Recursion.

A function that calls itself, ... until it doesnt.

assume we have a giftbox and contain inside lot of giftbox open ... find the price assume the price is the ball.

```
def open_gift_box():
    if ball:
        return ball
    open_gift_box()
```

} base case

if doesnt have base case cause stack overflow.

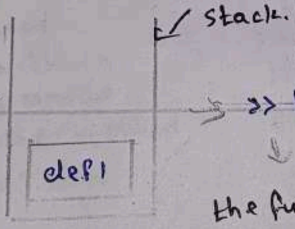
call stack.

Stack execution without resurvive method.

```
def func1():
    print(1)

func1()
```

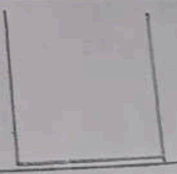
whenwe call the function, the function put in our stack



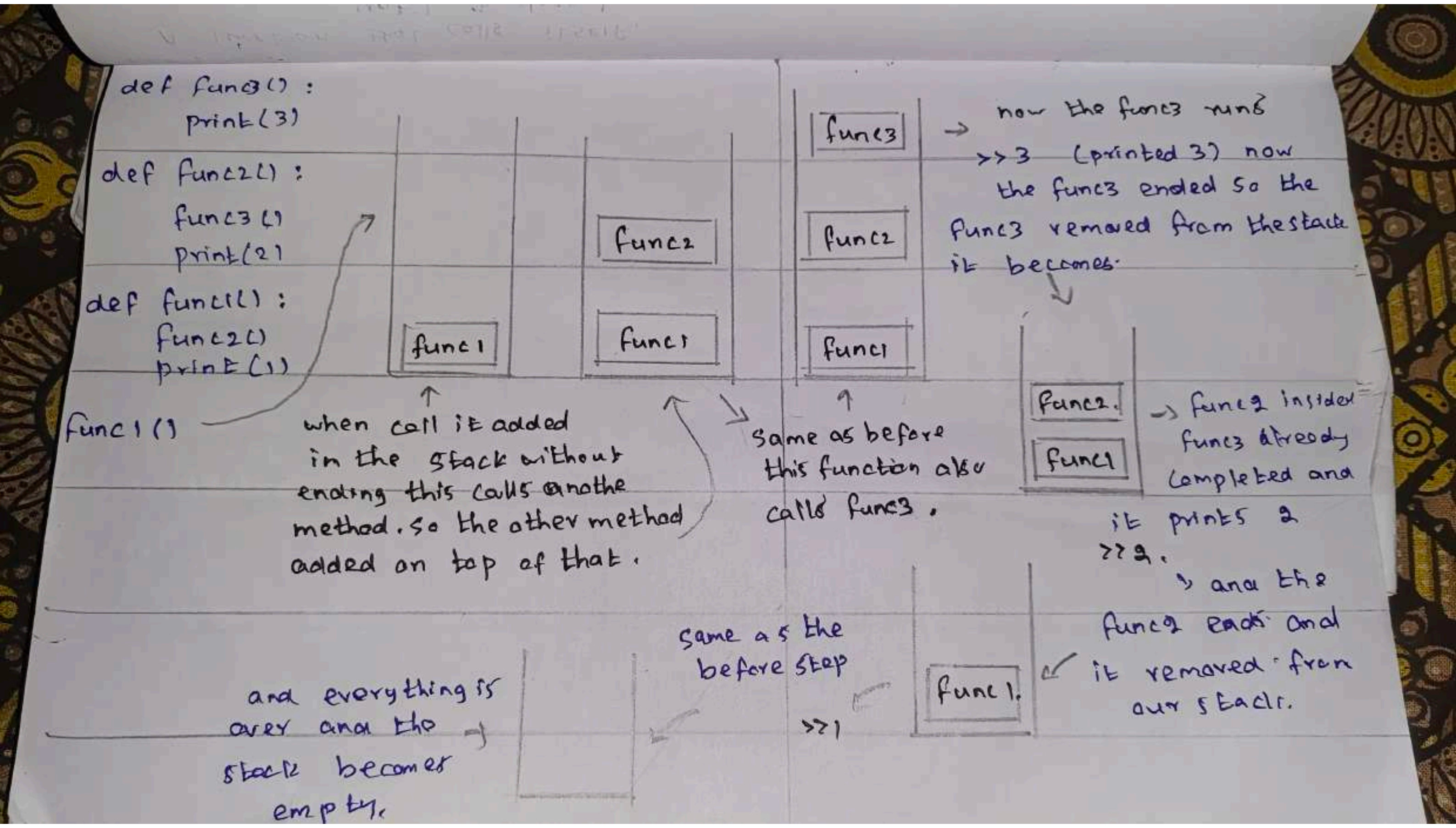
>> 1

the function printed 1 and now the function was end.

so.



the function was removed from our Stack



```
def func3() :
      print(3)
def func2() :
      func3()
      print(2)
def func1() :
      func2()
      print(1)
func1()
```

func1

when call it added in the stack without ending this calls anothe method. so the other method added on top of that.

func2
func1

Same as before this function also calls func3.

func3
func2
func1

→ now the func3 runs
>>3 (printed 3) now the func3 ended so the func3 removed from the stack it becomes:

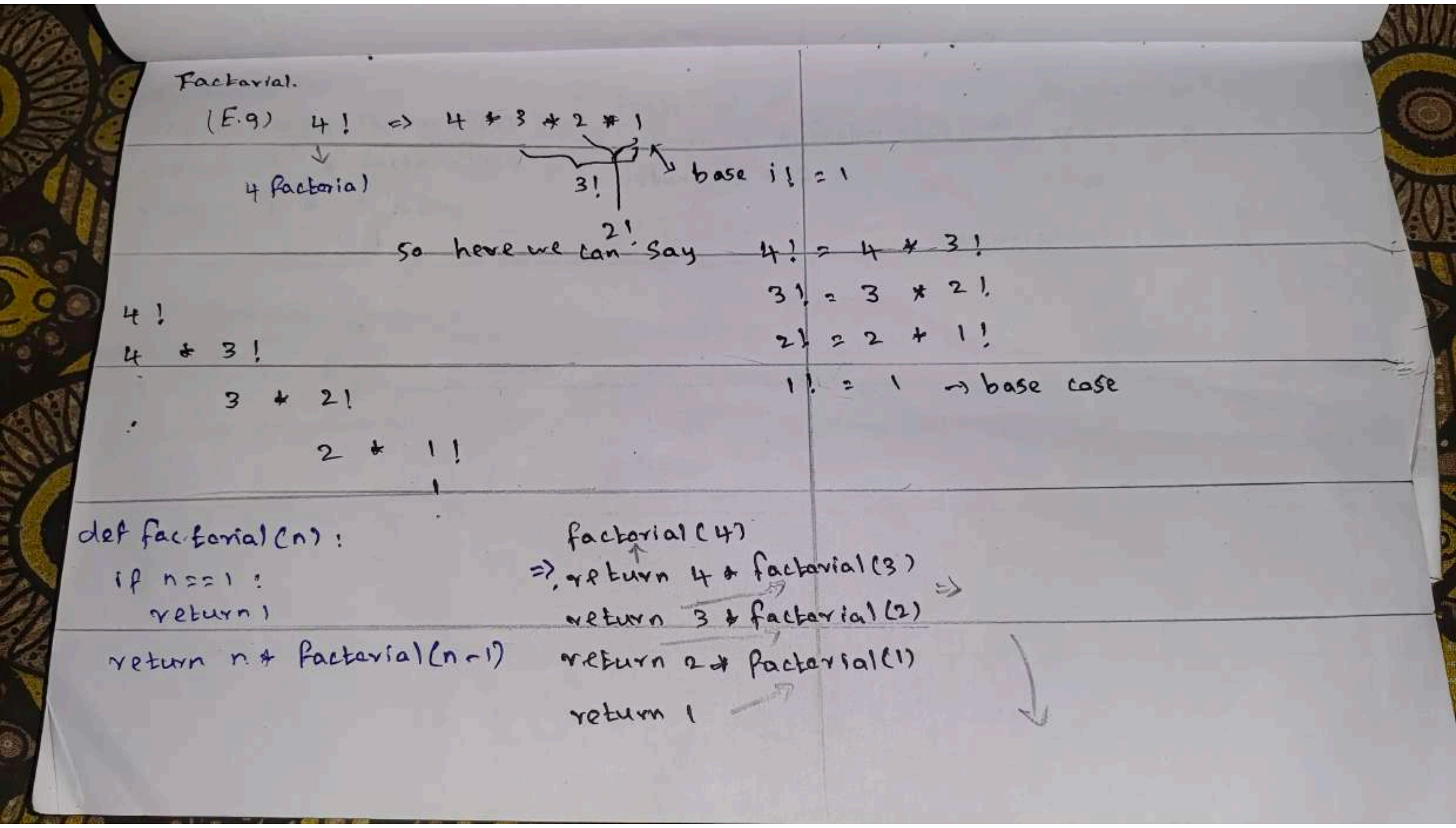
func2.
func1

→ func2 inside func3 already completed and it prints 2
>>2.
→ and the func2 ends and it removed from our stack.

func1.

same as the before step
>>1

and everything is over and the stack becomes empty.



Factorial.

(E.g) 4 ! => 4 * 3 * 2 * 1

4 factorial

3!

2!

base 1! = 1

So here we can say

4! = 4 * 3 !

3 ! = 3 * 2 !

2 ! = 2 * 1 !

1 ! = 1 → base case

4 !

4 * 3 !

3 * 2 !

2 * 1 !

```
def factorial(n) :
  if n==1 :
    return 1
  return n * factorial(n-1)
```

```
factorial(4)
=> return 4 * factorial(3) =>
return 3 * factorial(2)
return 2 * factorial(1)
return 1
```

factorial (4)
return 4 * factorial(3)
return 3 * factorial(2)
return 2 * 1 => 2 .

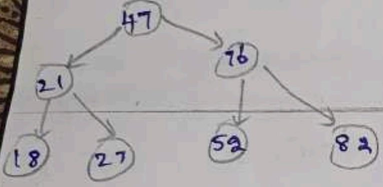
factorial (4)
=> return 4 * factorial(3) =>
return 3 * 2 => 6

factorial (4)
return 4 * 6
⇓
factorial(4) = 24

Tree traversal.

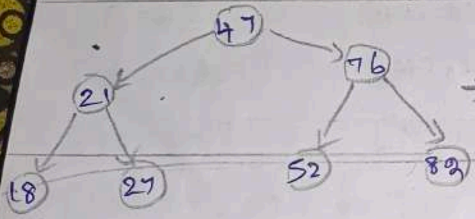
is that going to visit every node in our tree and then were going to take the values and put them in a ~~tree~~ list. and then return the list.

A. But the tree traversal is complicated than a some thing like finked list



start from top 47
→ and next raw 21, 76
and next 18, 27 and
52, 82

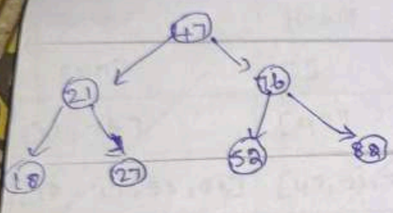
} This is called
Breath ~~for~~ First Search.



Start from bottom left
→ 18 and come up and
come down like 21, 27.
and come up to top
47. got go all way to
bottom left 52. and comup
and come down 76, 82

} Depth First Search

Breath first search.

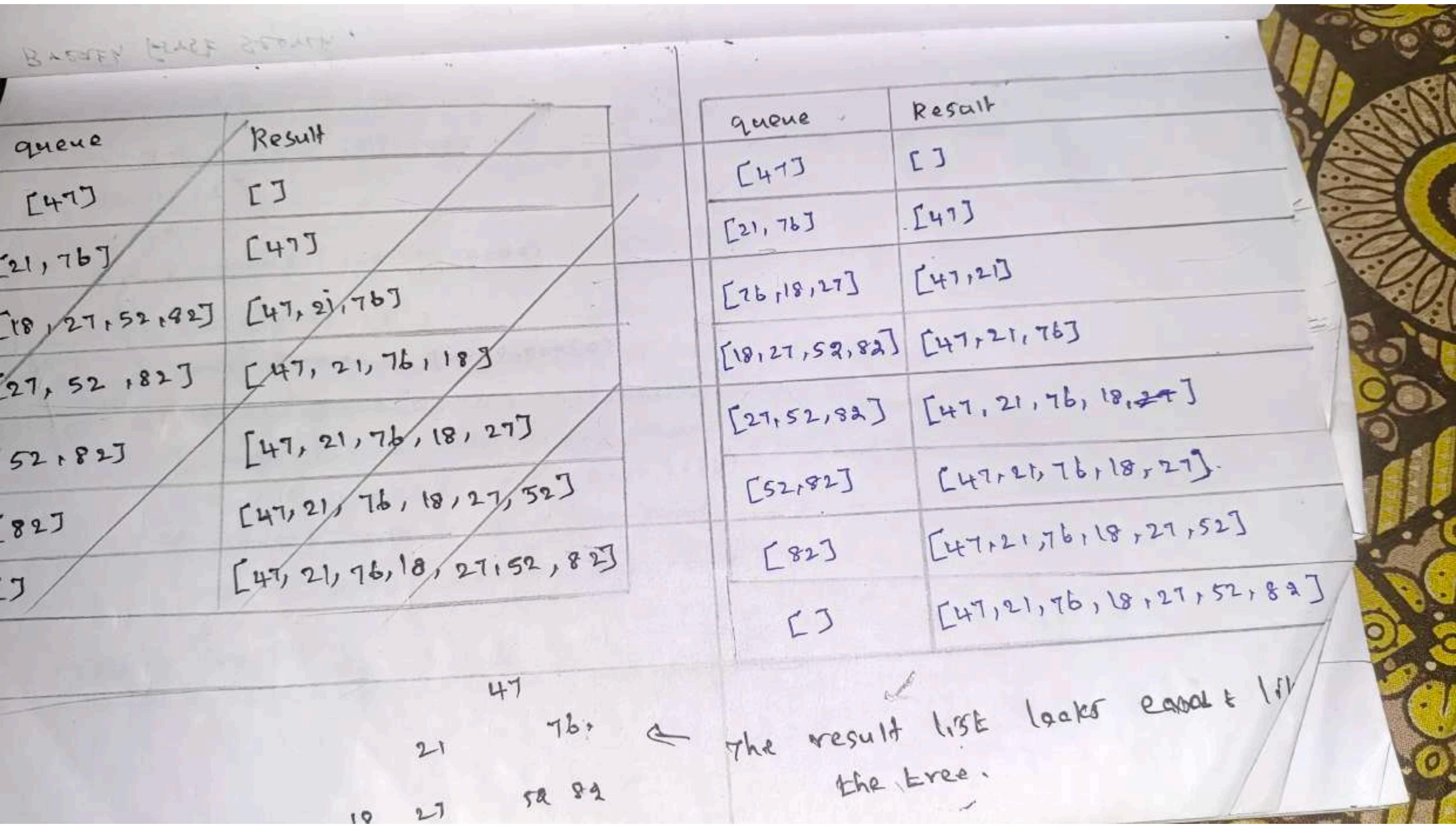


queue
[ ]
→ to Store node, not only value entire node with left and right.

result
[ ]
to return. only storing value not entime node,

queue	result
[47]	[ ]
[ ]	[47]
[21, 76]	[47]
[76]	[47, 21]
[76, 18, 27]	[47, 21]
[18, 27]	[47, 21, 76]
[18, 27, 52, 82]	[47, 21, 76]
[27, 52, 82]	[47, 21, 76, 18]
[52, 82]	[47, 21, 76, 18, 27]
[82]	[47, 21, 76, 18, 27, 52]
[ ]	[47, 21, 76, 18, 27, 52, 82]

after que is empty we can return. the result



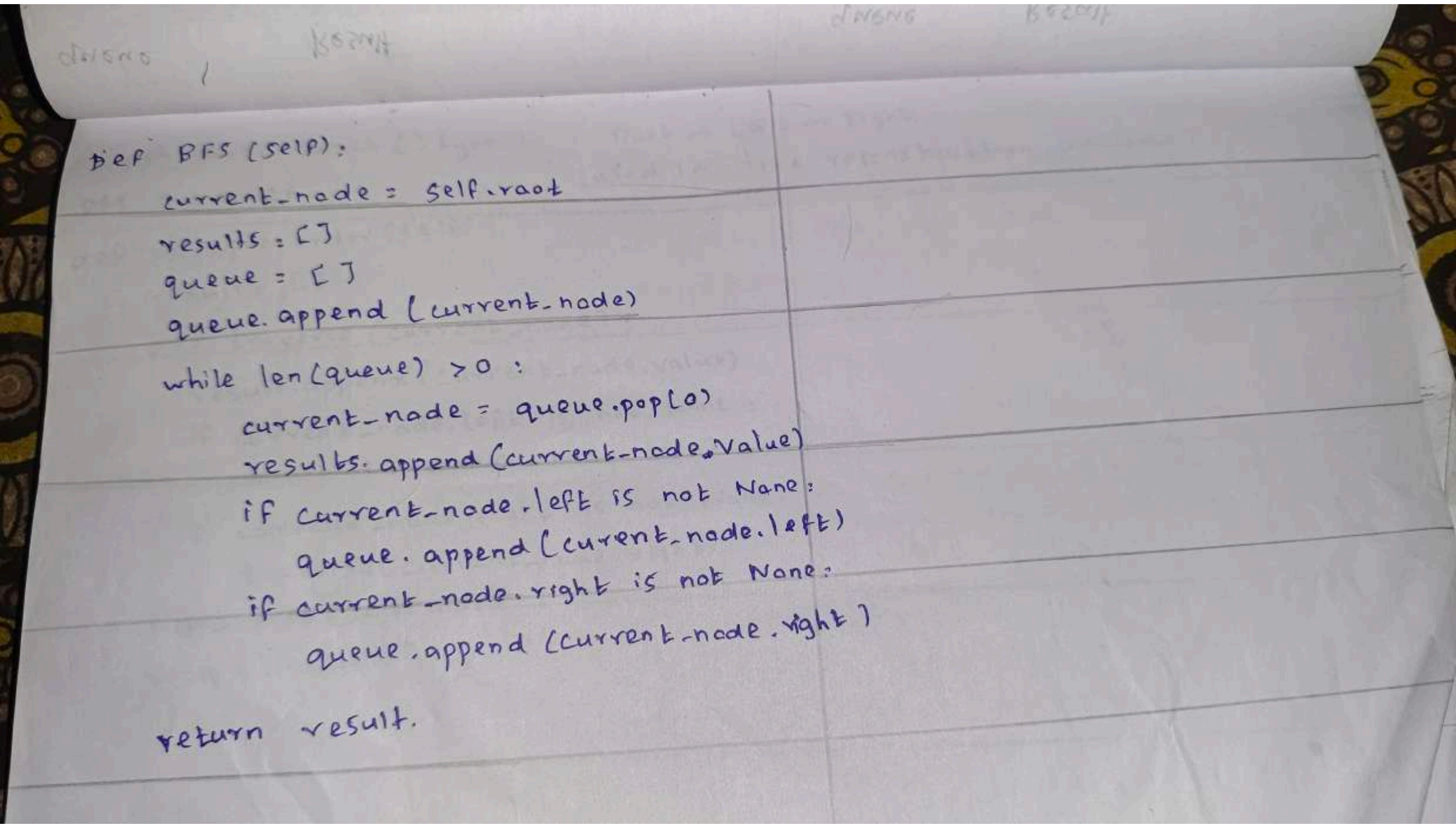
queue	Result
~~[47]~~	~~[ ]~~
~~21, 76]~~	~~[47]~~
~~[18, 27, 52, 82]~~	~~[47, 21, 76]~~
~~27, 52, 82]~~	~~[47, 21, 76, 18]~~
~~52, 82]~~	~~[47, 21, 76, 18, 27]~~
~~82]~~	~~[47, 21, 76, 18, 27, 52]~~
~~]~~	~~[47, 21, 76, 18, 27, 52, 82]~~

queue	Result
[47]	[ ]
[21, 76]	[47]
[76, 18, 27]	[47, 21]
[18, 27, 52, 82]	[47, 21, 76]
[27, 52, 82]	[47, 21, 76, 18, ~~27~~]
[52, 82]	[47, 21, 76, 18, 27].
[82]	[47, 21, 76, 18, 27, 52]
[ ]	[47, 21, 76, 18, 27, 52, 82]

47

21 76 ← The result list looks exact lik the tree.

18 27 52 82



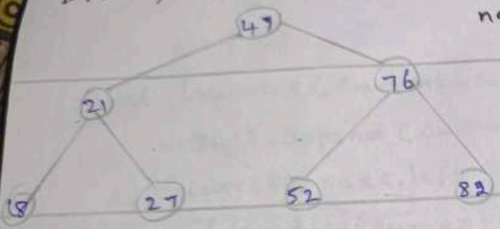
```
Def BFS (self):
    current_node = self.root
    results = []
    queue = []
    queue.append (current_node)
    while len(queue) > 0 :
        current_node = queue.pop(0)
        results.append (current_node.Value)
        if current_node.left is not None:
            queue.append (curent_node.left)
        if current_node.right is not None:
            queue.append (current_node.right)
    return result.
```

Depth first search (3 types). Root -> Left -> Right.
(used in tree reconstruction problems)

DFS pre order.

```
def dfs_pre_order(self):
    result = []
    def traverse(current_node):
        result.append(current_node.value)
        if current_node.left is not None:
            traverse(current_node.left)
        if current_node.right is not None:
            traverse(current_node.right)
    traverse(self.root)
    return results
```

DFS - post order. (used for deleting trees) delete children before parent left→right→root



diff here is. we just going to visit the 47 Node not going to write that value to the list yet. going to left visit 21 and left again 18, now look left no node there then look right of 18 no need there so write 18 value into result.

come up to 21 it has gone left, from 27 look left and right no values so write the valu? to result

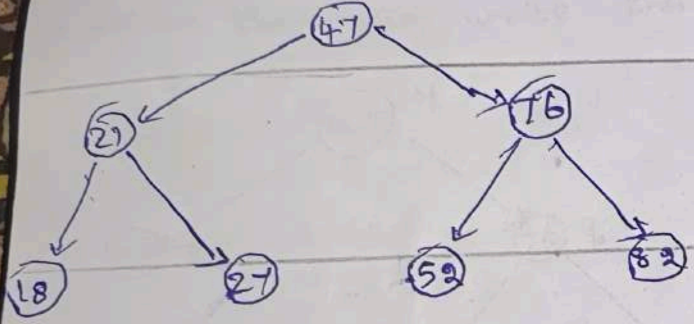
and come back to the 21 again and it goes already left and right so write that value to result. and bring back to 47. look left its already done so go right 76, look left go 52, look left and right no ~~values~~ node write value to result, and comeback 76, it gone left so go right 82 look left and right no nodes so write in result. now we in 76 that gone left and right and its brings backup to the 47. it gone left and it gone right now its value can be written to the result list.

```
def dfs_post_order(self):
    results = []
    def traverse(current_node):
        ~~result.append(current_node.value)~~
        ~~if current_node.left is not None :~~
            ~~traverse(current_node.left)~~
        ~~if current_node.right is not None :~~
            ~~traverse(current_node.right)~~

        if current_node.left is not None:
            traverse (current_node.left)

        if current_node.right is not None:
            traverse(current_node.right)
        result.append(current_node.value)
    traverse (self.root)
    return results.
```

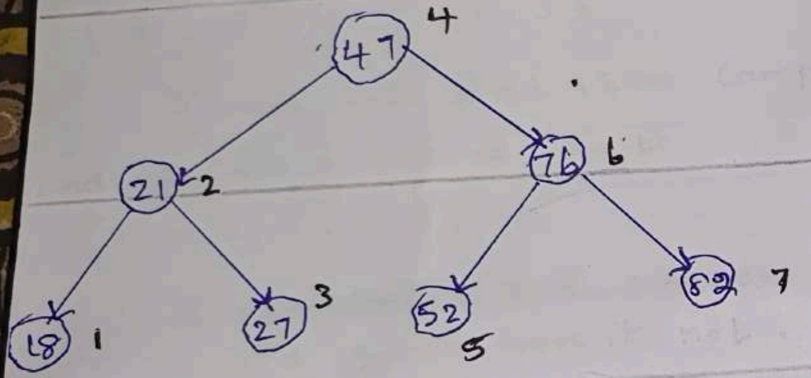
DFS inorder → (important for BST (gives sorted order)) left → root → right.



visit first and go to left 21 and go left 18
now 18 will try to go to the left nothing there,
write that 18 to result list
↓
in this point its looks exactly the the same as
did in DFS post order
But instead of going left and right writing
it value. it going left and writing the value

we get this in diff order.
now come up to the 21. it has gone left. ~~the 21 will go left write its~~
~~value and then~~ go right. now its gone to write its value and then go
right. The 27 will go left write its value and then go right.
That brings up 47 node. its gone left. now we can write its value and
go right. The 76 will start out by doing ~~by~~ left. The 52 go left write its
value and go right

the 76 already gone left so write its value and go right, 82 go left nothing there so write that value



[18, 21, 27, 47, 52, 76, 82]

↓

the order that written in the list 𝛾, the numerical order, small to ragage asending

```
def dfs_in_order (self):
    result = [ ]
    def traverse (current_node):
        if current_node, left is not Nane:
            traverse(current_node.left).          left. . . .
        ~~traverse~~
        result. append (current_node.Value)       traverse(self.root)
                                                  return results
        node.right is not None:
```