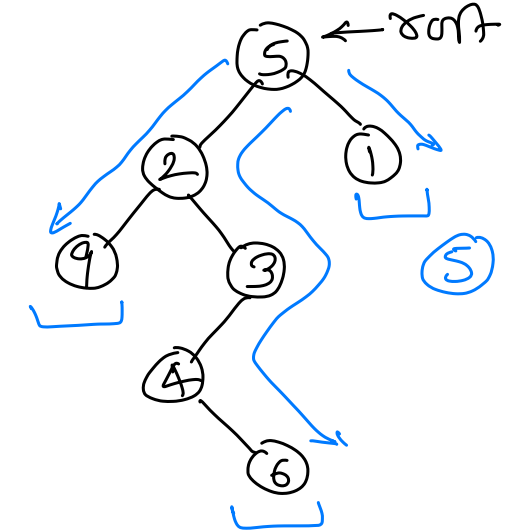
Height of a node in a tree

Number of edges on the longest path from that node to a leaf node.



5 → 1 = 1 edge

5 → 2 → 9

= 2 edges

5 → 2 → 3 → 4 → 6

= 4 edges

root → empty

height of root = 0

root → 5

height = 0

$$height(root) = \begin{cases} 0, & \text{if root is empty.} \\ 0, & \text{if root is leaf node} \\ 1 + MAX(height(root's\ left\ child), height(root's\ right\ child)), & \text{Otherwise} \end{cases}$$

AVL Tree

Height balanced binary search tree.

Each node has a balance factor between -1 and +1.

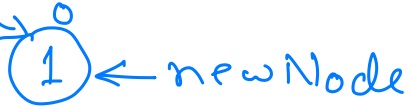
Balance factor (B.F.) = $h_L - h_R$

h_L → height of left subtree

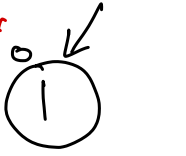
h_R → height of right subtree.

insert (1)

root → empty



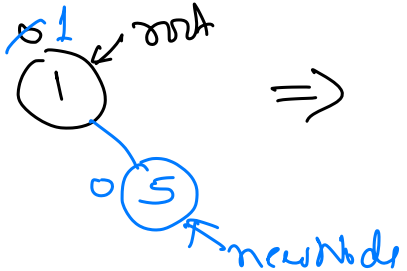
Balance factor as absolute



⇒ difference

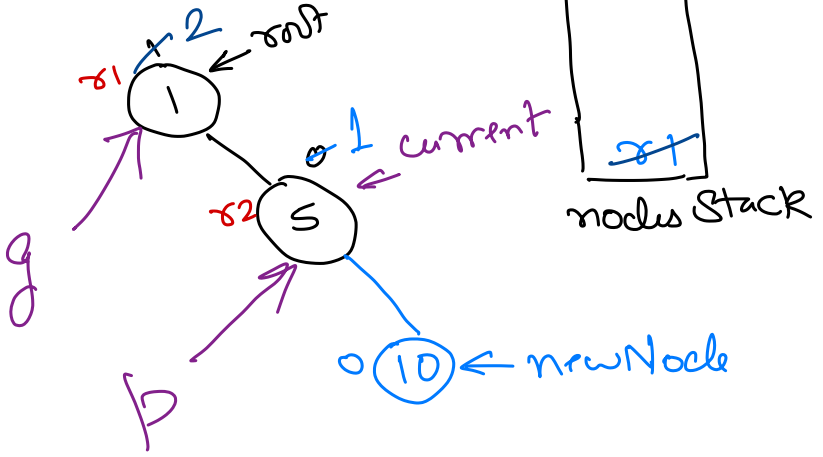
$|h_L - h_R|$

insert (5)



⇒

insert (10)



← To track nodes in path from root till newNode.

Left Rotation

RR imbalance

g → nearest parent of newNode, having incorrect balance factor.

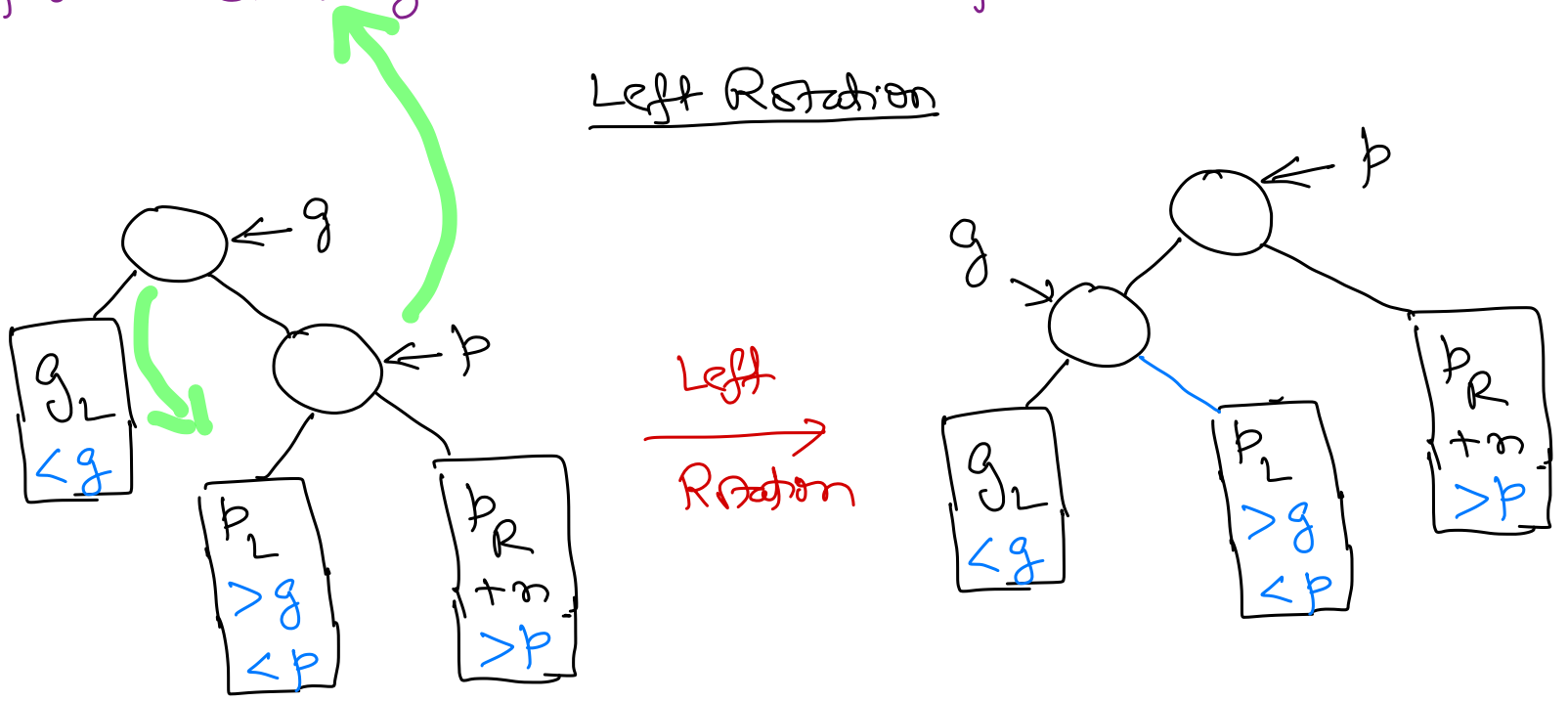
Take two steps from g towards newNode (we may reach newNode, we might not – doesn't matter).

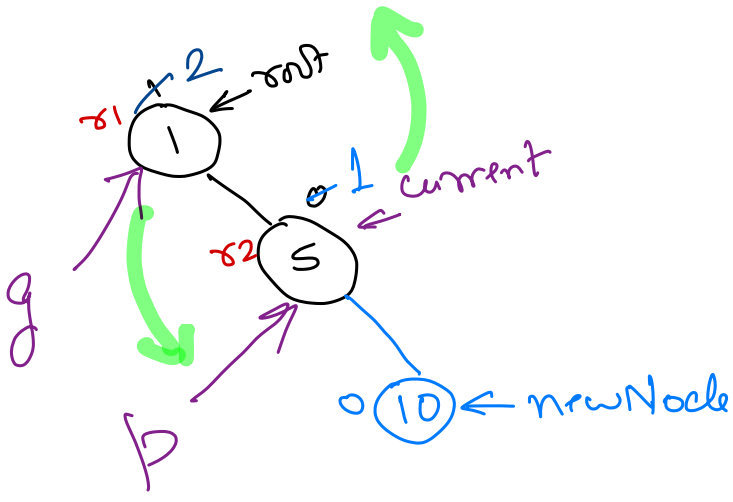
Record L, if followed left child.

Record R, if followed right child.

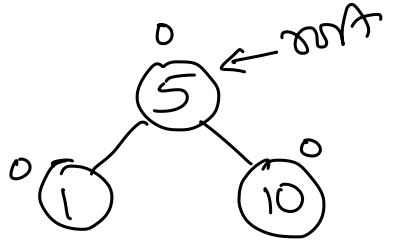
For Left or Right rotation, p is child node of g, in the path from g to new Node.

Left and Right rotations are performed around p.



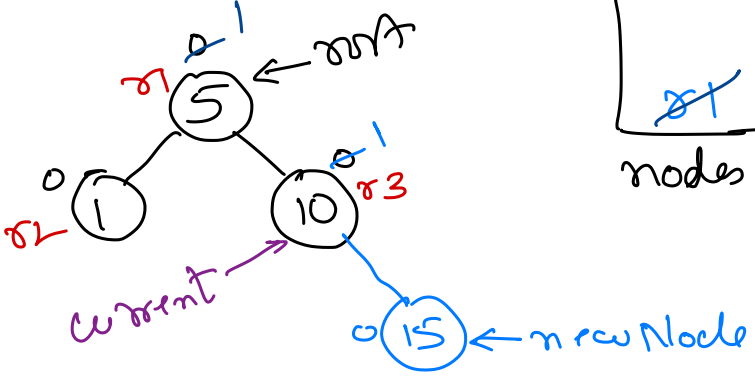


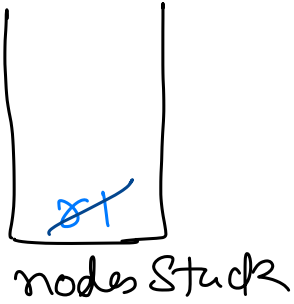
Left Rotation



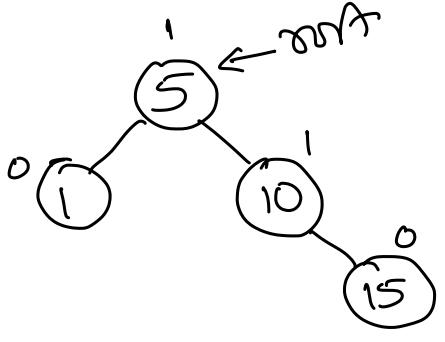
---

insert (15)

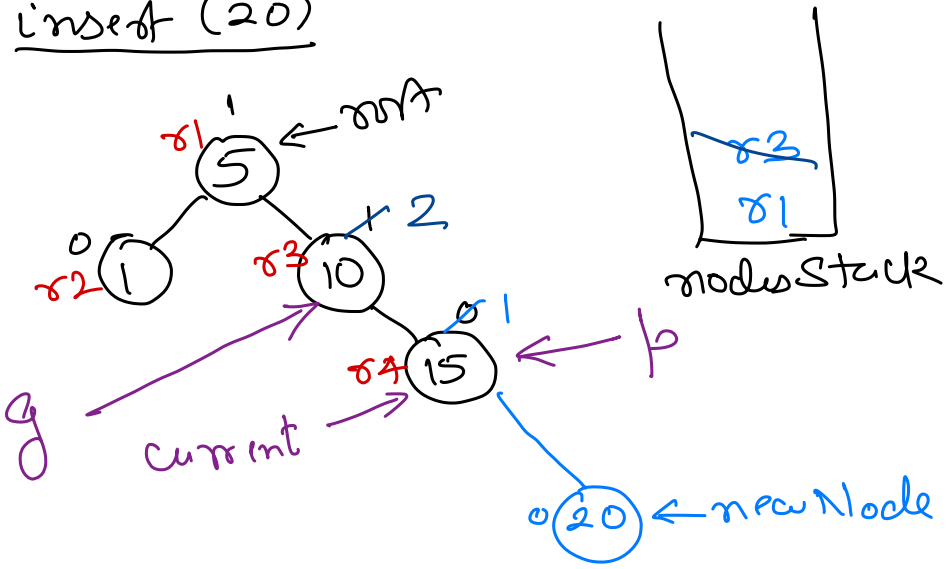




⇒



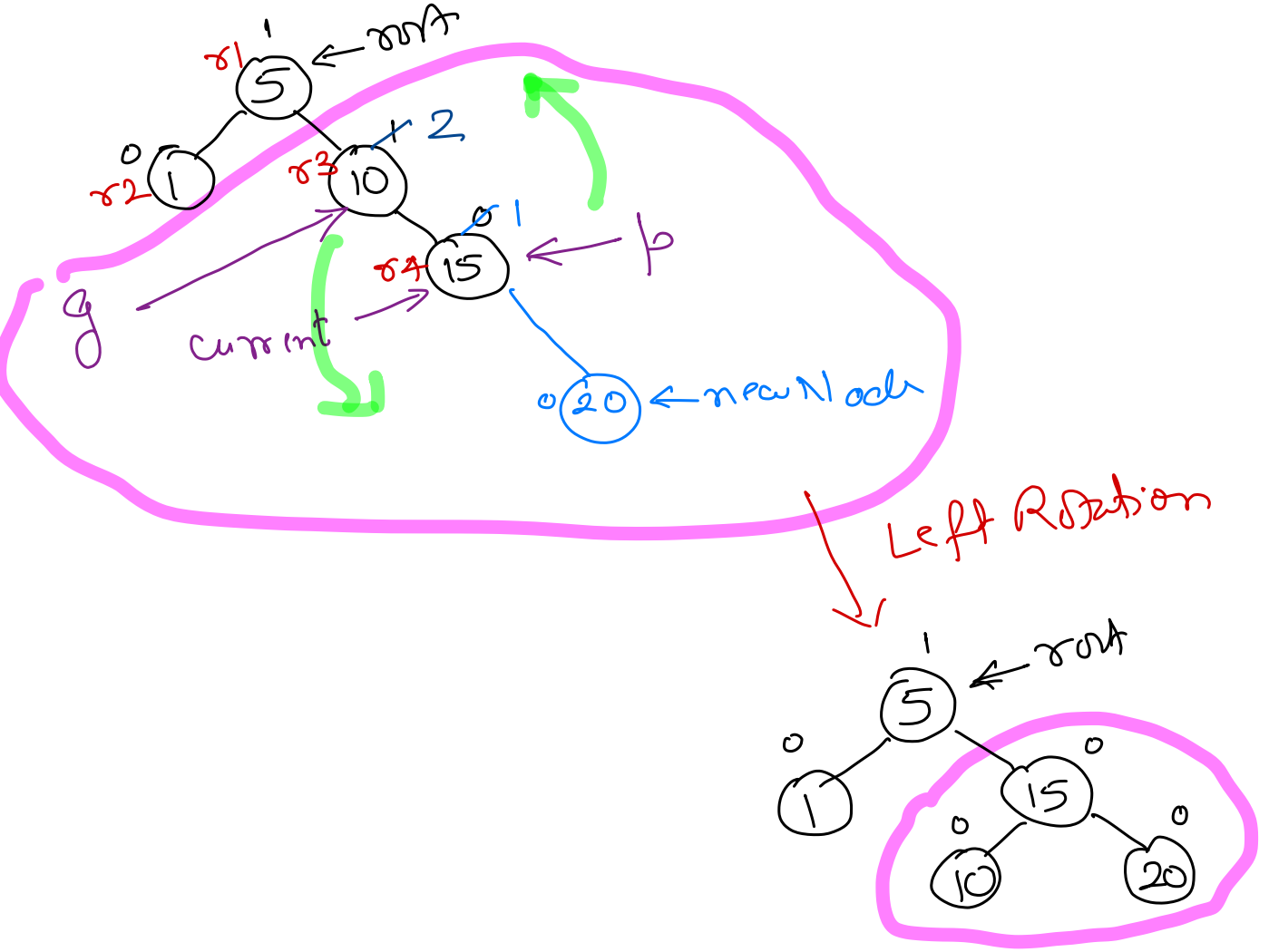
insert (20)



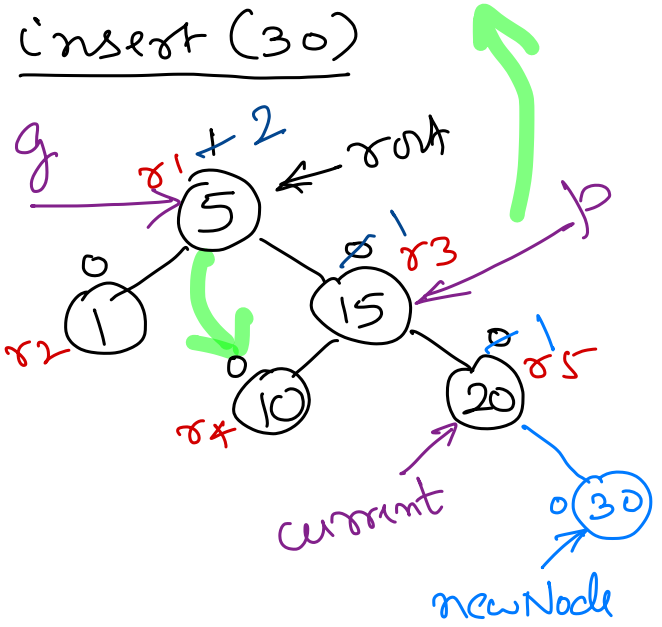
RR imbalance

⇓

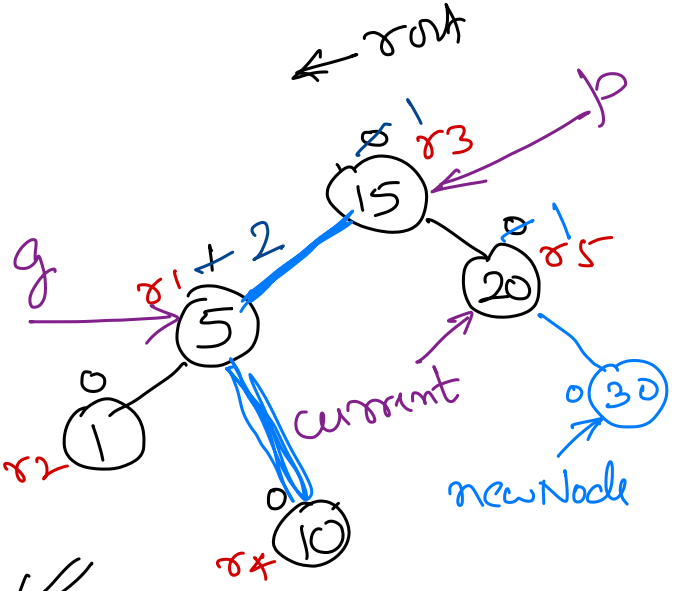
Left Rotation

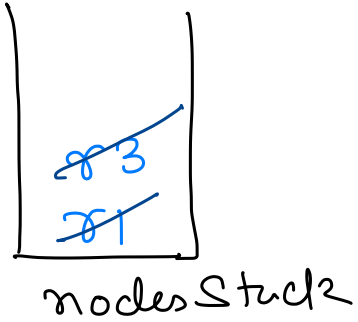

root
5
1
x1
0
1
x2
10
2
x3
0
15
1
x4
p
g
current
0
20
newNode
Left Rotation
root
5
1
0
1
0
15
0
0
10
0
20

insert (30)

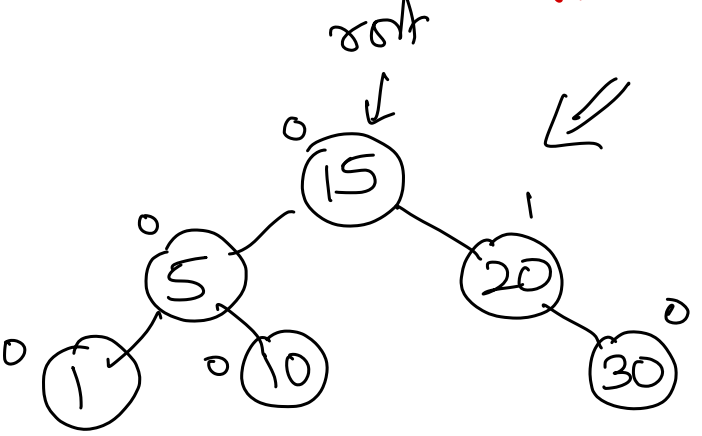


RR imbalance ⇒ Left Rotation

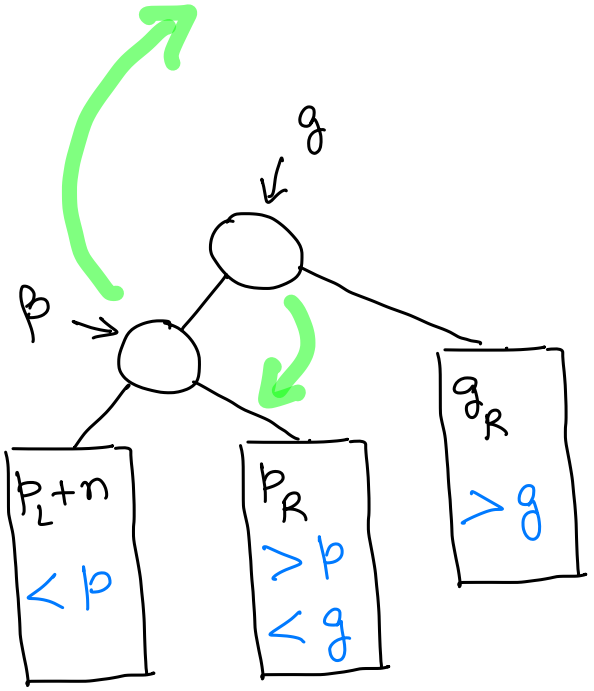




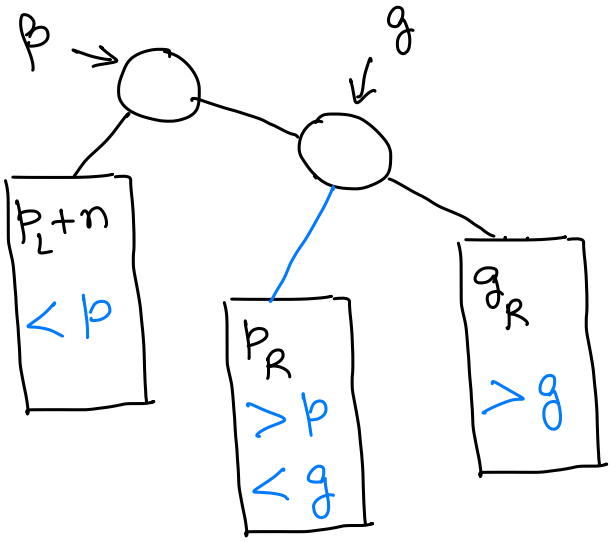
nodesStack



Right Rotation → Mirror image of Left Rotation.

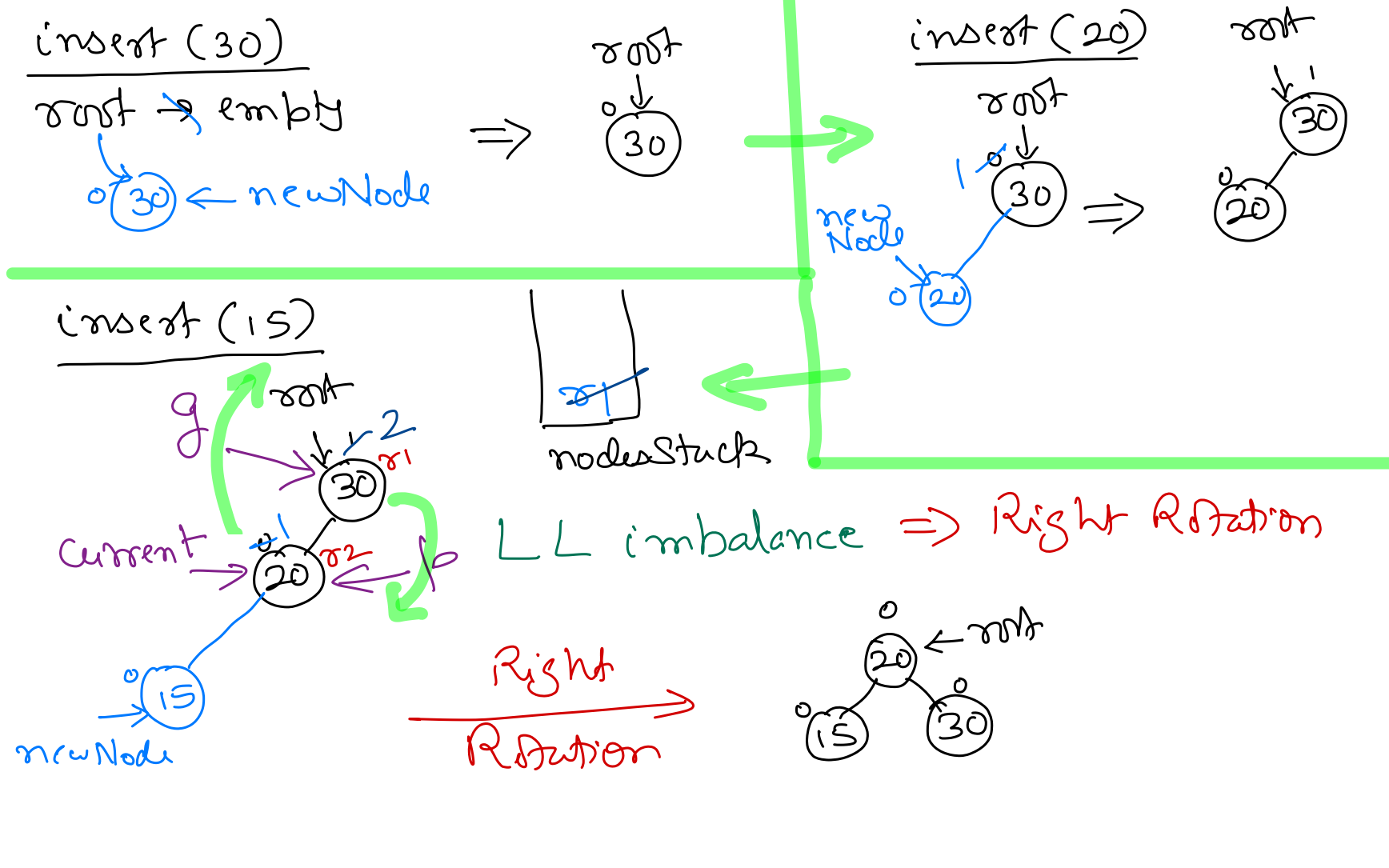


Right Rotation →

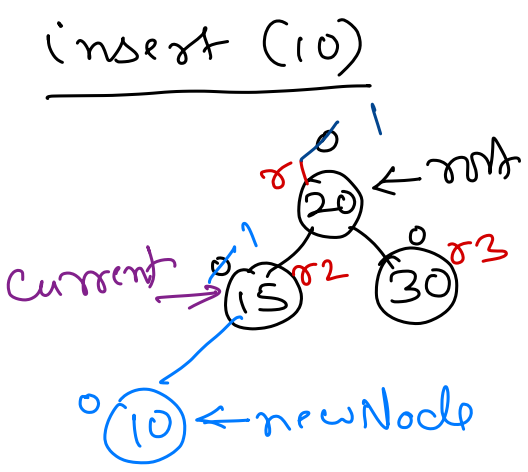


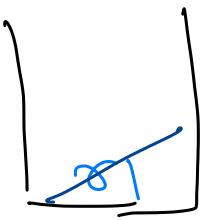
---

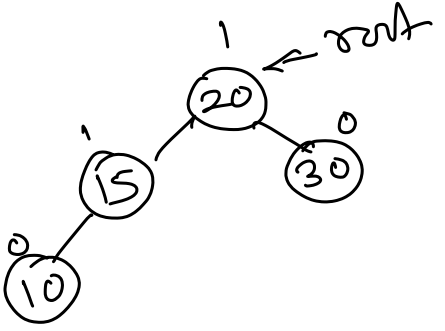
insert: 30 20 15 10 5 1


insert (30)
root → empty
30
newNode
root
30
insert (20)
root
1
30
newNode
20
root
1
30
20
insert (15)
g
root
2
30
r1
current
1
r2
20
p
15
newNode
nodesStack
LL imbalance ⇒ Right Rotation
Right
Rotation
root
20
15
30

insert (10)

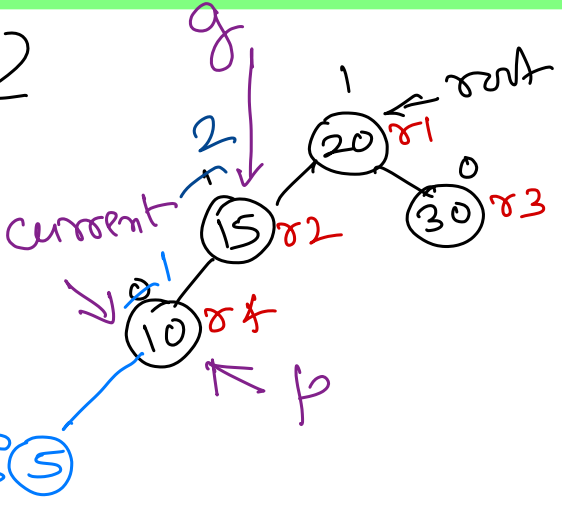






---

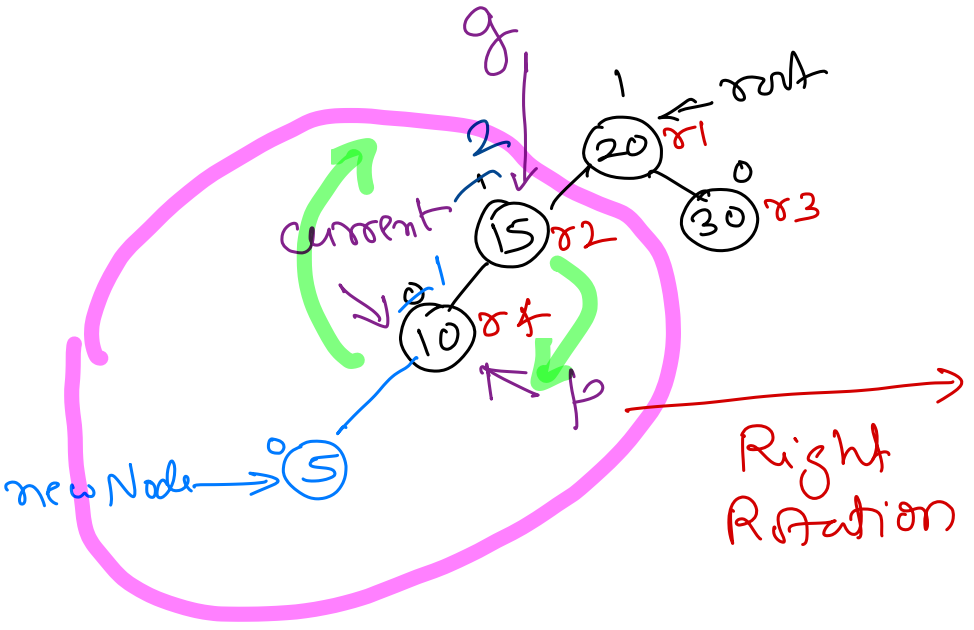
insert (5)

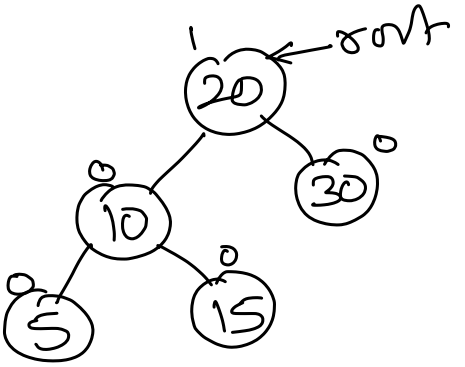


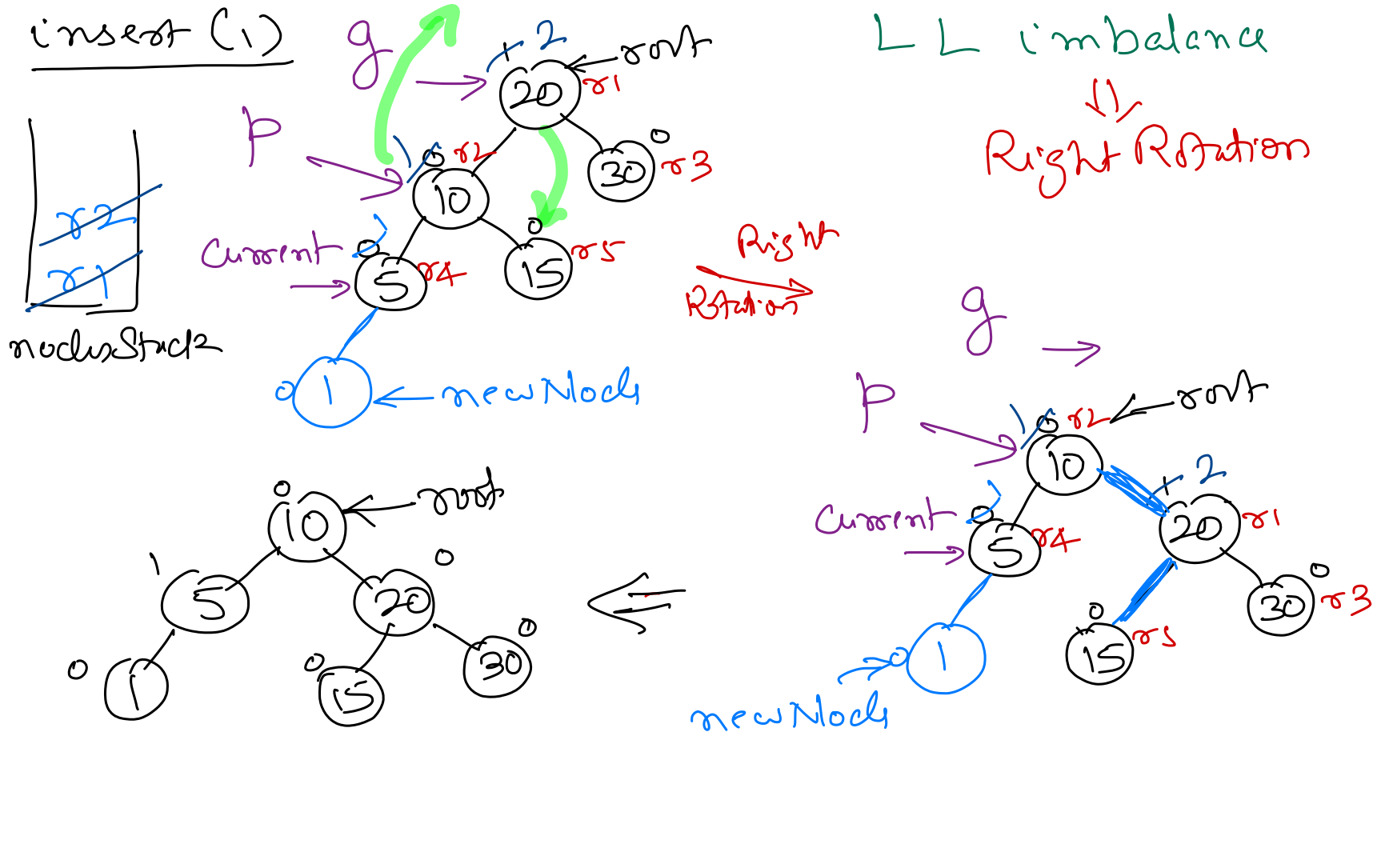
LL imbalance

⇓

Right Rotation

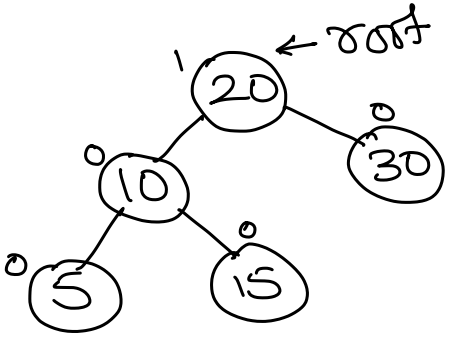

g
2
1
root
20
r1
current
15
r2
0
30
r3
1
0
10
r4
p
newNode
5
Right
Rotation


1
root
20
0
10
30
0
0
5
15

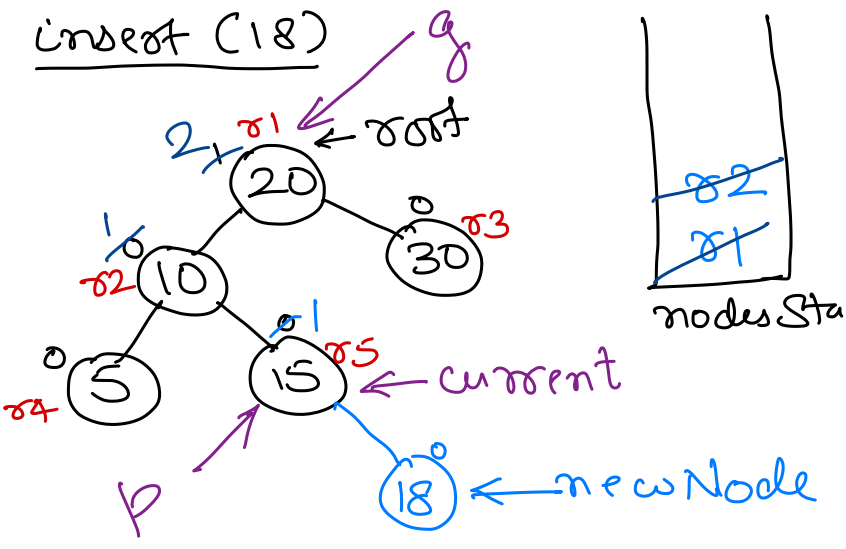

insert (1)
LL imbalance
Right Rotation
g
p
root
20
10
30
5
15
1
2
r1
r2
r3
r4
r5
Current
newNode
nodesStack
r2
r1
Right Rotation
10
5
20
1
15
30

Insert :

30 20 15 10 5



insert (18)

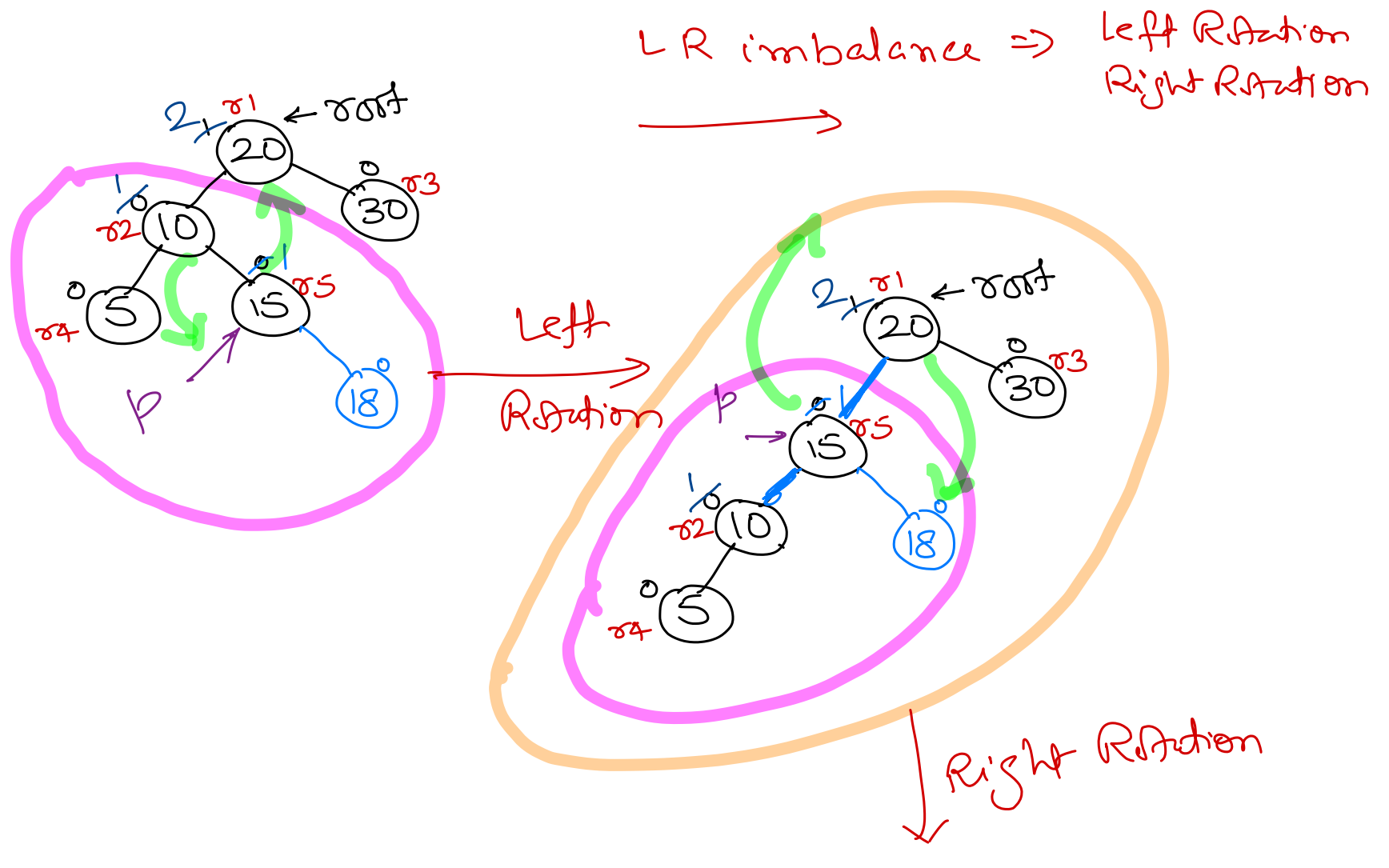


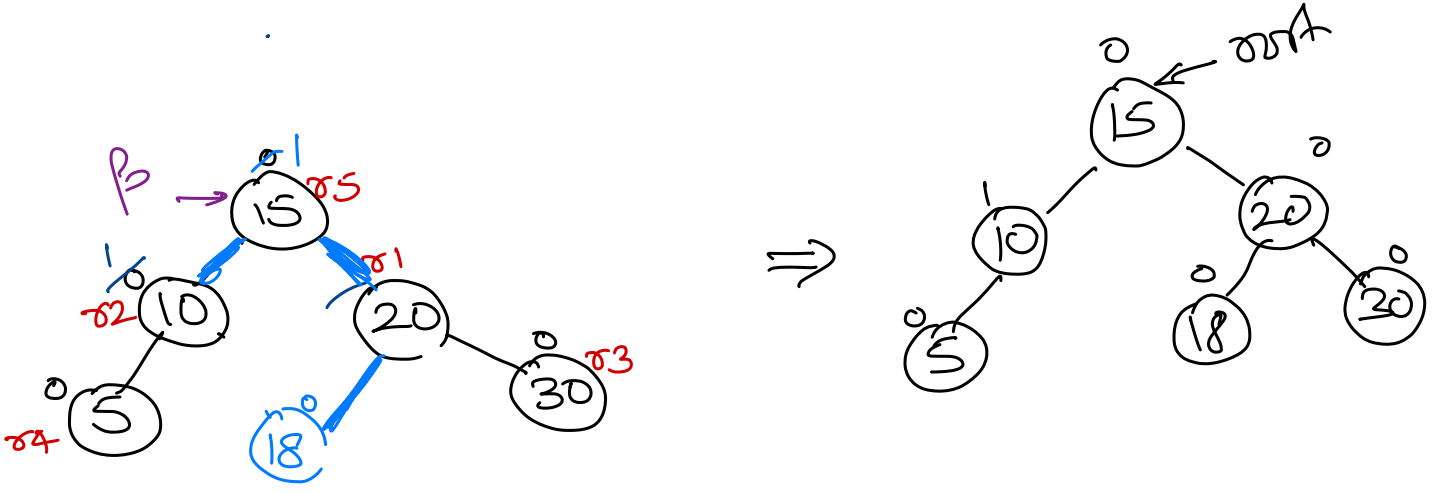
For LR imbalance and LR imbalance, we perform two rotations around p.

p → grand child of g, in path from g towards new Node.

LR imbalance ⇒ Left Rotation
Right Rotation

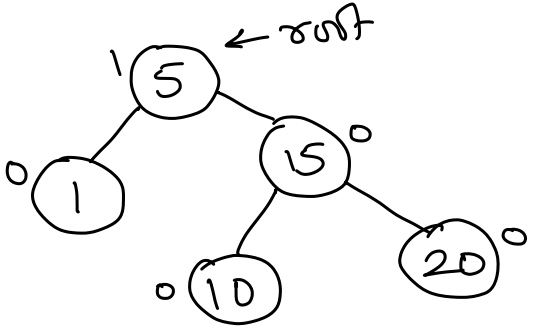
RL imbalance ⇒ Right Rotation
Left Rotation


LR imbalance ⇒ Left Rotation
Right Rotation
root
20
30
10
5
15
18
p
Left
Rotation
Right Rotation

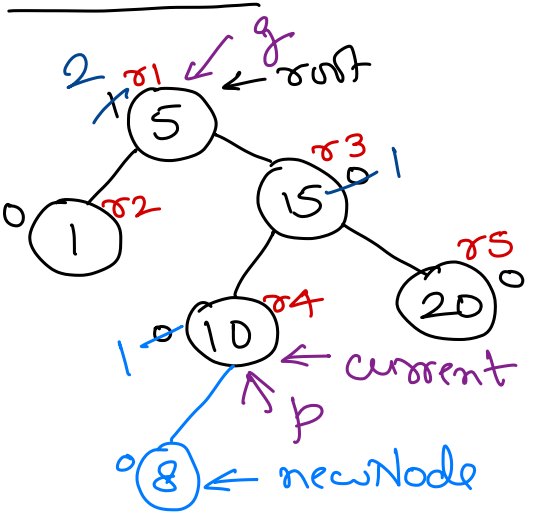

p
15
10
20
5
18
30
15
10
20
5
18
20

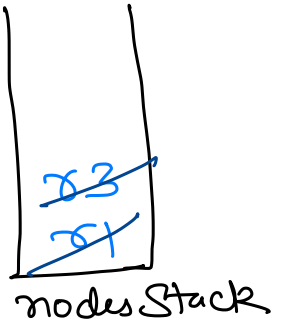
Insert:

1 5 10 15 20



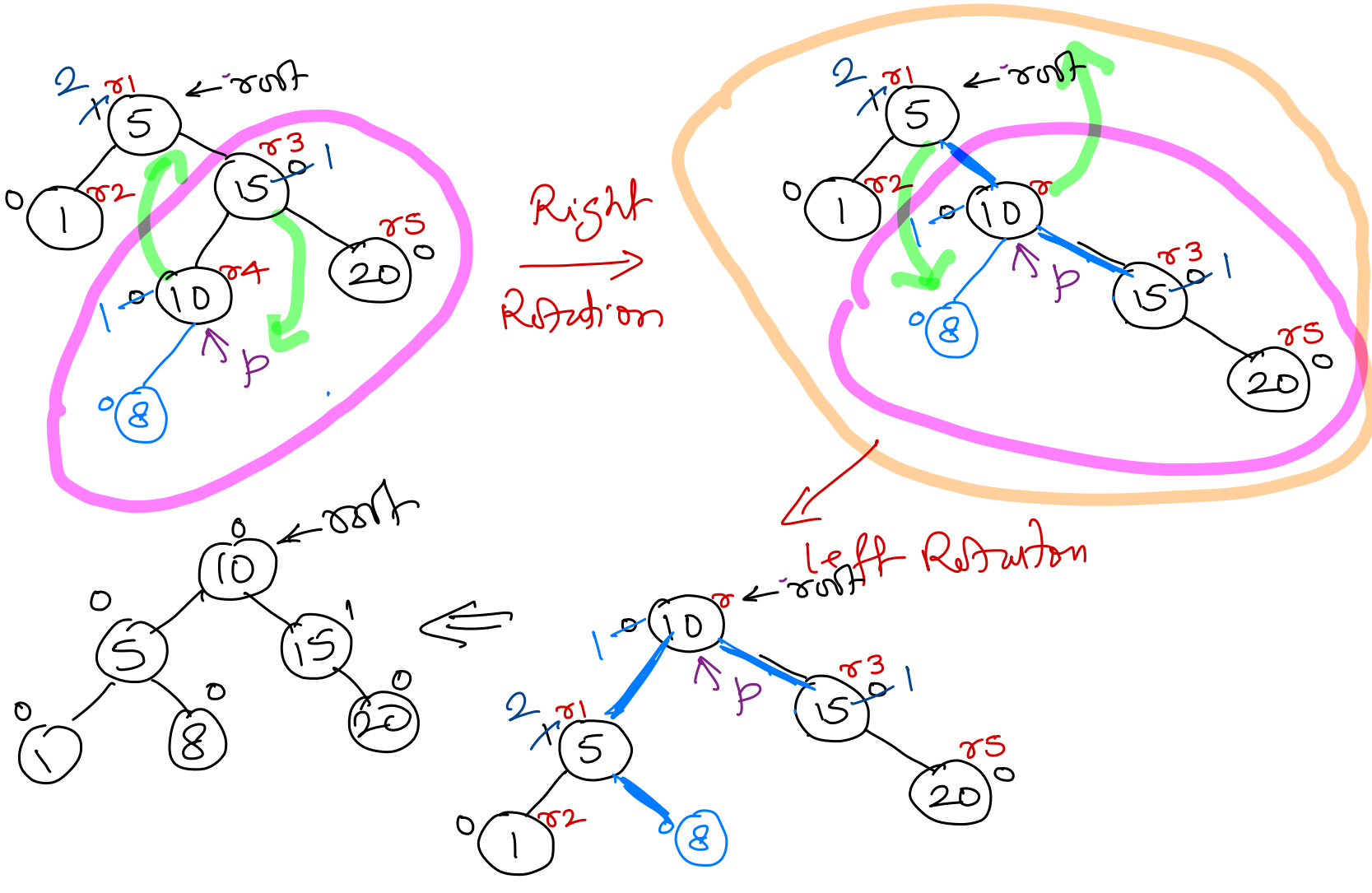
insert(8)



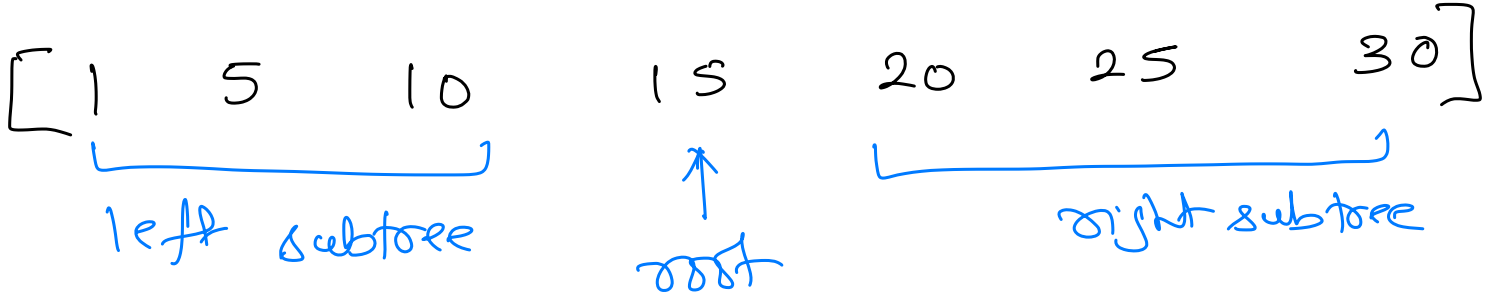


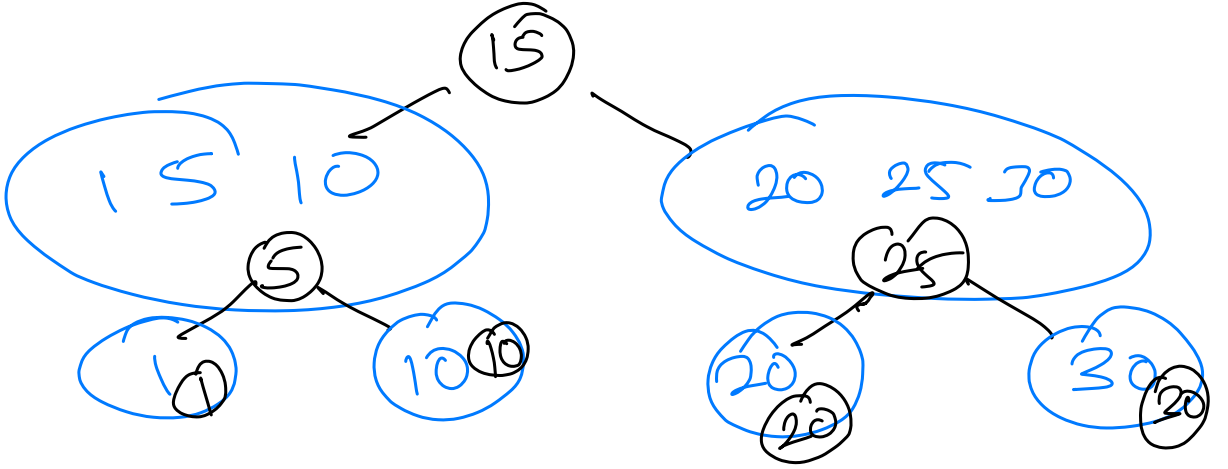
RL imbalance

Right Left

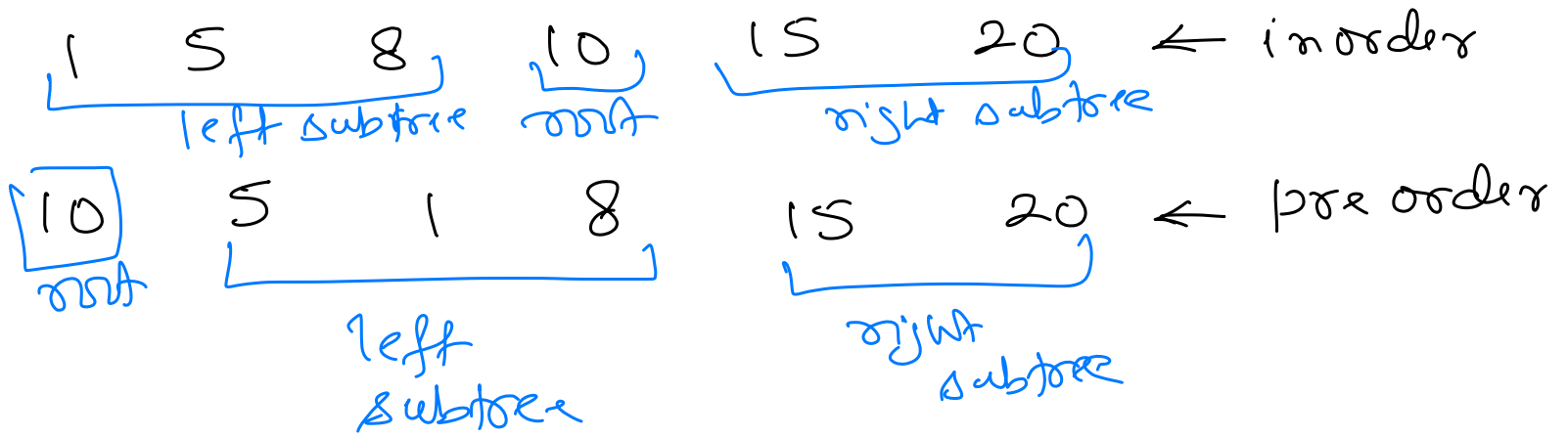

Right Rotation
Left Rotation
root
p
r1
r2
r3
r4
r5

Given a list of sorted elements, create a balanced BST.

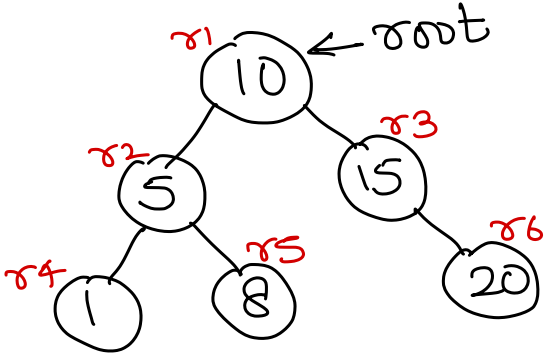


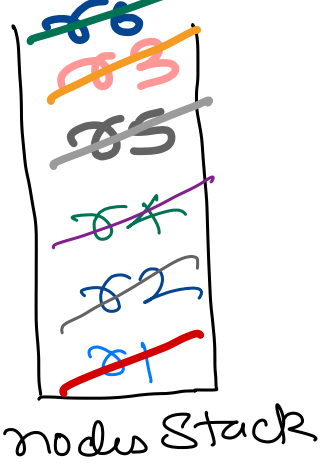


Given two traversals of a BINARY TREE, create the original tree back.



Iterative inorder traversal





current → r1 r2 r4 null r4 null r2 r5
null r5 null r1 r3 null r3 r6
null r6 null

O/P: 1 5 8 10 15 20

Iterative Inorder

- current = root
- while (current != null || (! nodesStack.empty()))
  - while (current != null)
    - Push current on nodesStack
    - current = current . left Child
  - current = Pop a node from nodesStack
  - Process current node.
  - current = current . right Child

Time complexity = O(n)

Space complexity = O(log n) ← Used by nodesStack.

Time complexity = $O(n)$

Space complexity = ~~$O(n)$~~

$= O(\log n)$ ← taken by recursive call on system stack.

InOrder(root)
- if (root is empty) then
  - Stop.
- If (root node's left child exists) then
  - InOrder(root's left child).
- Process root node's data.
- If (root node's right child exists) then
  - InOrder(root's right child).
- Stop.

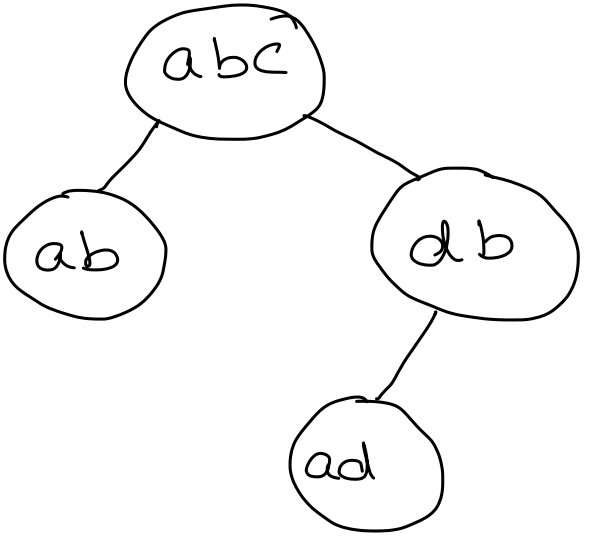
Space Complexity

Extra space taken by algorithm.

↓

to store additional data.

Dictionary of words : abc db ab ad



Using BST

$w \rightarrow$ length of a word.

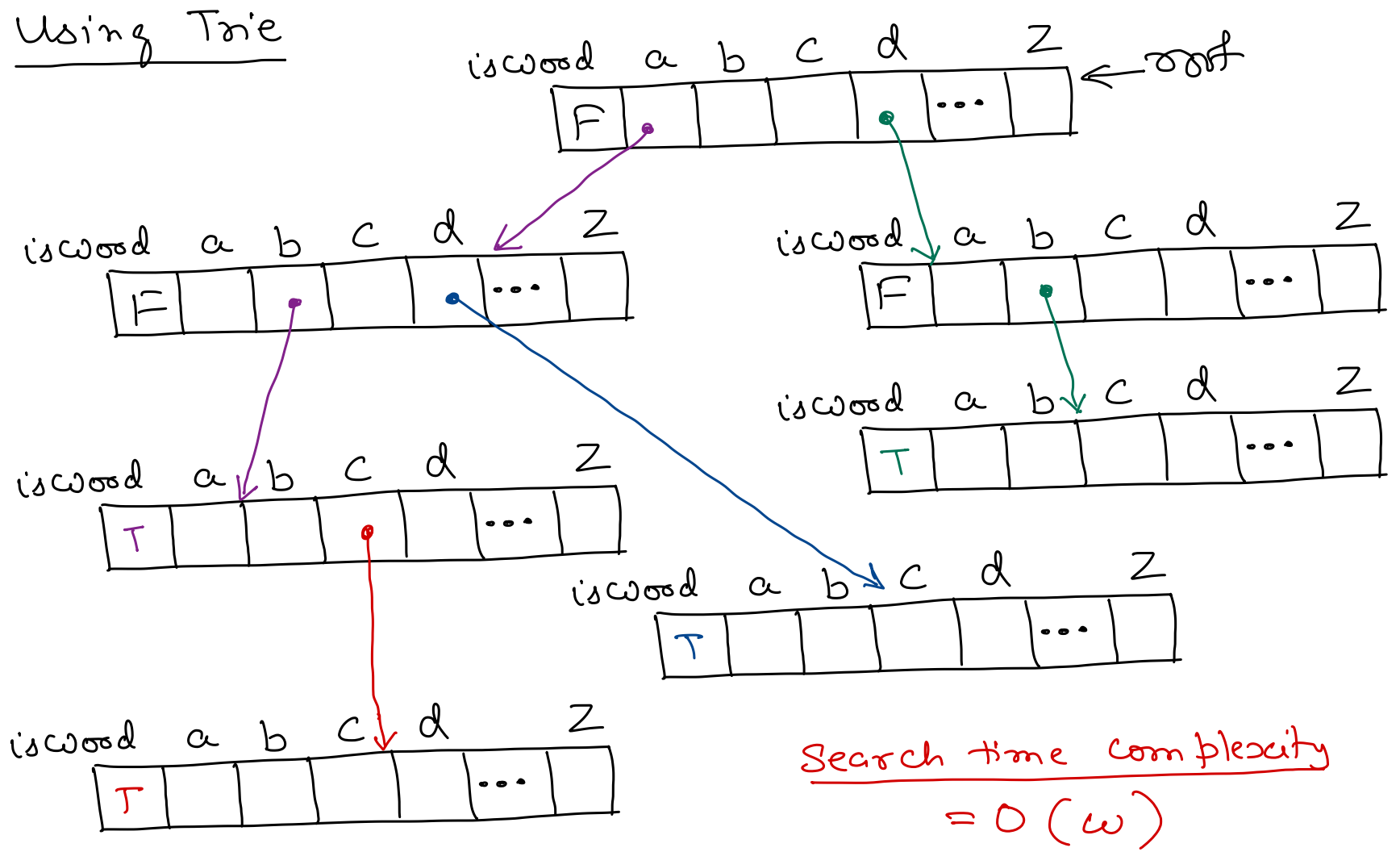
Search time complexity

$O(w \log_2 n)$

---

ab abc ad db

Using Trie

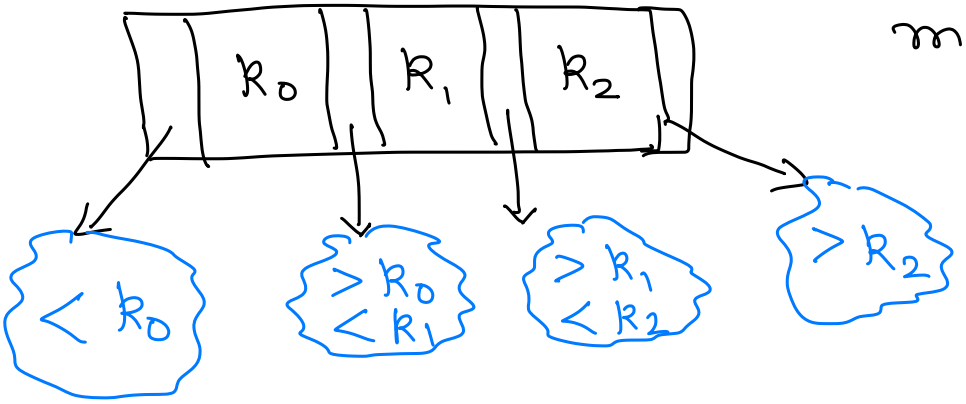


Search time complexity

$= O(w)$

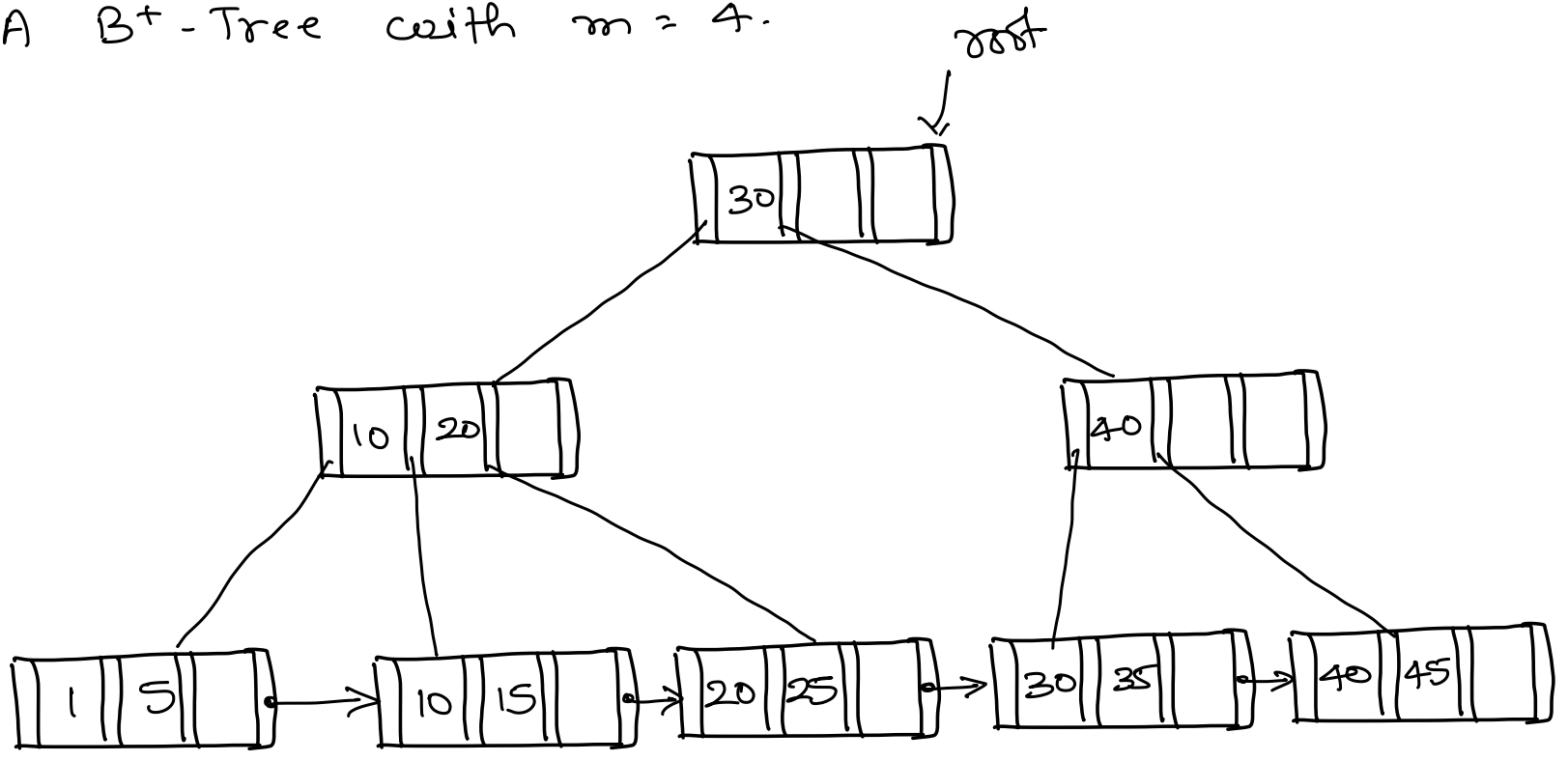
m-way Search Tree → B-Tree family.

B-Tree of order m, each node has at most m children.



$m = 4$.

A B^+-Tree with $m = 4$.



Searching → In linear data structures.

Linear Search

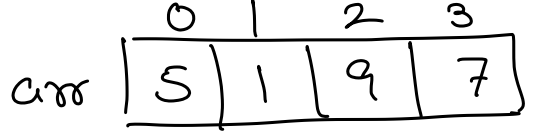
find element by checking every possible value, one by one.

Binary Search

find elements in ordered set of data.

find elements in unordered set of data.

Linear Search: we check each element one by one, to find the required value.



n = 4

```
boolean linearSearch(int[] arr, int value){
    for(int i=0; i<arr.length; ++i){
        if(arr[i]==value){
            return true;
        }
    }
    return false;
}
```

Linear

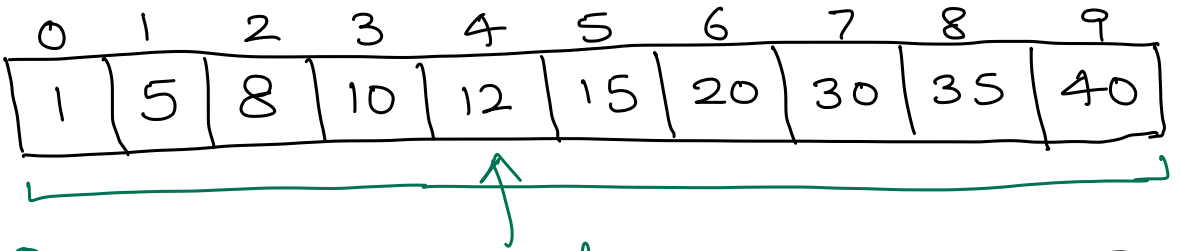
Time = O(n)

Space = O(1)

Constant

Binary Search

→ Require data to be sorted / arranged in a well defined order.



low → 0

high → 9

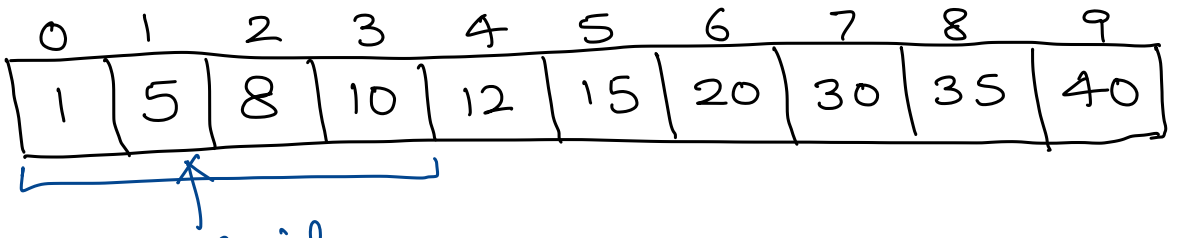
mid → $\frac{(low + high)}{2} = \frac{(0+9)}{2} = 4$

binary Search(8)

if element < arr[mid] then

element can be found before mid element

(set high end of range to before mid)



low → 0

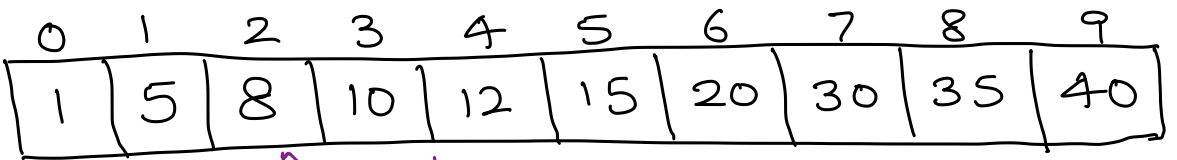
high → 3

mid → $\frac{0+3}{2} = 1$

if arr[mid] < element then

element can be found after mid element

(set low end of range to after mid)



low → 2

high → 3

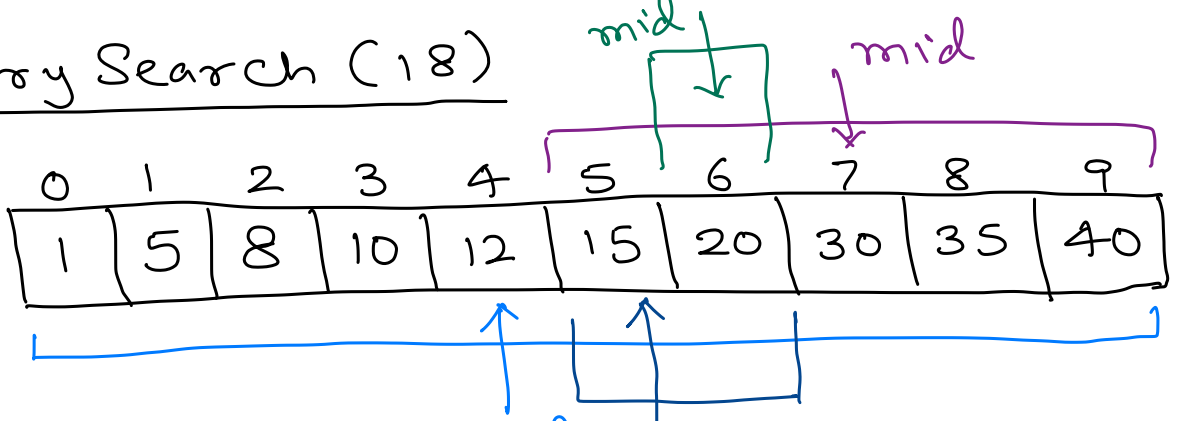
mid → $\frac{2+3}{2} = 2$

arr[mid] = value

⇓

FOUND

Binary Search (18)



low → ~~0~~ ~~5~~

high → ~~9~~ ~~6~~ 5

mid → 4 7 5 6

→ high < low

⇓

NOT FOUND

binarySearch (arr, value)

```
low = 0
high = arr.length - 1
while ( low <= high)
    → mid = (low + high)/2
    → if (arr[mid] = value)
        ↳ return true
    ↳ if (value < arr[mid])
        ↳ high = mid - 1
    else
        ↳ low = mid + 1
return false.
```

$low + \frac{(high - low)}{2}$

Time complexity $= O(\log_2 n)$

Space complexity $= O(1)$