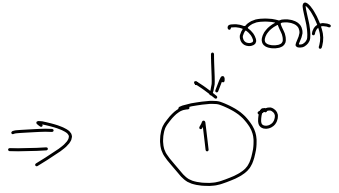
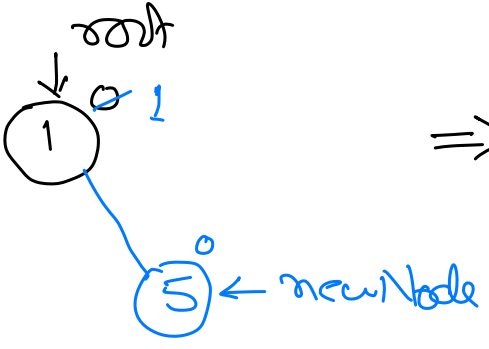
AVL Tree

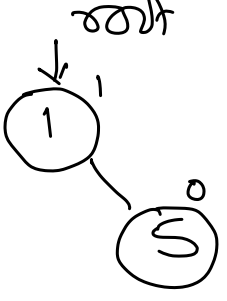
insert (1)

root = empty



insert (5)





⇒ Balance Factor (BF) for each node

$= h_L - h_R$

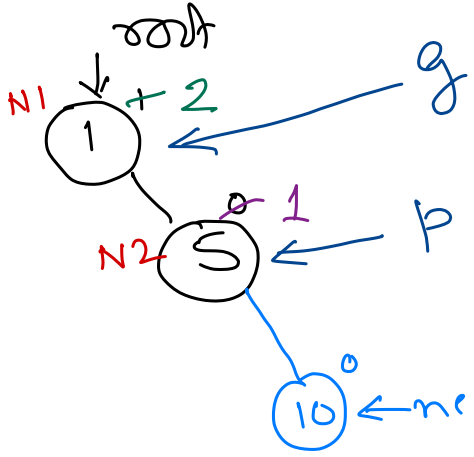
h_L ← Height of Left subtree

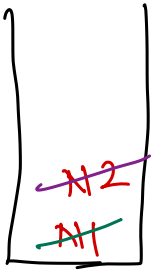
h_R ← Height of Right subtree

$-1 \leq BF \leq +1$

$|BF| \leq +1$

insert (10)





Left Rotation.

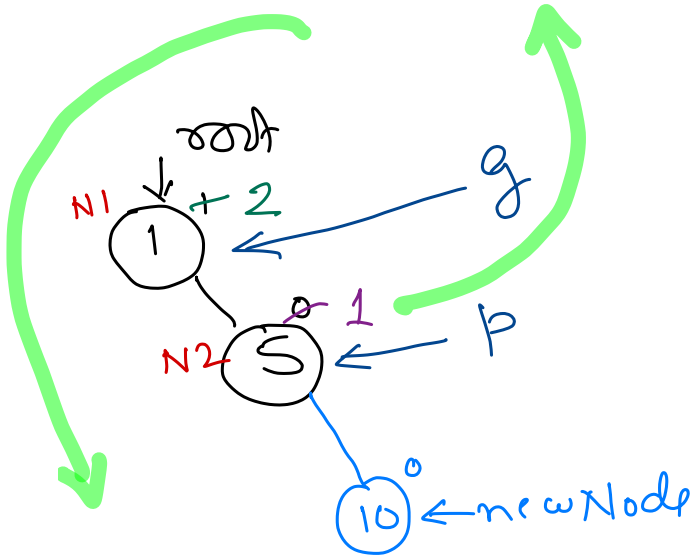
RR - imbalance

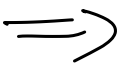
we took two steps from g towards newNode.

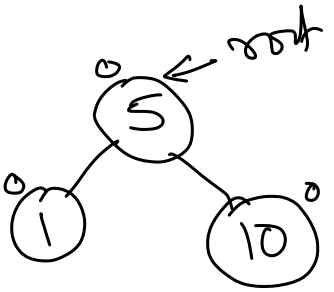
g → nearest parent of newNode having incorrect balance factor.

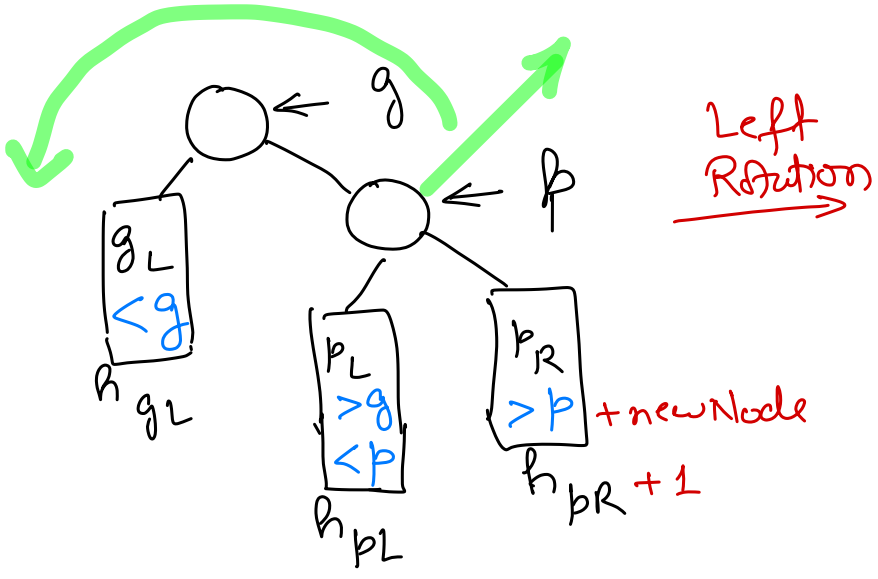
p → right child of g if doing Left rotation.

Left Rotation

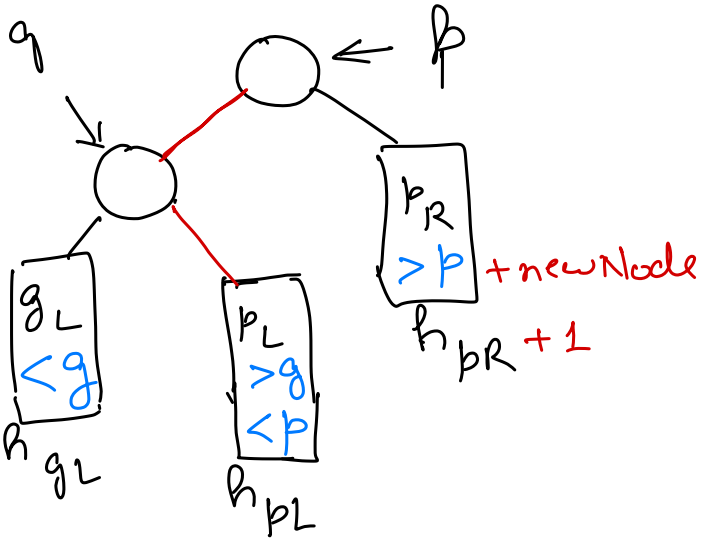

root
N1
1
+ 2
g
- 1
p
N2
5
10
newNode



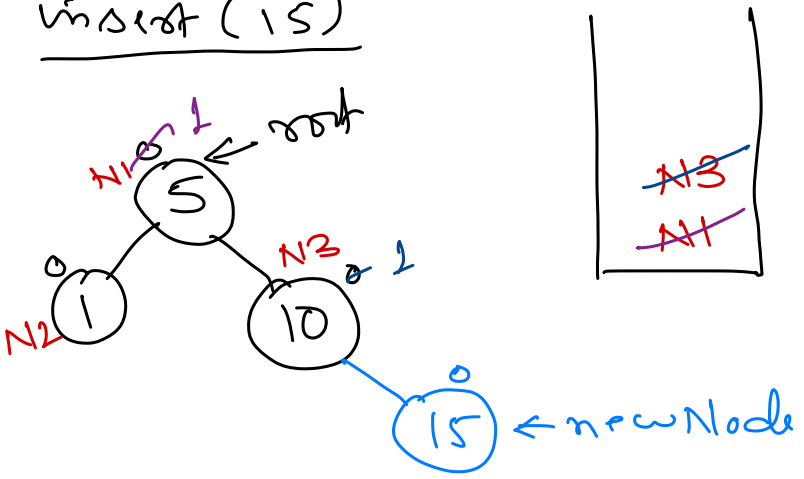

root
5
1
10

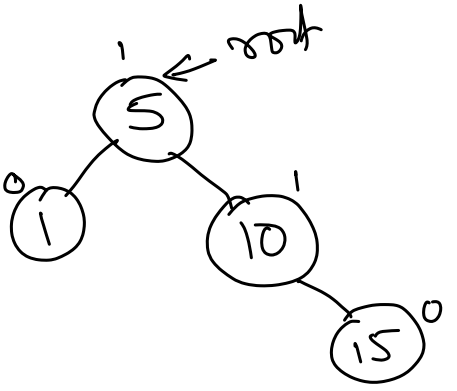


Left Rotation

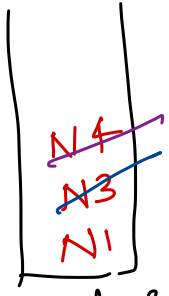


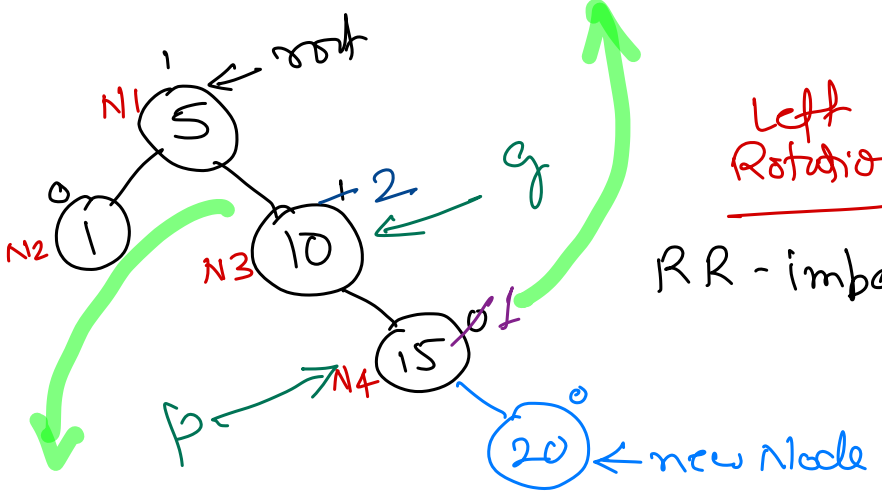
insert (15)





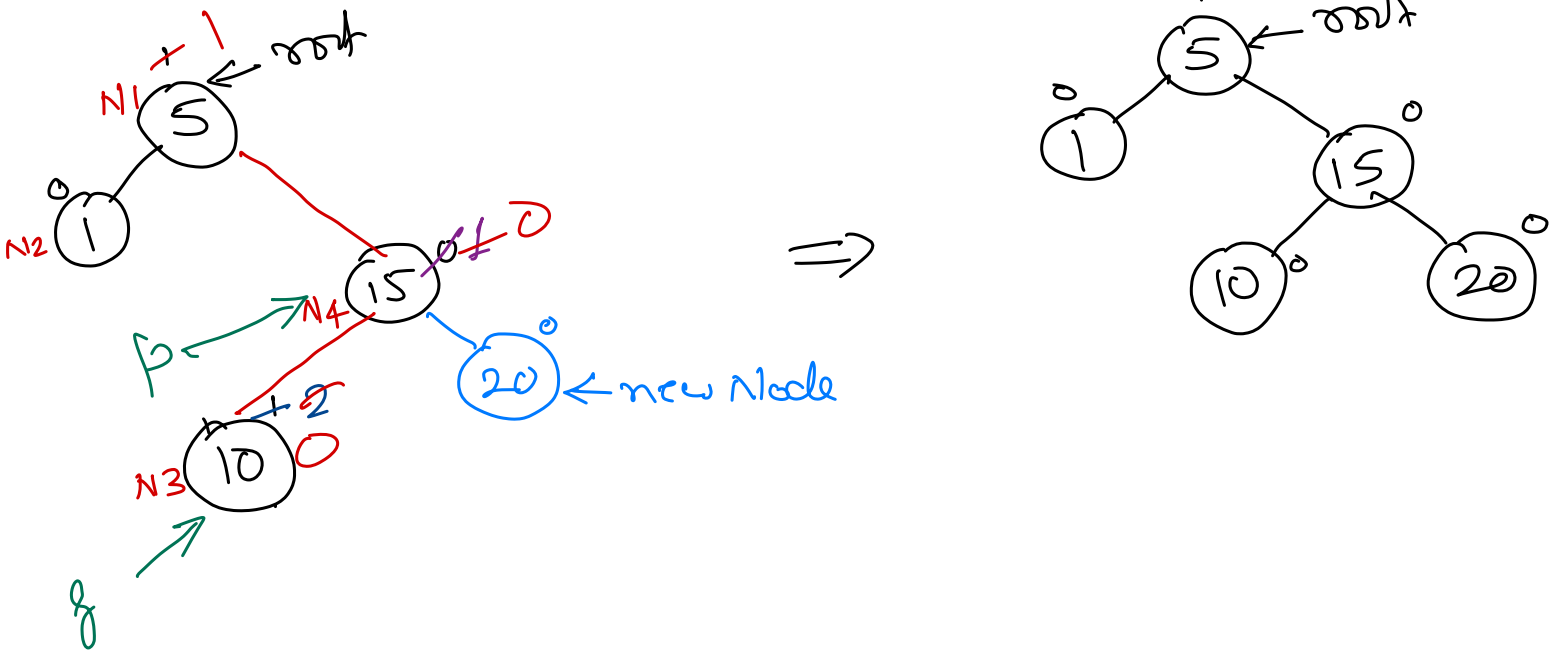
insert (20)

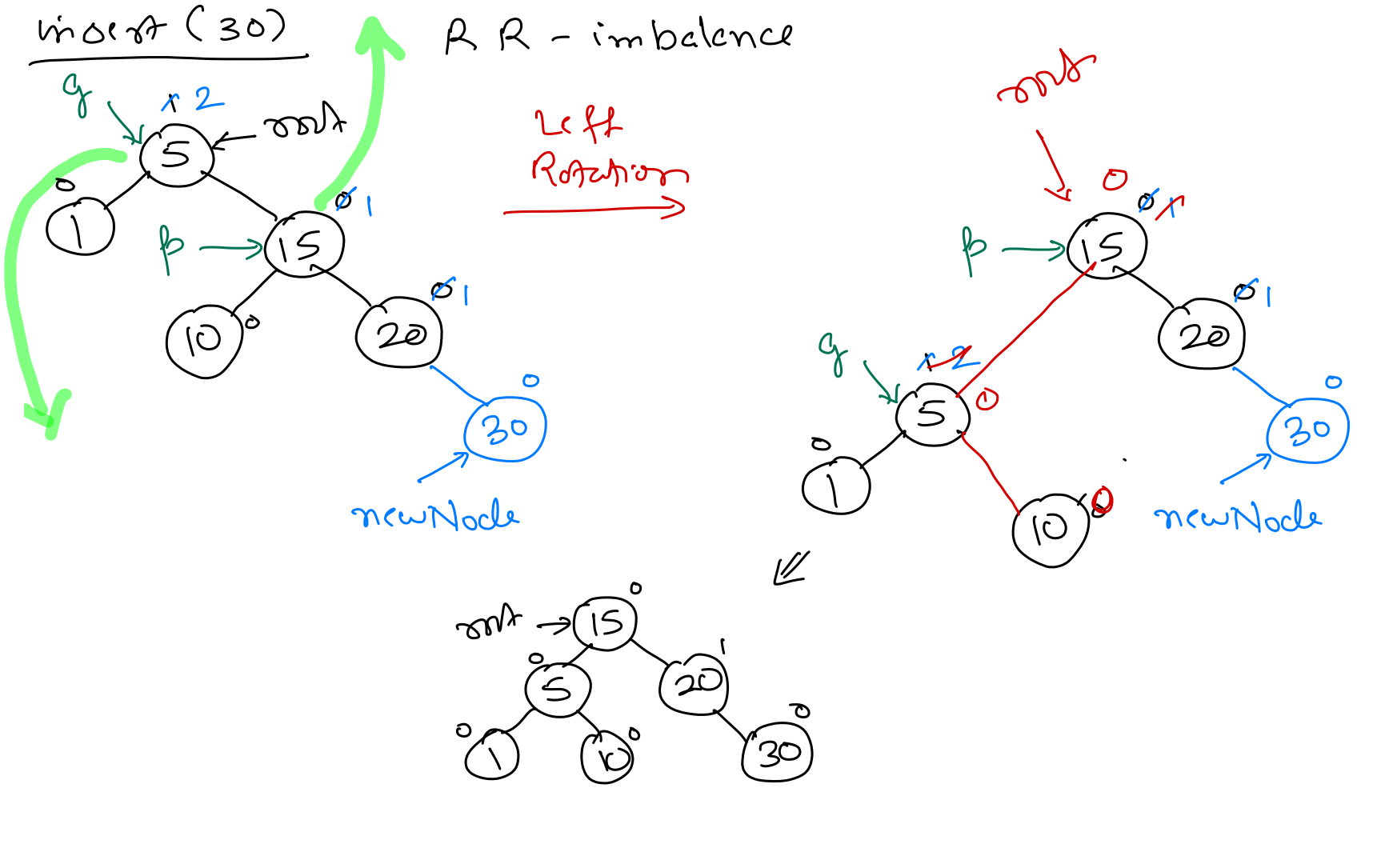




Left Rotation

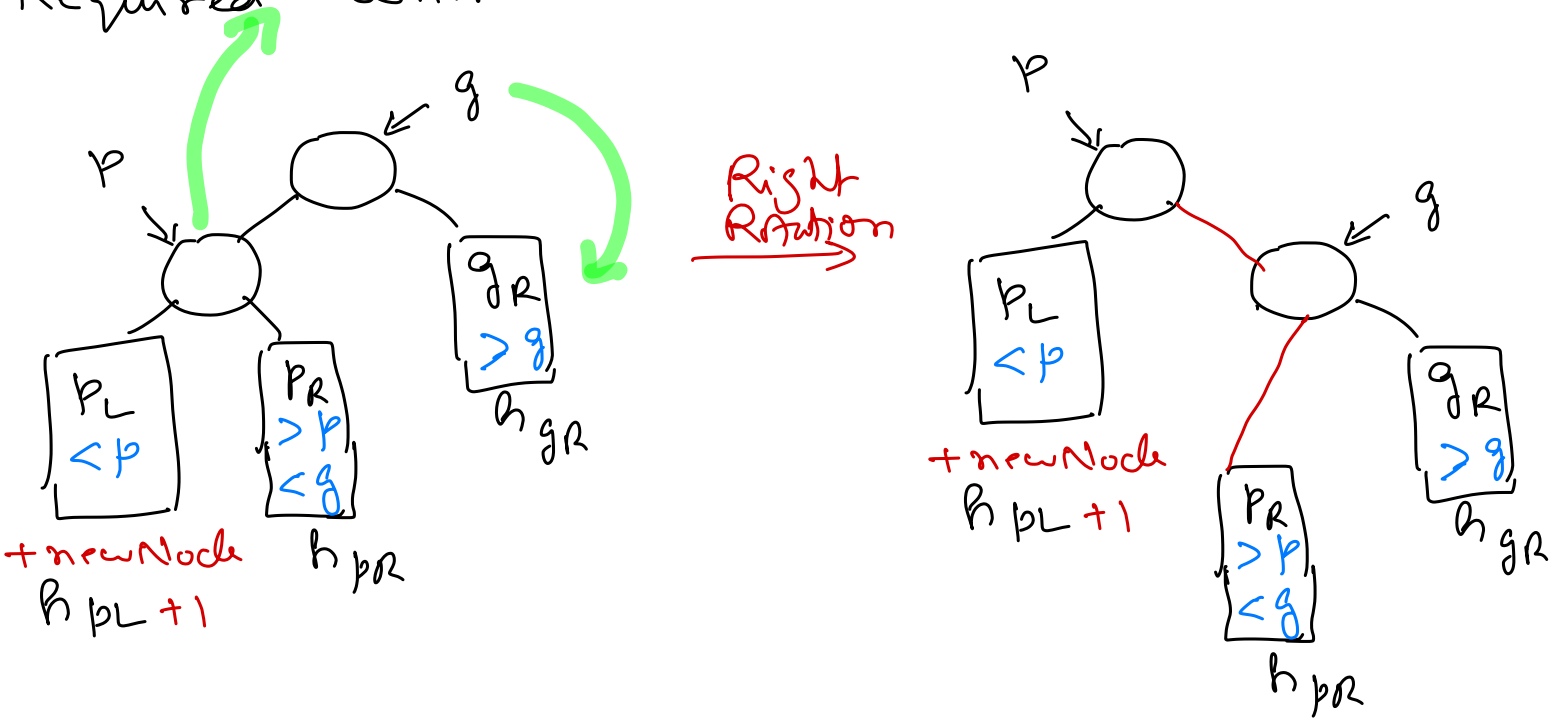
RR - imbalance


+1
root
N1
5
0
N2
1
15
0 1 0
N4
p
20
0
new Node
+2
0
N3
10
q


insert (30)
RR - imbalence
Left Rotation
g
+2
root
5
1
p
15
10
20
30
newNode
root
0
15
p
5
g
+2
1
10
20
30
newNode
15
5
20
1
10
30

Right Rotation → Mirror image of left Rotation.

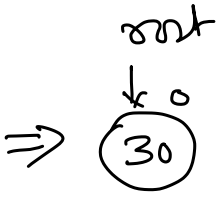
Required when LL- imbalance.



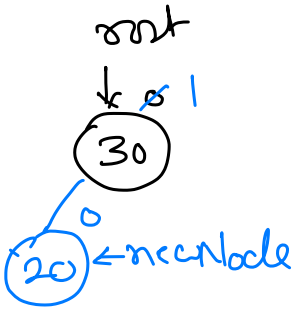
insert 30 20 15 10 5 1

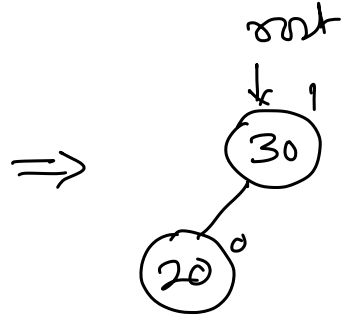
insert (30)

root → empty

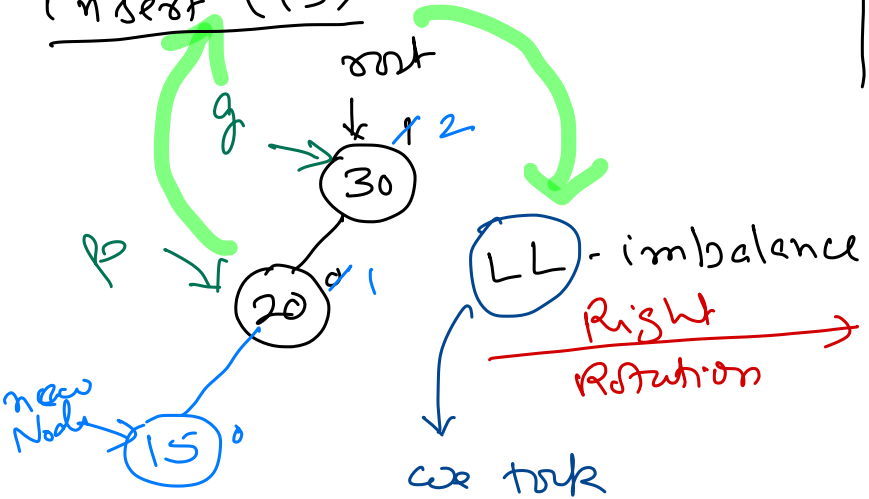


insert (20)

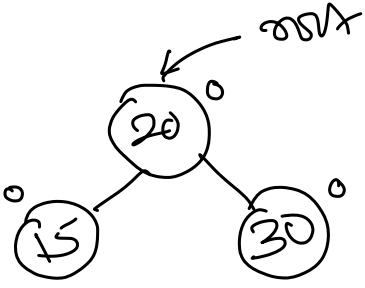




insert (15)



we took two steps from g towards newNode.

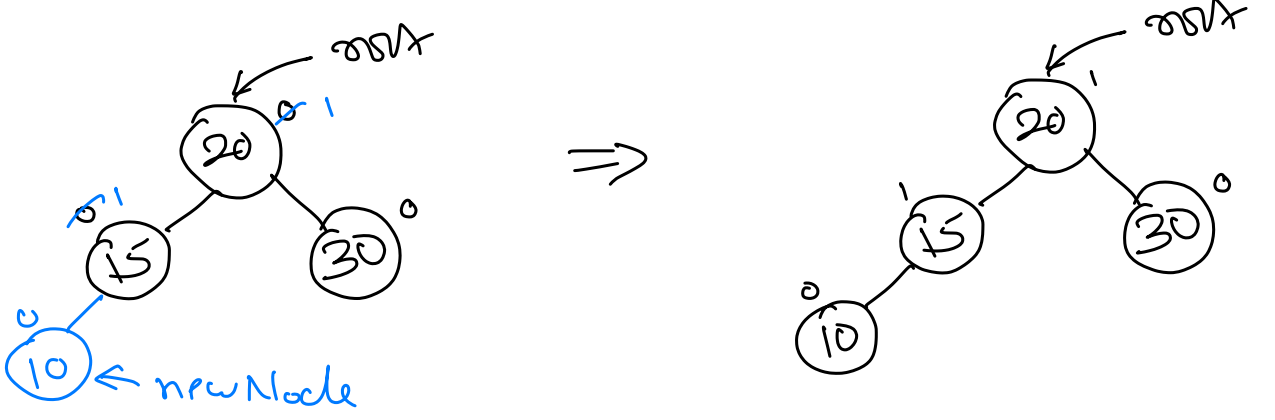


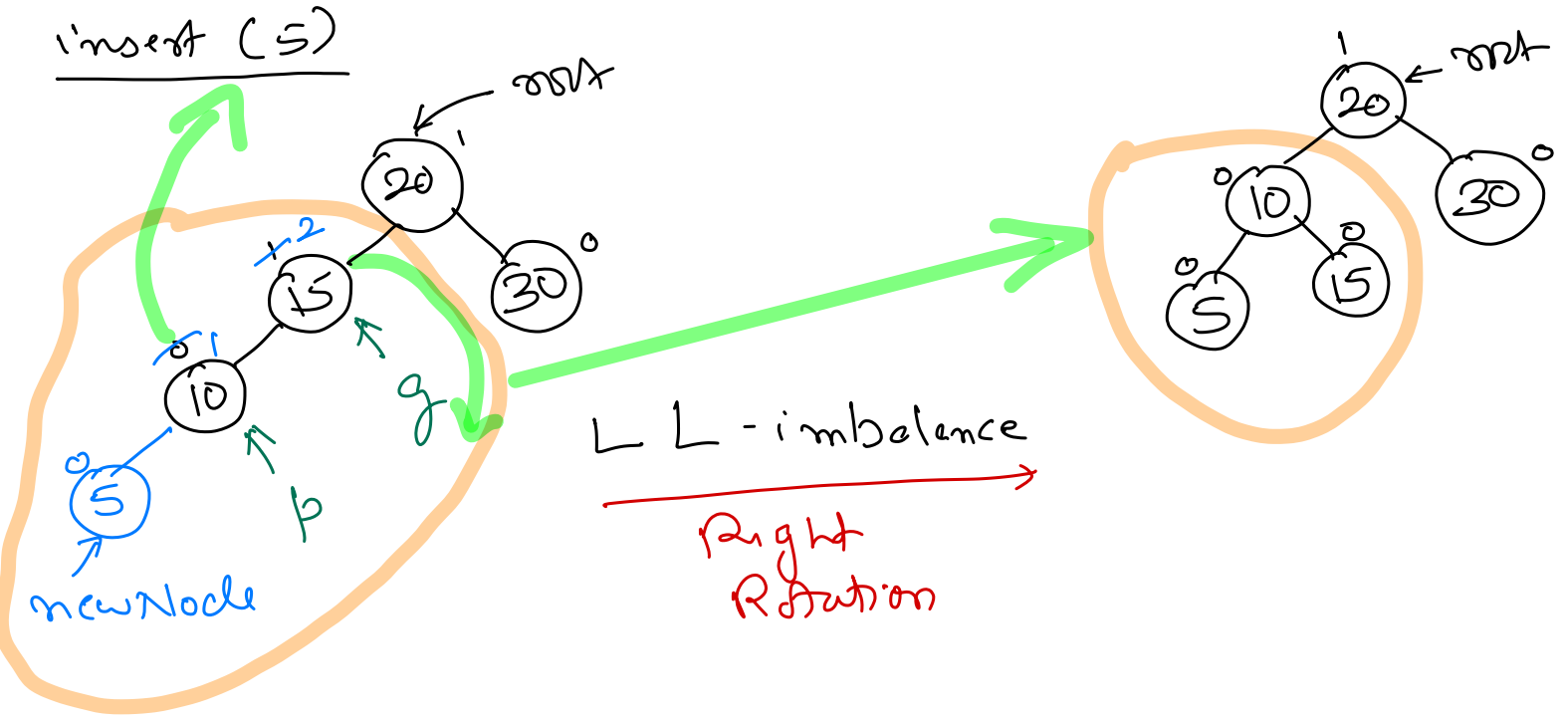
g → nearest parent of newNode having incorrect balance factor.

p → Left child of g if doing Right rotation.

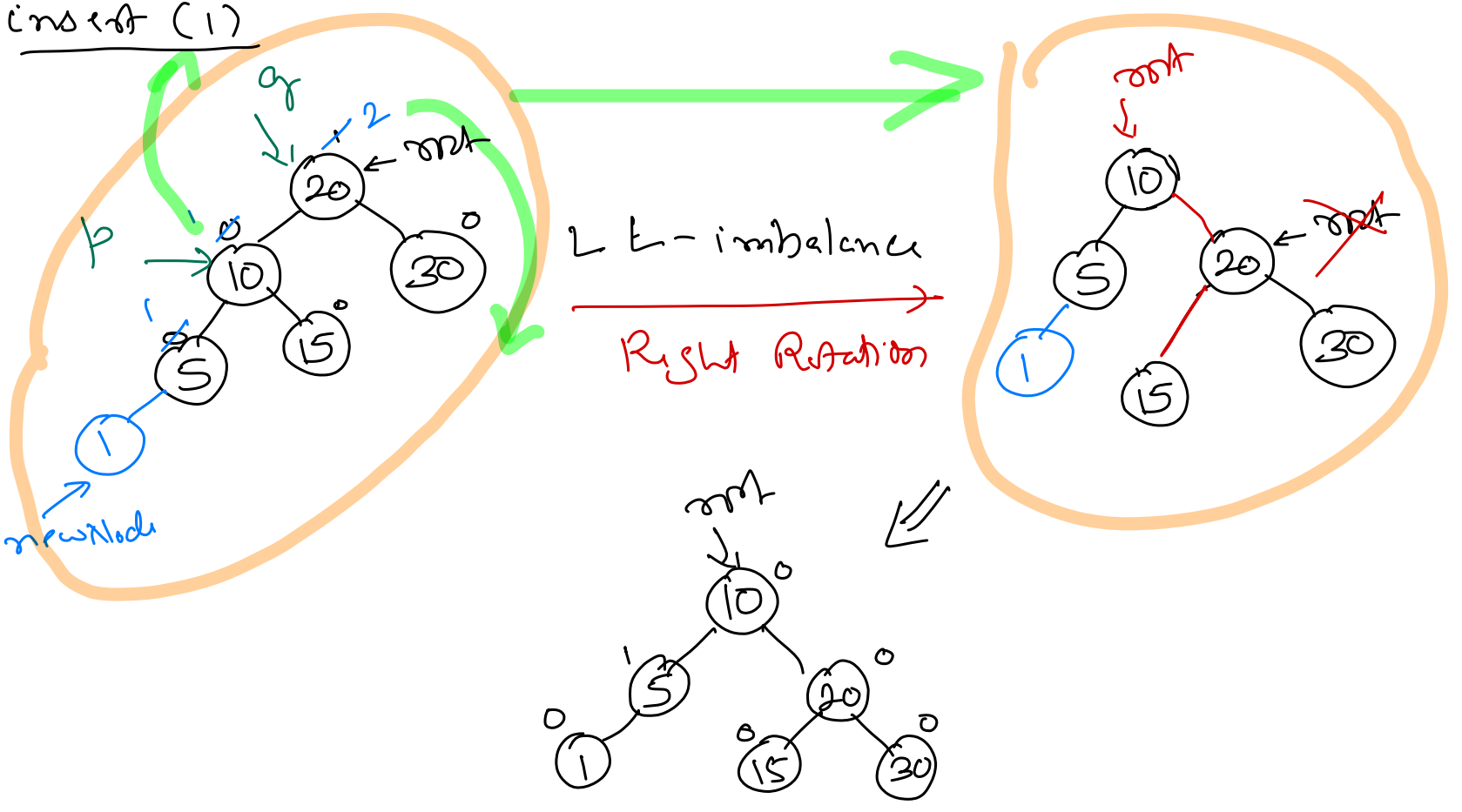
---

insert (10)

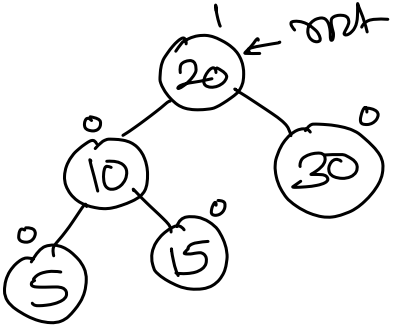



insert (5)
root
20
15
30
10
5
g
p
newNode
LL - imbalance
Right Rotation
root
20
10
30
5
15

insert (1)

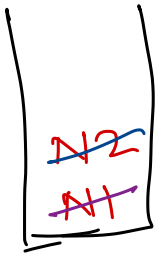

gp
2
root
20
p
10
30
1
15
5
1
newNode
LL-imbalance
Right Rotation
root
10
5
20
1
15
30
root
10
5
20
1
15
30

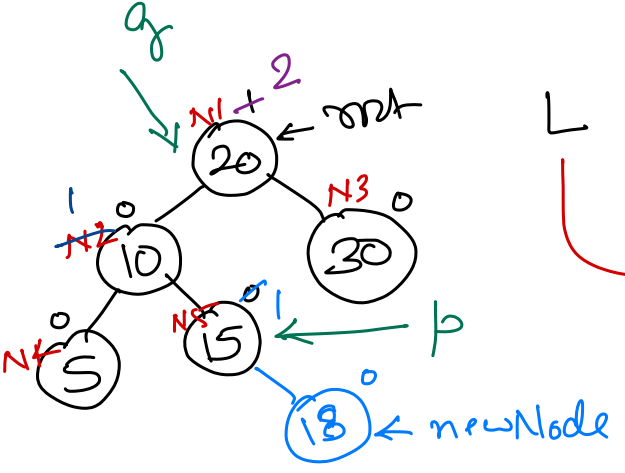
Insert → 30 20 15 10 5



p will be grand child of g in path towards newNode

insert (18)



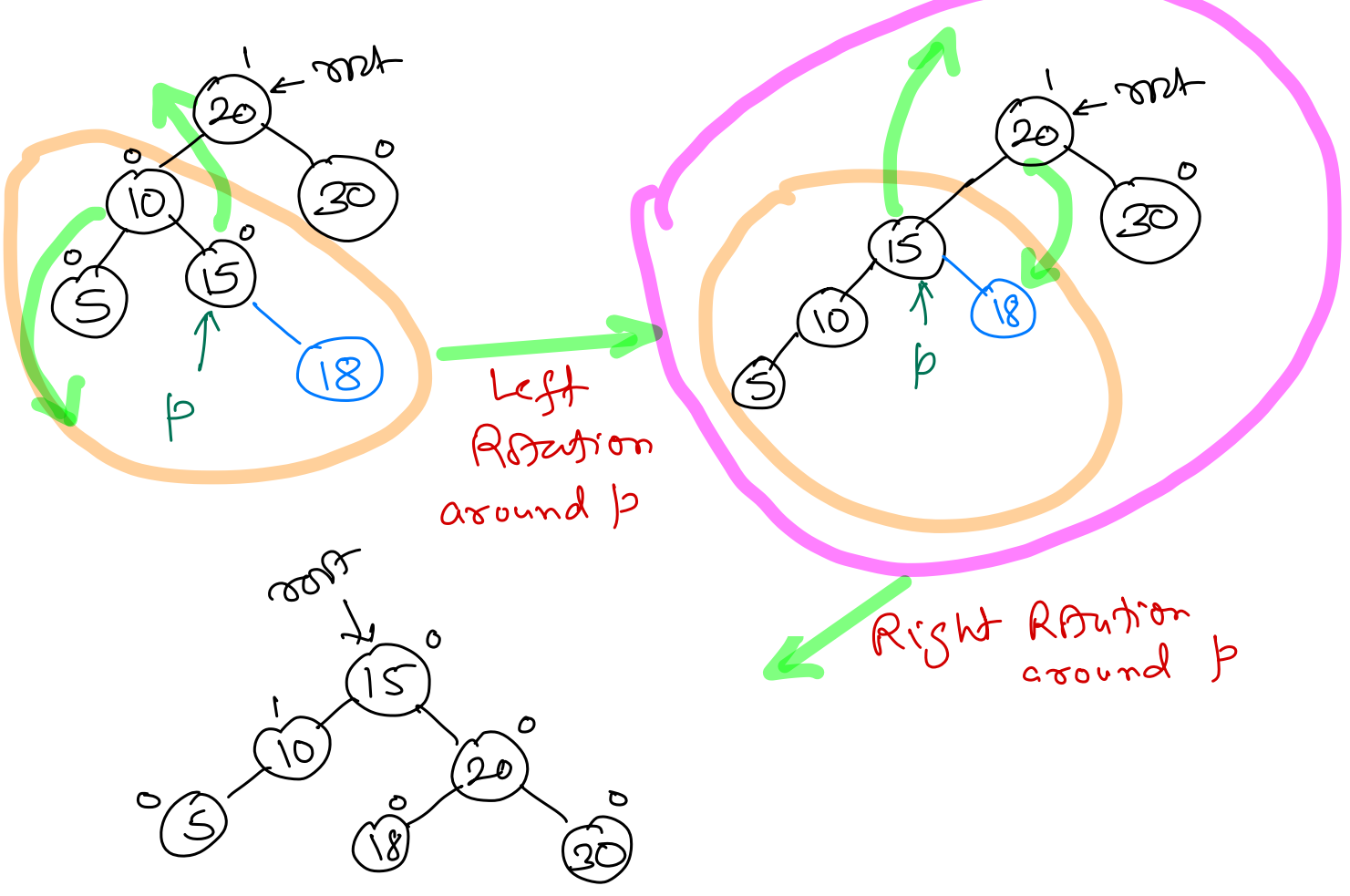


L R - imbalance

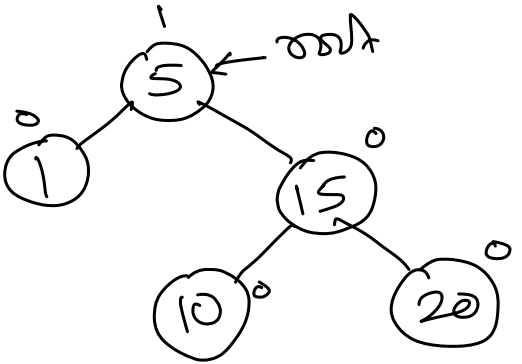
↓ 2 Rotation

→ Left Rotation

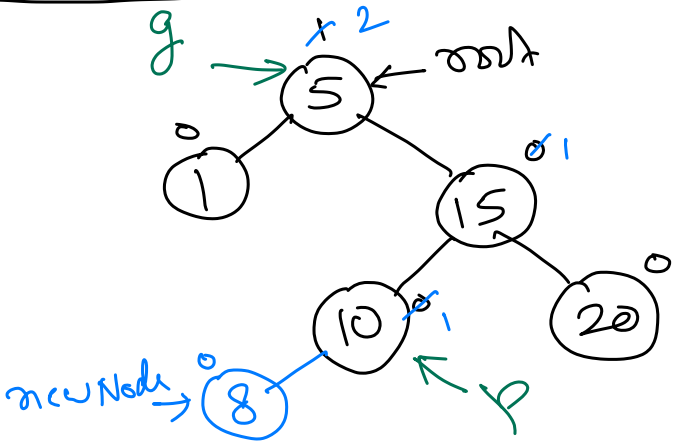
→ Right Rotation


root
20
10
30
5
15
18
p
Left Rotation around p
root
20
15
30
10
18
5
p
Right Rotation around p
root
15
10
20
5
18
30

Insert 1 5 10 15 20



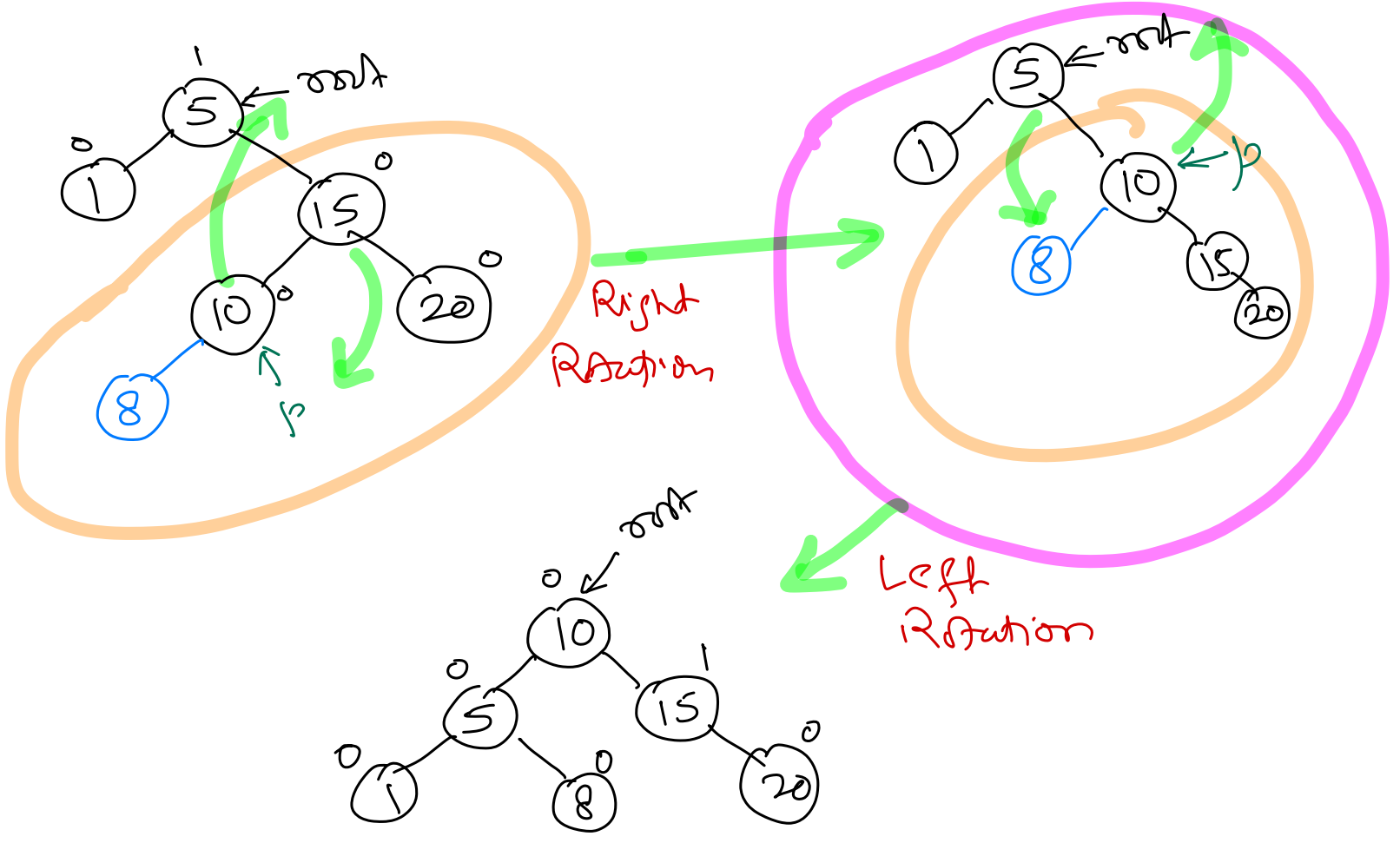
insert (8)



RL - imbalance ⇒ 2 rotations

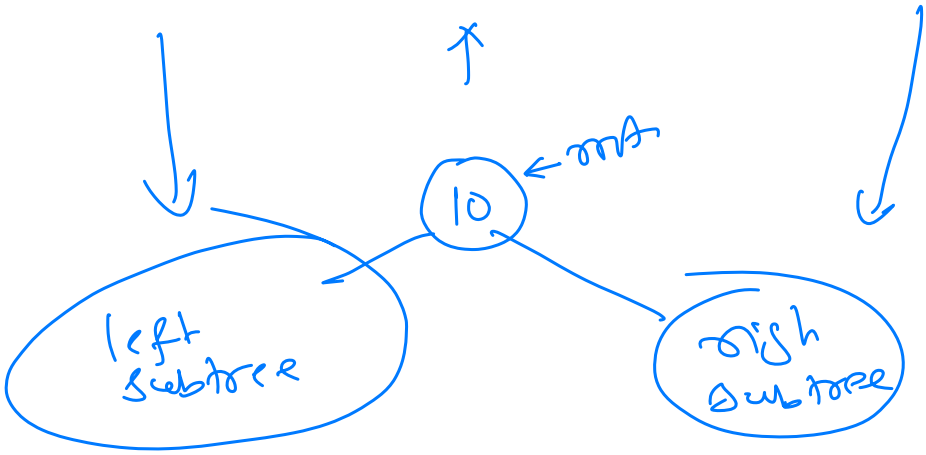
- Right Rotation around p
- Left Rotation around p

p = grand child of g in the path towards newNode.


root
5
1
15
10
20
8
y
Right
Rotation
root
5
1
10
y
8
15
20
Left
Rotation
root
10
5
15
1
8
20

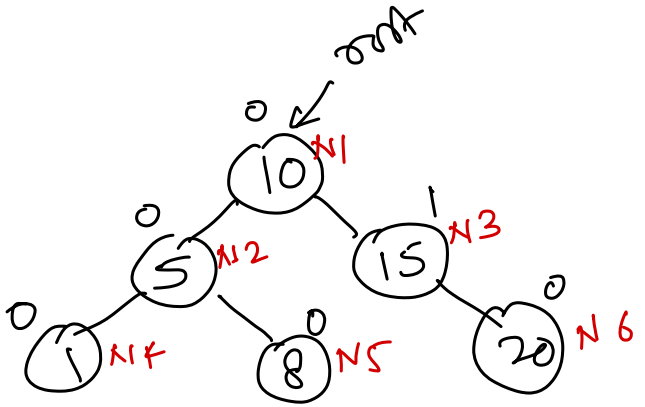
Sorted elements to balanced BST.

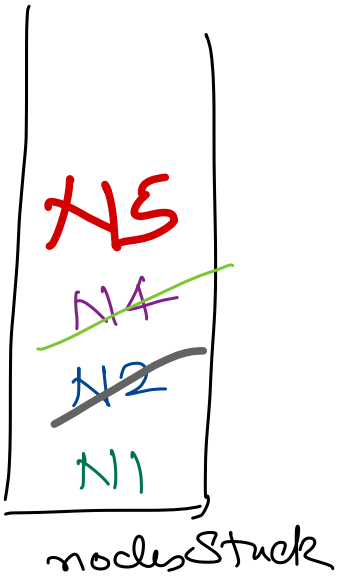
[[1 5 8] 10 [15 20 30]]



Iterative Depth first tree traversal

Iterative Inorder





o/p 1 5

current → N1 N2 N4 N4 empty empty N2 N5 empty

current = root

while ((current != null) || (! stack. is empty())

- while (current is not empty)
  - push current on stack
  - current = current. left
- current = pop node from stack
- process current
- current = current. right

$\log_2 n$

Iterative inorder ?

h = height of Binary Tree

Time complexity = O(n)

Space complexity = O(h)

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.

Recursive inorder

Time = O(n)

Space = O(h)

Space used on system stack to make recursive calls.

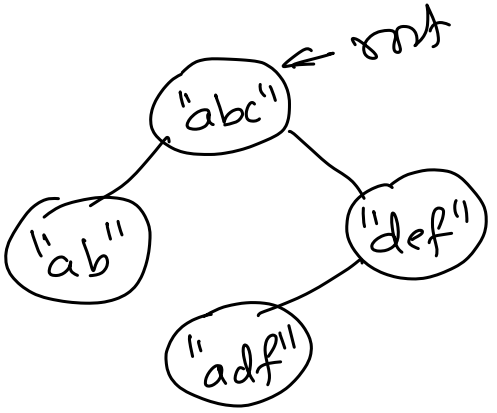
Space Complexity

Extra space required by algorithm to process data.

O(1) ⇒ Constant if extra space required is independent of input / data size.

Dictionary of words

"abc" "def" "adf" "ab"



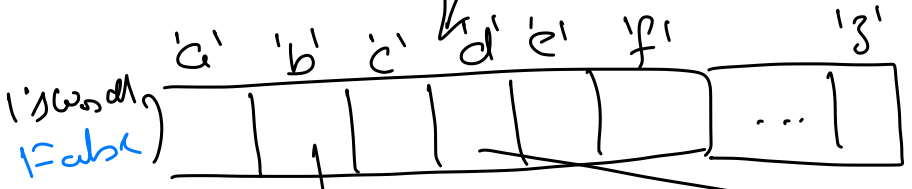
Search time complexity

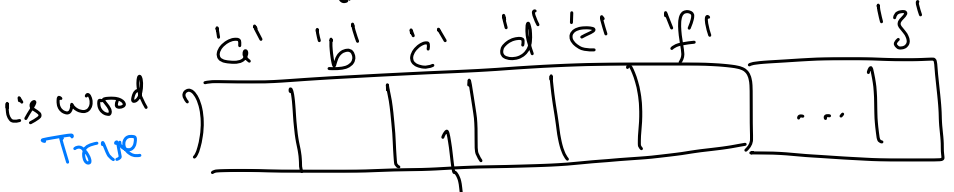
$= \omega \log_2 n$ ← total number of words.

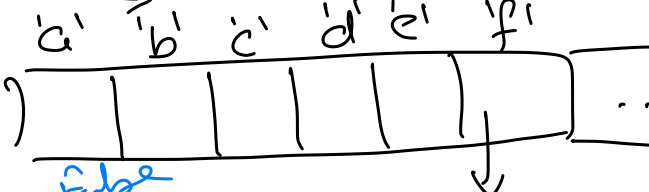
↑ word length

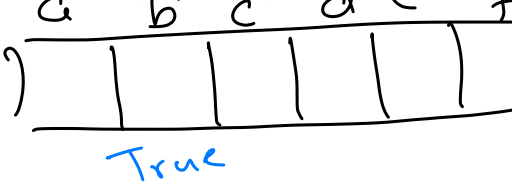
Trie

Search
O(w)





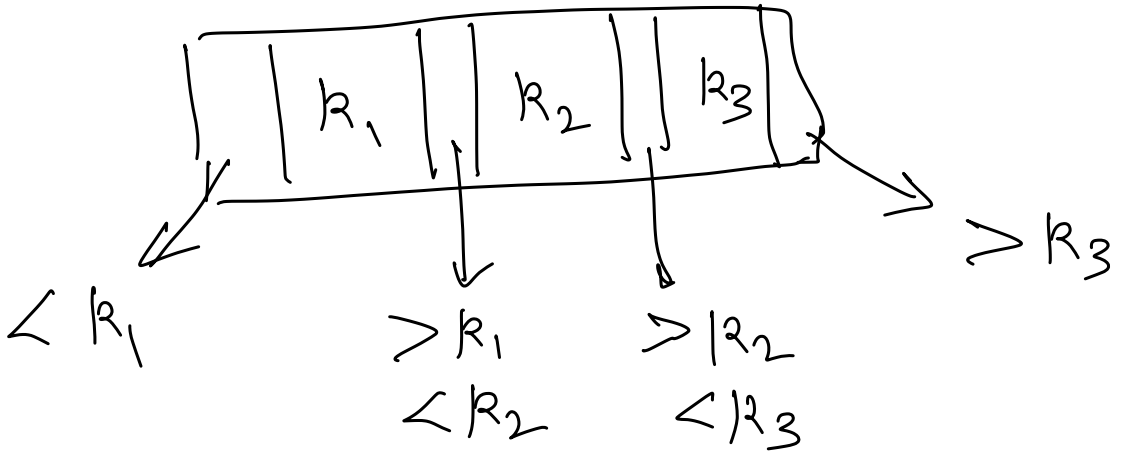


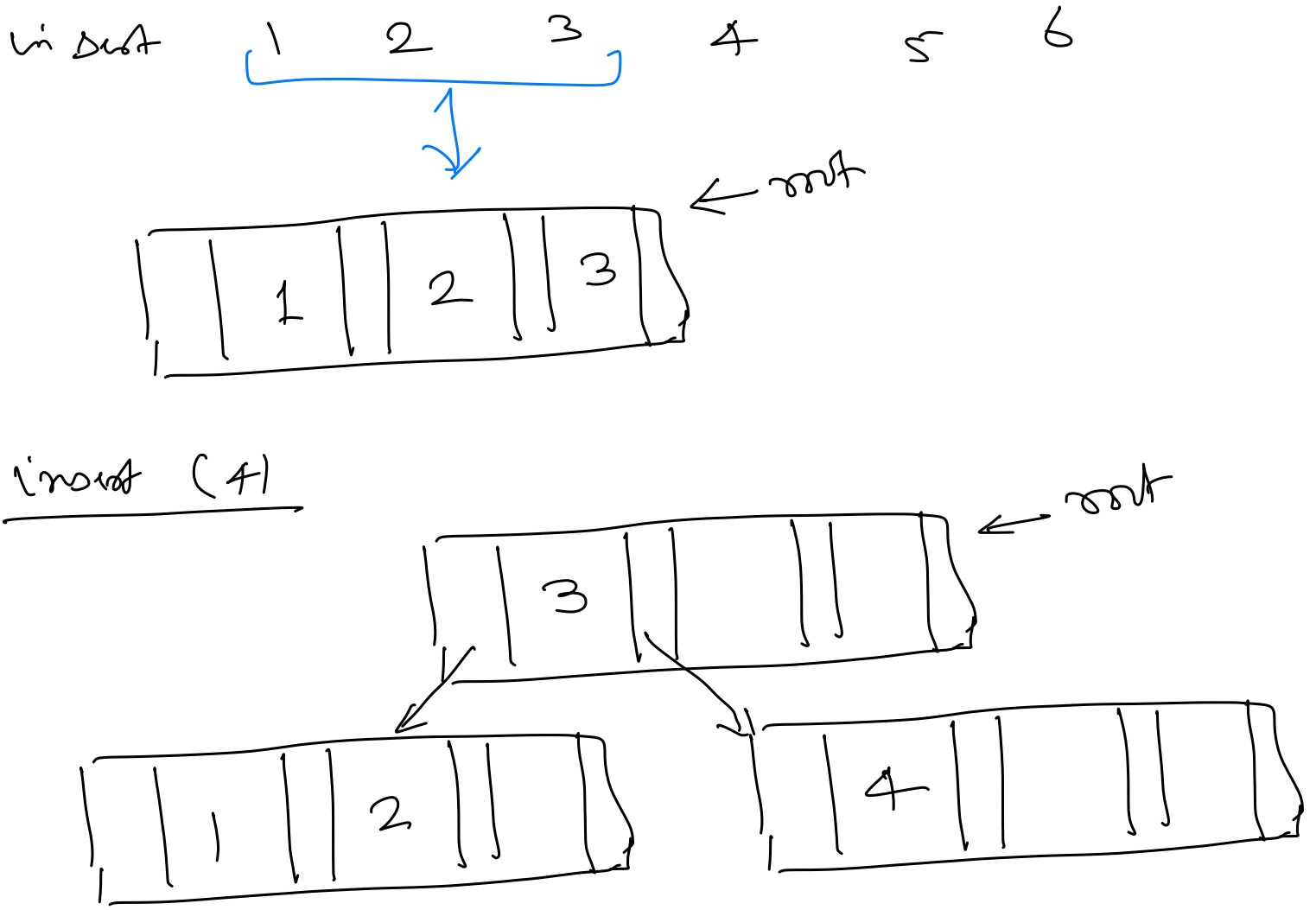


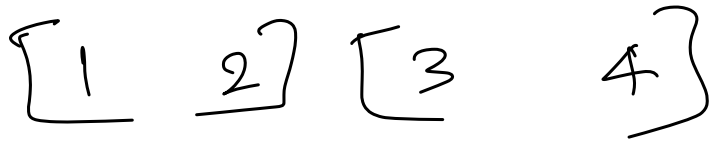
m- way Search Tree / B- Tree

B- Tree
- Search tree of order (k)
  - Number of child pointers in each node.
- typically a node is of size of disc block.

$k = 4$




insert
1
2
3
4
5
6
root
1
2
3
insert (4)
root
3
1
2
4

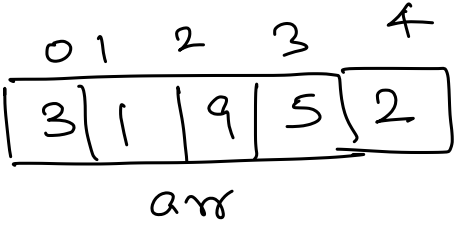


Height = $\log_R^n$

Searching → In linear data structure.

- Linear Search ⇓ Find element in unordered set of data.
- Binary Search ⇓ Find element in ordered set of data.

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



n = 5

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

Linear

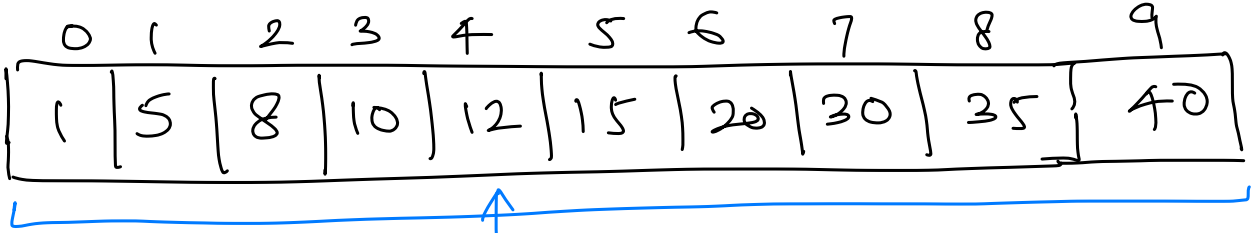
Time => O(n)

Space => O(1)

constant

Binary Search

Binary Search require data to be sorted/ arranged in a well difined order.



binarySearch (8)

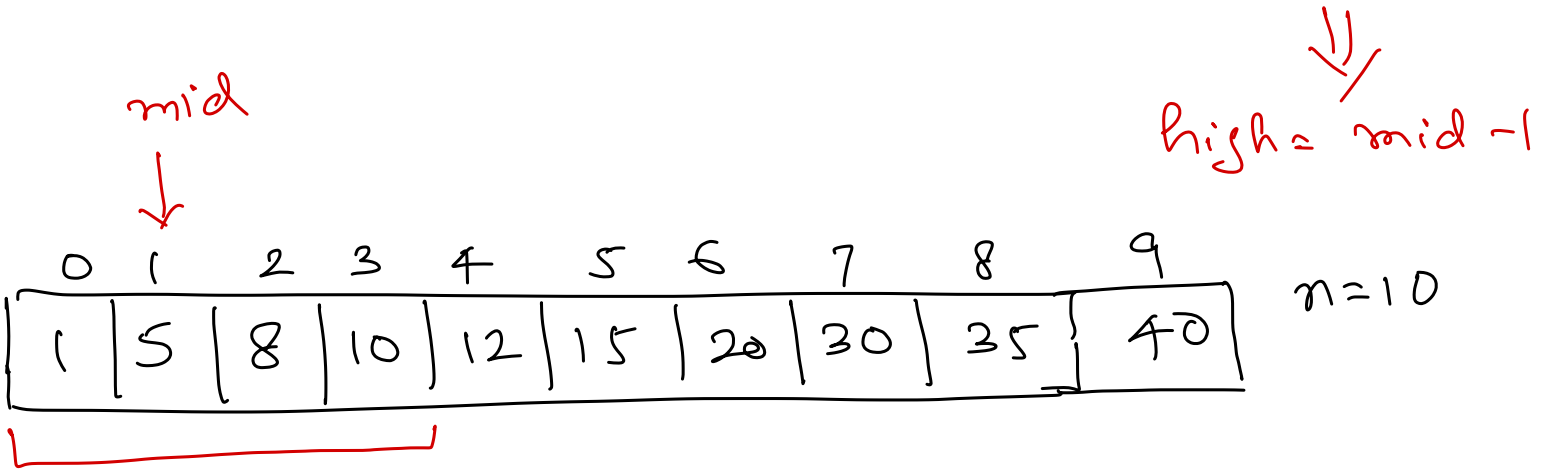
low → 0

high → ~~9~~ 3

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

as element < arr [mid]

⇒ element can be found before or to left of mid.



low → ~~0~~ 2

high → ~~9~~ 3

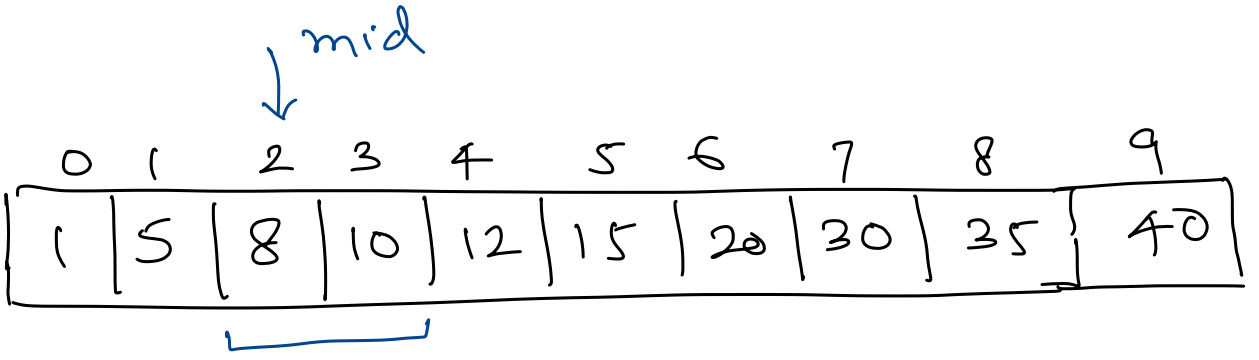
$mid = \frac{0+3}{2} = 1$

as element > arr[mid]

⇒ element can be found to right of mid

⇓

low = mid + 1



low → ~~0~~ 2

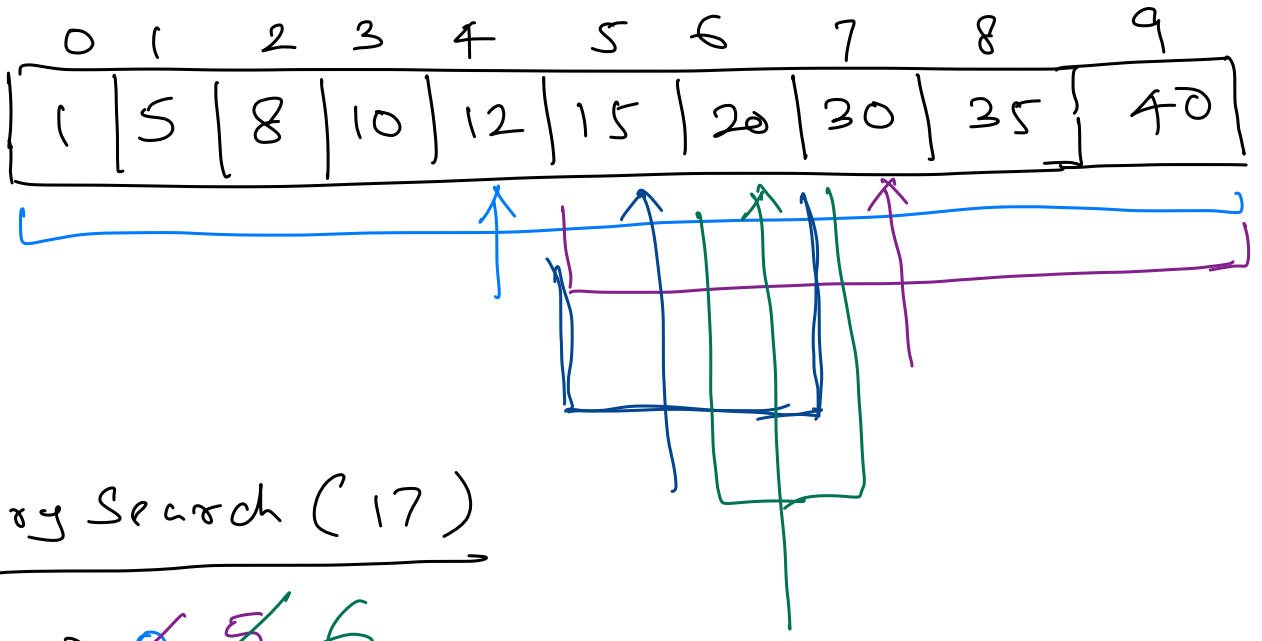
high → ~~9~~ 3

mid → $\frac{2+3}{2} = 2$

arr[mid] = element

⇓

FOUND



binary Search (17)

low → ~~0~~ ~~5~~ 6

high → ~~9~~ ~~8~~ 5

mid → ~~4~~ ~~7~~ ~~5~~ 6

high < low

⇓

NOT FOUND

Binary Search (arr, element)

- low = 0
- high = n-1
- while (low <= high)
  - mid = (low + high)/2 ⇒ low + (high-low)/2
  - if (arr[mid] = element) then
    - FOUND and STOP
  - if (element < arr[mid]) then
    - high = mid -1

else

- low = mid + 1

- NOT FOUND
- STOP.

Time ⇒ $O(\log_2 n)$

Space ⇒ $O(1)$

Iteration #	# of element in range
1 ⟶	n
2 ⟶	$\frac{n}{2}$
$\log_2 n$? ⟶	1

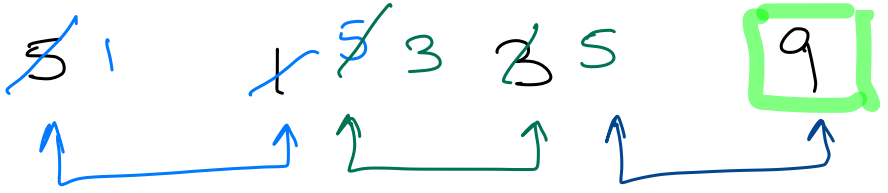
Sorting

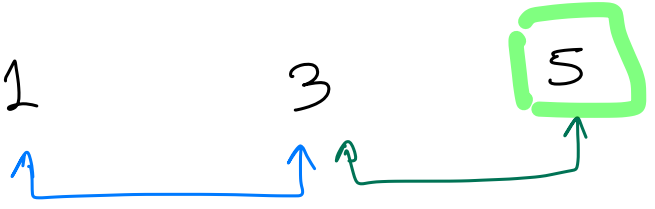
Arrange elements in a specific order

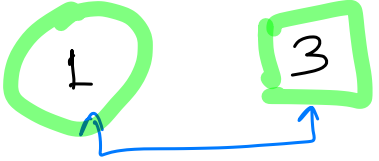
- Ascending order
  - Smaller to larger
- Descending order
  - larger to smaller.

Bubble Sort

Ascending order ⇒ in each pair of adjacent elements, left element < right element.
If not then swap the two elements.

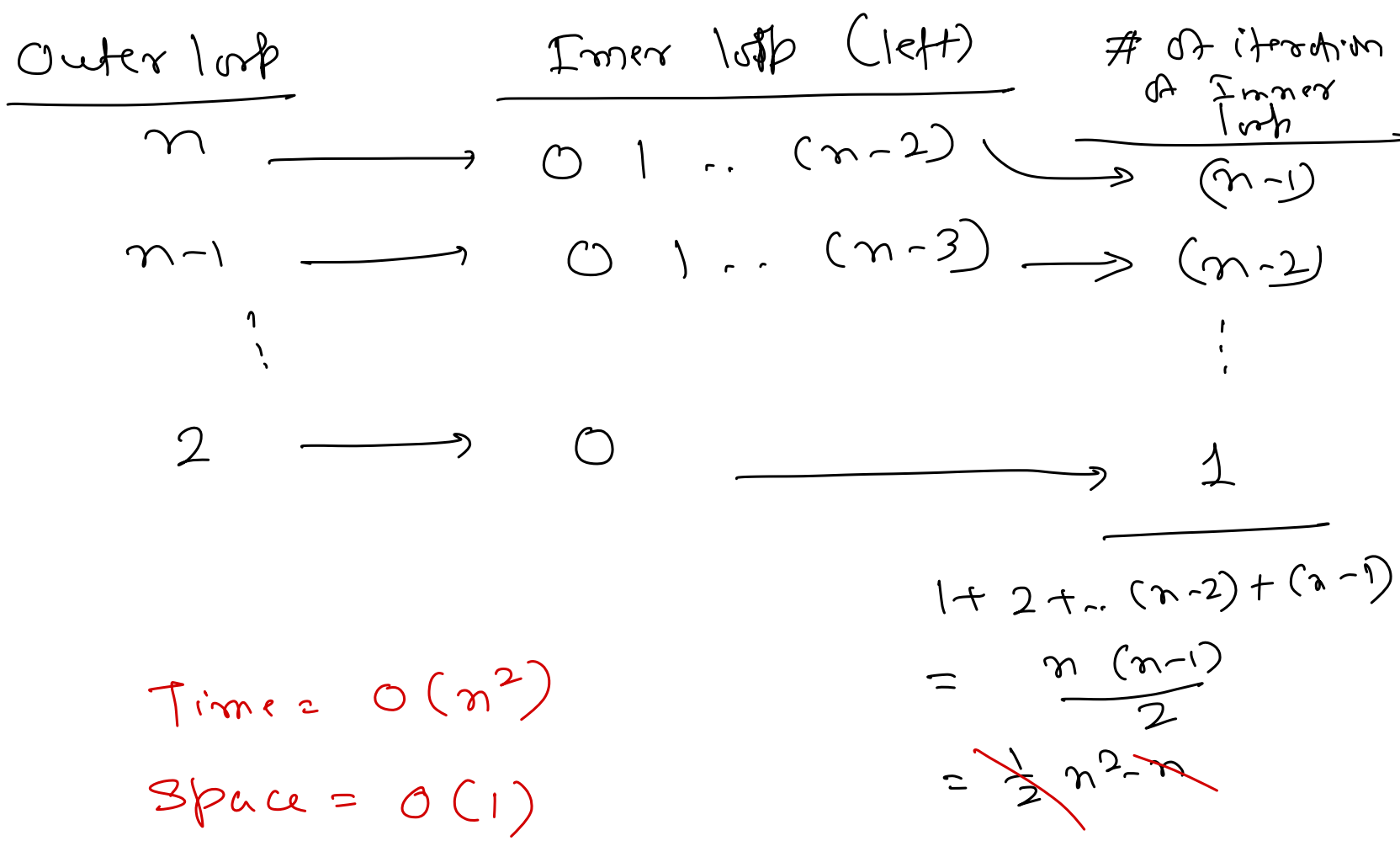






bubble Sort (arr)

- n = number of elements to sort.
- while (n > 1) do
  - left = 0
  - while (left < (n-1)) do
    - if (arr [left] > arr [left +1]) then
      - Swap elements at left and (left +1).
    - left = left +1.
  - n = n-1


Outer loop
Inner loop (left)
# of iteration of Inner loop
n
0 1 .. (n-2)
(n-1)
n-1
0 1 .. (n-3)
(n-2)
2
0
1
1+ 2+.. (n-2) + (n-1)
= n (n-1) / 2
= 1/2 n^2 - n
Time = O(n^2)
Space = O(1)

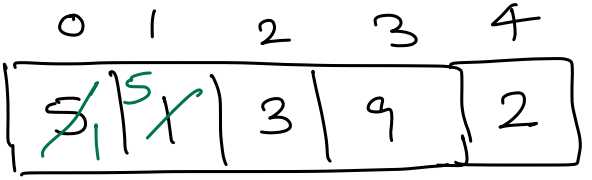
Selection Sort

n = 5

elementPos → ~~0~~ 1

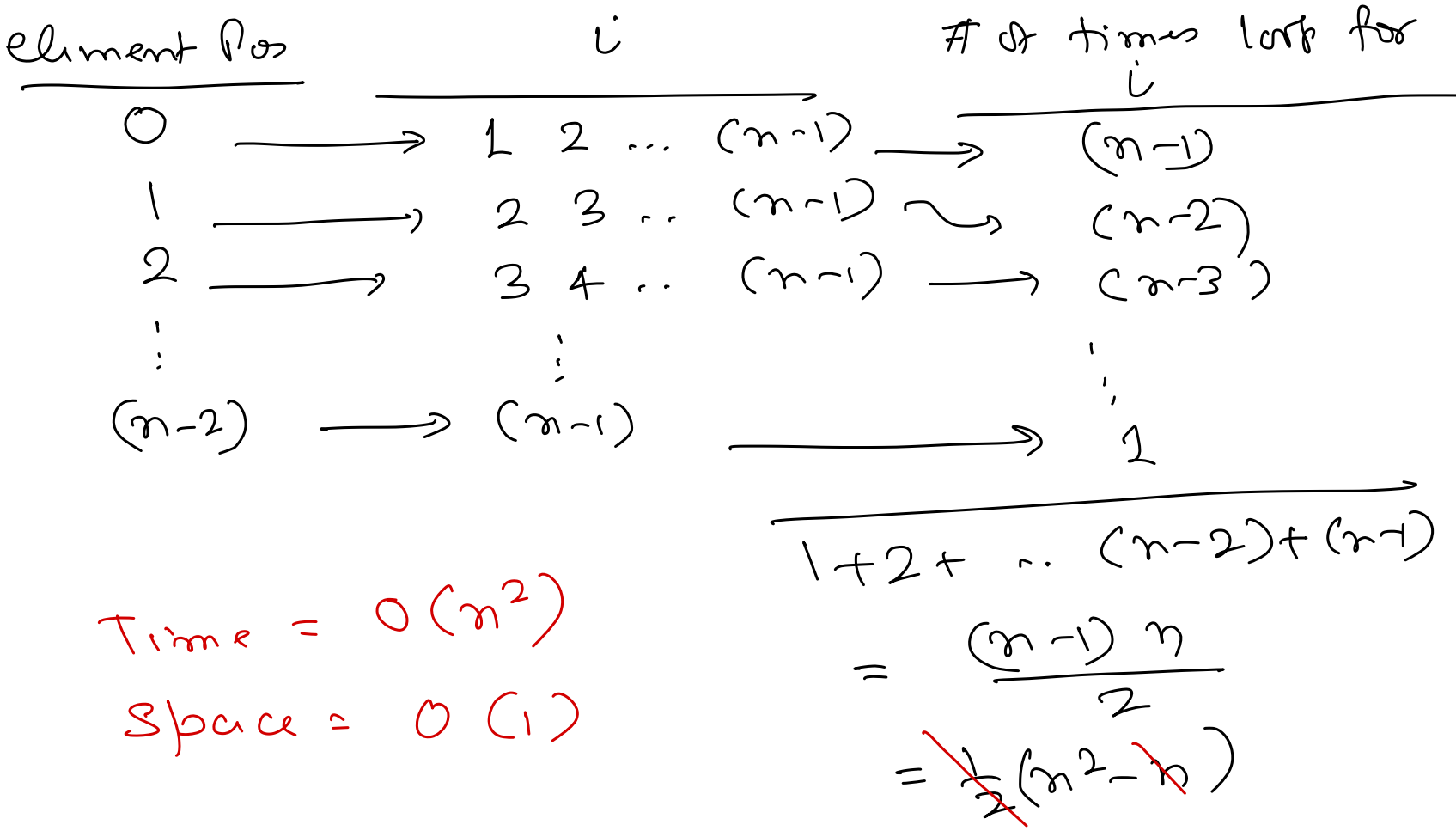
smallestElementPos → ~~0~~ 1

i → ~~1 2 3 4~~



swap elements at these positions

- elementPos = 0
- while (elementPos < (n-1)) do
  - smallestElementPos = elementPos
  - i = elementPos + 1
  - while (i < n) do
    - if (arr[i] < arr[smallestElementPos])
      - smallestElementPos = i
    - i = i+1
  - if (elementPos != smallestElementPos)
    - swap values at elementPos & smallestElementPos
  - elementPos = elementPos + 1



Time = $O(n^2)$

Space = $O(1)$