190475 - Mandar Bapat

190818 - Shorya Kumar

190908 - Tejesh Vaish

Q.1] The pseudo code is divided into 2 parts :

(i.) Sorting the given list of 'n' items L using Merge sort.

(ii.) Doing an inorder traversal of the BST and filling the nodes with values from the sorted list L'.

(A.) Sorting

→ We use the Mergesort algorithm to sort the given list.

Time complexity = $O(n \log(n))$

(B) Inorder Traversal :

→ We assume we have the sorted list L'. Elements of L' are denoted using '[ ]' brackets. So, L'[1] denotes the first element.

Pseudo Code :

```
1   int i = 1 ;   // global variable

2   makeBST (node* p , int* L') {
3       if (p == NULL) {   // null pointer check
4           return ;
5       }
6       makeBST (p → left , L') ;
7       p → data = L'[i] ;
8       if (i != n) { i++ ; } // avoiding overflow
9       makeBST (p → right , L') ;
10  }
```

Time complexity for Inorder traversal

$= O(n)$

Total time complexity of the entire algorithm $= O(n) + O(n \log(n))$

$= O(n \log(n))$

(c) Correctness:

→ The code is similar to Inorder Traversal algorithm. We prove the correctness using Structural Induction.

Base Case 1: When there is only 1 node in BST and length of list is 1 too.

Only line 7 of pseudo code affects the BST. Root node is filled with the only element present.

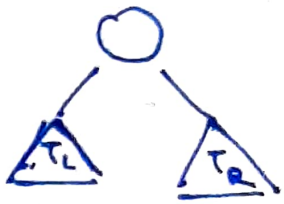
Hence Base case 1 holds true.

Base Case 2: Empty tree

Condition on line 3 holds true. Algorithm outputs nothing.

Hence Base Case 2 holds true.

Constructor case: Assume that the algorithm holds true for left and right subtree of the given tree:



Algorithm starts from root node. Line 6 is executed. Hence T_L is filled first. T_L is correctly filled based on our assumption.

Now, assume T_L has n_L nodes. Hence, value of $i = n_L + 1$ after execution of T_L.

$\because$ L' is sorted,

$L'[1], L'[2] \ldots, L'[n_L] \leq L'[n_L+1]$

Root node is filled with $L'[n_L+1]$.

Hence, value at root node $\geq$ value at any node of T_L

After filling the root node, T_R is filled recursively.

T_R is filled correctly based on our assumption.

$\because$ L' is sorted

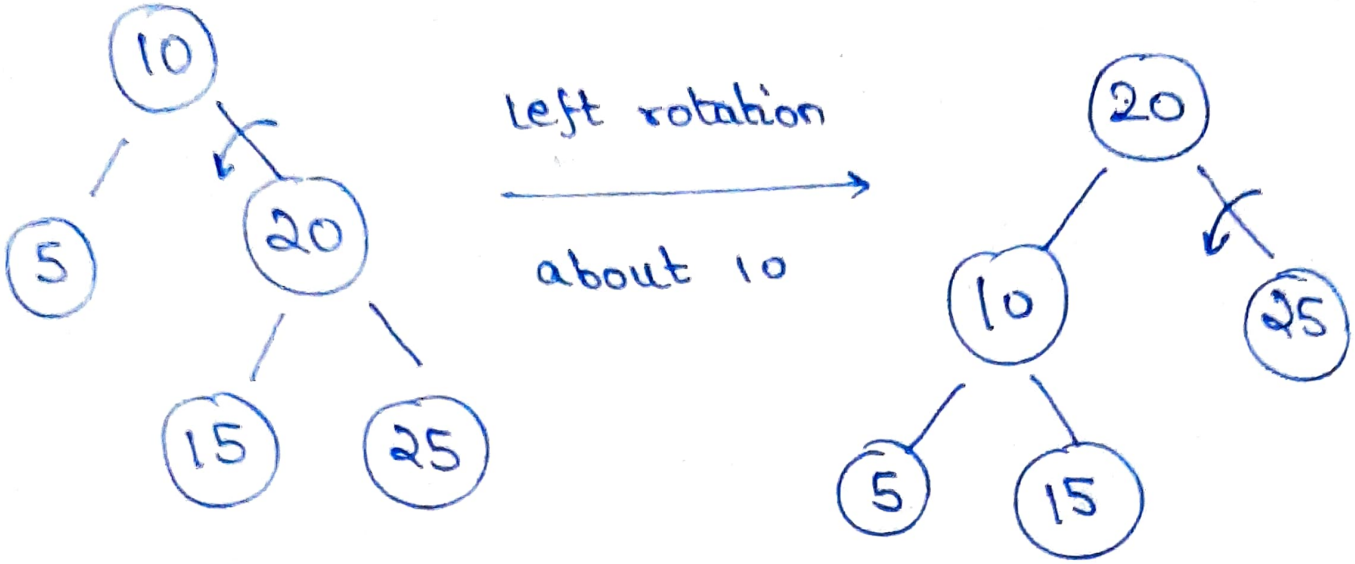
$L'[n_L+1] \leq L'[n_L+2], L'[n_L+3] \ldots L'[n]$

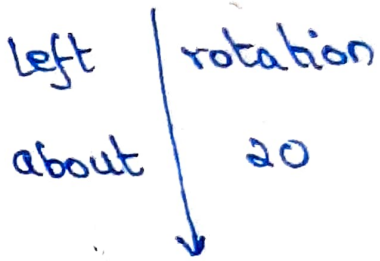
Hence, value at root node $\leq$ value at any node of T_R

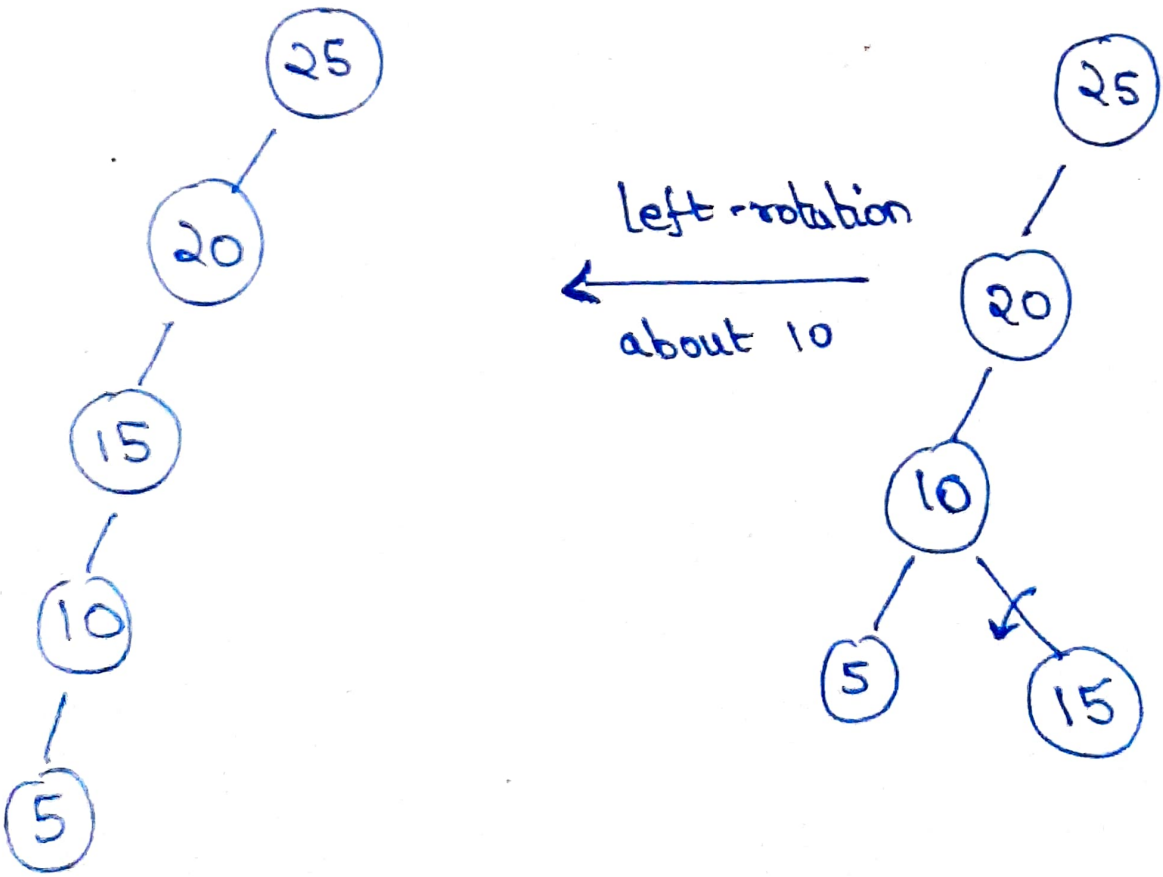
Hence the entire Bstree is correctly filled.

By structural Induction, the algorithm outputs correctly for any size of BST.

(a.)


10
5
20
15
25
left rotation
about 10
20
10
25
5
15


left rotation
about 20


25
20
15
10
5
left-rotation
about 10
25
20
10
5
15

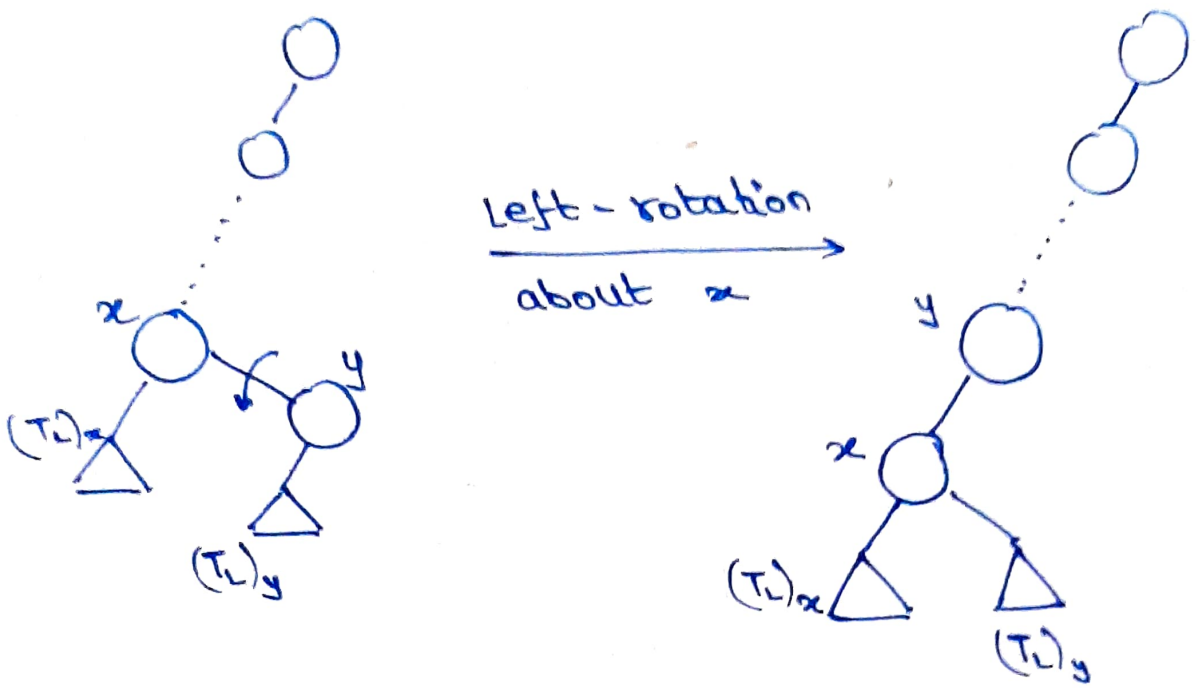
(b.) Procedure: First we find the topmost node that has a non-null right child. Then we apply left rotation about it, till there are no such nodes in the entire tree.

- Correctness of the Algorithm:

→ Let 'd' denote the depth of the topmost node with a non-null right child.

→ All the BSTs can be effectively grouped into 2 cases. Let 'x' be the topmost node at any step of algorithm that has a non-null right child 'y'.

Case 1: y has a null right child



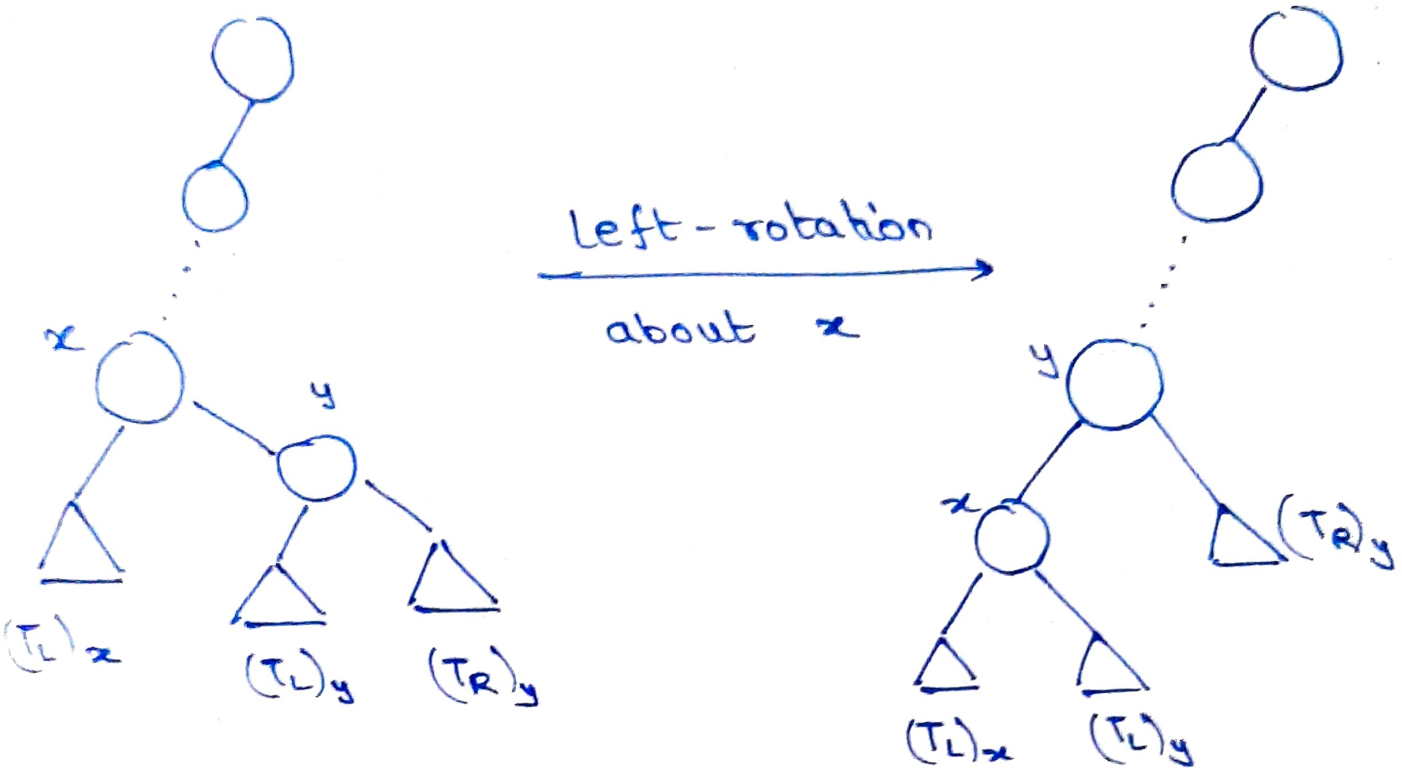
In this case, we perform a left-rotation about x. $(T_L)_x$ = Left subtree of x

$(T_L)_y$ = Left subtree of y

Depth of x increased by 1.

Algorithm can be recursively applied again.

Case 2 : Right child of y is non-null.



In this case, we perform left-rotation about x.

Now, the topmost node with a non-null right child is 'y'.

Depth of $(T_R)_y$ from x (the topmost node with non-null right child)

= 2

Depth of $(T_R)_y$ from 'y' after left-rotation is performed = 1

Hence, the depth of the right subtree from the topmost node with non-null right child decreases.

After almost Height(subtree($(T_R)_y$)) operations, we effectively reduce the case to case 1.

Since the number of nodes are finite the algorithm exhaustively reduces all right subtrees in finite time

→ Algorithm reduces the BST to a left-linear BST in finite number of steps.

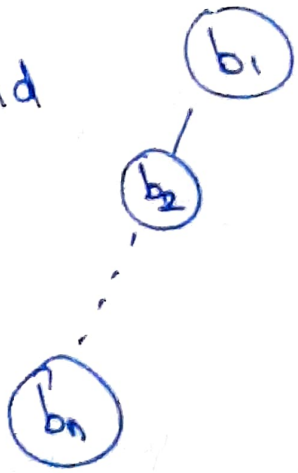
(c) Let S_1 = set of all BSTs

S_2 = set of all BSTs which are left-linear.

(1) Note that for a given set of nodes, the BSTs made from the given set have the same unique left-linear BST.

Proof: Suppose we have 2 BSTs B_1 and B_2 that have the same set of nodes, and have 2 different left-linear BST T_1 and T_2, say

and



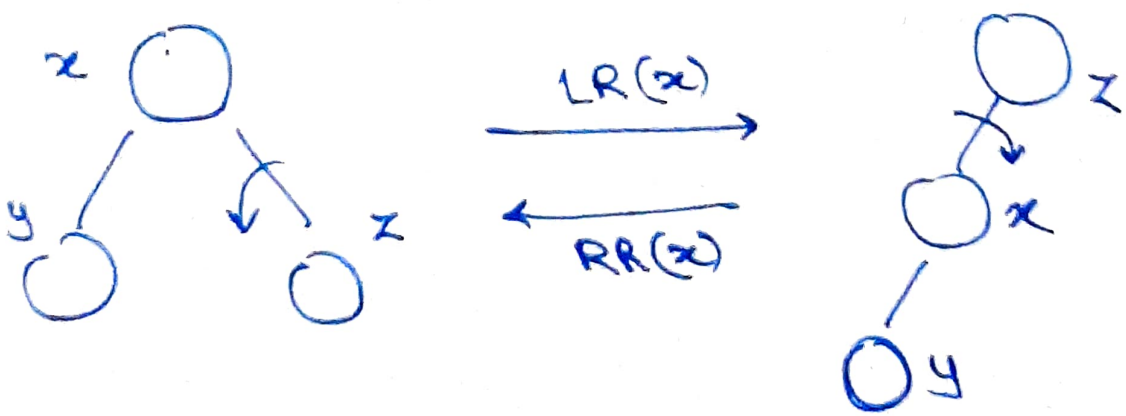
Now, given a set of numbers/nodes, there can be only 1 unique

decreasing (or increasing) order.

Hence, this violates our assumption that T_1 and T_2 are different.

Hence Proved.

(2.) Also note that left rotations and right rotations are inverse operations on each other, i.e, when applied on the same edge, the configuration of the tree remains unchanged.



(Proof not provided as this is given in lecture notes)

(3.) Now consider a map from set S_1 to S_2. This is a many to one map [From result (1.)]

This suggets that when we consider the reverse map, a left-linear BST consisting of a certain nodes can be converted to any of its parent BST from which it was made, by apply the reverse operation of right rotations in exactly the reverse order.

Existence of a many to one map is guranteed by the proof in part (b.).

So suppose for a given BST B_1, we apply left-rotation operations $[L_1, L_2 \dots L_i]$ on B_1 to convert it to a left-linear BST, then we can apply equal amounts of right-rotations $[R_i, R_{i-1} \dots R_1]$ on the respective edges to get a BST from the left-linear BST.

Q.2.] (d.) Worst case happens when the BST to be converted to left-linear tree is entirely right-linear tree.

For any right-linear BStree, with n nodes, maximum (n-1) left-rotations are required.

Hence upper bound on number of left rotations = (n-1)

→ A similar case happens for upper bound on Part (c). For to convert an entirely left-linear BStree to right-linear BST, (n-1) right rotations are required.

Hence upper bound on number of right rotations = $(n-1)$

A (3) (9)

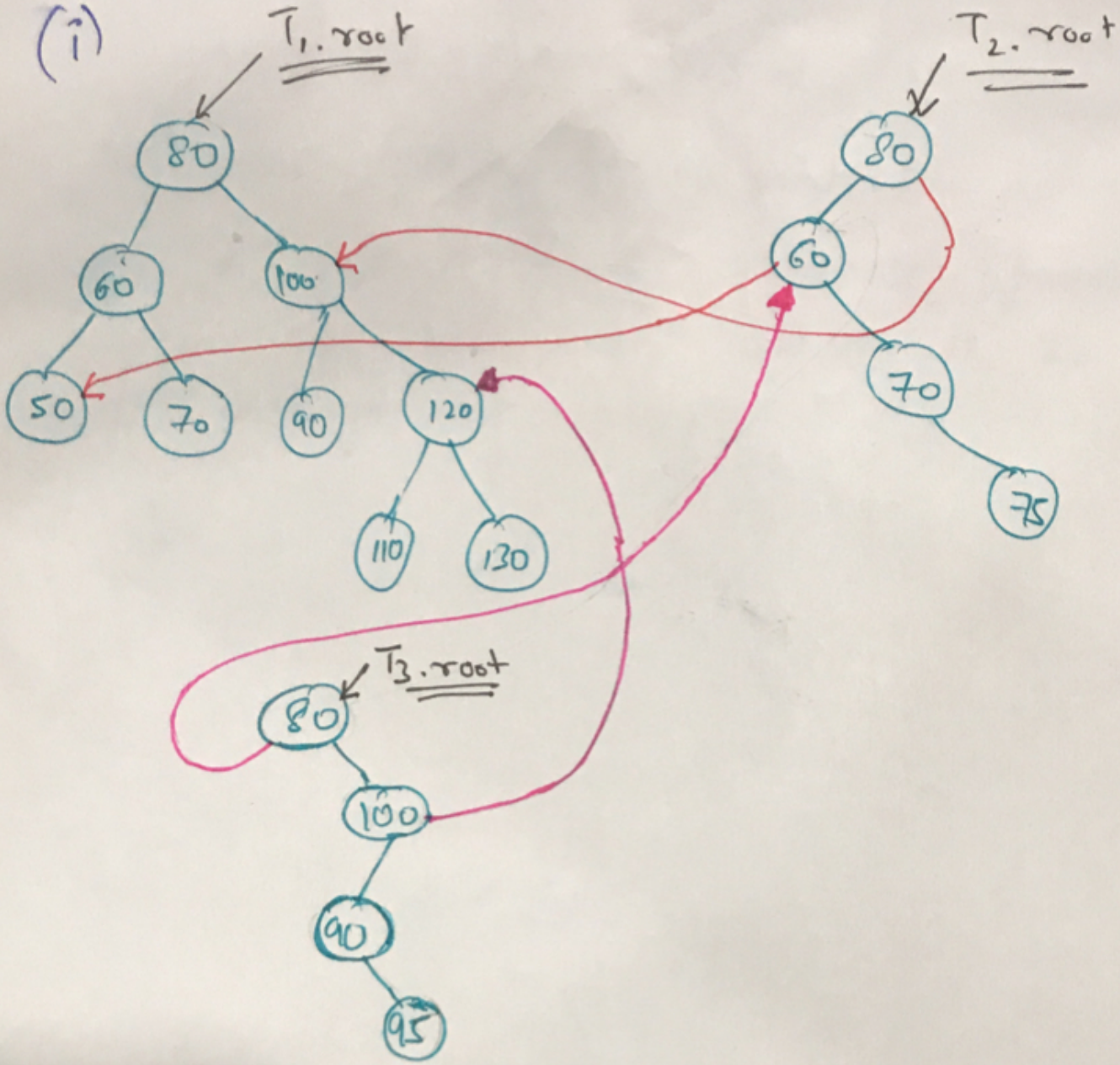
We always downwards from root, to do the insertion or deletion and, store that path from root to the node where insertion is to be done, which doesn't require any parent pointer.

→ We don't traverse the tree upwards.

→ We can have parent pointers, but it will be a wastage of memory.

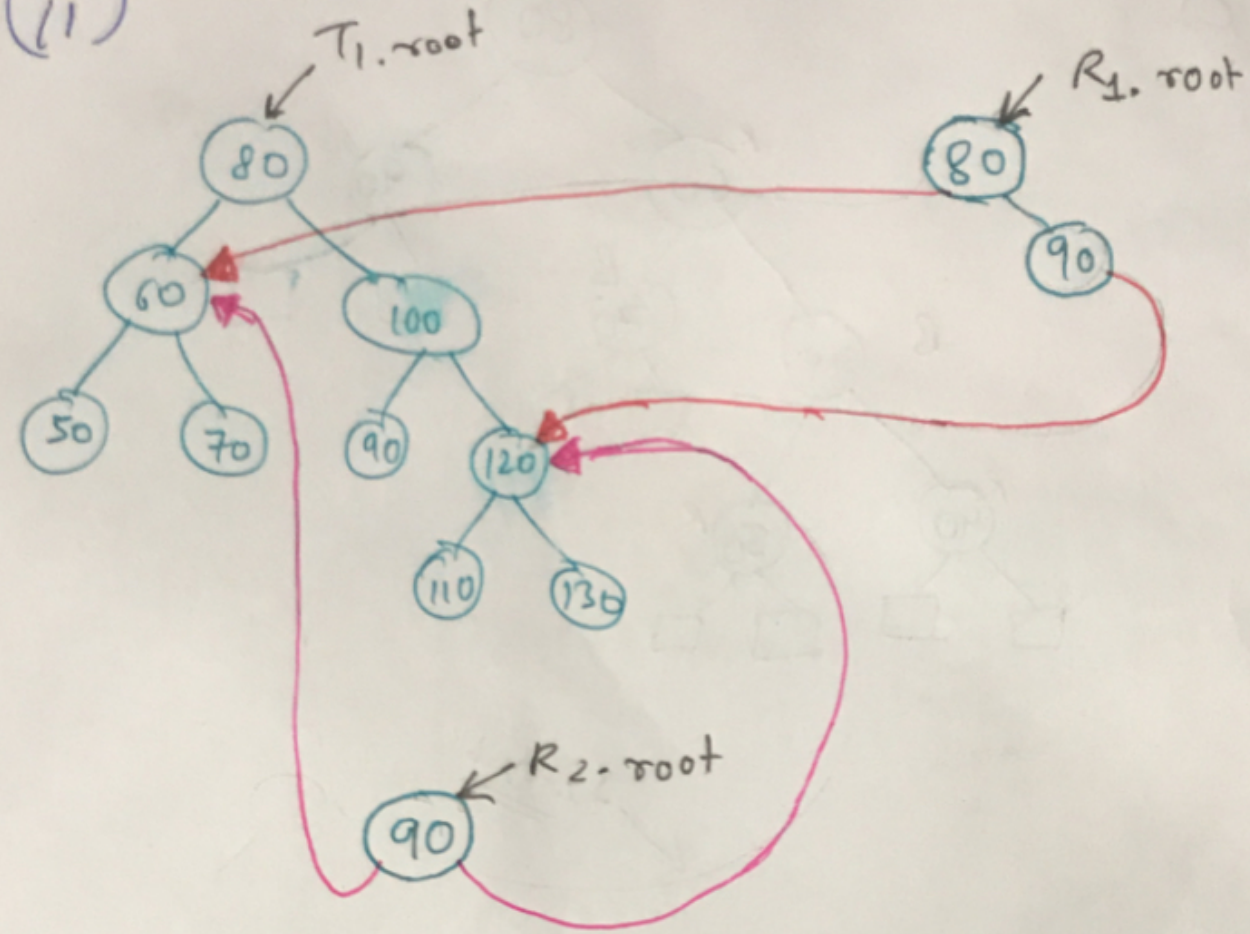
3b

(i)



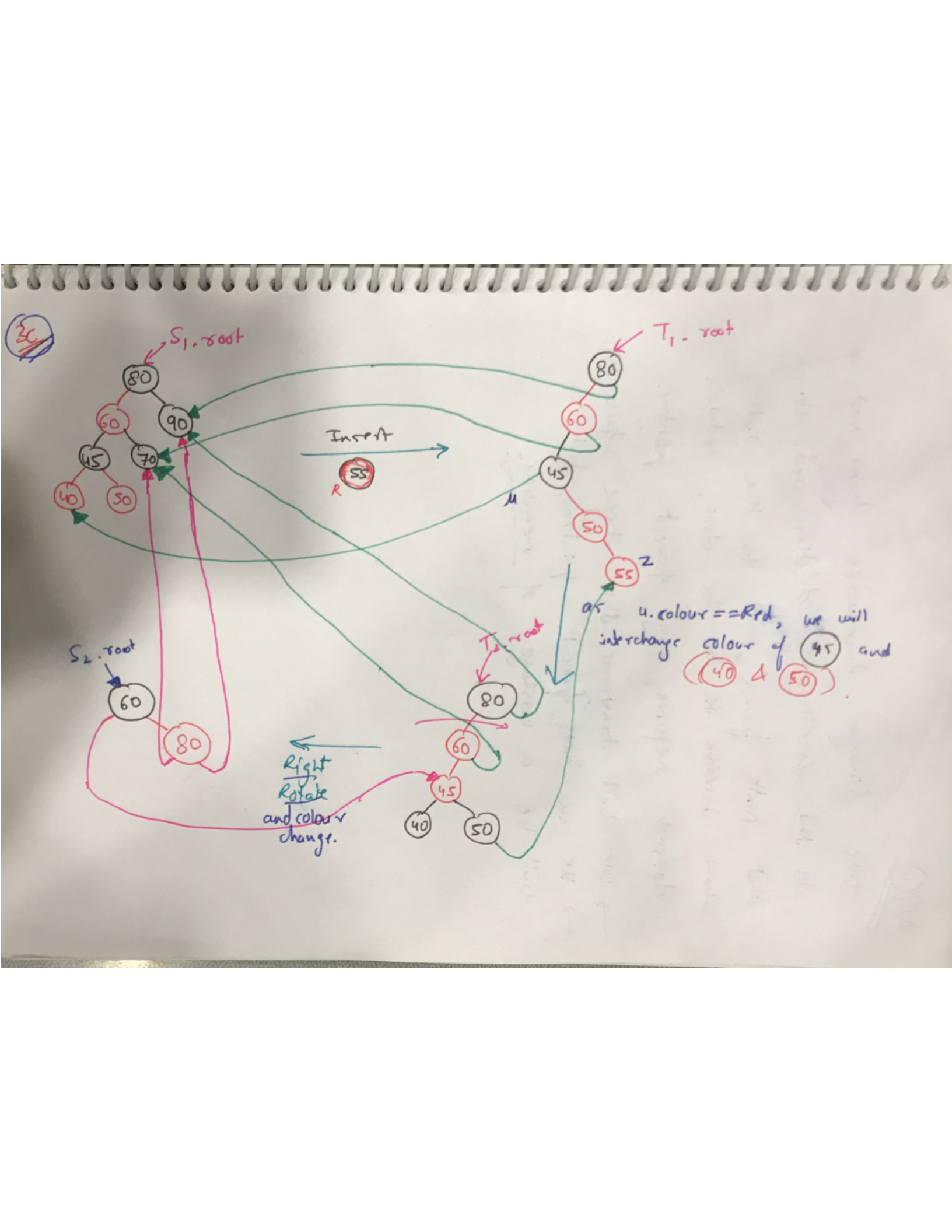
⊙ path upto 95 should be atleast made, and then we can link the leftover parts of the tree with T_1 and T_2.

3b
(ii)



(•) To create as few new nodes as possible, we are moving (90) in place of (100), this may only two new nodes have to created for R_1.

(•) after delting (80) from R_1, we have choosen (90) as the new node, as it will make sure, that we create minimum new nodes and links.


S_1.root
80
60
90
45
70
40
50
Insert
55
R
T_1.root
80
60
45
50
55
Z
u
ar
u.colour == Red, we will interchange colour of 45 and (40 & 50)
S_2.root
60
80
T_2.root
80
60
45
40
50
Right Rotate and colour change.

(4) Insert procedure:

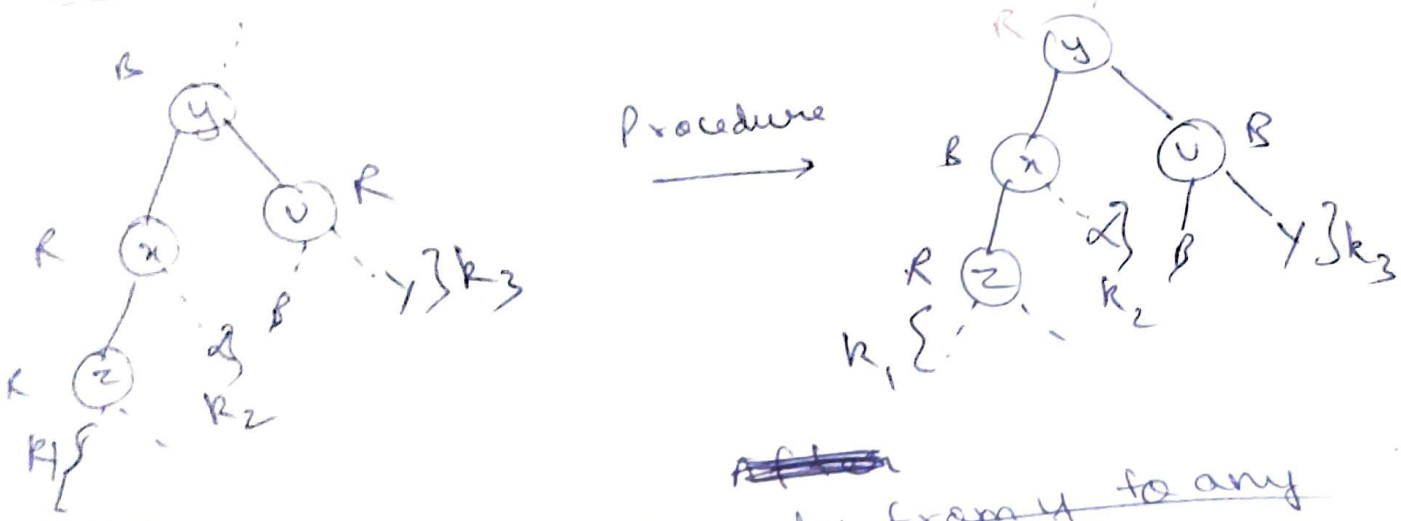
Insert (T, z)

Insert1 (T, z)

No change in these procedures is required as the inserted node is red which does not affect the black height of tree (i.e. T.bh)

Ifixup (T, z) // We will look at various cases.

uncle has red colour



~~After~~

~~Property → No. of black nodes from y to any descendant leaf is~~

~~after~~

~~B~~

If y is root → Case 1 (a)

(referring to figure on previous page)

Before

Black nodes $y-z\cdots = 1+k_1$

$y-\alpha\cdots = 1+k_2$

$y-\beta\cdots = 1+k_3$

After →

$y\cdots\alpha = 1+k_2$

$y-z\cdots = 1+k_1$

$y-\beta\cdots = 1+k_3$

But to preserve property 2, the colour of y is made black. So T.bh increases by 1.

Conclusion:

IfixUP(T,x) in Case 1

We only need to change the code in ^

if y ~~before~~ is the root.

① if (u.col == red) → Uncle is red

- - -
- -
- -

if(y == T.root) → y is root

y.col = black

T.bh = T.bh + 1

elif

- - - -

- - -

- -

}

Conclusion

We only need to change the code in Case 1 if y is the root.

① if (y.col == red)

if (y == T.root)
    y.col = black

elif

Delete procedure

z → node being deleted

y → node which was at the position in T, which got deleted

x → is the node that comes in place of y after deletion

Case → When y is red then R-B properties are still satisfied and T.bh does not change as a red node is removed from the position in T.

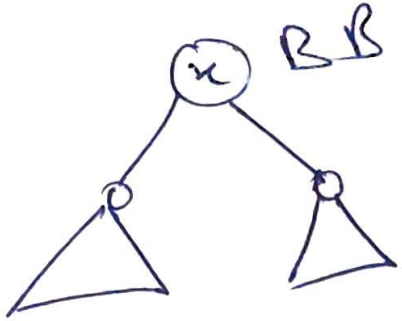
Delete (T, z)

→ No change is required in Delete(T, z) as a ~~[illegible]~~ if the black height of the tree changes we will get to know it while restoring the R.B properties as for the time being we have conserved the property 5, by denoting colour of x to Red-Black or Black-Black to maintain property 5 in case y was coloured black.

D fixUp(T, z)

Case-1(b) When $x == T.root$ and x is [black black]

→



As x is root we drop off a Black from x hence the black height of Tree decreases by 1.

$[T.bh = T.bh - 1]$

Pseudo code

Splited the code given in lecture notes.

```
Def fix VP(T, x).
    If (x.col == red)
        x.col = black
        return

    If (x.col == black) and (x == T.root)
        T.bh = T.bh - 1
        return

    .
    .
    .

}
```

Time complexity

We have just added O(1) operation [T.bh = T.bh - 1] if x.col == black and x == T.root which does not change the complexity.