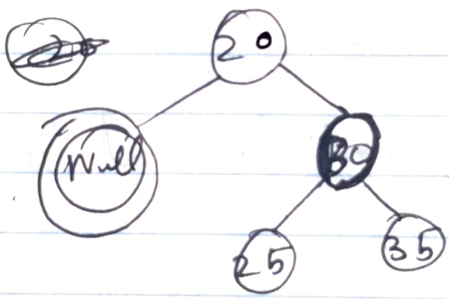
Quiz 12:

1) Step 1:

Element to be deleted is 10 and it is a leaf node.
We need to apply the delete algorithm for deleting a leaf node V and replacing it with a child 'u'

Step 2: Here V=10 u=NULL
we delete the leaf node 'v' which is 10



Since v is a leaf node, u=null
Color the node u as double black

According to 3.2), we should do the following while u is double black:

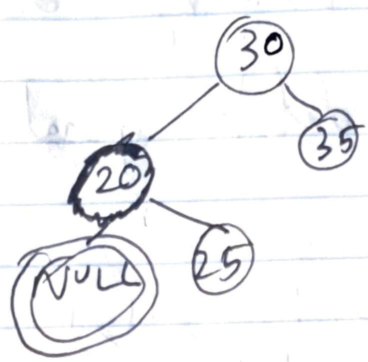
Step 3
Sibling is red, so we perform a rotation to move old sibling up, recolor the old sibling and parent

The new sibling will be always black.

We are in case (ii) of (3.2 c) of delete algorithm

According to case (ii).

ii) We left rotate parent p.



NULL is still double black

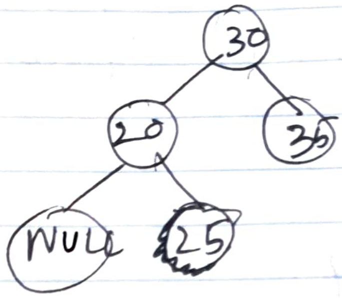
We continue.

Step 5:

Our ~~ta~~ tree now satisfies another usecase in our algorithm. (3.2 b)

Our sibling is black and both its children are black, do recolor and recurse for parent if parent is black.

Applying the above steps



Now u(NULL) is no longer doubleblack.

We have completed delete operation of v=10

---

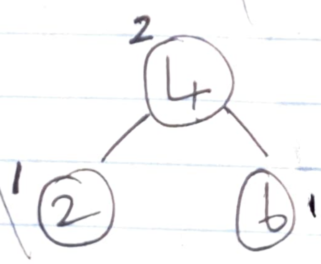
2) 4, 2, 6, 4, 5, 11, 3, 7, 8, 9, 10, 11, 1

~~4, 2, 6, 4, 11, 3, 7, 8, 9, 10, 11, 1~~

AVL Tree

~~AL~~ A) Insert 4

Insert 2 and 6



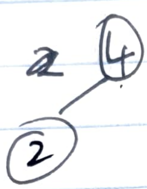
~~Insert 4~~

2) ~~4, 2, 6 3,~~

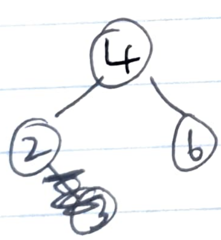
4, 2, 6, 3, 5, 11, ~~8~~, 7, 8, 9, 10, ~~4~~ 13, 1

Insert 4

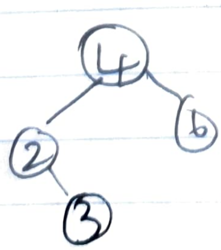
Insert 2



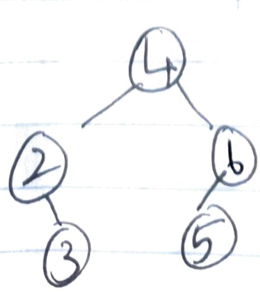
Insert 6



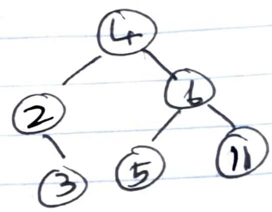
Insert 3



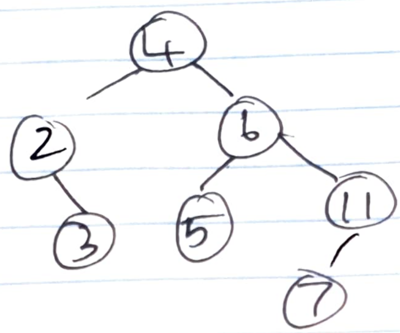
Insert 5



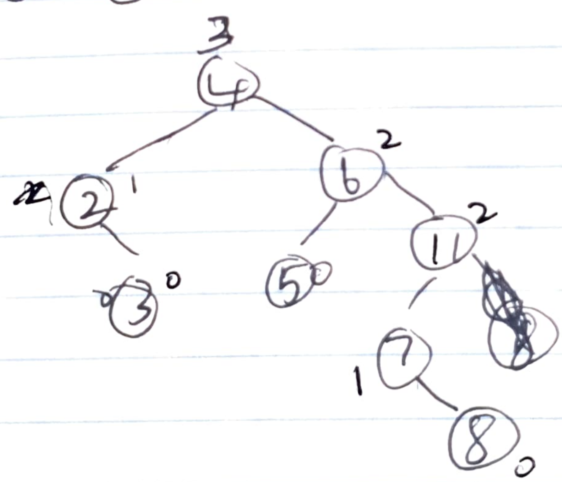
Insert 11



Insert 7

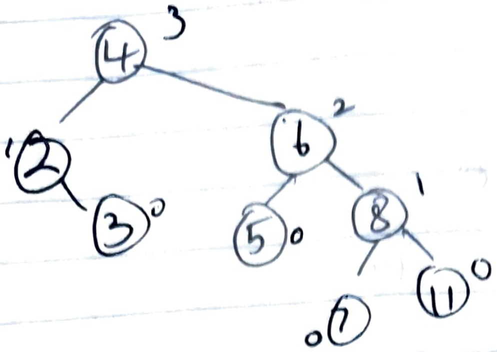


Insert 8



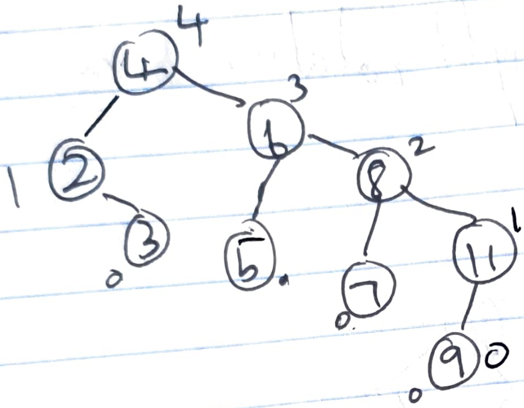
Height unbalanced

Double rotate right 8



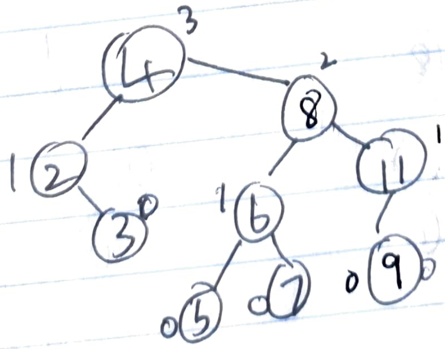
Now height balance is restored.

Insert 9

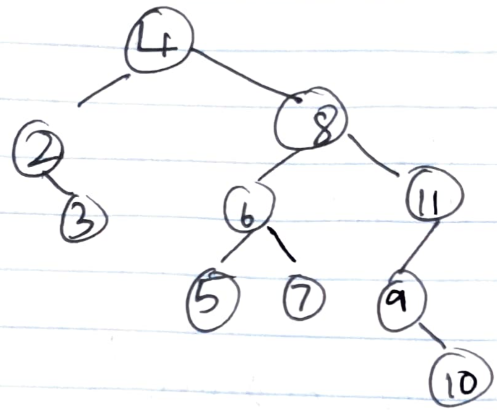


Right tree unbalanced.

Single Rotate left on 8

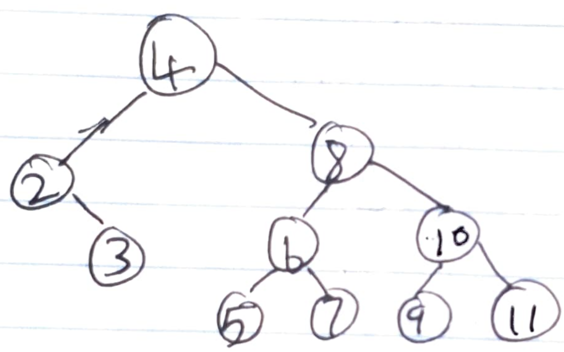


Insert 10

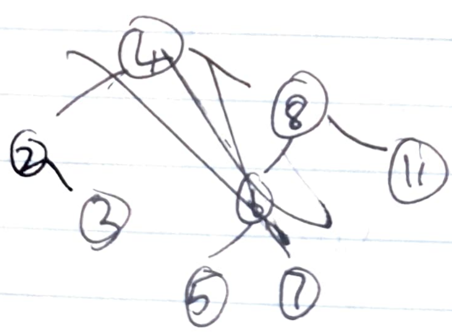


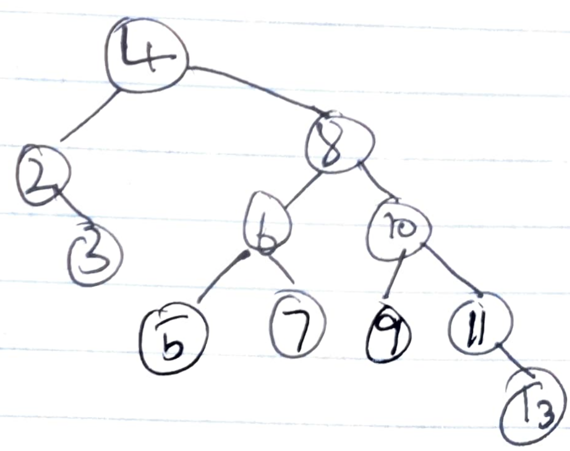
Tree is unbalanced.

Do a single right rotation. on 10



Insert 13

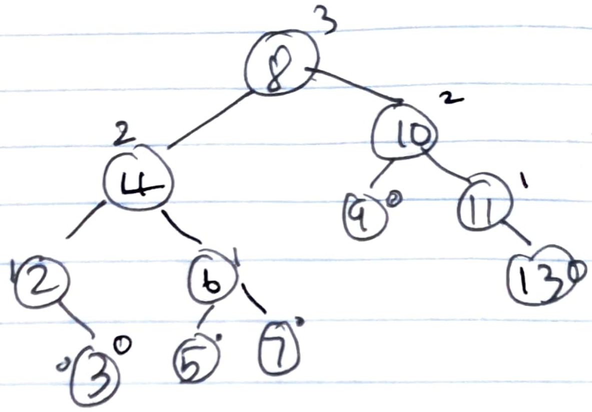




Height unbalanced again

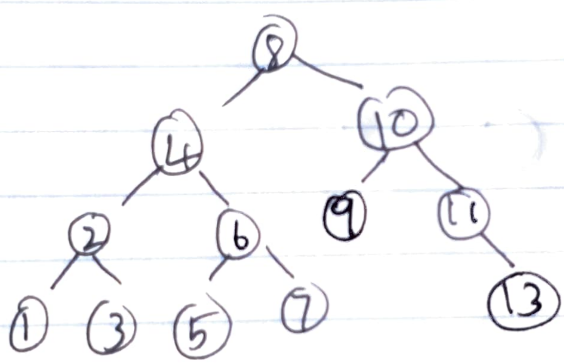
Do a single left rotation on 8

Shift it to root



Now ~~height~~ ~~At~~ height balance is ~~p~~ correct.

Insert 1



No balance issues found.

AVL tree is complete

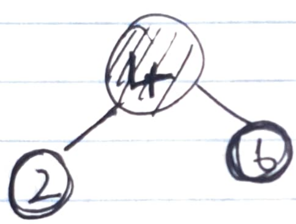
B) 4, 2, 6, 3, 5, 11, 7, 8, 9, 10, 13, 1

→ black node   → red node.

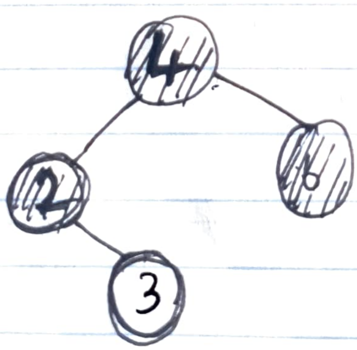
Insert 4

Insert 2

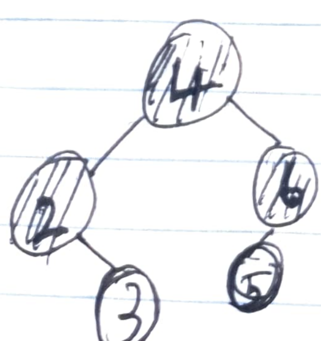
Insert 6



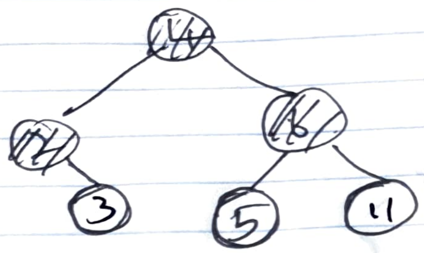
Insert 3.



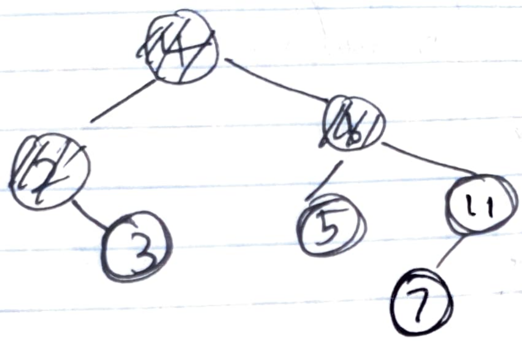
Insert 5.



Insert 11

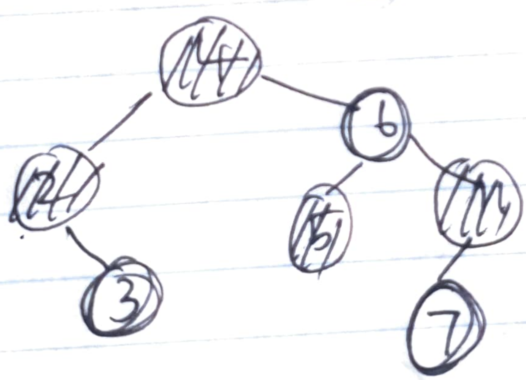


Insert 7



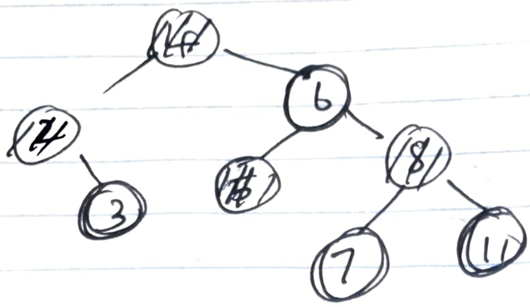
~~No.~~ ~~leaf~~ Child and parent cannot both be ~~red~~ red.

Recolor

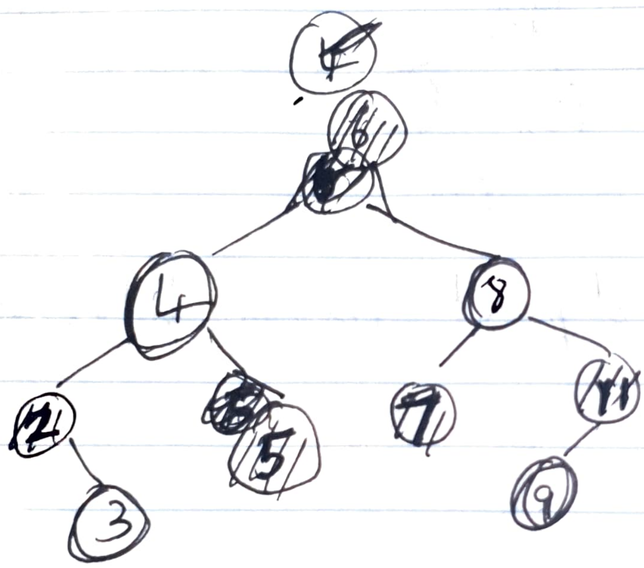


Insert 8

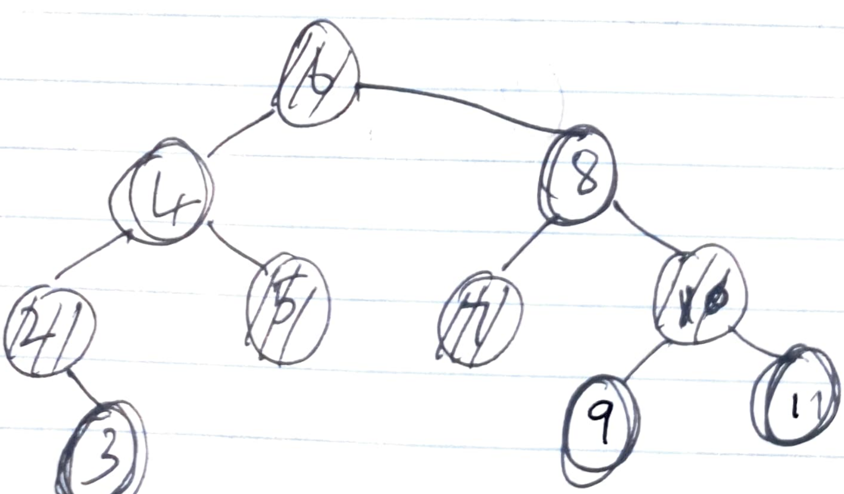
Recolor after inserting 8



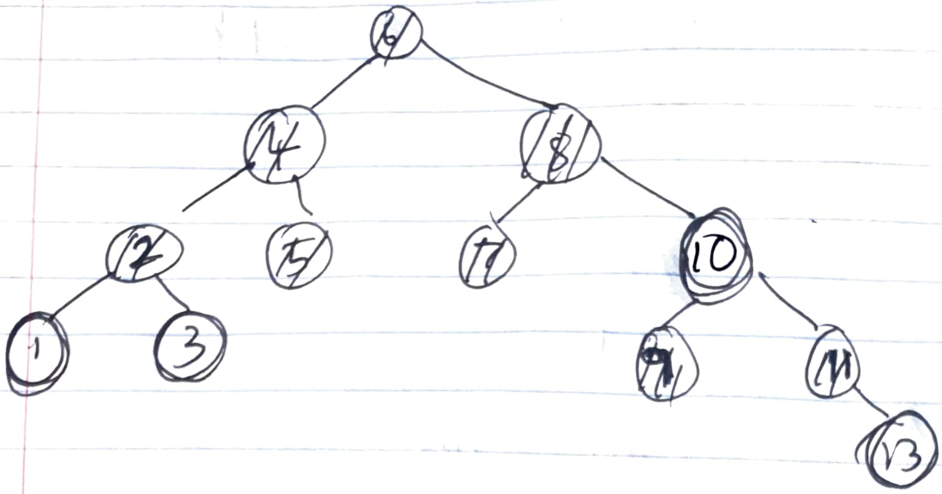
Insert 9, ~~10~~ and rebalance



Insert 10



Insert 13 and 1



RedBlack Tree complete.

C)

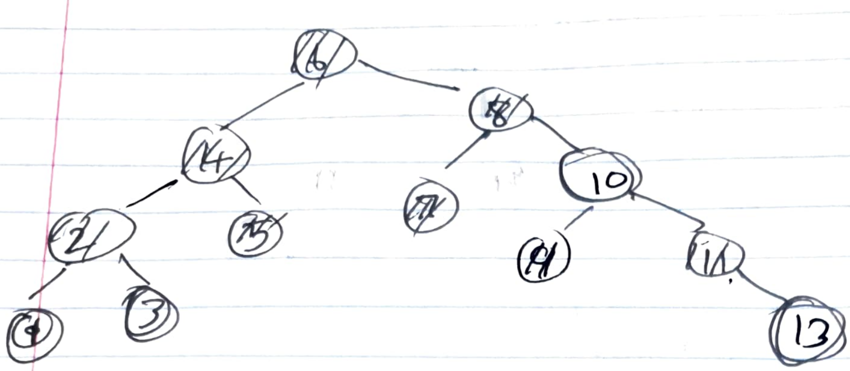
i) Height of AVL tree is 3

Height of Red-Black tree is 4

D) For A and B, time complexity for search insert and delete is log(N) in worst case.

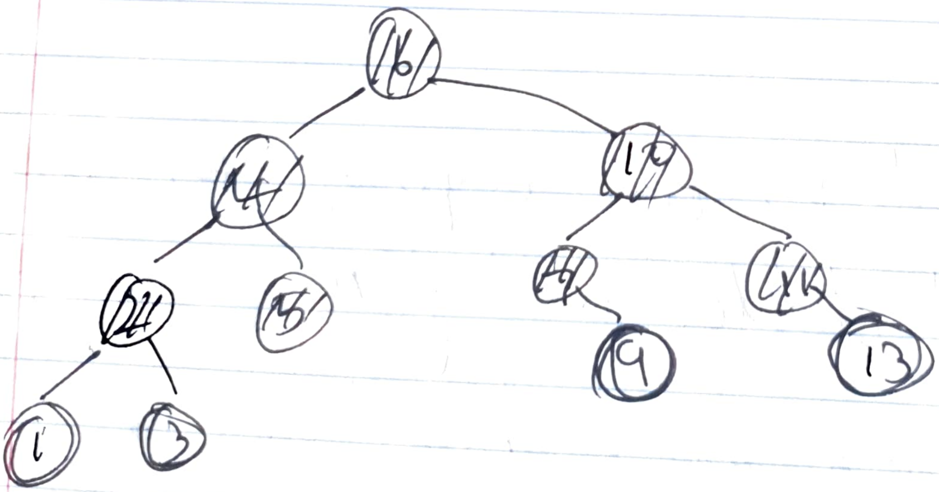
This is because ~~skewed~~ skewed trees are not possible even in worst case due to the height bala nature of both R-B and AVL tree compared to BSTs and other binary trees.

3) R-B tree   O - Red   → Black.



Found 8, Pick largest node of left subtree and swap and delete 8.

Rotate tree and recolor



8 deleted.

B) time complexity of deletion is ~~O(N)~~ O(log N) worst for both AVL and R-B tree.