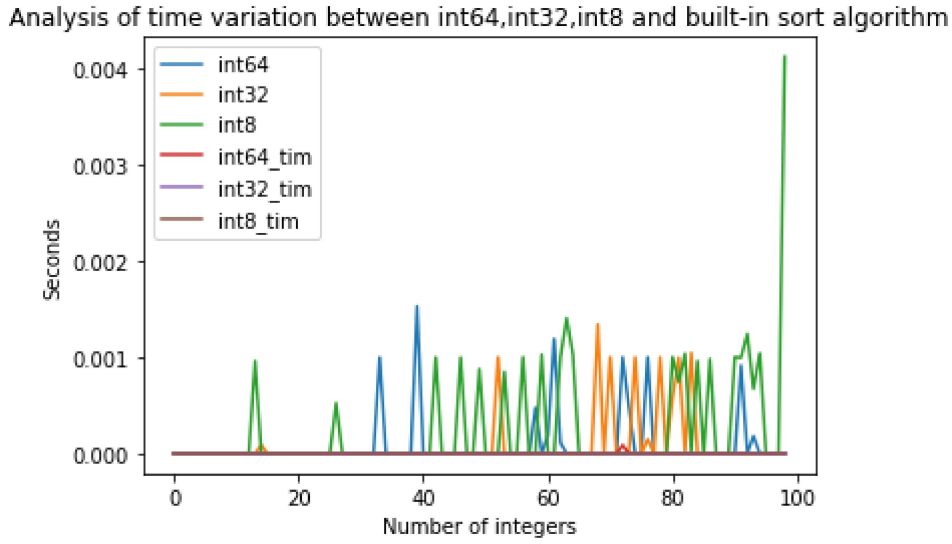
EX1: In this exercise I implemented Bucket sort algorithm. It is a distribution sort that works by arranging elements into several 'buckets' which are then sorted using another sort(I used insertion sort) and merged into a sorted list. To compare bucket sort algorithm I chose the python's built-in algorithm Timsort.

In [1]:
```
import random, sys, time, numpy as np, math
import matplotlib.pyplot as plott
```

In [2]:
```
def bucketSort(arr):
    largest = max(arr)
    length = len(arr)
    size = largest/length

    buckets = [[] for _ in range(length)]
    for i in range(length):
        j = int(arr[i]/size)
        if j != length:
            buckets[j].append(arr[i])
        else:
            buckets[length - 1].append(arr[i])

    for i in range(length):
        insertion_sort(buckets[i])

    result = []
    for i in range(length):
        result = result + buckets[i]
#     print(result)
    return result

def insertion_sort(arr):
    for i in range(1, len(arr)):
        temp = arr[i]
        j = i - 1
        while (j >= 0 and temp < arr[j]):
            arr[j + 1] = arr[j]
            j = j - 1
        arr[j + 1] = temp
```

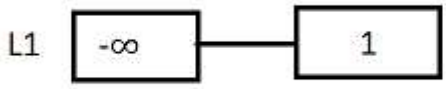
In [3]:
```
def measure_time(arr):
    startTime = time.time()
    bucketSort(arr.copy())

    duration = time.time() - startTime
    return duration
def measure_time_tim(arr):
    arr1 = arr.copy()
    startTime = time.time()
    arr1.sort()
    duration = time.time() - startTime
#     print(arr1)
    return duration
def increase():
    measures_arr_int64 = []
    measures_arr_int32 = []
    measures_arr_int8 = []
    measures_arr_int64_tim = []
    measures_arr_int32_tim = []
    measures_arr_int8_tim = []
    f_x = []
    for i in range(1,100):
        arr_int64 = np.array(np.random.randint(0,100,i, dtype=np.int64))
        arr_int32 = arr_int64.astype(np.int32).tolist()
        arr_int8 = arr_int64.astype(np.int8).tolist()
        arr_int64 = arr_int64.tolist()
        #int64
        duration = measure_time(arr_int64)
        measures_arr_int64.append(duration)

        duration = measure_time_tim(arr_int64)
        measures_arr_int64_tim.append(duration)
        #int32
        duration = measure_time(arr_int32)
        measures_arr_int32.append(duration)

        duration = measure_time_tim(arr_int32)
        measures_arr_int32_tim.append(duration)
        #int8
        duration = measure_time(arr_int8)
        measures_arr_int8.append(duration)

        duration = measure_time_tim(arr_int8)
        measures_arr_int8_tim.append(duration)



    plott.title("Analysis of time variation between int64,int32,int8 and built
-in sort algorithm")
    plott.ylabel('Seconds')
    plott.xlabel('Number of integers')

    plott.plot(measures_arr_int64, label='int64')
    plott.plot(measures_arr_int32, label='int32')
    plott.plot(measures_arr_int8, label='int8')
    plott.plot(measures_arr_int64_tim, label='int64_tim')
```

```
    plott.plot(measures_arr_int32_tim, label='int32_tim')
    plott.plot(measures_arr_int8_tim, label='int8_tim')

    plott.legend()
    plott.show()

increase()
```

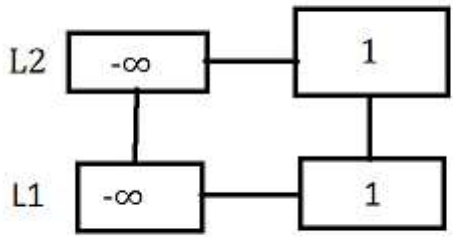


We can observe from the graph that with increasing the number of integers in array, bucket sort starts to sort slower than the built-in sorting algorithm - Timsort and this tendency can be observed with all types of integers(int64,int32,int8). Generally it is slower 200-300 times with big-sized arrays. I came to this conclusion by dividing duration of bucket sort algorithm in each occassion to 200 and 300 and somewhere between this numbers the graphs are similar. On the other hand among the int types the most slow one is int64. The second one is int32 and the fastest sorting int type is int8. After this experiment we can make a conclusion that integers with smallest number of bytes are faster than integers with big amount of bytes.

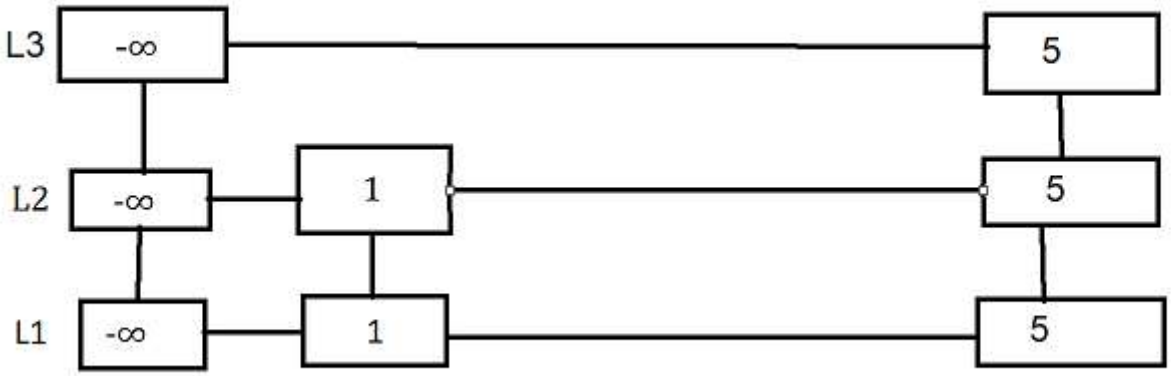
EX2: To visualize the skip-list insertion I chose the simple software visualization tool - Paint. In the first step I created the first level of skip list with the smallest element of minus infinity(-∞) and then put the first number(1) into the list.



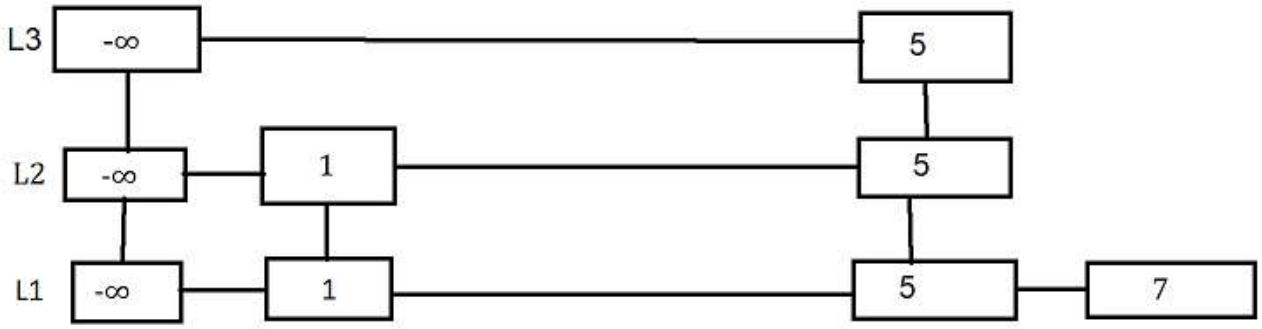
As the structure of skip list is randomized in this task we were obliged to use coin to simulate the randomization. I chose the simple 20 cent coin for it, where heads meant to move to upper level, and tails - to stop. So first flipping ended up with heads, I moved to the upper level, actually created it and flipped the coin for the second time and got the heads, so I stopped.



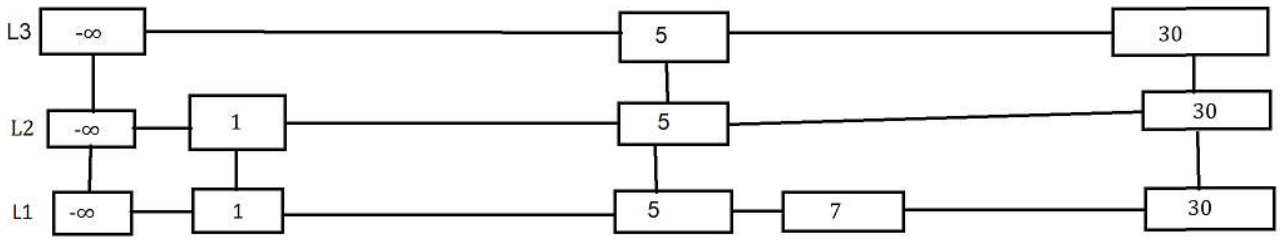
The actual process of insertion when it is processed in machine or computer goes from upper level and searches by going everytime to the bottom level, for example if we have to insert 5, we should search its place from the top level by comparing it with values on each level,for example, when 5 is less than the next value on the current level we move to the bottom level, compare 5 to the values here and find its place among the values. As I described the actual process of insertion,in the next steps I will just insert the number to the bottom level without giving the description of a whole process.



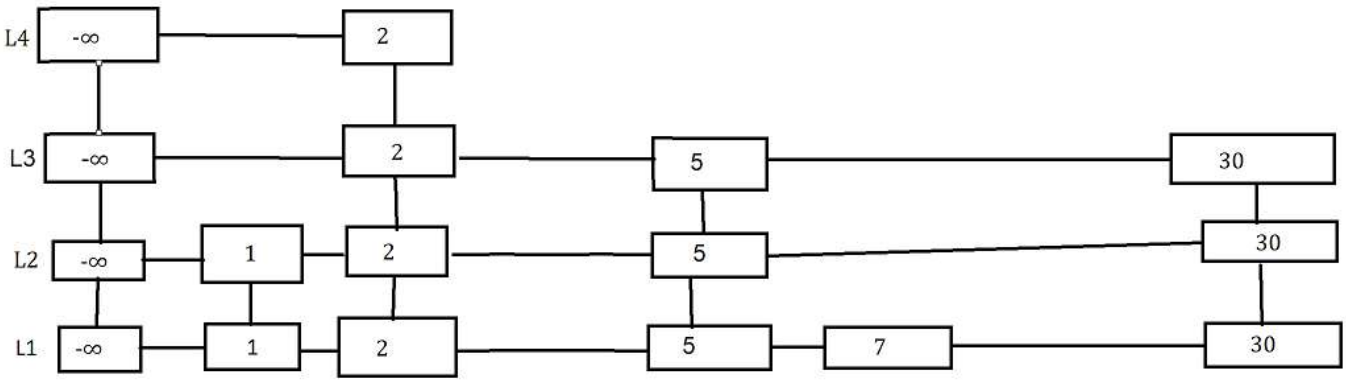
I've inserted 5 and coin showed the tails two time and the flipped to the heads, so now we have 3 levels with value 5 in each of them.



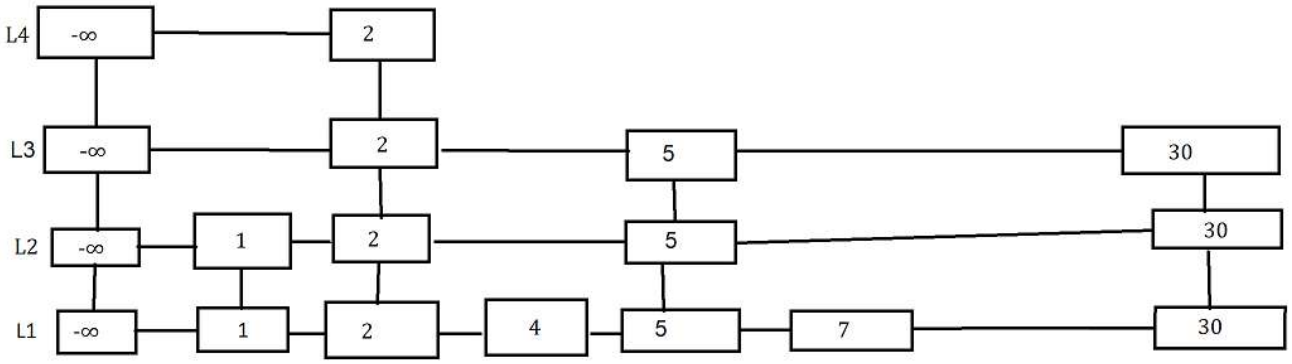
On the next step I inserted 7 to our list and by flipping the coin I've got heads from the very first try, so I stopped on the bottom level. How many steps it took for us to insert 7? Firstly on the 3rd level we compared our value with 5, and 7 was bigger than it. As 5 was the biggest number in our list we come to the level1 and insert it right after 5. Totally we have 4 steps with insertion itself.

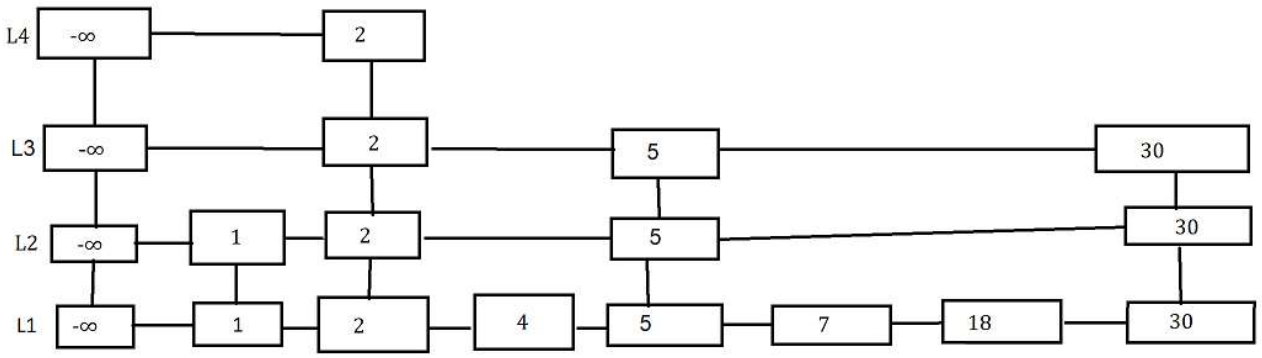


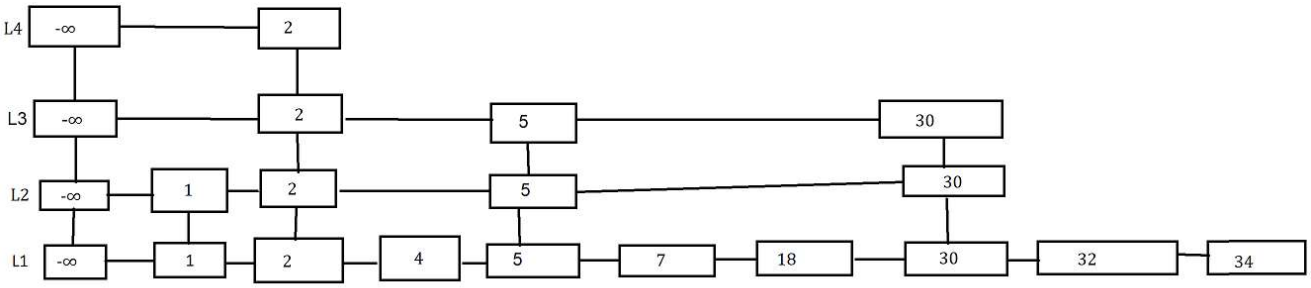
The fourth value had the same actions as 5, so now we have 5 and 30 on the third level.



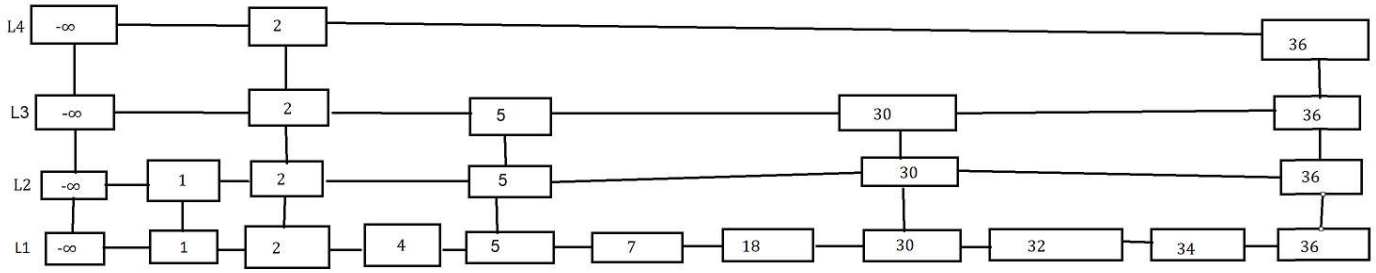
As I described the process of insertion we are searching the place of new element from the top level and for value 2 we found its place right after 1, then coin showed the tails 3 times and then heads and we stopped by creating the level 4 as I showed in the picture.







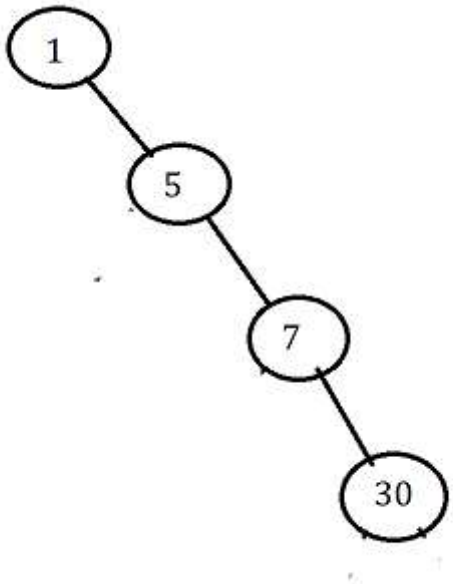
After insertion of 4,18,32,34 the the coin showed heads and now we have these values only on the bottom level.



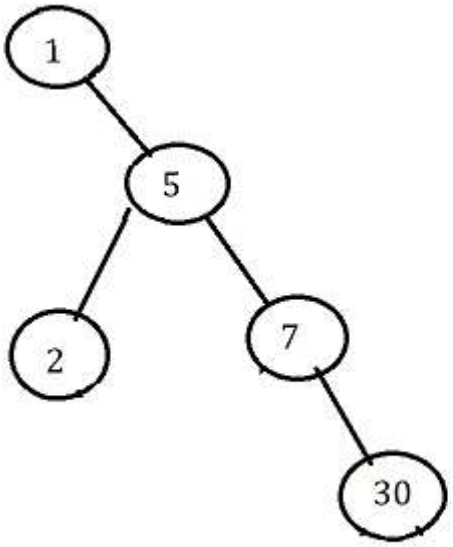
Insertion of 36 and coin reacts was the same as with 2, so now on the 4th level we have minus infinity,2 and 36.

To find a vale we start as with insertion from top level. 35 is more than 2, but less than 36, so we go down to the 3rd level. Here we compare it with 5 and 30 and in both occasions 35 is more than these values, but again less than 36, so we are moving to the second level. Here again the next value is 36 and we repeat our moving down and now we are on the bottom level. 35 is more than 32 and 34, but the next value is again 36. As we are on the bottom level it means that we couldn't manage to find our wished value. We performed 9 comparisons before figuring out that we don't have 35 in our skip list.

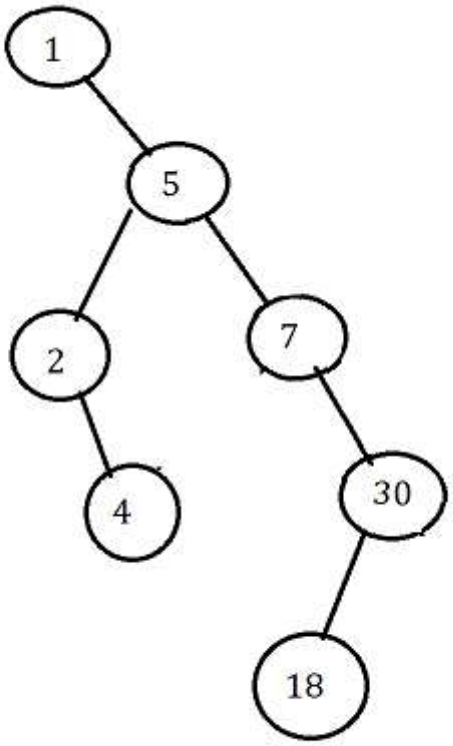
EX3: In this exercise we should insert the same values into binary search tree in the same manner. To visualise it I used the same software. The first 4 steps are skewed to the right. 1 is root. 5>1,7>5 and 30>7.



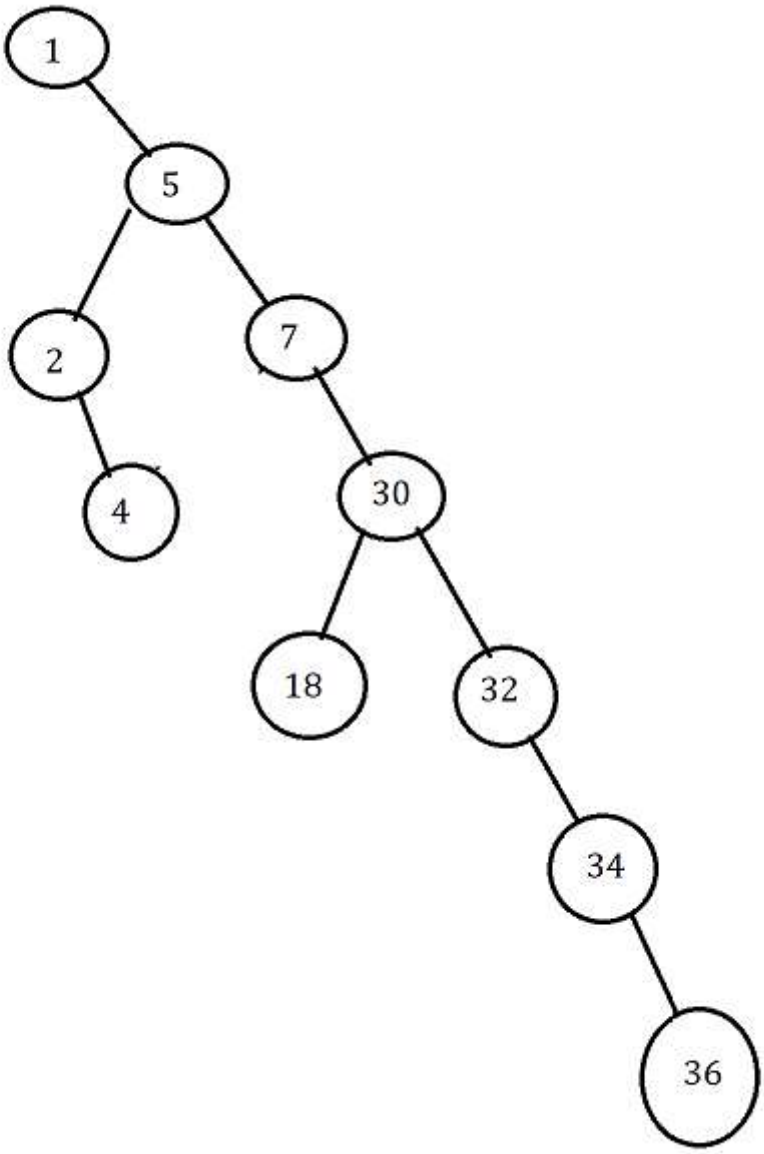
Then we should add '2' to the tree and now we start to compare. 2 is bigger than 1, so we go to the right and less than 5, so we go to the left.



Then we should insert 4 and 18. 4 is bigger than 1, less than 5 and bigger than 2, so we build a new branch to the right of 2. 18 is bigger than 1,5,7 but less than 30, so we build a new branch for 18 to the left of 30.

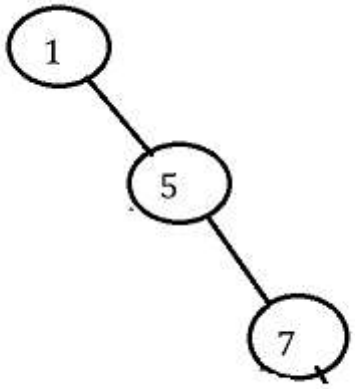


The last numbers are all bigger than what we had in a tree and among each other the next one is bigger than previous one, so we end up with tree like that.

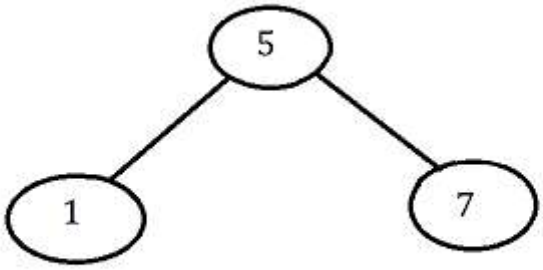


As we can observe from the picture of out final tree, it is not complete BST and it is skewed to the right, but also cannot be considered as the right skewed binary tree because some of the nodes has 2 children.

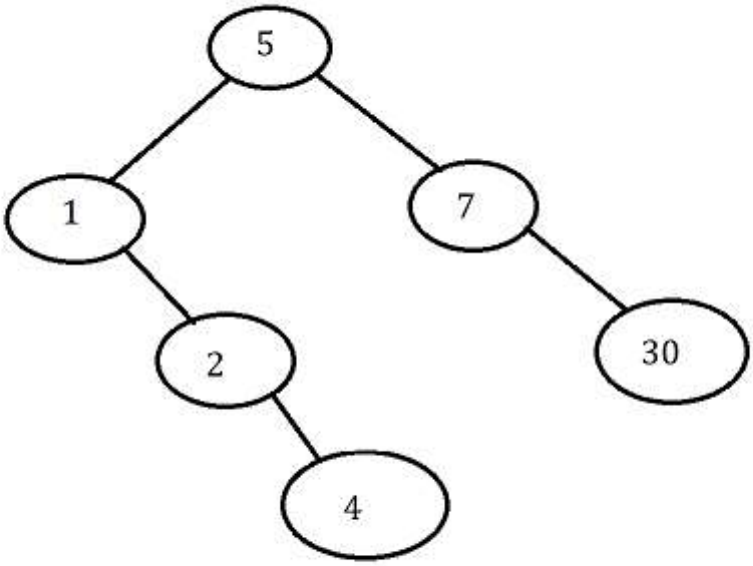
EX4: The general definition of AVL tree: it is a self-balancing Binary Search Tree where the difference between heights of left and right subtrees cannot be more than one for all nodes. So it means that when the height of 2 children differs more than 1, then the tree should be balanced. So, firstly when we insert the elements to the tree we have this structure.

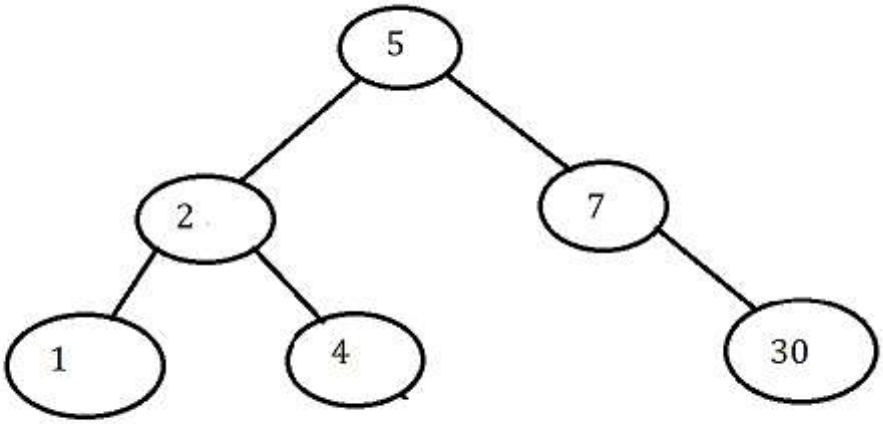


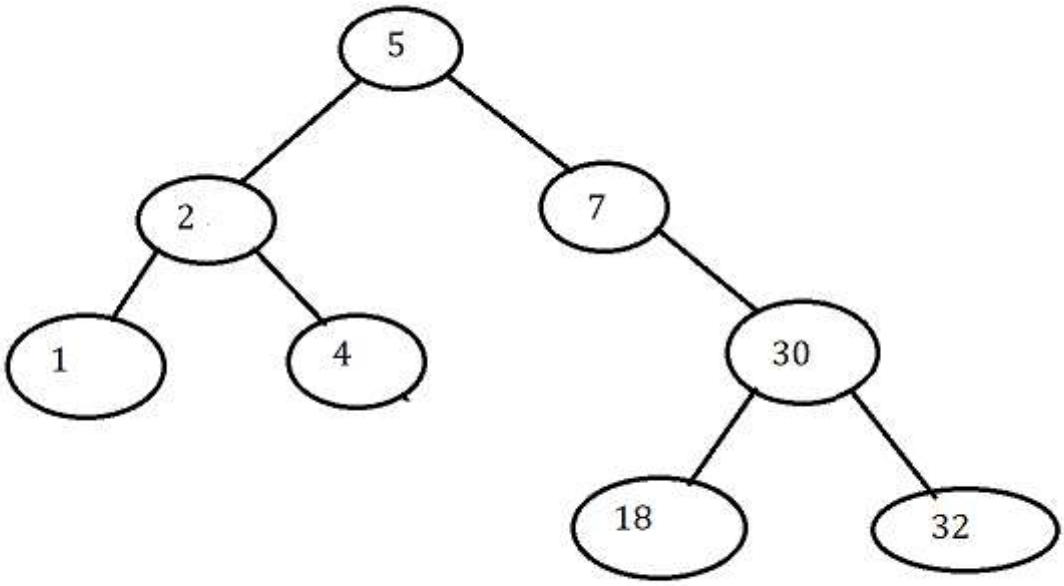
It contradicts to the defintion of AVL tree, because on the right we already have 2 child nodes and none on the leftm so we re-balance the tree.

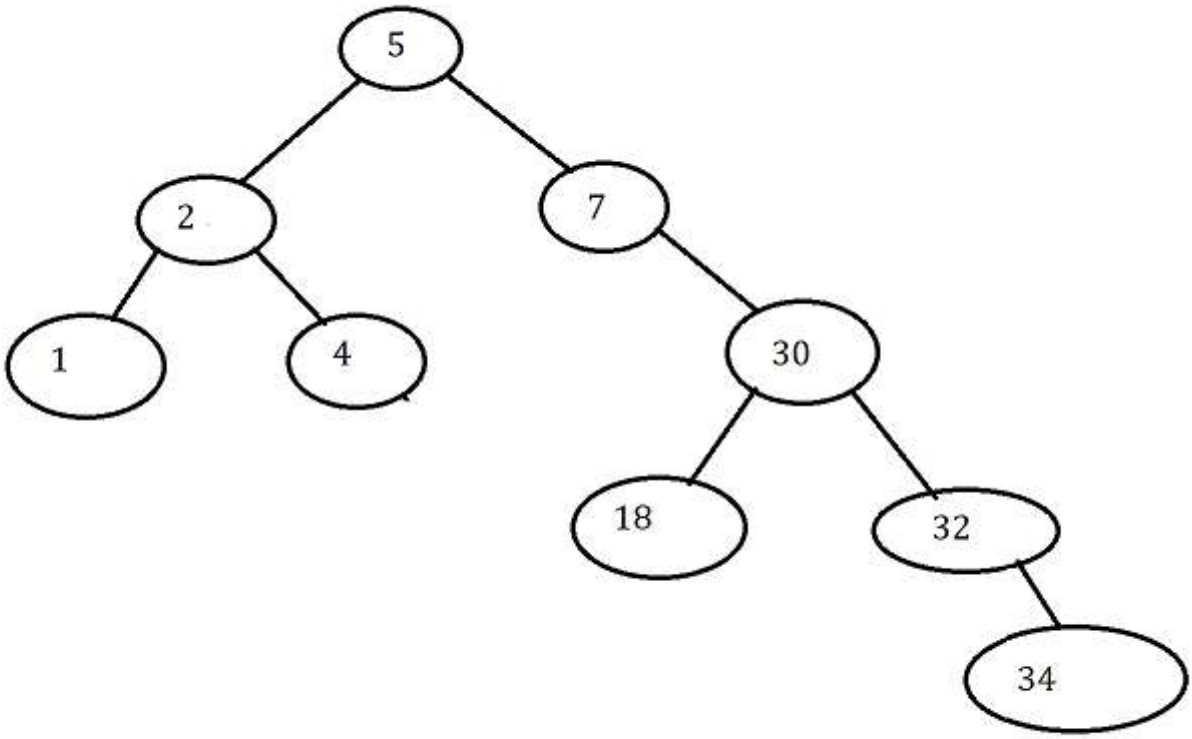


Then we add 3 more numbers and our tree again becomes disbalanced. We change the position of 2 and 1.

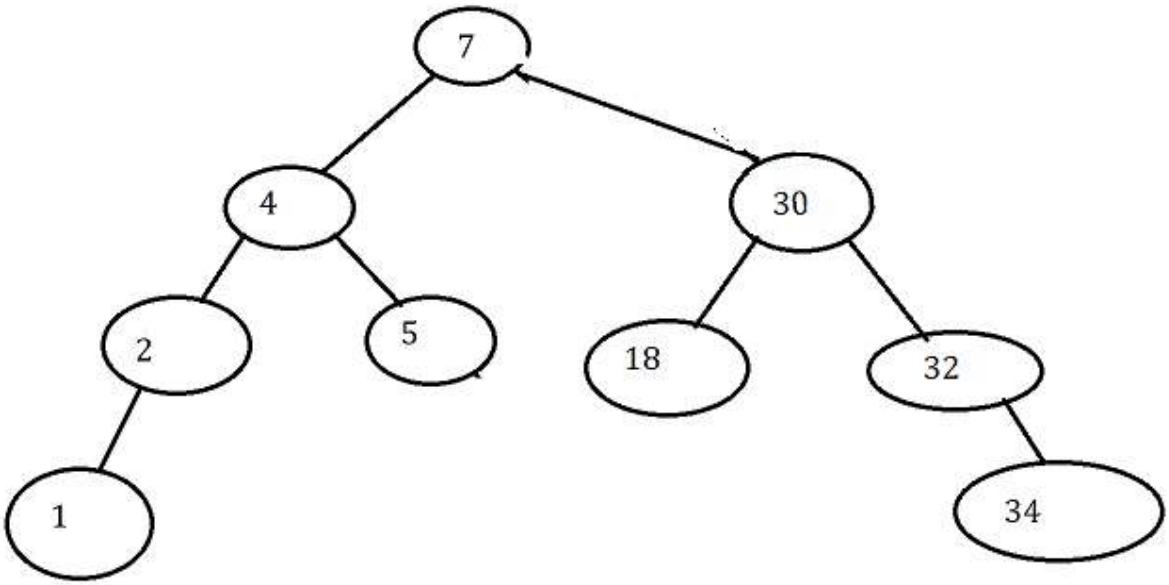

5
1
7
2
30
4

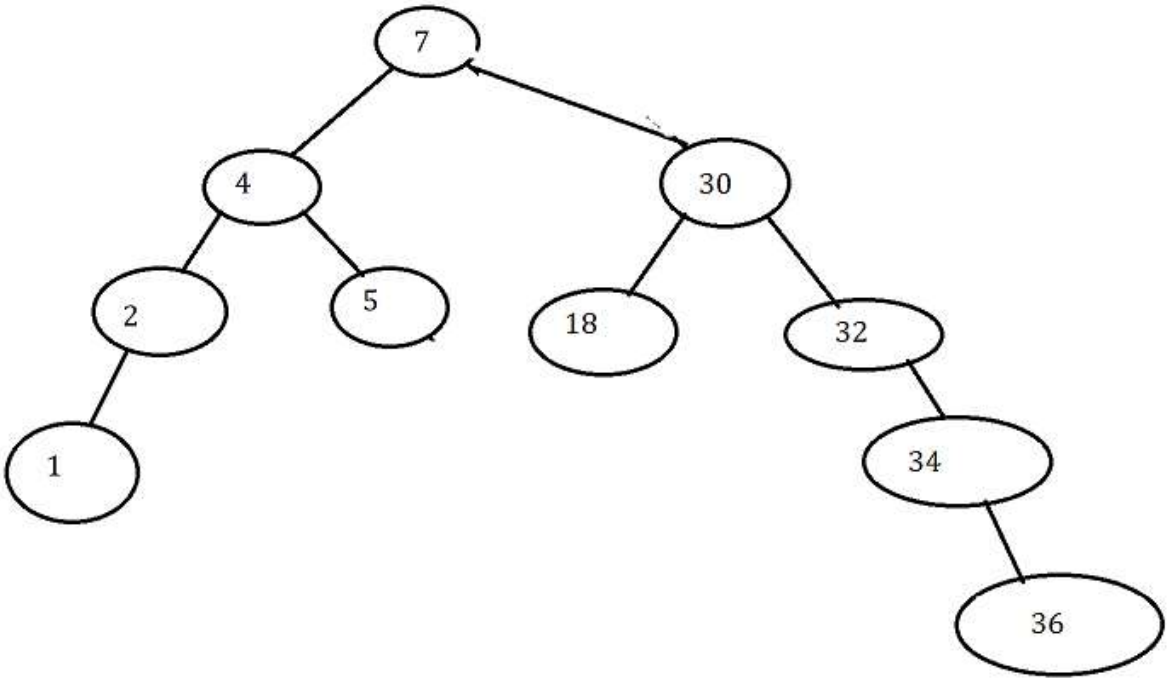

5
2
7
1
4
30


5
2
7
1
4
30
18
32

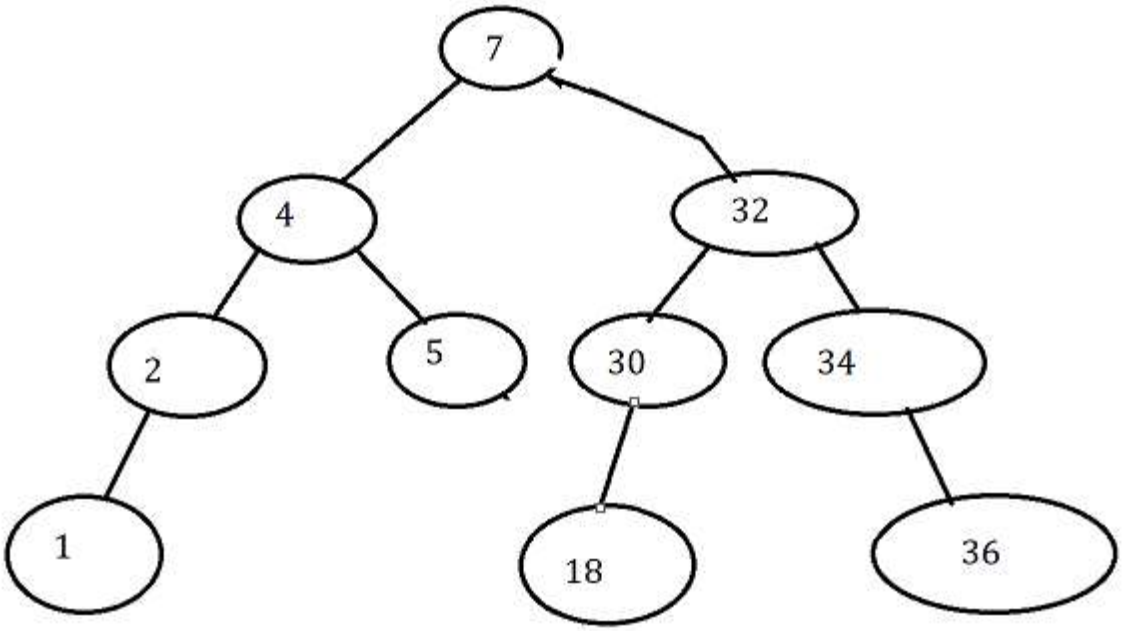

5
2
7
1
4
30
18
32
34

Now our tree disbalanced again because height of 2 and 7 differs with more than 1 branch(2). So we change the position of 7 and move it to the top. Actually the left side after this movement also became disbalanced, so 4 moved to the top and left branch is become balanced.



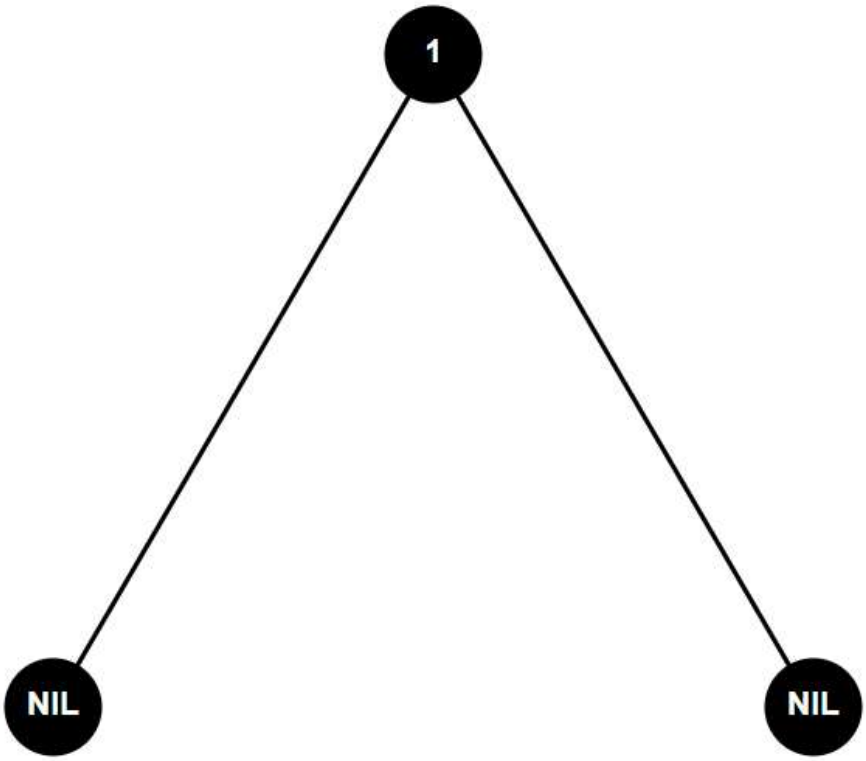


After inserting 36 we should again balance our tree, as 32 has two more children than 18. So we move 32 to 30th place and below it is the BST with AVL-balancing.

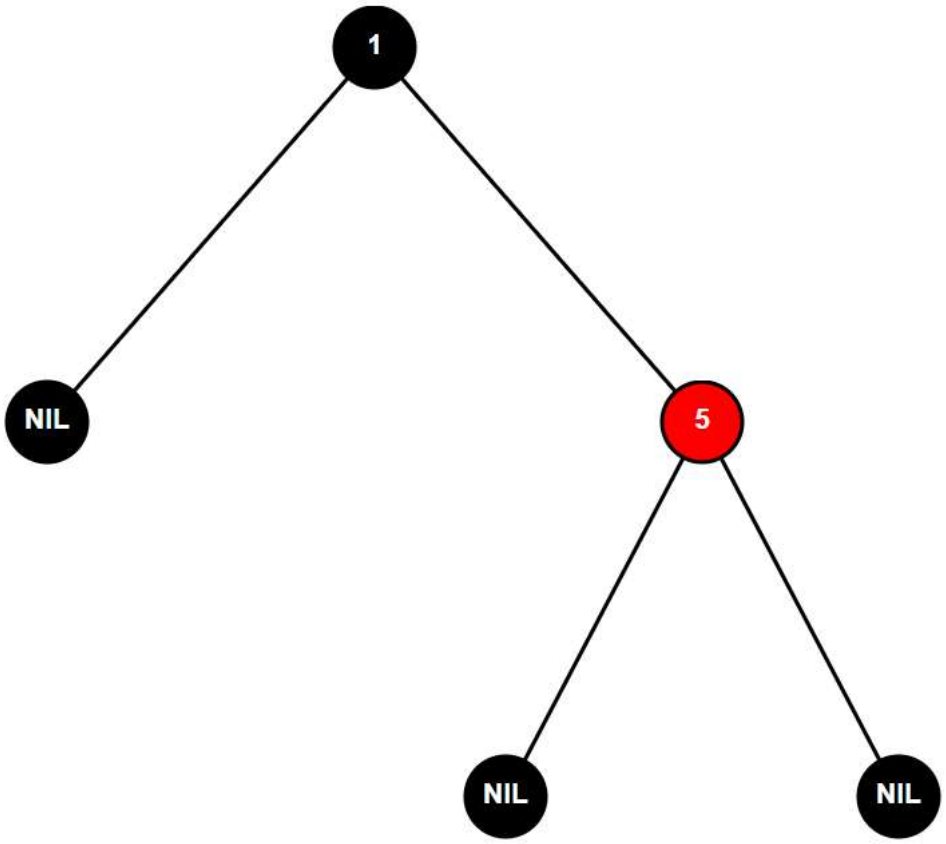


To search 35 in this tree we perform 4 comparisons and figure out that actually we don't have 35 in our tree(35>7,32,34 but less than 36).

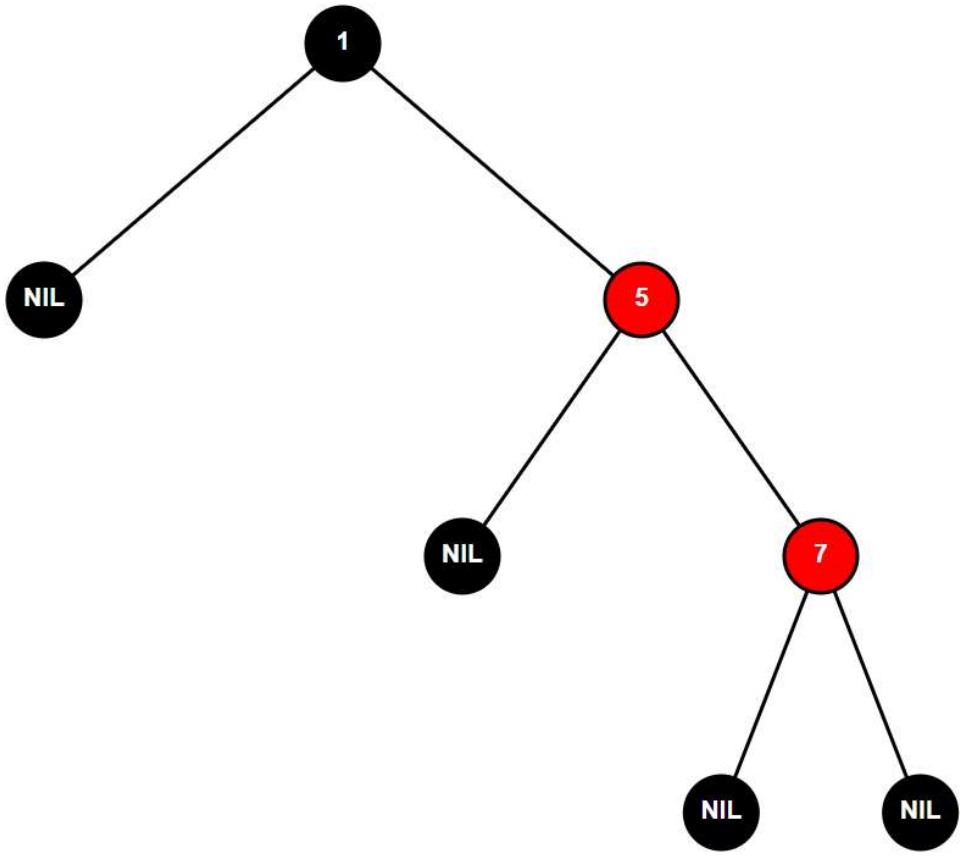
EX5: Red black BST has the several properties or rules that should be obeyed: 1.Tree is a valid binary search tree 2.All nodes are either red or black 3.The root is black 4.All leaves are black 5.Every red node has two black children 6.Every path from root to leaf contain the same number of black nodes



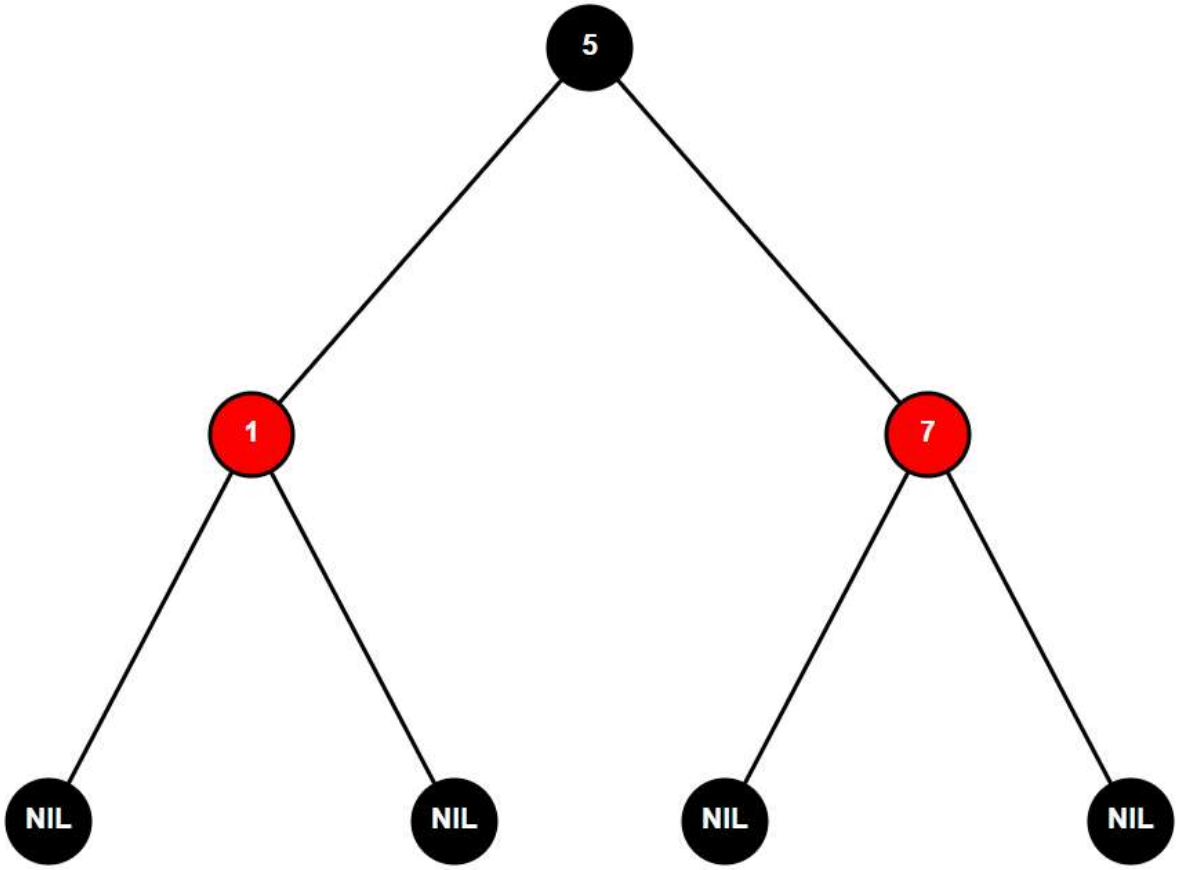
This is what we have after first insertion and as all requirements are fulfilled we can move to the second step.



We have inserted 5 and it is red because "Every path from root to leaf contain the same number of black nodes".

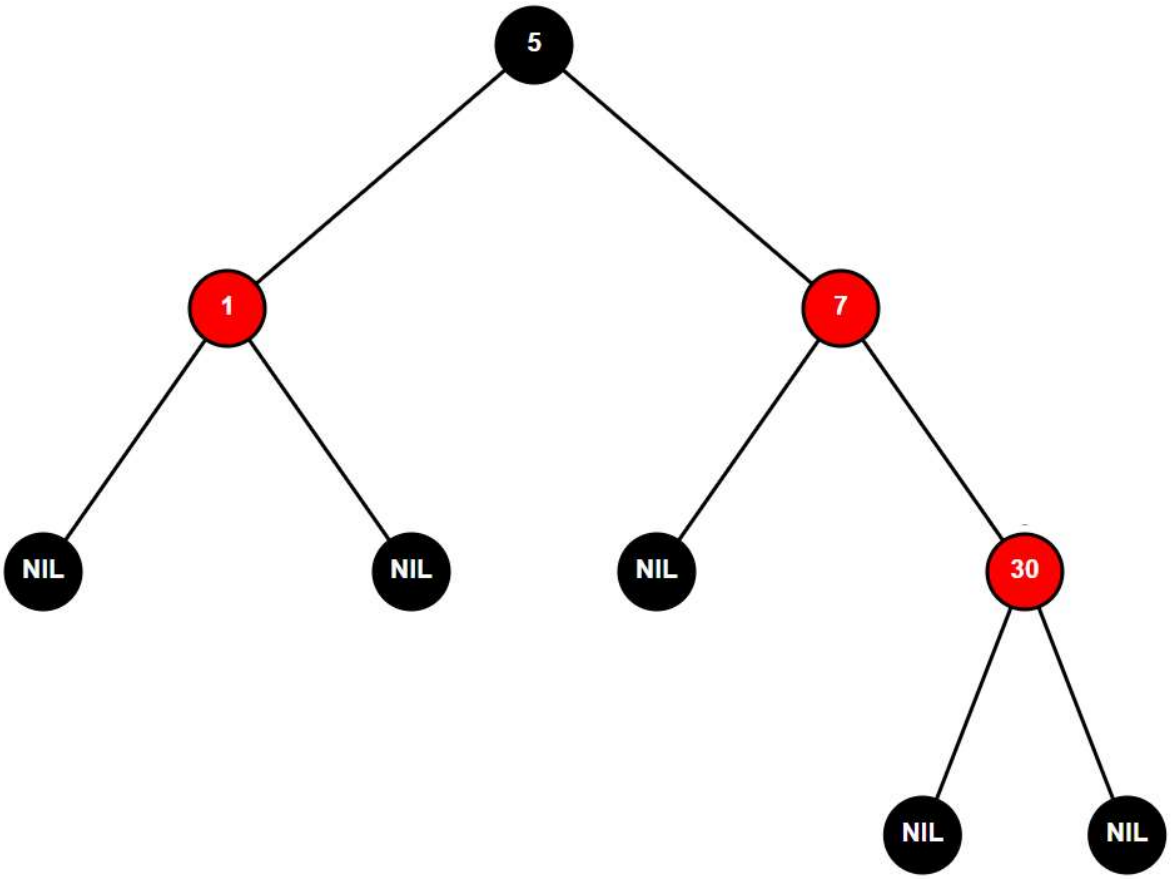


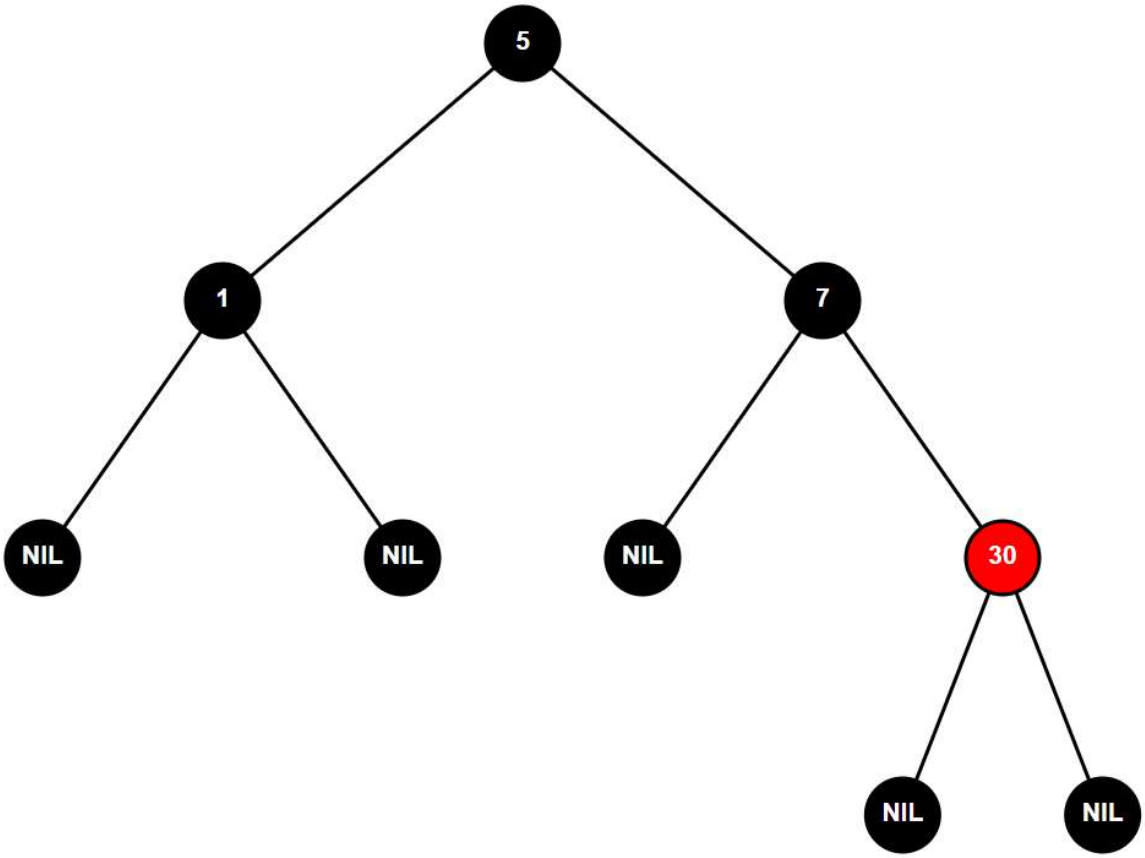
By inserting 7 now we have a conflict and we should rotate the tree, move 5 to the top and change its color with 1. This is right right case.



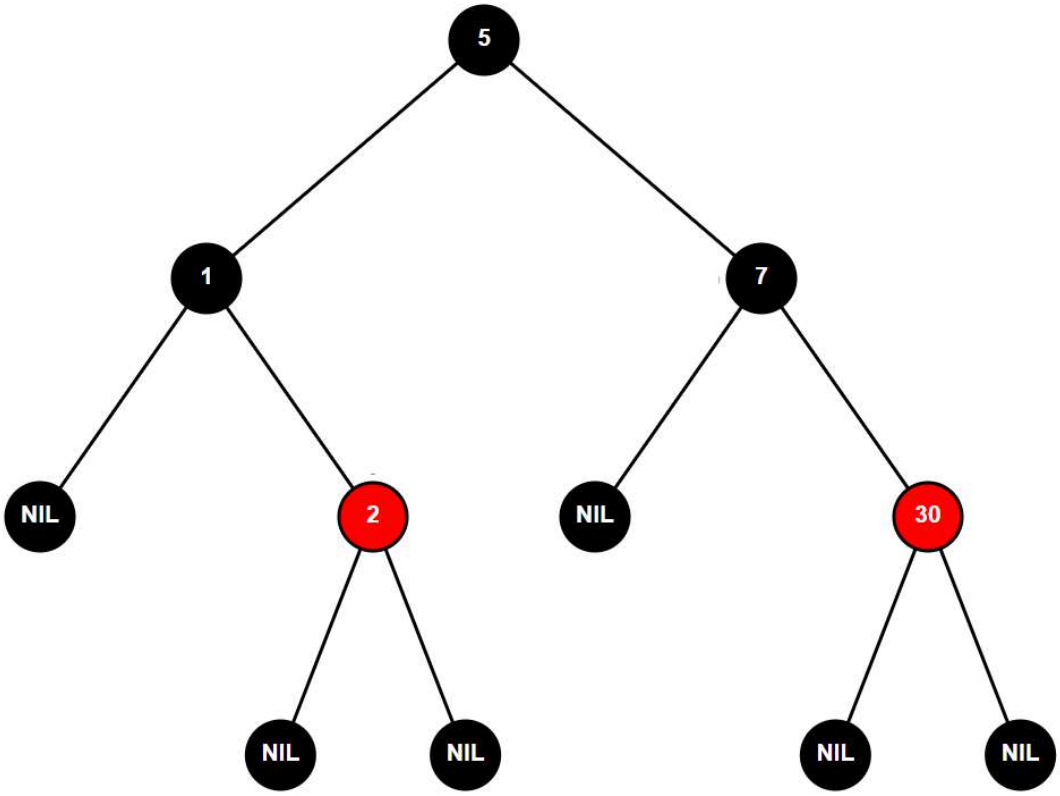
After insertion of 30 we have new kind of conflict. Every red node should have two black children, but 7 does not. So we change the color of 7 and 1.

1. 30's uncle is RED
2. Change color of 1 and 7 as BLACK.
3. Change color of grand parent as RED.
4. Change 30 = 30's grandparent.

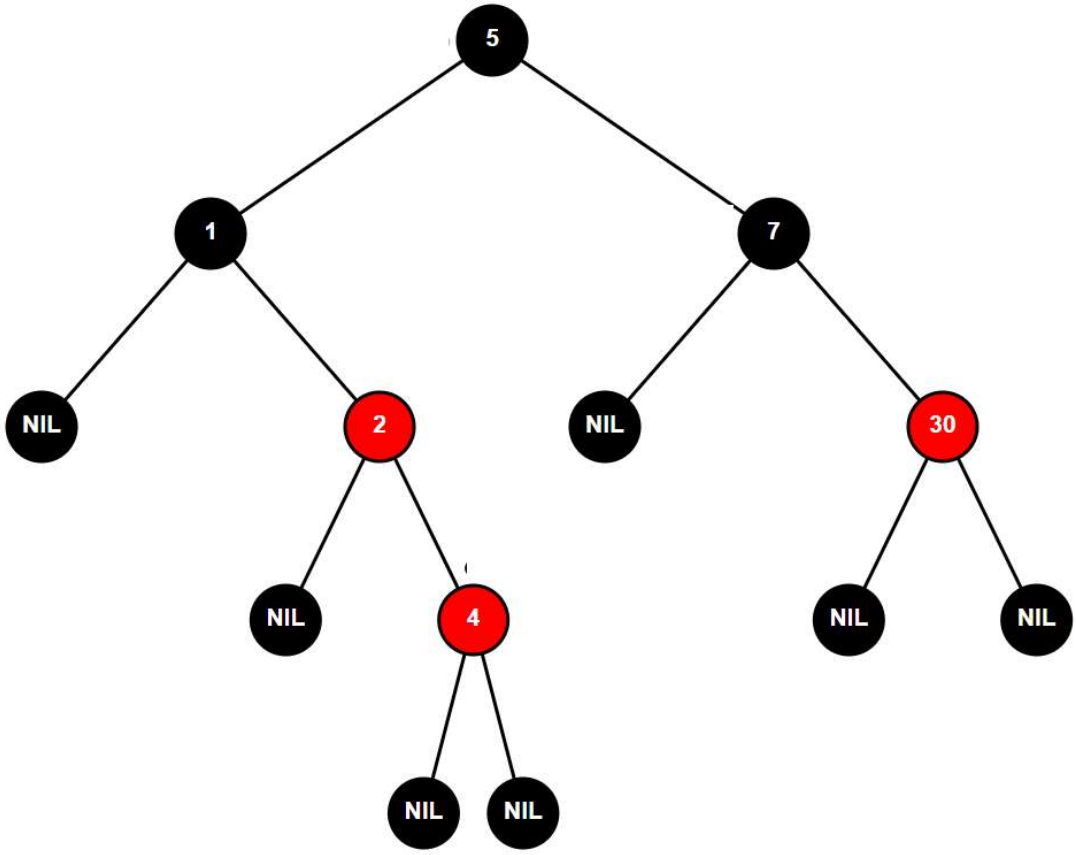

5
1
7
NIL
NIL
NIL
30
NIL
NIL



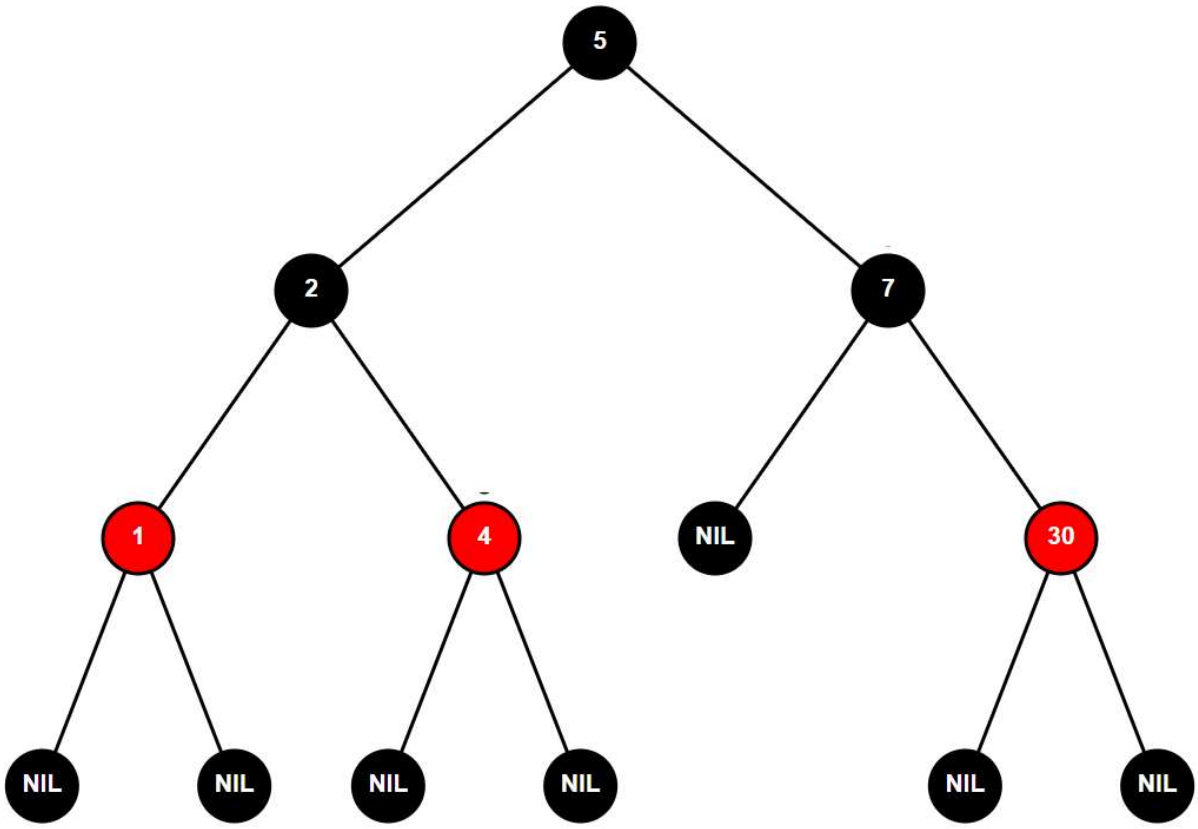
Inserting 2 happenned without a conflict.

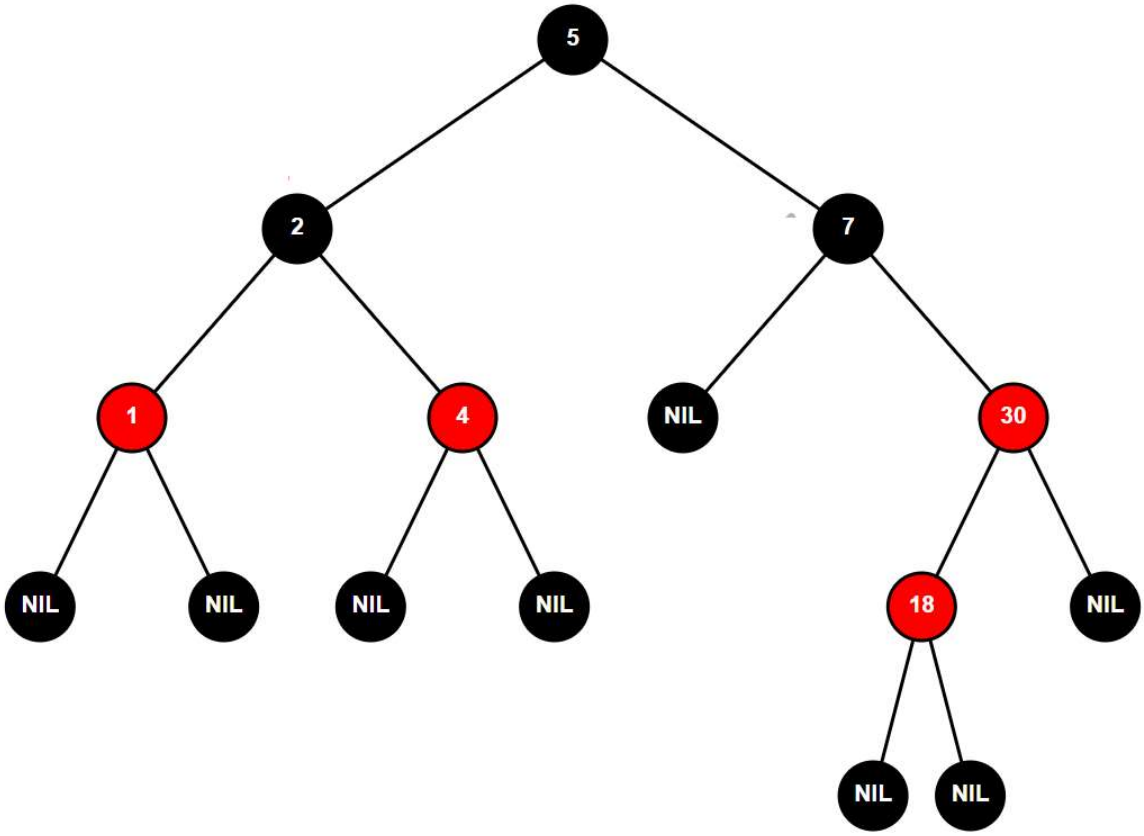


Now we insert 4.

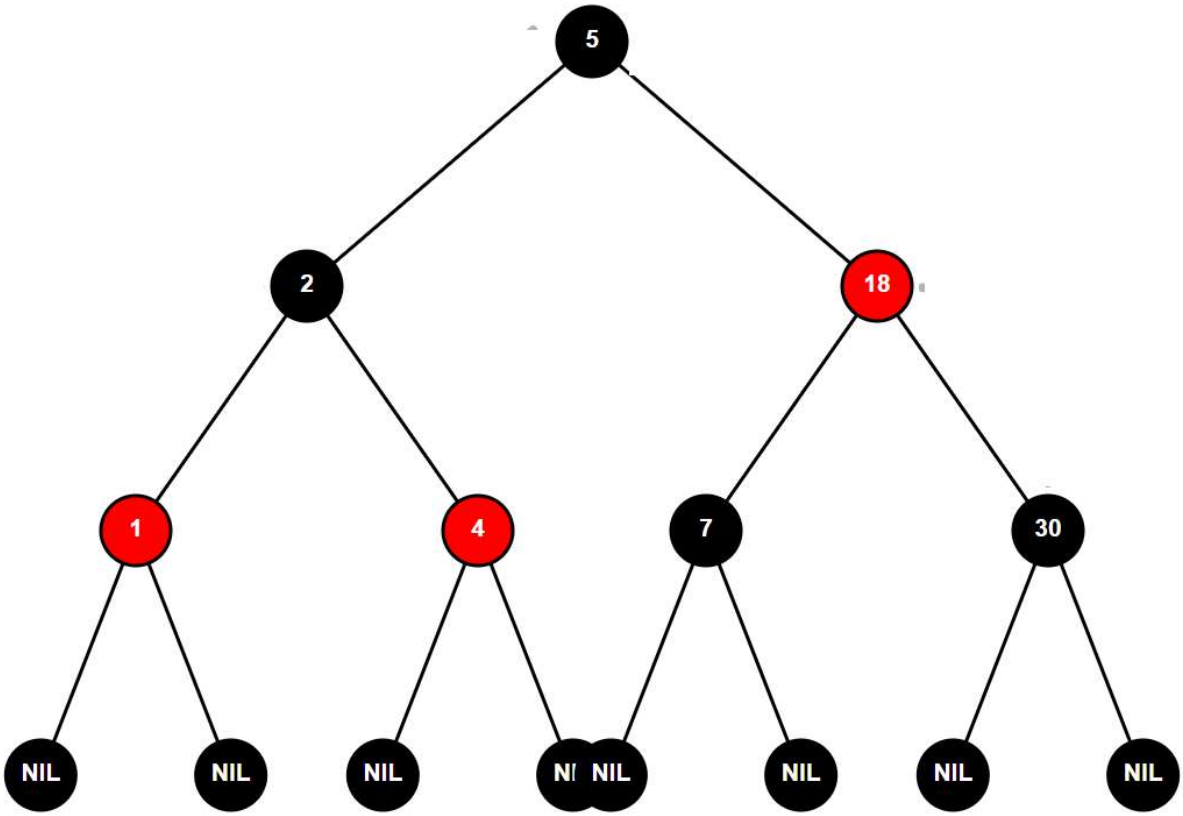


and again we have "Every red node has two black children" conflict", so we move 2 to the upper level, change its and 1's color. Left Right Case.

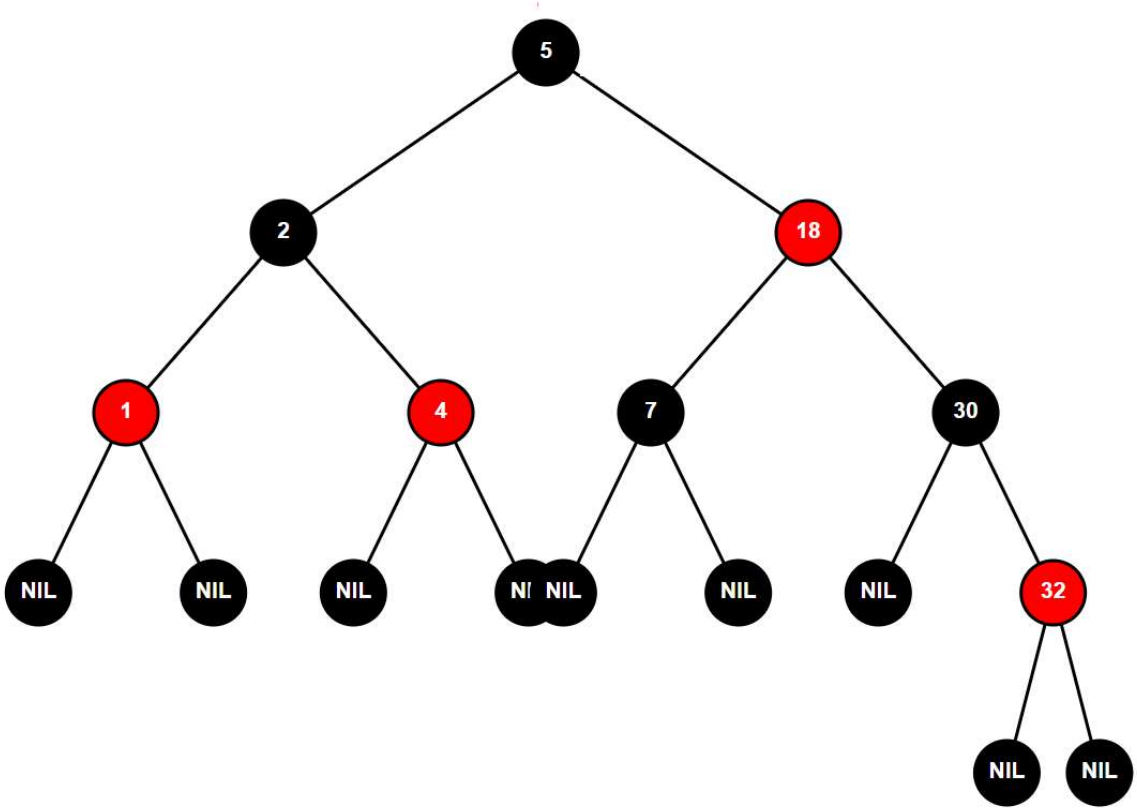

5
2
7
1
4
NIL
30
NIL
NIL
NIL
NIL
NIL
NIL


5
2
7
1
4
NIL
30
NIL
NIL
NIL
NIL
18
NIL
NIL
NIL

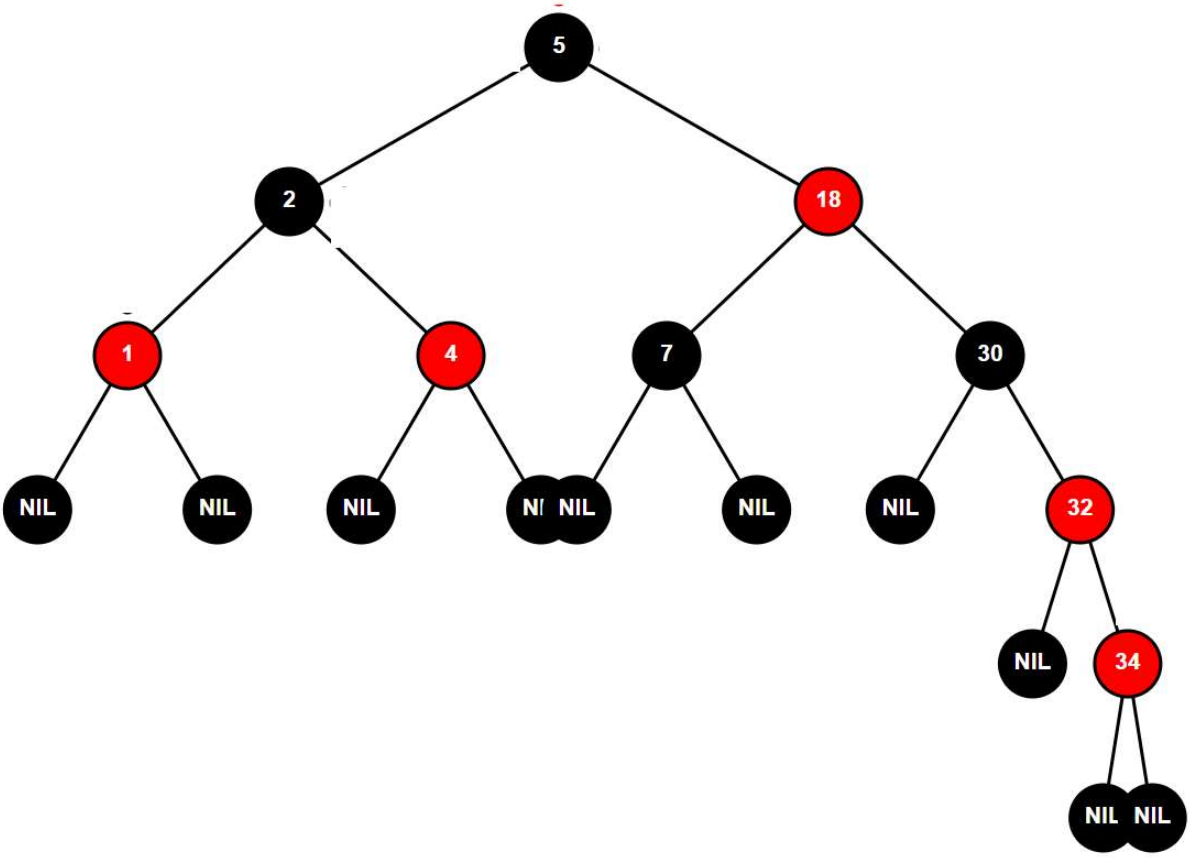
By inserting 18 we have the red node which has only one black node, so again we should rotate. Right right case.



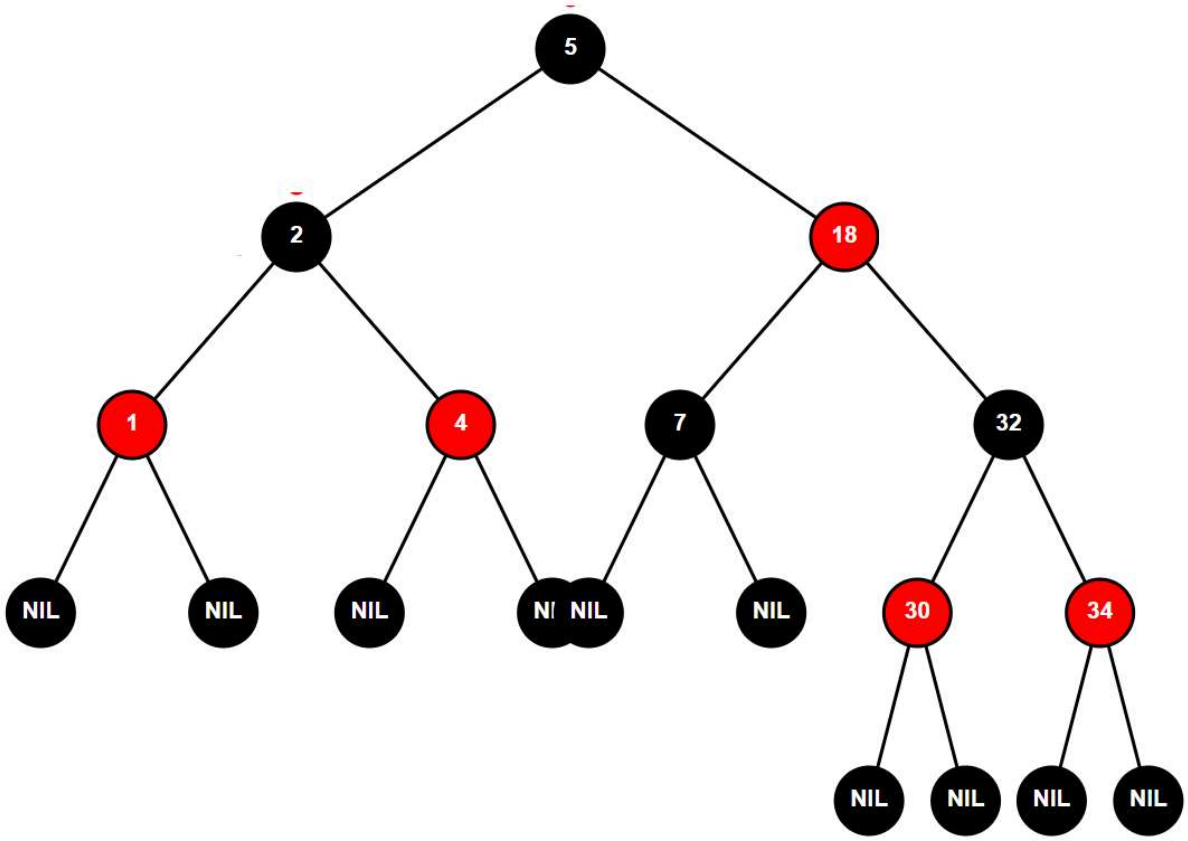
now 18 is red and on the second level.



With inserting 32 we don't have any trouble.

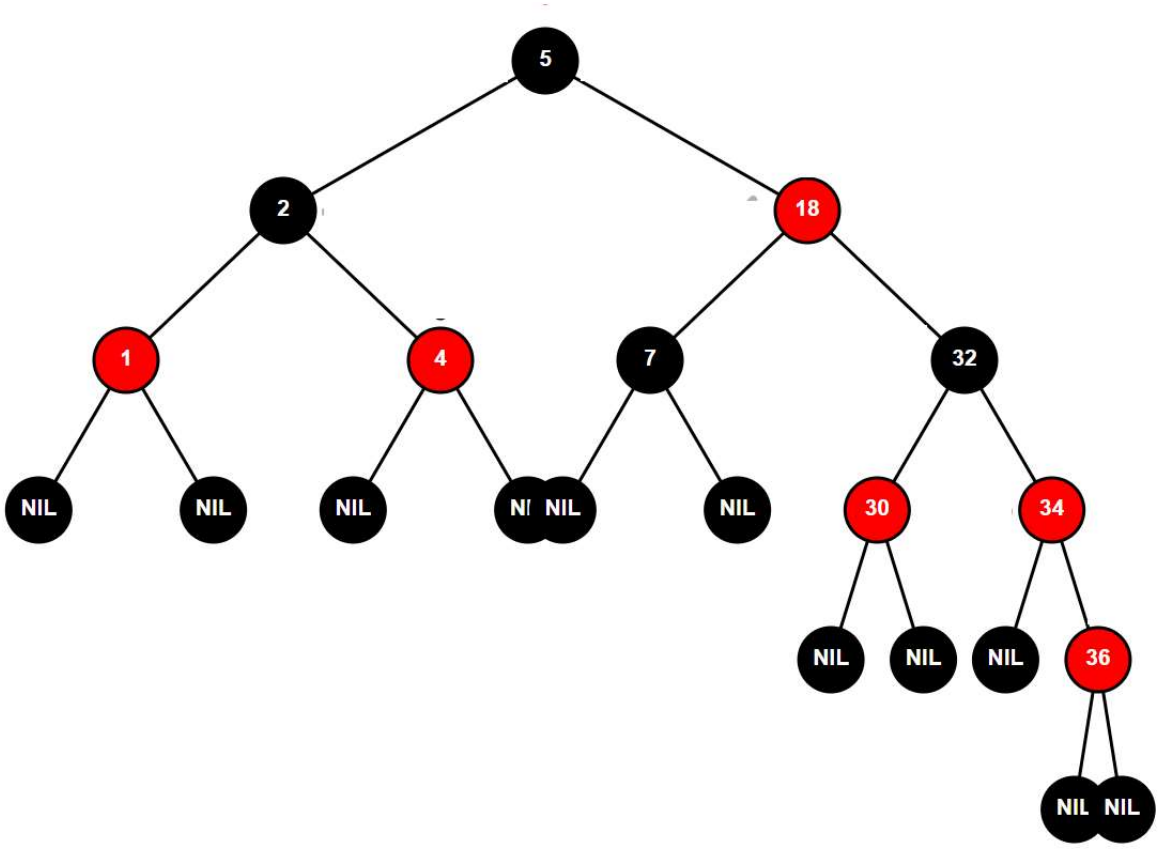


With inserting 34, 32 has only one black child node. Right right case.

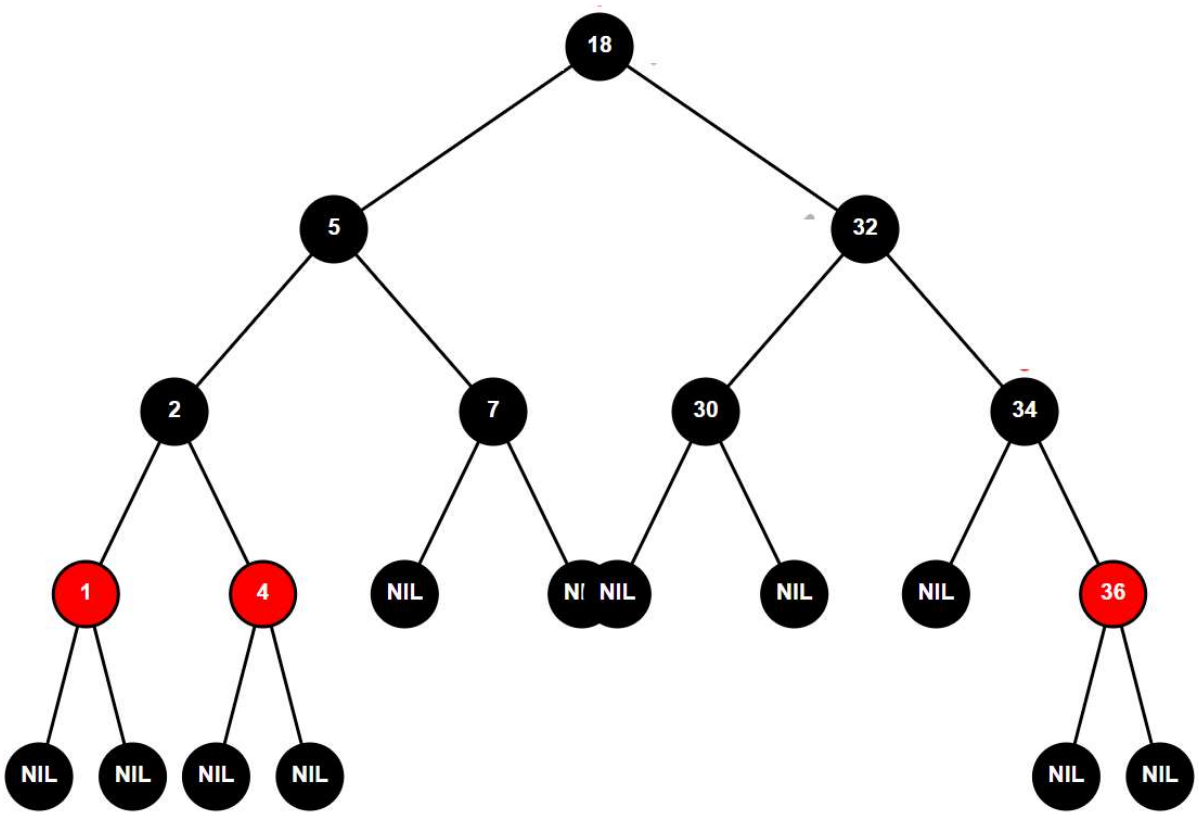


We rotate 32 and 30 and change theirs color.

The final step was inserting 36 and it totally messed up the structure that I was trying to apply.



Here 34 had only one black node, but as the structure of tree becomes skewed to the rightm I decided to change the top node and make it 18.



Changed 18's color to black and remain only 1,4 and 36 red. This tree obey to all 6 rules that are prescripted to black-red trees. I rotated the tree after 6 insertions.