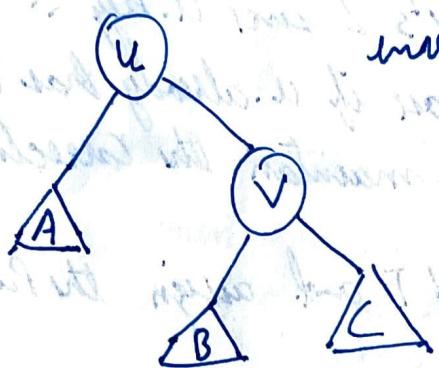


Q.1.

a)



invocation call.  $(T, u)$

(ii) Left Rotate  $(T, u)$ :

if  $u.\text{Right} \neq \text{NIL}$ :

temp =  $u.\text{Right}$

$u.\text{Right} = v$

$v.\text{Left} = \text{temp}$

else:

$u.\text{Right} = v$

if  $T.\text{Root} = v$ :

$v = T.\text{root}$

$v.P = u$

else:

iii.  $P = v.P$

$v.P = u$

Return  $T$ ;

Run-Time =  $O(1)$

The Run time is Constant time

There are only 2 operations happening that of Comparison & assignment so both of them will happen in constant time  $O(1)$ .

Correctness:

- ① The algorithm interchanges the parents of  $u$  &  $v$ 's & since  $u.key \leq v.key$ ,  $v$  will become the right child of  $u$ . In the case if  $u$  already has a right child it becomes the left child of  $v$  to maintain the correctness.
- ② We also take care of the case when  $v$  is root of  $T$  and assign the parent to both  $u$  &  $v$ .

As mentioned before that the algorithm runs in  $O(1)$  so the correctness have been justified.

c.

~~DELETE ROTATE ( $T, z$ )~~

~~$v = z.P.$~~

~~$v.left = z.left$~~

~~left Rotate ( $T, v$ )~~

~~Transplant ( $T, z$ )~~

~~Routine = Constant ( $O(1)$ )~~

~~DELETE ROTATE ( $T, z$ )~~

~~while deg ( $z$ ) > 1 :~~

~~Left Rotate ( $T, z.left$ )~~

~~Transplant ( $T, z$ )~~

Routine =  $O(m)$

Correctness.

- Base Case  $\rightarrow$  We need to check for all nodes smaller than  $2$  ad.  $2$ .
- $\rightarrow$  Transplant can delete a node only if there are less than  $2$  children.
- $\rightarrow$  So we keep pushing down until we have  $2$  children.
- $\rightarrow$  As soon as we reach this, the loop terminates & now know our algorithm is true.

d) Assumption

u. right  $\neq$  nil.

Rotate successor ( $T, u$ ):

min = u. right

while = min. left  $\neq$  nil:

left rotate ( $T, \text{min. left}$ )

Return u. right.

Correctness.

$\Rightarrow$  We are trying to find the successor of node  $u$ .

- As per the rule, it should lie in the sub-tree of  $u$ .right.
- As we have assumed that  $u$ .right  $\neq$  NIL, we know that  $u$ .right exists and is the successor to  $u$ .
- If  $u$ .right has no children, then we just RETURN  $u$ .right, but if child is found, then we find the smallest by moving up the left node. using Left ROTATE.

$\Rightarrow$  [The loop terminates, when all the left nodes are traversed]

- $v$ .right will now be the smallest element in the subtree.
- It will be the in-order successor of node  $u$ .
- $\therefore$  it will RETURN  $v$ .right

Runtime:-  $O(\log n)$

$n \rightarrow$  number of nodes in the sub-tree of u.right.

Since we have used a WHILE loop to traverse till the end of  $u$ .right the runtime will be  $O(\log n)$ .

## (e) ROTATE SEARCH Path ( $T, n$ )

if  $T.\text{root}. \text{key} = n$ :

    Return  $T$

— else if  $T.\text{root}. \text{key} < n$ :

    subnode = Rotate Search. ( $T.\text{root}. \text{right}. n$ )

— else if  $T.\text{root}. \text{key} > n$ :

    subnode = ROTATE SEARCH ( $T.\text{root}. \text{left}. n$ )

RETURN ROTATE ( $T.\text{subnode}. \text{root}$ )

Correctness:

- ① Base Case: → If the key of the root matches  $n$ , we return the tree.
- ② All the other cases we need to check. Root.key: Post this we recursively traverse the subtree where the root is equal to  $n$ .  
→ That tree is then returned & the ROTATE is applied on node. which will make the target node. the root of the subtree to which it is returned to finally.
- By doing this the target node will keep moving up until  $\text{root}. \text{key} = n$   
& ROTATE function returns a tree.
- Recursion calls are made until we have located the target node.
- Since there exist a target node, we can state that the algorithm must end.
- Hence, we have justified the correctness.

Runtime:

$O(n)$  when we have the entire tree one side.

$O(\log n)$  → This is for the balanced tree

Answer  $\rightarrow O(n) \rightarrow \text{worst case.}$

## Problem 2.

(a) FillTree( $T, v, A, \text{start}$ )

If  $v = \text{NIL}$

Return  $\text{start}$

Else.

$m = \text{FILLTREE}(T, v.\text{left}, A, \text{start})$

$v.\text{key} = A[m]$

$m+1$

$s = \text{FILLTREE}(T, v.\text{right}, (A, m))$

Return  $s$ .

Base Call.  $\rightarrow v = \text{NIL}$ . (i.e. Simply return  $s$ )

Proof by Induction

Algorithm is true for trees  $0$  to  $k-1$ .

For step  $k$ ,

$\rightarrow$  the Algo calls on the left subtree with size less than  $k$ , which is assumed correct.

So  $A[\text{start}, m-1]$  placed. &  $v.\text{key} = A[m]$ .

## (b) Transform Balanced. ( $T, N$ )

1. Start
2. BST  $T$  inputs having  $n$  nodes.
3. Traverse BST in order & store result in Array. (This array will have the in order traversal)
4. Make middle value as root node.
5. Recursively do the same for left & right subtrees.
6. ~~Cut the middle part of left subtree~~
7. Take the middle of left half & make it left child of the root node.
8. " " " " right " " " " right " " " "
9. Recursively store the items in  $T$ .
10. End.

Time Complexity =  $O(n)$

Correctness.

- Traverses the tree downwards to find left & right nodes on both sides.
- Array will store values of each towards the peak.
- Since the program terminates the algorithm is correct.

(c) IS equivalent ( $T, T', m$ ):

1. Array A = In Order Traversal ( $T$ ).
2. for i in A:  
    if Search ( $T', i$ ) = True;  
        continue  
    else.  
        Return False.
3. Return True.

Run Time =  $O(m \log n)$ .

Correctness :- The algorithm stores element from one tree  $T$ , iteratively checks each element with the other tree.

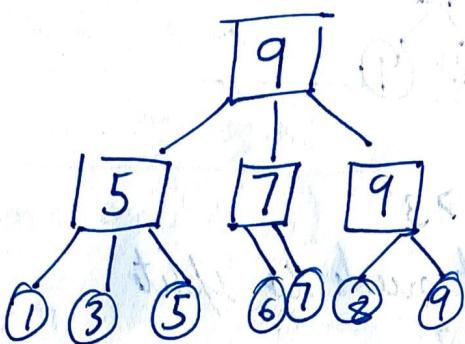
- Even if a single element is missing return false.
- Only if the loop finishes we return true.

# Problem 3 - 2-3 Tree

QUESTION

(a)  $m = 6$

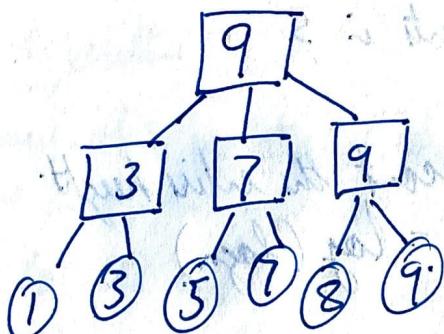
INSERT(6)



$\rightarrow 6$  should be inserted under 9 as  $6 > 5$ .

$\rightarrow 9$  has a total of 4 nodes so it need to split & give up a child

DELETE(6)

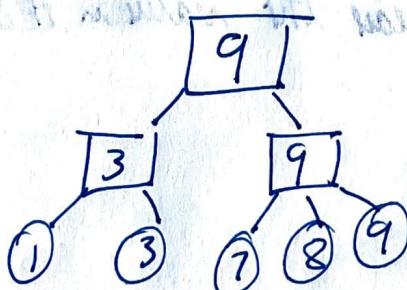


$\rightarrow 6$  is deleted then 7 is left with only 1 node.

$\rightarrow$  It checks with the other siblings with unbalanced degree 3 & takes a child to balance the tree.

(b)  $m = 5$

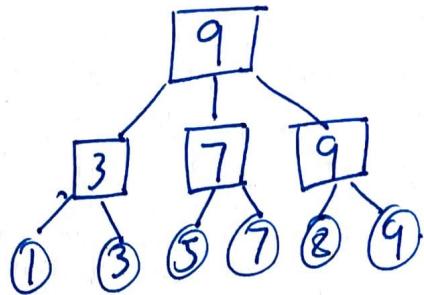
DELETE(5)



$\rightarrow 5$  is deleted in a skip step.

$\rightarrow$  The degree still remains greater than 1 so no change is needed.

Insert(5) :



→ 5 is put under 9 since  $5 > 3$ .

→ 9 will have 5 nodes so it needs to split.

→ Post split all nodes are  $\geq 2$ .

(c) We need to store all the occurrences of a Node as well as the number of nodes appeared in the augmented 2-3 tree.

⇒ Number of elements in tree

= No of distinct elements in S.

=  $\ln n$ .

Since we need to take in account the entire height.

=  $\log(\ln n)$ .

### Algorithm

1. → Insert elements from S by traversing down the tree & increment the count the leaf node based upon occurrence.

2. → Initialize the Array A.

3. → Traversing to each leaf I increase the value in A as many times as the count of that leaf node.

4. Return Array A.

## Running time

$$\begin{aligned} T &= \text{Insertion} + \text{Traversal of tree} \\ &= O(n \log \log n) + O(\log n \log \log n) \\ &= \boxed{O(n \log \log n)} \end{aligned}$$

$$\text{height} = h = \log(\log n)$$

$$\text{Insertion} = n \cdot h = n \log \log n$$

$$\text{Traversal} = \log(N) \cdot h$$

$$= \log n \times \log \log n$$

Since we are traversing the height, it will be  $\log n$  for each node.

## Correctness of Algorithm

- For every element of S, we travel the tree & insert it. If it is already present then we just increment the count.
- While adding to the array the elements are in sorted order. & we also append each element as many times as its count.
- Hence the array that we return after the two steps runs in  $(n \log \log n)$  time.
- As both loops terminate after returning the sorted array, hence our algorithm will also terminate.

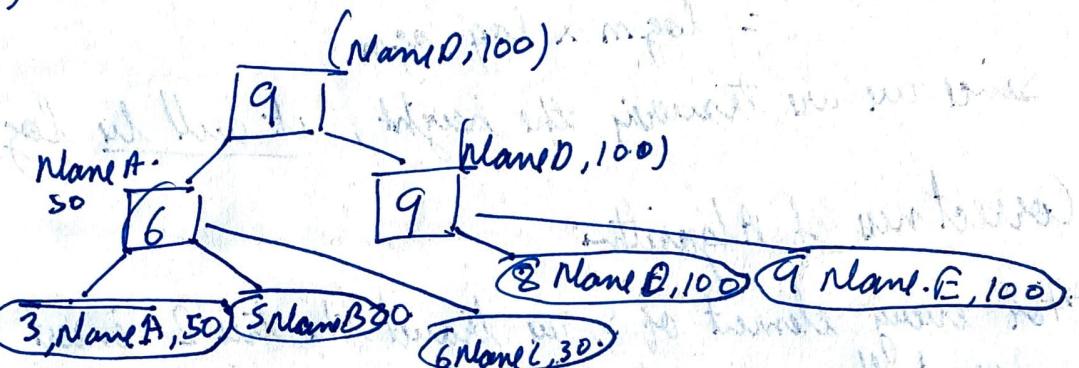
Next Page

d) As the question describes, an 2-3 augmented tree

- ① leaf nodes will contain the book entries [Book #, Name, Book Price].
- ② Internal nodes will contain the max number of books among the sub-tree as value is added with book name & price. among the sub tree = [Book Number] (Book name with max price)

Let the entries be.

(3, NameA, 50), (9, NameF, 80), (6, NameB, 30), (5, NameG, 20).  
(8, NameD, 100)



Above Mention 2-3 tree Shows

- leaf nodes contain records.
- Internal nodes contain the max book numbers.
- Finally the max priced book in its sub-tree

→ Indexing Operation → The indexing of the books is dependent upon an unique element that is Book Number.

$\Rightarrow$  Report Book Name with max price.

$\rightarrow$  Just need to return the book name sorted in the root node. or that is storing max priced Book Name. for the whole tree.

Relevant time =  $O(1)$  time.

$\Rightarrow$  Report price of a given book number.

Time =  $O(\log n)$ .

$\rightarrow$  Just like creating a book, we search for book by its book number.

$\rightarrow$  Report Book name with largest priced Book I.D in range (low, high)

(in  $O(\log n)$ ):

- Interval nodes having value less than low are ignored.

- As we traverse down the tree we keep track of max book with max price.

- All the values in siblings between (low, high) are compared & stored in our variable.

- When we traverse to the next interval node having value greater than high.

- Any other node with value  $\geq$  high are ignored.

- Since these nodes will have values which are greater than a value which is greater than high.

( $low \dots \dots \text{high} < y \rightarrow$  2nd interval nodes  $\{x\}$  ignore.)

book traversing down.

## ~~Deletion Operation~~

⇒ Merging of operations

→ Create a book in  $O(\log n)$

Compare the book numbers & search for an interval node that can store this entry.

This takes  $O(\log n)$  time as height of tree =  $\log n$ . i.e. travel down.

The max book number & max priced book values are updated after the new entry to compare its values with its sibling under same parent.

All of the data are completed under  $\log n$  time.

∴ The creation of book is  $O(\log n)$  time justified. (similar to INSERT)

⇒ Closing a book in  $\log n$ .

→ We first search the book with the book number which takes  $(\log n)$  time.

→ After finding the leaf, we delete that leaf node & update the values of its parent interval nodes.

∴ Justification remains the same for opening the book in time  $(\log n)$

⇒ Add Subtract price of a given book.

→ Again we need to locate the book with book numbers, taking  $O(\log n)$  time & then we update its book price value by adding / subtracting given value.

→ After this, the price is compared with sibling & we update the max priced value of its parent's interval node.

Time =  $O(\log n)$

If we keep traversing down this path we will reach  $ed$  & return the stored value.

$$\text{Runtime} = O \log n + \text{Comparison } O(1)$$
$$\boxed{O(\log n)}.$$

1. START
2. Initialize a new head\_node= NULL and prev\_node= NULL, head=NULL
3. Now, recursively travel the right sub-tree of the root to reach the right most node. Name this as temp.
4. Now, make rightNode as temp->right and leftNode as temp->left.
5. Check if the head\_node is NULL. If it is, then head\_node= temp;  

```
head=head_node;
temp.right= NULL;
prev_node= head_node;
```
- else  

```
prev_node->right = temp;
temp->right= NULL;
prev_node= temp;
```
6. Now, recursively do this for left sub-tree of the root.
7. head contains the left transformed tree.
8. STOP

Time Complexity of the above algorithm will be  $O(n)$  since we are traversing the tree and visiting every node.

Correctness of algo

- Algo traverses the tree to find every leaf in node
- Finally it reduces to 2 elements
- Loop ends