1. Left shifting 1:

```
arr = np.array([1,2,3,4,5,6])
for i in range(0, len(arr)-1):
    arr[i] = arr[i+1]
arr[i+1] = 0
print(arr)
```

2. Left shift 2:

```
arr = np.array([1,2,3,4,5,6])
for i in range(1, len(arr)):
    arr[i-1] = arr[i]
arr[i] = 0
print(arr)
```

3. Right shift 1:

```
arr = np.array([1,2,3,4,5,6])
for i in range(len(arr)-1, 0, -1):
    arr[i] = arr[i-1]
arr[i-1] = 0
print(arr)
```

4. Right shifting 2:

```
arr = np.array([1,2,3,4,5,6])
for i in range(len(arr)-2,-1,-1):
    arr[i+2] = arr[i]
arr[i] = 0
print(arr)
```

5. Left shift at any index:

```
arr = np.array([1,2,3,4,5,6])
index = 3
for i in range(0,index,len(arr)-1):
    arr[i-1] = arr[i+1]
arr[i+1] = 0
print(arr).
```

Type 2:

6. index = 3

```
for i in range(index+1, len(arr)):
    arr[i-1] = arr[i]
arr[i] = 0
print(arr)
```

7. Right shifting at any indx 1:

```
arr = np.array([1,2,3,4,5,6,7])
indx = 3
for i in range(len(arr)-1, indx, -1):
        arr[i] = arr[i-1]
arr[i-1] = 0
print(arr)
```

8. type2:

```
arr = np.array([1,2,3,4,5,6,7])
indx = 3
for i in range(len(arr)-2, indx-1, -1):
        arr[i+1] = arr[i]
arr[i] = 0
print(arr)
```

9. Left rotate:

```
arr = np.array([1,2,3,4,5,6])
temp = arr[0]
for i in range(0, len(arr)-1):
    arr[i] = arr[i+1]
arr[i+1] = temp.
```

10. Left rotate at any index:

```
arr = np.array([1,2,3,4,5,6,7])
indx = 3
for i in range(indx+1):
    temp = arr[0]
    for j in range(0, len(arr)-1):
        arr[j] = arr[j+1]
    arr[j+1] = temp.
print(arr)
```

11. Right rotate:

```
arr = np.array([1,2,3,4,5,6,7])
temp = arr[len(arr)-1]
for i in range(len(arr)-1, 0, -1):
    arr[i] = arr[i-1]
arr[i-1] = temp
print(arr)
```

12. Right rotate at any indx:

```
arr = np.array([1,2,3,4,5,6,7])
indx = 4
indx = len(arr) - indx
For i in range(indx):
    temp = arr[len(arr)-1]
    for j in range(len(arr)-1, 0, -1):
        arr[j] = arr[j-1]
    arr[j-1] = temp.
print(arr)
```

13. Reverse in place

```
arr = np.array([1,2,3,4,5,6])
left = 0
right = len(arr)-1
while left < right:
    arr[left], arr[right] = arr[right], arr[left]
    left += 1
    right -= 1
print(arr)
```

14. Revers in (out)place

```
arr = np.array([1,2,3,4,5,6,7])
reversed = np.array([0] * len(arr))
pointer = 0
for i in range(len(arr)-1, -1, -1):
    reversed[pointer] = arr[i]
    pointer += 1
print(reversed)
```

15. Insertion

```
def resize (array, size):
    arr = np.array([0]*(size+1))
    for i in range(len(array)):
        arr[i] = array[i]
    return arr
def insertion(array, size, value, index):
    if index<0 or index>size:
        return "Invalid"
    elif index == size:
        array = resize(array, size)
        array[index] = value
        return array
    else:
        array = resize(array, size)
        for i in range(len(array)-1, index, -1):
            array[i] = array[i-1]
        array[i-1] = value
        return array.
```

16. deletion:

```
def deletion (array, size, index):
    if size <= 0 or (index < 0 or index >= size):
        return "Invalid"
    elif index == size-1:
        array [index] = 0
    else:
        for i in range(index, len(array)-1):
            array [i] = array [i+1]
        array [i+1] = 0
        return array.
```

Duplicate remove:

```
arr = np.array([1,1,1,2,2,2,4,4,7,8,9,9])
right=0
for i in range(0, len(arr)-1):
    if arr[right] == arr[right+1]:
        for j in range(right, len(arr)-1):
            arr[j] = arr[j+1]
        arr[j+1] = 0
    else:
        right+=1
print(arr)
```

Compress Matrix in 2X2:

```
a, b, c, d = 0, 0, 0, 0
Outmiddle = int(len(array)/2)
Inmiddle = int(len(array[0])/2)
for i in range(0, Outmiddle):
    for j in range(0, Inmiddle):
        a += array[i][j]
for i in range(0, Outmiddle):
    for j in range(Inmiddle, len(array[0])):
        b += array[i][j]
for i in range(Outmiddle, len(array)):
    for j in range(0, Inmiddle):
        c += array[i][j]
for i in range(outmiddle, len(array)):
    for j in range(Inmiddle, len(array[0])):
        d += array[i][j]
```

2D Array

1. 2D array ⟶ 1D array row major

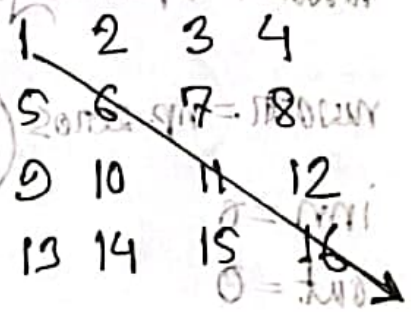
```
arr2D = np.array([[1,2,3,4],[5,6,7,8]])
arr1D = np.array([0]*(len(arr2D[0])*len(arr2D)))
pointer=0
for i in arr2D:
    for j in i:
        arr1D[pointer] = j
        pointer+=1
print(arr1D)
```

2. 2D array ⟶ 1D array column major

```
arr2D = np.array([[1,2,3,4],[5,6,7,8]])
arr1D = np.array([0]*(len(arr2D[0])*len(arr2D)))
pointer=0
for i in range(len(arr2D[0])):
    for j in range(len(arr2D)):
        arr1D[pointer] = arr2D[j][i]
        pointer+=1
print(arr1D)
```

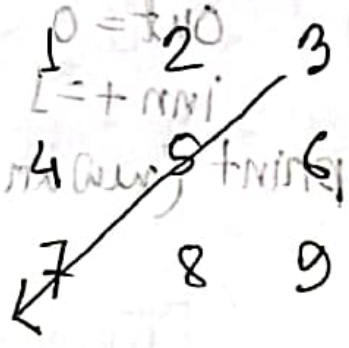
3. Diagonal Matrix primary:

```
arr = np.array([[1,2,3,4],[5,6,7,8],[9,10,11,12],[13,14,15,16]])
pointer = 0
sum = 0
for i in arr:
    sum += i[pointer]
    pointer += 1
print(sum)
```



4. Diagonal Matrix Secondary

```
arr = np.array([[1,2,3],[4,5,6],[7,8,9]])
pointer = len(arr[0])-1
sum = 0
for i in arr:
    sum += i[pointer]
    pointer -= 1
print(sum)
```

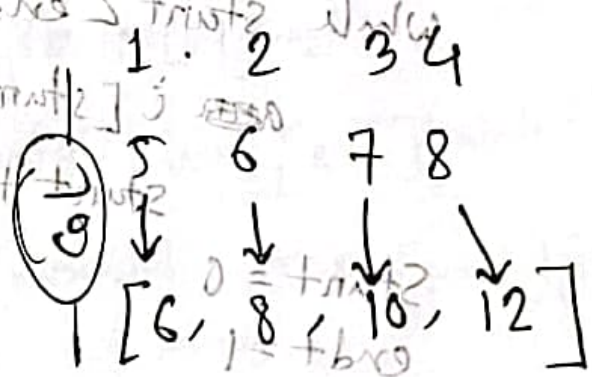


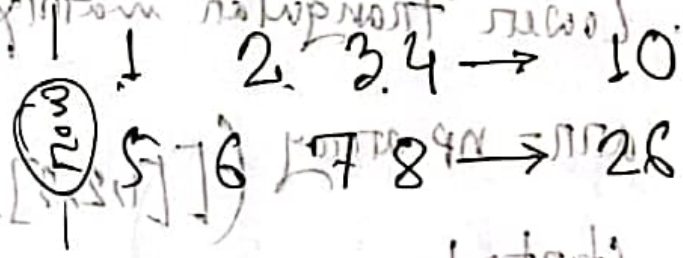
5. transpose Matrix

```
arr=np.array([[1,2,3,4],[5,6,7,8],[9,10,11,12]])
~~newarr=np.array([0]*len(arr))~~
newarr=np.zeros((len(arr[0]),len(arr)))
inn=0
out=0
for i in arr:
    for j in i:
        newarr[out][inn]=j
        out+=1
    inn+=1
print(newarr)
```

6. Sum of rows:

```
arr = np.array([[1,2,3,4],[5,6,7,8]])
~~rowArr = np.array([0])~~
rowArr = np.zeros((len(arr),1), dtype=int)
sum = 0
pointer = 0
for i in arr:
    for j in i:
        sum += j
    rowArr[pointer] = sum
    sum = 0
    pointer += 1
print(rowArr)
```



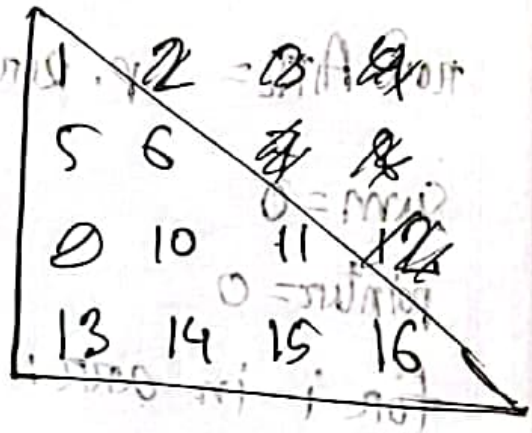


7. Sum of Columns:

```
arr = np.array([[1,2,3,4],[5,6,7,8]])
colArr = np.array([0]*len(arr[0]))
sum = 0
for i in range(len(arr[0])):
    for j in range(len(arr)):
        sum += arr[j][i]
    colArr[i] = sum
```

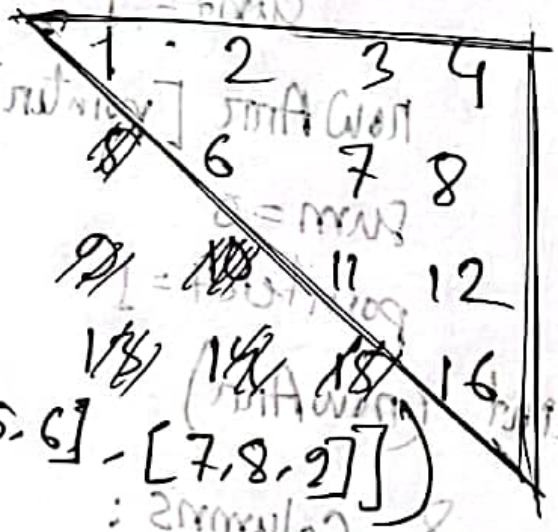
8. Upper triangular matrix:

```
arr = np.array([[1,2,3],[4,5,6],[7,8,9]])
start = 0
end = 0
for i in arr:
    while start < end:
        i[start] = 0
        start += 1
    start = 0
    end += 1
print(arr)
```



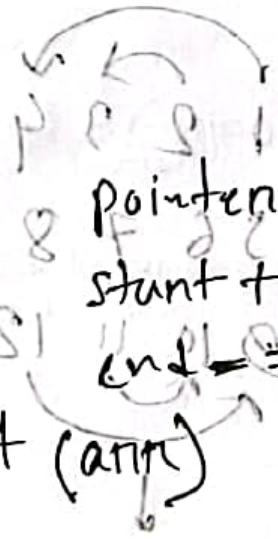
9. Lower triangular matrix:

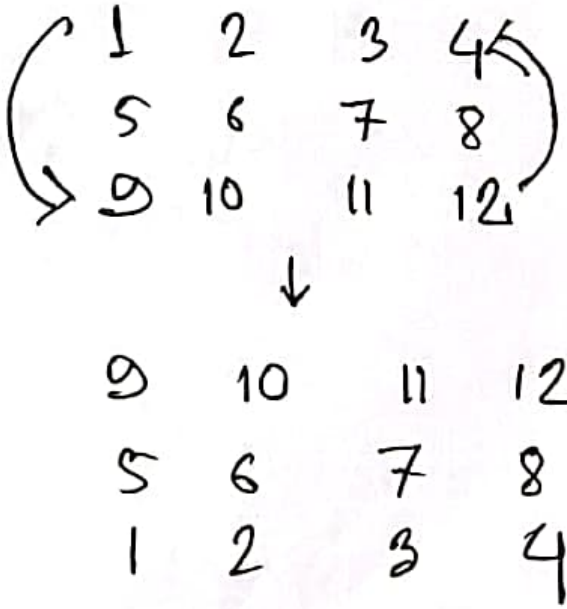
```
arr = np.array([[1,2,3],[4,5,6],[7,8,9]])
start = 1
for i in arr:
    for j in range(start, len(i)):
        i[j] = 0
    start += j
print(arr)
```



10. Swipe Rows :

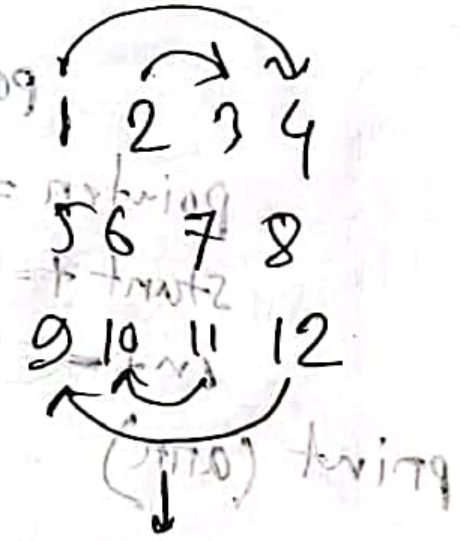
```
arr = np.array([[1,2,3,4],[5,6,7,8],[9,10,11,12]])
pointer = 0
start = 0
end = len(arr) - 1
while start < end:
    while pointer <= len(arr[0]) - 1:
        arr[start][pointer], arr[end][pointer]
        = arr[end][pointer], arr[start][pointer]
        pointer += 1
    pointer = 0
    start += 1
    end -= 1
print(arr)
```

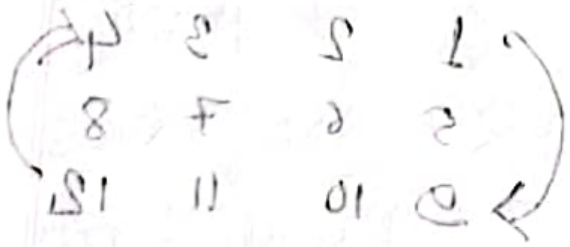


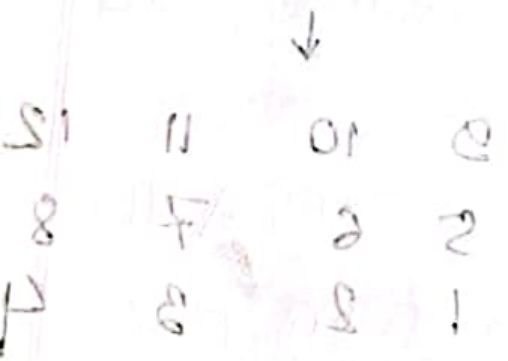


11. Swipe column:

```
arr = np.array([[1,2,3,4],[5,6,7,8],[9,10,11,12]])
start = 0
end = len(arr[0]) - 1
while start < end:
    for i in arr:
        i[start], i[end] = i[end], i[start]
    start += 1
    end -= 1
print(arr)
```







4 3 2 1
8 7 6 5
12 11 10 9

12. Row rotate:

```
arr = np.array([[1,2,3,4],[5,6,7,8],[9,10,11,12]])
stunt = 0
for v in range(3):
    newArr = np.array([0]*len(arr[0]))
    pointer = 0
    for i in arr[len(arr)-1]:
        newArr[pointer] = i
        pointer += 1
for i in range(len(arr)-1, 0, -1):
    while stunt < len(arr[i]):
        arr[i][stunt] = arr[i-1][stunt]
        stunt += 1
    stunt = 0
arr[i-1] = newArr
print(arr)
```

13. Column Swipe: rotate:

```
arr = np.array([[1,2,3,4],[5,6,7,8],[9,10,11,12]])
for i in range(2):
    for j in ~~range~~(: arr:
        temp = j[len(j)-1]
        for k in range(len(j)-1, 0, -1):
            j[k] = j[k-1]
        j[k-1] = temp
print(arr)
```

14. Linear Array of length 128 3D array = 4x4x8

box[x][y][z] → linear ~~arr~~ index = 111

~~x(*4~~

x*(4x8) + y*(8) + z = 111

x = 111 // (4*8) = 111//32
= 3.46875
or, 3

111 % (4*8) = 15

y = 15//(8) = 1.875 = 1

15 % 8 = 7

z = 7//1 = 7

111/32 = 3.46875

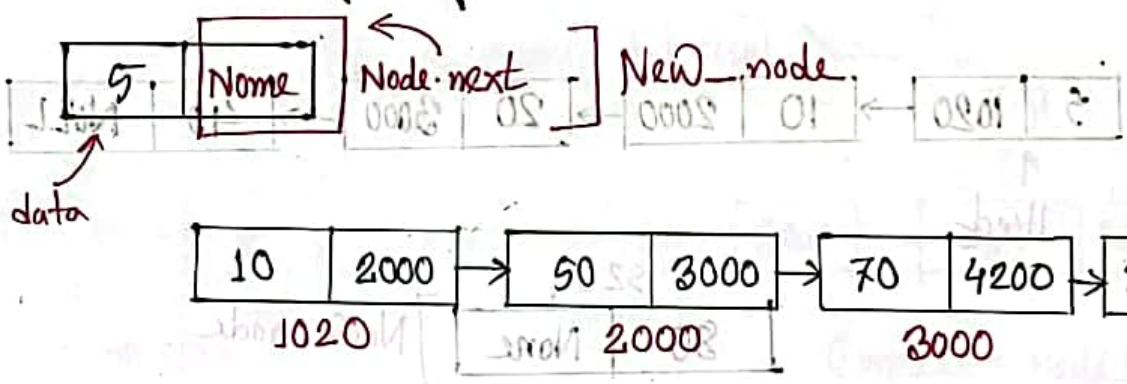
3.46875 − 3
= 0.46875

= 0.46875 x 32
= 15

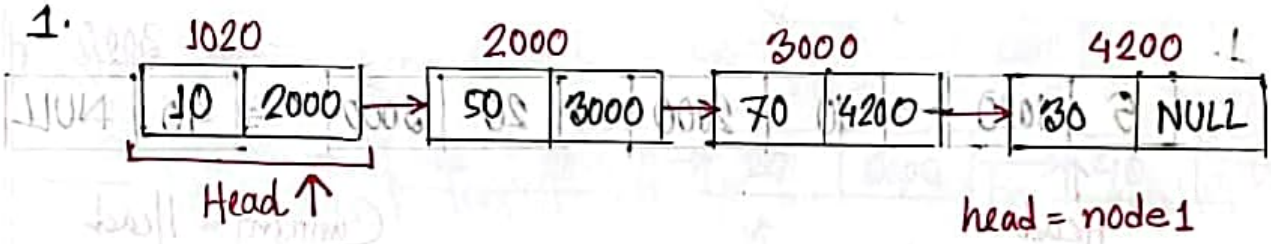
15/8 = 1.875
= 0.875 x 8

Linked List

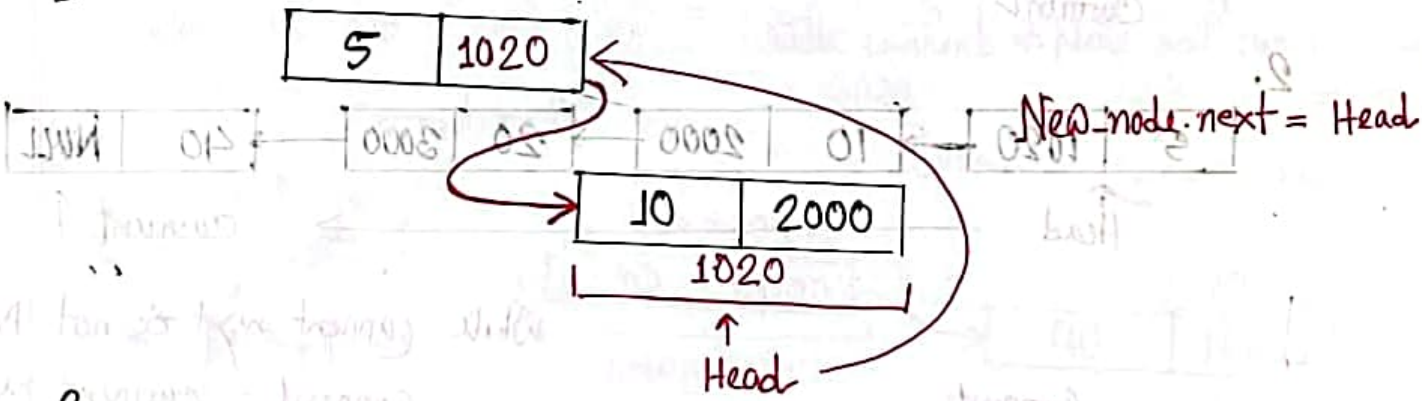
1. Insertion at the begaining of the list :



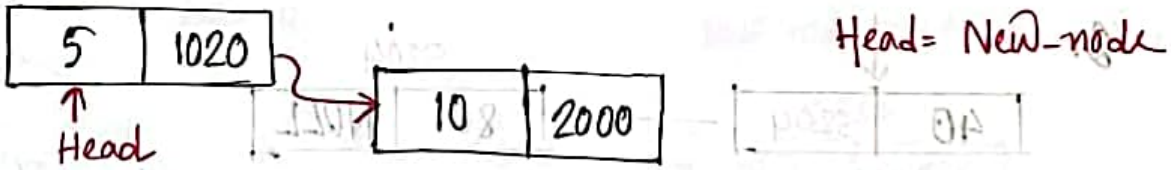
1.



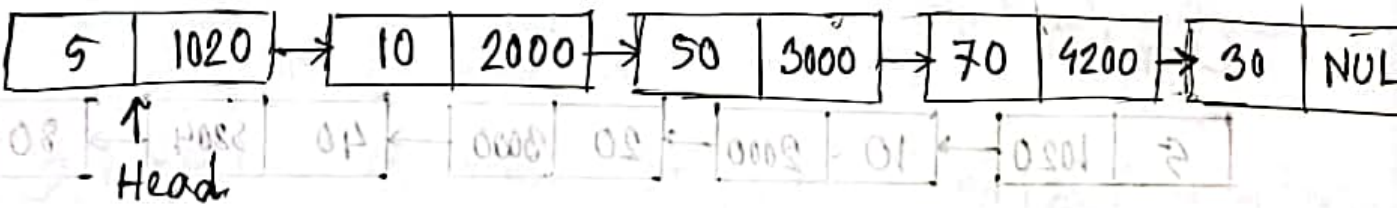
2.



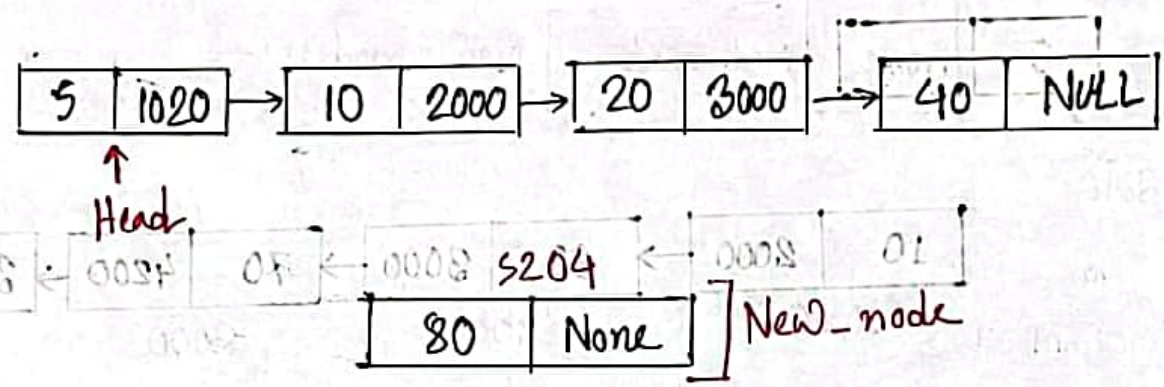
3.



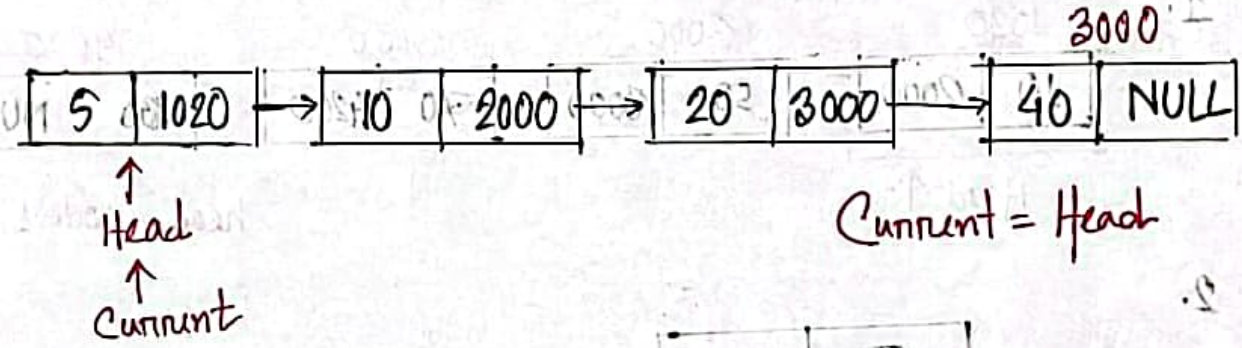
Output :



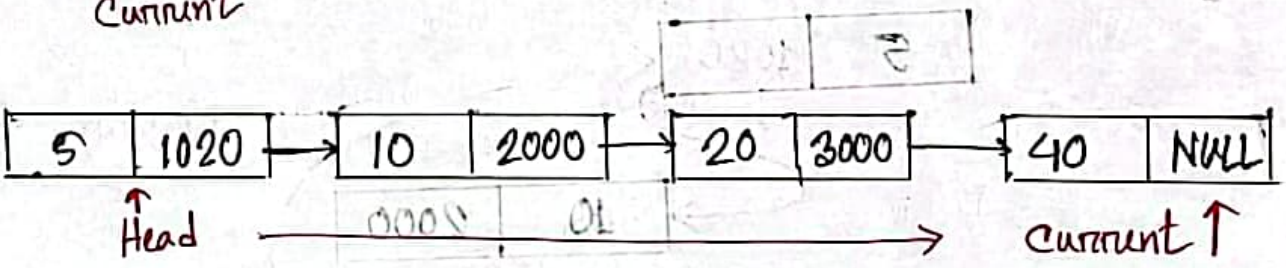
2. Inserting at the end of the linked list:



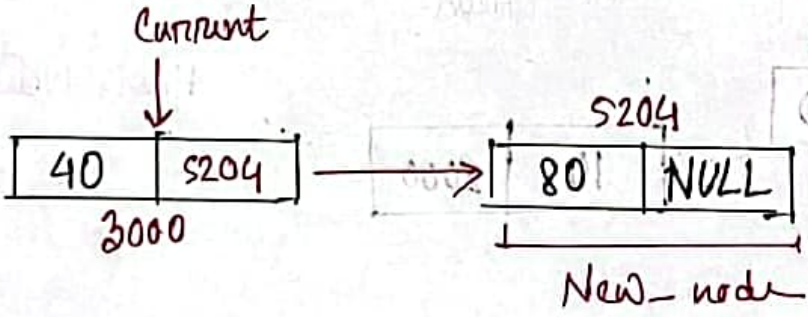
1.



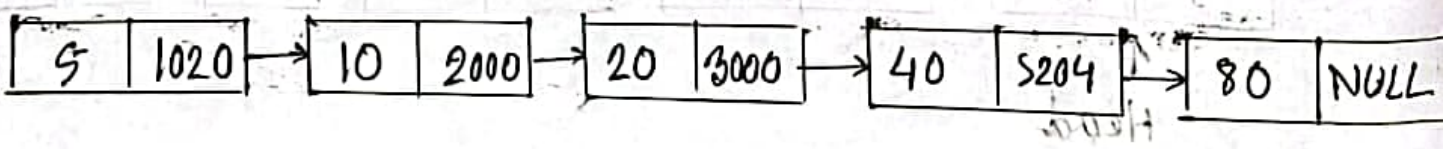
2.



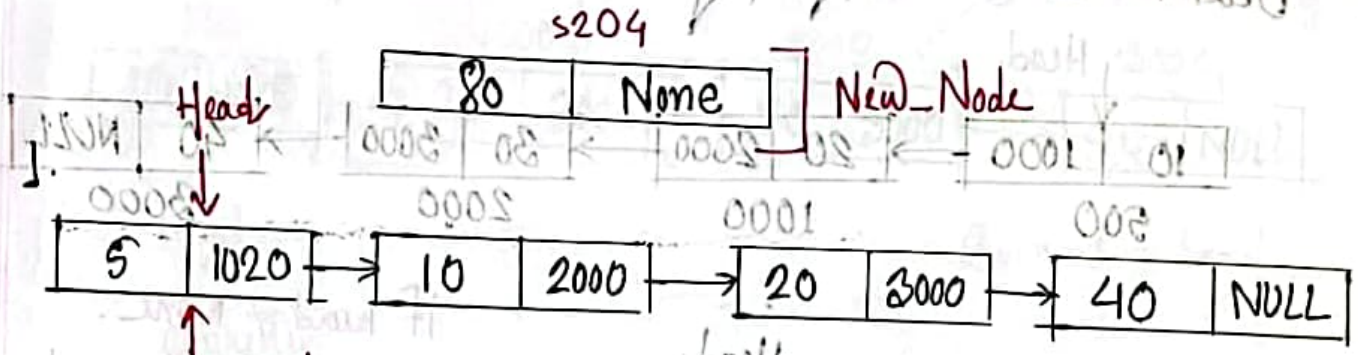
3.



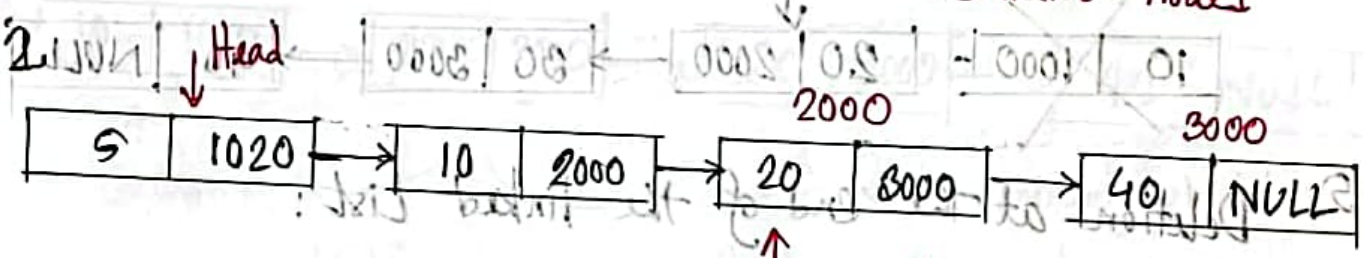
Output:



3. Insert at the middle of the linked List:

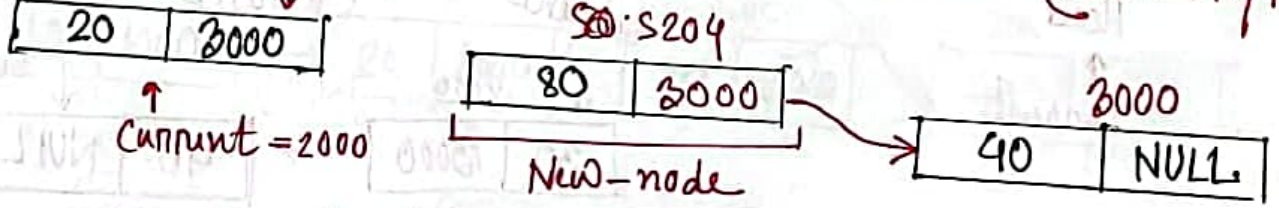


Current = node1



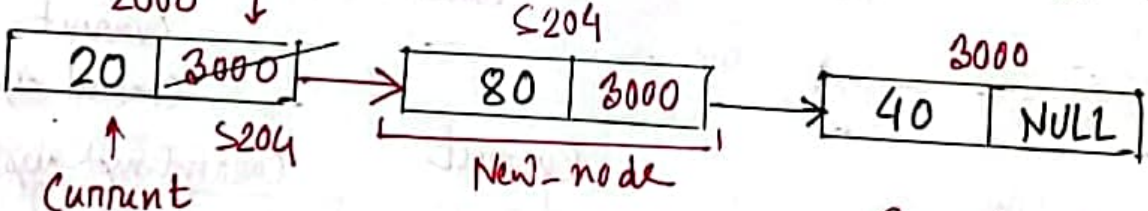
While current ≠ None and current.data ≠ 20:
Current = Current.nxt.

3. Current.next = 3000

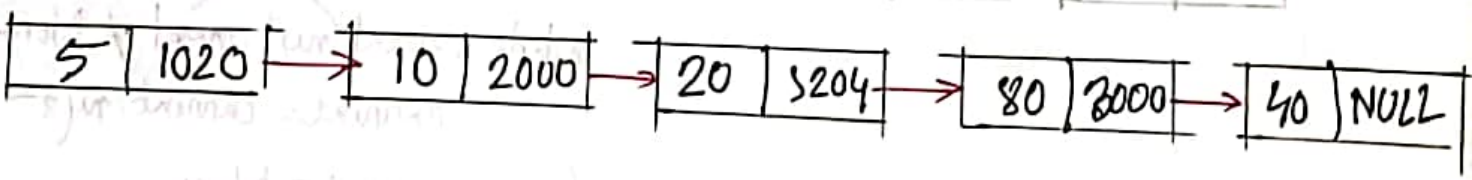


4. Current.nxt

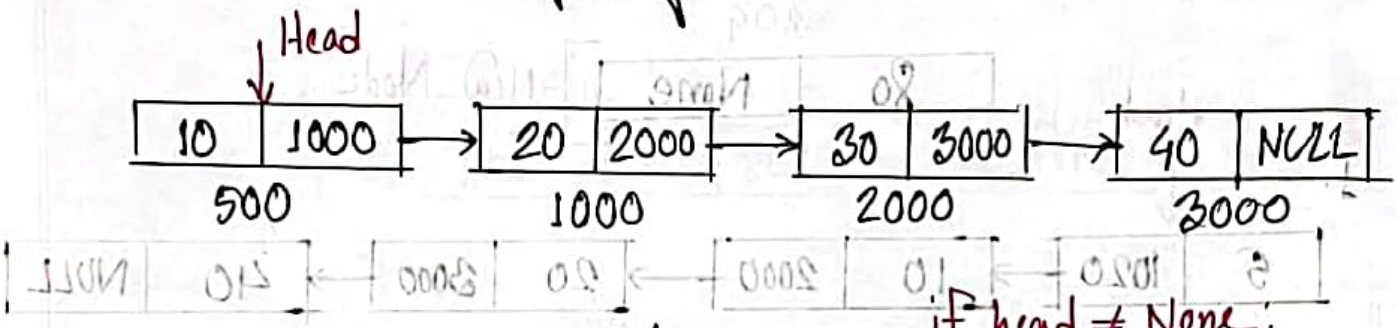
New_node.next = current.next



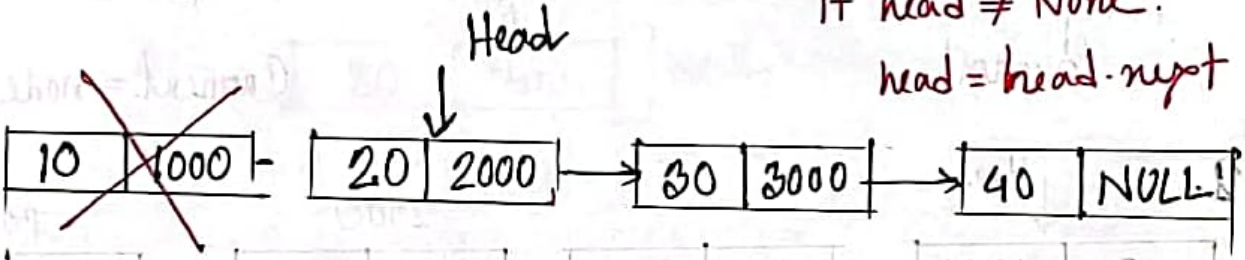
Current.next = new_node



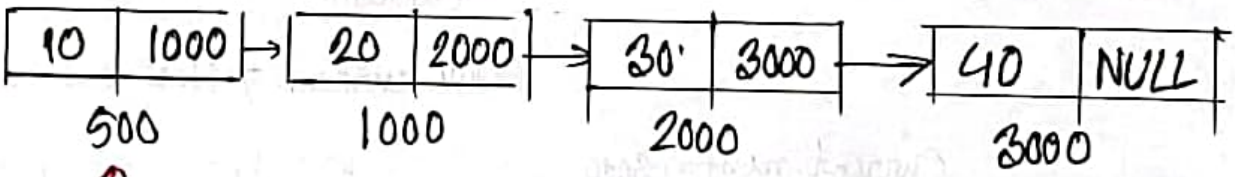
4. Deletion at the beginning of the linked list:



if head ≠ None:
head = head.next



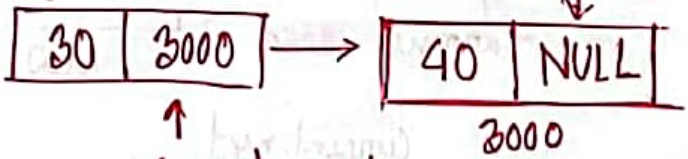
5. Deletion at the end of the linked List:



Head
Current

Current = head

Current.next
.next



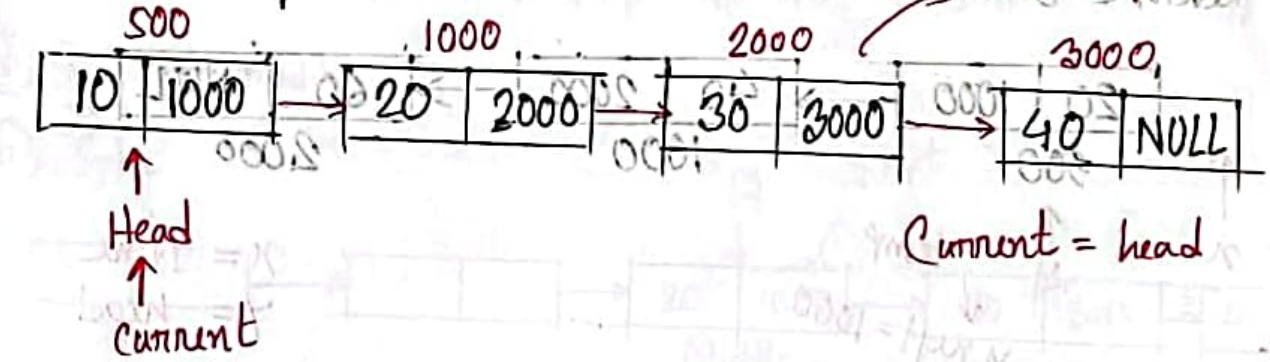
Current.next

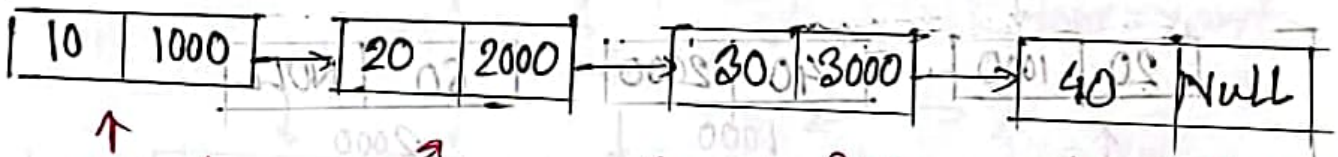
Current = 2000
Current.next = 3000
Current.next.next = NULL

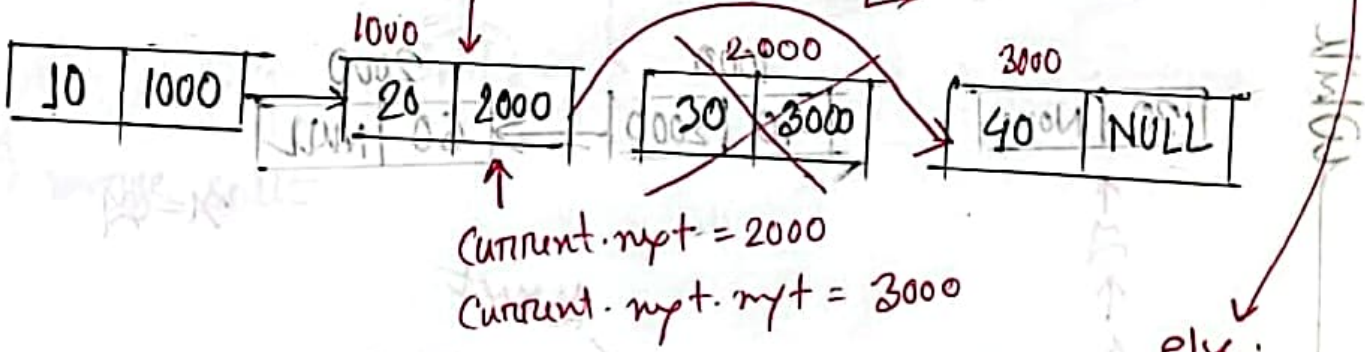
Current

While current.next.next ≠ None:
Current = current.next
Current.next = None

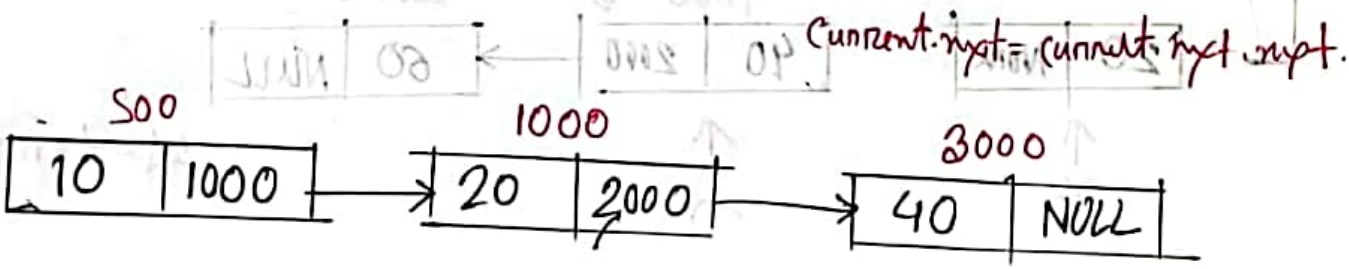
6. Deletion at a particular node:







else:



```
class Node:
    def __init__(self, value, next=None):
        self.value = value
        self.next = next
```

Length of a linked list:

```
def length(head1):
    count=0
    tail = head1
    while tail:
        count+=1
        tail=tail.next
    return count
```

nodeAt function to find any node with index:

```
def nodeA(head1, index):
    count = 0
    tail= head1
    while tail:
        if count == index:
            return tail
        count+=1
        tail=tail.next
```

3. Change value with index:

```
def changeValue(head1, value, index):
    count = 0
    tail = head1
    while tail:
        if count == index:
            break
        count += 1
        tail = tail.next
    tail.value = value
    return head1
```

4. def search(head1, element):

```
    tail = head1
    while tail.value != element:
        tail = tail.next
    return tail
```

5. Insertion:

```
def insertion (head1, index, value, size):
    if index < 0 or index > size:
        print("Invalid Index")
        return
    else:
        n = Node (value)
        if index == 0:
            node = NodeAt (head1, index)
            n.next = node
            head1 = n
        elif index == size:
            node = NodeAt (head1, index-1)
            node.next = n
        else:
            node = nodeAt (head1, index)
            prevNode = nodeAt (head1, index-1)
            prevNode . next = n
            n.next = ~~nextNo~~ node
    return head1
```

6. Removal:

```
def removal (head1, index, size):
    if index<0 or index>=size:
        print("Invalid Index")
        return
    else:
        if index == 0:
            node = nodeAt(head1, index)
            current = node
            head1 = head1.next
            current.next = None
            current.value = None
        elif index == size-1:
            node = nodeAt(head1, index-1)
            current = node.next
            node.next = None
            current.value = None
        else:
            node = nodeAt(head1, index)
            prev = nodeAt(head1, index-1)
            nxt = nodeAt(head1, index+1)
            prev.next = nxt
            node.next = None
            node.value = None
    return head1
```

7. Find center of the linked list:

```
def center (head1):
    fast = head1
    slow = head2
    while fast and fast.next:
        fast = fast.net.next
        slow = slow.next
    return slow
```

8. Copy a linked list:

```
def copyList (head1):
    head2 = None
    tail = head1
    while tail:
        if head2 == None:
            node = Node (tail.value)
            head2 = node
            current = head2
        else:
            node = Node (tail.value)
            current.net = node
            current = current.next
        tail = tail.next
    return head2
```

9. Reverse In Place:

```
def reverseInPlace(head1):
    dummy = None
    previous = dummy
    current = head1
    While current != None:
        nxt = current.next
        current.next = previous
        previous = current
        current = nxt
    return previous
```

Reverse Out Place:

```
def reverseOutPlace(head1):
    head = Node(head1.value)
    tail = head1.next
    while tail:
        node = Node(tail.value, head)
        head = node
        tail = tail.next
    Return head
```

Left Rotate:

```
def LeftRotate (head1, rotation):
    for i in range (rotation):
        newHead = head1.next
        tail = newHead
        While tail.next:
            tail = tail.next
        tail.next = head1
        head1.nxt = None
        head1 = newHead
    Return head1
```

Right Rotate:

```
def rightRotate (head1, rotation):
    for i in range (rotation):
        previous = head1
        Current = head1.next
        while current.next:
            current = current.next
            previous = previous.next
        current.next = head1
        head1 = current
        previous.next = None
    return head1
```

Circular Linked List

1. Find length:

```
def length(head):
    counter=1
    fast = head.next
    slow = head
    while fast != slow:
        counter +=1
        fast = fast.next.next
        slow = slow.next
    return counter
```

2. Find center:

```
def center (head):
    fast = head.next
    slow= head
    while fast != head and fast.next != head:
        slow = slow.next
        fast = fast.next.next
    return slow.value
```

find a node:

```
def nodeAt(head, indep):
    count = 0
    tail = head
    while tail:
        if count == indep:
            return tail
        count += 1
        tail = tail.next
```

5. Insertion:

```
def insertion (head, value, index, size):
    if index<0 or index>size:
        print("Invalid")
    else:
        newnode = Node(value)
        if index == 0:
            currentNode = nodeAt(head, index)
            newnode.next = currentNode
            head = newNode
            tail.next = head
        elif index == size:
            tail.next = currentnode newNode
            tail = newNode
            tail.next = head
        else:
            currentnode = nodeAt(head, index)
            previousnode = nodeAt(head, index-1)
            previousnode.next = newNode
            newnode.next = currentnode
    return head
```

```
tail = head.next
while tail.next != head:
    tail = tail.next
```

6. Deletion:

```
def Remove (head, index, size):
    if index<0 or index>=size:
        print("Invalid index")
    else:
        if index = 0:
            node = Node(head, index)
            current = node
            head = head.next
            current.next = None
            current.value = None
            tail.next = head
        elif index == size-1:
            node = Node(head, index-1)
            current = node.next
            node.next = head
            current.value = None
        else:
            (Same)
```

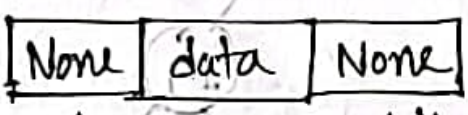
Class Tree:

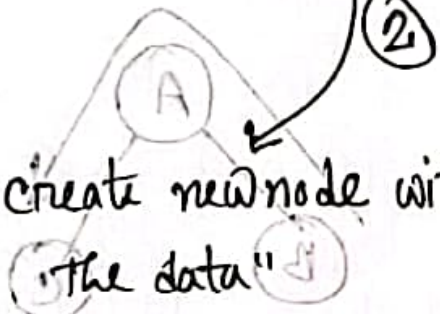
```
def createNode(self, data):
    return Node(data)

def insert(self, node, data):
    if node is None:
        return self.createNode(data)
    if data < node.data:
        node.left = self.insert(node.left, data)
    else:
        node.right = self.insert(node.right, data)
    return node
```

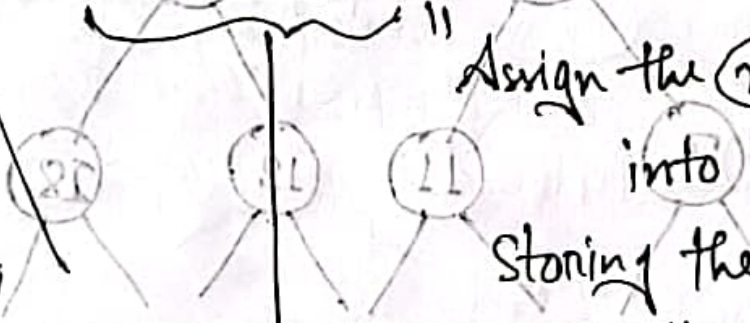
① "check if the node is empty"

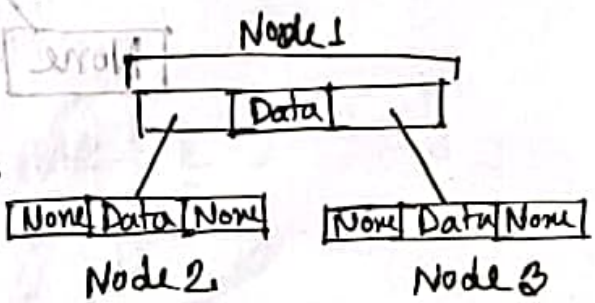
② "create new node with the data"

③ Assign the new node into left or right instead of storing the node address like linked list.







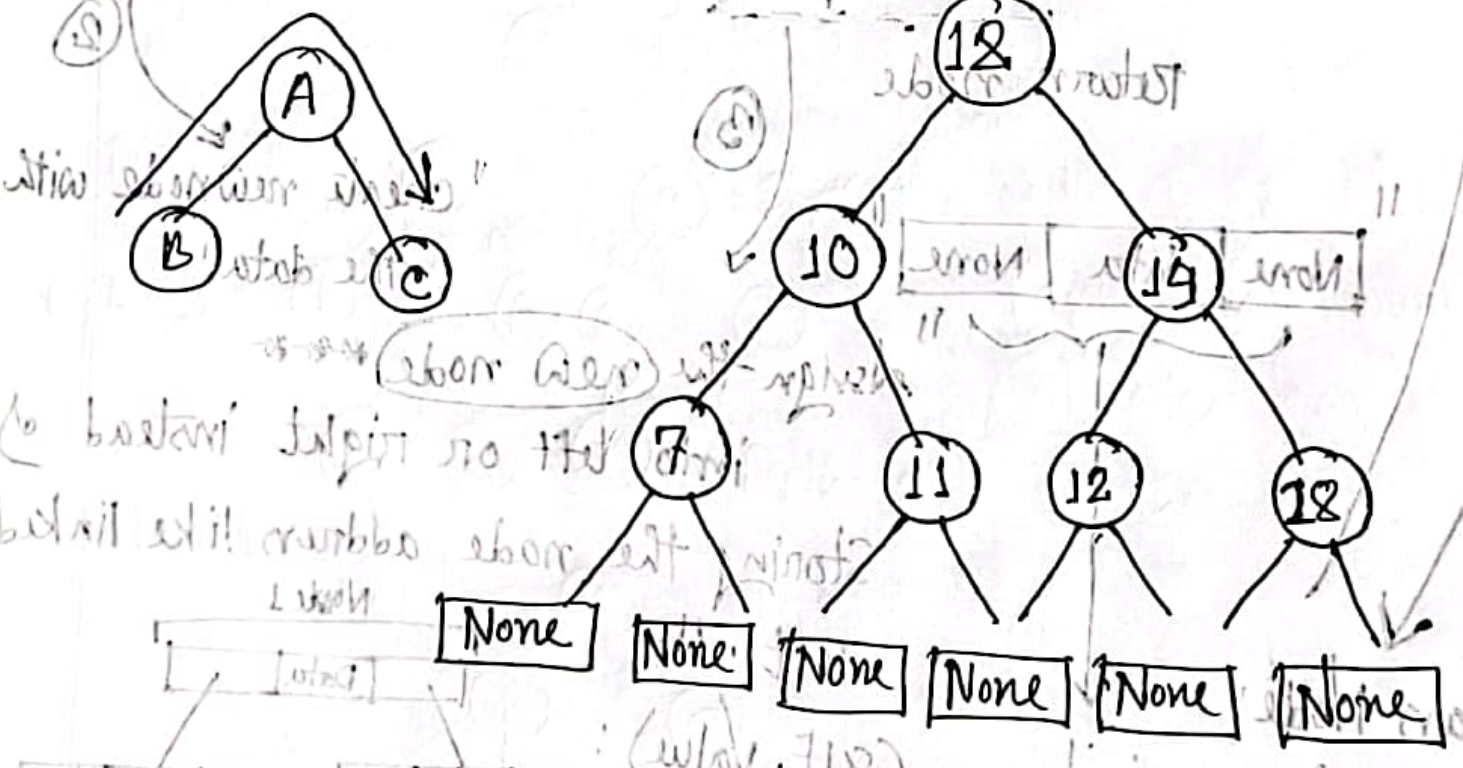



```
class Node:
    def __init__(self, value):
        self.left = None
        self.data = value
        self.right = None
```

1. Inorder traversal:

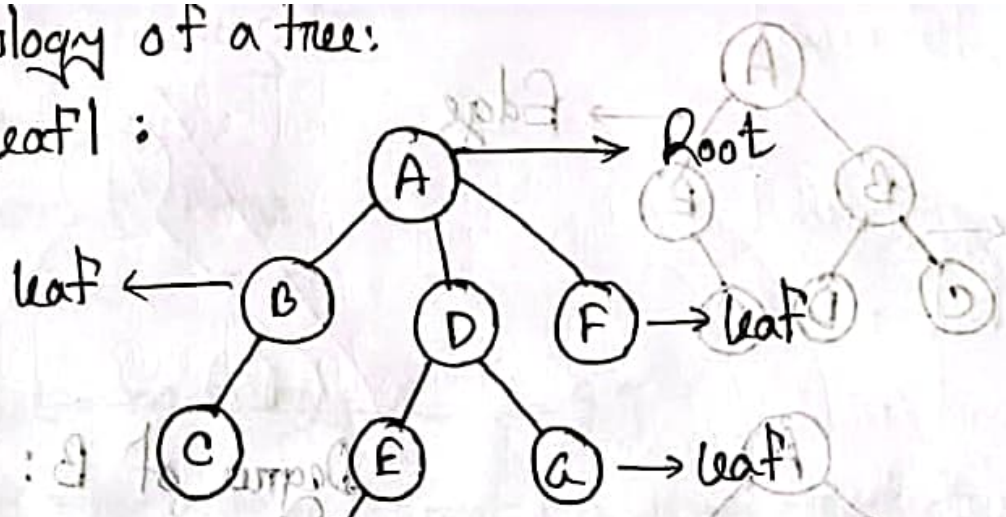
```
result = [ ]
def inOrder(root):
    global result
    if not root:
        return
    inOrder(root.left)
    result.append(root.value)
    inOrder(root.right)
```

```
inOrder(root)
```

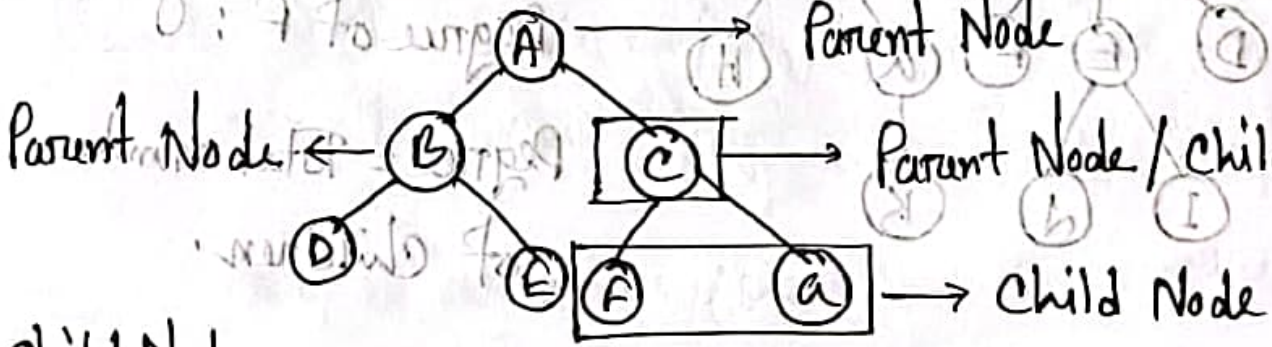


Terminology of a tree:

Root | Leaf | :

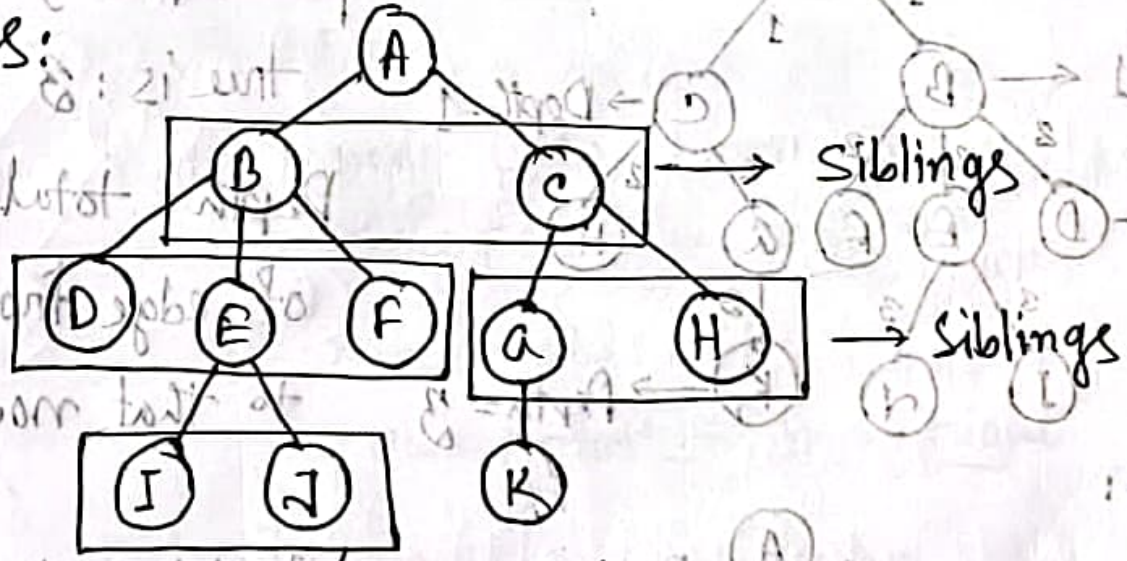


Parent:

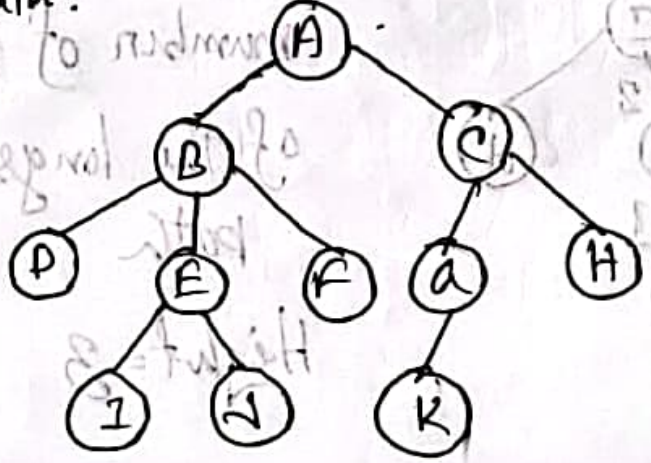


Child Node:

Siblings:



~~Edge~~ / Path:



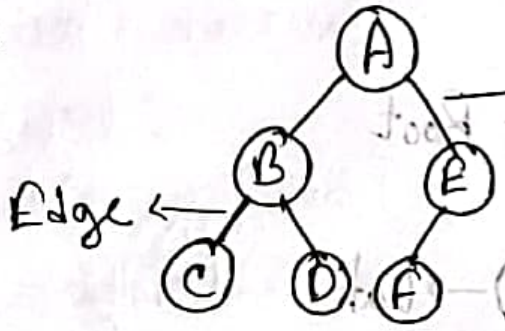
Path between A and J is

A → B → E → J

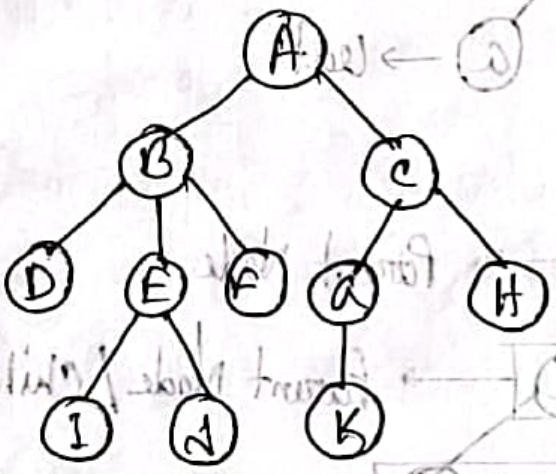
Path between A and K is

A → C → G → K

Edge:



Degree:



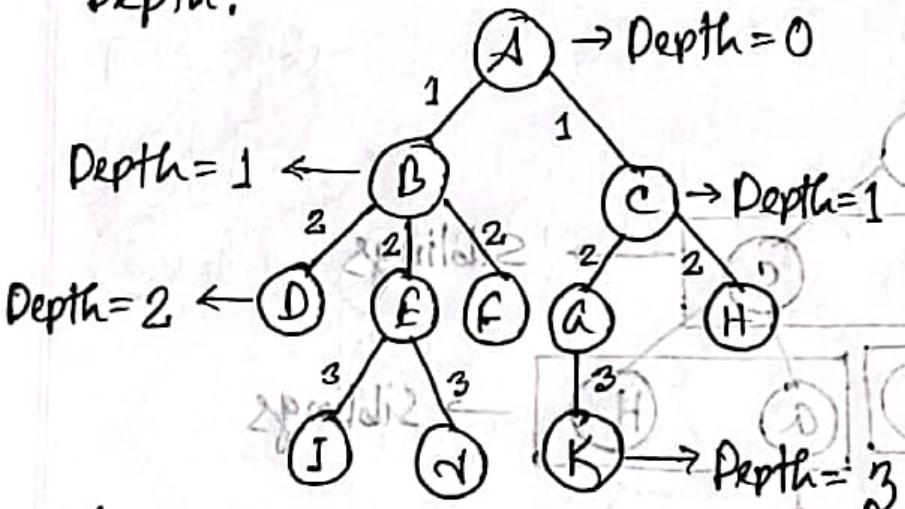
Degree of B : 3

Degree of A : 2

Degree of F : 0

Degree = Total number of children.

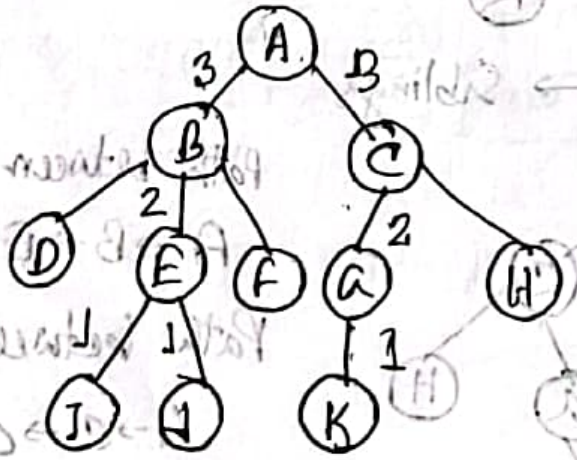
Depth:



Here depth of the tree is : 3

Depth = total number of edge from root to that node

Height:

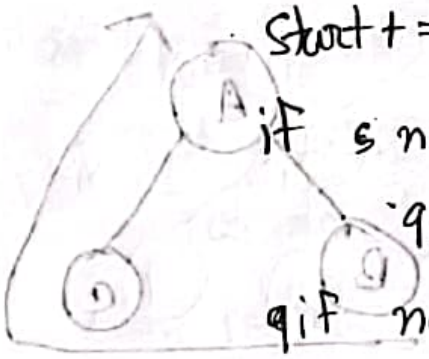


Height is the total number of edge of the longest path

Height = 3

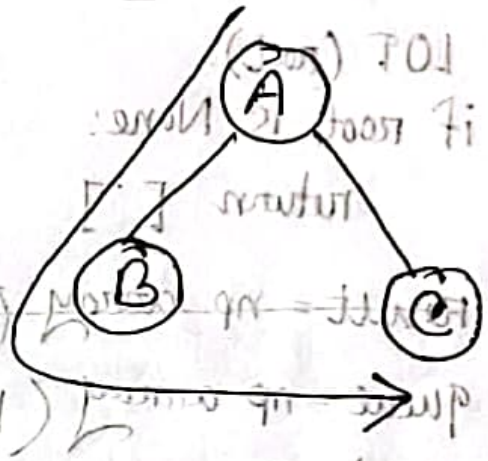
Level Order Traversal

```
def LOT (root):
    if root is None:
        return [ ]
    result = np.array( d = { } )
    queue = np.array ([root], dtype = object)
    start = 0
    while start < len(queue):
        size = len(queue) - start
        for i in range (size):
            node = queue[ start ]
            start += 1
            if node.left:
                queue [start] = node.left
            if node.right:
                queue [start] = node.right
        d [start] = node
    return d
```



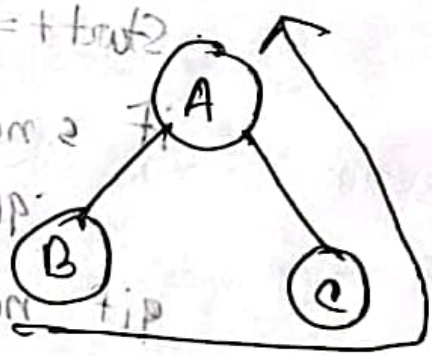
2. PreOrder Traversal:

```
res = []
def preOrder(root):
    global res
    if not root:
        return
    res.append(root.value)
    preOrder(root.left)
    preOrder(root.right)
preOrder(root)
```



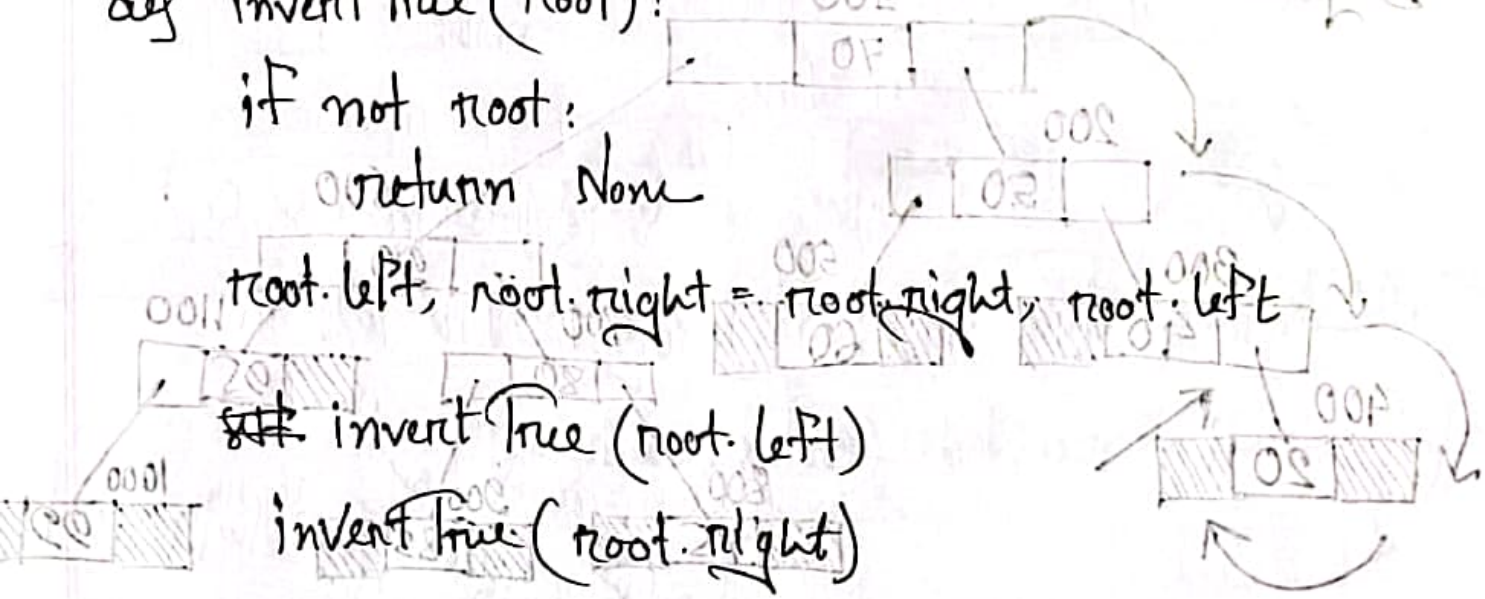
3. postOrder Traversal:

```
res = []
def postOrder(root):
    global res
    if not root:
        return
    postOrder(root.left)
    postOrder(root.right)
    res.append(root.value)
postOrder(root)
```



Invert Binary Tree: [35]

```
def invertTree(root):
    if not root:
        return None
    root.left, root.right = root.right, root.left
    invertTree(root.left)
    invertTree(root.right)
    return root
```



Maximum depth of a Binary tree: [36]

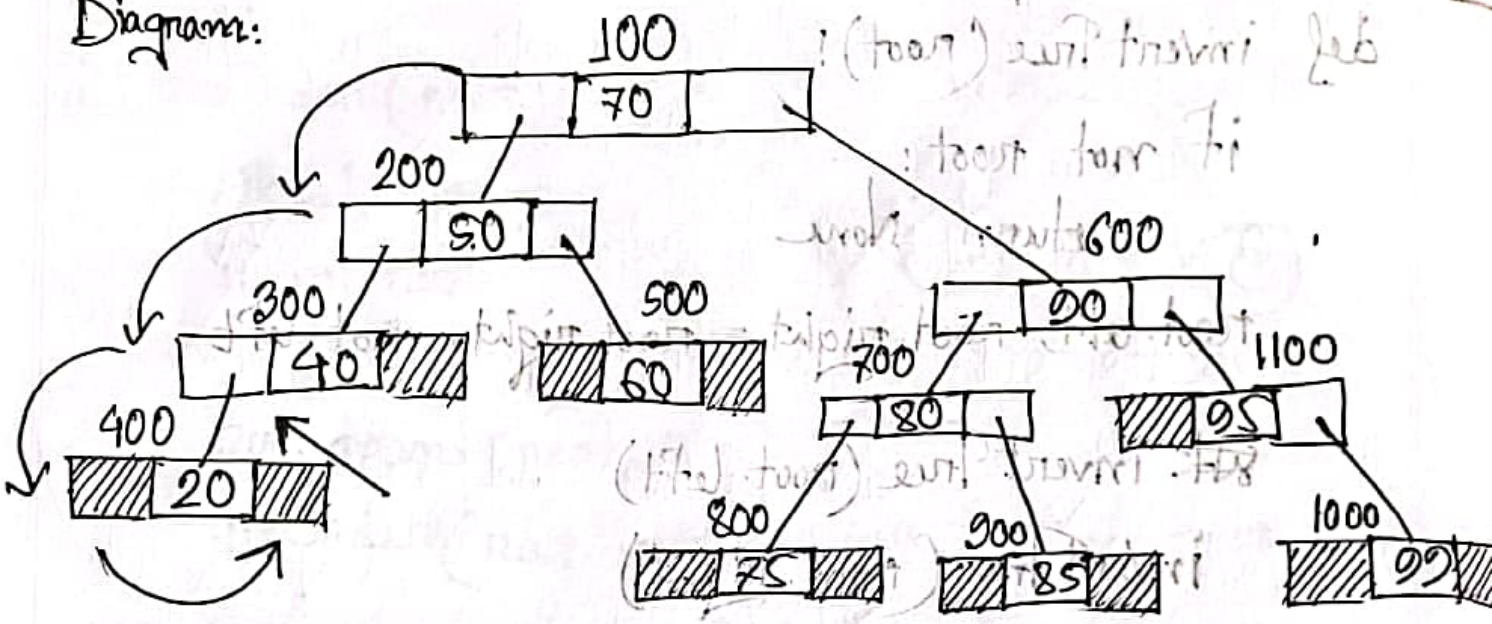
```
def maxdepth(root):
    if not root:
        return 0
    return 1+ max(maxdepth(root.left), maxdepth(root.right))
```

~~Total number of nodes:~~ Minimum depth of a binary tree [37]

```
def minDepth(root):
    if root is None:
        return 0
    if root.left is None:
        return minDepth(root.right)+1
    if root.right is None:
        return minDepth(root.left)+1
    return min(minDepth(root.left), minDepth(root.right))+1
```

7 Check if balanced or unbalanced binary tree: [34]

Diagram:

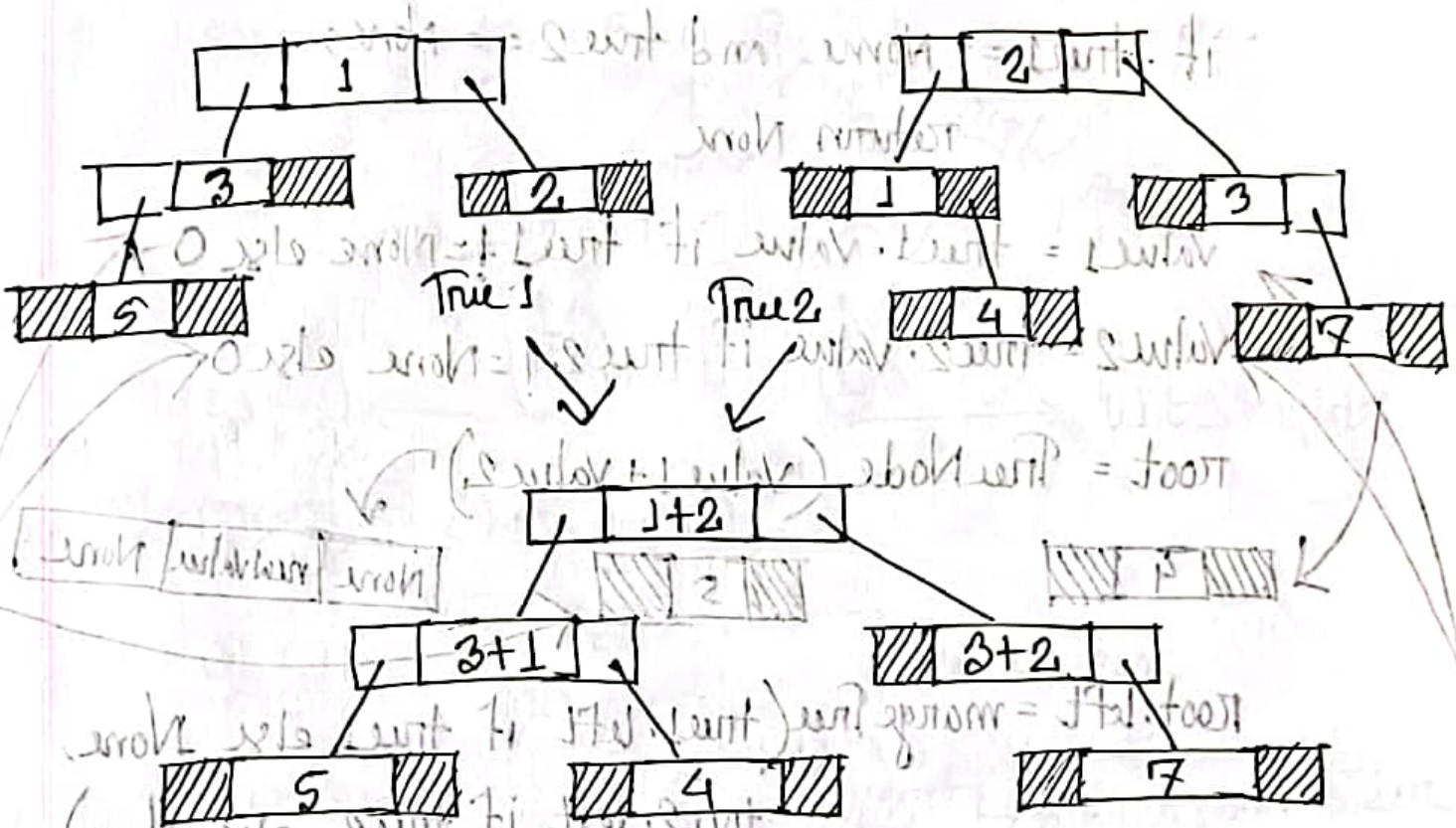


Steps:

1. Depth first search [go to bottom node]
2. (a) find the difference of left and right and check either its less than or equal 1 or not
   (b) if true then count the depth with 1 + max(left, right)

```
def balanced(root):
    if not root:
        return [True, 0]
    left = balanced(root.left)
    right = balanced(root.right)
    check = (left[0] and right[0]) and
            (abs(left[1] - right[1]) <= 1)
    return [check, 1 + max(left[1], right[1])]
balanced(root)
```

8. Merge two binary tree:



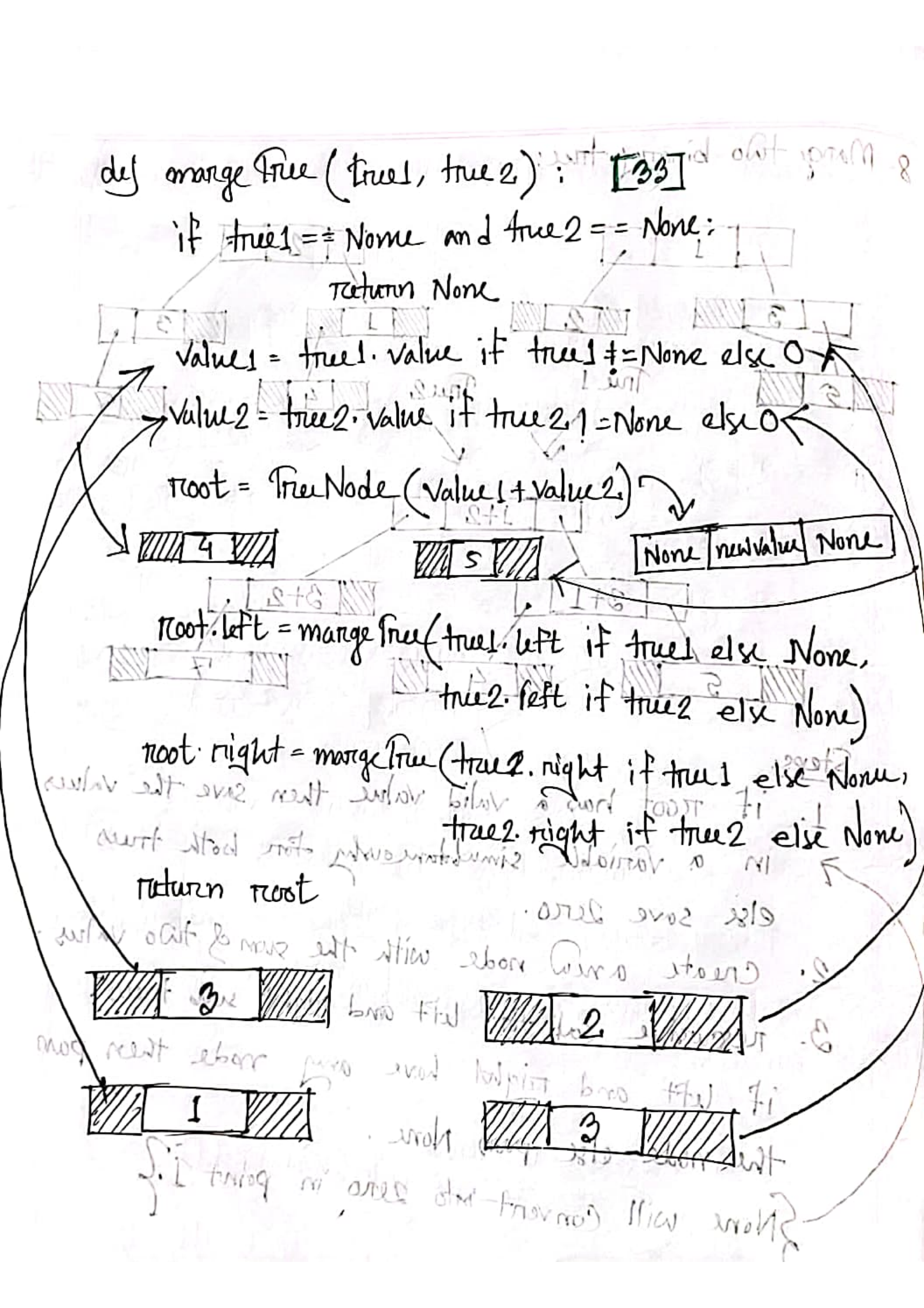
Steps:

1. if root has a valid value then save the values in a variable simultaneously for both trees else save zero.
2. Create a new node with the sum of two values.
3. recursive call for left and right sub trees if left and right have any node then pass the node else pass None.

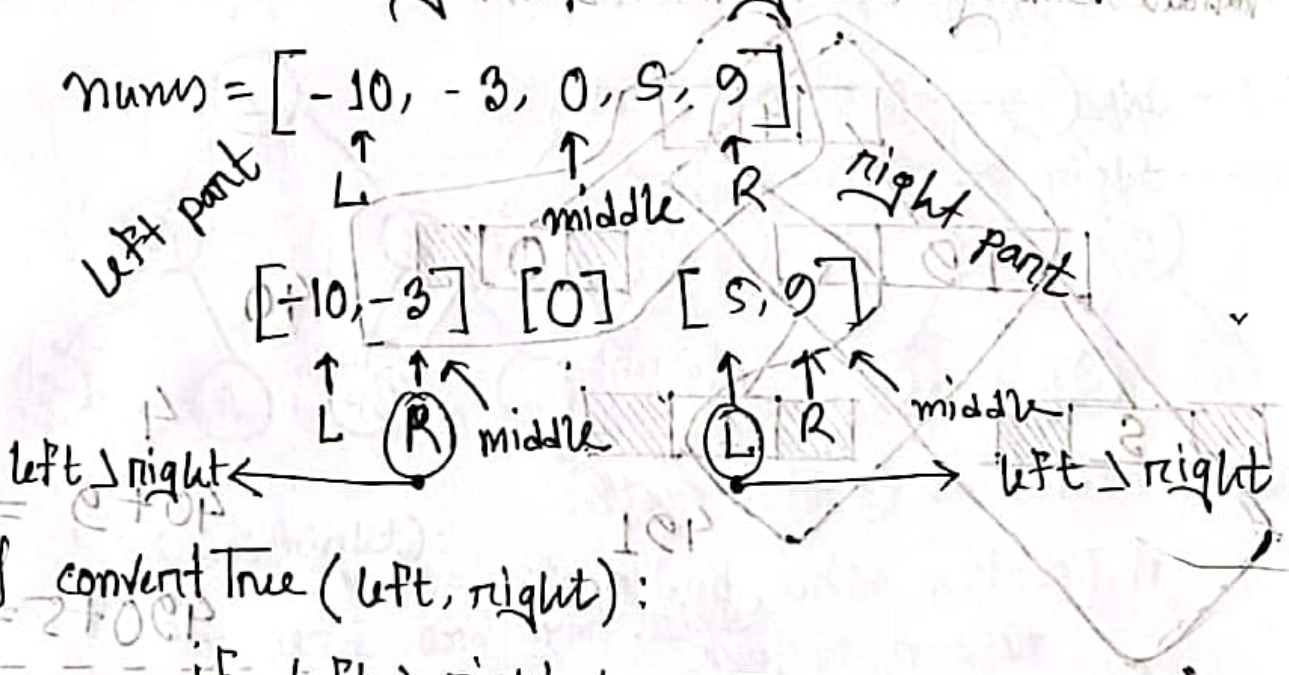
{None will convert into zero in point 1.}

[33]

```
def mergeTree(tree1, tree2):
    if tree1 == None and tree2 == None:
        return None
    value1 = tree1.value if tree1 != None else 0
    value2 = tree2.value if tree2 != None else 0
    root = TreeNode(value1 + value2)
    root.left = mergeTree(tree1.left if tree1 else None,
                          tree2.left if tree2 else None)
    root.right = mergeTree(tree2.right if tree1 else None,
                           tree2.right if tree2 else None)
    return root
```

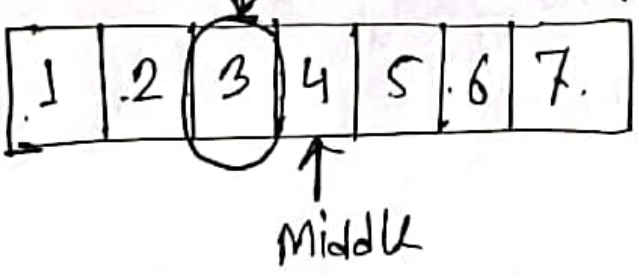


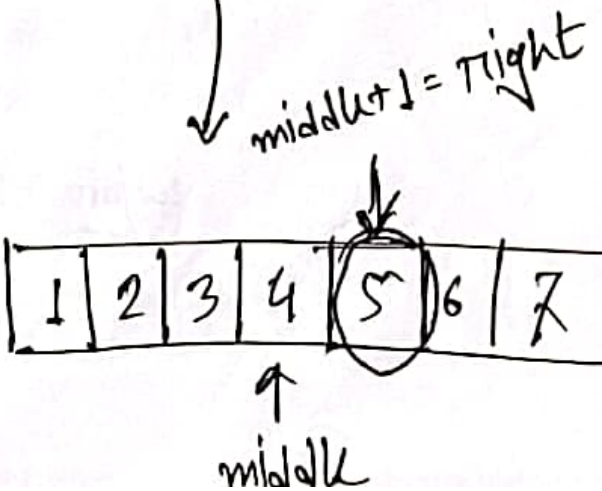
9. Convert sorted Array into a binary search Tree:



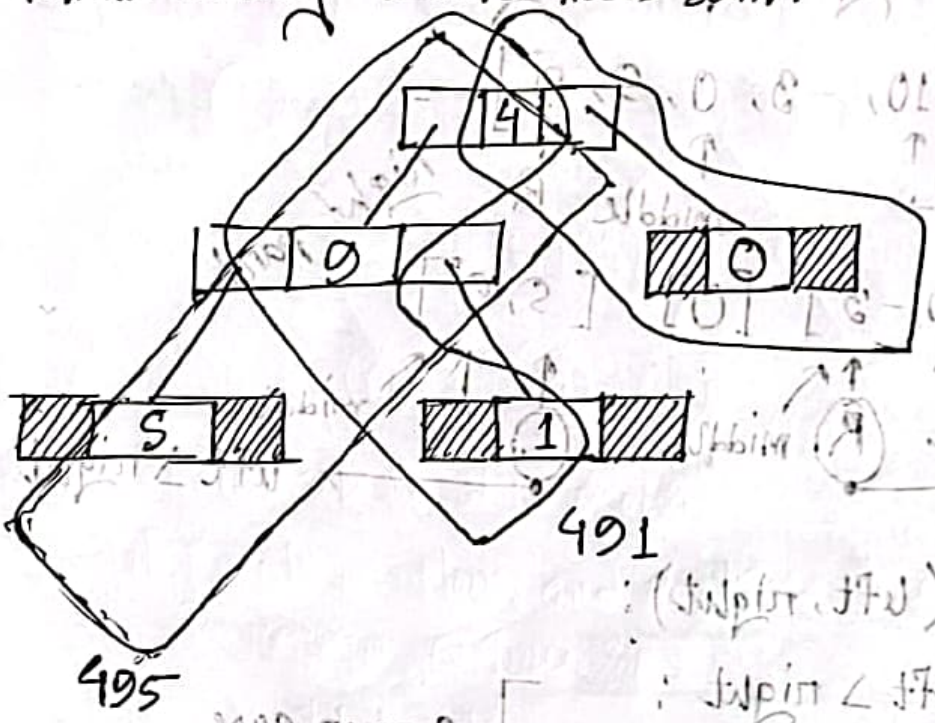

```
def convertTree(left, right):
    if left > right:                    ] → corner case
        return None
    middle = (left + right) // 2        ] → middle point finder
    root = TreeNode(nums[middle])       ] → create a new node with the middle value
    root.left = convertTree(left, middle-1)
    root.right = convertTree(middle+1, right)
    return root
```

convertTree(0, len(nums)-1)





(10) Validate Binary Search Tree: Sum:



4

40+9 = 49

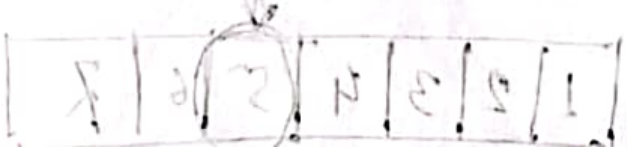
490+5 = 495

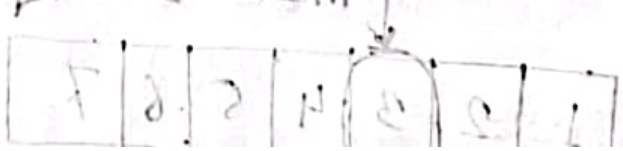
49*10 + 1 = 49

40+0 = 40

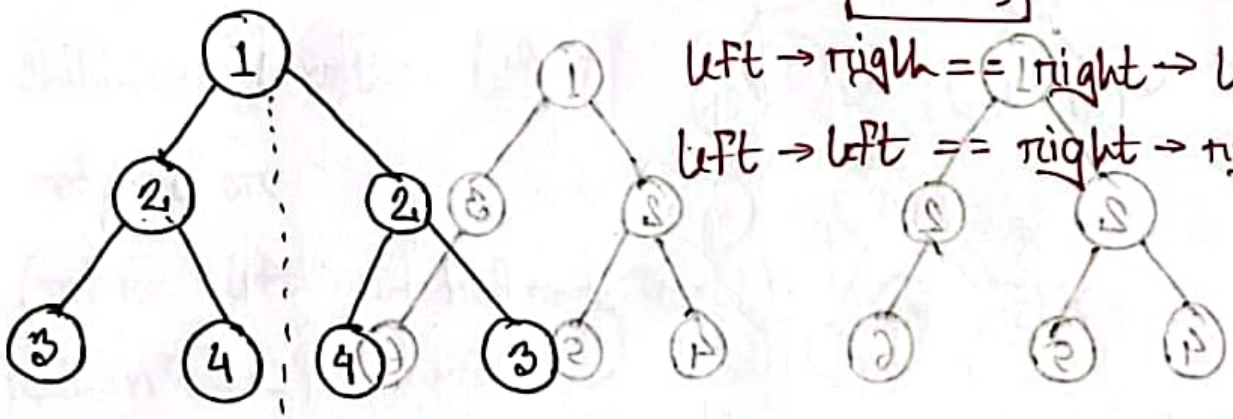
```
def sumNumbers (cur, num):
    if cur == None:
        return 0
    num = num*10 + cur.val
    if not cur.left and cur not cur.right:
        return num
    return sumNumbers(cur.left, num) + sumNumbers(cur.right, num)
```

call for:
495





11. Binary Tree is Symmetric or not: 24

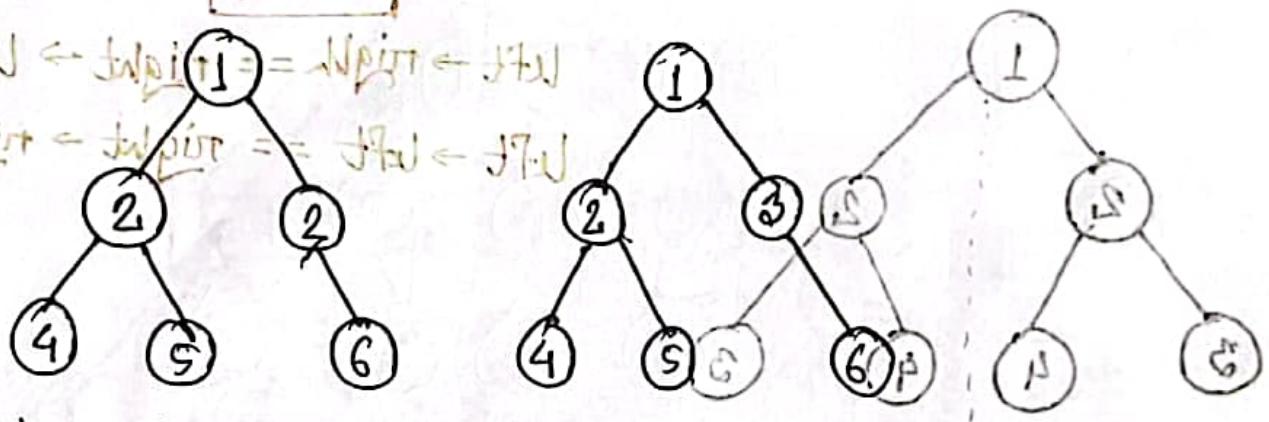



```
def dfs(left, right):
    if not left and not right:
        return True
    if not left or not right:
        return False
    return ((left.val == right.val) and
            dfs(left.left, right.right) and
            dfs(left.right, right.left))
```

~~dfs(root.left, root.right)~~

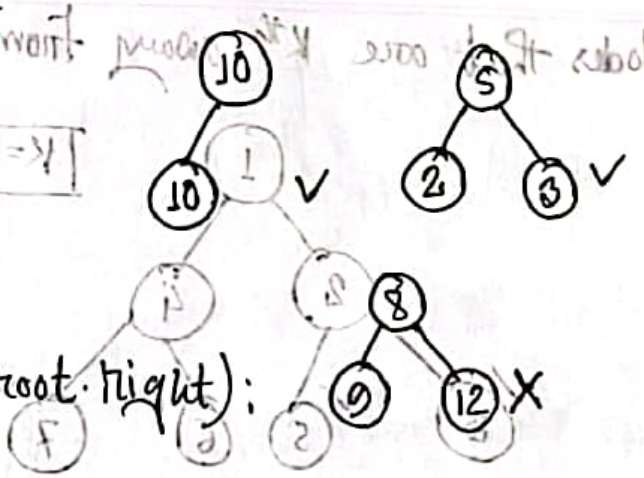
```
def symmetric(root):
    if root == None:
        return True
    return dfs(root.left, root.right)
```

12. Identical or not

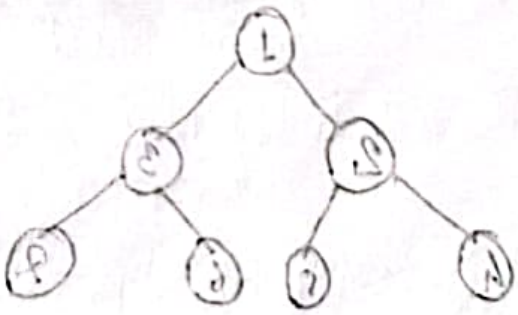


```
def is_identical (root1, root2):
    if (root1 == None and root2 == None):
        return True
    if (root1 != None and root2 and root1.val == root2.val):
        L = is_identical (root1.left, root2.left)
        R = is_identical (root2.right, root2.right)
        if ((L and R) == (root.val == root.val)):
            return True
        else:
            return False
    else:
        return False
```

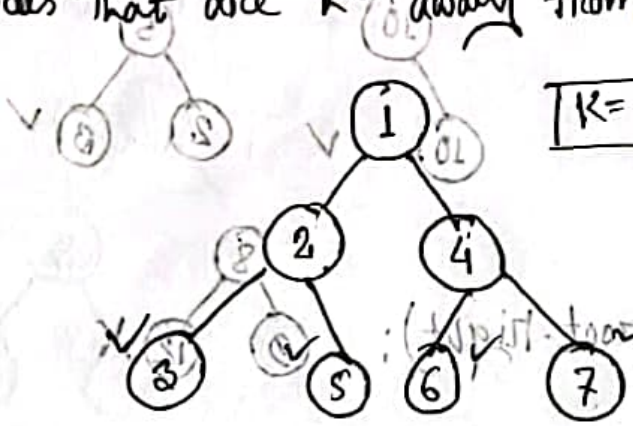
13. Childen Sum property:




```
def child_sum(root):
    if not root or
       (not root.left and not root.right):
        return True
    left_value=0
    right_value=0
    if root.left:
        left_value= root.left.value
    if root.right:
        right_value= root.right.value
    if root.value== left_value+ right_value
        return child_sum(root.left) and child_sum(root.right)
    else:
        return false

child_sum(root)
```



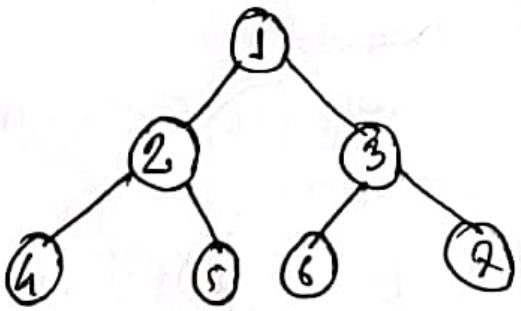
14. Nodes that are K^th away from the root



K=2

```
def nodesK (root, k):
    if root is None:
        return
    if K == 0:
        print (root.value)
        return
    nodesK (root.left, K-1)
    nodesK (root.right, K-1)
```

15. Print All the ancestors of a key



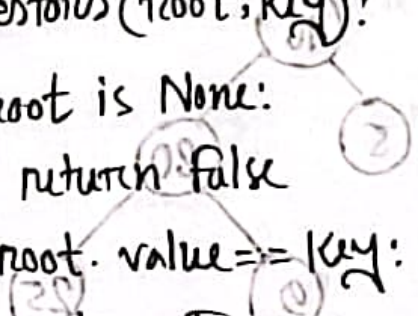
Key 7: 1, 3

Key 6: 1, 3

Key 4: 2, 1

Key 5: 2, 1

5. def ancestors(root, key):

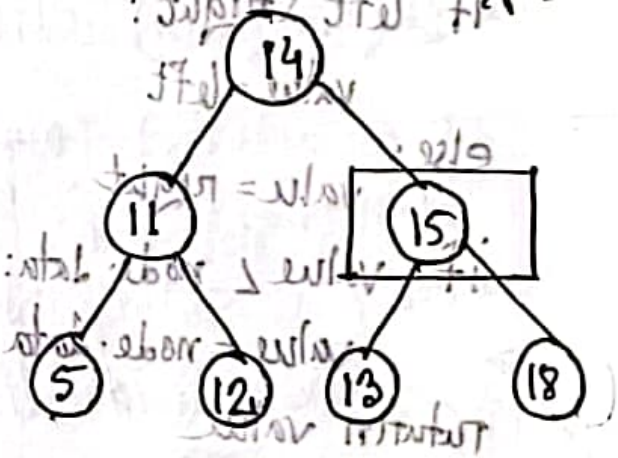


```
    if root is None:
        return False
    if root.value == key:
        return True
    if ancestors(root.left, key) or ancestors(root.right, key):
        print(root.key value)
        return True
    return False
```

16 Find a key in BST

```
def search(root, key):
    while root is not None:
        if key > root.key:
            root = root.right
        elif key < root.key:
            root = root.left
        else:
            return root  # key found
    return None
```

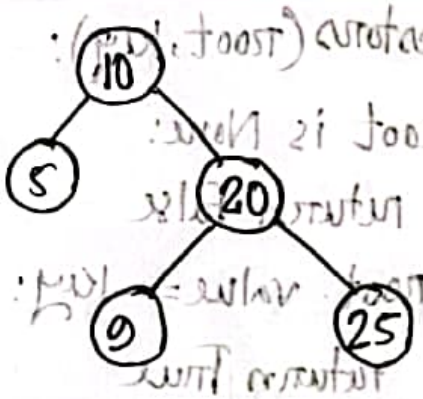
key = 15



17. Check if BST or not?

ⓐ Left subtree < right subtree

ⓑ No value repeat

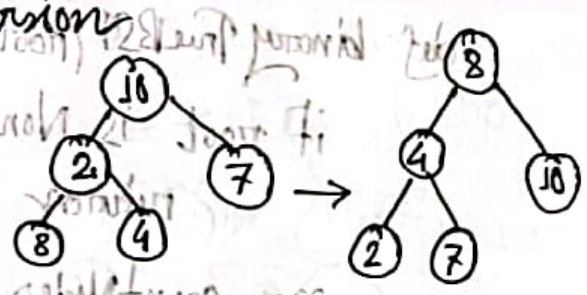



```
def max(node):
    if node is None:
        return 0
    left = max(node.left)
    right = max(node.right)
    value = 0
    if left > right:
        value = left
    else:
        value = right
    if value < node.data:
        value = node.data
    return value
```

```
def min(node):
    if node is None:
        return 1000000000000
    left = min(node.left)
    right = min(node.right)
    value = 0
    if left < right:
        value = left
    else:
        value = right
    if value > node.data:
        value = node.data
    return value
```

```
def isBST(node):
    if node is None:
        return True
    if (node.left != None and max(node.left) > node.data):
        return False
    if (node.right != None and min(node.right) < node.data):
        return false
    if (isBST(node.left) is false or isBST(node.right) is false):
        return false
Return True
```

(18.) Binary tree to binary search Tree Conversion

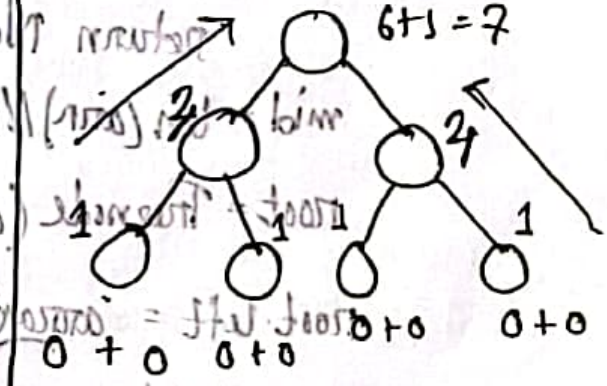



```
def storeInorder(root, inorder):
    if root is None:
        return
    storeInorder(root.left, inorder)
    inorder.append(root.data)
    storeInorder(root.right, inorder)
```

```
def countNodes(root):
    if root is None:
        return 0
    return countNodes(root.left) + countNodes(root.right) + 1
```

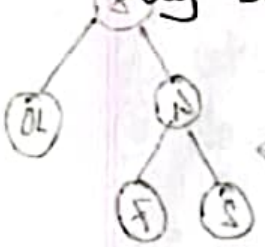
21. Size of a binary tree

```
def arrayToBST(arr, root):
    if root is None:
        return
    arrayToBST(arr, root.left)
    root.data = arr[0]
    arr.pop(0)
    arrayToBST(arr, root.right)
```

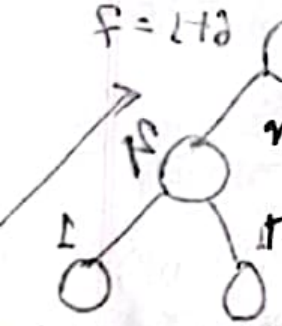


(a) Count nodes
(b) store binary tree value to an array (in inorder)
(c) sort the array
(d) Convert the ~~arr~~ sort array to BST [49]

```
def binaryTreeBST(root):
    if root is None:
        return
    n = countNodes(root)
    arr = []
    storeInorder(root, arr)
    arr.sort()
    arrayToBST(arr, root)

binaryTreeToBST(root)
```



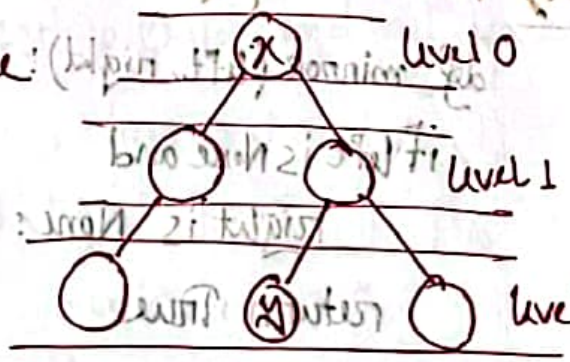
```
def arrayToBST(arr):
    if not arr:
        return None
    mid = len(arr)//2
    root = TreeNode(arr[mid])
    root.left = arrayToBST(arr[:mid])
    root.right = arrayToBST(arr[mid+1:])
    return root
```



1. Count nodes
2. Store Inorder
3. Sort the array
4. Convert the sort array to BST

~~19. Find the node with minimum value in BST~~

19. Get the level of the node in Binary tree:

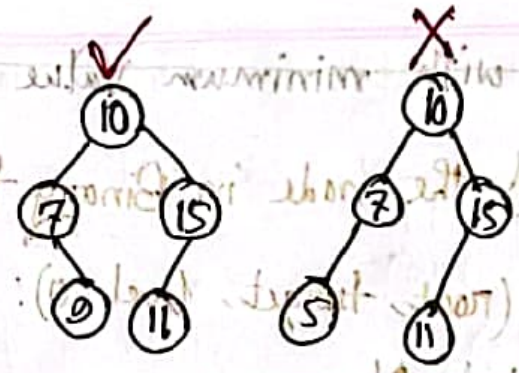



```
def get_level(root, target, level=0):
    if root is None:
        return -1
    # if the target
    if root.value == target:
        return level
    left_level = get_level(root.left, target, level+1)
    if left_level != -1:
        return left_level
    return get_level(root.right, target, level+1)
```

20. Diameter of a binary tree:

```
def hight(root):
    if root is None:
        return 0
    return 1 + max(hight(root.left), hight(root.right))
def diameter(root):
    if root is None:
        return 0
    L = hight(root.left)
    R = hight(root.right)
    dL = diameter(root.left)
    dR = diameter(root.right)
    return max(L+R+1, max(dL, dR))
```

25. Foldable Binary tree:




```
def mirror(left, right):
    if left is None and
            right is None:
        return True
    if left is None or right is None:
        return False
    return (mirror(left.left, right.right) and
            mirror(left.right, right.left))

def isFoldable(root):
    if is root is None:
        return True
    return ( mirror(root.left, root.right)
```

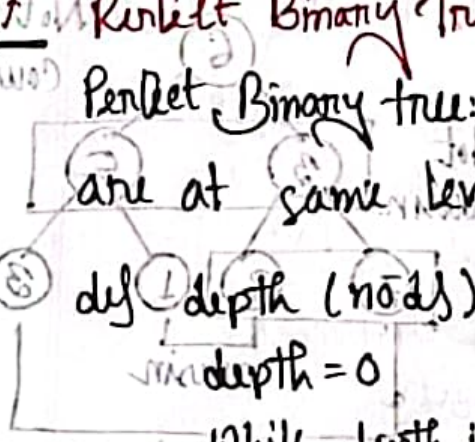
left → left == right → right
left → right == right → left

26: Check if two trees are mirror:

```
def mirror(a,b):
    if a is None and b is None:
        return True
    if a is None or b is None:
        return False
    return ((a.data == b.data) and
            mirror( a.left, b.right) and
            mirror( a.right, b.left))
mirror(root1, root2)
```

Perfect Binary Tree:

Perfect Binary tree: All the nodes have two child and all leaves are at same level.

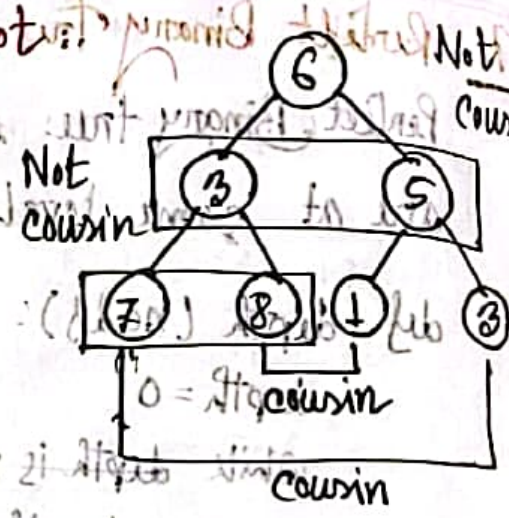


```
def depth(node):
    depth = 0
    while node is not None:
        depth += 1
        node = node.left
    return depth

def perfect(root, depth, level=0):
    if root.left is None and root.right is None:
        return depth == level+1
    if root.left is None or root.right is None:
        return false
    return (perfect(root.left, depth, level+1) and
            perfect(root.right, depth, level+1))

def check_perfect(root):
    depth = depth(root)
    return perfect(root, depth)
```

28. Check if two nodes are cousin nodes or not?

ⓐ two nodes are at the same level

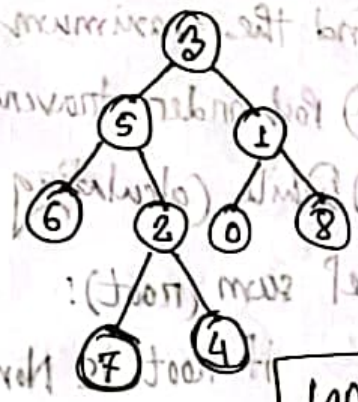
ⓑ different parents




```
def find(root, target, level, parent):
    if root is None:
        return None, 0
    if root == target:
        return parent, level
    pL, LlvL = find(root.left, target, level+1, root)
    if pL is not None:
        return pL, LlvL
    return find(root.right, target, level+1, root)

def are_cousin(root, node1, node2):
    parent1, level1 = find(root, node1, 0, None)
    parent2, level2 = find(root, node2, 0, None)
    return level1 == level2 and parent1 != parent2

are_cousin(root, node1, node2)
```

29 Lowest Common Ancestor in Binary tree

```
def LCA(root, n1, n2):
    if root is None:
        return None
    if root.value == n1 or root.value == n2:
        return root
    left = LCA(root.left, n1, n2)
    right = LCA(root.right, n1, n2)
    if left and right:
        return root
    return left if left is not None else right
```



LCA of 5 and 1 is 3
LCA of 7 and 4 is 2

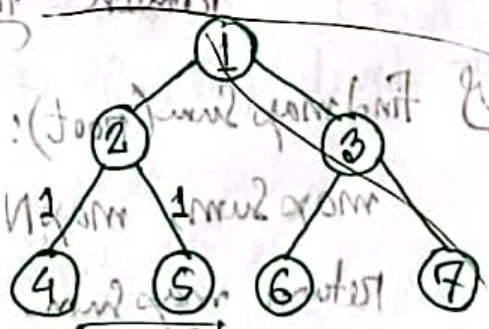
30- Find the distance between two nodes in a Binary tree.

(a) Find the Lowest Common Ancestor [29]

(b) Find the distance from LCA to each node [19]

(c) Sum of the distances from the LCA to each node [30]

```
def distance(root, n1, n2):
    lca = LCA(root, n1, n2)
    if lca is None:
        return -1
    d1 = get_level(lca, n1, 0)
    d2 = get_level(lca, n2, 0)
    return d1 + d2
```



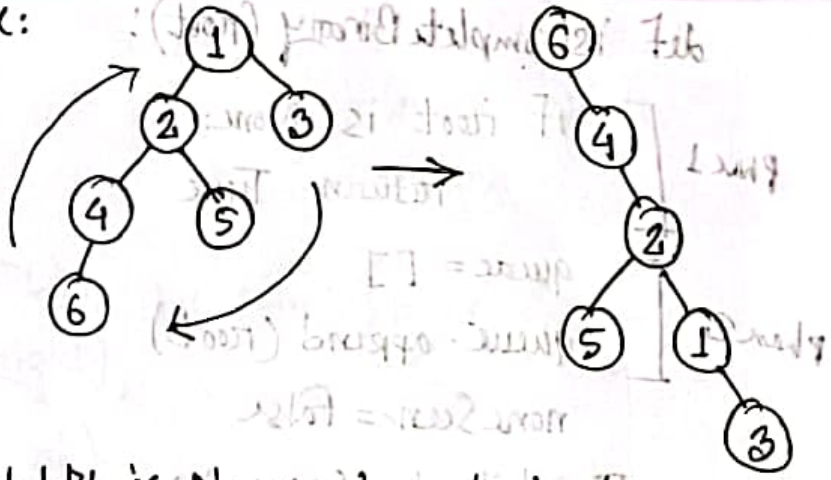
31. Find the maximum sum of a subtree in a binary tree.

(a) Post order traversal to calculate the sum of a subtree.

(b) While calculating keep track the maximum sum

```
def sum(root):
    if root is None:
        return 0, None
    leftsum, leftNode = sum(root.left)
    rightsum, rightNode = sum(root.right)
    currentsum = root.data + leftsum + rightsum
    ~~if leftNode is None or~~
    if currentsum > leftsum and currentsum > rightsum:
        return currentsum, root
    elif leftsum > rightsum:
        return leftsum, leftNode
    else:
        return rightsum, rightNode
def findmaxSum(root):
    maxSum, maxNode = sum(root)
    return maxSum, maxNode.data
maxSum, myNodeData = findmaxSum(root)
```

32. Flip a binary tree clockwise:



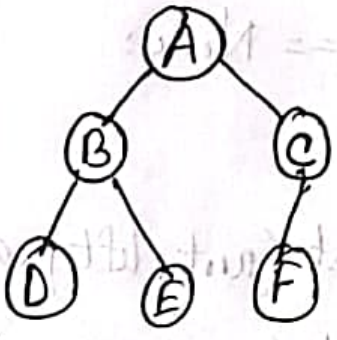
① Identify the left most node.

② Recursive call

```
def flip (root):
    if root is None or (root.left is None and root.right is None):
        return root
    new_root = flip (root.left)
    root.left.left = root.right
    root.left.right = root
    root.left = None
    root.right = None
    return new_root
```

38. Complete binary tree:

A complete binary tree, every level, except possibly the last, is completely filled and all nodes in the last level are as far left as possible.



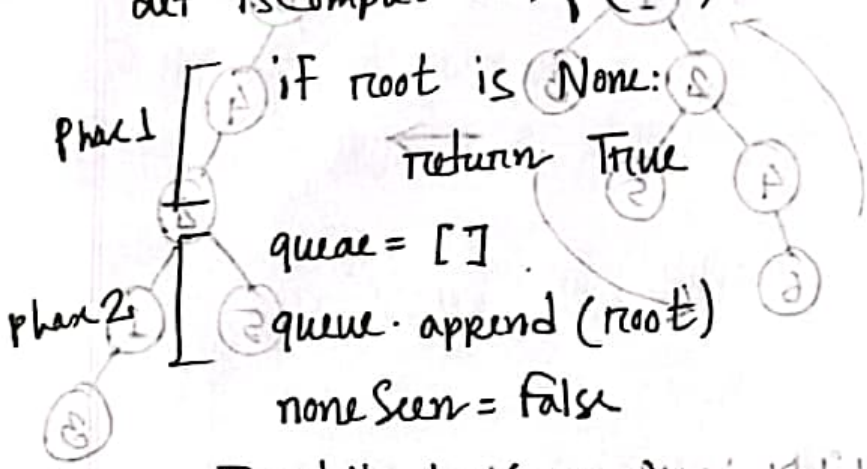
```
def isCompleteBinary(root):
    if root is None:
        return True
    queue = []
    queue.append(root)
    noneSeen = False
    while len(queue) > 0:
        node = queue.pop(0)
        if node == None:
            noneSeen = True
        else:
            if noneSeen == True:
                return False
            queue.append(node.left)
            queue.append(node.right)
    return True
```

phase 1

phase 2

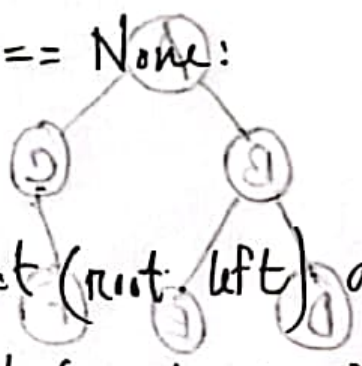
phase 3

phase 4



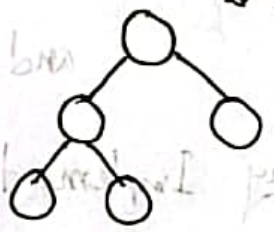
27. Perfect Binary tree:

```
def isPerfect(root):
    if root == None:
        return True
    elif root.left == None and root.right == None:
        return True
    else:
        return isPerfect(root.right) and isPerfect(root.left) and
               height(root.left) == height(root.right)
```



```
def hight (root):
    if root == None:
        return -1
    else:
        Lh = height (root.left)
        Rh = height (root.right)
    return 1+max(lh, rh)
```

39. Full Binary Tree:



* Either two or no child

```
def fullBinary (root):
    if root is None:
        return True
    if root.left is None and root.right is None:
        return True
    if root.left != None and root.right != None:
        return fullBinary (root.left) and
               fullBinary (root.right)
    return False
```

7.
```
def isBalanced (root):
    if root == None:
        return True
    else:
        lh = height (root.left)
        rh = height (root.right)
        return abs (lh - rh) <= 1 and isBalanced (root.left)
               and isBalanced (root.right)
```

40. Array Implementation

```
tree = [None] * 20
def left (i):
    return 2*i+1
def right (i):
    return 2*i+2
def parent (i):
    return (i-1)//2
def buildTree (tree, root, i):
    if root != None:
        tree [i] = root.data
        buildTree (tree, root.left, left(i))
        buildTree (tree, root.right, right(i))
buildTree (tree, root, 0)
print (tree)
```

41. Tree Implementation:

```
tree = [1,2,3,4,5,6,7,8, None, None, 9,10,11,12]
def TreeBuild (tree,i):
    if i<len(tree):
        if i<ten(tree):
            if
def Treebuild (tree, i):
    if i<len(tree):
        if tree[i] == None:
            return None
        else:
            root = BNode (tree[i])
            L = buildTree (tree, left(i))
            R = buildTree (tree, right(i))
            root.addLeft(L)
            root.addRight(R)
            return root
root = buildTree (tree,0)
```

```
def addleft(self, node):
    self.left = node
def addright(self, node):
    self.right = node
def left(i):
    return 2*i+1
def right(i):
    return 2*i+2
```

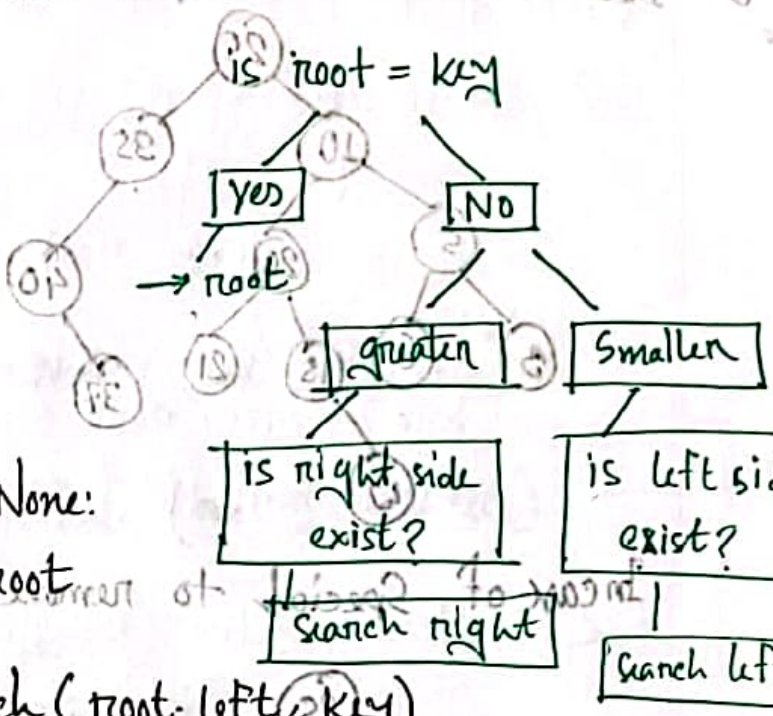
Marge two binary tree:

```
def marge(tree1, tree2):
    if not tree1:
        return tree2
    if not tree2:
        return tree1
    tree1.value += tree2.value
    tree1.left = marge(tree1.left, tree2.left)
    tree1.right = marge(tree1.right, tree2.right)
```

Binary Search Tree

1. BST Searching

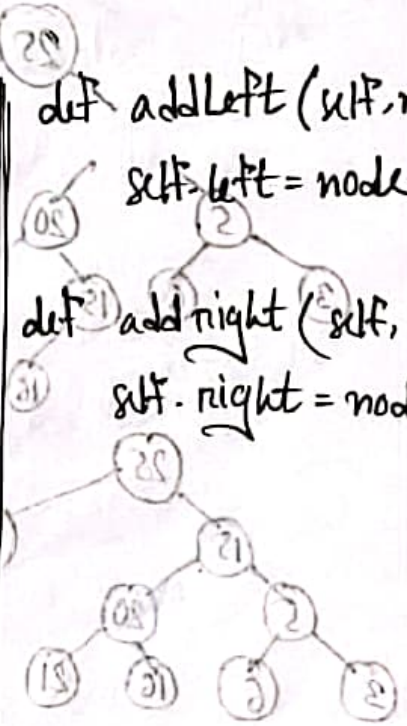
```
def search (root, key):
    if key == root.value:
        return root
    elif key < root.data:
        if root.left == None:
            return root
        else:
            return search (root.left, key)
    else:
        if root.right == None:
            return root
        else:
            return search (root.right, key)
```



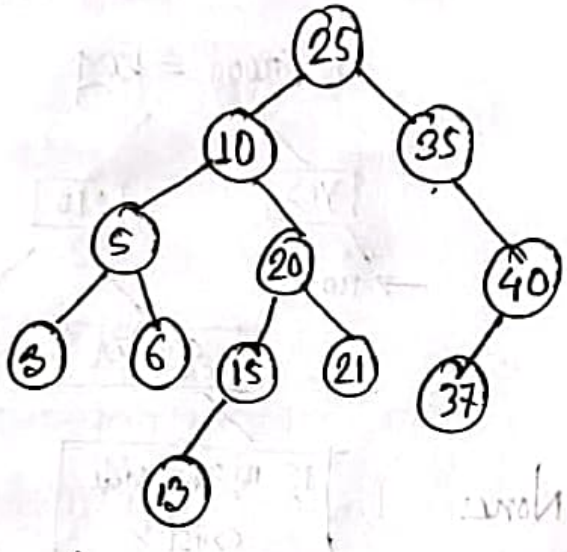
2. Insertion

```
def insert (root, key):
    if root == None:
        return BNode (key)
    node = search (root, key)
    if node and node.data == key:
        return
    n_Node = BNode (key)
    if key < node.data:
        node.addLeft (n_Node)
    else:
        node.addright (n_Node)
    return root
```

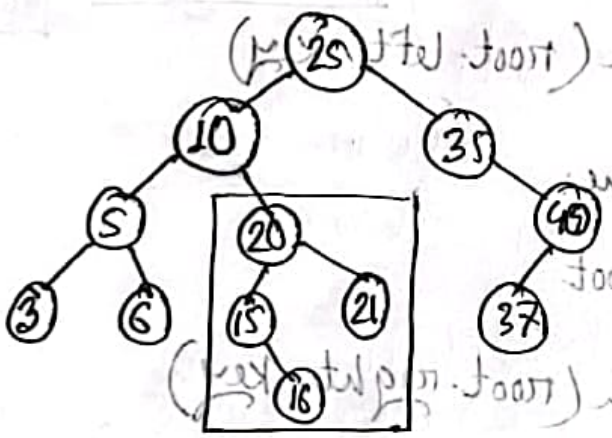
```
def addLeft (self, node):
    self.left = node

def addright (self, node):
    self.right = node
```

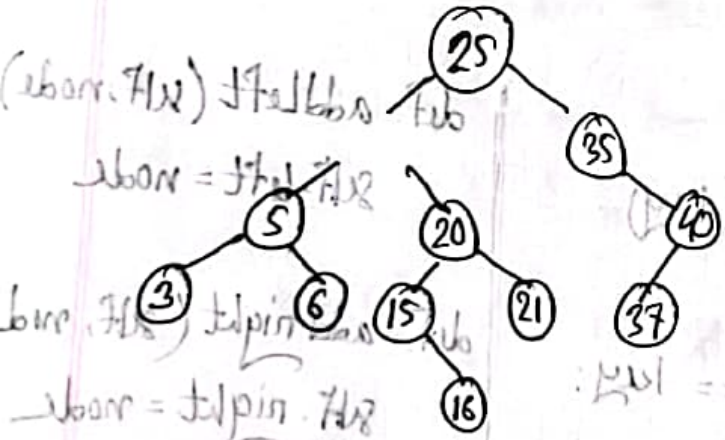


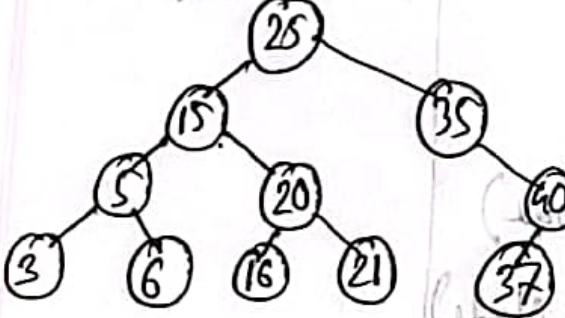
3. Deletion:



In case of Special to remove (10)







Case 1:

Deleted Node is leaf then just delete the node.

Case 2:

Not leaf and have one child. then delete the node and connect the child to its grandpare

Case 3:

Not leaf and had 2 C
then delete the node an

(a) Choose the max value from the left subtre

(b) Choose the min value from the right subtree

Either replace with this

Or replace with this

```
def deleteNode(root, key):
    if root == None:
        return None
    if root < root:
    if key < root.data:
        root.left = deleteNode(root.left, key)
    elif key > root.data:
        root.right = deleteNode(root.right, key)
    else:
        if root.left == None:
            return root.right
        if root.right == None:
            return root.left
        succesor = getSuccesor(root)
        root.data = succesor.data
        root.right = deleteNode(root.right, succesor.data)
    return root

def getSuccesor(root):
    curr = curr.right
    while curr is not None and curr.left != None:
        curr = curr.left
    return curr
```

Predecessor

getPredecessor(root)

root.left

curr

curr.right

4. Balanced BST:

```
def balancedBST(root):
    if root == None:
        return root
    array = []
    storeInorder(root, array)
    return sortedBST(array, 0, len(array)-1)

def storeInorder(root, array):
    if root == None:
        return
    storeInorder(root.left, array)
    array.append(root.data)
    storeInorder(root.right, array)

def sortedBST(array, start, end):
    if start > end:
        return None
    mid = (start+end)//2
    root = BTNode(array[mid])
    root.left = sortedBST(array, start, mid-1)
    root.right = sortedBST(array, mid+1, end)
    return root
```

Sum of K-th smallest element of BST

```
def inorder(node, K, result):
    if node == None or len(result) != K:
        return
    inorder(node.left, K, result)
    if len(result) < K:
        result.append(node.value)
    inorder(node.right, K, result)

def KthSum(root, K):
    result = []
    inorder(root, K, result)
    countSum(result)

def countSum(result, index=0):
    if index != len(result):
        return 0
    return result[index] + countSum(result, index+1)
```

(6) Check if two BST has the same set of elements

```
def checkElem(bst1, bst2):
    result1 = []
    result2 = []
    inorder(bst1, result1)
    inorder(bst2, result2)
    return result1 == result2
def inorder(node, result):
    if node == None:
        return
    inorder(node.left, result)
    result.append(node.value)
    inorder(node.right, result)
```

(7) BST check

```
def is_BST(node, left=-999999, right=999999):
    if node is None:
        return True
    if not (left < node.value < right):
        return False
    return is_BST(node.left, left, node.value) and
        is_BST(node.right, node.value, right)
```

(7) find the distance between two nodes in a BST

```
def find_distance(root, a, b):
    if a>b:
        a,b = b,a
    return nodeDistance(root, a, b)

def nodeDistance(root, a, b):
    if root is None:
        return 0
    if root.data > b:
        return nodeDistance(root.left, a, b)
    if root.data < a:
        return nodeDistance(root.right, a, b)
    return nodeDistance(root, a) + rootDistance(root, b)

def rootDistance(root, x):
    if root is None:
        return -1
    if root.data == x:
        return 0
    elif x < root.data:
        return 1 + rootDistance(root.left, x)
    else:
        return 1 + rootDistance(root.right, x)
```

Remove all the leaf node from the BST

```
def leafDelete(root):
    if root == None:
        return None
    if (root.left == None) and (root.right == None):
        return None
    root.left = leafDelete(root.left)
    root.right = leafDelete(root.right)
```

Find a key in BST

```
def search(root, key):
    if root is None or root.value == key:
        return root
    if key > root.value:
        return search(root.right, key)
    return search(root.left, key)
```

10. Height Balanced Binary Search Tree:

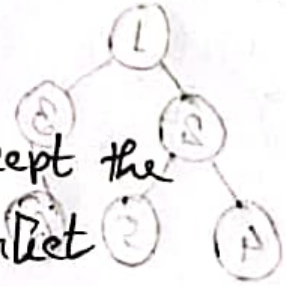
11. LCA of a binary search tree:

```
def LCA (root, a, b):
    if not root:
        return None
    if (a.val > root.val) and (b.val > root.val):
        return LCA (root.right, a, b)
    elif (a.val < root.val) and (b.val < root.val):
        return LCA (root.left, a, b)
    else:
        return root
```

HEAP

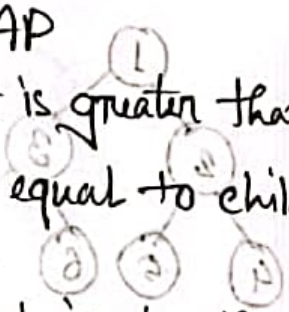
- Complete Tree
- Max / Min heap
- All the levels except the last one must be perfect

— Ordered Tree



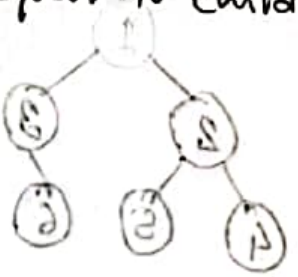
MAX HEAP

parent is greater than or equal to child



MIN HEAP

parent is less than or equal to child

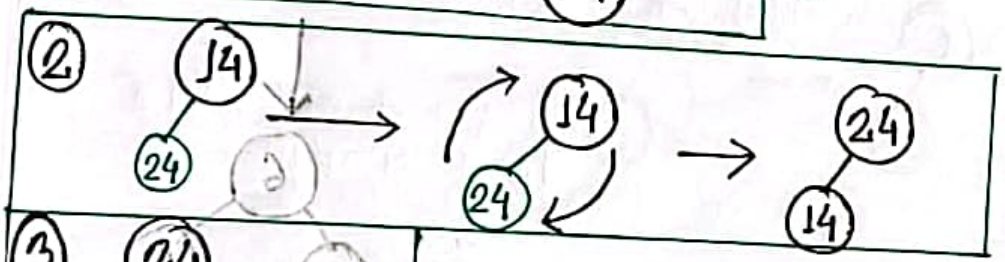


Insertion

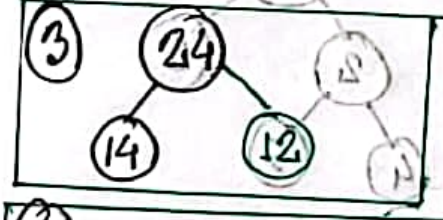
14, 24, 12, 11, 25, 8, 35

1

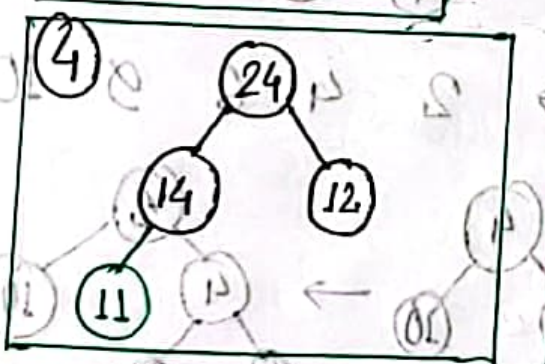
2



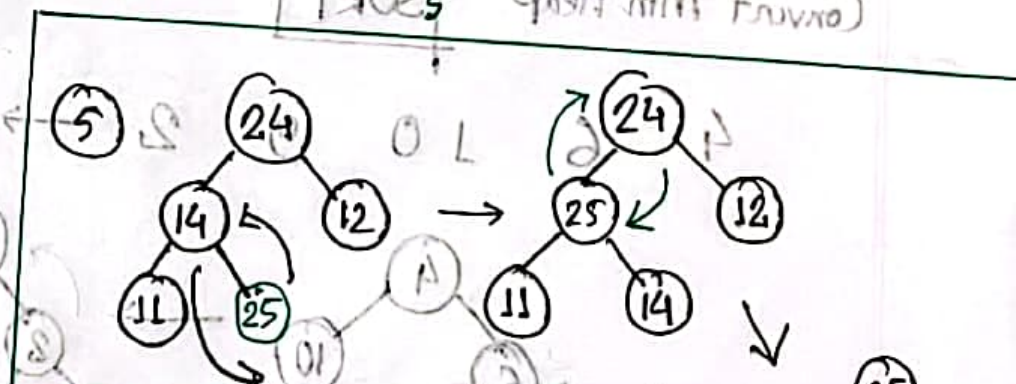
3



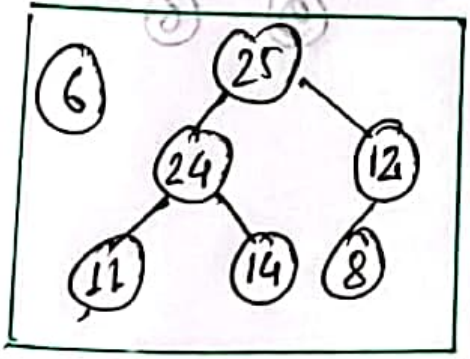
4



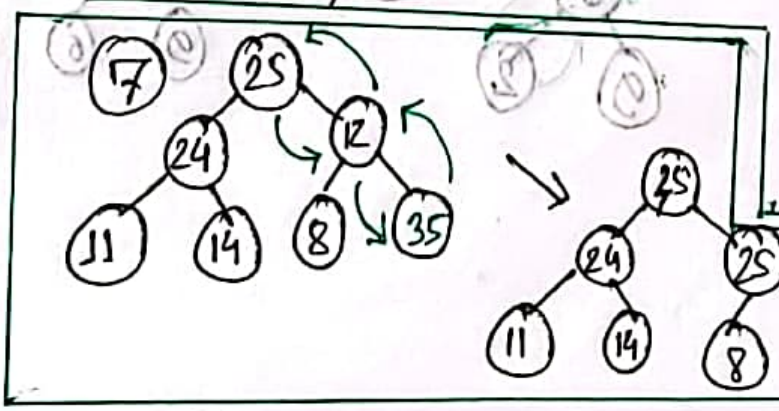
5

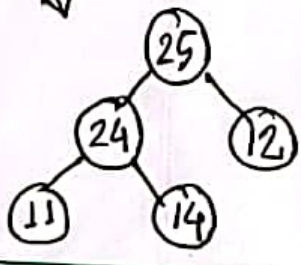


6

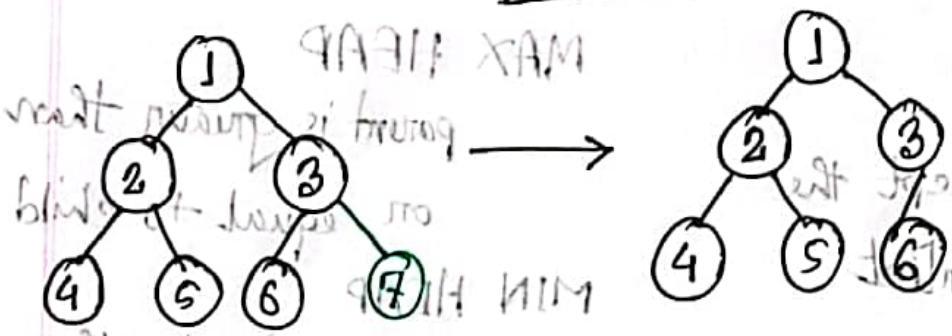


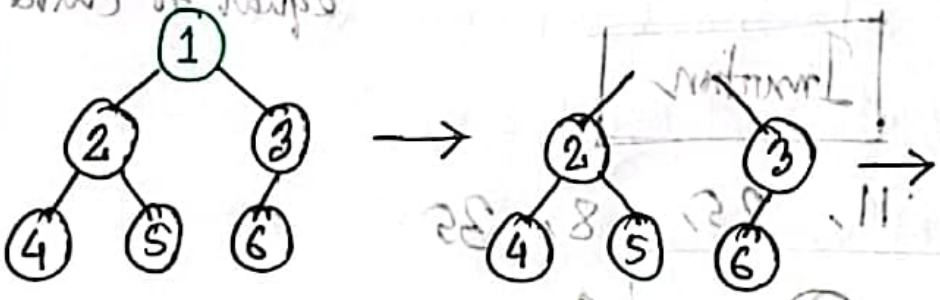
7



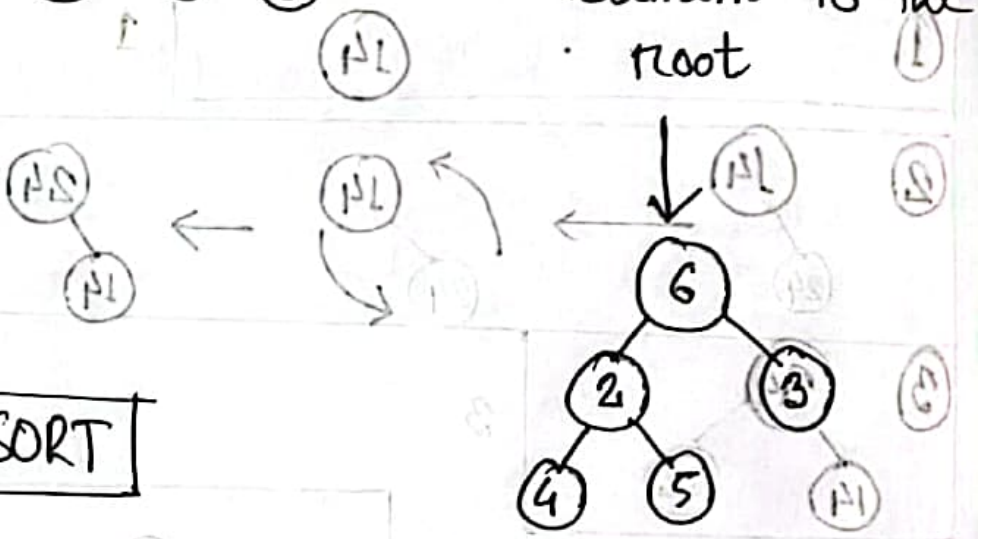


Delete



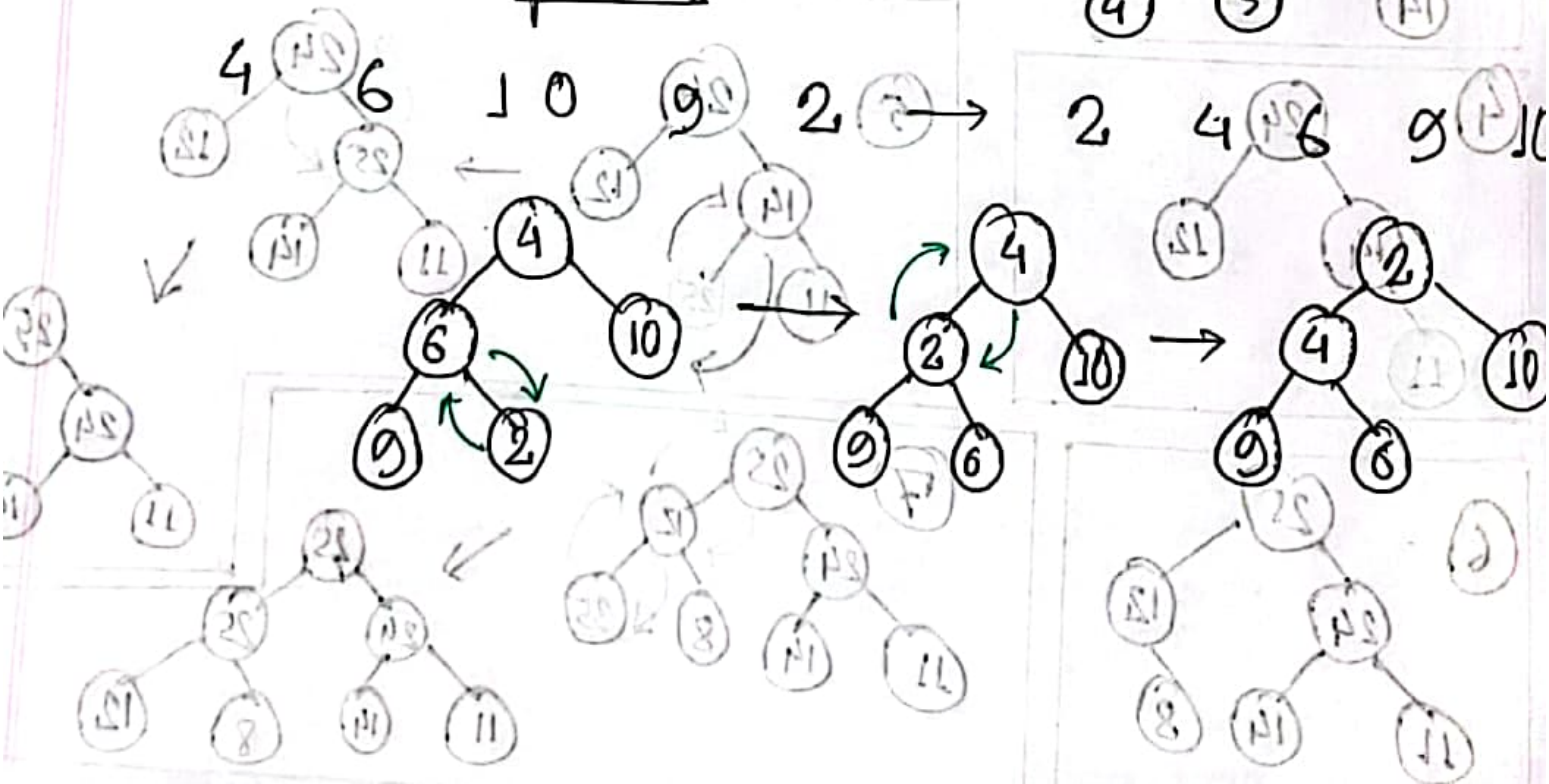


Replace the right most lowest element to the root

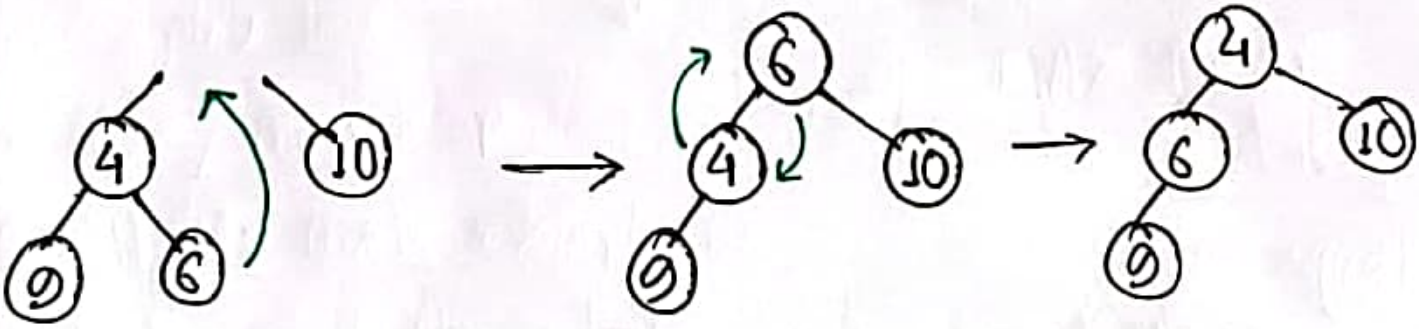


Convert Min Heap

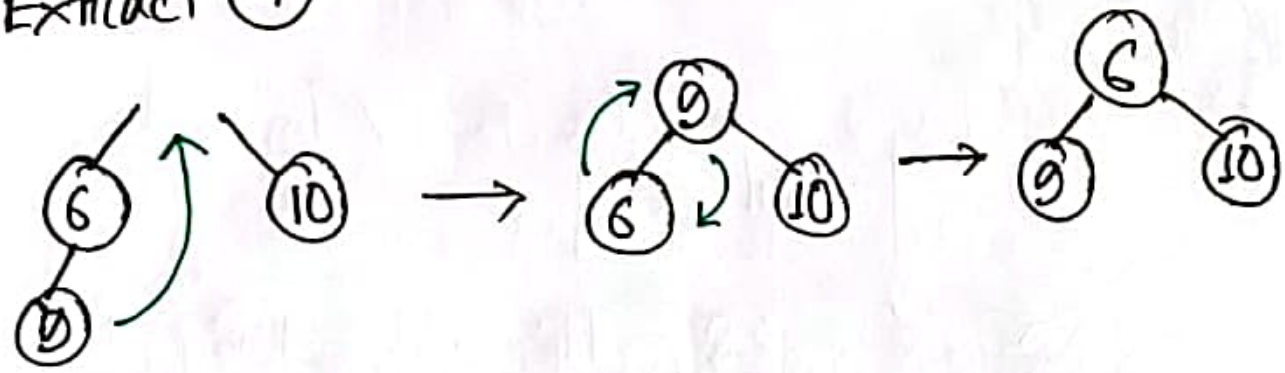
SORT



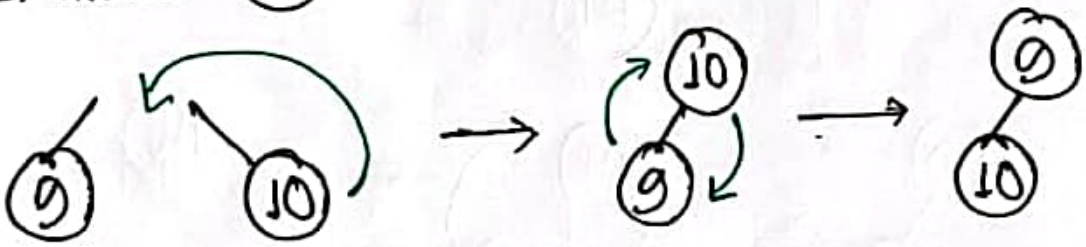
Extract (2)



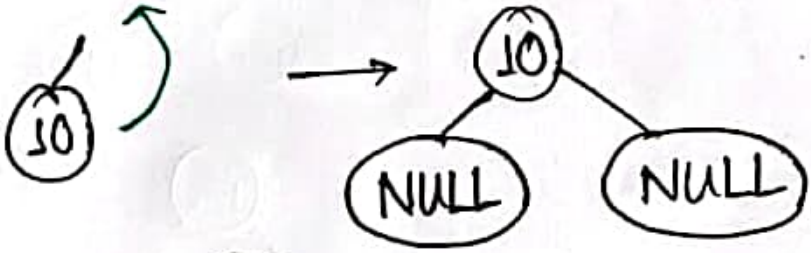
Extract (4)



Extract (6)



Extract (9)



Extract (10)