Stack | Queue — FIFO

FILO

Stack

Array

Queue

Array

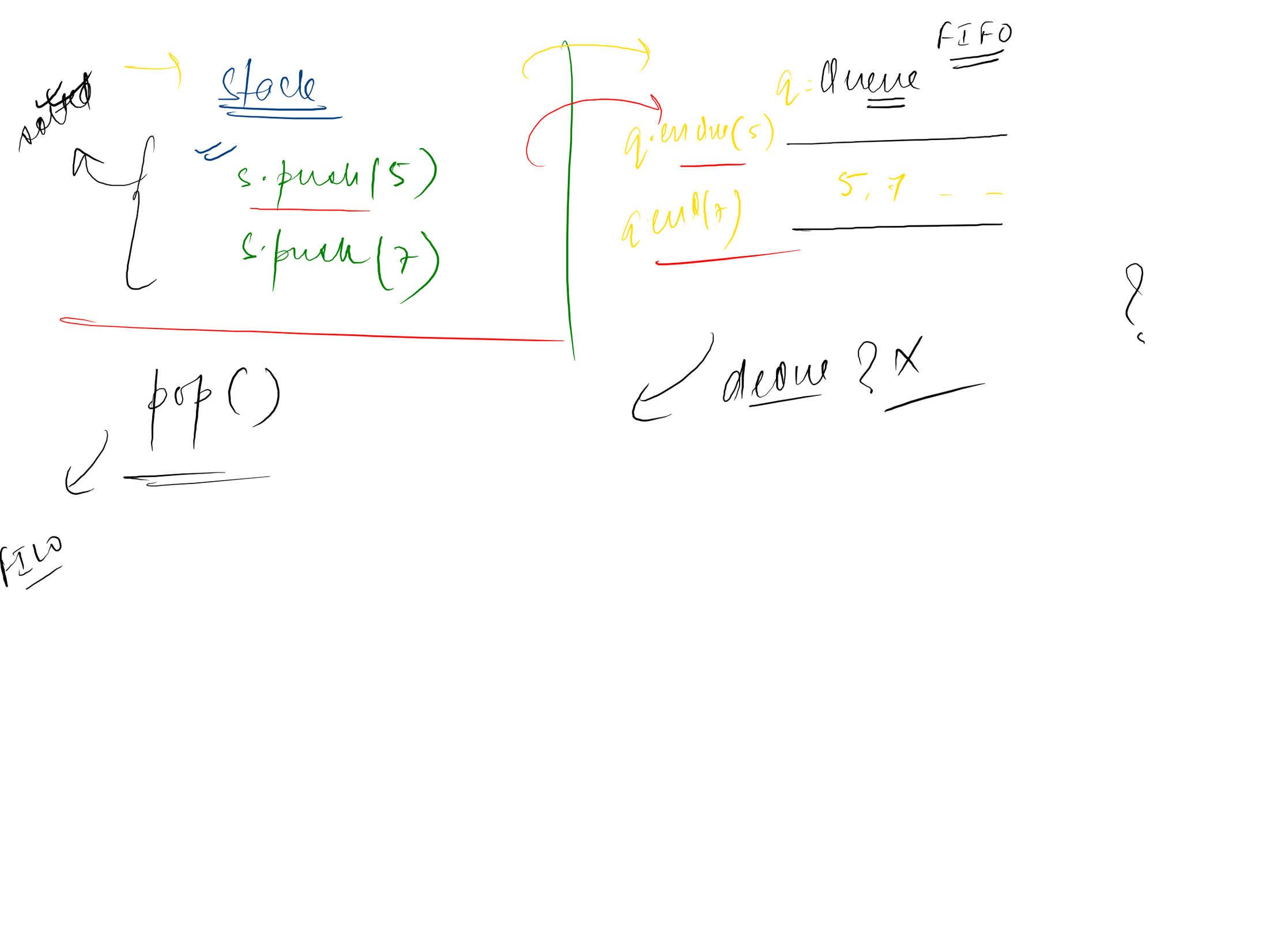
Stack way down !

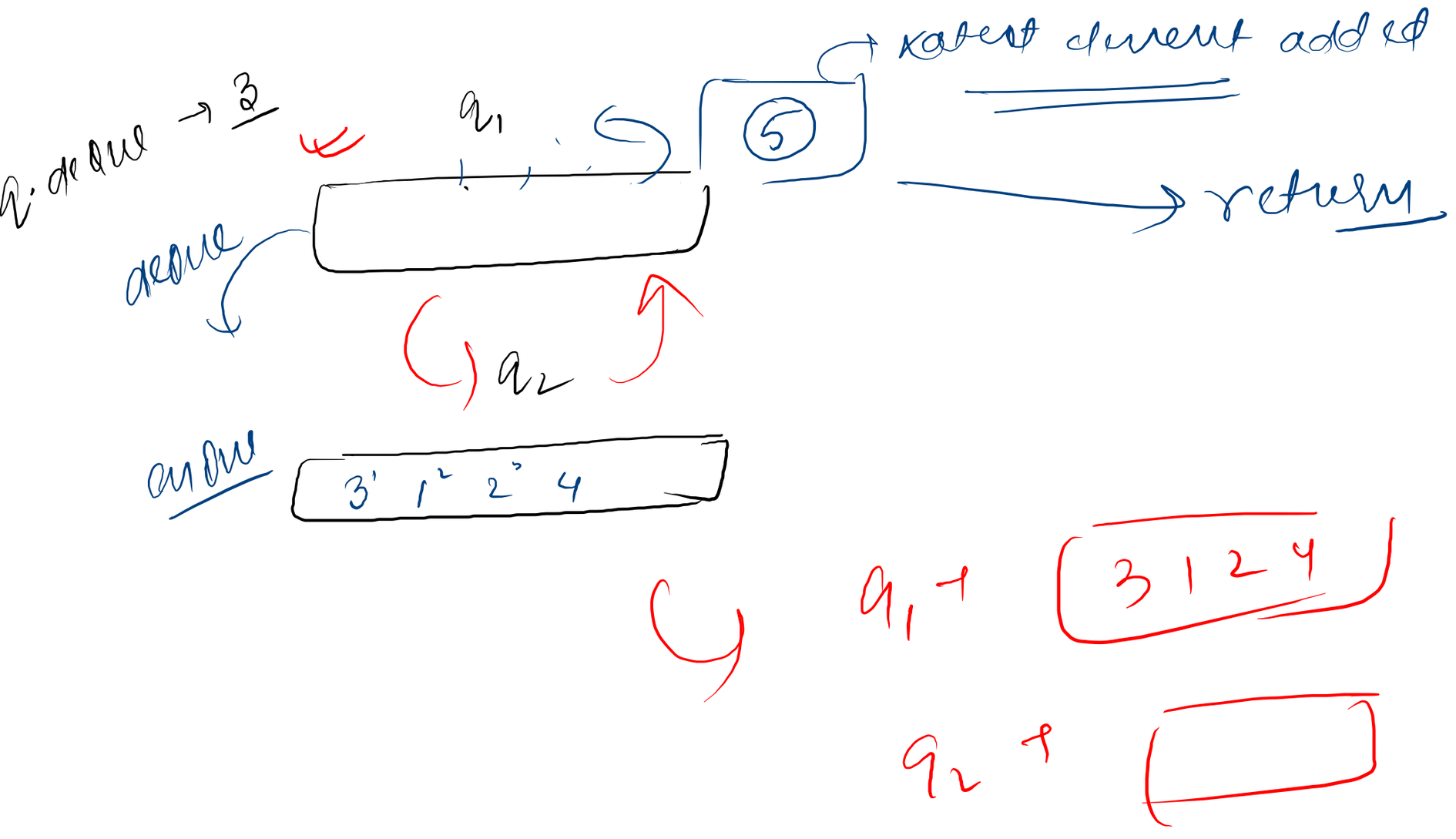
Stack

↳ push()
pop()
isEmpty

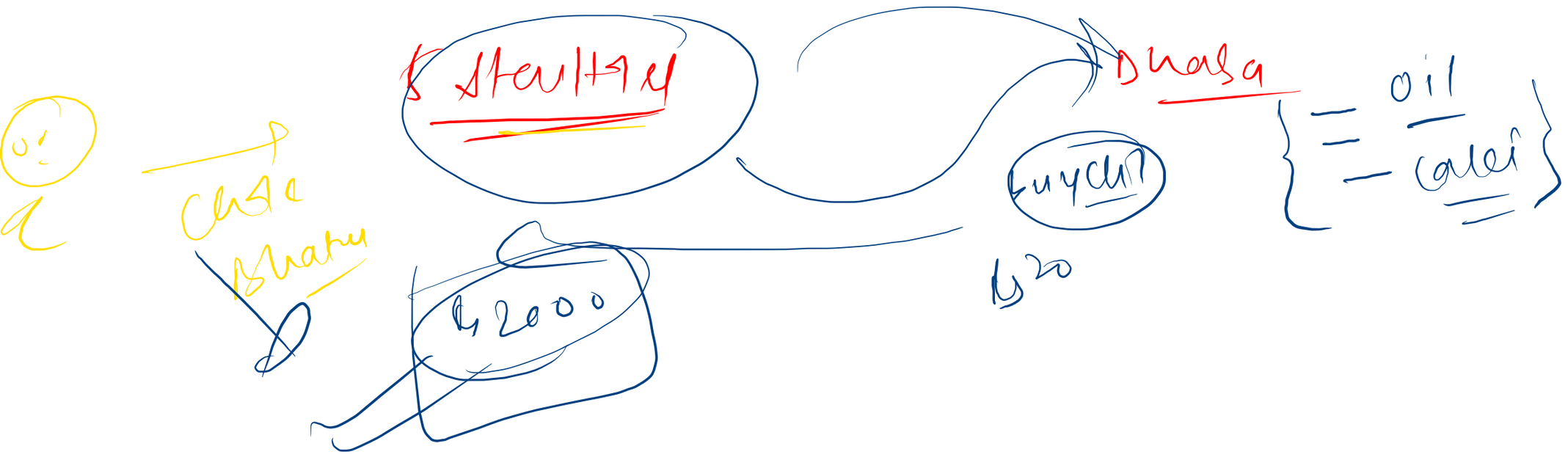
Queue

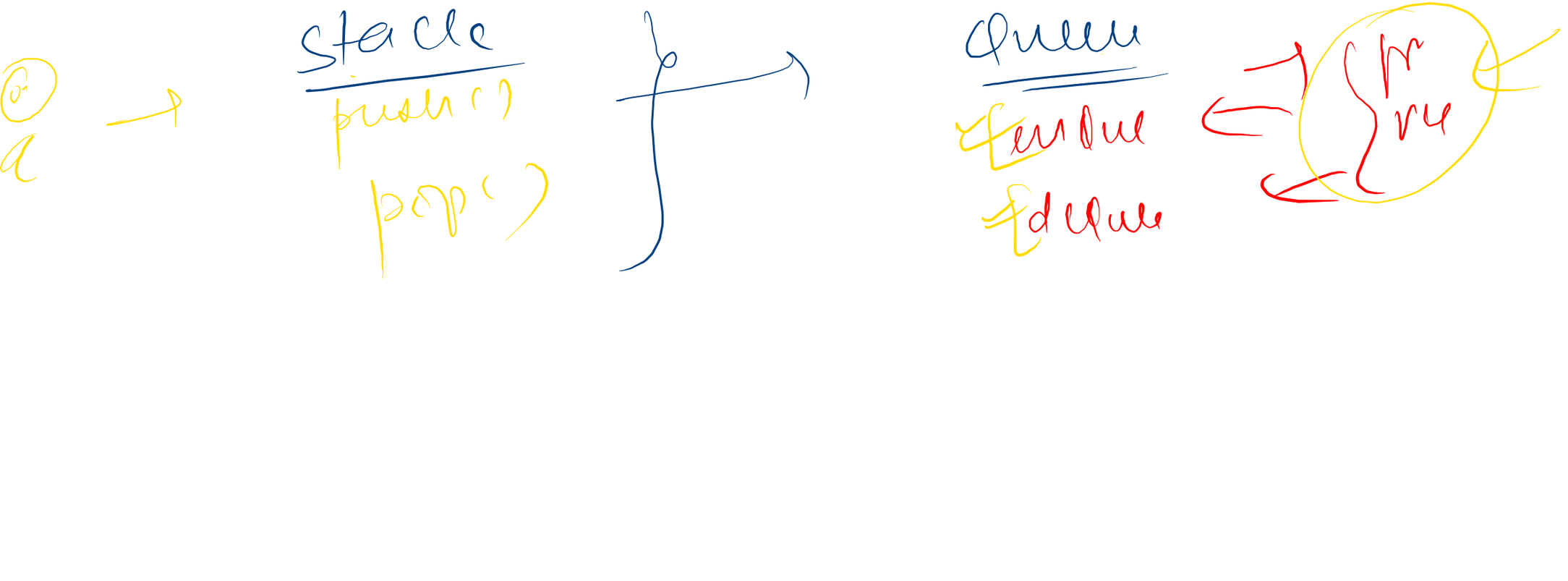
↳ enQueue()
deQueue()


stack
s.push(5)
s.push(7)
pop()
FILO
queue
q.enqueue(5)
q.enq(7)
5, 7
deque ?x
FIFO


q. queue → 3
q1
5
Value current address
return
queue
q2
queue
3 1 2 4
q1 →
3 1 2 4
q2 →


Stack
Array
fr/rear
Queue
en Queue
de Queue


Dhara
= oil
= ghee
₹20
₹2000


stack
push()
pop()
queue
enqueue
dequeue

Stacle

s.push(5)

s.push(7)

s.push(9)

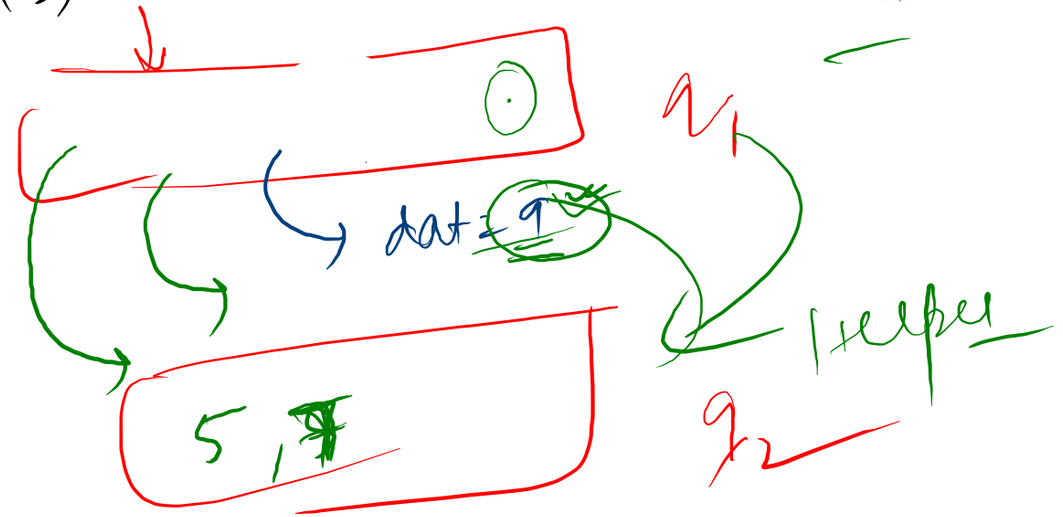
s.pop() → 9

peek

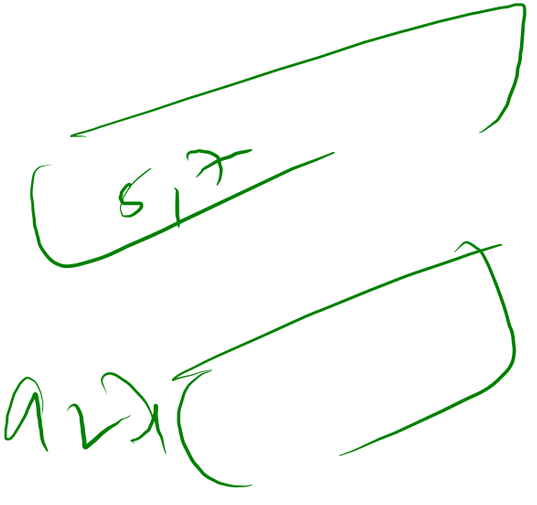
q.enq(5)

q.enq(7)

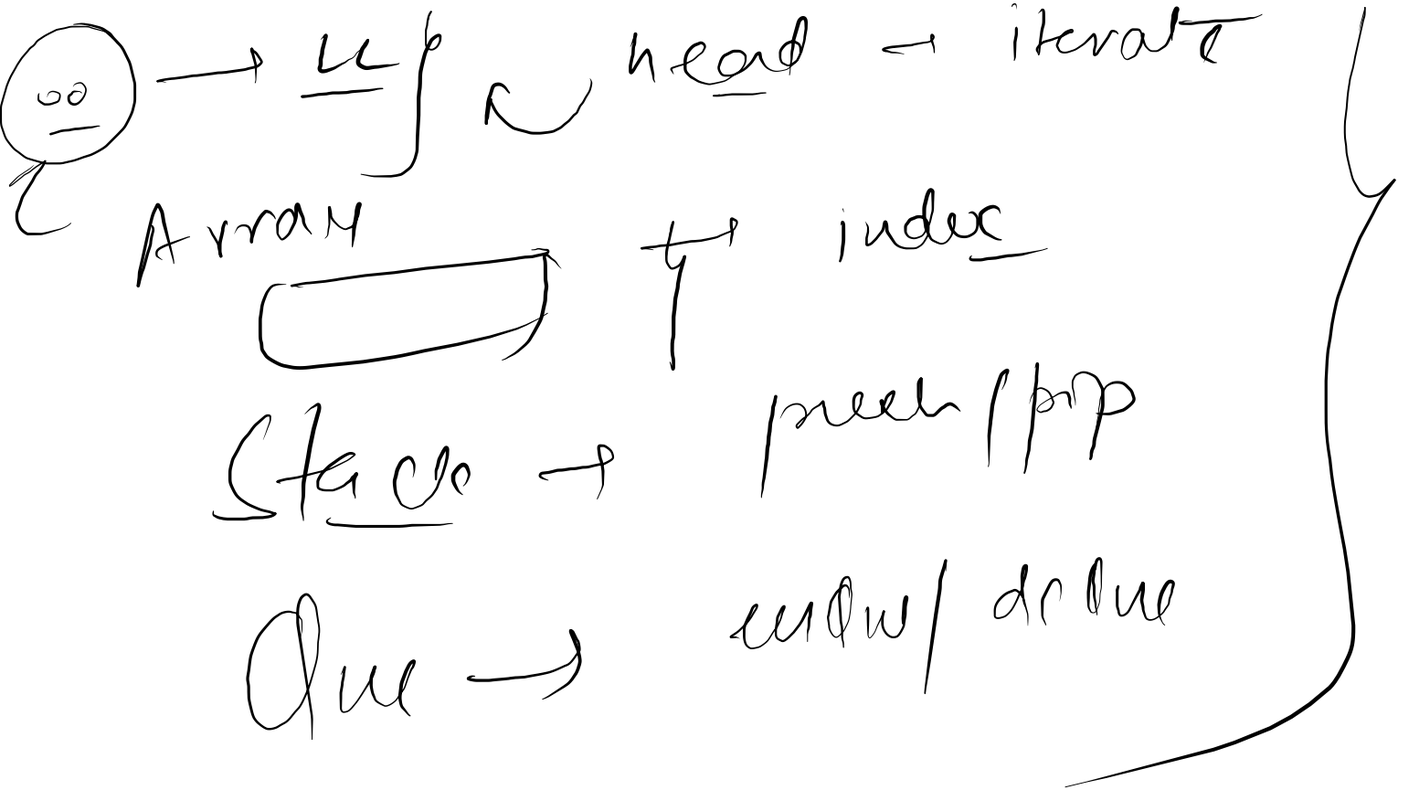
q.enq(9)



q1



Problem } S / Q ↳ S can be implemented using Q }

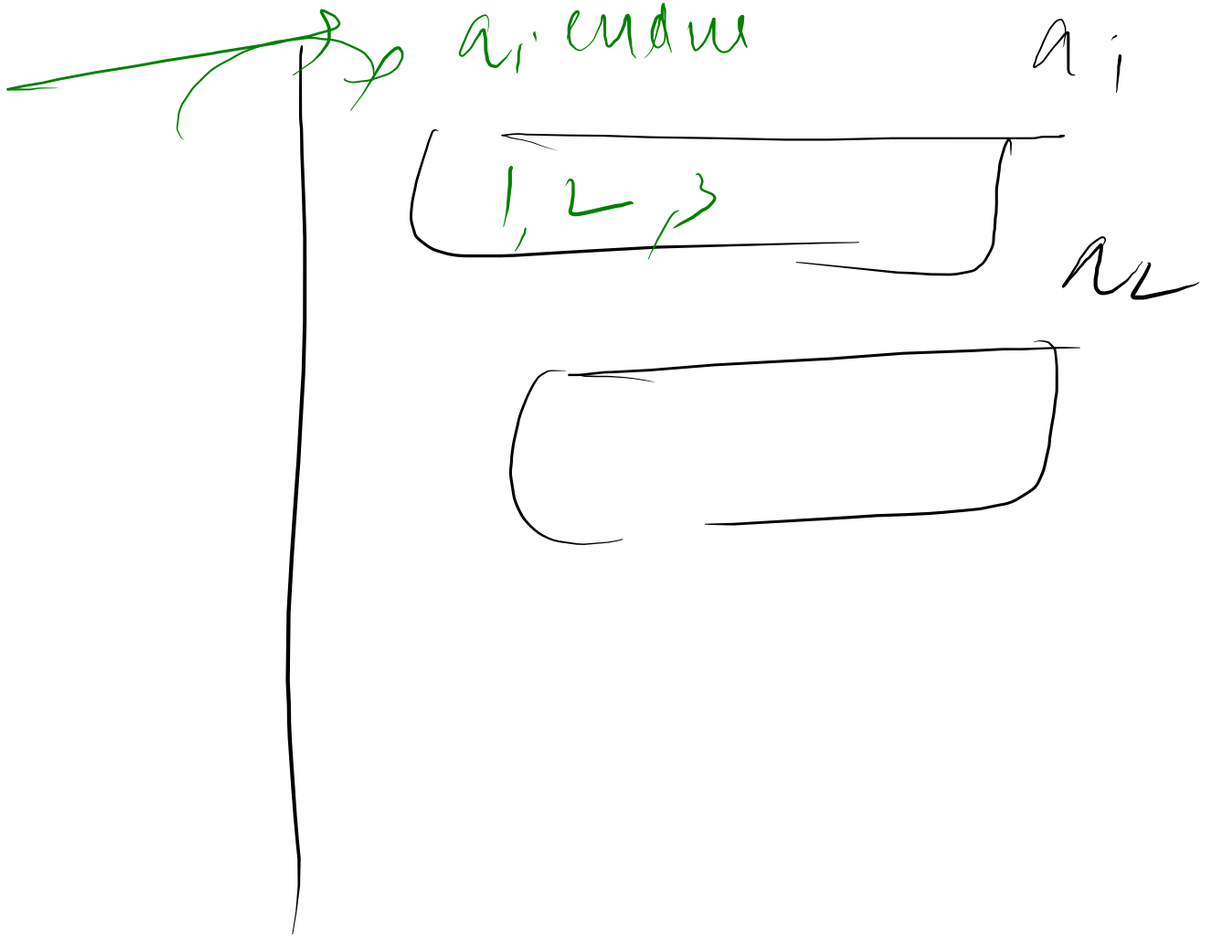

LL
head → iterate
Array
index
Stack → push/pop
Que → enqu/ dequ

S

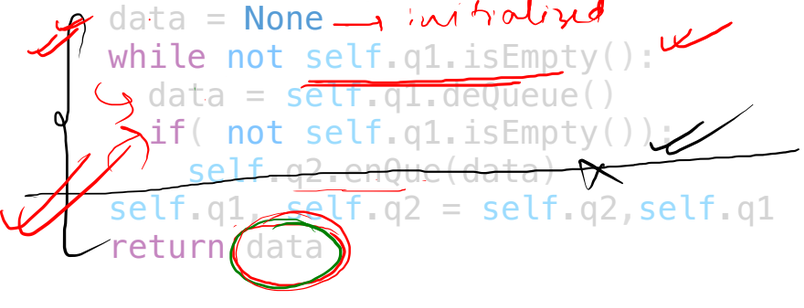
s.push(1)

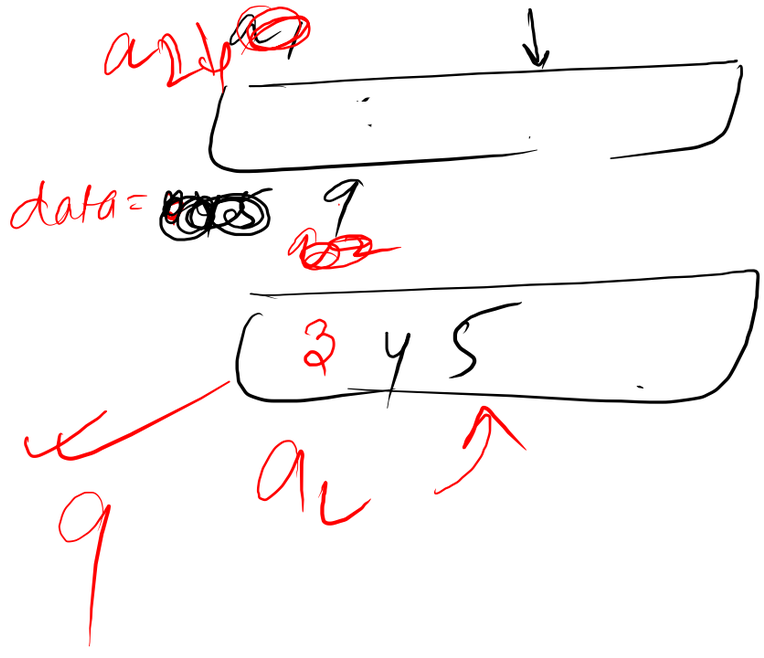
s.push(2)

s.push(3)



```python
#Implement Stack using Queue
class stackQ :
  def __init__(self):
    self.q1 = Queue() # it will act as the main queue
    self.q2 = Queue()


  def isEmpty(self):
    return self.q1.isEmpty()


  def push(self , data):
    self.q1.enQue(data)


  def pop(self):
    if self.isEmpty():
      print("Stack underflow error")
    # We need to pick the last element added
    data = None
    while not self.q1.isEmpty():
      data = self.q1.deQueue()
      if( not self.q1.isEmpty()):
        self.q2.enQue(data)
    self.q1, self.q2 = self.q2,self.q1
    return data


  def traverse(self):
    self.q1.traverse()
```

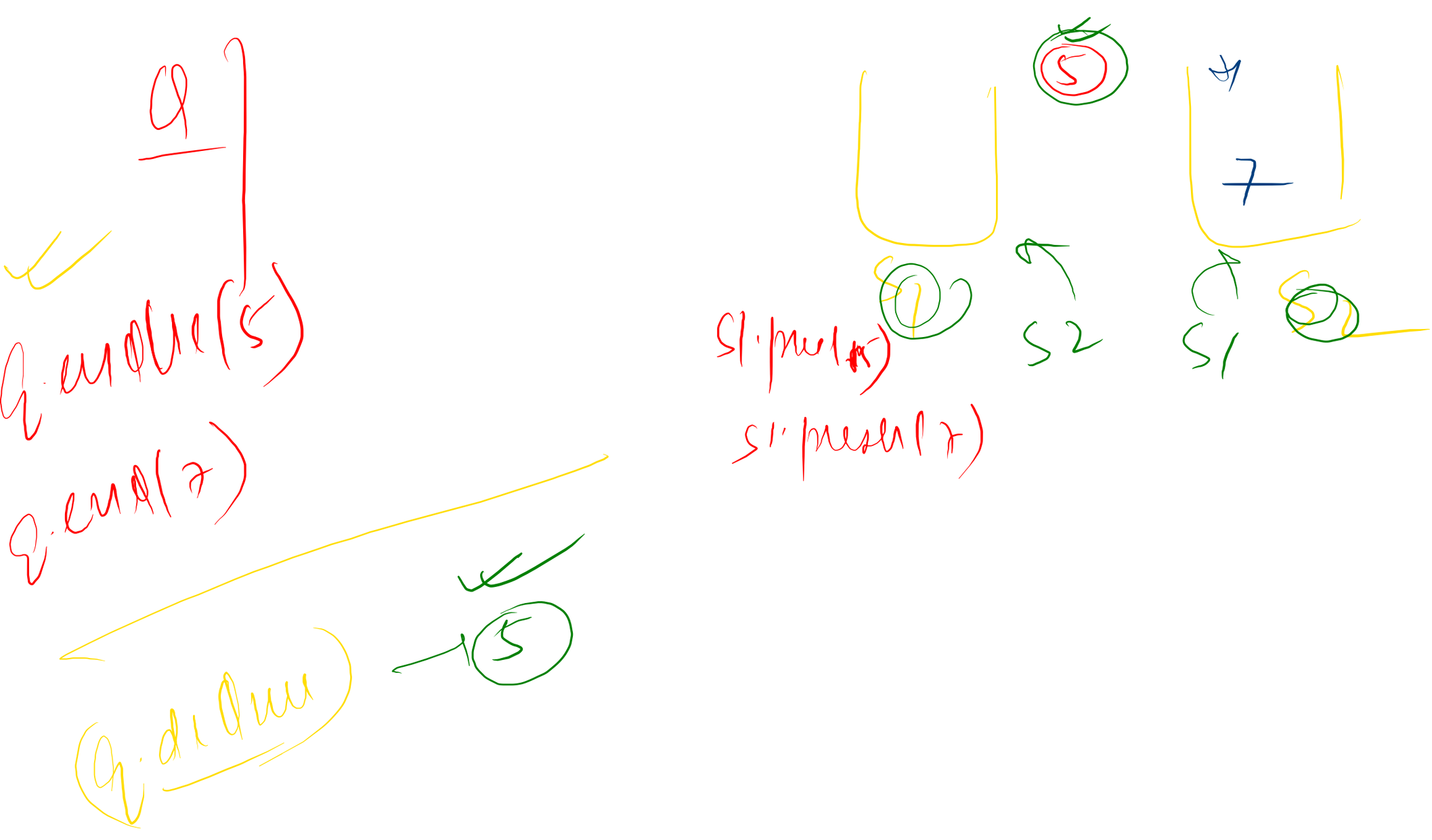




Implement Queue using Stacks

↓

enQueue | deQueue


Q
q.enqueue(5)
q.enqueue(7)
q.dequeue()
5
5
7
S1
S2
S1.push(x)
S1.pop()

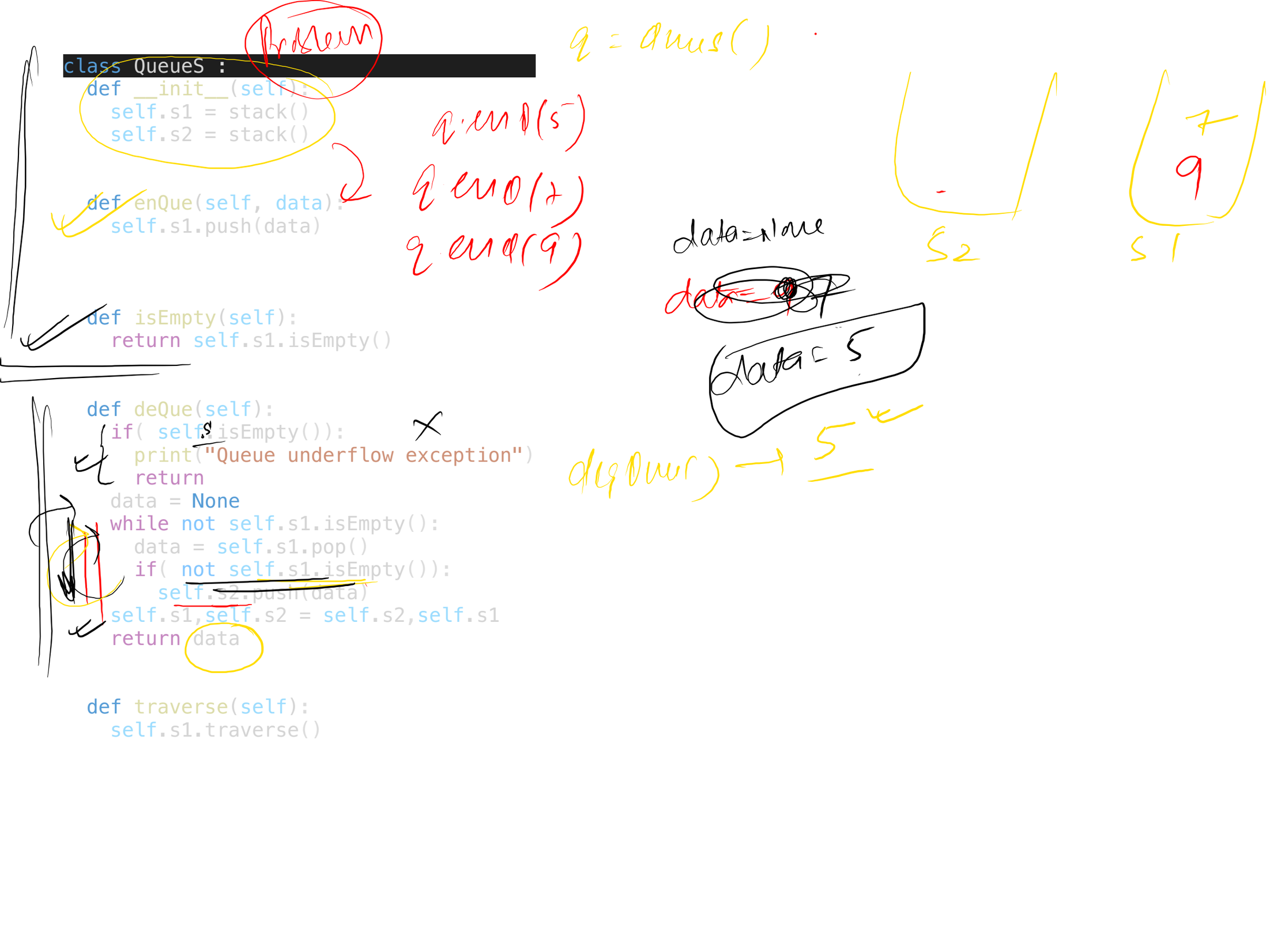
s.push(3)
s.push(5)
s.push(7)
s.push(9)

9
5
7
9

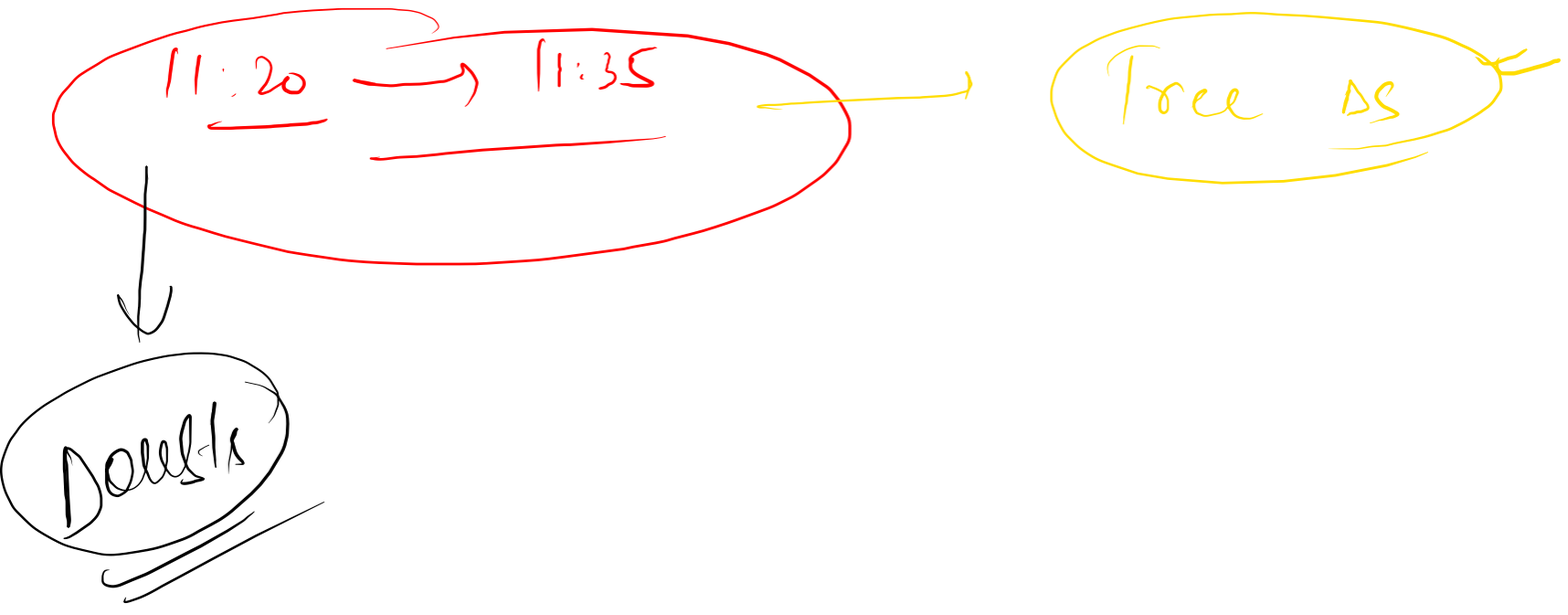
while (!s.isEmpty()) {
    s.pop()
}

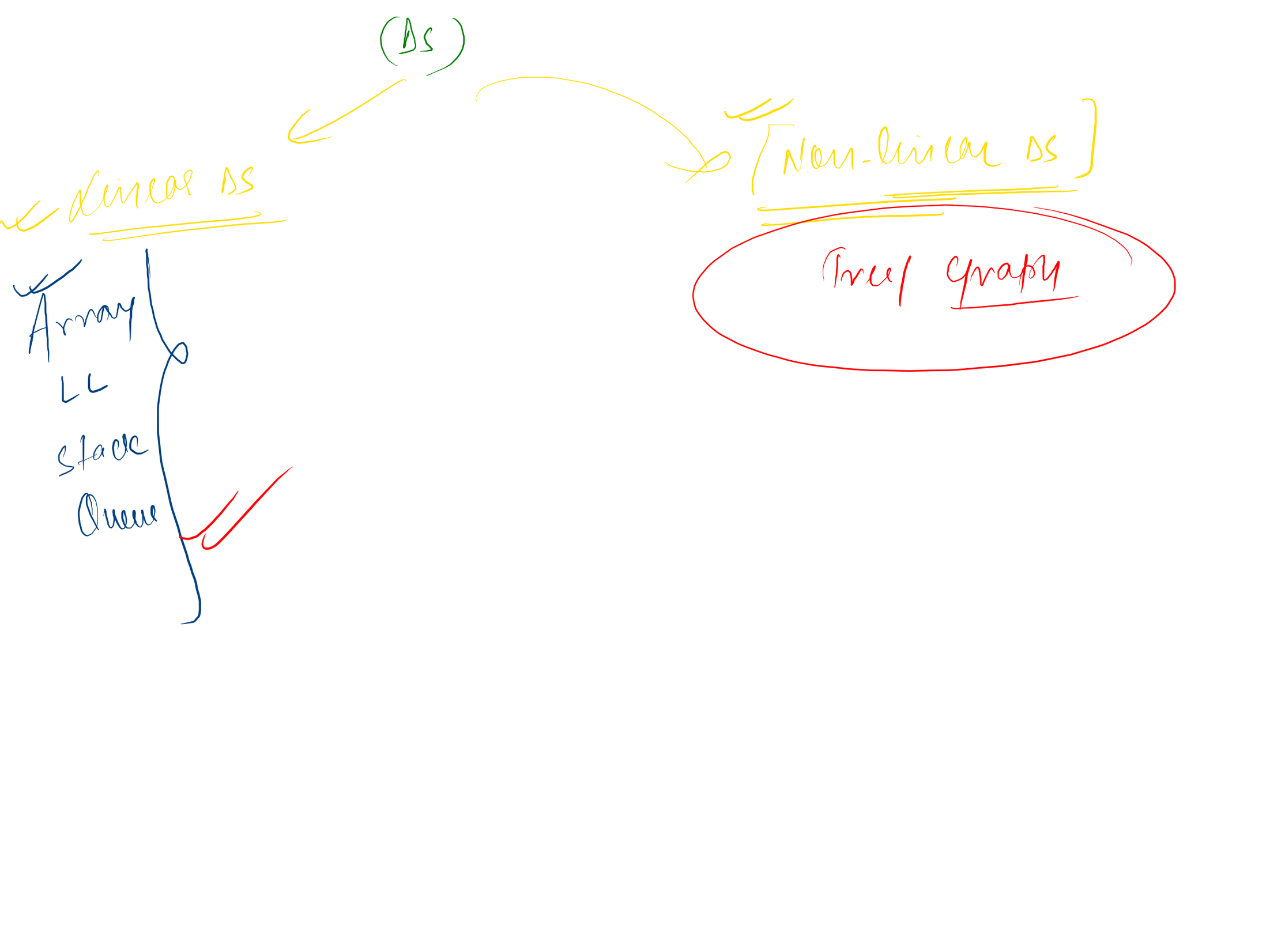
9 7 5 3

traverse → 3, 5, 7, 9

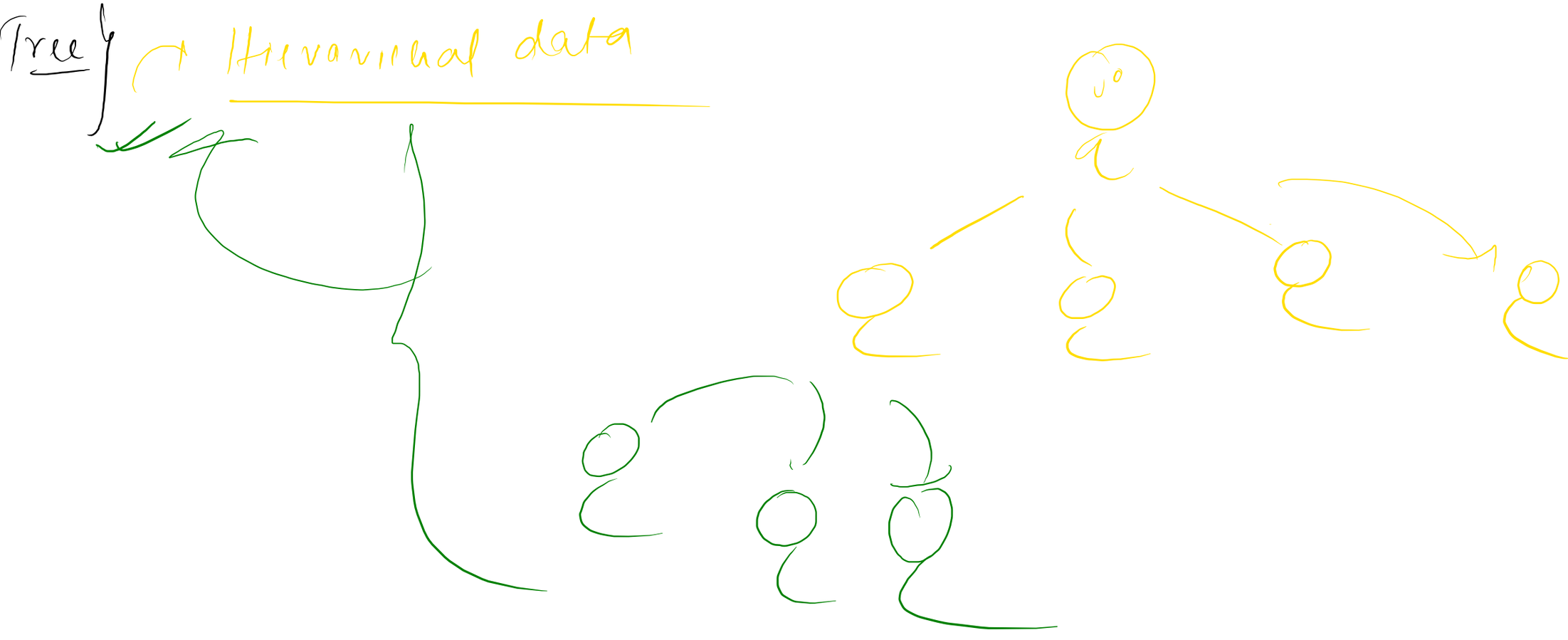


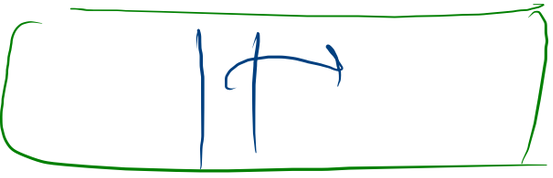
```
class QueueS :
  def __init__(self):
    self.s1 = stack()
    self.s2 = stack()

  def enQue(self, data):
    self.s1.push(data)

  def isEmpty(self):
    return self.s1.isEmpty()

  def deQue(self):
    if( self.isEmpty()):
      print("Queue underflow exception")
      return
    data = None
    while not self.s1.isEmpty():
      data = self.s1.pop()
      if( not self.s1.isEmpty()):
        self.s2.push(data)
    self.s1,self.s2 = self.s2,self.s1
    return data

  def traverse(self):
    self.s1.traverse()
```

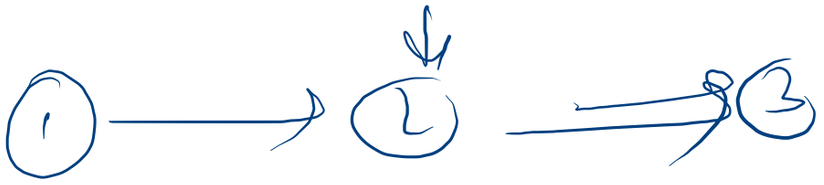

11:20
11:35
Tree DS
Doubts


(DS)
Linear DS
Array
LL
stack
Queue
Non-linear DS
Tree
Graph

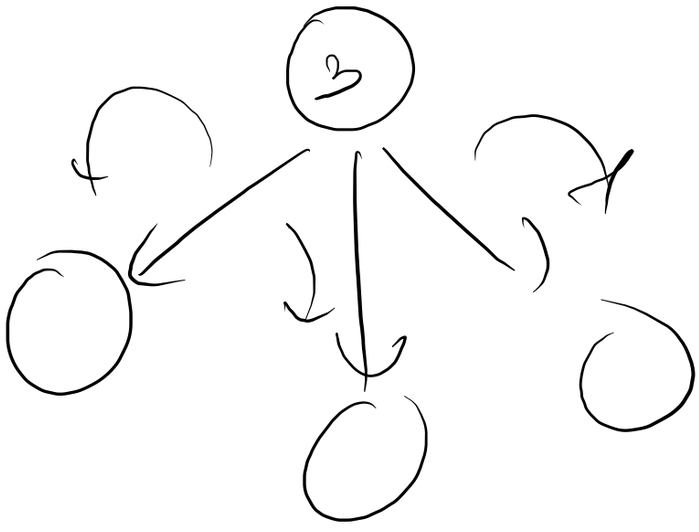
Tree → Hierarchical data



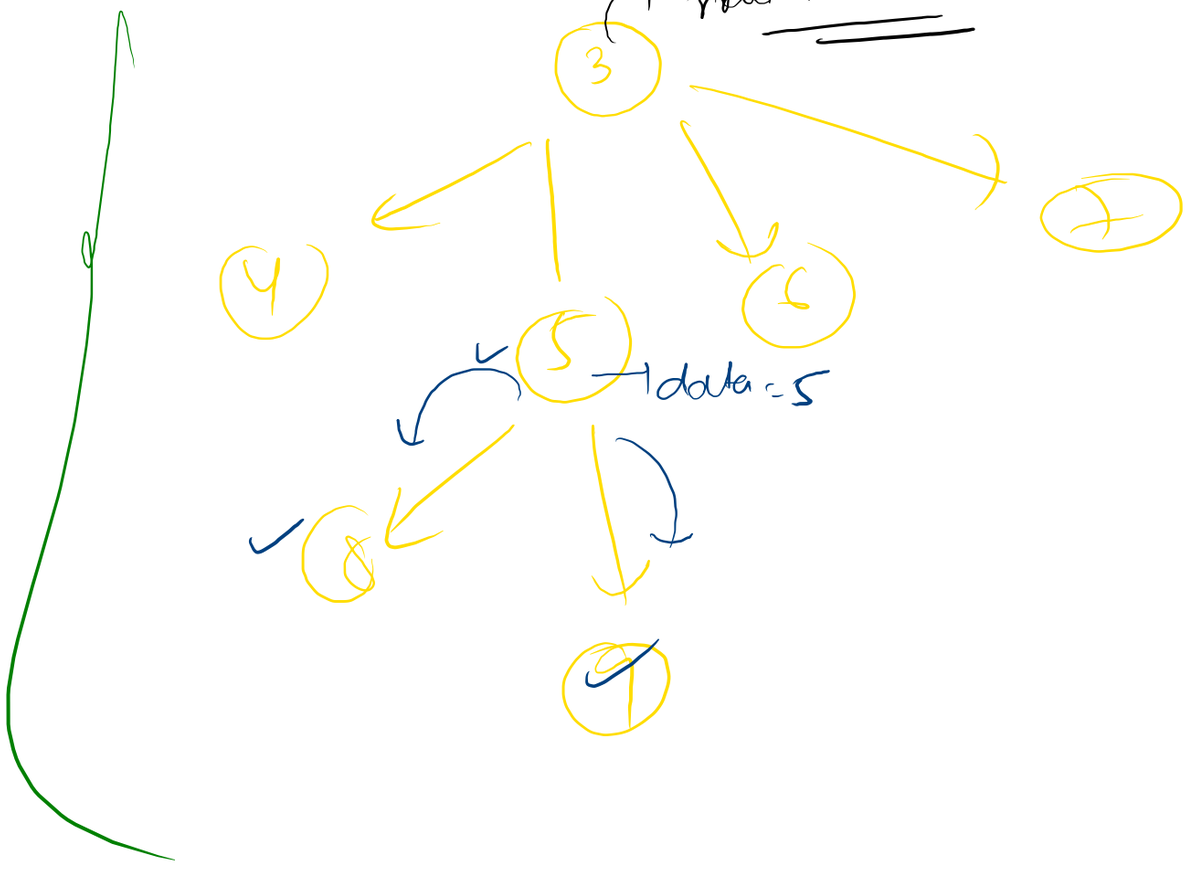



1
2
3

True



Tree



Node
- data
- Address of child Nodes

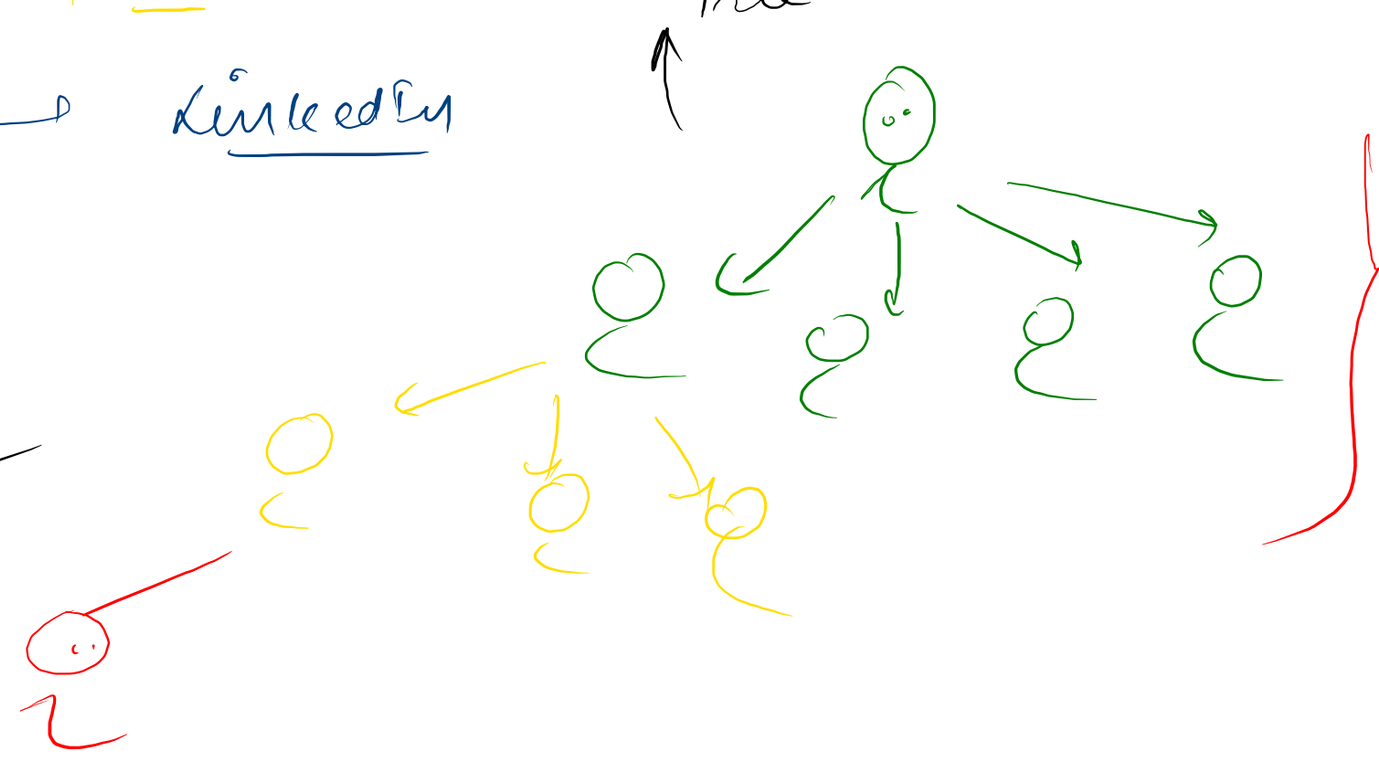
Use cases -

① Hierarchical data

Trees

LinkedIn

Tree



Post → comment tail

preparation etc

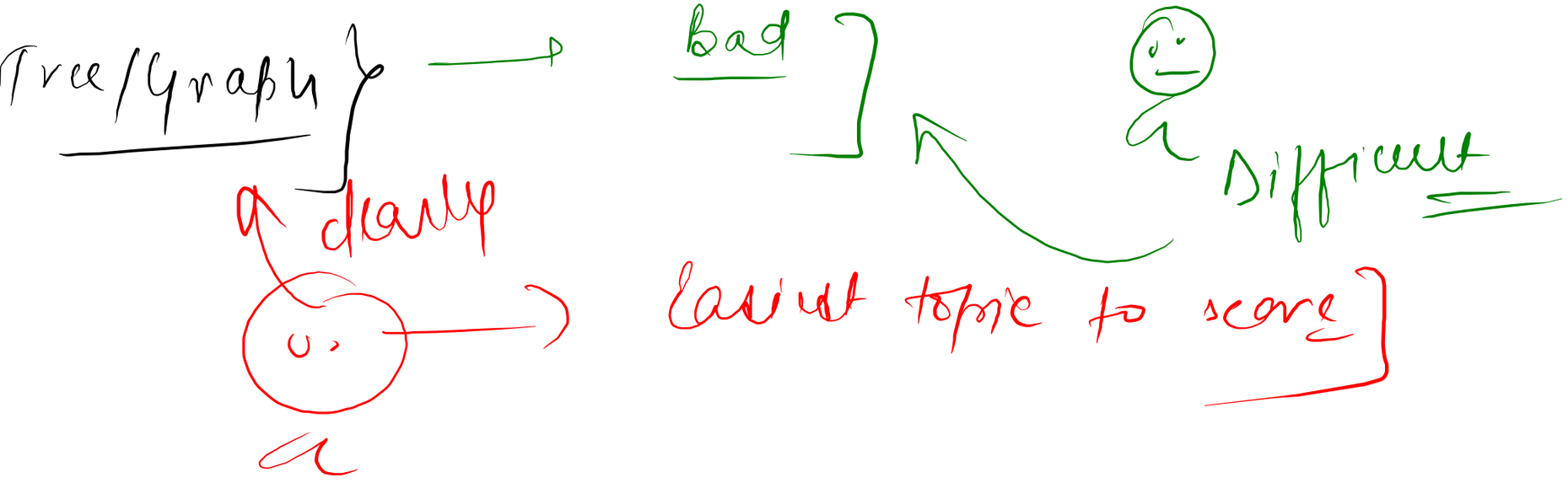
(2) Tree / Decisions → (ML) → Decision tree

(3) 

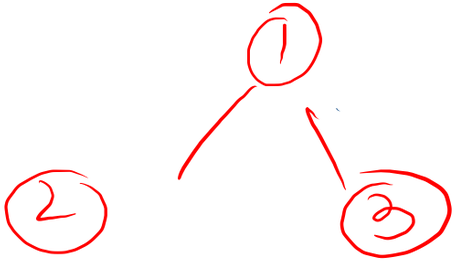
databases
↳ indexing → Read fast
↳ (As) → Tree

(4) Make search faster → BST

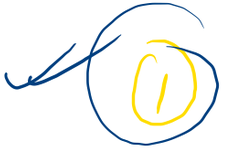
Hot favourite
↳ RB | G
Google
AVL
Trees/Graphs


Tree/Graph
bad
Difficult
dsalgo
Easiest topic to score

Binary Tree → Any node → 0/1/2 child Nodes

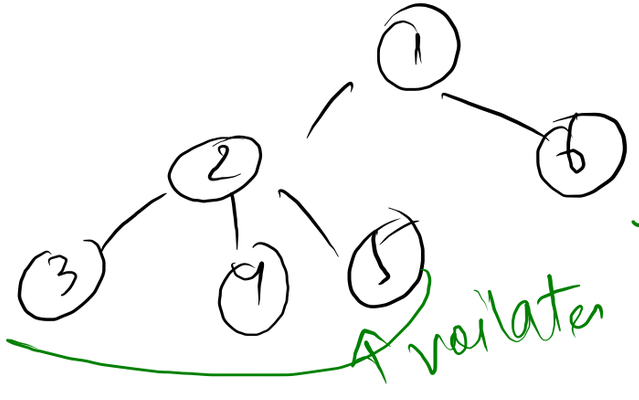


→ BT



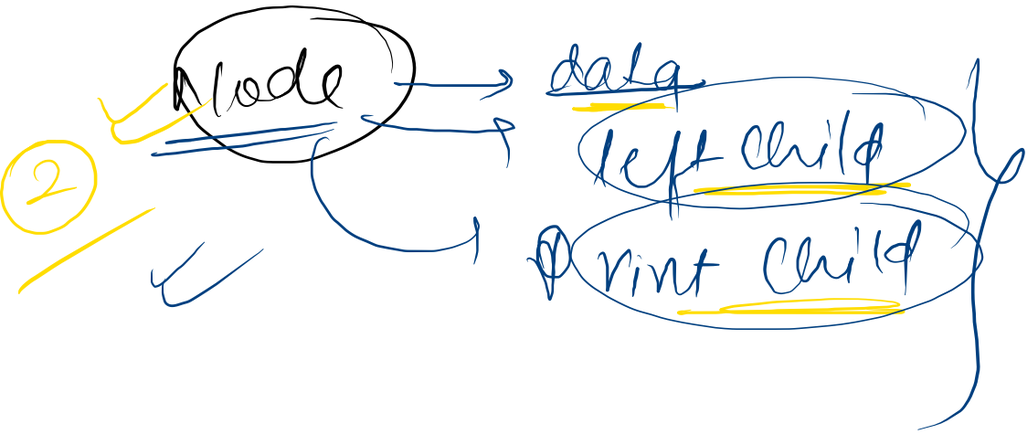
→ BT

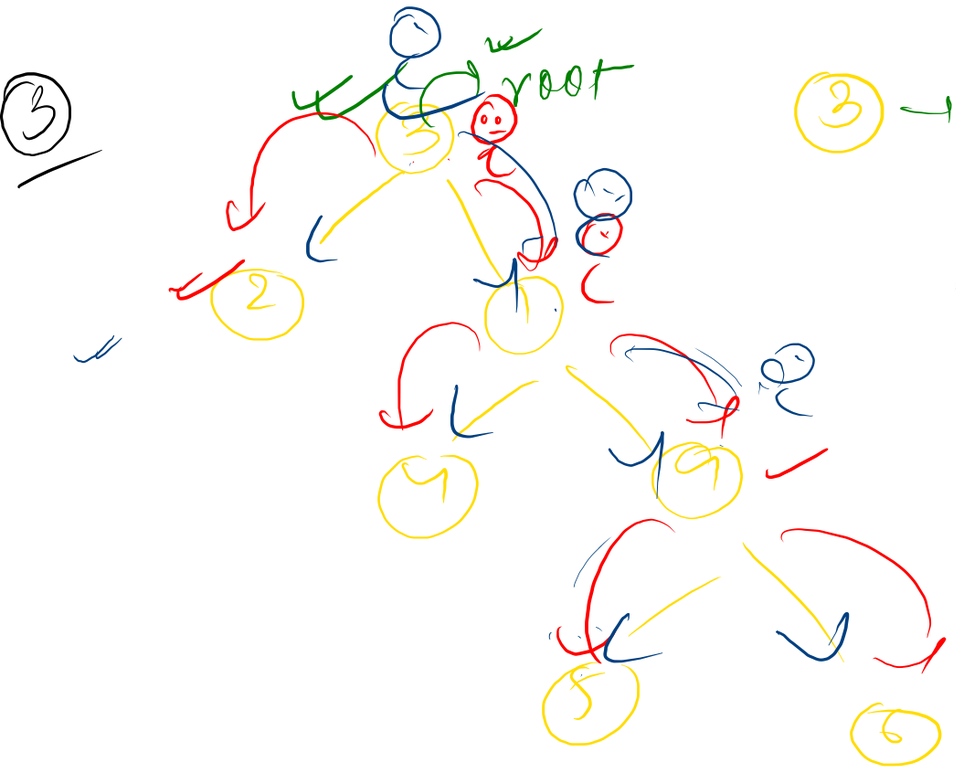
→ BT



violates BT

X Not a Binary T


Node
data
left child
right child
2



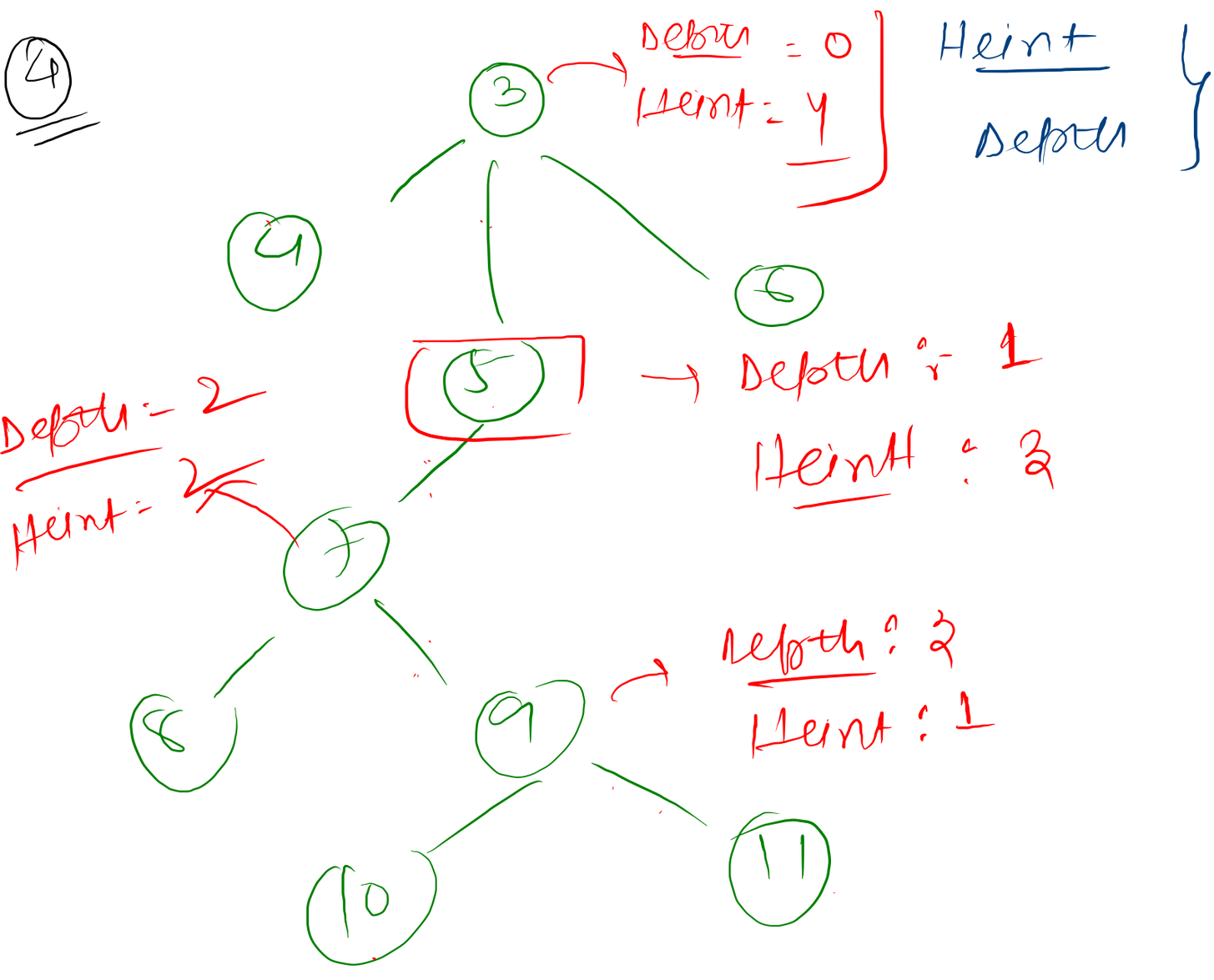
3 → I can go to any Node

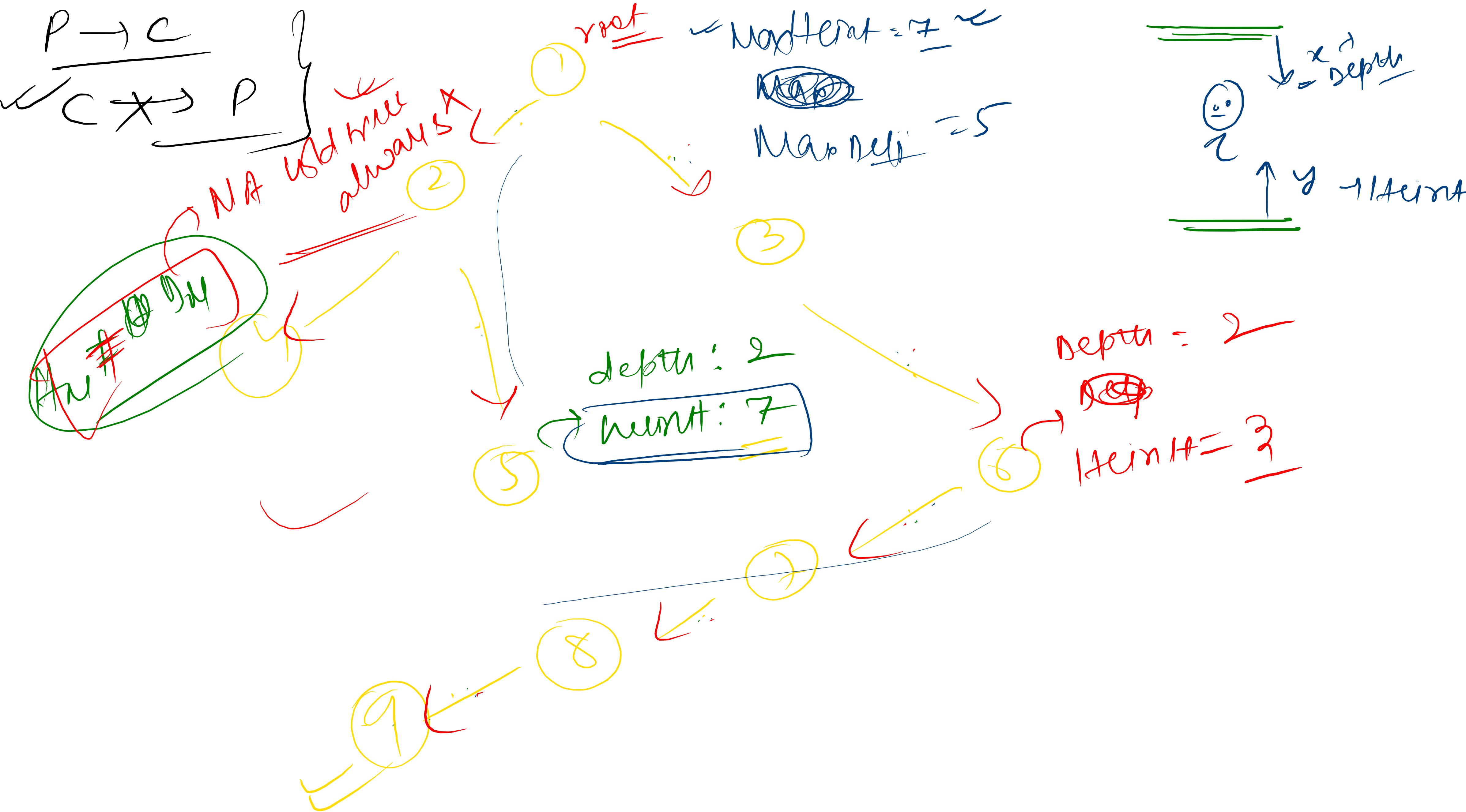
Trees are always represented by root

LL → Head Node

P → child

(4)


3
Depth = 0
Height = 4
Height
Depth
4
6
5
Depth = 1
Height = 3
Depth = 2
Height = 2
7
8
9
Depth = 2
Height = 1
10
11


P → C
C ↛ P
root
MaxHeight = 7
MaxDepth = 5
depth : 2
height : 7
Depth = 2
Height = 3
NA with always
depth
Height
1
2
3
5
6
7
8
9

Height of the tree / Depth of the tree

Maximum depth of any Node

Max height of any Node

Balanced tree

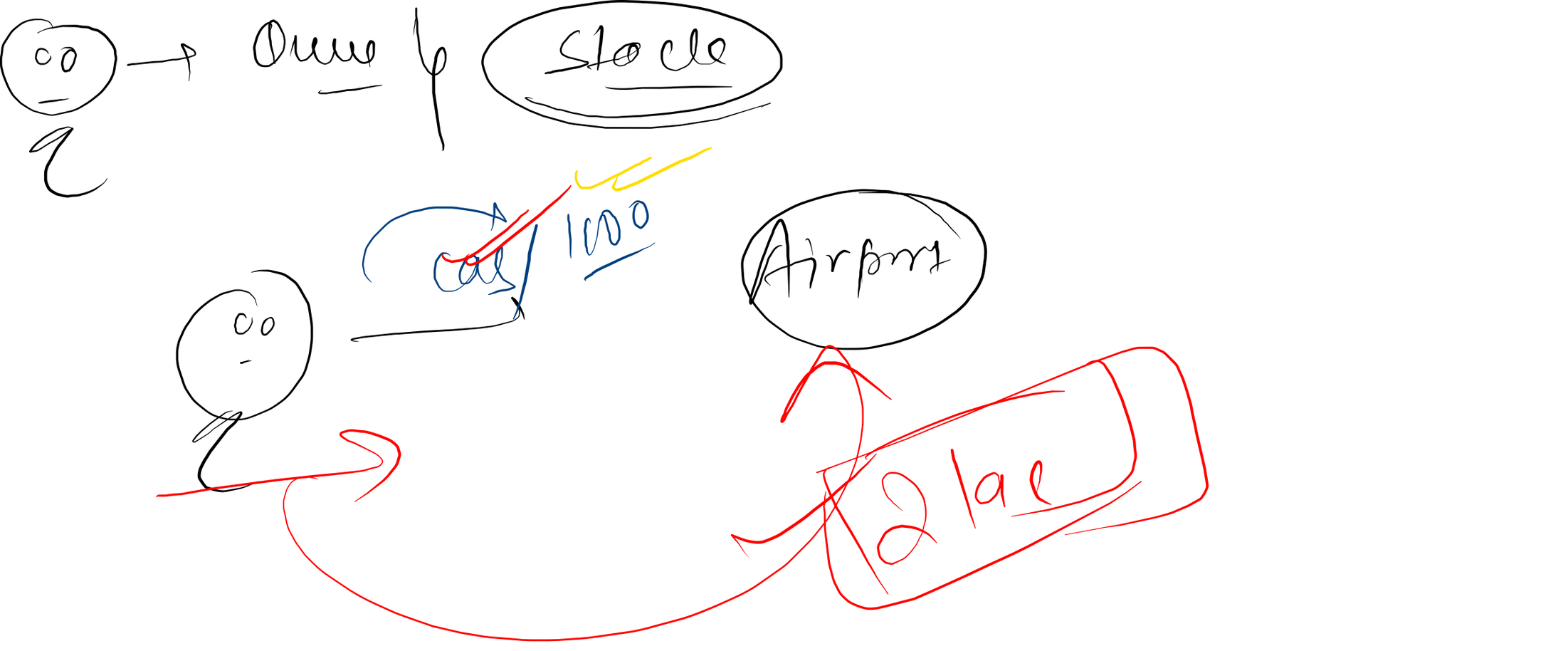
1

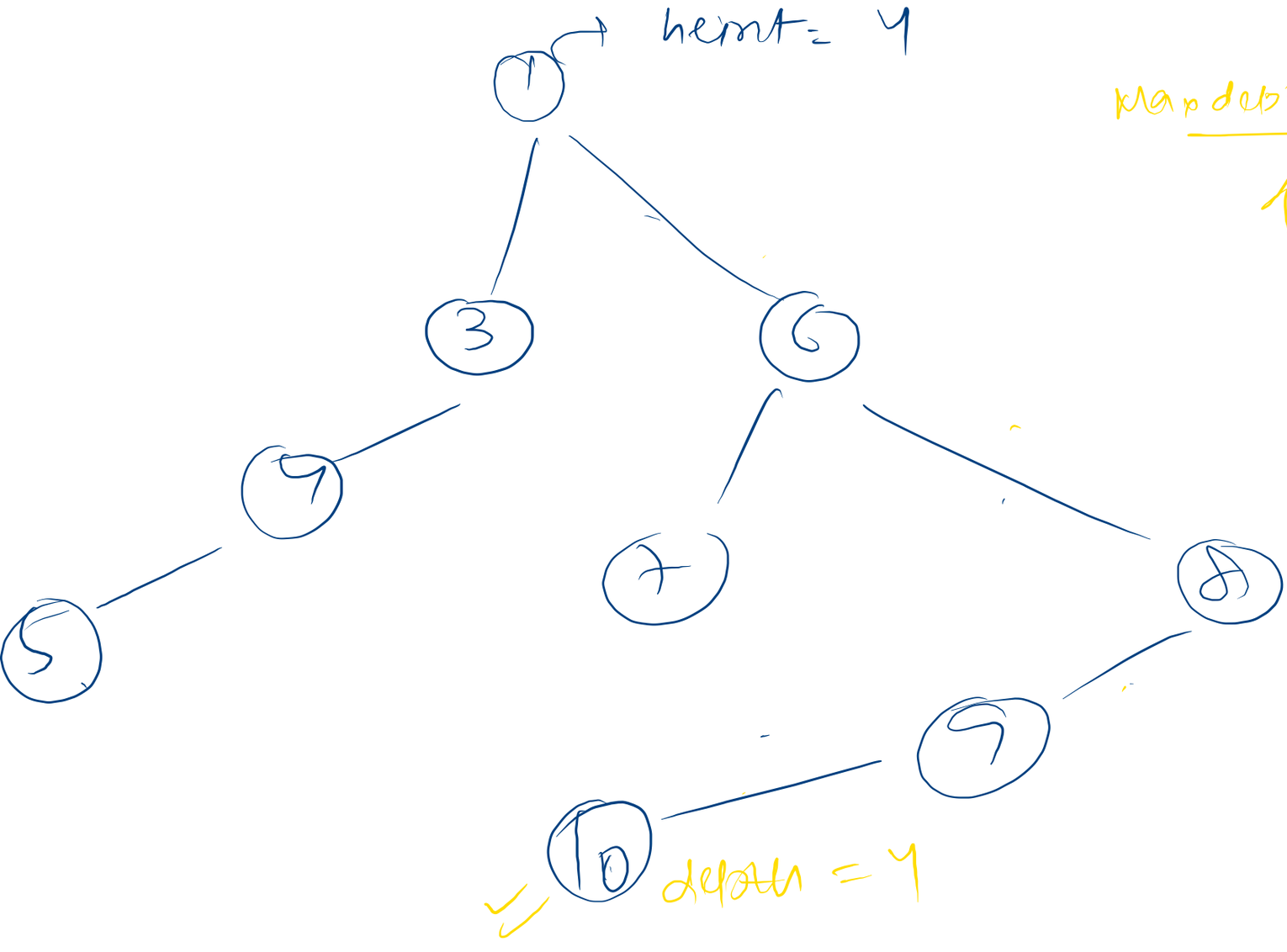
Height of tree = depth of the tree

distance b/w root and deepest node

Depth : distance b/w node and root

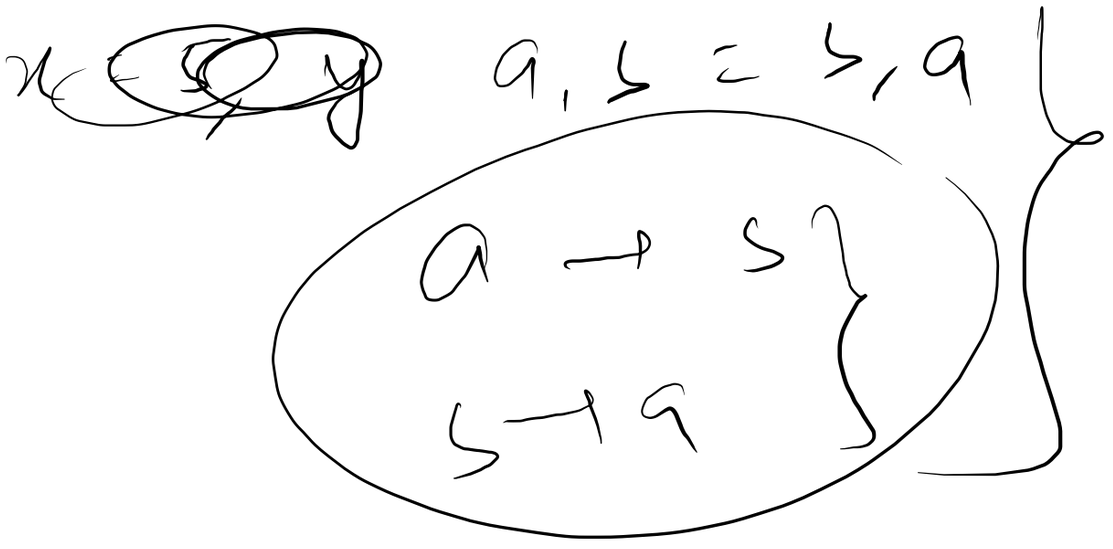
Height = distance b/w node and deepest node

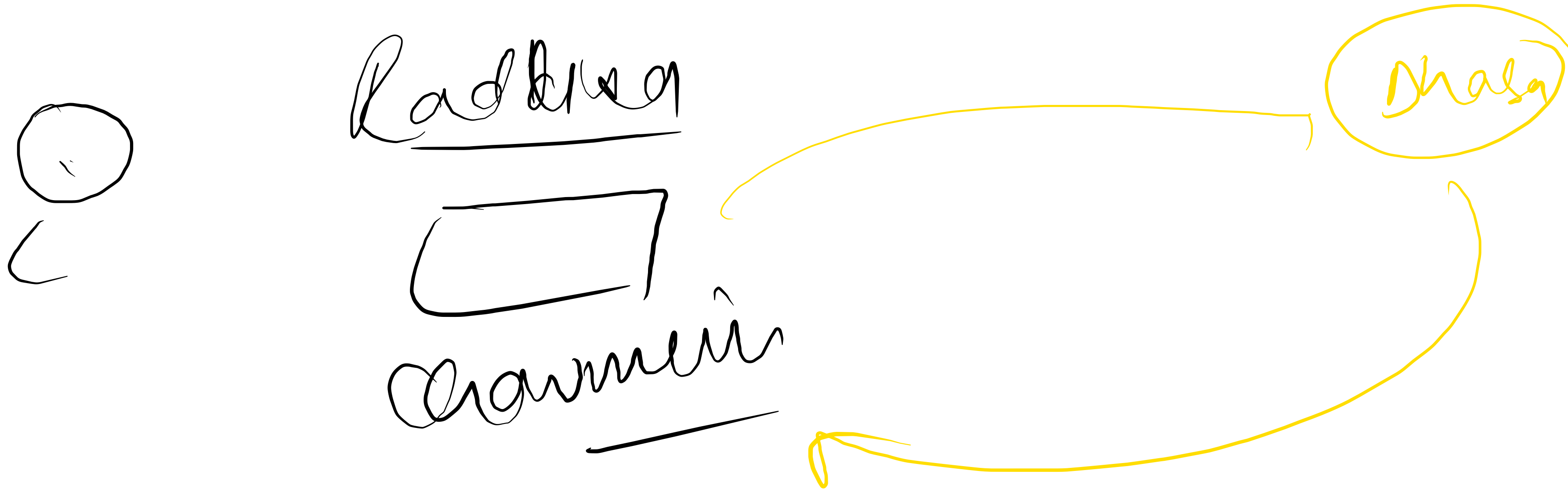

Stacle
call 100
Airport
Place



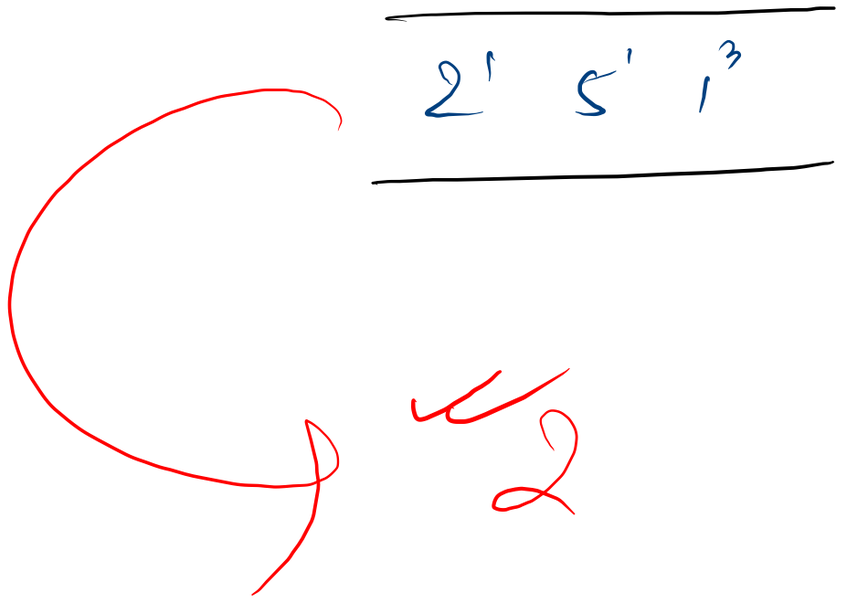
Map depth = Map height

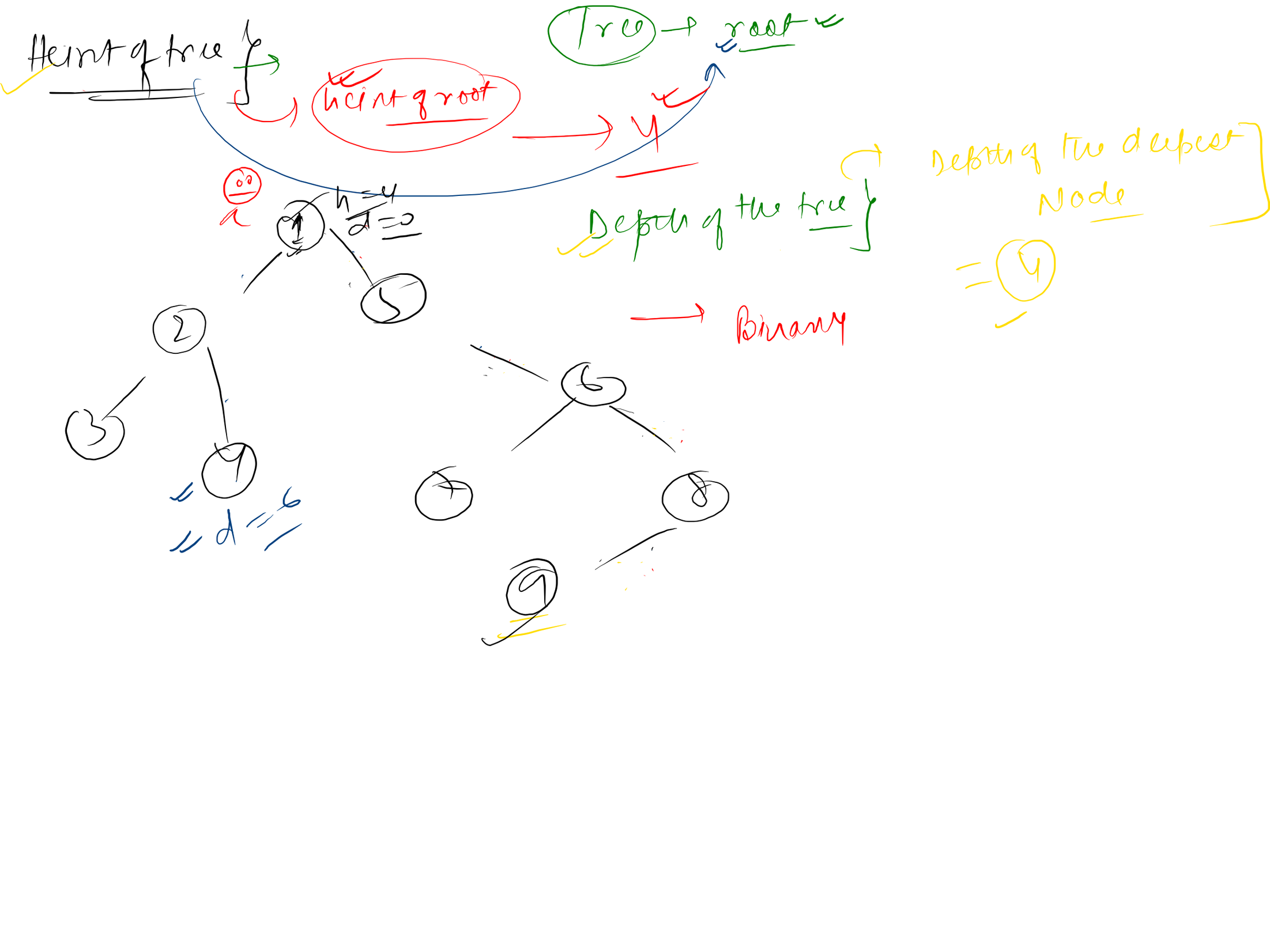
↑ Height / Depth of the tree





Q → FIFO

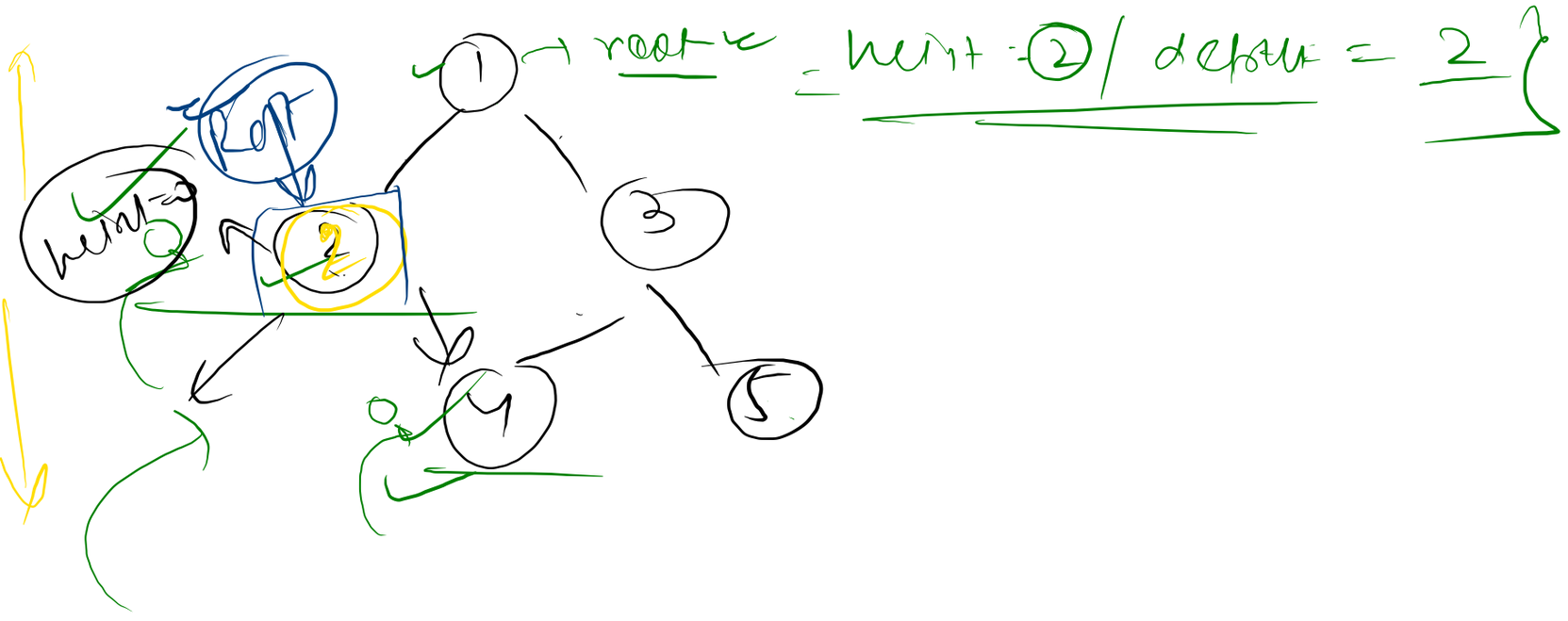



Height of tree
height of root
Tree → root
y
Depth of the tree
Depth of the deepest Node
= 4
Binary
h = 4
d = 0
d = 6

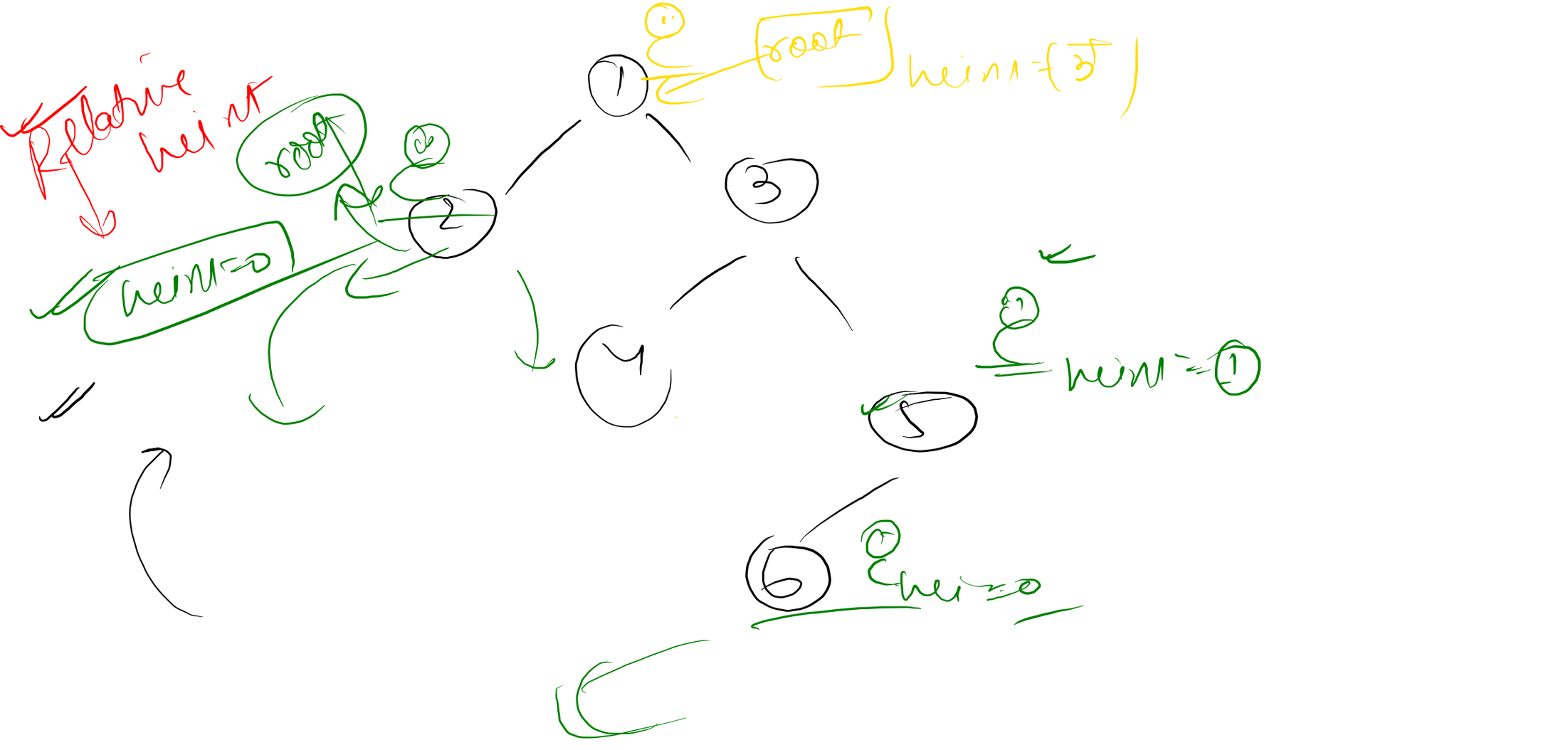
Generic BT

Node → data

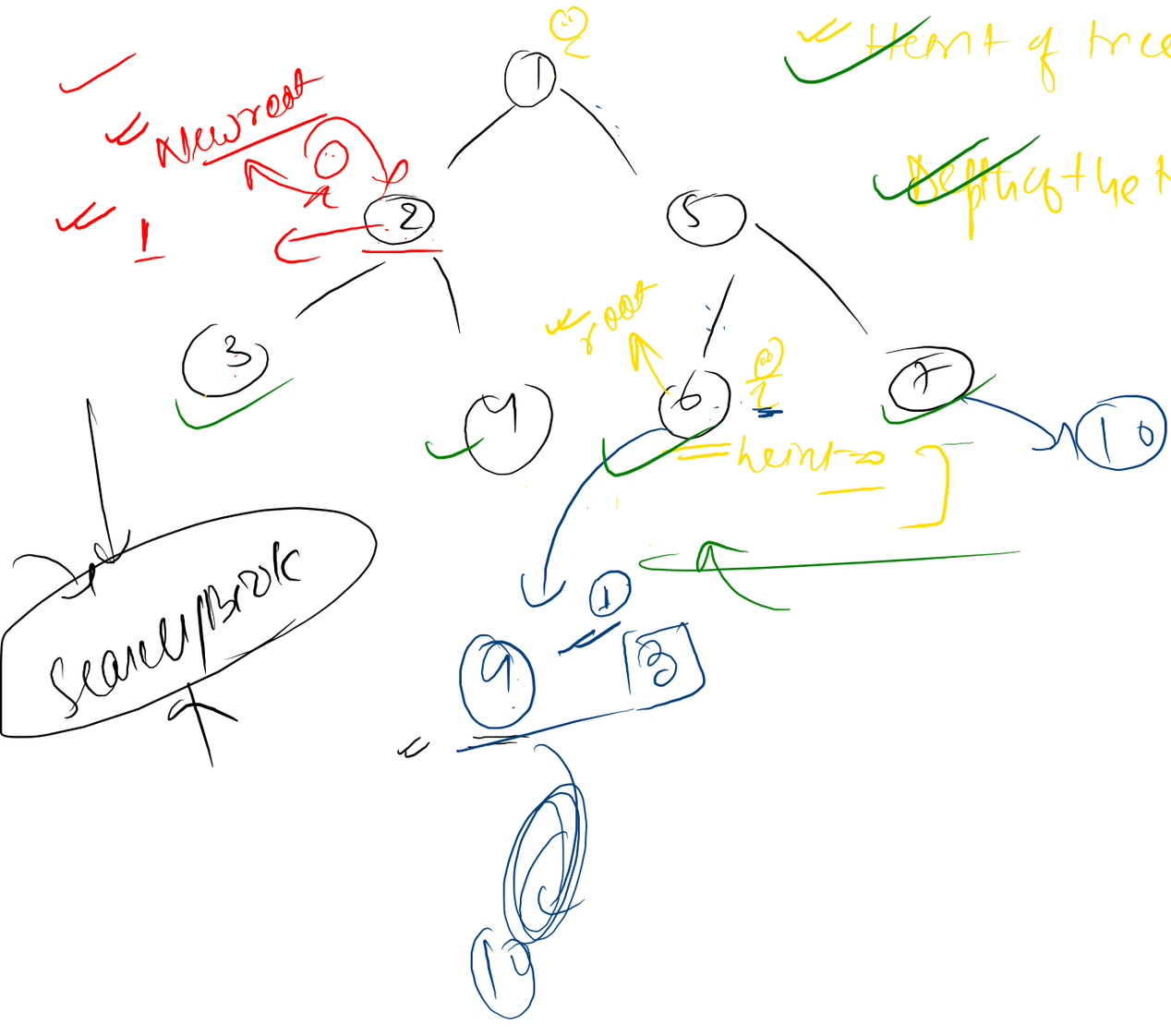
0/1/2 child Node



height - ② / depth = 2


Relative height
root
height = 0
root
height = 1
height = 0

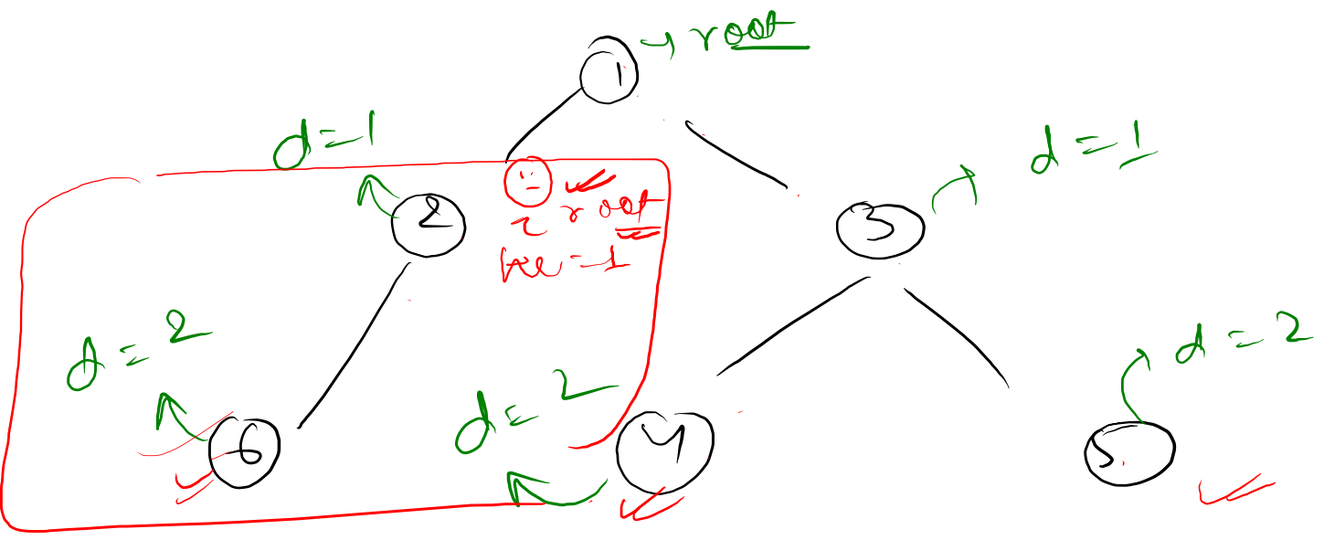
Optic spectra



Height of tree = Height of root = 2

Depth of the tree = Depth of the deepest Node from root

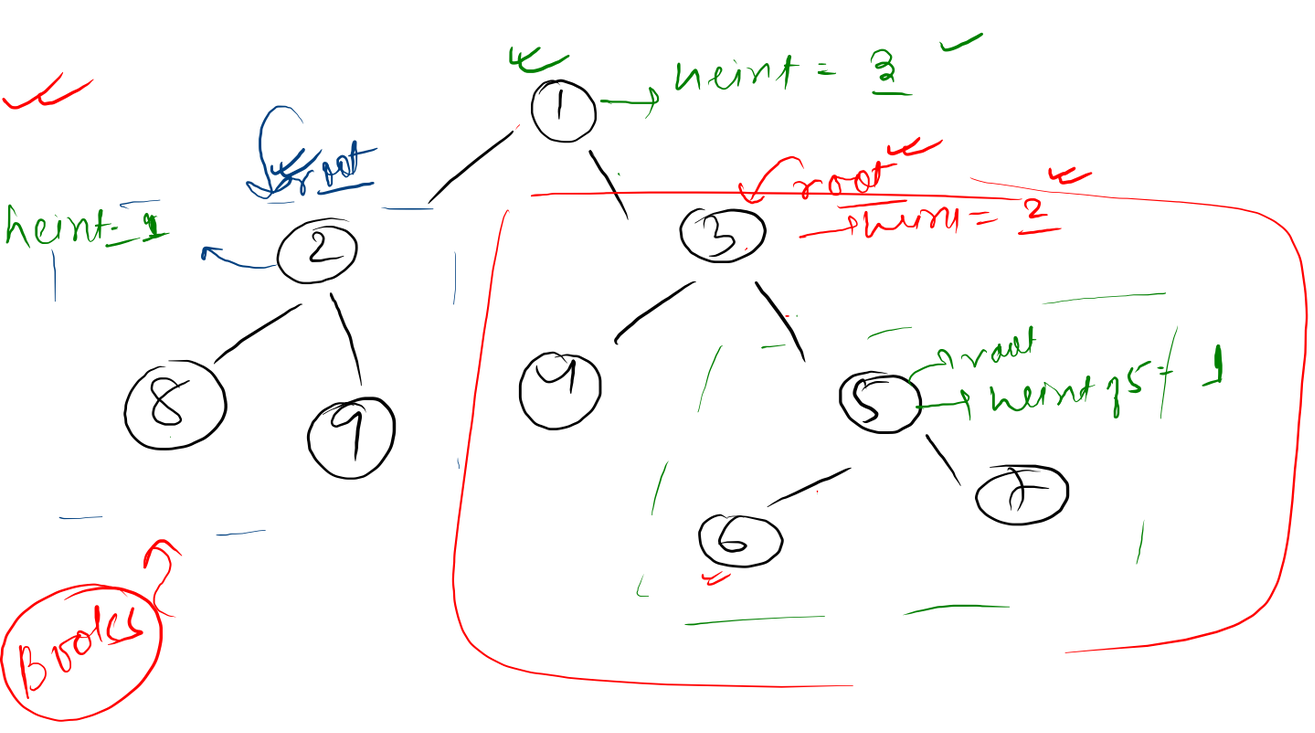
= 2



= Depth → Map distance from root of tree

Deepest Nodes :- (6) / (4) / (5)

Depth of tree = 2

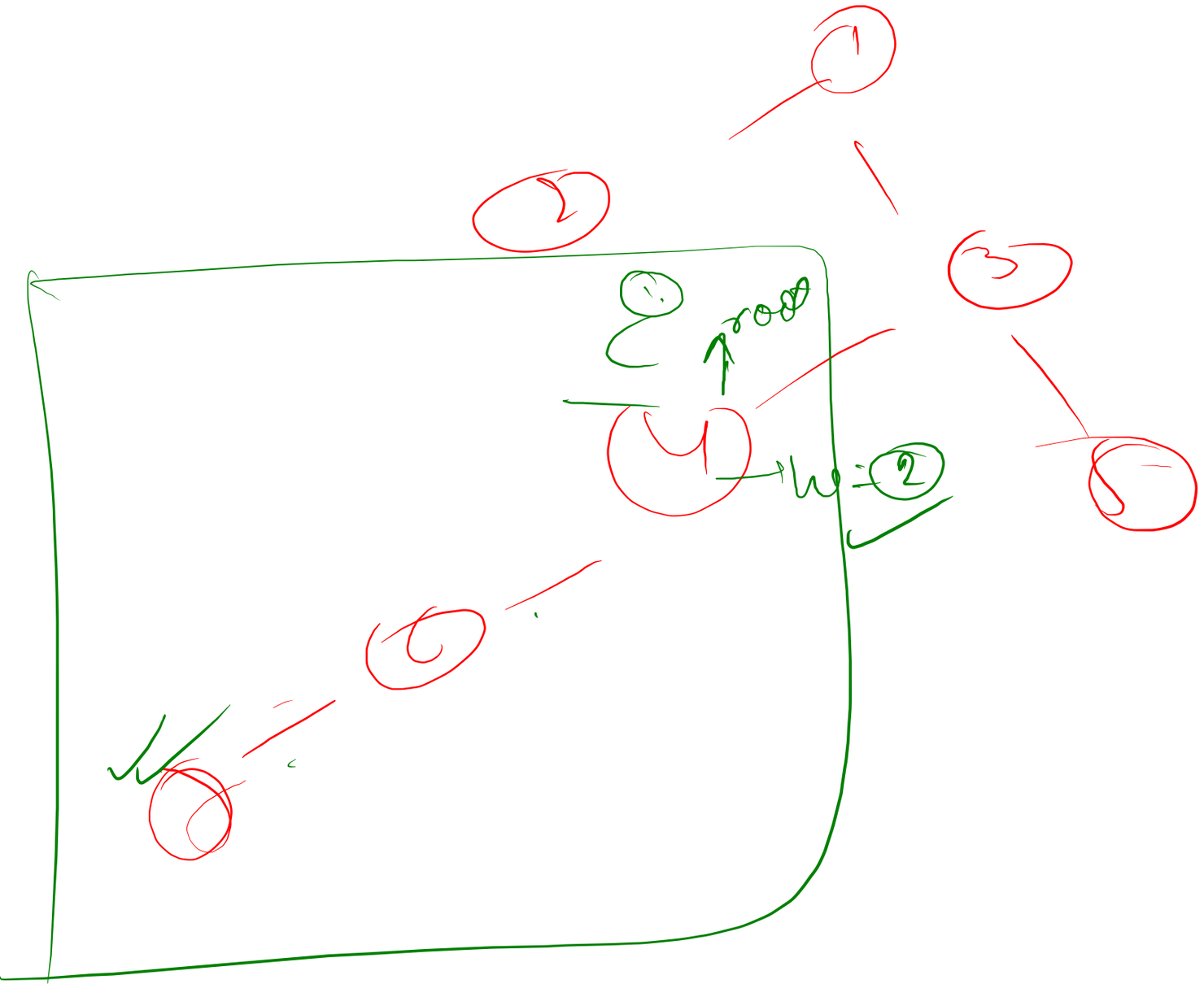


Height

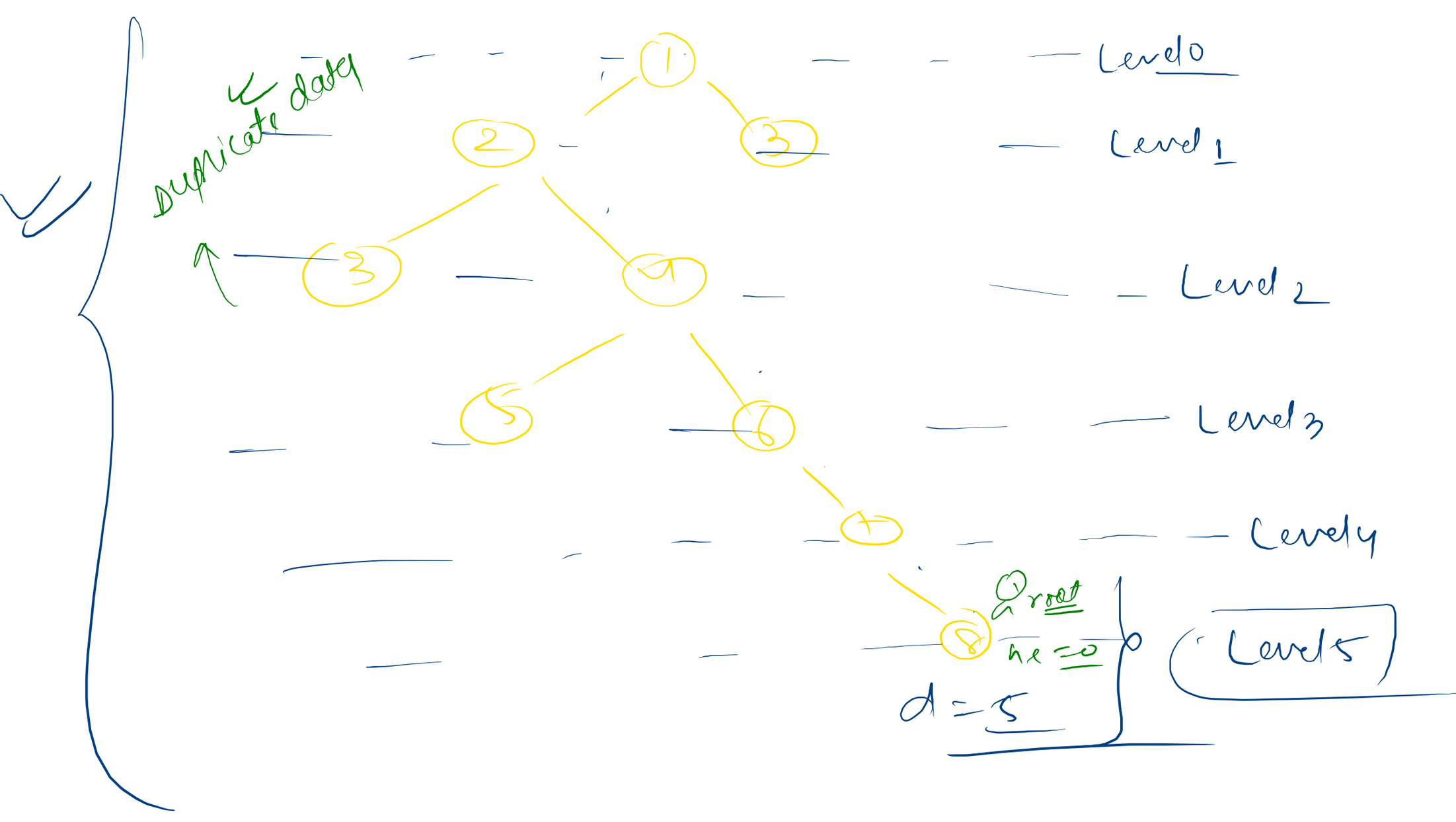
Always the distance of root with deepest Node

Books / Google

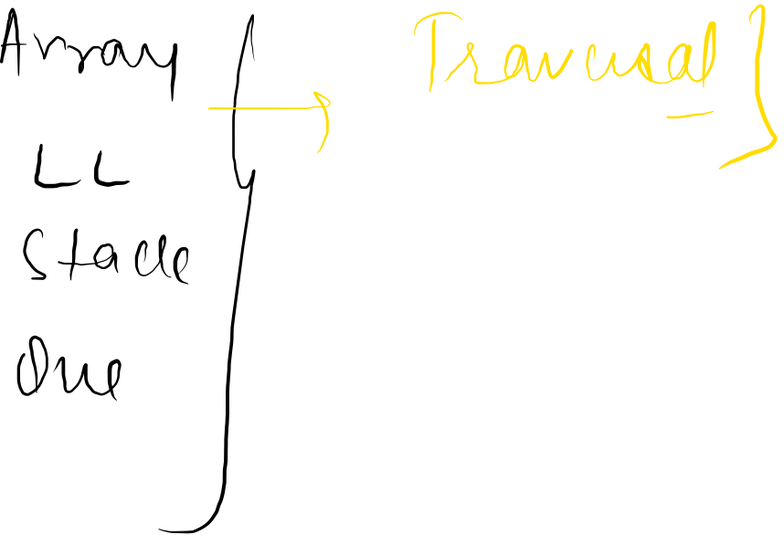
Height of tree = Height of root = 3

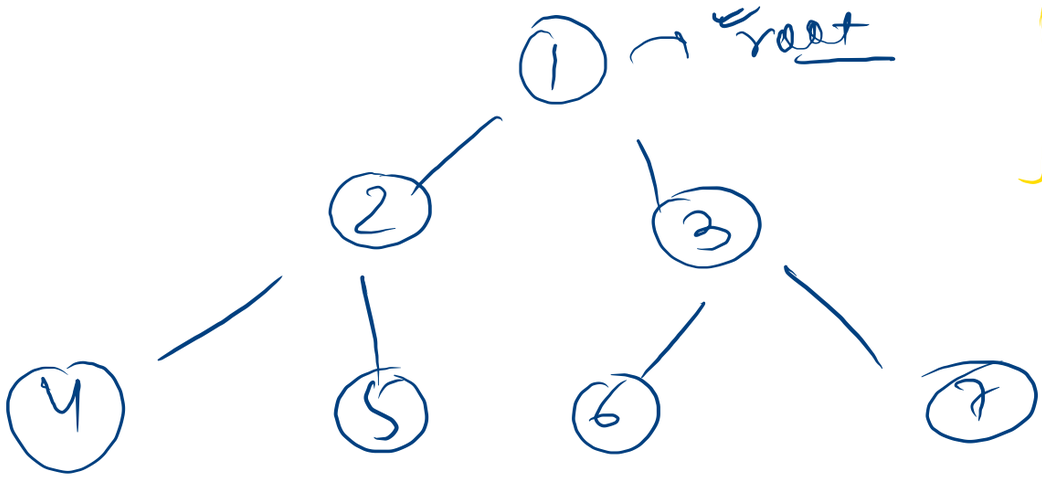

root
w = 2

Height / Depth } → Level


Duplicate data
1
2
3
3
4
5
6
7
8
root
ht = 0
d = 5
Level 0
Level 1
Level 2
Level 3
Level 4
Level 5

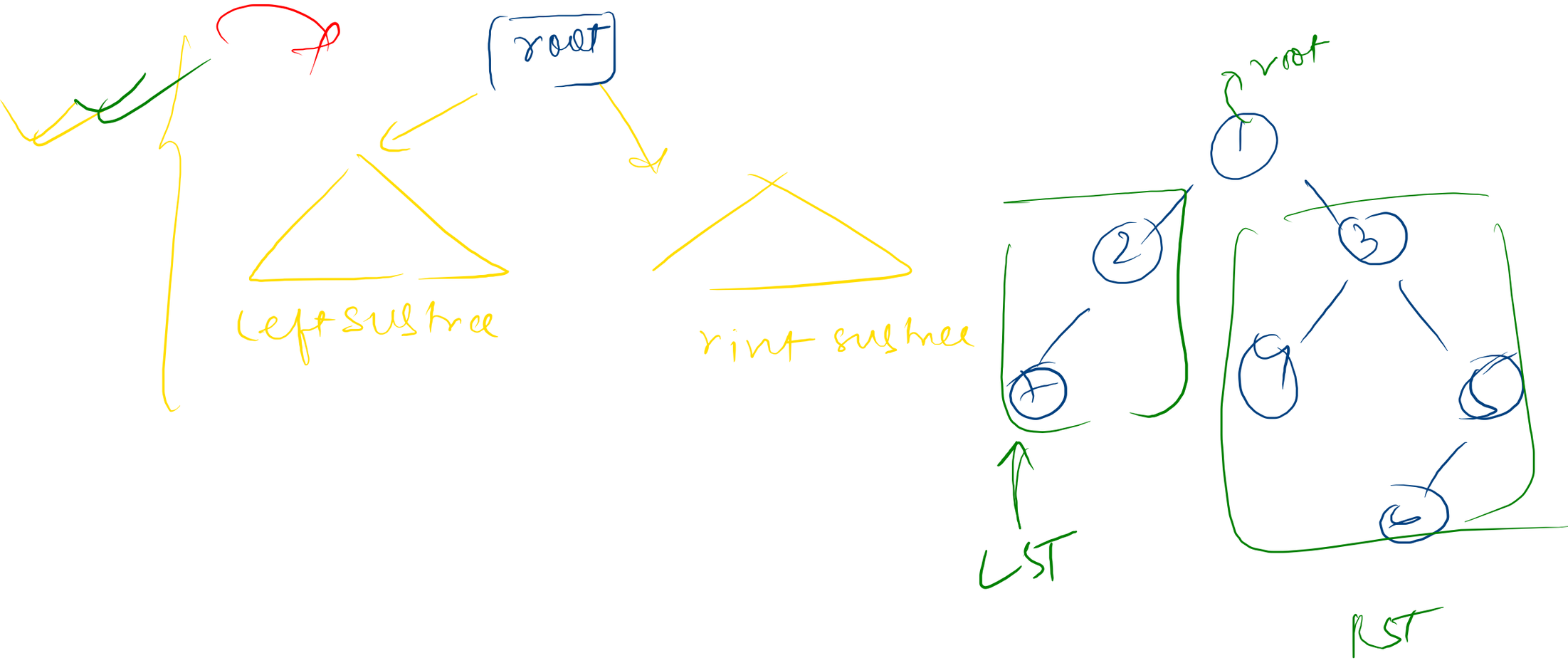
No of leads

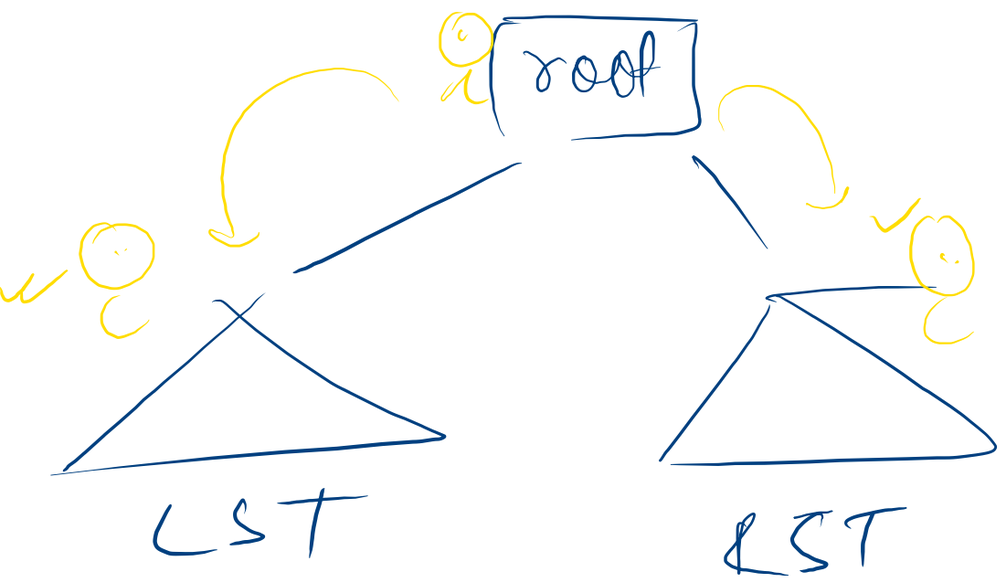

Array
LL
Stack
Que
Traversal



traversing

printing all the elements of the tree

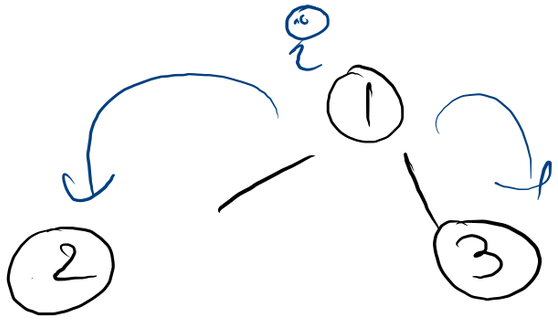

root
left subtree
right subtree
root
1
2
3
4
5
6
LST
RST



Traversing :-

root → LST → RST

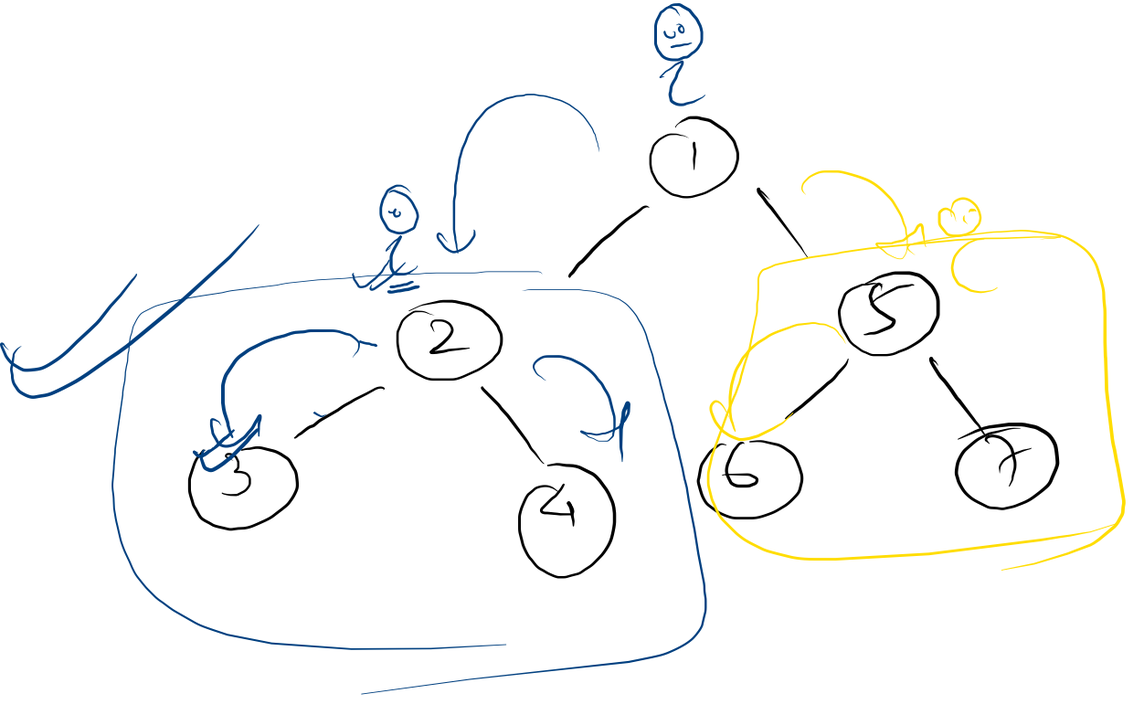
↑ Preorder traversal



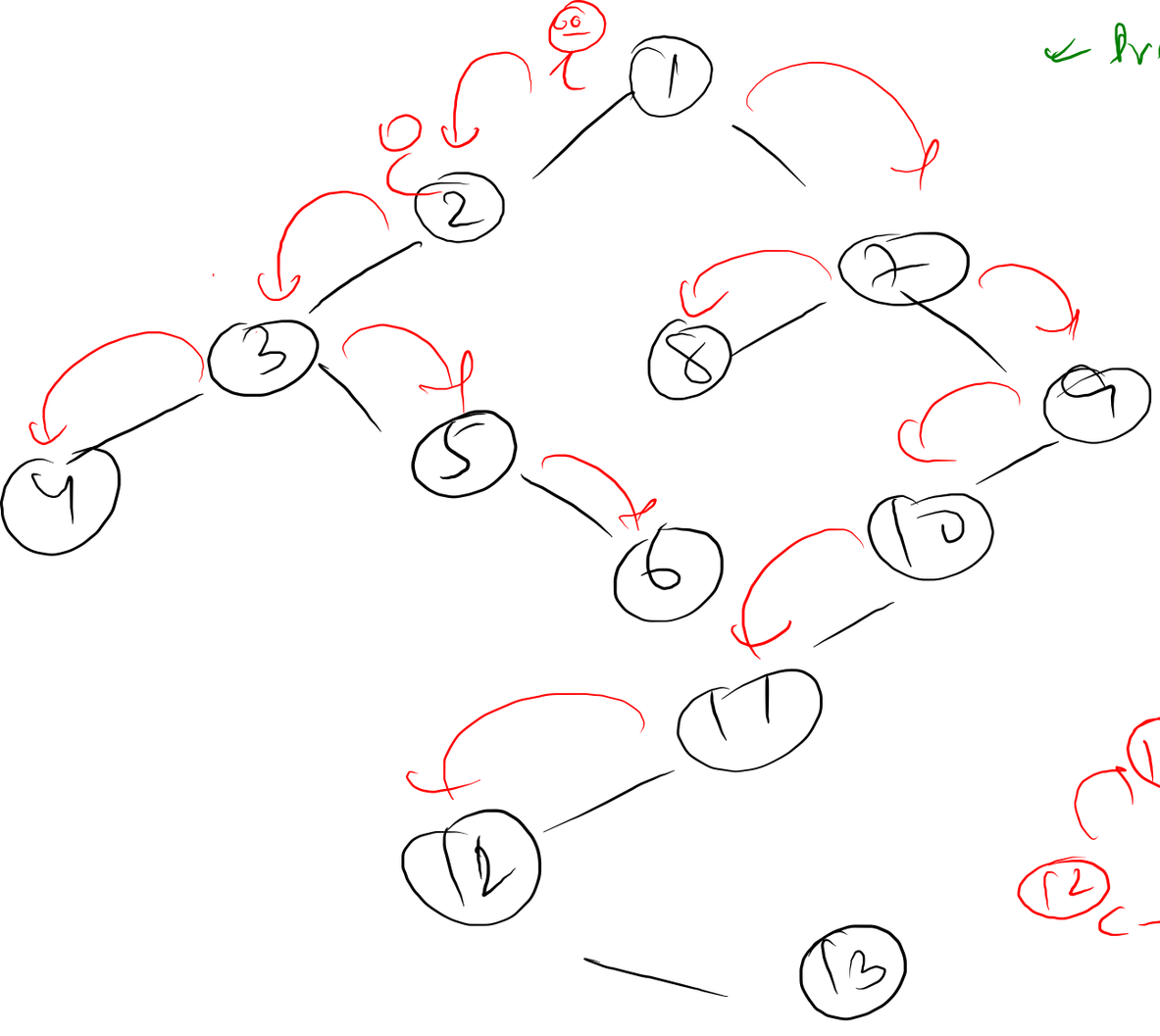
Pre-order traversal

Root → LST → RST

1 → 2 → 3

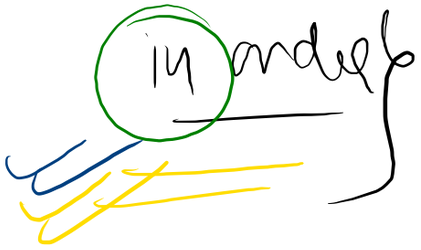


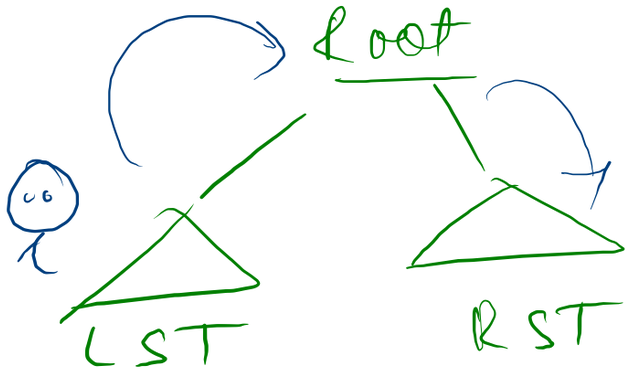
1 → 2 → 3 → 4 → 5 → 6 → 7



← Pre-order traversal

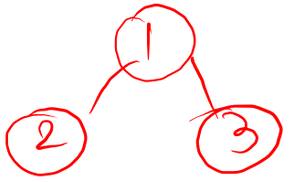
1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9 → 10 → 11 → 12 → 13





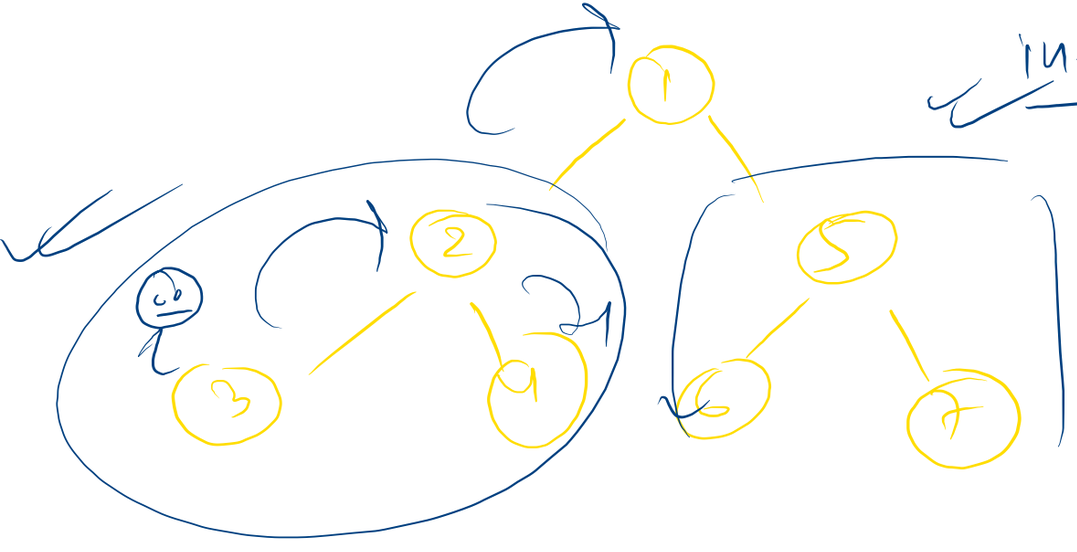
LST → Root → RST

Inside



in-order :-

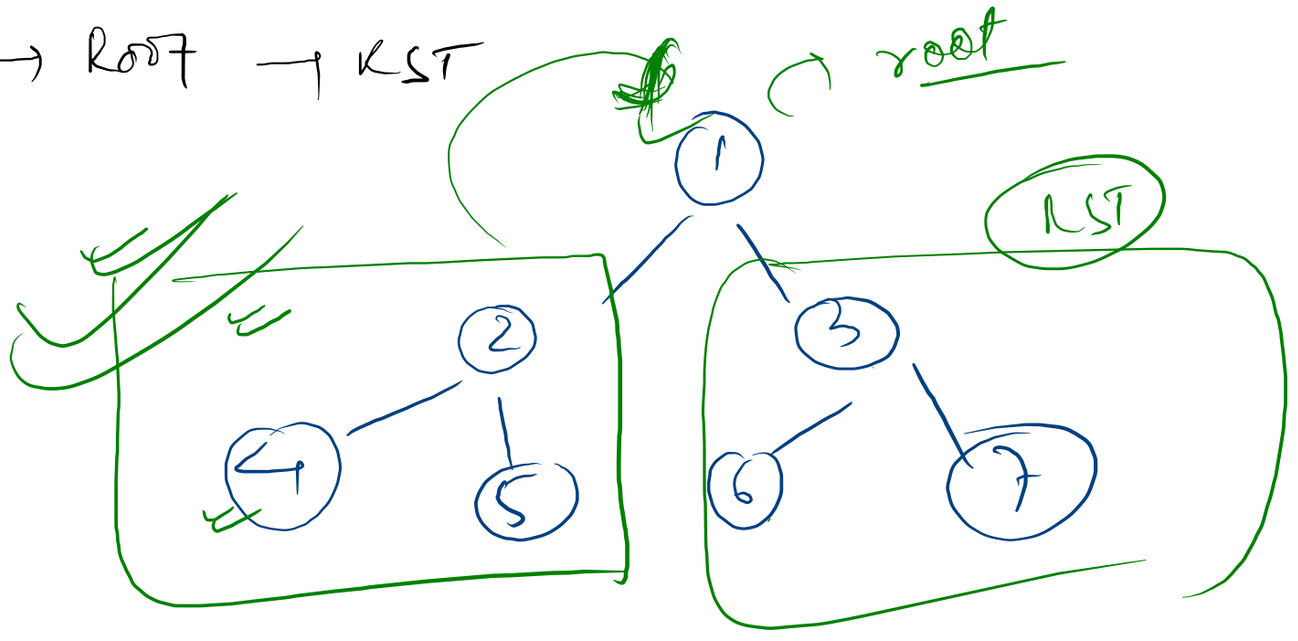
2 → 1 → 3



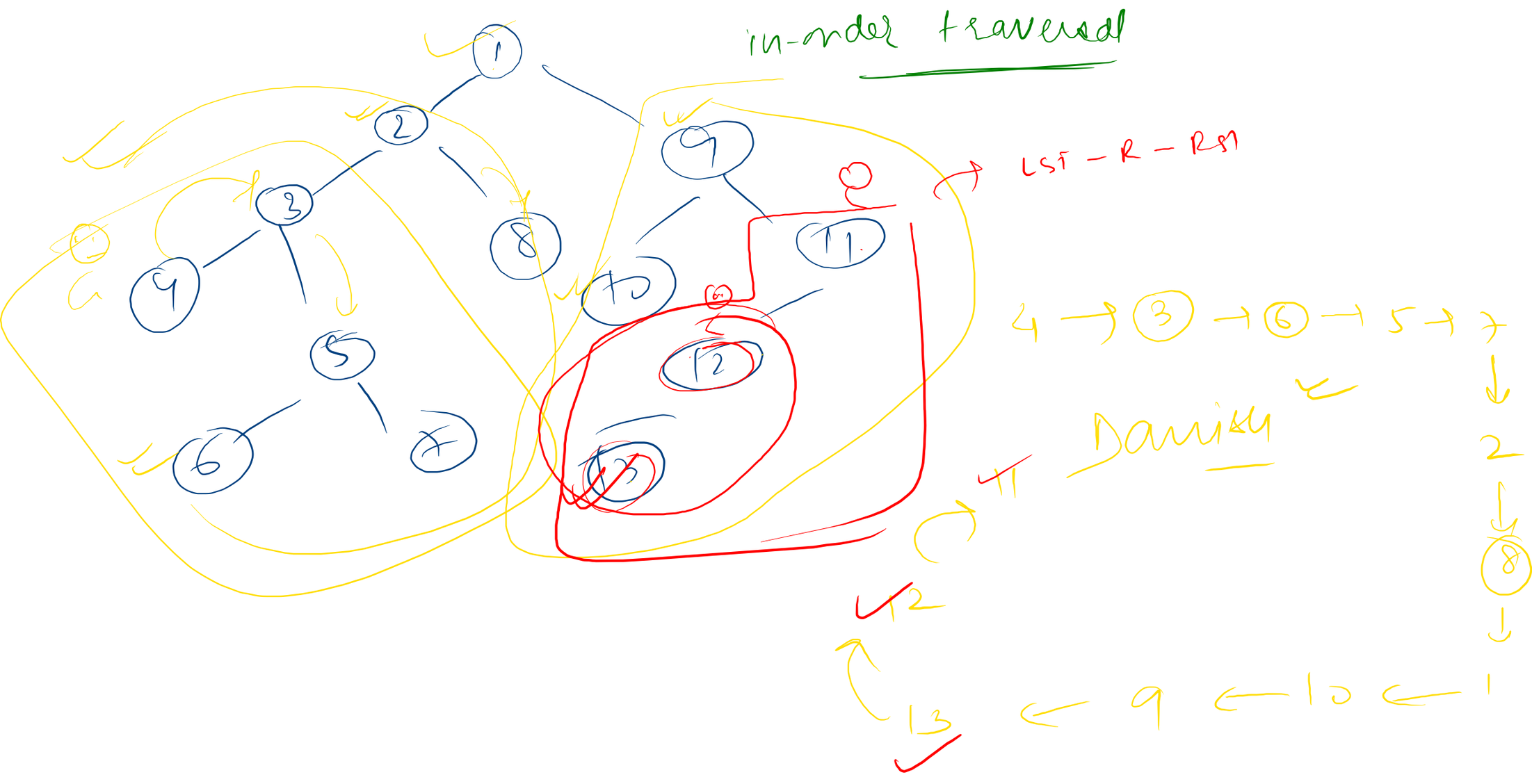
in-order :-

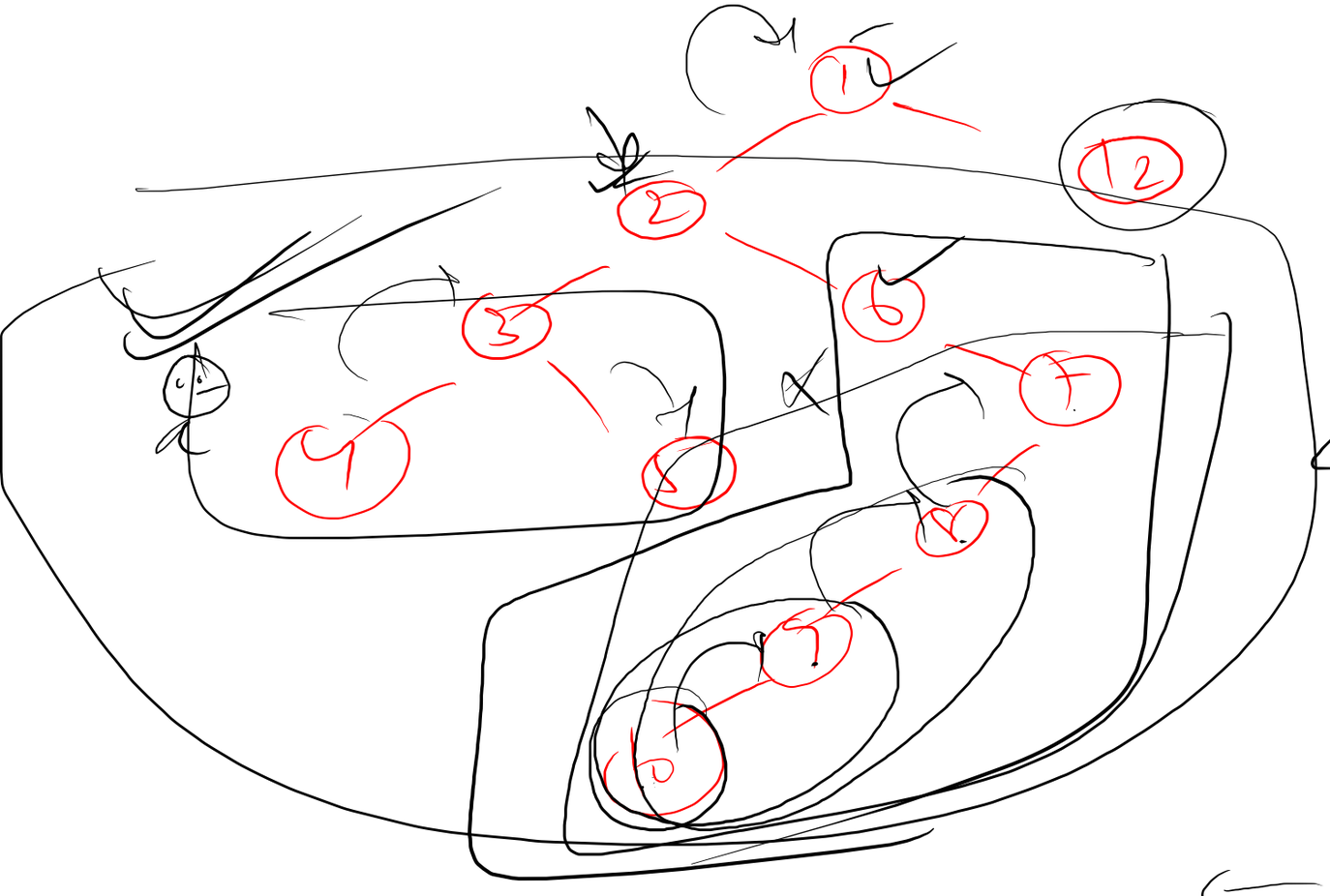
3 → 2 → 4 → 1 → 6 → 5 → 7

LST → Root → RST



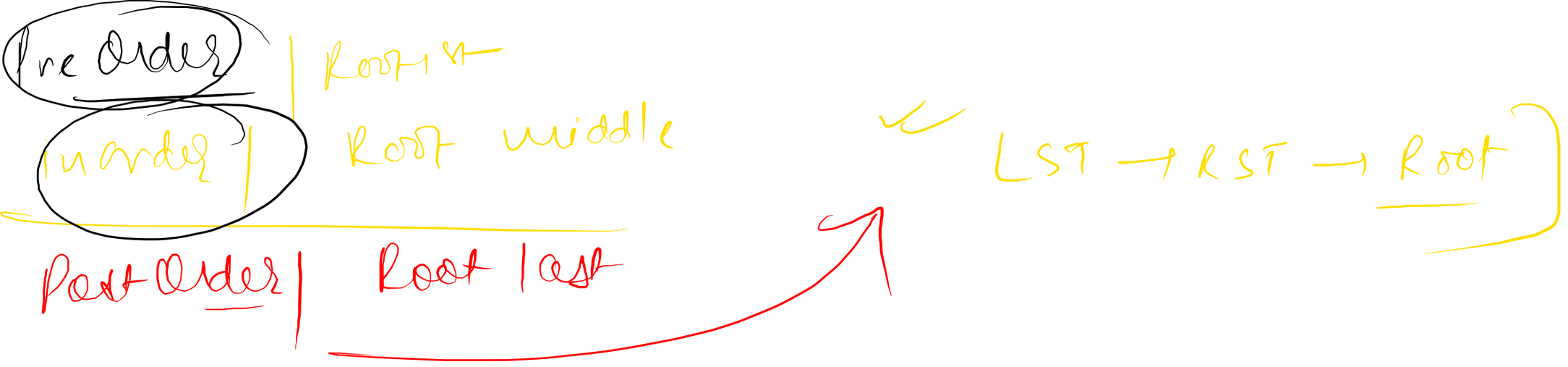
4 → 2 → 5 → 1 → 6 → 3 → 7

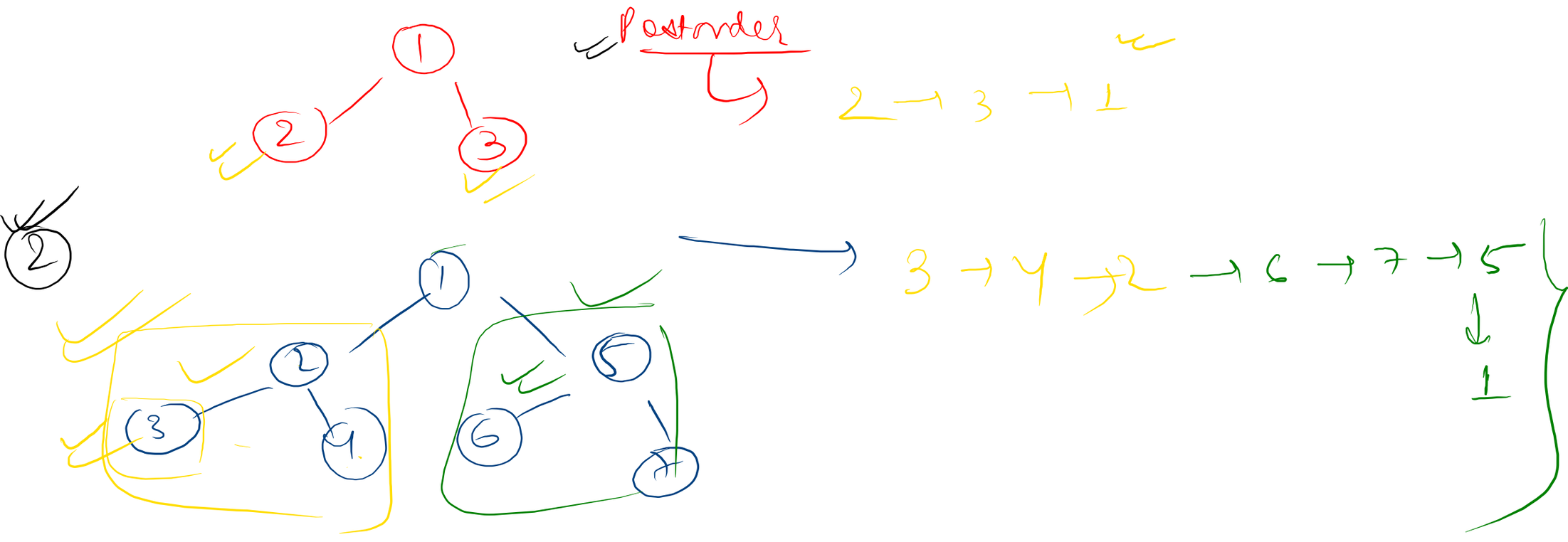

in-order traversal
LST - R - RST
4 → 3 → 6 → 5 → 7 → 2 → 8 → 1 → 10 → 9 → 13 → 12 → 11
Daniku

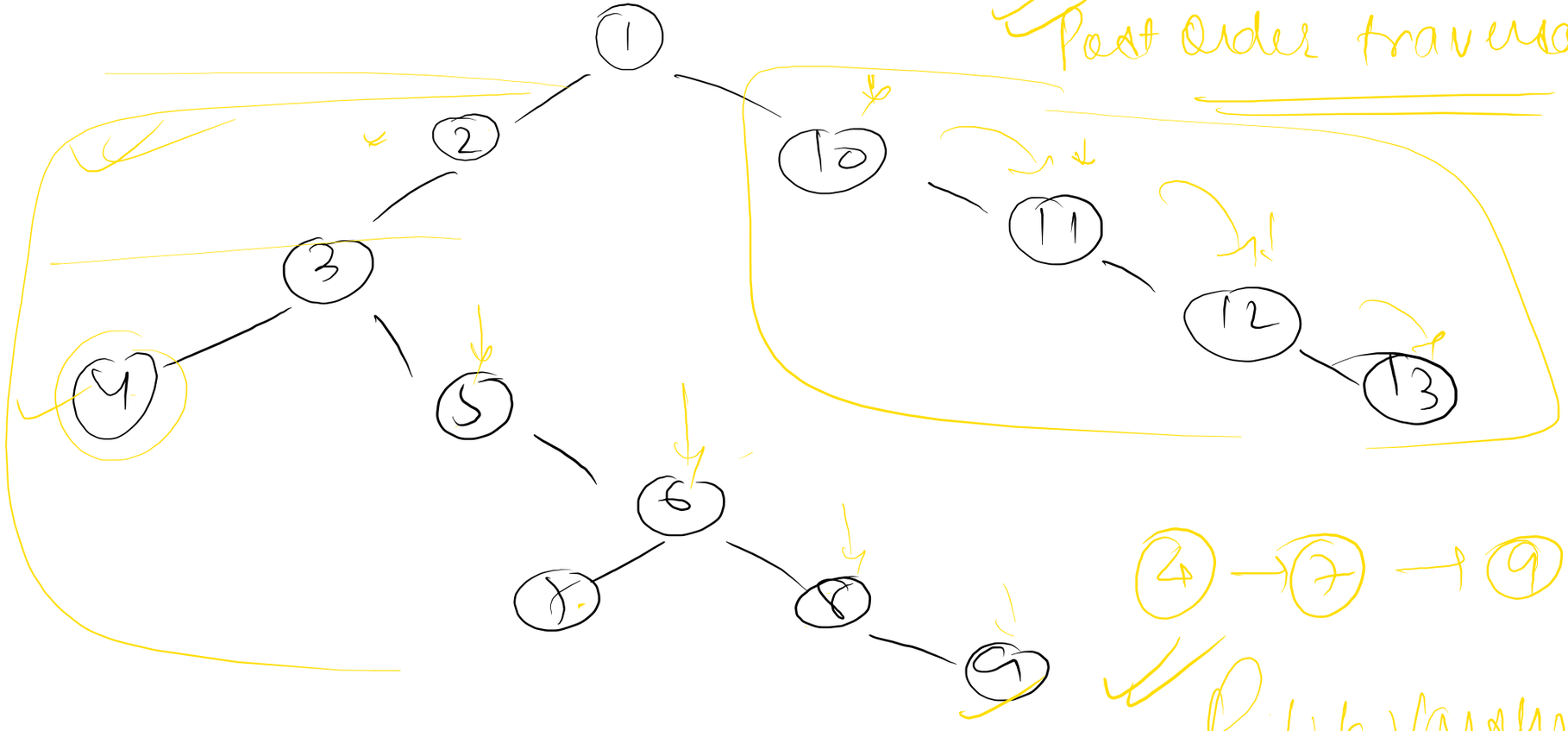


In-order | LST - R - RST

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


Pre Order
Root
In order
Root middle
LST → RST → Root
Post Order
Root


Postorder
1
2
3
2 → 3 → 1
2
1
2
3
4
5
6
7
3 → 4 → 2 → 6 → 7 → 5 → 1



Post Order traversal

4 → 7 → 9 → 8 → 6 → 5 → 3 → 2 → 13 → 12 → 11 → 10 → 1

Right Traversal

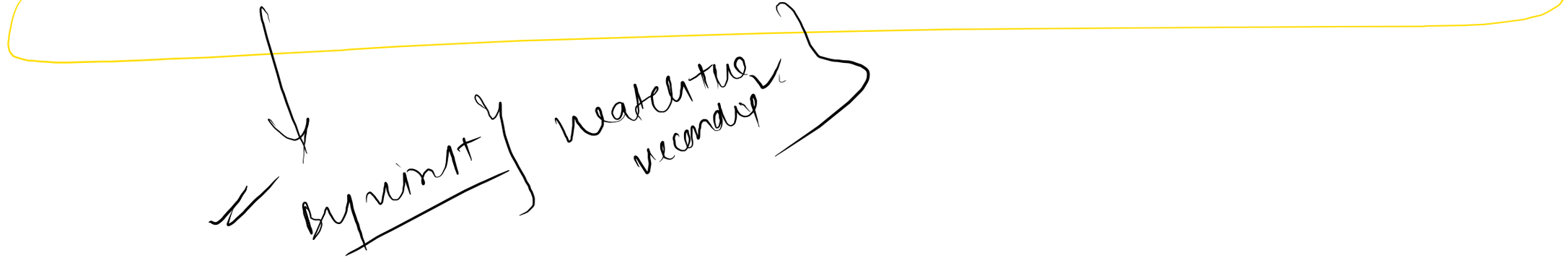
Tree → Traversal + recursion → super strong already !!!

implement tree → solve same coding problems

Revise recursion ( ) and then come

Pre Order | in order | post order

ROOT – LST – RST | LST – ROOT – RST | LST – RST – ROOT



Height of tree = Depth of tree → distance b/w root and d

~~Det~~ D.