

# Advanced Data Structure and Algorithm

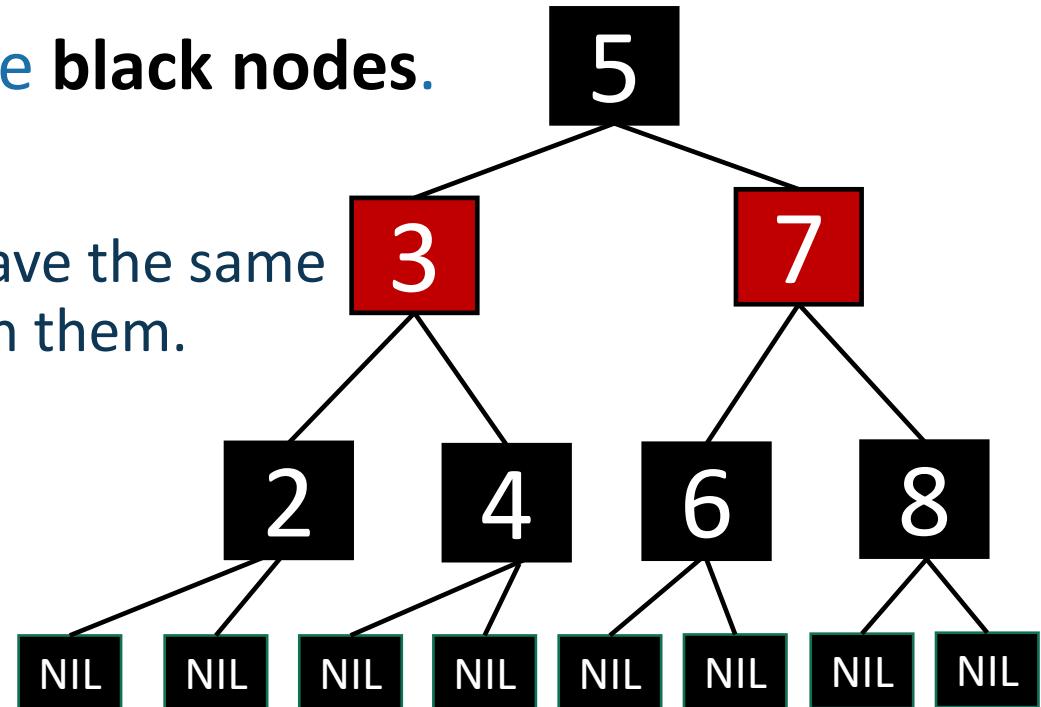
Red-Black Trees

Deletion

# Red-Black Trees

obey the following rules (which are a proxy for balance)

- Every node is colored **red** or **black**.
- The root node is a **black node**.
- NIL children count as **black nodes**.
- Children of a **red node** are **black nodes**.
- For all nodes x:
  - all paths from x to NIL's have the same number of **black nodes** on them.



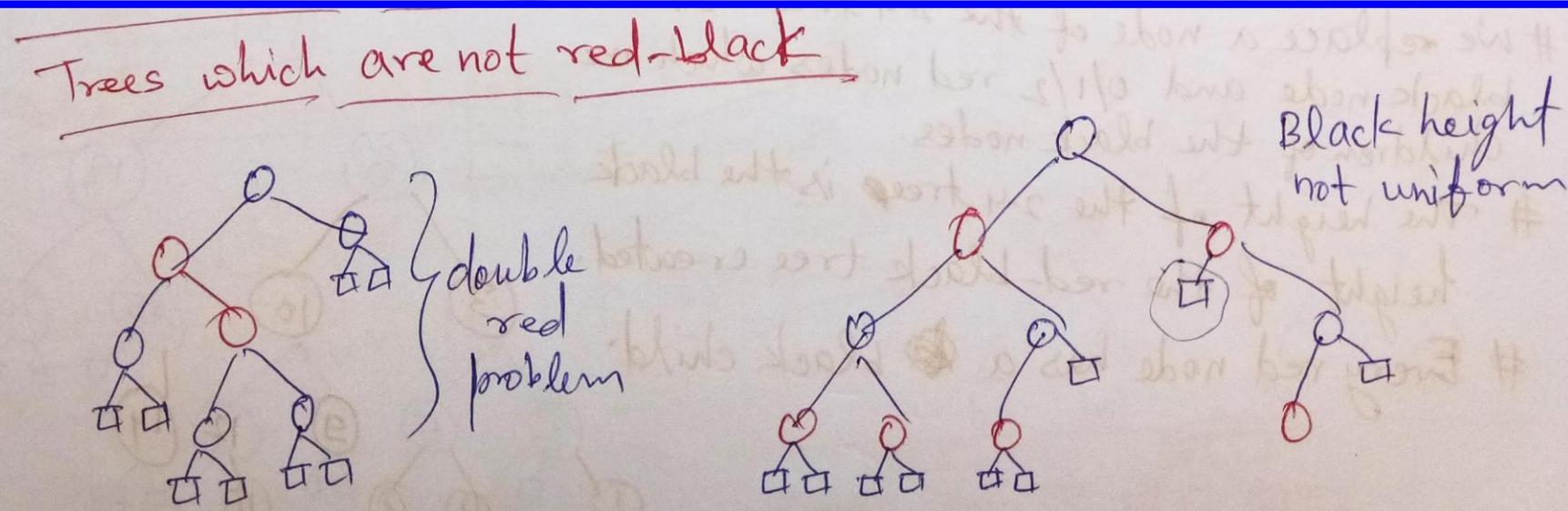
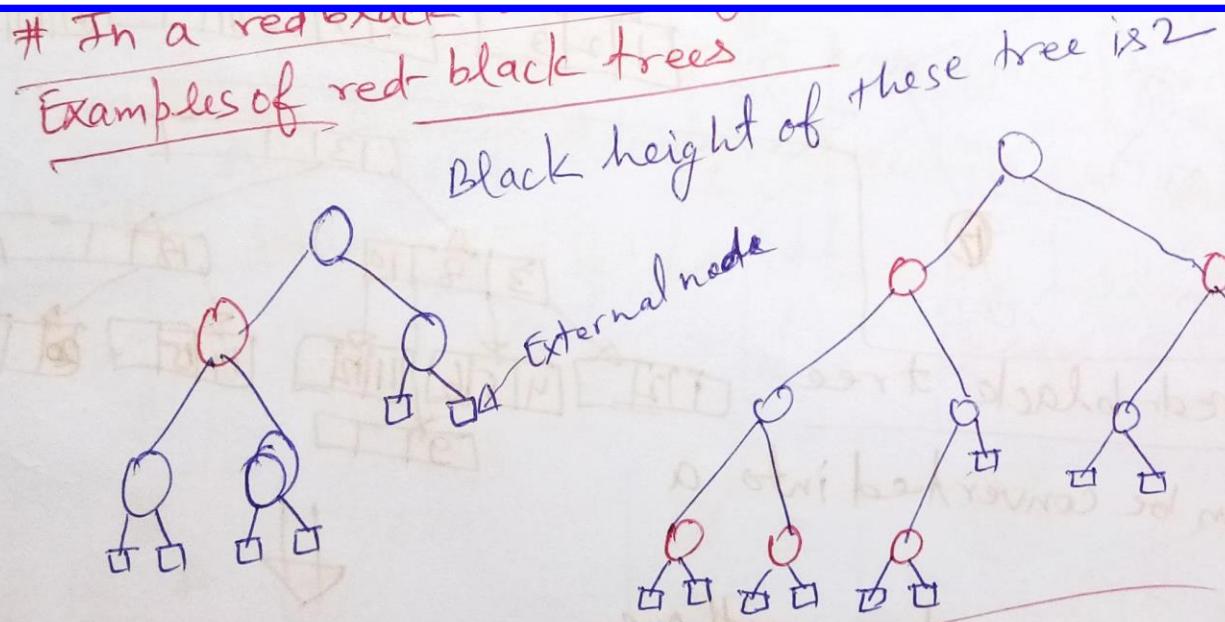
I'm not going to draw the NIL children in the future, but they are treated as black nodes.

# Red-Black Trees

## Red Black Trees

- # A red-black tree is a binary search tree in which each node is colored red or black.
- # The root is colored black.
- # A red node can have only black children.
- # If a node of the BST does not have a left and/or right child then we add an external node.
- # External nodes are not colored.
- # The black depth of an external node is defined as the number of black ancestors it has.
- # In a red black tree every external node has the same black depth.

# Red-Black Trees



# Red-Black Trees

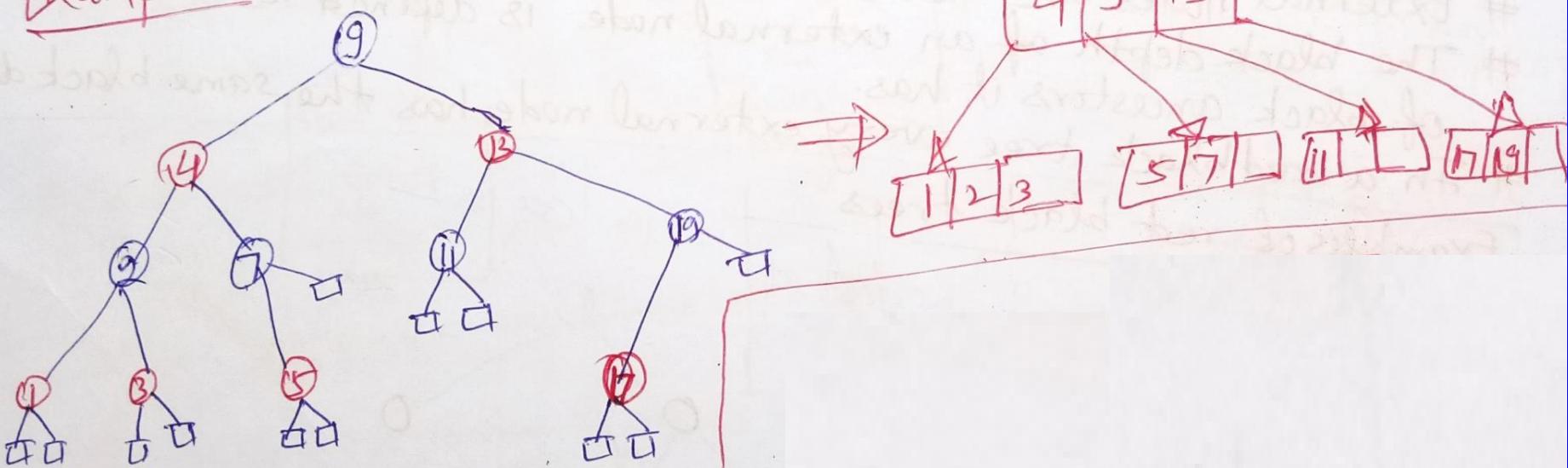
## ④ Height of a red-black tree

- # let  $h$  be the black height of a red-black tree on  $n$  nodes
- #  $n$  is smallest when all nodes are black. In this case
- #  $n$  is smallest when all nodes are black. In this case
- #  $n$  is largest when alternate levels of tree are red. Then height is  $2h$  and  $n = 2^{2h} - 1$ .
- # Hence,  $\log_4 n < h < 1 + \log_2 n$ .

# Red-Black Trees

- Red-Black tree to 2-4 trees can be converted into a 2-4 tree.
- # Any red-black tree can be converted into a 2-4 tree.
  - # Take a black node and its red children (at most 2) and combine them into one node of a 2-4 tree.
  - # Each node so formed has at least 1 and at most 3 keys.
  - # Each black height of all external nodes is same, in the resulting 2-4 tree all leaves are at same level.

Example:



# Red-Black Trees

2-4 Tree to RB Tree

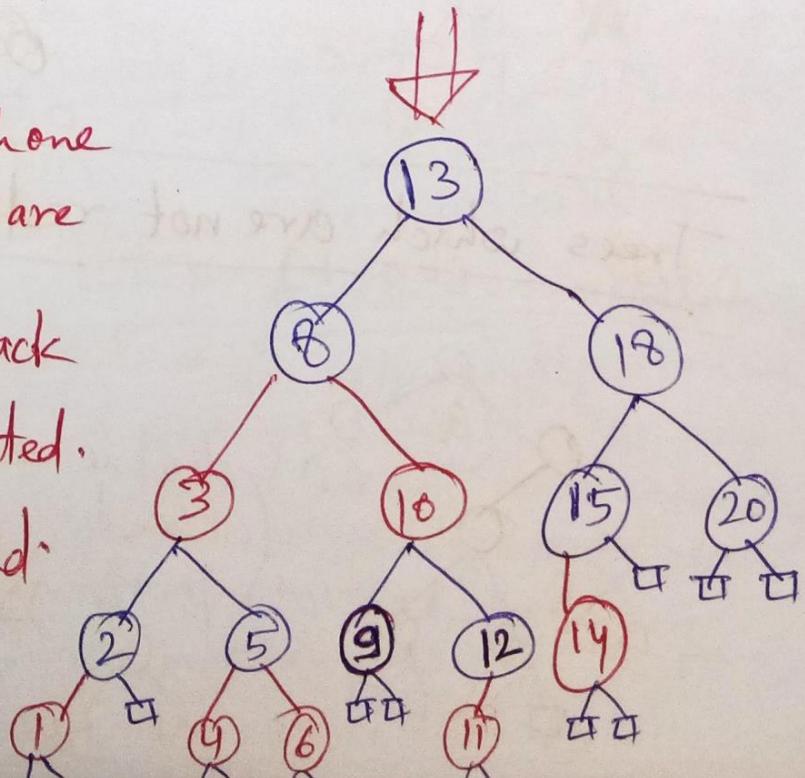
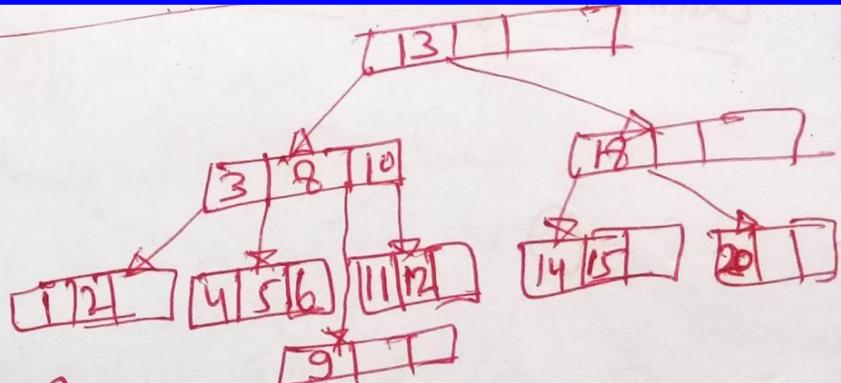
2-4 trees to red-black tree

# Any 2-4 tree can be converted into a red-black tree.

# We replace a node of the 2-4 tree with one black node and 0/1/2 red nodes which are children of the black nodes.

# The height of the 2-4 tree is the black height of the red-black tree created.

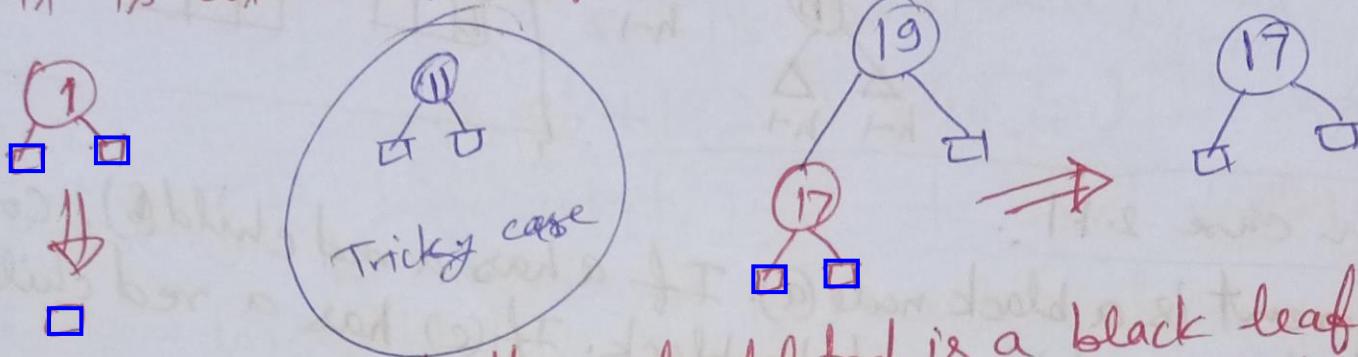
# Every red node has a black child.



# Red-Black Trees

## Deletion: preliminaries

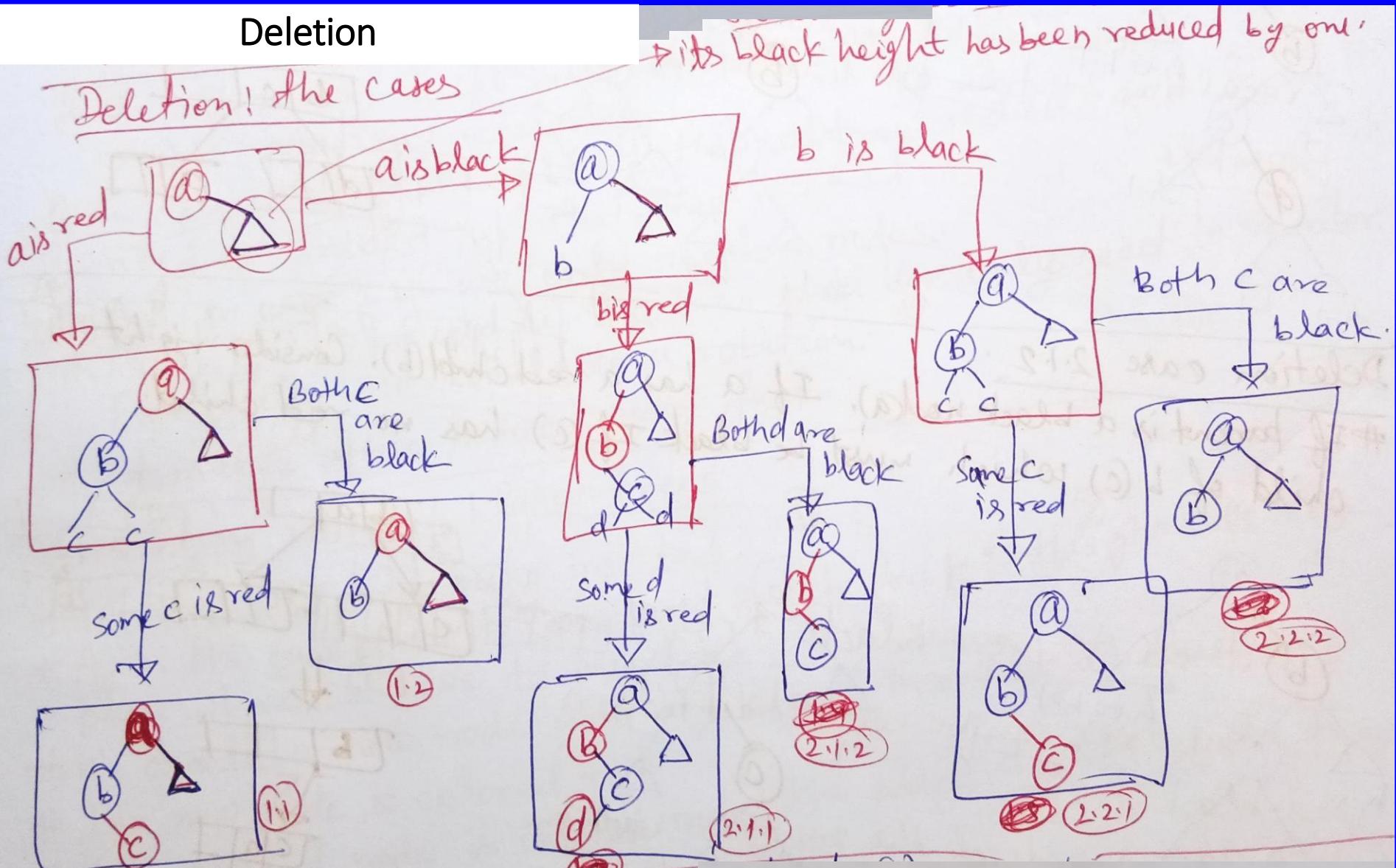
- # To delete a node we proceed as in a BST.
- # Thus the node which is deleted is the parent of an external node.
- # Hence it is either a leaf or the parent of a leaf.



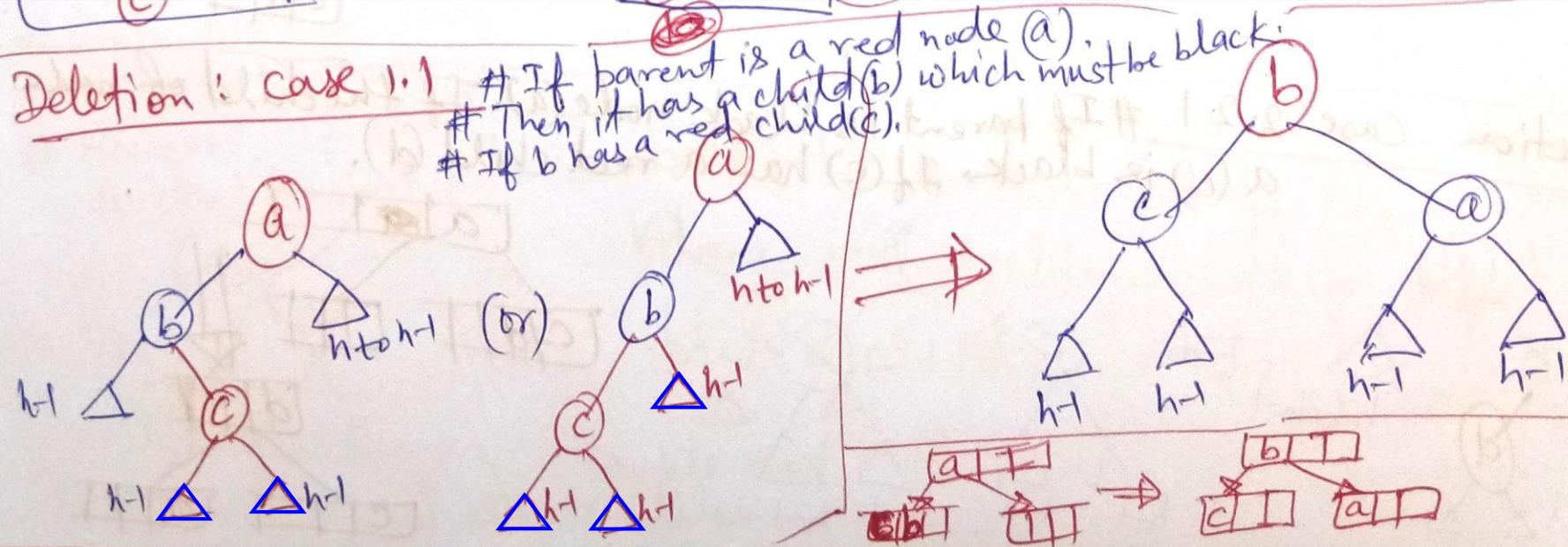
- # Hence we can assume that the node deleted is a black leaf.
- # Removing this reduces black depth of an external node by 1.
- # Removing this reduces black depth of an external node by 1.
- # Hence, in a general step, we consider how to reorganize the tree when the black height of some subtree goes down from  $h$  to  $h-1$ .

# Red-Black Trees

## Deletion



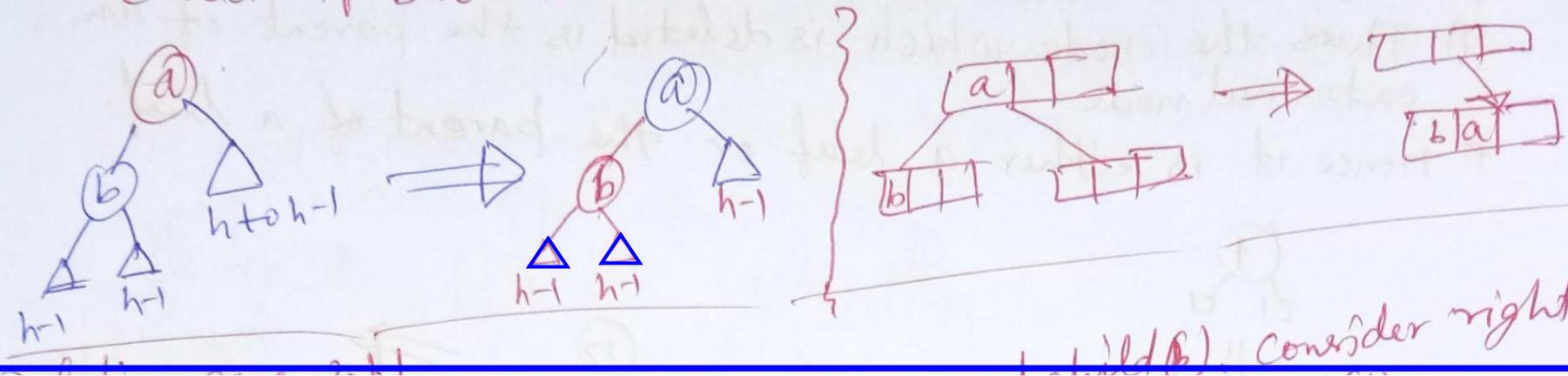
# Red-Black Trees



# Red-Black Trees

⑥ Deletion: case 1-2

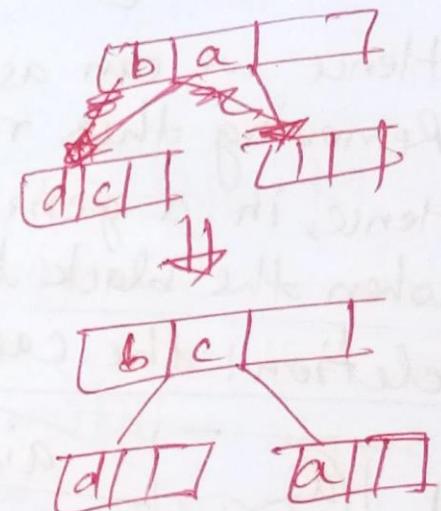
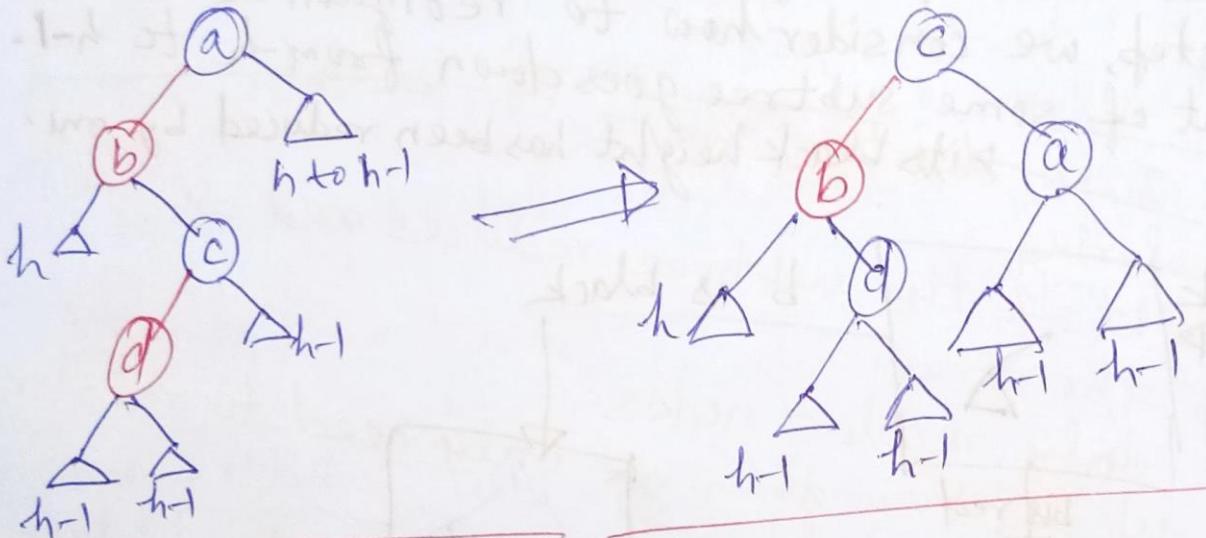
If parent is a red node ( $a$ ), then it has a child ( $b$ ) which must be black. If  $b$  has no red child.



# Red-Black Trees

Deletion case 2.1:

# If parent is a black node (a). If a has a red child (b). consider right child of b (c) which must be black. If (c) has a red child (d).

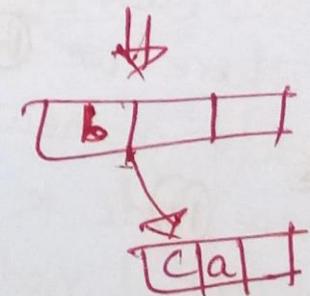
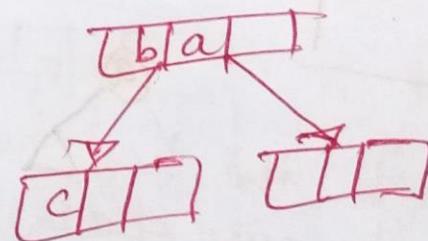
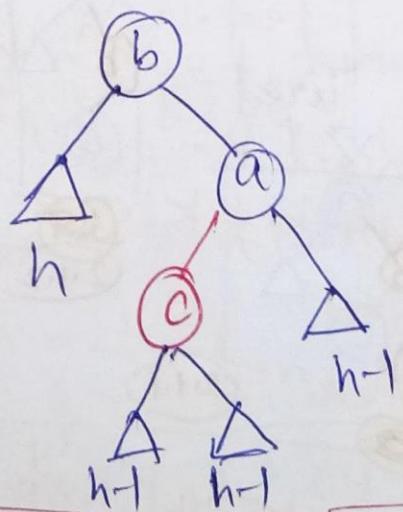
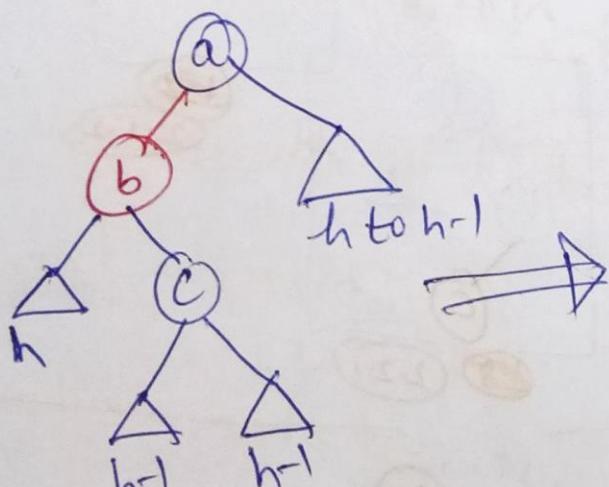


# Red-Black Trees

$h-1$      $h-1$

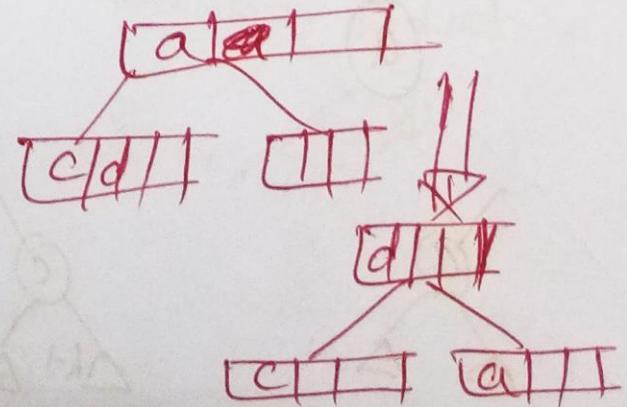
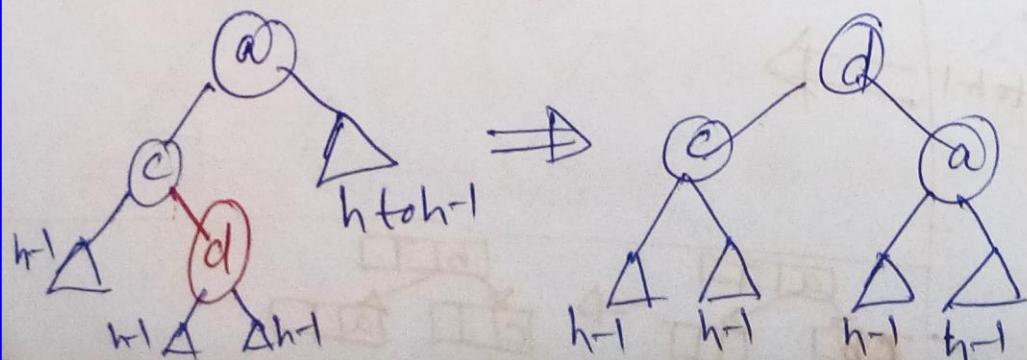
Deletion case 2.1.2:

If parent is a black node(a). If a has a red child(b). Consider right child of b (c) which must be black. If (c) has no red child.



# Red-Black Trees

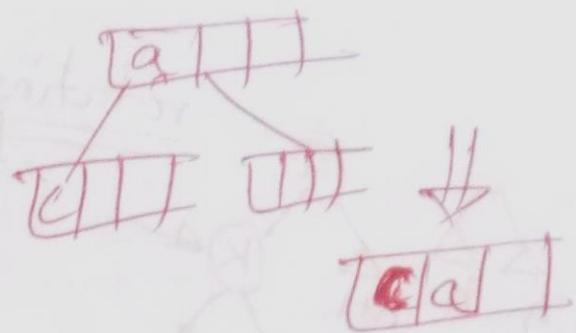
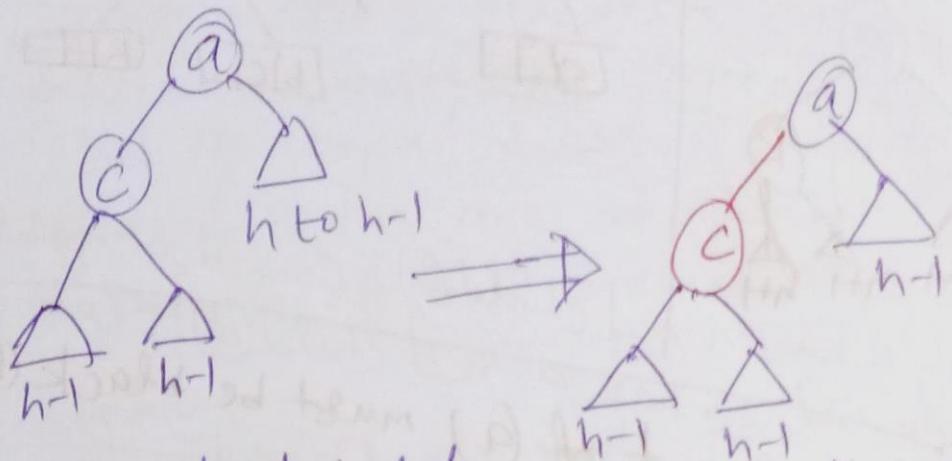
Deletion Case 2.2.1 # If parent is black node (a). If the child of node a (c) is black. If (c) has a red child (d).



# Red-Black Trees

Deletion: Case 2.2.2

# If parent is a black node(a). If the child of node a(c) is black. If (c) has no red child.



# Black height got reduced in this case.  
# It will cascade.

Noted by a

# Red-Black Trees

- Deletion: Summary
- # In all cases, except 2.2.2, deletion can be completed by a simple rotation/recoloring.
  - # In case 2.2.2, the height of the subtree reduces and so we need to proceed up the tree.
  - # But in case 2.2.2 we only recolor nodes.
  - # Thus, if we proceed up the tree then we only need to recolor.
  - # Eventually we would do a rotation.
  - # It makes this process very fast.

# Red-Black Trees

# It makes

In insertion in Red-black trees

# Let K be the key being inserted.  
# As in the case of a BST we first search for  $k$ ; this gives us the

place where we have to insert  $K$ .

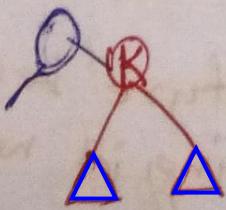
# We create a new node with key  $k$  and insert it at this place.

# The new node is colored red.

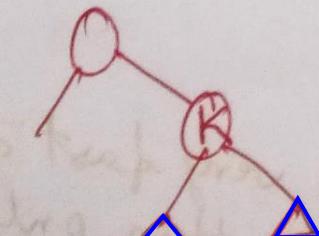
# Since inserted node is colored red, the black height of the tree

remains unchanged.

# However, if the parent of inserted node is also red then we have  
a double red problem.



No problem



Double Red  
problem

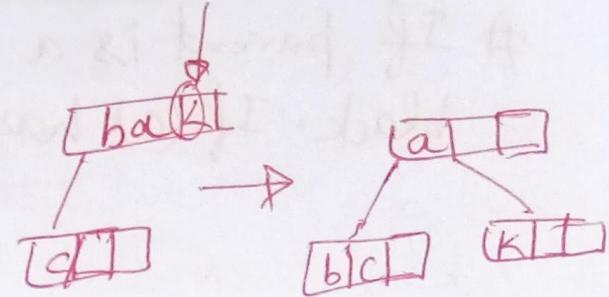
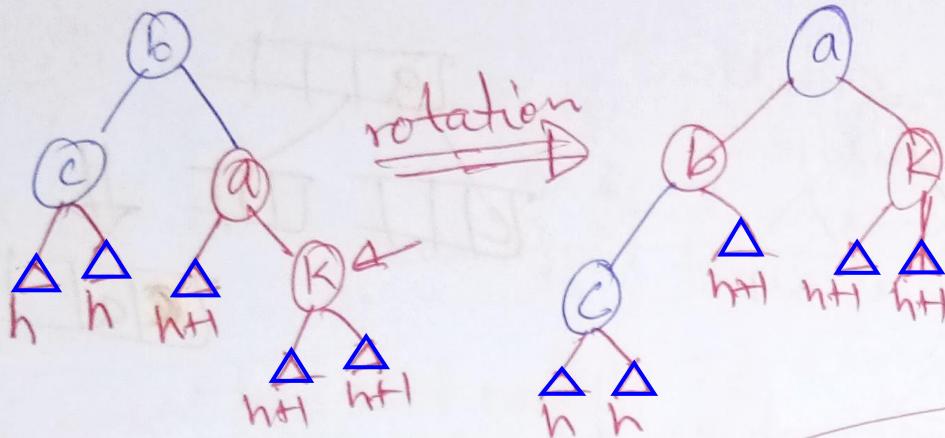
# Red-Black Trees

## ④ Insertion Case 1

# Parent of inserted node (a) is red.

# Parent of (a) must be black (b).

# The other child of (b) is black (c)

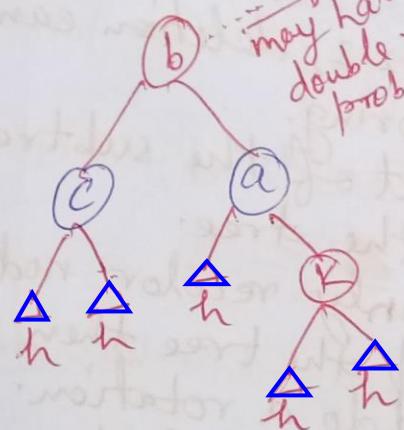
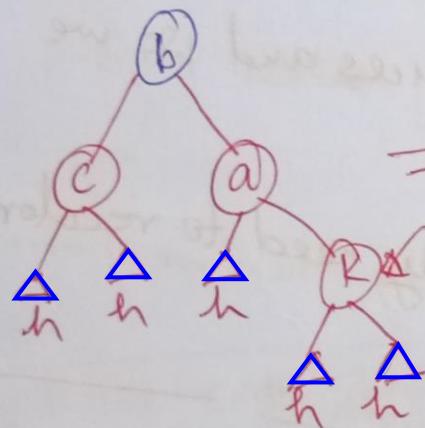


+ (a) must be black (b).

# Red-Black Trees

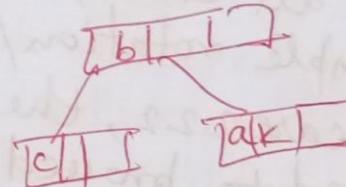
## Insertion Case 2

- # Parent of inserted node (a) is red. Parent of (a) must be black (b).
- # The other child of (b) is also red (c).



c b a

↑  
split



- # The parent of b could also be red. In that case the double red problem moves up a level.
- # We repeat this process at the next level.
- # Eventually, we might color the root node red.
- # In this case we recolor the root black. This increases the black depth of every external node by 1.
- # In the 2-4 tree this corresponds to splitting the root.

# Red-Black Trees

- # In the ~~end~~ summary:  
Insertion and Deletion: Summary
- # In both insertion and deletion we need to make at most one rotation.
- # We might have to move up the tree but in doing so we only recolor nodes.
- # Time taken is  $O(\log n)$ .
- # Thus, red-black tree is very fast data structure because when we move up in the tree then only recoloring is needed.

# This is great!

- SEARCH in an RBTree is immediately  $O(\log(n))$ , since the depth of an RBTree is  $O(\log(n))$ .
- What about INSERT/DELETE?
  - Turns out, you can INSERT and DELETE items from an RBTree in time  $O(\log(n))$ , while maintaining the RBTree property.

# INSERT/DELETE

- INSERT/DELETE for RBTrees
  - You should know what the “proxy for balance” property is and why it ensures approximate balance.

# Conclusion: The best of both worlds

	Sorted Arrays	Linked Lists	Red Black Trees
Search	$O(\log(n))$	$O(n)$	$O(\log(n))$
Delete	$O(n)$	$O(n)$	$O(\log(n))$
Insert	$O(n)$	$O(1)$	$O(\log(n))$

# Recap

- Balanced binary trees are the best of both worlds!
- But we need to keep them balanced.
- **Red-Black Trees** do that for us.
  - We get  $O(\log(n))$ -time INSERT/DELETE/SEARCH
  - Clever idea: have a proxy for balance