Name : Seetharaman Krishnamoorthy
ID : N13274420

CS6033 - Assignment 4
NetID : SK6599

1) - We construct a Red-Black Tree for maintaining P.

- We call RB-INSERT(T,x) and RB-DELETE(T,x) when an element : is added or removed.

- The nodes also store their size in addition to key.

- So the function LEFT-ROTATE and RIGHT-ROTATE will maintain size correctly using

$$y.size = x.size$$

$$x.size = x.leftsize + x.rightsize + 1$$

- The pseudocode to calculate mean and median are as follows,

GET_MEAN (P) :

```
root = ROOT(P)
Treesize = root.size
Totalsum = CALCULATE_SUM (root)
mean = Totalsum / Treesize
return mean.
```

CALCULATE_SUM (x) :

```
if x == NULL
    return 0
else
    return (x.value + CALCULATE_SUM (x.left) +
                      CALCULATE-SUM(x.right))
```

GET_MEDIAN (P) :

```
Treesize = ROOT (P).size
if Treesize % 2 == 0
    median = [ Treesize/2 + (Treesize/2 + 1) ] / 2
```

else

median = Treesize/2 + 1

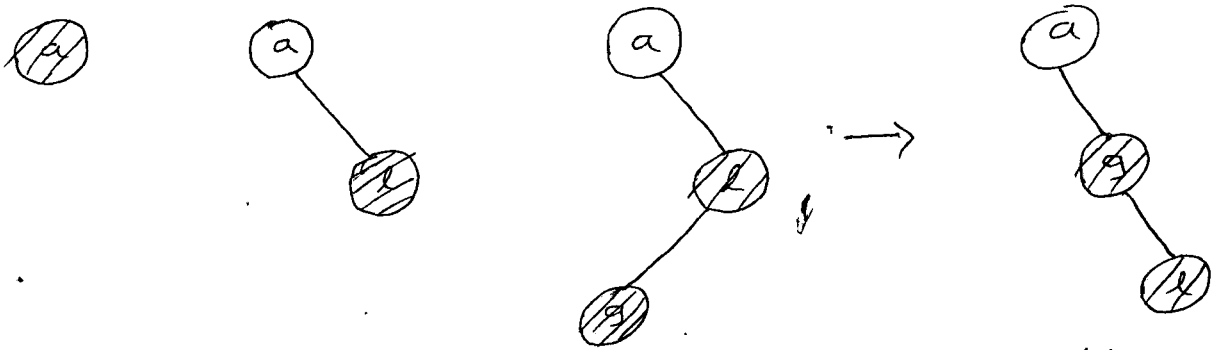
return OS-SELECT (CP, median).valve.

- All these function run in O(logn) time as tree is traversed.

2) - Insert a, l, g, o, r, i, t, h, m, s into R-B tree

- we represent ◎ as red node and ○ as black node.

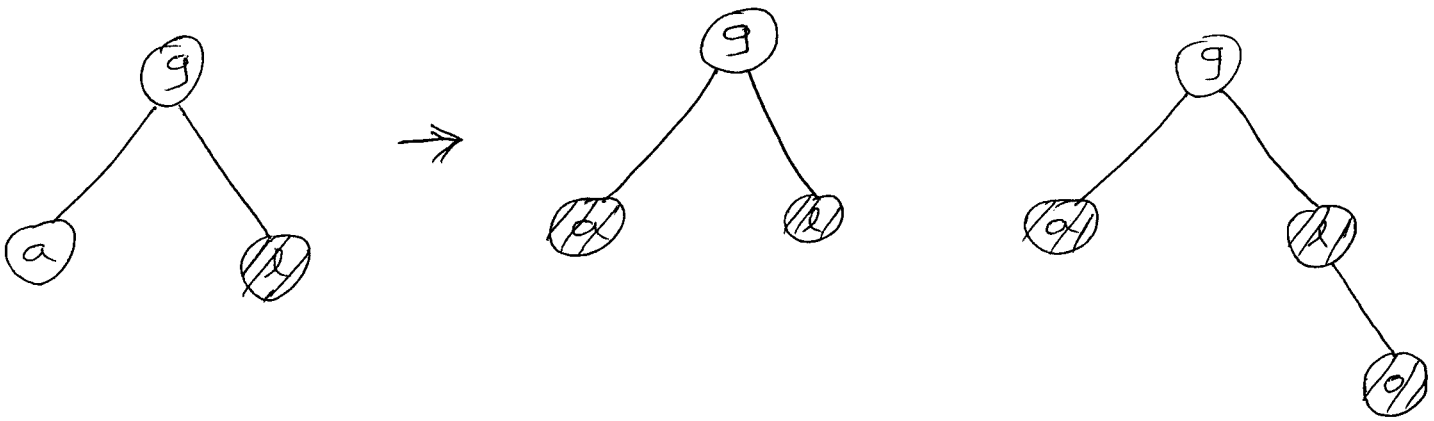
insert: a l g




Double red violation case 2 violation

Double red case 3 violation.

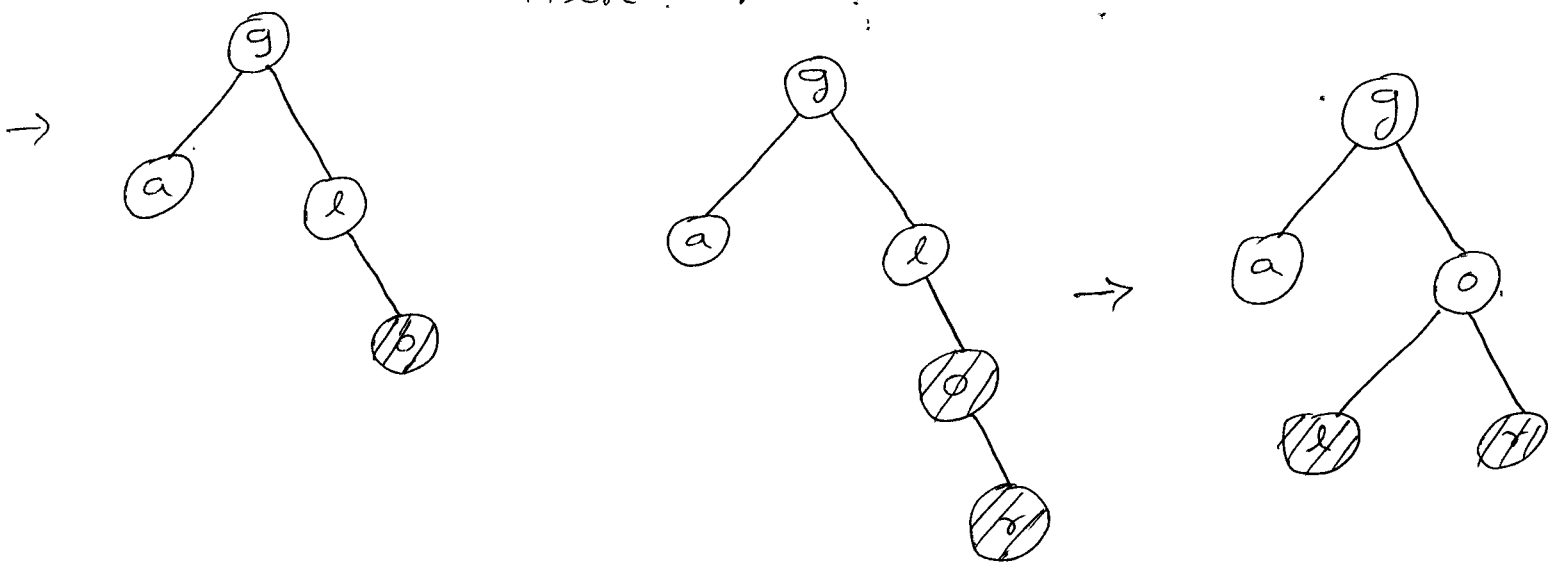
insert : o




Double red violation

case 1

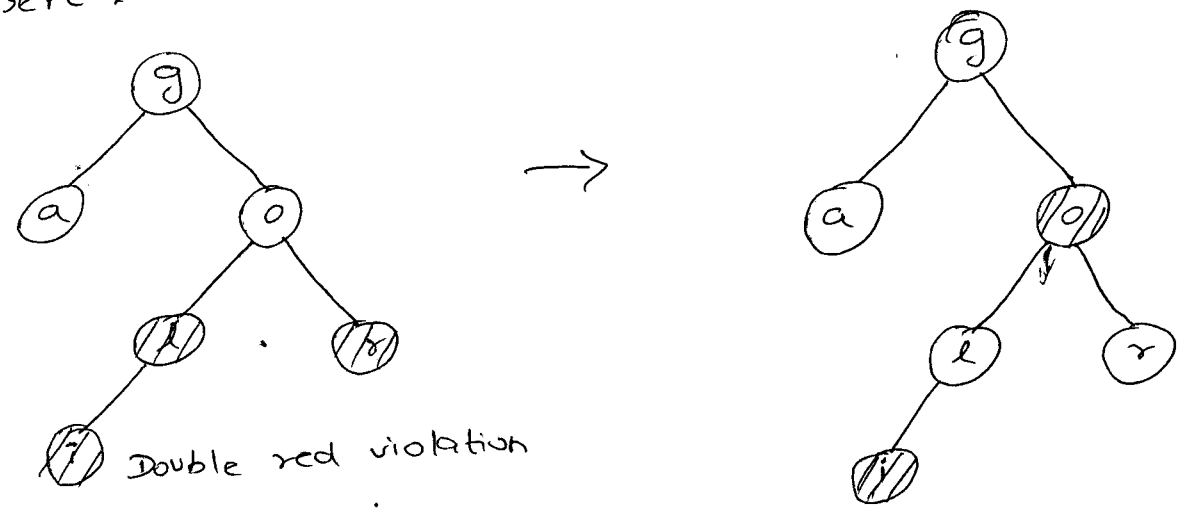
insert : r




Double red violation

case 3

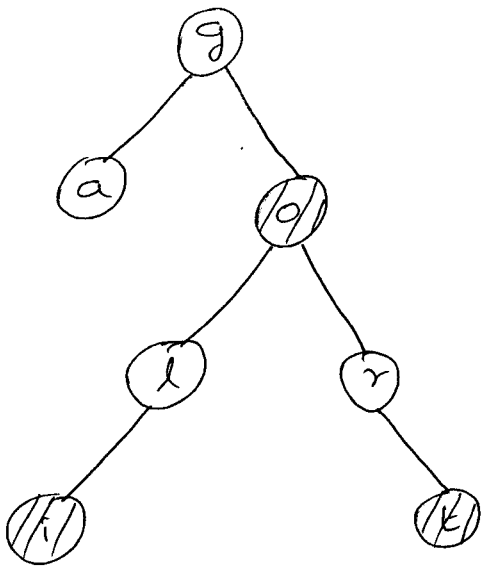
insert : i



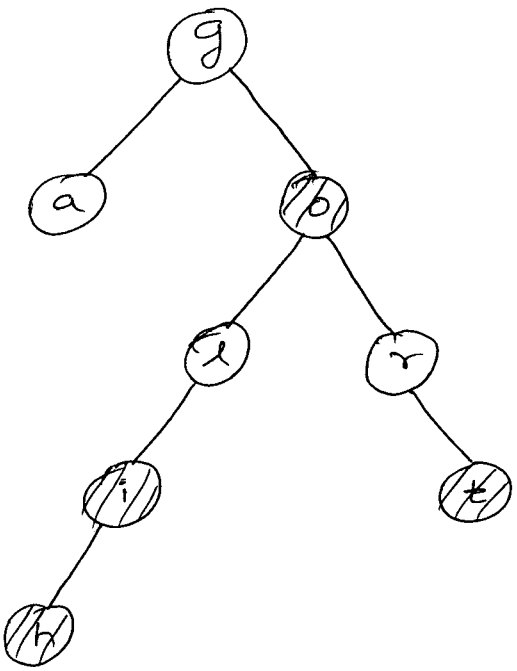

Double red violation

case 1

insert : t



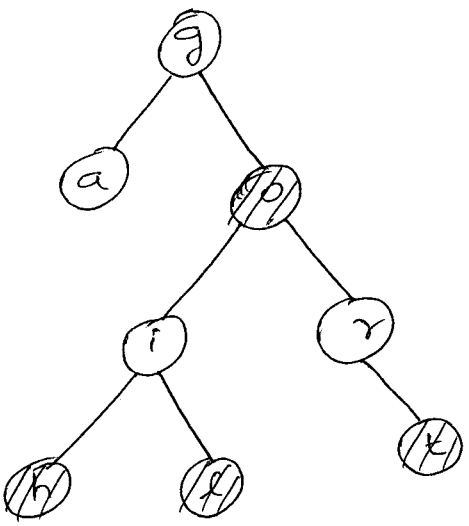

insert : h



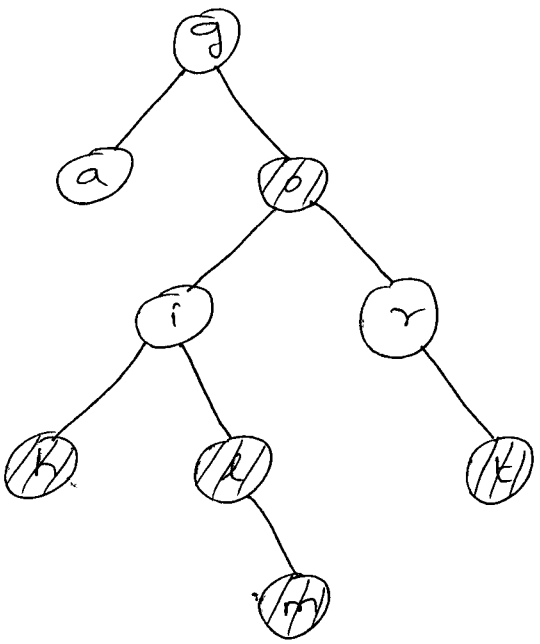

Double red violation

case 3

→

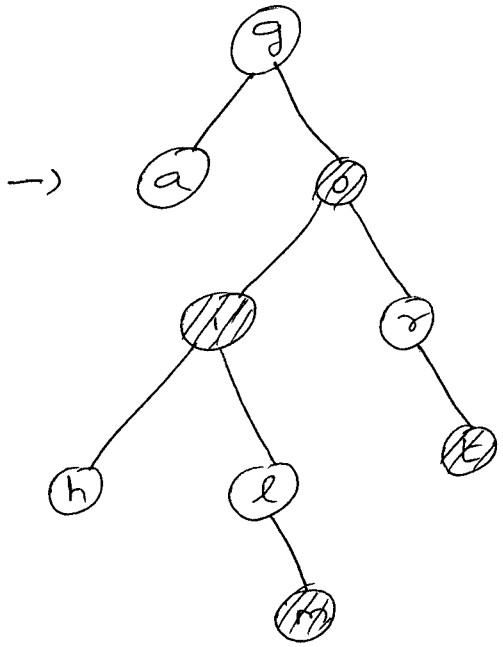



insert : m



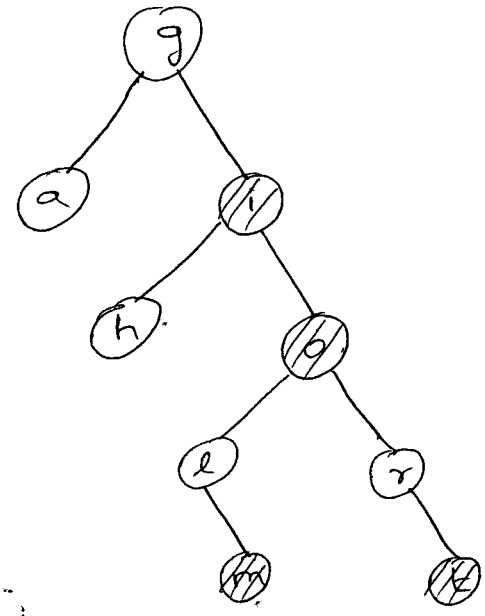

Double red violation

case 1




Double red violation

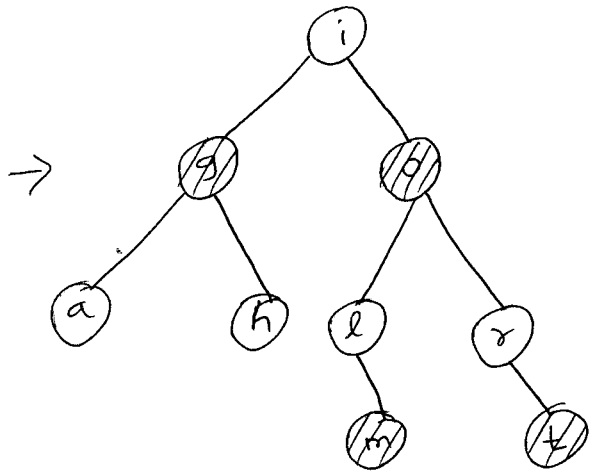
case 2

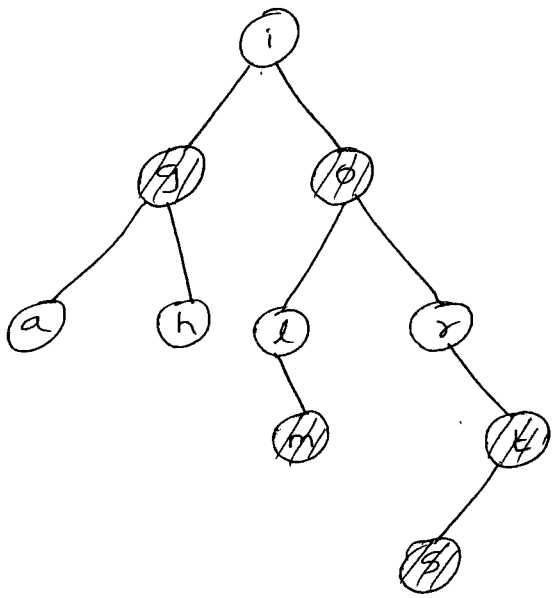



Double red violation

case 3.

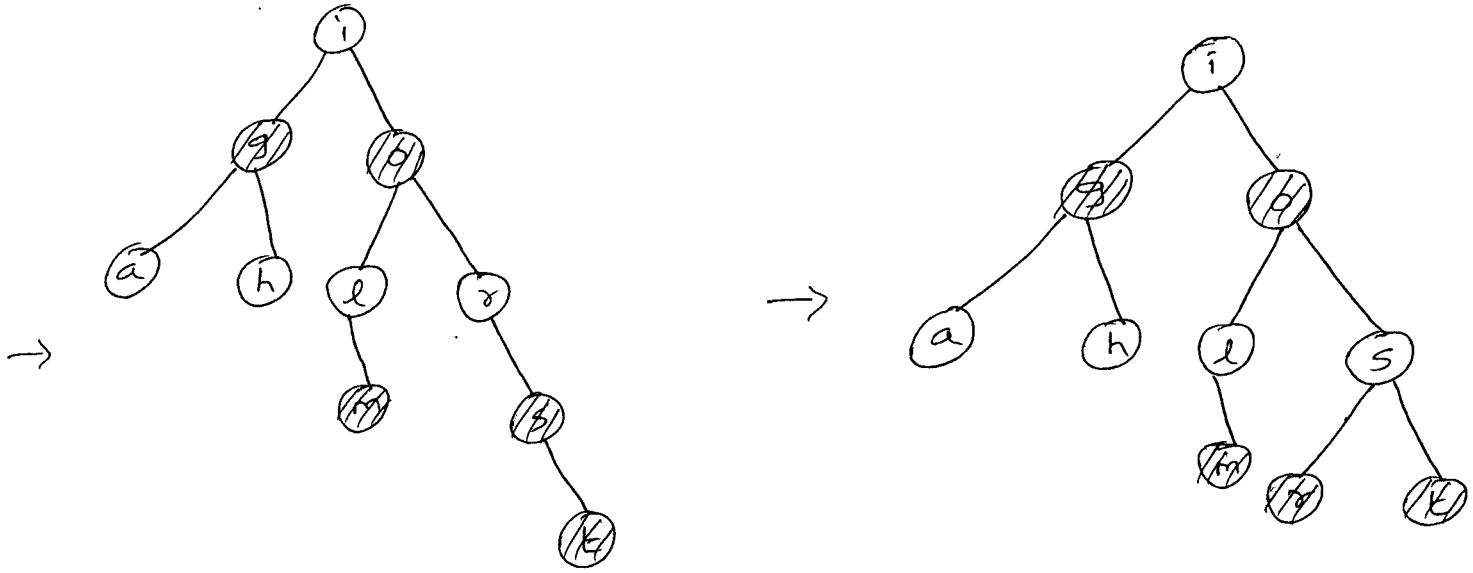
insert : 5




Double red violation

case 2

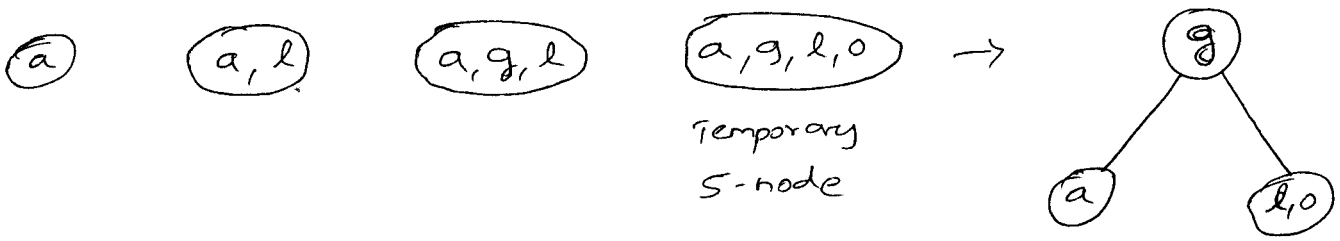


Double red violation :

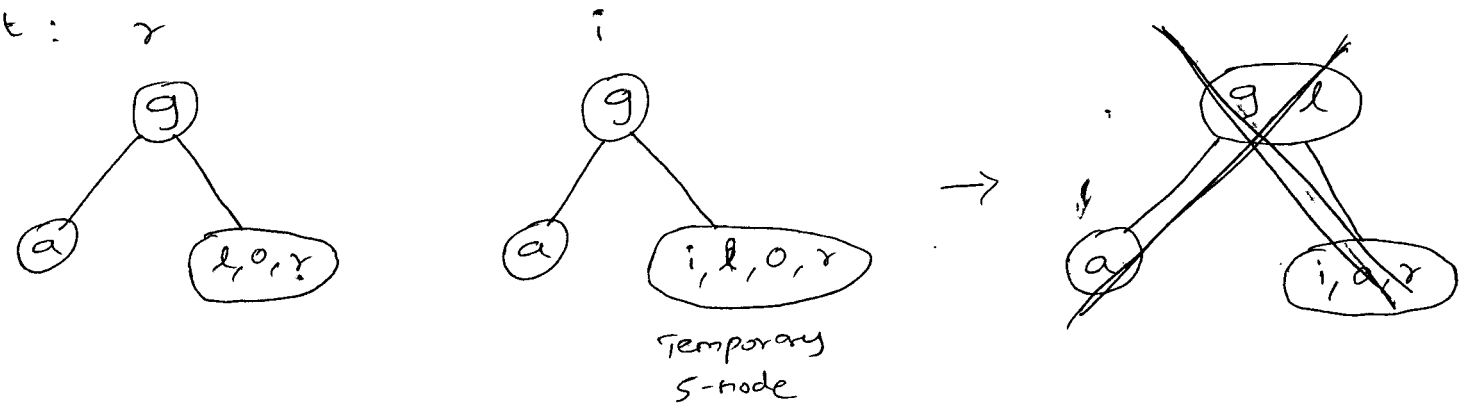
case 3

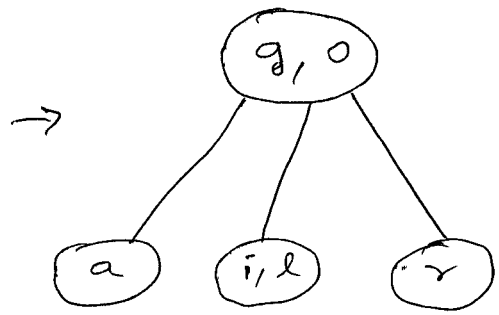
3) - Insert a, l, g, o, r, i, t, h, m, s into 2-3-4 tree

insert : a l g o

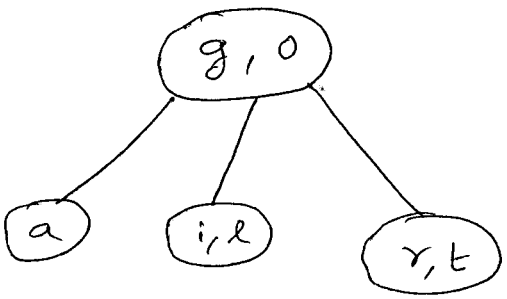



insert : r i

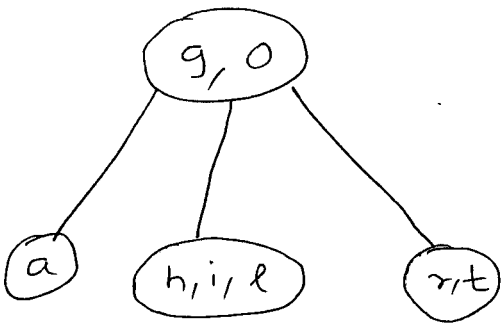




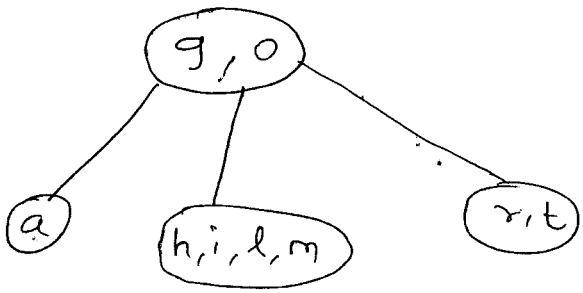

insert : t



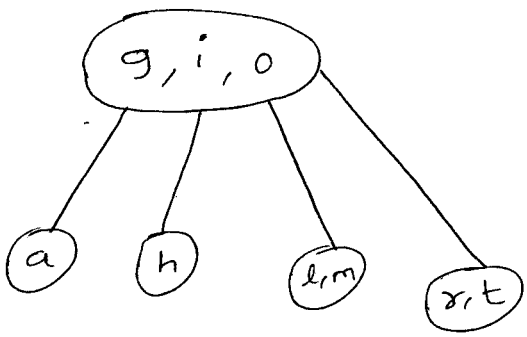

insert : h



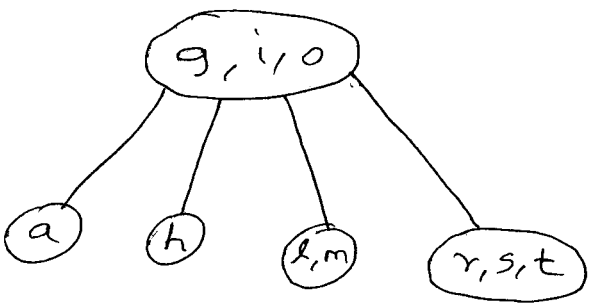

insert : m




Temporary
5-node.




insert : s



4) - The height of RB Tree is h.

max. internal nodes $= \sum_{i=0}^{h} 2^i$

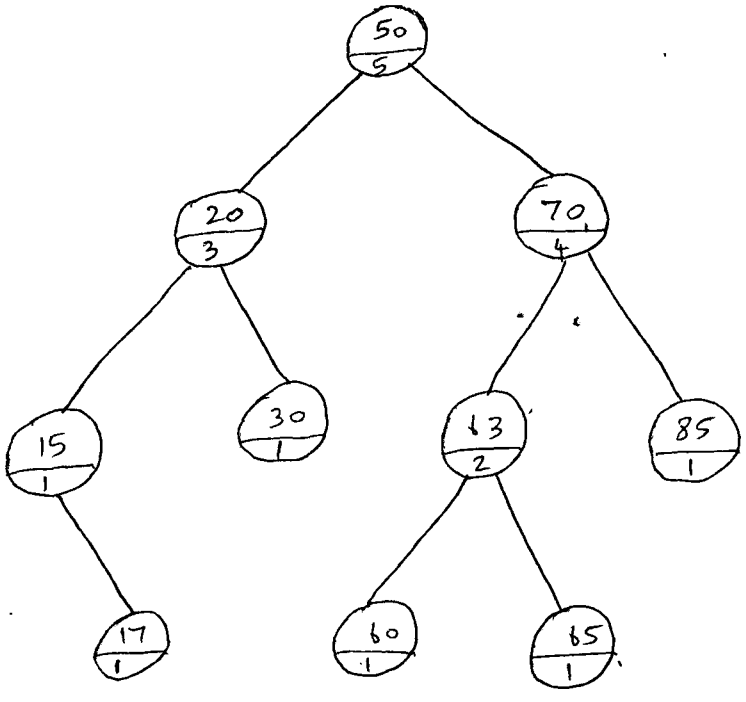
$\therefore$ max. no. of internal nodes in RB Tree $= 2^h - 1$

- when there is a RB Tree of height h, then there should be atleast 1 node in each level

$\therefore$ min no. of internal nodes in RB Tree $= h$

5) - Lets store the rank of each node along with its key.

- Lets consider the sample tree




Insertion :

- when we insert a node in Red-Black tree, there are two phases.
- In phase 1, we insert the node at correct position in tree as leaf.
- Then we do rotations to maintain the property.

## Insertion

### Phase 1 :

- Here we should modify the rank of each node in traversal when the new node takes left path.

- In RB-INSERT(T,z) while loop

```
while x ≠ T.nil
    y = x
    if z.key < x.key
        x.rank = x.rank + 1      → update rank.
        x = x.left
    else
        x = x.right
```

### Phase 2 :

- Here the rank changes when we do a right and left rotation.

- The method is mostly similar and last 2 lines are updated in both methods.

LEFT-ROTATE (T, x)

- In this method we just add.

$$y.rank = y.rank + x.rank.$$

- This is because x gets added to y's left which increases its rank.

RIGHT-ROTATE (T, x)

- In this method we just add

$$y.rank = y.rank - x.rank.$$

- This is because x goes from y's left to right.

## OS-SELECT(x, i)

```
r = x.rank
if i == r
    return x
elseif i < r
    return OS-SELECT(x.left, i)
else
    return OS-SELECT(x.right, i-r)
```

## OS-RANK(T, x)

```
r = x.rank
y = x
while y ≠ T.root
    if y == y.p.right
        r = r + y.p.rank
    y = y.p
return r
```

Running Time:

Insertion = $\Theta(\log n)$

OS-SELECT = $\Theta(\log n)$

OS-RANK = $\Theta(\log n)$

- Storing the rank would be an efficient method because in insertion not all nodes rank are updated, only the left traversal affects the rank of node.
- And it is easy to calculate rank using OS-RANK.

6) a) INSERT (T, date, no.of people)

- for this operation we will use the Red-Black tree as our data structure.

- Each node has date as key and also has size and no.of tickets sold in the node.

INSERT (T, D, n) :

for D in list:

RB-INSERT (T, D, n)

- Each node has three properties. Say x is the node, then

x.key = D

x.size = size of tree with x as root.

x.tickets = n

- The size of node is maintained when performing rotations.

LEFT-ROTATE (T, x)

$$y.size = x.size$$

$$x.size = x.leftsize + x.right.size + 1$$

RIGHT-ROTATE (T, x)

$$y.size = x.size$$

$$x.size = x.left.size + x.right.size + 1$$

Running time

- The insert operation takes $O(\log n)$ time as it traverse the tree to insert the node.

b) TOTAL (T, date 1, date 2)

- Here we are given 2 dates D1 and D2.
- The nodes contain the date along with size and no. of tickets.

~~Total~~

TOTAL (T, d1, d2):

$r_1$ = OS-RANK (T, d1)

$r_2$ = OS-RANK (T, d2)

for i = $r_1$ to $r_2$

total = total + OS-SELECT (T, i)

return total

- Here we find the 2 dates and we iterate through each date between those two and keep track of the total tickets.

Running time :-

- The total running time of the TOTAL() method will atmost be O(logn) since at worst case the two dates may be the left and right most child in the tree. [2 logn which is O(logn)]

c) DELETE (T, date) :

- Here we are given a date d and it should be deleted from the tree.
- After deletion, the nodes should be updated properly to maintain R-B property and also correct size.

c) - The DELETE operation takes a date D as input and removes the node from Tree.

- The size of subsequent node should be maintained when deleting the node.

DELETE (T, date):

R-B-DELETE (T, d)

- Here the LEFT_ROTATE (T, x) and RIGHT-ROTATE (T, x) also updates the size similar to the one in INSERT method.

- The running time of this method is O(logn) as deleting a node requires tree traversal and also updating the size of subtrees.