references. Nodes in a trie do not store entire keys, instead, they store a part of the key(usually a character of the string). When we traverse down from the root node to the leaf node, we can build the key from these small parts of the key.

Let's build a trie by inserting some words in it. Below is a pictorial representation of the same, we have 5 words, and then we are inserting these words one by one in our trie.



As it can be seen in the image above, the key(words) can be formed as we traverse down from the root node to the leaf nodes. It can be noted that the green highlighted nodes, represents the `endOfWord` boolean value of a word which in turn means that this particular word is completed at this node. Also, the root node of a trie is empty so that it can refer to all the members of the alphabet the trie is using to store, and the children nodes of any node of a trie can have at most 26 references. Tries are not balanced in nature, unlike AVL trees.

Why use Trie Data Structure?

When we talk about the **fastest ways to retrieve values** from a data structure, **hash tables generally comes to our mind**. Though very efficient in nature but still very less talked about as when compared to hash tables, **trie's are much more efficient than hash tables** and also they possess several advantages over the same. Mainly:

- There won't be any collisions hence making the worst performance better than a hash table that is not implemented properly.
- No need for hash functions.
- Lookup time for a string in trie is `O(k)` where **k = length of the word**.
- It can take even less than `O(k)` time when the word is not there in a trie.

Implementing Tries:

We will implement the Trie data structure in Java language.

Trie Node Declaration:

```
class TrieNode {

    boolean isEndOfWord;

    TrieNode children[];



    public TrieNode(){

        isEndOfWord = false;

        children = new TrieNode[26];

    }

}
```

Copy
Note that we have two fields in the above **TrieNode** class as explained earlier, the boolean isEndOfWord keyword and an array of Trie nodes named children. Now let's initialize the root node of the trie class.

```
TrieNode root;



public Trie() {

    root = new TrieNode();

}
```

Copy
There are two key functions in a trie data structure, these are:

- **Search**
- **Insert**

Insert in trie:

When we insert a character(part of a key) into a trie, we start from the root node and then search for a reference, which corresponds to the first key character of the string whose character we are trying to insert in the trie. Two scenarios are possible:

- A reference exists, if, so then we traverse down the tree following the reference to the next children level.
- A reference does not exist, then we create a new node and refer it with parents reference matching the current key character. We repeat this step until we get to the last character of the key, then we mark the current node as an end node and the algorithm finishes.

Consider the code snippet below:

```
public void insert(String word) {

    TrieNode node = root;



    for (char c : word.toCharArray()) {

        if (node.children[c-'a'] == null) {

            node.children[c-'a'] = new TrieNode();

        }

        node = node.children[c-'a'];

    }
```

```
    node.isEndOfWord = true;

}
```

Copy

Search in trie:

A key in a trie is stored as a path that starts from the root node and it might go all the way to the leaf node or some intermediate node. If we want to search a key in a trie, we start with the root node and then traverses downwards if we get a reference match for the next character of the key we are searching, then there are two cases:

1. A reference of the next character exists, hence we move downwards following this link, and proceed to search for the next key character.
2. A reference does not exist for the next character. If there are no more characters of the key present and this character is marked as `isEndOfWord = true`, then we return **true**, implying that we have found the key. Otherwise, two more cases are possible, and in each of them we return **false**. These are:
    1. There are key characters left in the key, but we cannot traverse down as the path is terminated, hence the key doesn't exist.
    2. No characters in the key are left, but the last character is not marked as `isEndOfWord = false`. Therefore, the search key is just the prefix of the key we are trying to search in the trie.

Consider the code snippet below:

```
public boolean search(String word) {

    return isMatch(word, root, 0, true);

}



public boolean startsWith(String prefix) {

    return isMatch(prefix, root, 0, false);

}



public boolean isMatch( String s, TrieNode node, int index, boolean isFullMatch) {

    if (node == null)

        return false;



    if (index == s.length())

        return !isFullMatch || node.isEndOfWord;



    return isMatch(s, node.children[s.charAt(index) - 'a'], index + 1, isFullMatch);

}
```

Copy
The method `startsWith()` is used to find if the desired key prefix is present in the trie or not. Also, both the `search()` and `startsWith()` methods make use of `isMatch()` method.

Entire Code:

```
class Trie {

    class TrieNode {
```

```
        boolean isEndOfWord;
        TrieNode children[];

        public TrieNode(){
            isEndOfWord = false;
            children = new TrieNode[26];
        }
    }

    TrieNode root;

    public Trie() {
        root = new TrieNode();
    }

    public void insert(String word) {
        TrieNode node = root;

        for (char c : word.toCharArray()) {
            if (node.children[c-'a'] == null) {
                node.children[c-'a'] = new TrieNode();
            }
            node = node.children[c-'a'];
        }
        node.isEndOfWord = true;
    }

    public boolean search(String word) {
        return isMatch(word, root, 0, true);
    }

    public boolean startsWith(String prefix) {
        return isMatch(prefix, root, 0, false);
```

```
    }


    public boolean isMatch( String s, TrieNode node, int index, boolean isFullMatch) {
        if (node == null)
            return false;

        if (index == s.length())
            return !isFullMatch || node.isEndOfWord;

        return isMatch(s, node.children[s.charAt(index) - 'a'], index + 1, isFullMatch);
    }


    public static void main(String[] args){
        Trie trie = new Trie();
        trie.insert("cat");
        trie.insert("car");
        trie.insert("dog");
        trie.insert("pick");
        trie.insert("pickle");
        boolean isPresent = trie.search("cat");
        System.out.println(isPresent);
        isPresent = trie.search("picky");
        System.out.println(isPresent);
        isPresent = trie.startsWith("ca");
        System.out.println(isPresent);
        isPresent = trie.startsWith("pen");
        System.out.println(isPresent);


    }
}
```

Copy
The output of the above looks like this:

true

false

true

false

Trie Applications

- The most common use of tries in real world is the **autocomplete feature** that we get on a search engine(now everywhere else too). After we type something in the search bar, the tree of the potential words that we might enter is greatly reduced, which in turn allows the program to enumerate what kinds of strings are possible for the words we have typed in.
- Trie also helps in the case where we want to store additional information of a word, say the popularity of the word, which makes it so powerful. You might have seen that when you type **"foot"** on the search bar, you get **"football"** before anything say **"footpath"**. It is because **"football"** is a much popular word.
- Trie also has helped in checking the correct spellings of a word, as the path is similar for a slightly misspelled word.
- **String matching** is another case where tries to excel a lot.

Key Points

- The time complexity of creating a trie is `O(m*n)` where m = number of words in a trie and n = average length of each word.
- Inserting a node in a trie has a time complexity of `O(n)` where n = length of the word we are trying to insert.
- Inserting a node in a trie has a space complexity of `O(n)` where n = length of the word we are trying to insert.
- Time complexity for searching a key(word) in a trie is `O(n)` where n = length of the word we are searching.
- Space complexity for searching a key(word) in a trie is `O(1)`.
- Searching for a prefix of a key(word) also has a time complexity of `O(n)` and space complexity of `O(1)`.

Conclusions

- We learned what a Trie is, and why do we need one.
- We also implemented trie in Java, where insert and search were the main methods implemented.
- We then talked about the applications of Trie data structure in real world, followed by several key points that we should also remember about trie's.

B Trees (M-way Trees) Data Structure

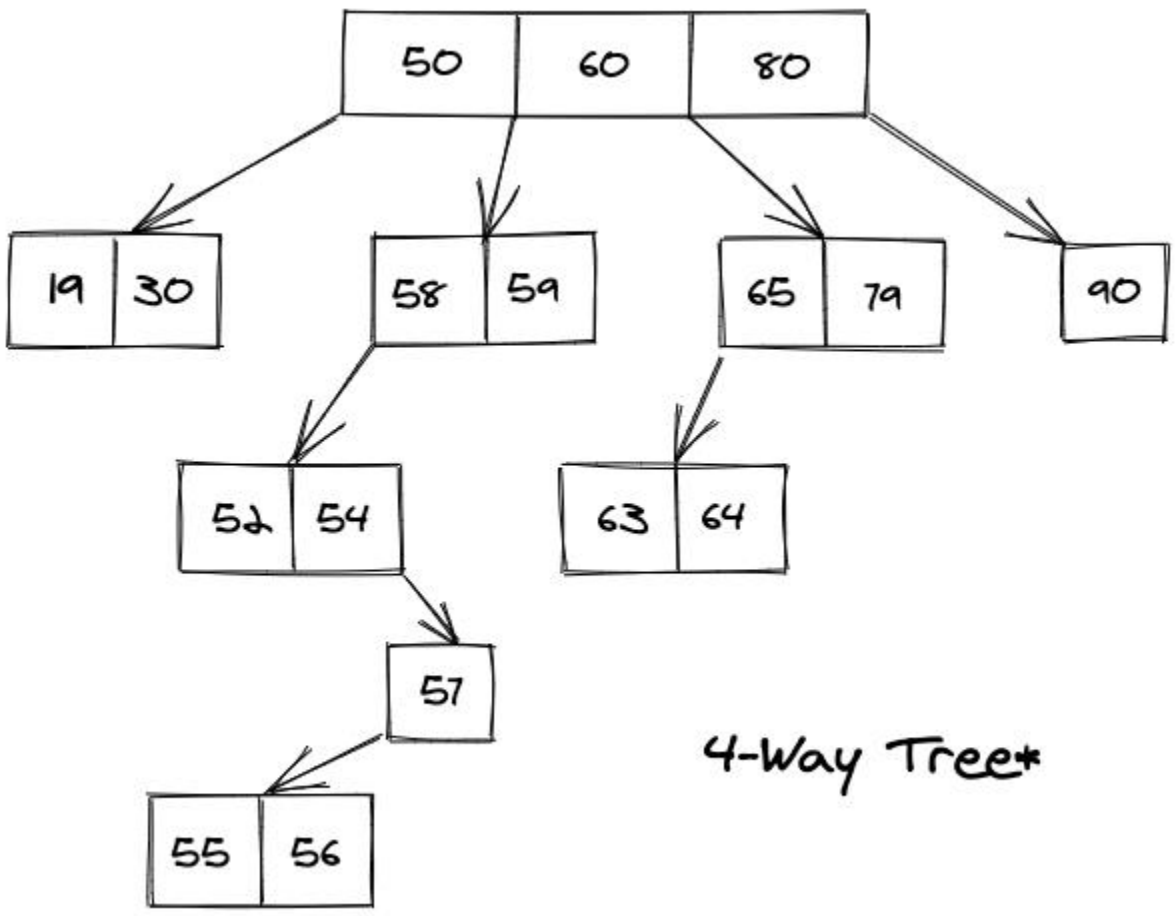
A B-Tree is a special type of M-way search tree.

M-way Trees

Before learning about B-Trees we need to know what M-way trees are, and how B-tree is a special type of M-way tree. 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.

Consider the pictorial representation shown below of an **M-way tree**.



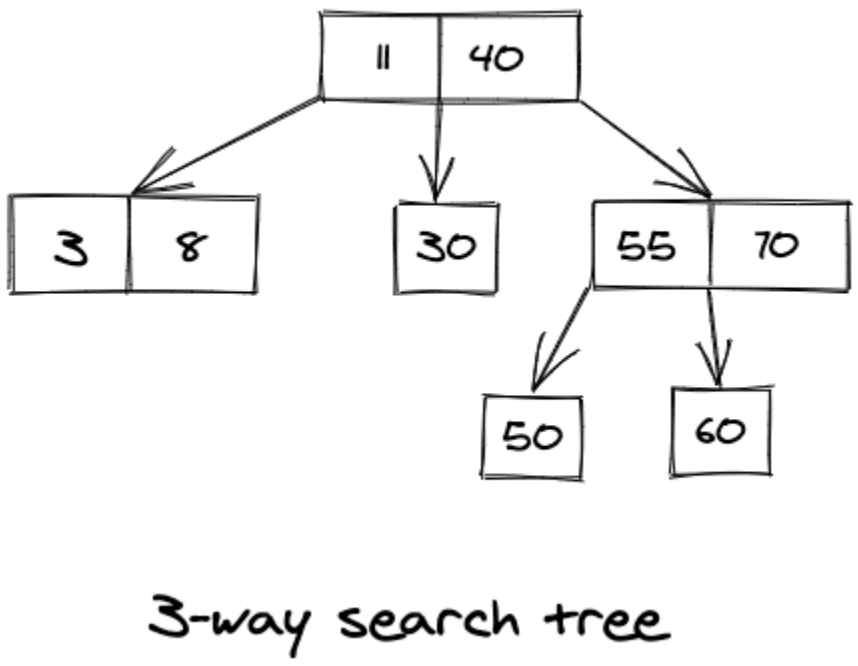
The above image shows a **4-way tree**, where each node can have at most **3(4-1) key fields** and at most **4** children. It is also a **4-way search tree**.

M-way Search Trees

An M-way search tree is a more constrained m-way tree, and these constrain mainly apply to the key fields and the values in them. The constraints on an **M-way** tree that makes it an M-way search tree are:

- 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 **Kth** key of this node.
- The keys in the last **(m-K**) children are higher than the **Kth** key.

Consider the pictorial representation shown below of an M-way search tree:



M-way search trees have the same advantage over the M-way trees, which is making the search and update operations much more efficient. Though, they can become unbalanced which in turn leaves us to the same issue of searching for a key in a skewed tree which is not much of an advantage.

Searching in an M-way Search Tree:

If we want to search for a value say **X** in an M-way search tree and currently we are at a node that contains key values from **Y1, Y2, Y3,.....,Yk**. Then in total 4 cases are possible to deal with this scenario, these are:

- If **X < Y1**, then we need to recursively traverse the left subtree of **Y1**.
- If **X > Yk**, then we need to recursively traverse the right subtree of **Yk**.
- If **X = Yi**, for some **i**, then we are done, and can return.
- Last and only remaining case is that when for some **i** we have **Yi < X < Y(i+1)**, then in this case we need to recursively traverse the subtree that is present in between **Yi** and **Y(i+1)**.

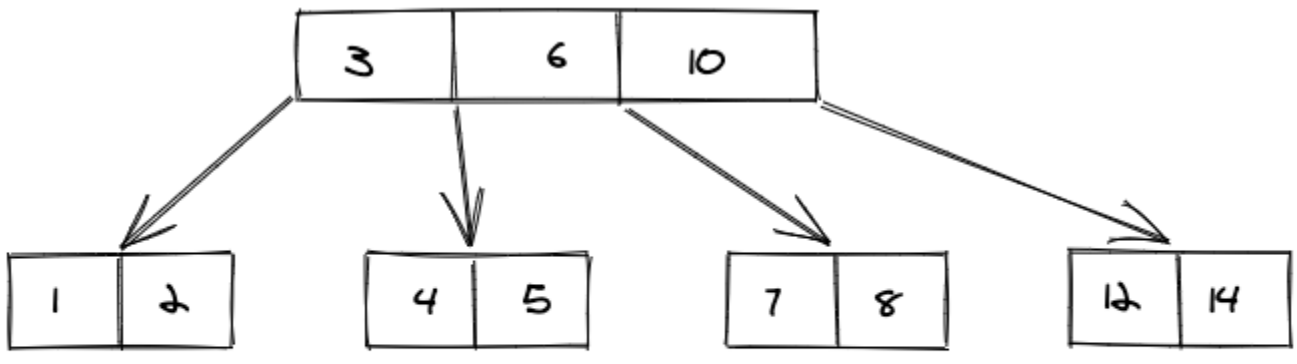
For example, consider the 3-way search tree that is shown above, say, we want to search for a node having key(X) equal to 60. Then, considering the above cases, for the root node, the second condition applies, and **(60 > 40)** and hence we move on level down to the right subtree of **40**. Now, the last condition is valid only, hence we traverse the subtree which is in between the **55** and **70**. And finally, while traversing down, we have our value that we were looking for.

B Trees Data Structure:

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 the leaf nodes in a B tree are at the same level.
- All internal nodes must have **M/2** children.
- If the root node is a non leaf node, then it must have at least two children.
- All nodes except the root node, must have at least **[M/2]-1** keys and at most **M-1** keys.

Consider the pictorial representation of a B tree shown below:



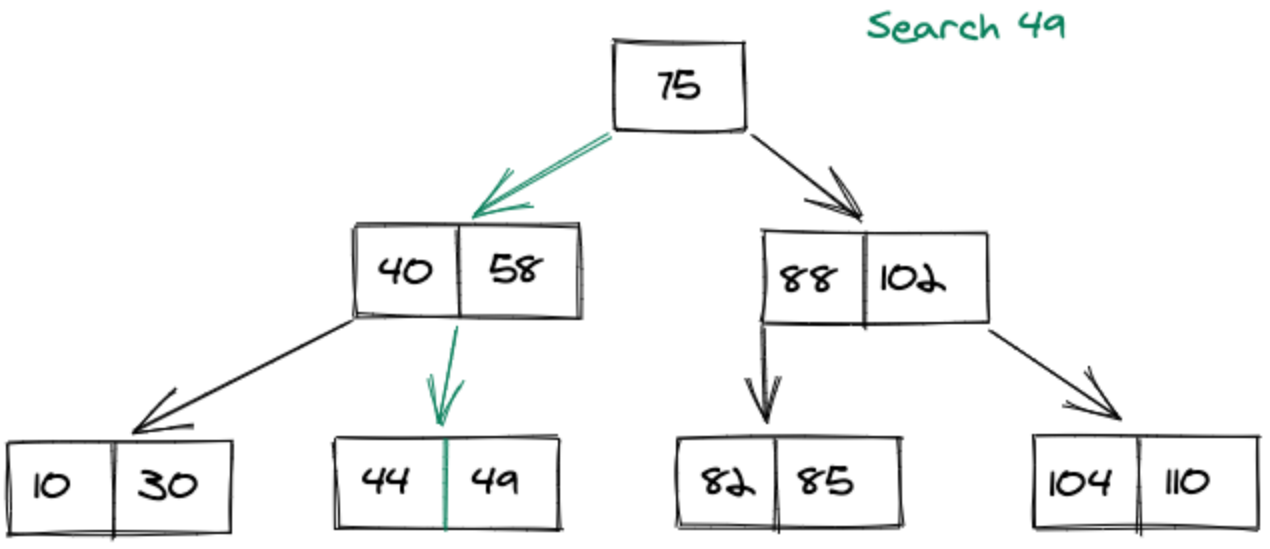
B tree

Searching in a B Tree:

Searching for a key in a B Tree is exactly like searching in an M-way search tree, which we have seen just above. Consider the pictorial representation shown below of a B tree, say we want to search for a key 49 in the below shown B tree. We do it as following:

- Compare item **49 with root node 75**. Since **49 < 75** hence, move to its left sub-tree.
- Since, **40<49<58**, traverse right sub-tree of 40.
- **49>44**, move to right. Compare **49**.
- We have found 49, hence returning.

Consider the pictorial representation shown below:

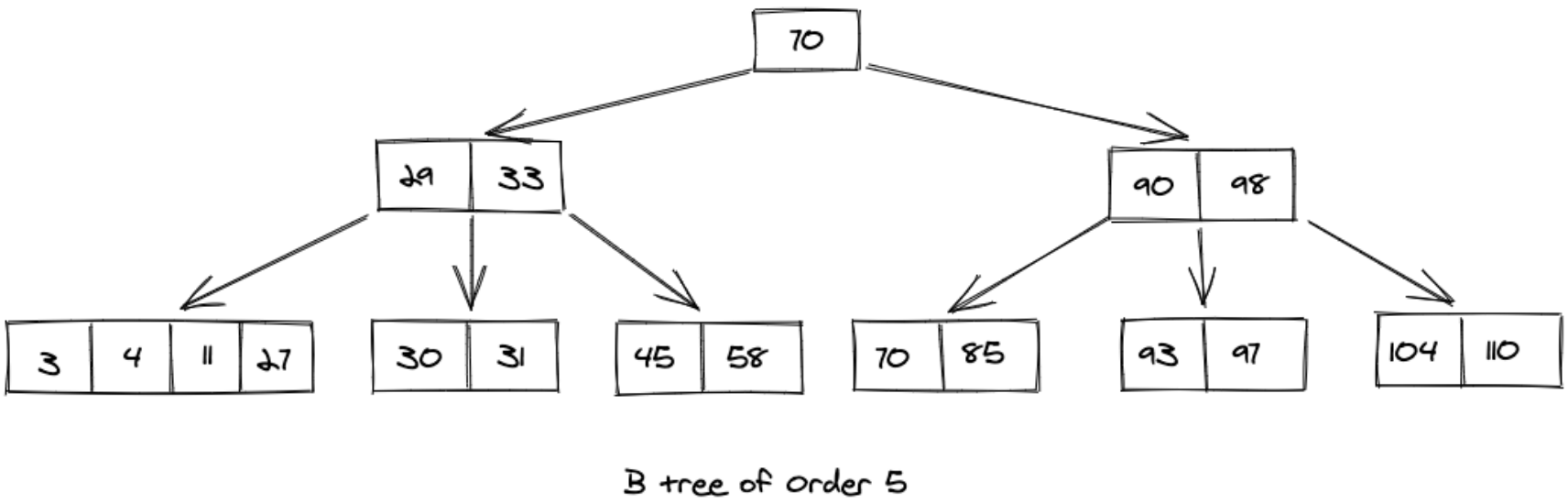


Inserting in a B Tree:

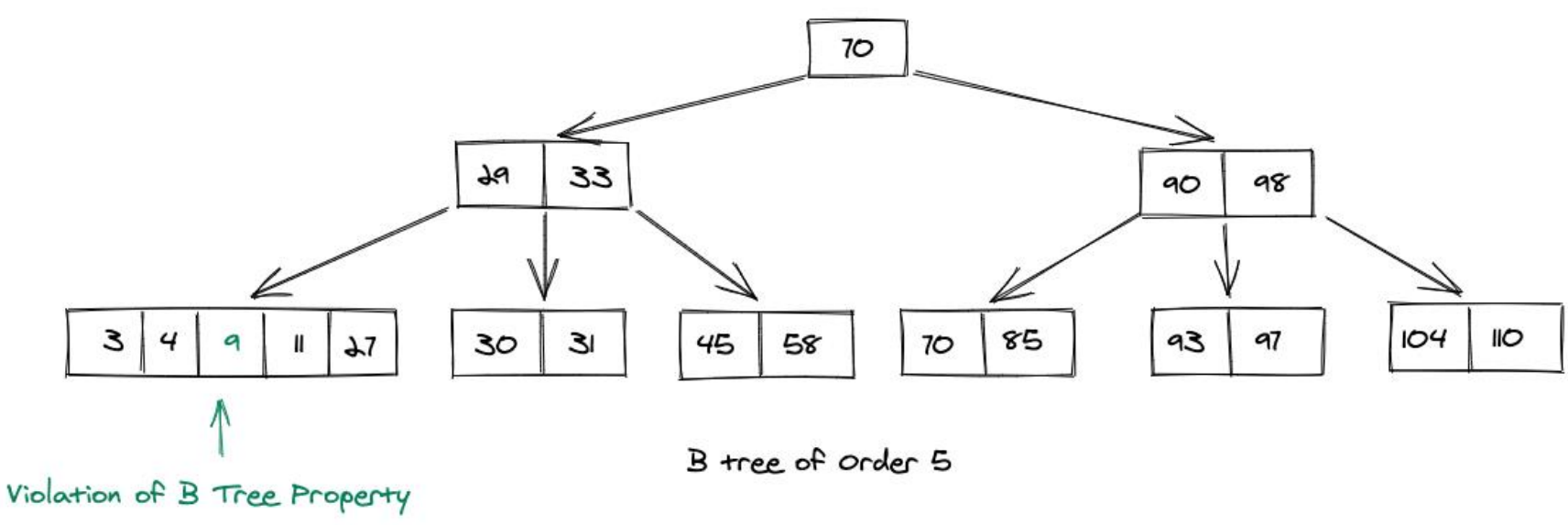
Inserting in a B tree is done at the leaf node level. We follow the given steps to make sure that after the insertion the B tree is valid, these are:

- First, we traverse the B tree to find the appropriate node where the to be inserted key will fit.
- If that node contains less than **M-1** keys, then we insert the key in an increasing order.
- If that node contains exactly **M-1** keys, then we have two cases ? Insert the new element in increasing order, split the nodes into two nodes through the median, push the median element up to its parent node, and finally if the parent node also contains **M-1 keys**, then we need to repeat these steps.

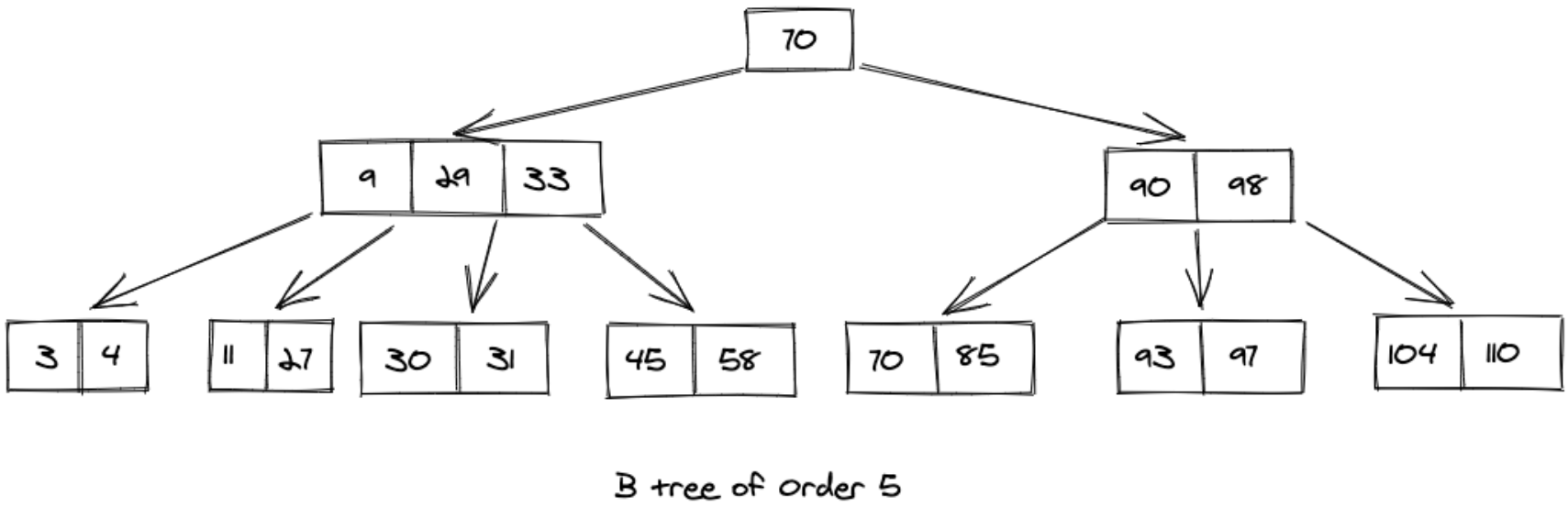
Consider the pictorial representation shown below:



Now, consider that we want to insert a key 9 into the above shown B tree, the tree after inserting the key 9 will look something like this:



Since, a violation occurred, we need to push the median node to the parent node, and then split the node in two parts, hence the final look of B tree is:



B tree of order 5

Deletion in a B Tree:

Deletion of a key in a B tree includes two cases, these are:

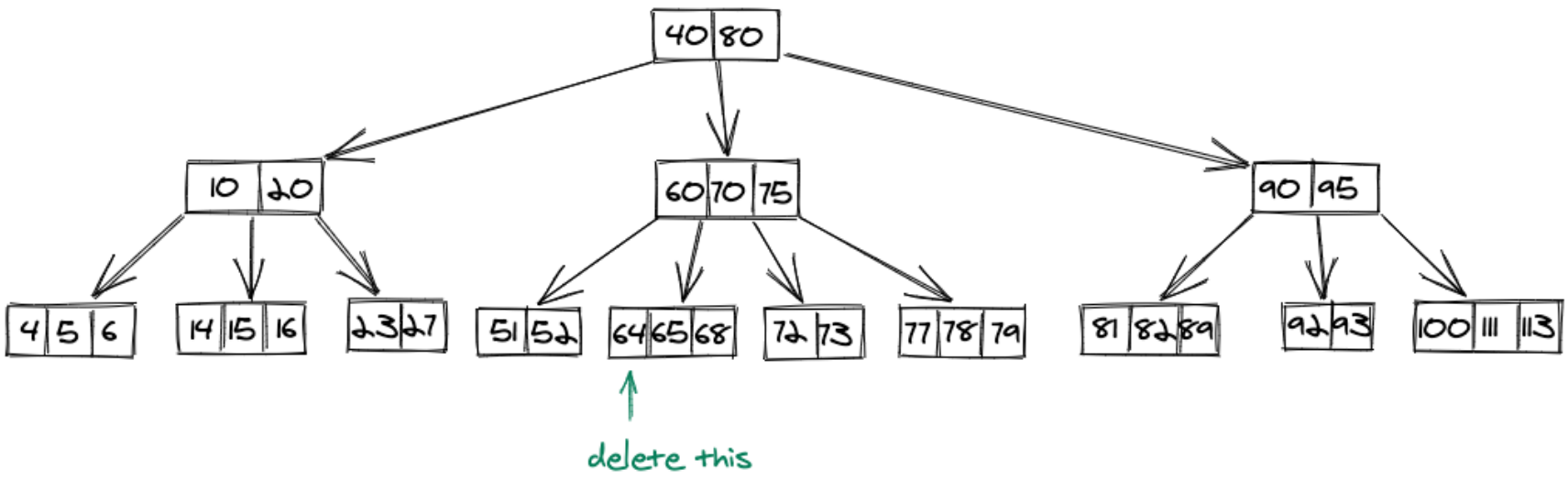
- **Deletion of key from a leaf node**
- **Deletion of a key from an Internal node**

Deletion of Key from a leaf node:

If we want to delete a key that is present in a leaf node of a B tree, then we have two cases possible, these are:

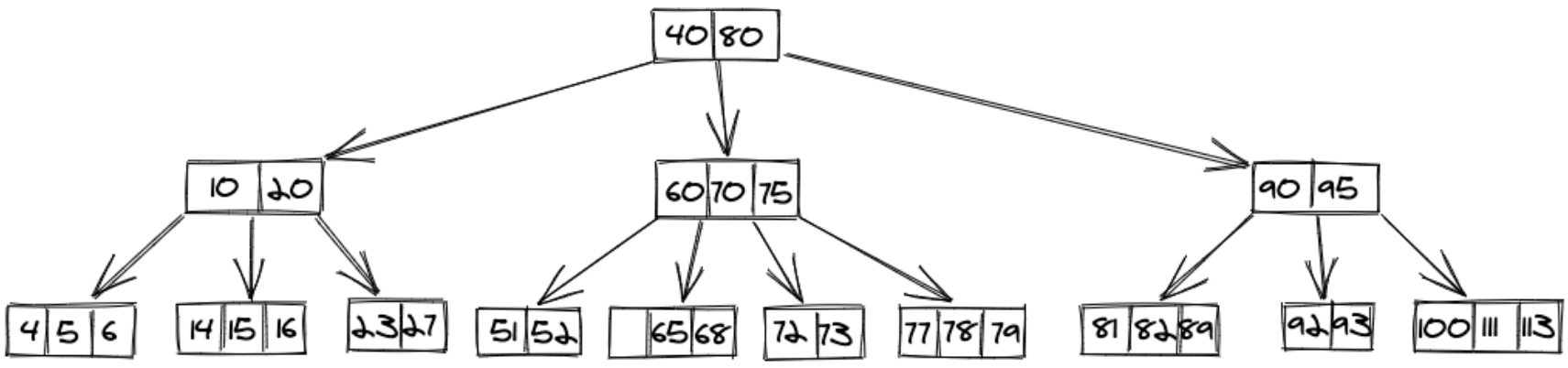
- If the node that contains the key that we want to delete, in turn **contains more than the minimum number of keys required** for the valid B tree, then we can simply delete that key.

Consider the pictorial representation shown below:



Say, we want to delete the key 64 and the node in which 64 is present, has more than minimum number of nodes required by the B tree, which is 2. So, we can simply delete this node.

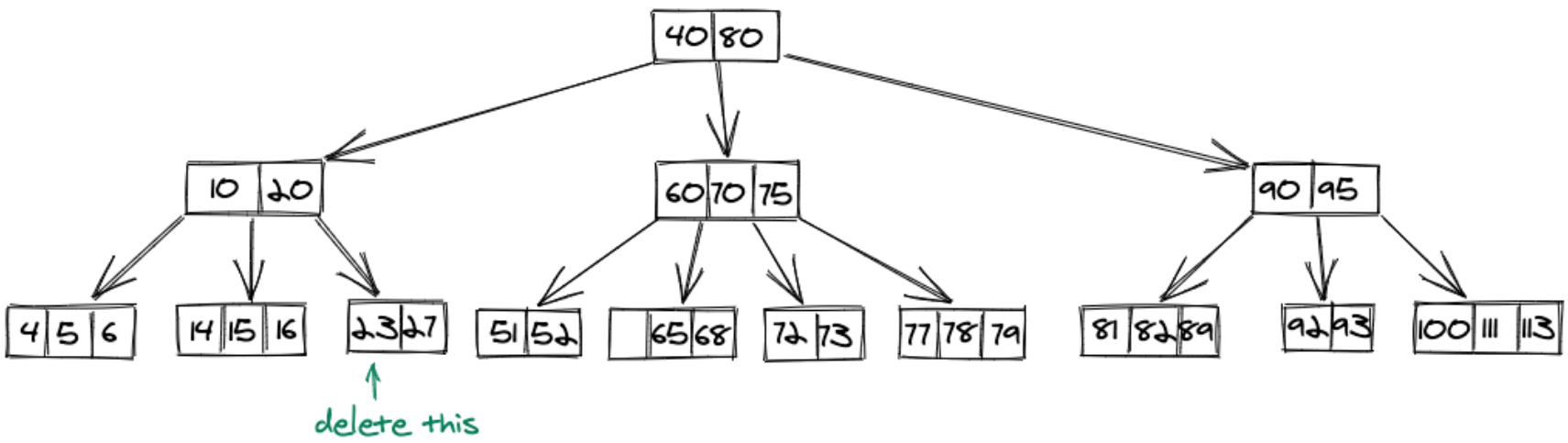
The final tree after deletion of 64 will look like this:



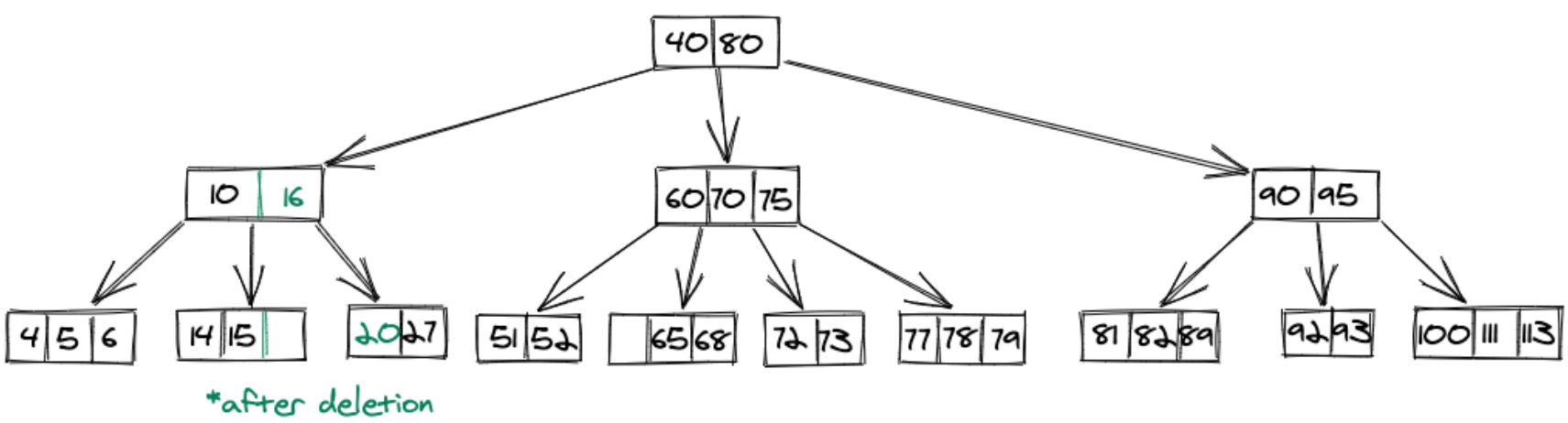
- If the node that contains the key that we want to delete, in turn **contains the minimum number of keys required** for the valid B tree, then three cases are possible:
  - In order to delete this key from the B Tree, we can borrow a key from the immediate left node(left sibling). The process is that we move the highest value key from the left sibling to the parent, and then the highest value parent key to the node from which we just deleted our key.
  - In another case, we might have to borrow a key from the immediate right node(right sibling). The process is that we move the lowest value key from the right sibling to the parent node, and then the highest value parent key to the node from which we just deleted our key.

- Last case would be that neither the left sibling or the right sibling are in a state to give the current node any value, so in this step we will do a merge with either one of them, and the merge will also include a key from the parent, and then we can delete that key from the node.

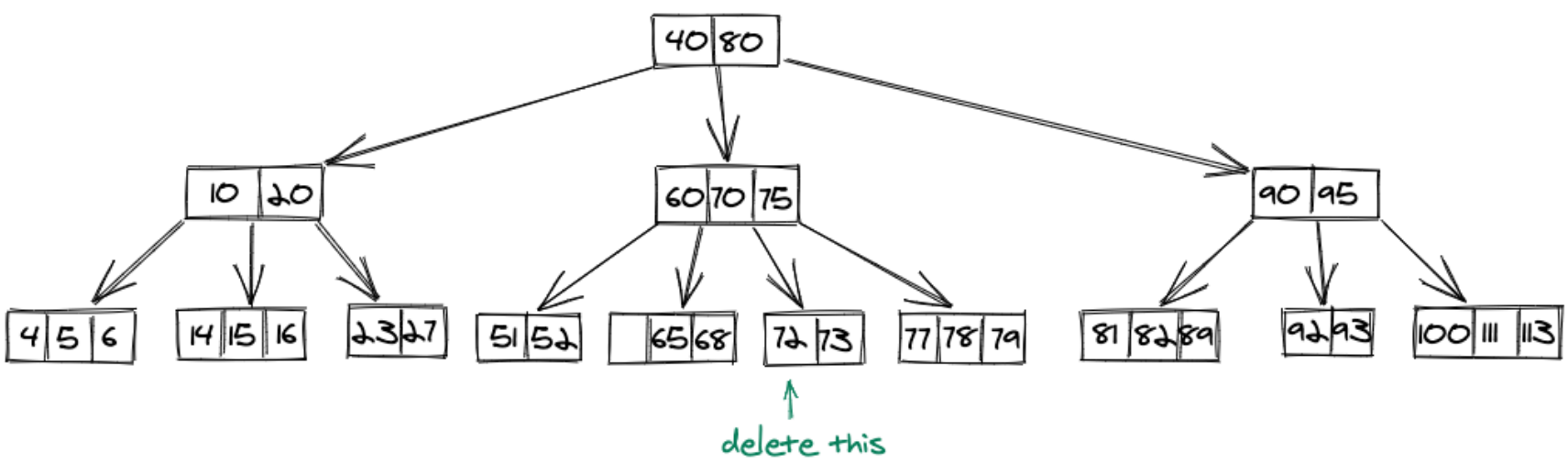
Case 1 pictorial representation:



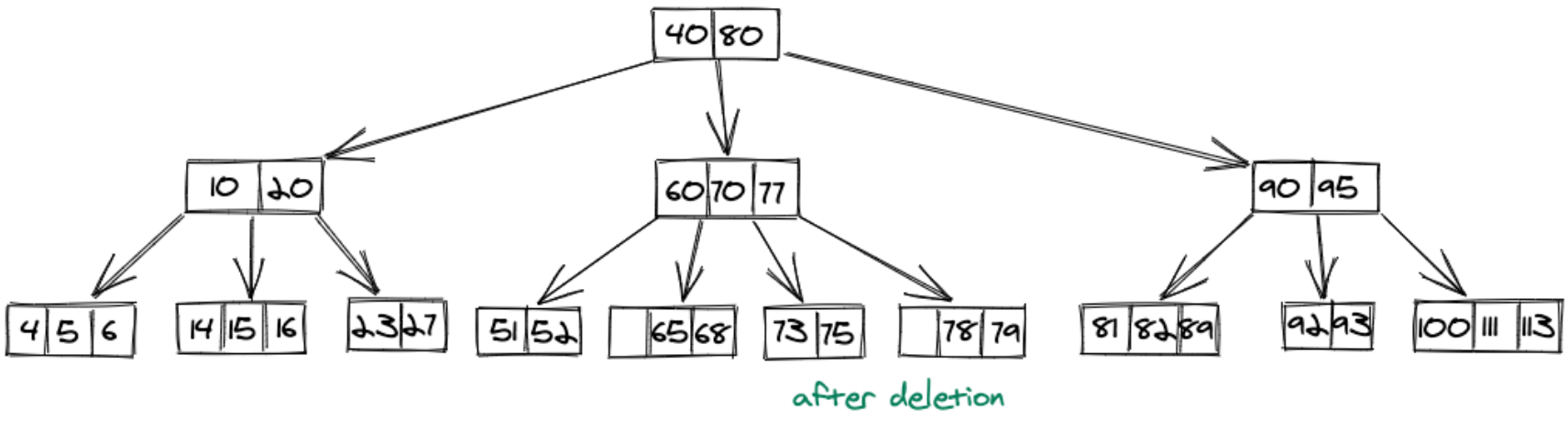
After we delete 23, we ask the left sibling, and then move 16 to the parent node and then push 20 downwards, and the resultant B tree is:



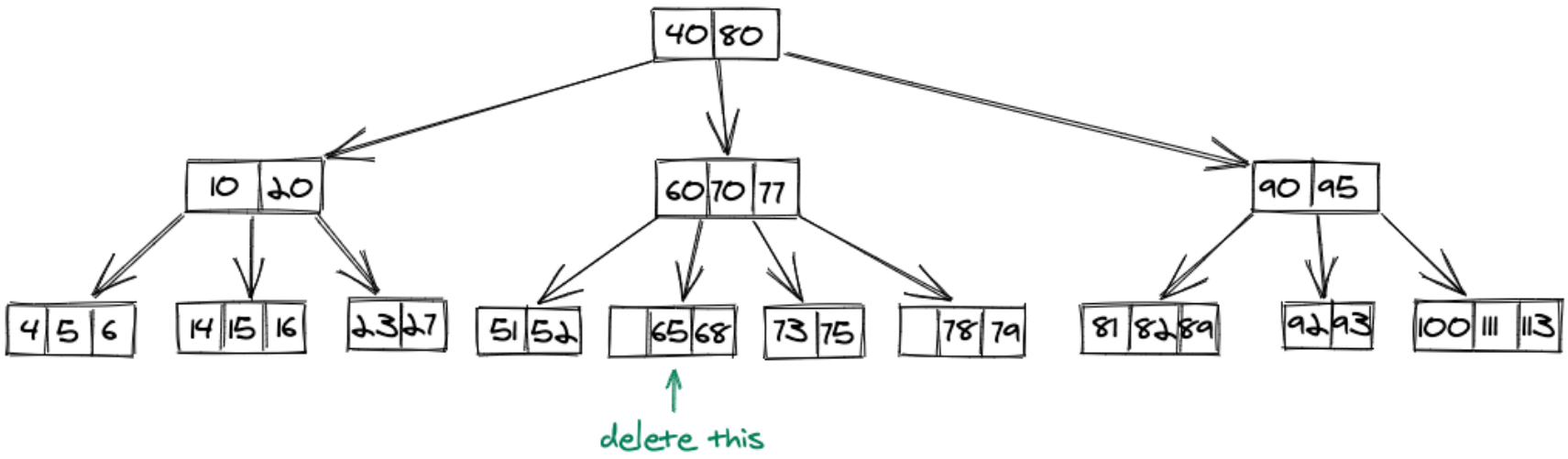
Case 2 pictorial representation:



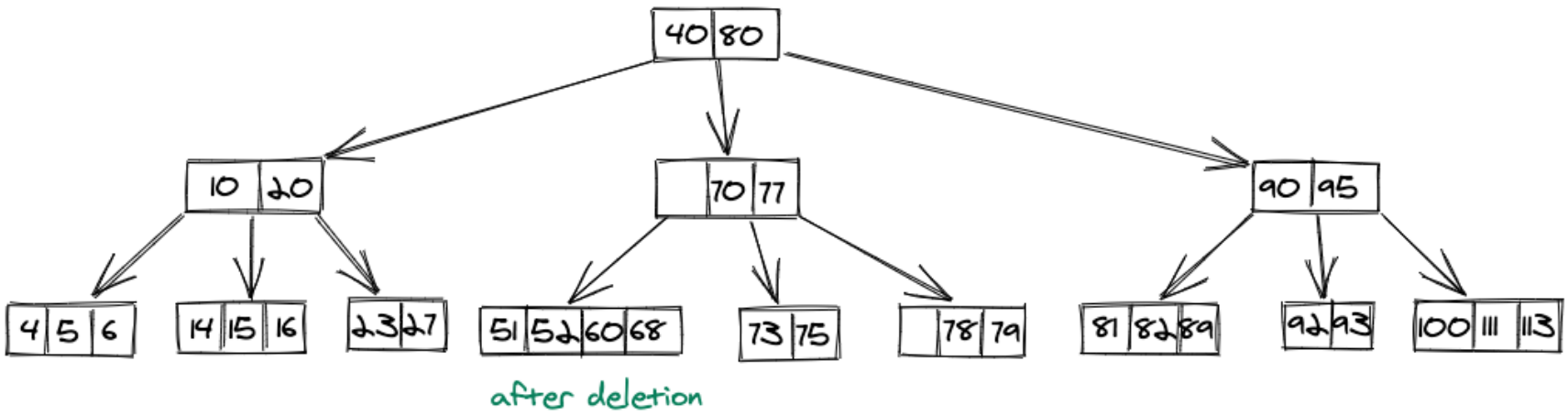
After we delete 72, we ask the right sibling, and then move the 77 to the parent node and then push the 75 downwards, and the resultant B tree is:



Case 3 pictorial representation:



After deleting 65 from the leaf node, we will have the final B tree as:

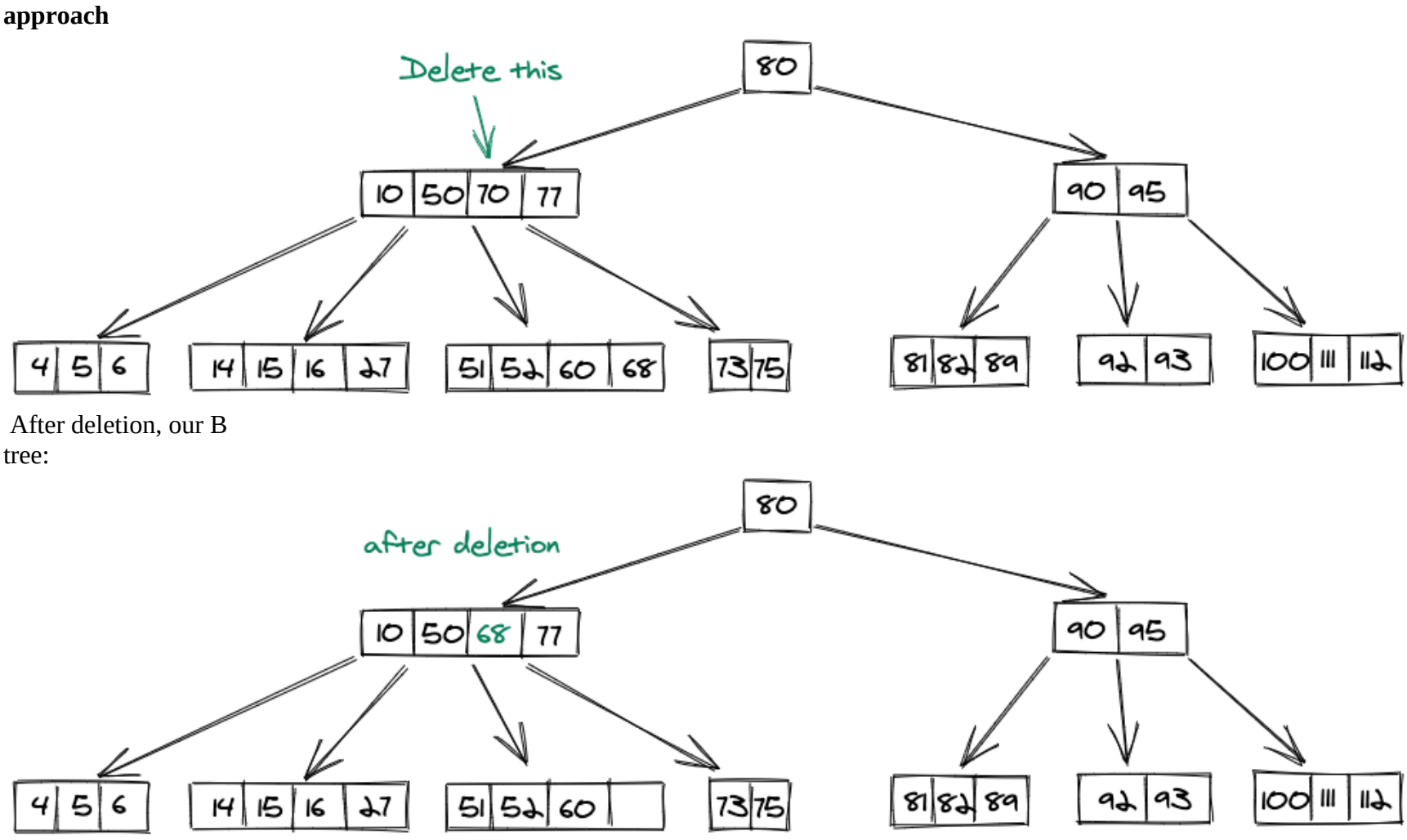


Deletion of Key from an Internal node:

- If we want to delete a key that is present in an internal node, then we can either take the value which is in order **predecessor** of this key or if taking that inorder predecessor violates the B tree property we can take the **inorder successor** of the key.
- In the inorder predecessor approach, we extract the highest value in the left children node of the node where our key is present.
- In the inorder successor approach, we extract the lowest value in the right children node of the node where our key is present.

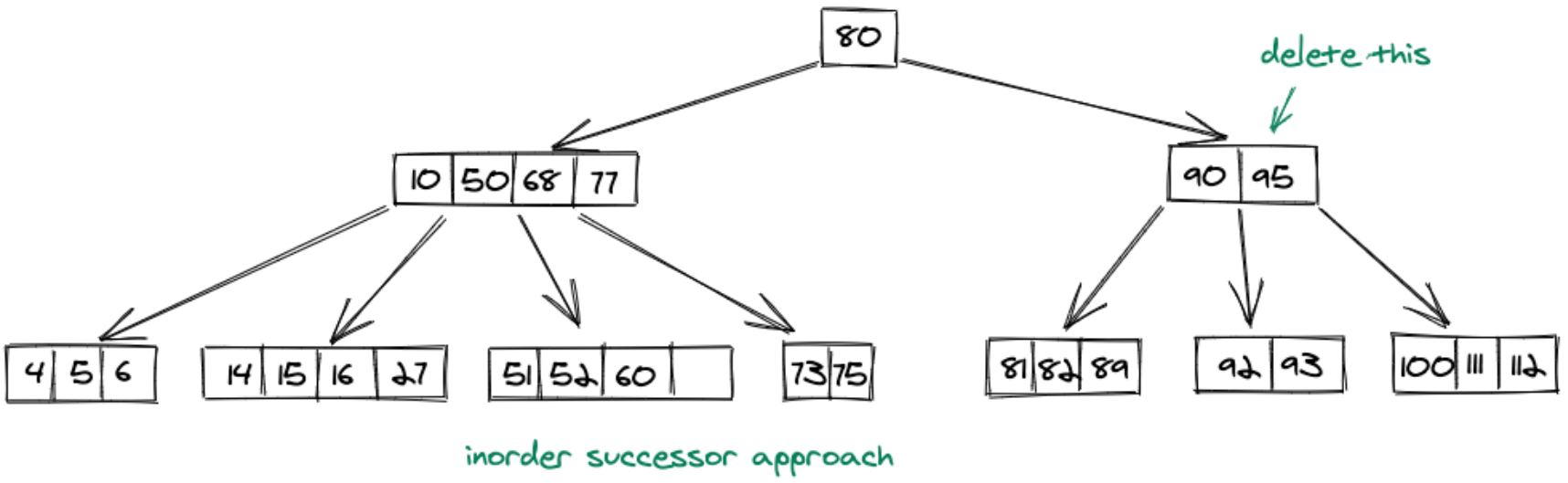
Pictorial Representation of the above cases:

- **Internal predecessor approach**

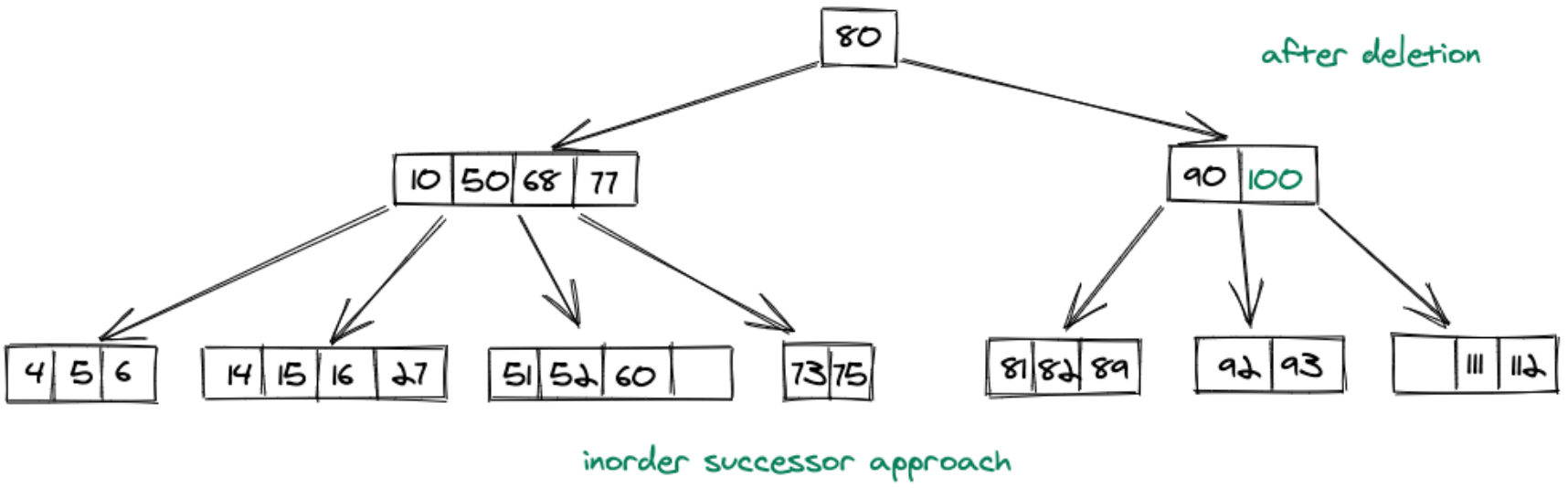


After deletion, our B tree:

- **Internal successor approach**



After deletion of 95, our tree will look like this:



Key Points:

- The time complexity for search, insert and delete operations in a B tree is **O(log n).**
- The minimum number of keys in a B tree should be **[M/2] - 1.**
- The maximum number of keys in a B tree should be **M-1.**
- All the leaf nodes in a B tree should be at the same level.
- All the keys in a node in a binary tree are in increasing order.
- B Trees are used in SQL to improve the efficiency of queries.
- Each node in a B Tree can have at most **M** children.

Conclusion:

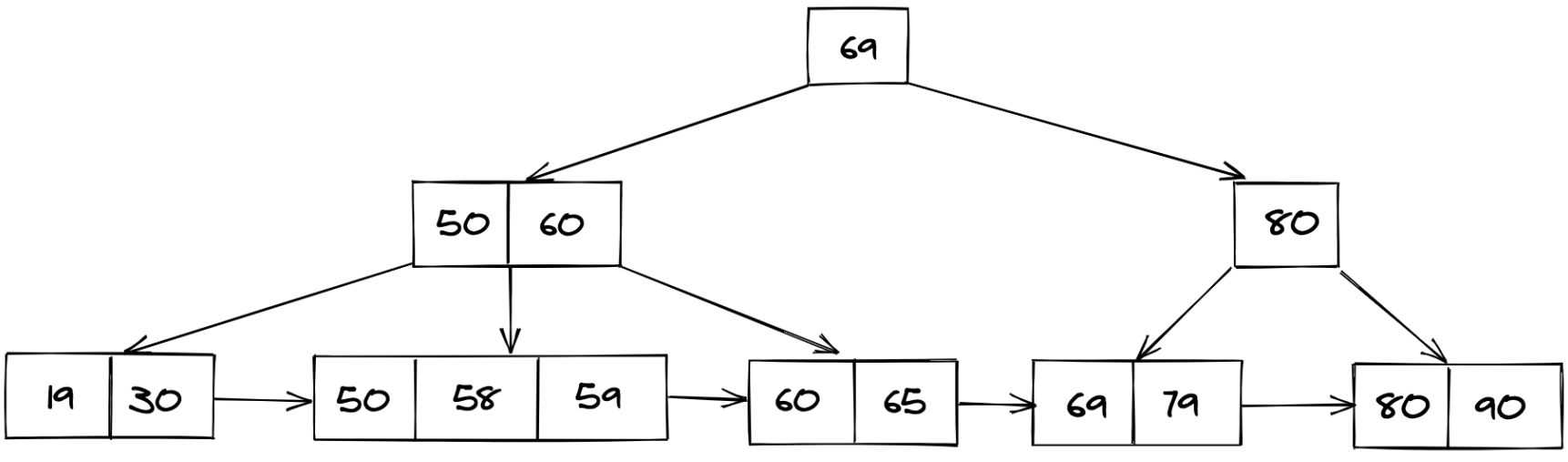
In this article, we learned what an M-way tree is, what are the differences between the M-way tree and M-way search tree. Also, we learned the constraints that are applied to an M-way tree to make it a B tree. Then we learned about the search, insert and delete operations.

B+ Trees Data Structure

A **B+ tree** is an extension of a B tree which makes the search, insert and delete operations more efficient. We know that B trees allow both the data pointers and the key values in internal nodes as well as leaf nodes, this certainly becomes a drawback for B trees as the ability to insert the nodes at a particular level is decreased thus increase the node levels in it, which is certainly of no good. B+ trees reduce this drawback by simply **storing the data pointers at the leaf node level** and only storing the key values in the internal nodes. It should also be noted that the nodes at the leaf level are linked with each other, hence making the traversal of the data pointers easy and more efficient.

B+ trees come in handy when we want to store **a large amount of data** in the main memory. Since we know that the size of the main memory is not that large, so make use of the B+ trees, whose internal nodes that store the key(to access the records) are stored in the main memory whereas, the leaf nodes that contain the data pointers are actually stored in the secondary memory.

A pictorial representation of a B+ tree is given below:



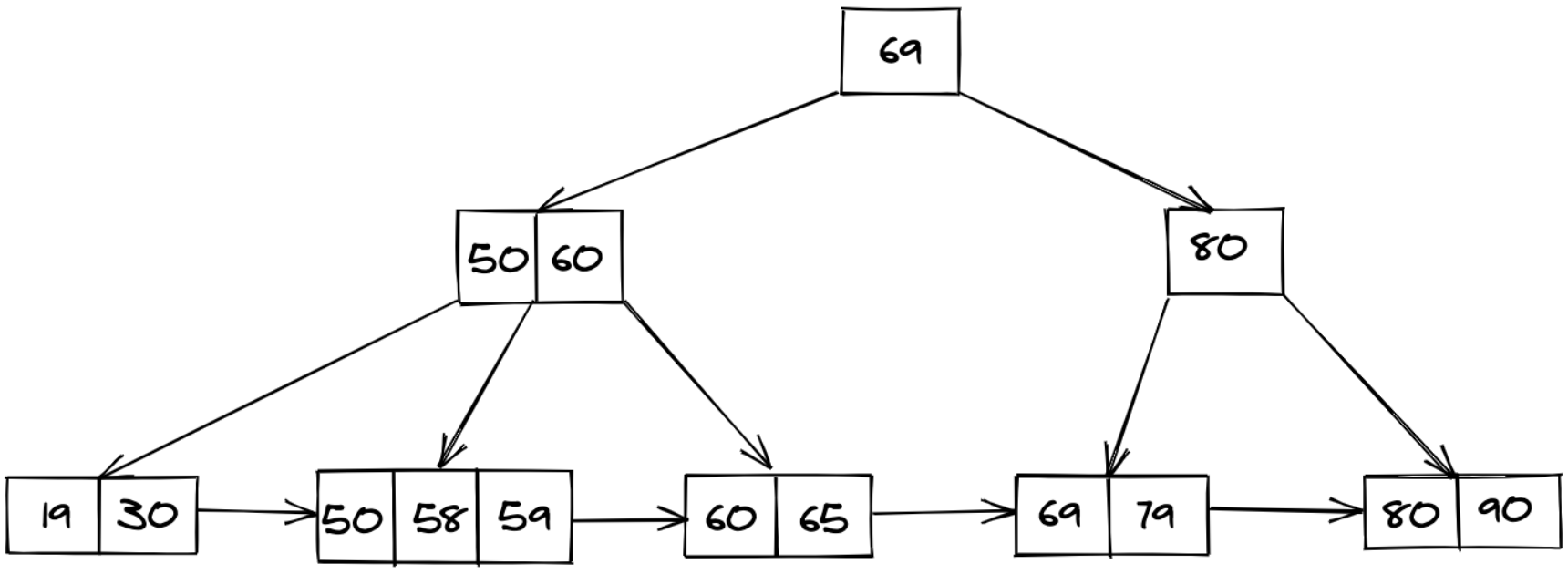
Why B+ trees?

- B+ trees store the records which later can be fetched in an equal number of disk accesses.
- The height of the B+ tree **remains balanced** and very less as when compared to B trees even if the number of records store in them is the same.
- Having less number of levels makes the accessing of records very easy.
- As the leaf nodes are connected with each other **like a linked list**, we can easily search elements in sequential manners.

Inserting in B+ tree

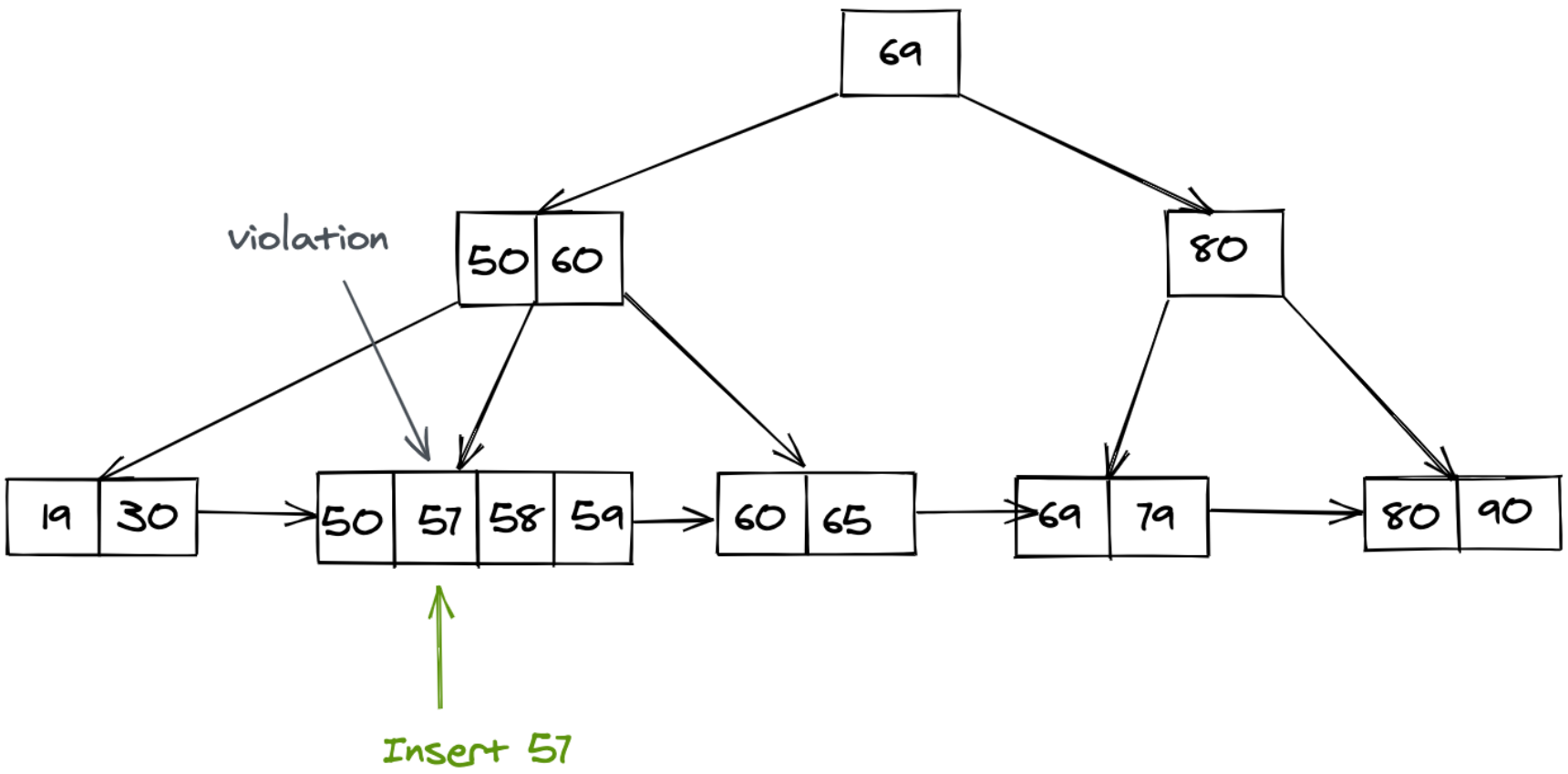
- Perform a search operation in the B+ tree to check the **ideal bucket location** where this new node should go to.
- If the bucket is not full(does not violate the B+ tree property), then add that node into this bucket.
- Otherwise split the nodes into two nodes and push the middle node(median node to be precise) to the parent node and then insert the new node.
- Repeat the above steps if the parent node is there and the current node keeps getting full.

Consider the pictorial representations shown below to understand the Insertion operation in the B+ tree:

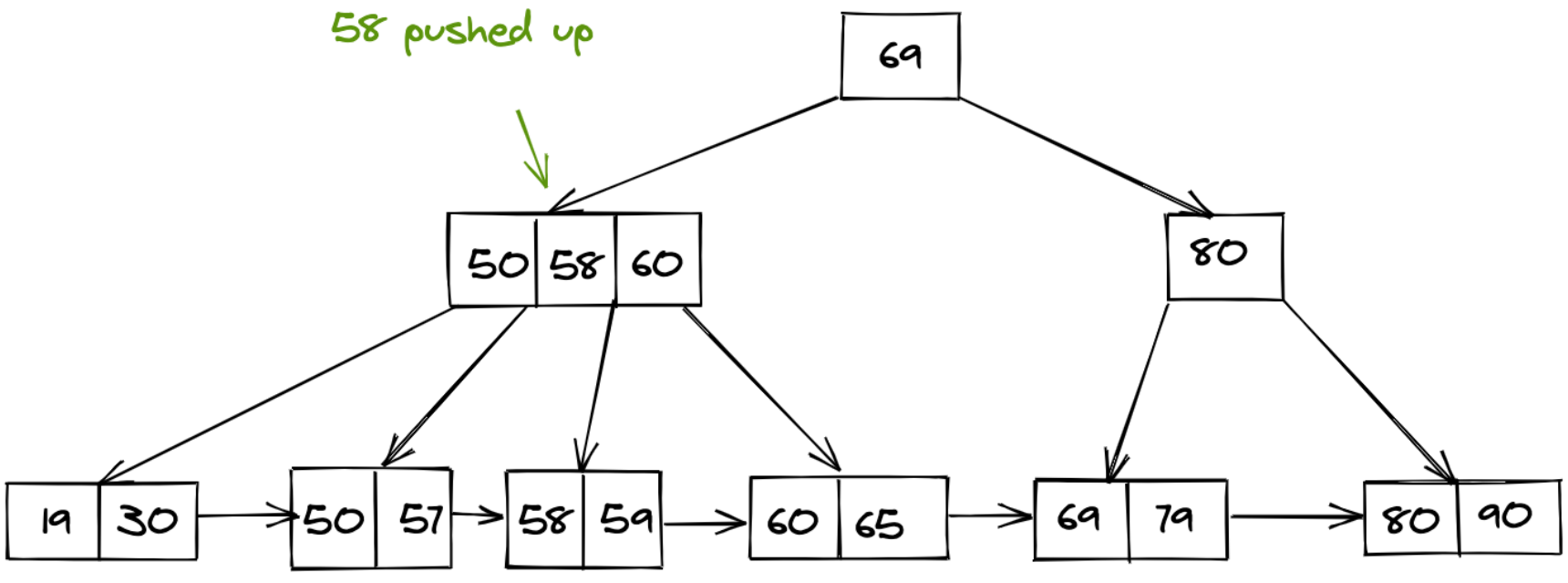


B+ tree of order 4

Let us try to insert 57 inside the above-shown B+ tree, the resultant B+ tree will look like this:



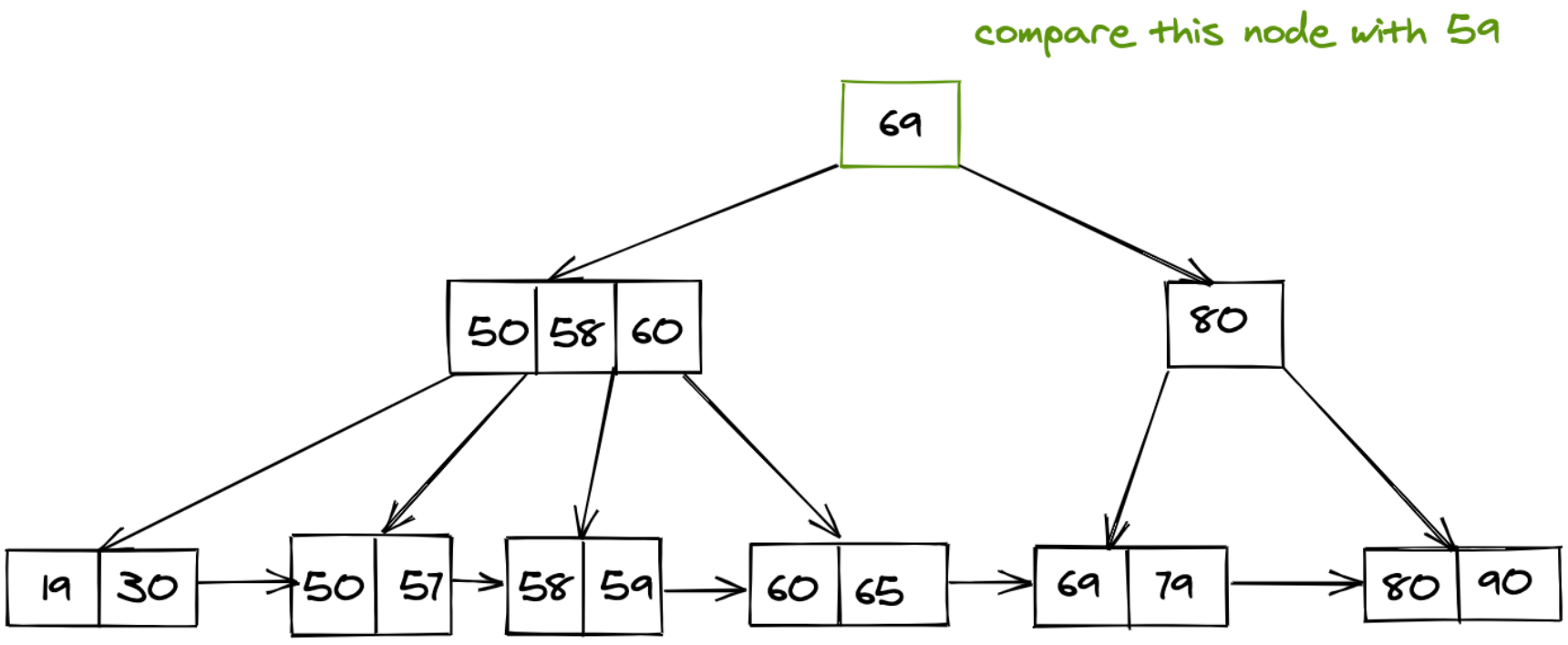
We know that the bucket(node) where we inserted the key with value 57 is now violating the property of the B+ tree, hence we need to split this node as mentioned in the steps above. After splitting we will push the median node to the parent node, and the resulting B+ tree will look like this:



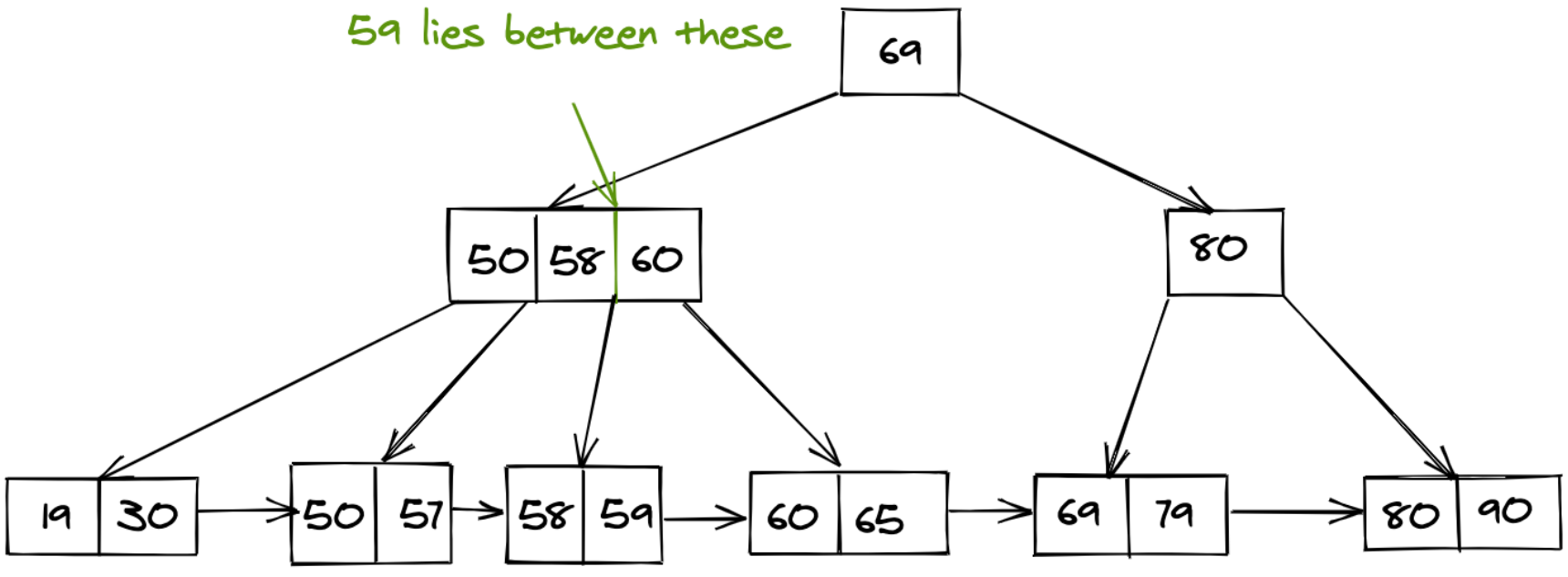
Searching in B+ tree:

Searching in a B+ tree is similar to searching in a BST. If the current value is less than the searching key, then traverse the left subtree, and if greater than first traverse this current bucket(node) and then check where the ideal location is.

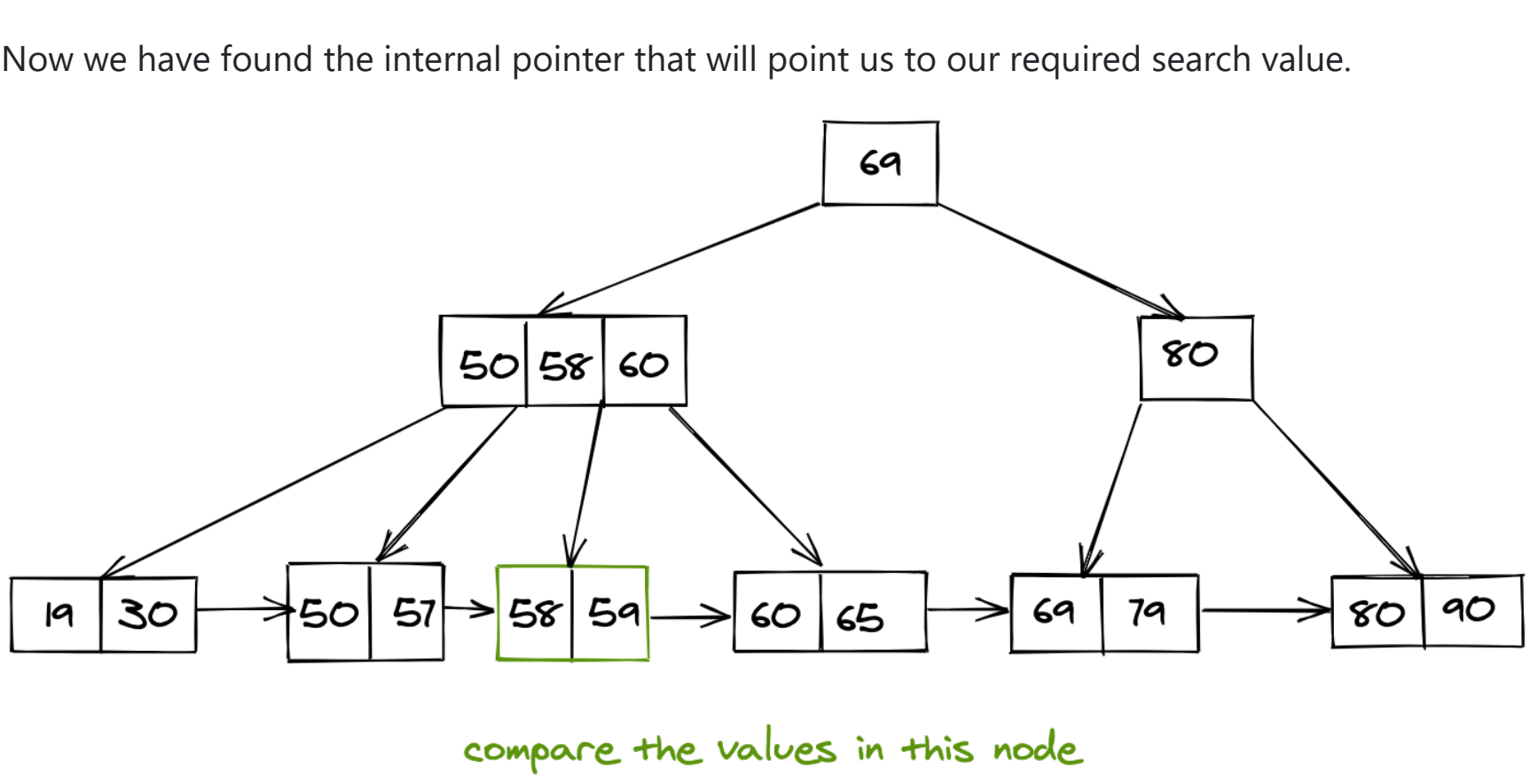
Consider the below representation of a B+ tree to understand the searching procedure. Suppose we want to search for a key with a value equal to 59 in the below given B+ tree.



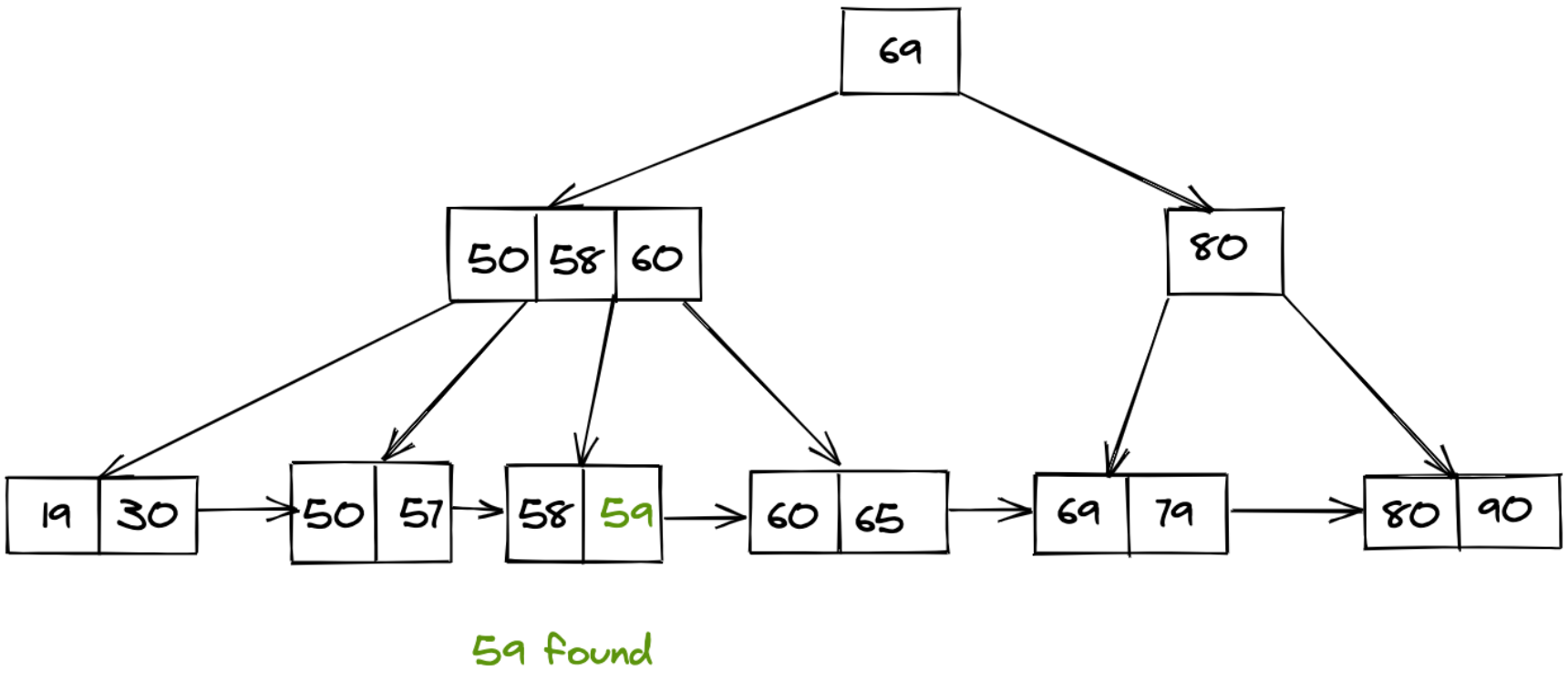
Now we know that 59 < 69, hence we traverse the left subtree.



Now we have found the internal pointer that will point us to our required search value.



Finally, we traverse this bucket in a linear fashion to get to our required search value.



Deletion in B+ tree:

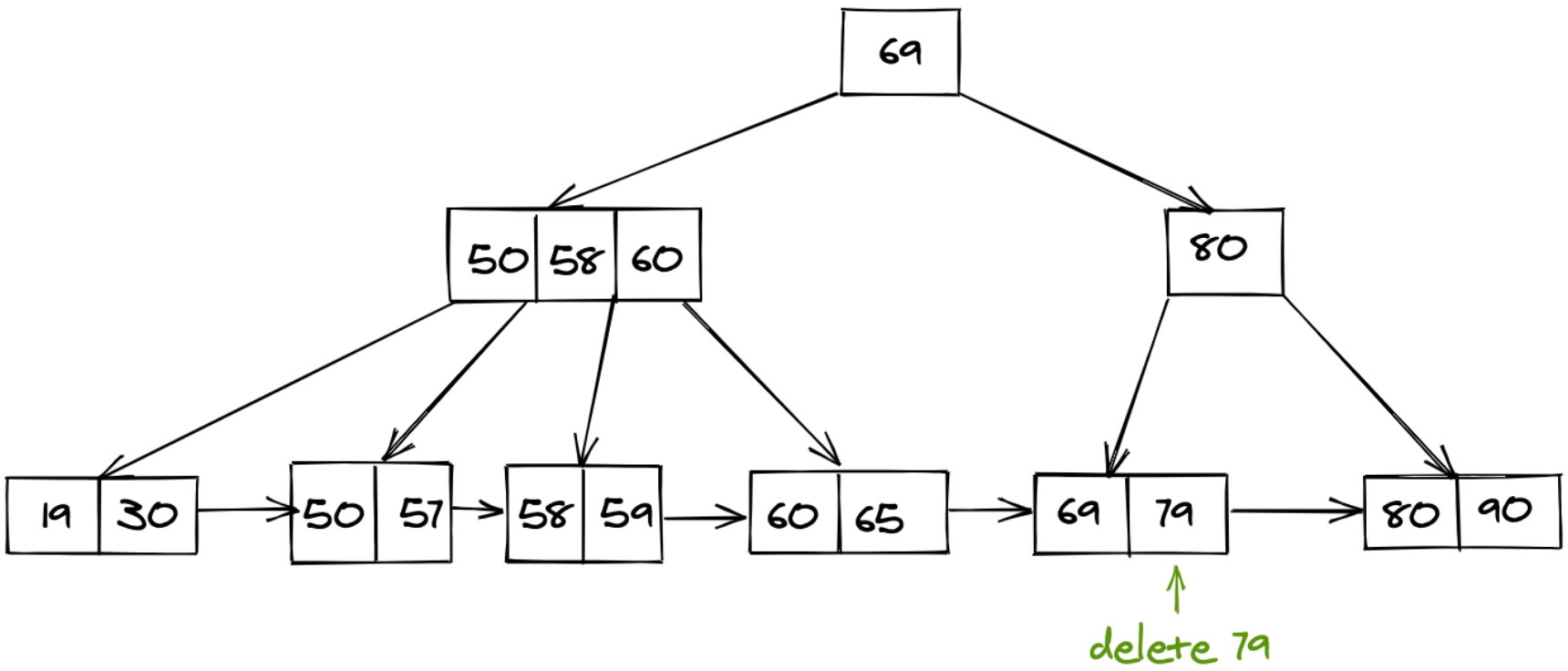
Deletion is a bit complicated process, two case arises:

- It is only present at the **leaf level**
- or, it contains a pointer from an internal node also.

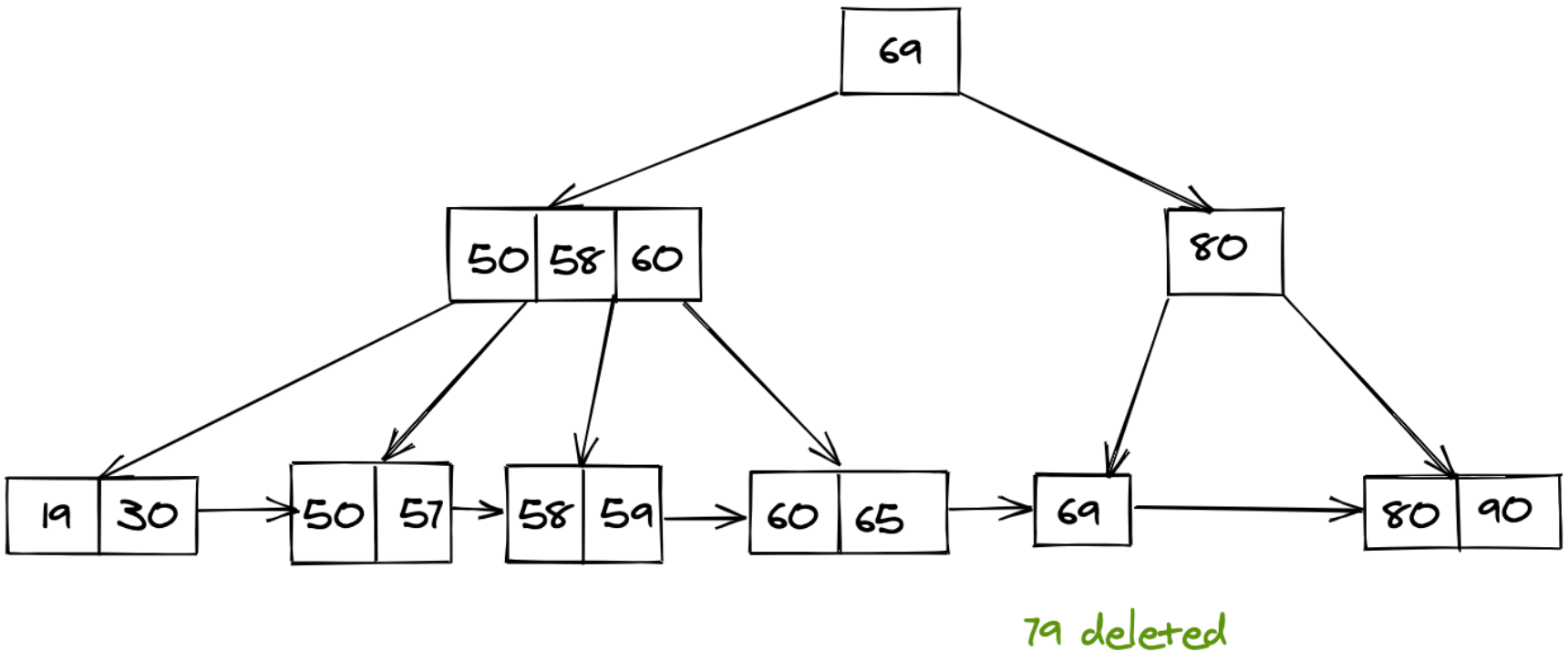
Deletion of only the leaf-node:

If it is present only as a **leaf node position**, then we can simply delete it, for which we first do the search operation and then delete it.

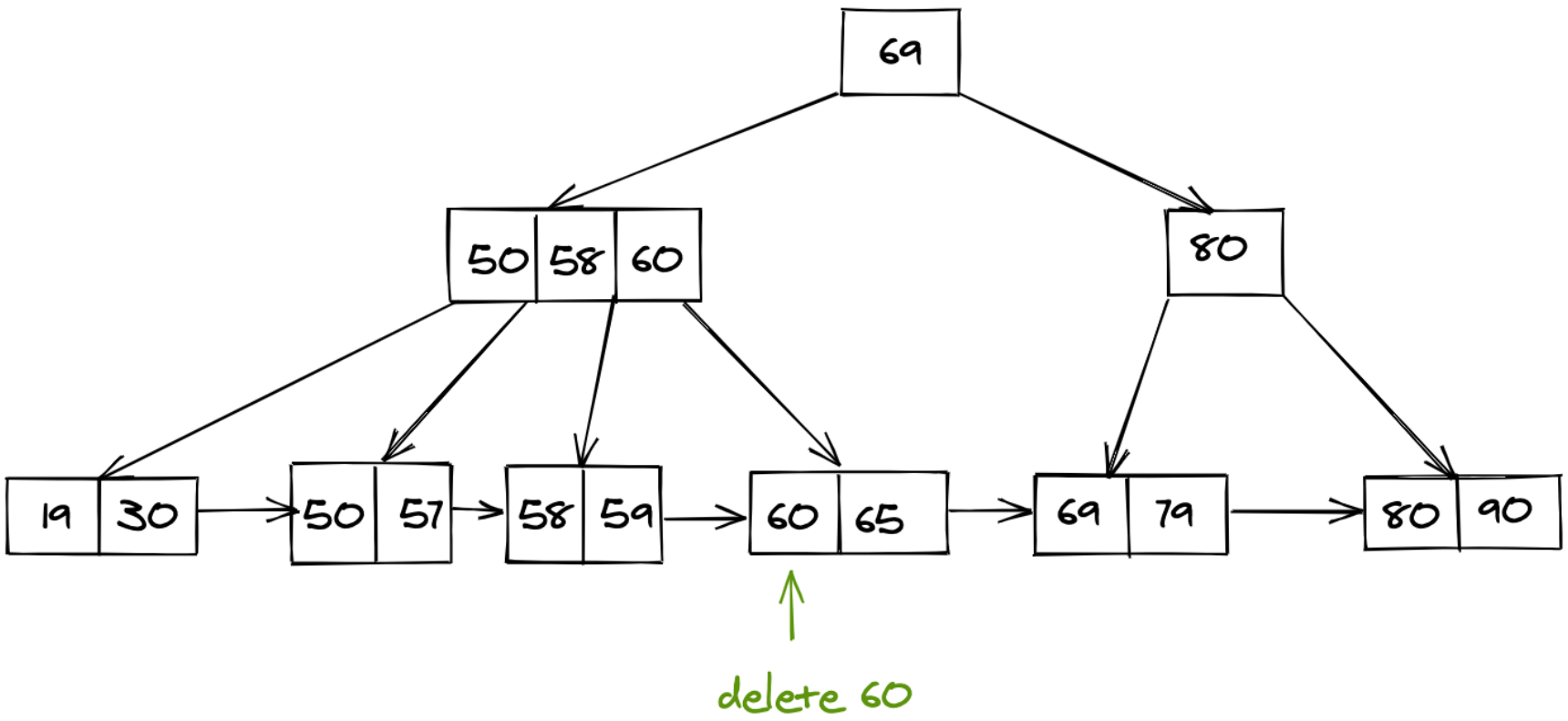
Consider the pictorial representation shown below:



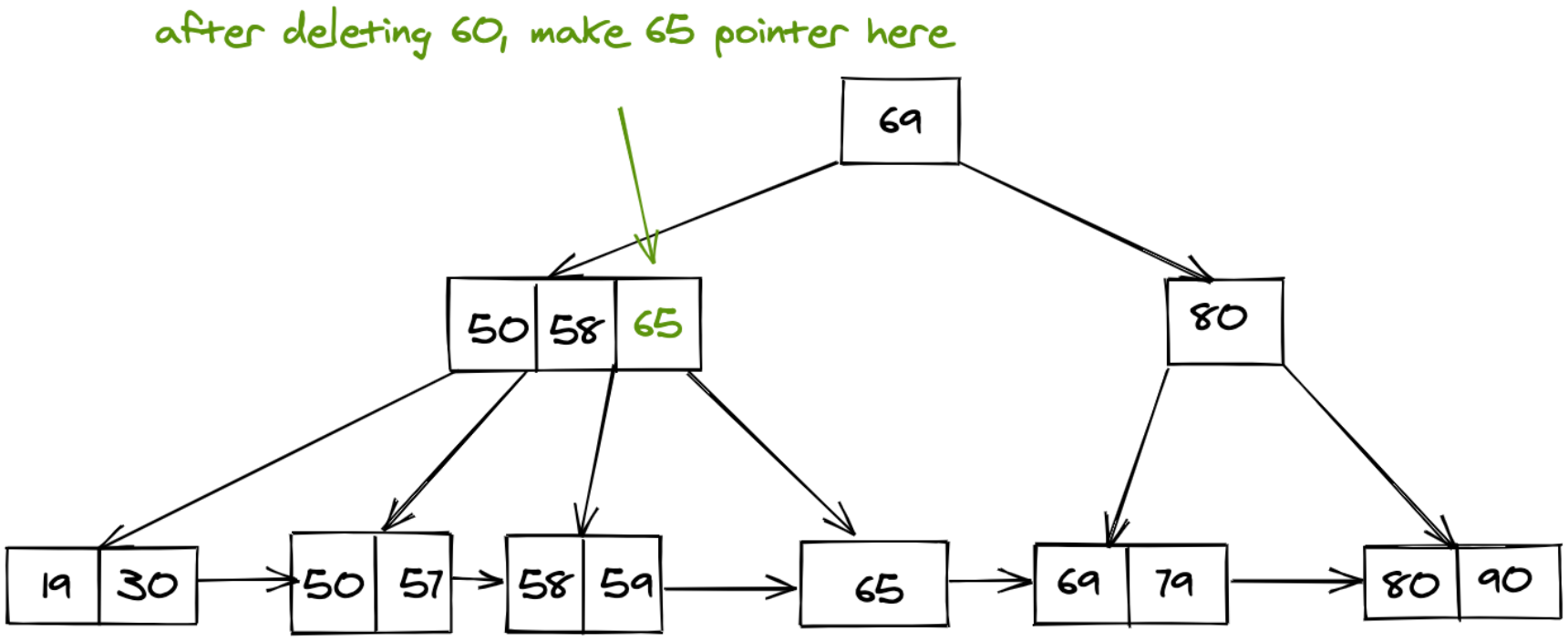
After the deletion of 79, we are left with the following B+ tree.



Deletion if a pointer to a leaf node is there:



After we locate the node we want to delete we must also delete the internal pointer that points to this node, and then we finally need to move the next node pointer to move to the parent node.



Conclusion:

- We learned what a B+ tree is, and how it is different from a B tree.

- Then we learned why we need a B+ tree.
- Lastly, we learned different operations that we do on a B+ tree-like searching, Insertion, and deletion of a record(key).