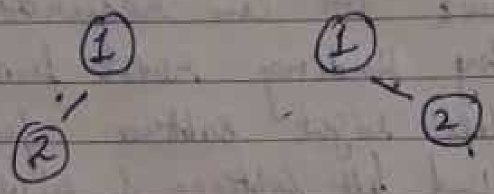
Assignment - 3

A) (a) ~~Attempt~~ Given a Binary Trees :-

(a) Post order & Level order (Not Possible)

We cannot identify uniquely if a tree is ~~to~~ the binary tree which was given, as Post order is left child root child then parent we can not identify if a node has only one root so will that be ~~an~~ the left child or the right child even if the level order traversal is given. We can get close to the tree with help of level order but not be sure.

Ex :



level order : 1, 2
Post order : 2, 1

(b) Inorder & Preorder. (Possible)

We can identify the tree uniquely as with help of preorder we can identify the root (first element) and then from inorder identify elements on left subtree and right subtree then as before in left subtree find root (first element of ~~...~~ ~~...~~ remaining left nodes) from preorder & then left left subtree & ~~p~~ right right subtree we can do this recursively till we find our unique tree.

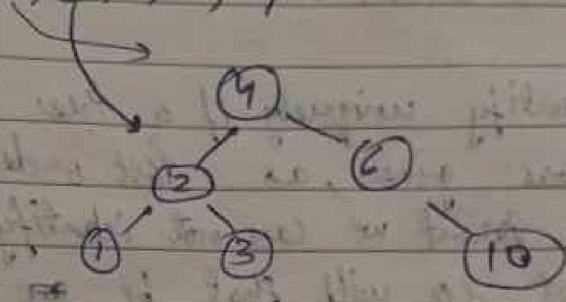
(c) ~~As~~ Inorder & Level order (Possible)

As in the previous one we can do similarly in this one by finding roots from the level order (first element)

& finding left subtree & right subtree from inorder and do this recursively till we find the unique tree.

Example) In order : 1 $\boxed{2}$ 3 $\boxed{4}$ 6, 10 (L. Tree ← , → R Tree)

Level Order : 4, 2, 6, 1, 3, 10



(d) Inorder & Postorder (Possible)

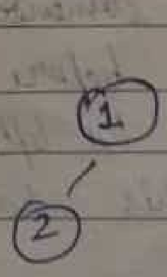
Similar to above cases we can recursively make the unique tree ~~with~~ by finding root from post order (last element) & left right subtree from inorder. & then recursively find left subtree & right subtree.

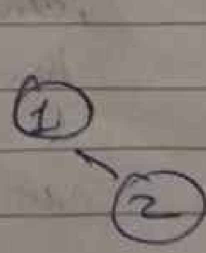
(e) Postorder & Preorder (Not Possible)

Similar to first case we do not know in single child if the child is on the left subtree or right subtree

Example : Preorder ~~Inorder~~ : 1, 2

Post order : 2, 1





(b) Pseudo code for Preorder & Inorder

Node* Unique tree (Inorder , Preorder)

← List of inorder Elements

← list of Preorder Elements

if (inorder & preorder are empty)
    return nullptr   // Base case
Root = Preorder [0]   ← Root is first element of preorder
Left Inorder ← Elements before Root in the ~~preorder~~ inorder
Right Inorder ← " after " " " inorder
Left Preorder ← Preorder of same Elements in left inorder
Right Preorder ← " " " " " right "
Root → LChild = Unique tree (left Inorder, left Preorder)
Root → RChild = Unique tree (Right Inorder, Right Preorder)
return Root

(C) Pseudo code for Level order & Inorder.

Node* Unique tree (Inorder, LevelOrder)

if (Inorder & LevelOrder are empty)
    return nullptr
~~Root~~ = LevelOrder [0]
left Inorder   // similar ~~word~~ to the above code.
Right Inorder
Left Level Order
Right Level order
Root → LChild = Unique tree (left Inorder, left level Order)
Root → RChild = Unique tree (~~Right~~ Inorder, Right level order)
return Root

Example: 1st Call: Inorder = [1, 2, 3, 4, 6, 10]
Level Order = [4, 2, 6, 1, 3, 10]

(LChild)
2nd Call: Inorder = [1, 2, 3]
Levelorder = [2, 1, 3]

(RChild)
Inorder = [6, 10]
Preorder = [6, 10]

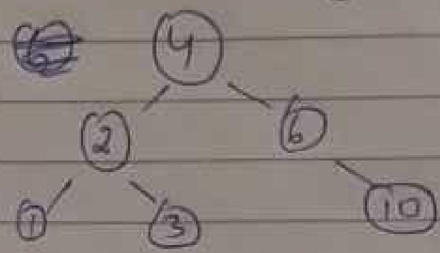
3rd Call:
LChild
Inorder: [1]
LO: [1]

RChild
IO: [3]
LO: [3]

return null
LL child
IO: [ ]
LO: [ ]

IO: [10]
LO: [10]

Tree =



(d) Pseudo Code for Inorder & Postorder.

Node* Unique tree (Inorder, Postorder)

if (Inorder & Postorder are empty)
return nullptr.
root = Post Order [-1] ← Last element of post order.
Left Inorder
Right Inorder // Similar to above codes.
Left Postorder
Right Postorder
Root → LChild = Unique tree (Left Inorder, left Postorder)
Root → RChild = Unique tree (Right Inorder, Right Preorder)
return Root.

A2) I am using vector, array representation of the binary tree

First going through array I find out the location of bomb then I store the effect of bomb as vector of vectors where first ~~represent~~ element represents first location of bomb's effect, second represents secondary regions to be affected & so on.

First I find location of bomb in $O(n)$ time then call function bomb Effect that calls function around that ~~with~~ has time complexity $O(1)$ ~~...~~ which returns the surrounding junctions which is to stored by bombEffect function that then calls around ~~fund~~ function on these junctions until nothing comes back from around function.
Then I just print the vector of vectors to get desired output.

Time Complexity : (function) Bomb Effect $\Rightarrow O(n)$ (as it calls around maximum of n times the ~~nodes~~ n = no. of nodes)
(function) Around $\Rightarrow O(1)$ $\Rightarrow$ (const time)

$\boxed{\text{Total time complexity} \Rightarrow O(n)}$

Space Complexity : $\boxed{\text{Binary Tree} \Rightarrow O(n)}$ (n memory allocations in vector, array)
(function) Around $\Rightarrow O(1)$ $\Rightarrow$ (const space)
(function) Bomb Effect $\Rightarrow O(n)$
$\Rightarrow$ 3 vector, arrays (ans, current, cur) have maximum n elements so 3n

$\boxed{\text{Total space complexity} \Rightarrow O(n)}$

A3. In this I make a tree for the students then find indexes of students that will make a pair (indexing starts with 1). I check if log is same for indexes (on same level students)

Then check if parents are the same or not.

Time Complexity

Finding Students ⇒ O(n)
Taking log & parents ⇒ O(1)

$\boxed{\text{Total Time Complexity} \Rightarrow O(n)}$

Space Complexity

Binary Tree ⇒ O(n)
Index storing & have constant space ⇒ O(1)
(2 elements)

$\boxed{\text{Total Space Complexity} \Rightarrow O(n)}$

A4) In fourth question I use the AVL data structure with number of left nodes & number of right nodes as additional attributes of a node with the height

- While inserting if on left subtree i just add 1 to count of left nodes similarly for right nodes & decrement while deleting.

- After creating a tree & getting the query by index I use search function to find the node at if

index = left Nodes + 1 → Current element needed
index <= left Nodes → Element in left subtree so go find in " ".

index > left Nodes → Element in Right subtree, so Go find in right subtree the index = index − (left nodes + 1)

Time Complexity:

Insert = ~~log(n)~~ O(log(n)) ⇒ as balanced.

left Rotate, Right Rotate ⇒ O(1)

After Query Received.

- Search ⇒ O(log(n)) ⇒ Worst case go to the deepest level that is log n height.
  (So max visit log(n) nodes)
- Delete ⇒ O(log(n)) ⇒ Delete the query after choosing.

Other Helper function ⇒ O(1).

$$\boxed{\begin{array}{l}\text{Insert} \Rightarrow O(\log n)\\ \text{Query execution} \Rightarrow O(\log n)\end{array}}$$ ⇒ search & delete.

Space Complexity:

Tree ⇒ O(n)
Attributes ⇒ O(1)

~~Also~~. Constant additional space used in other function if used

$$\boxed{\text{Total space Complexity} \Rightarrow O(n)}$$