Book

A Simplified Approach to

Data Structures

Prof.(Dr.)Vishal Goyal, Professor, Punjabi University Patiala

Dr. Lalit Goyal, Associate Professor, DAV College, Jalandhar

Mr. Pawan Kumar, Assistant Professor, DAV College, Bhatinda

Shroff Publications and Distributors

Edition 2014

PRESENTATION
ON
RED BLACK TREE
&
AVL TREE

Contents:

- Introduction to Red Black Tree.
- Operation On Red Black Tree.

Introduction To Red-black Trees

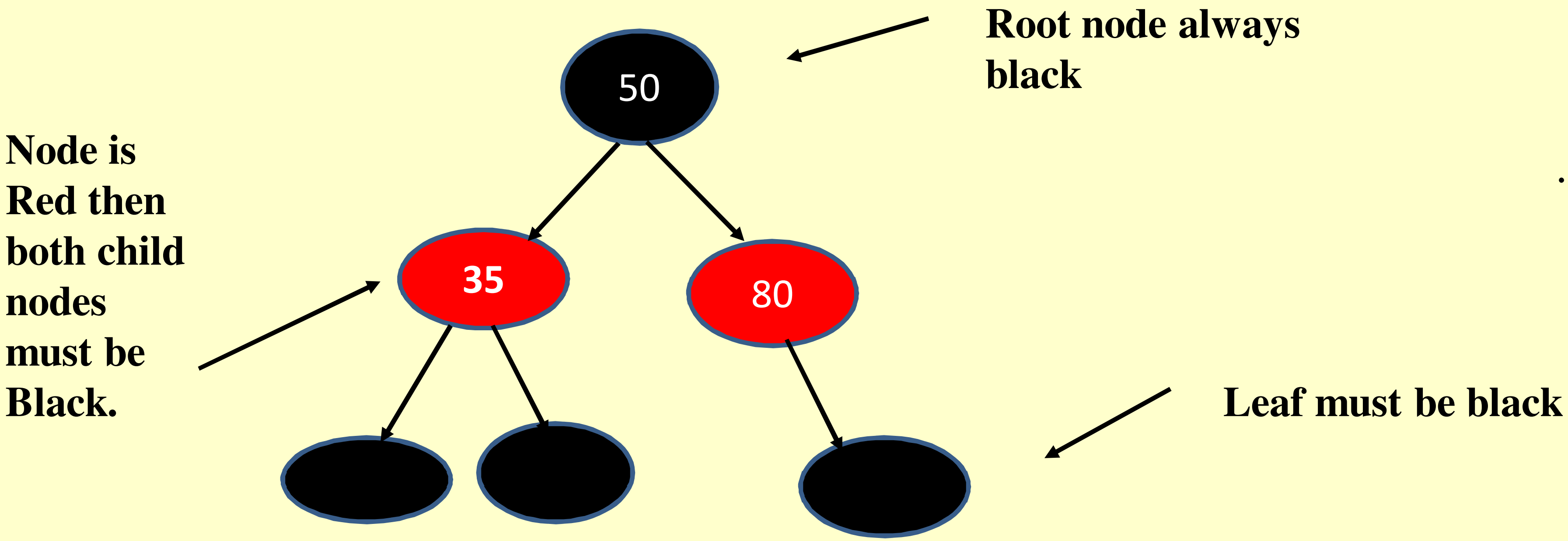
- Red-Black Tree is a binary search tree in which every node is colored red or black.
- The red black tree is a balanced tree because no path is more than twice as long as any other path.
- Each node of red-black tree contains five fields.

color	**Left**	**Key**	**Right**	**Parent**

Red-black Tree Properties

- Every node of the tree is either red or black.
- The root node of the tree is always black.
- If a node in the tree is red then its both the child nodes must be black.
- All the paths from a node to its descendent leaf nodes contain the same number of black nodes.
- All the leaf nodes must be black.

Example Of Red Black Tree



About Red-black Tree

- A red black tree with n nodes has the maximum height of almost $\boldsymbol{2log_2(n+1)}$**.**
- The number of black nodes on any path from root node to leaf nodes is known as **black height** of the tree,
- No path is more than twice as long as any other path in the tree.

Operations On RB Trees

- Searching
- Insertion
- Deletion

Searching

- Search Operation Is Defined As Finding The Address Of A Node Containing Desired Element.
- Search Operation On Red-black Tree Is Performed Similar To The Search Operation For Binary Tree Because Red Black Tree Is A **Binary Search Tree**.
- Complexity Of Search Operation In Red-black Tree Is Taken As $O(\log_2 n)$.

Searching Algorithm

BSTSearch(Root,Item,Position,Parent)

Step1: If **Root=Null** Then

set **Position = Null**

set **Parent = Null**

Return

[End If]

Step 2: **Pointer=Root** And **Pointer P = Null**

Step 3: Repeat Step 4 While **Pointer ≠ Null**

Step 4: If **Item = Pointer→Info** Then

set **Position = Pointer**

set **Parent = PointerP**

Return

Else If **Item<Pointer→Info** Then

Searching(cont.)

Set **PointerP=Pointer**

Set **pointer =Pointer → Left**

Else

Set **PointerP=Pointer**

Set **pointer =Pointer → Right**

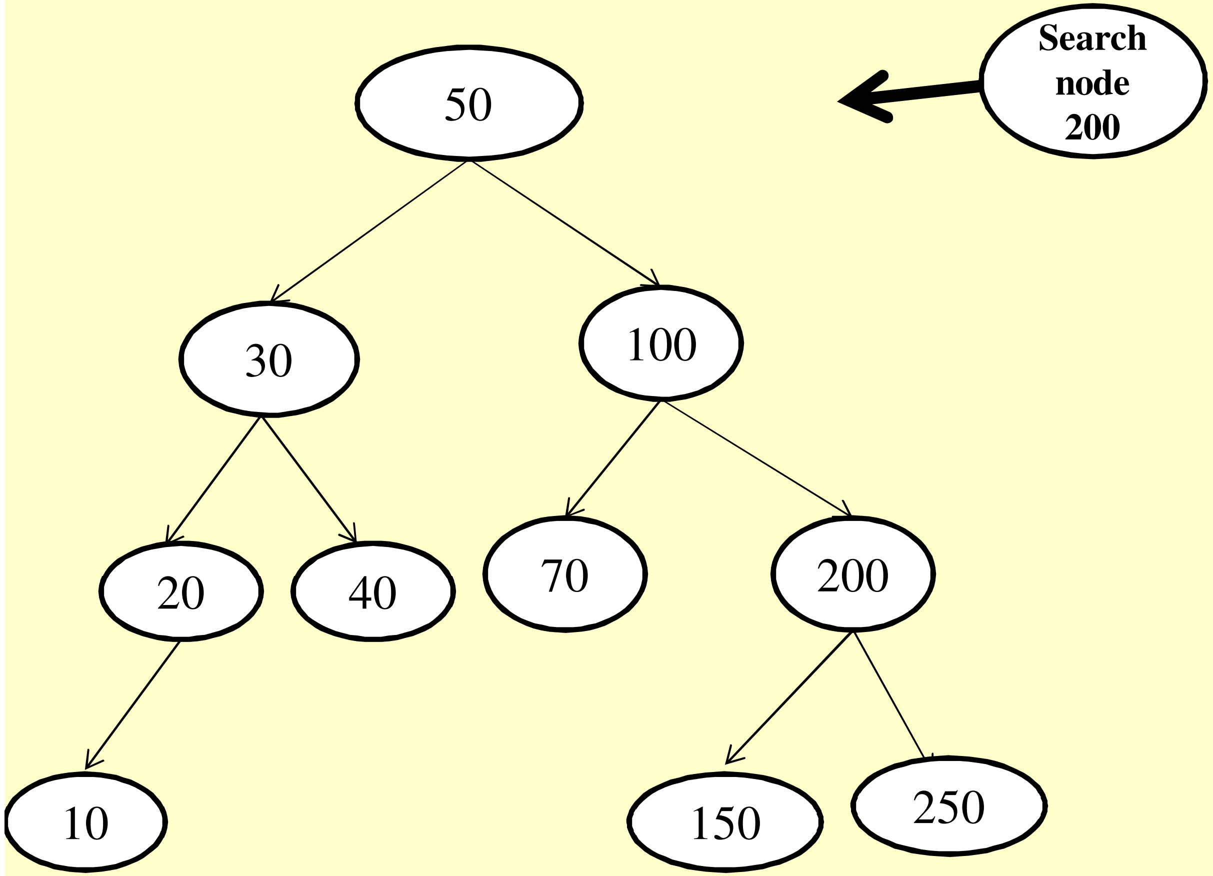
[End of IF]

[End of Loop]

Step 5: Set **Pointer =Null and Parent=Null**

Step 6: Return

Example Of Searching:



If **Item = Pointer →Info**

Item is root;

Else If **Item<Pointer→Info**

item is found in left;

Else

item is found in right;

Item not found

Insertion

- Insertion operation in RB tree is performed in similar manner as in Binary search tree with possibility that it may result in violation of red-black tree properties because new node to be inserted is always red.
- So, to restore the properties of RB Tree after inserting new node, we may need to
  - Change the color of some nodes.
  - Rotate the tree in left or right direction.

Rotations

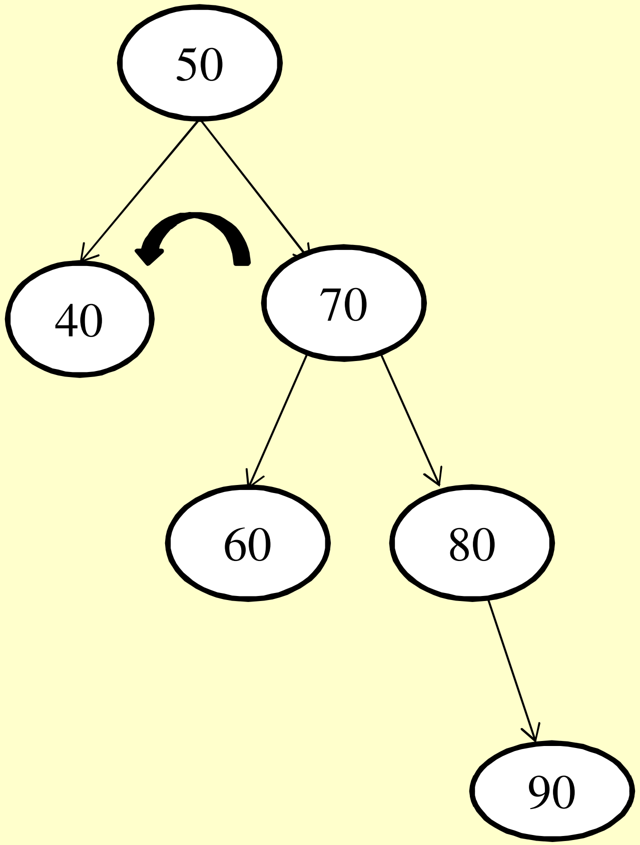
- Before discussing **insertion** operation on RB tree you must clear with the concept of Rotation and how its performed.
- Two types of rotation that are:
- Left Rotation
- Right Rotation

Left Rotation

- The left rotation is performed by performing the configuration of two nodes on the right into the configuration on the left side .

Example Of LR Rotaion:

Apply left rotation on key =50



Left-Rotate

Left- Rotation Algo:

Left-Rotate (*T, x*)

Step 1 : If **x →Right ≠ Null** Then

//Attach y's left sub tree as x's right sub tree.

Step 2. x →Right = y → Left

// If y has a left child then make x as parent.

Step 3: If **y → Left ≠ Null** Then

y → Left → Parent = x

[End If]

Step 4: y → Parent = x → Parent // Make x's parent as y's parent.

//if x is the left child of its parent then make y as its left child.

Step 5: If **x → Parent → Left = x** then

x →Parent → Left = y

//if x is the right child of its parent then make y as its right child.

Cont.

Else If **x → Parent → Right = x** Then

x → Parent → Right = y

[EndIf]

Step 6: y → Left = x //Make x as y's left child

Step 7: x → Parent = y //Make y as x's new Parent

Else

Print:"Left Rotation is not Possible"

[End If]

Step 8: Exit

Right Rotation:

- The right rotation is performed by transforming the configuration of two nodes on the left into the configuration on the right side .

Right Rotation – Pseudo-code

<u>Right-Rotate (*T, x*)</u>

Step 1: If **x → Left ≠ Null** Then

Step 2 : **x → Left = y → Right** //Attach y's right sub tree as x's left sub tree.

// If y has a right child then make x as its parent.

Step 3: If **y → Right ≠ Null** Then

y → Right → Parent = x

[End If]

Step 4: y → Parent = x → Parent // Make x's parent as y's parent

//if x is the left child of its parent then make y as its left child.

Cont.

Step 5: If **x → Parent → Left = x** then

x →Parent → Left = y

//if x is the right child of its parent then make y as its right child.

Else If **x → Parent → Right = x** Then

x → Parent → Right = y

[End If]

Step 6: y **→Right=x** //Make x as y's right child

Step 7: **x → Parent=y** //Make y as x's new Parent

Else

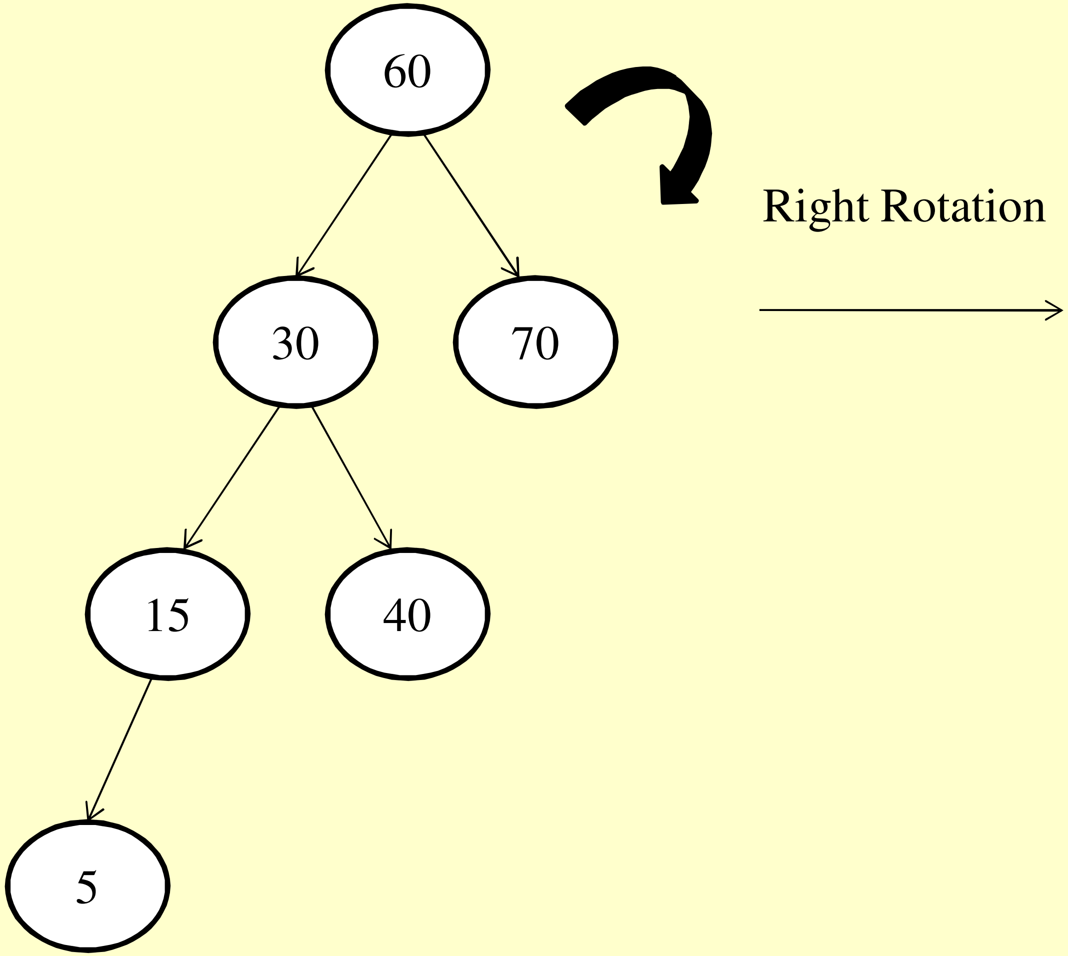
Print "Right rotation is not possible"

[End if]

Step 8: Exit

Example Of Right Rotation

Right Rotation on key =60

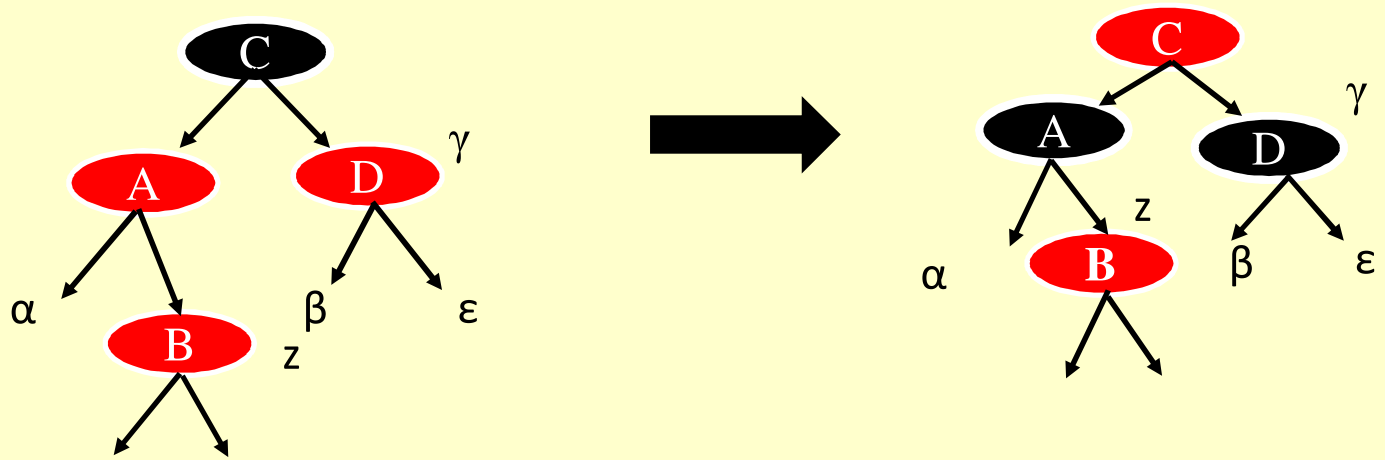


INSERTION IN RB TREES

- Insertion must preserve all red-black properties Should an inserted node be colored Red? Black?
- Basic steps:
  - o Perform Insertion as same in BST.
  - o Color the node x red.
  - o Fix the modified tree by re-coloring nodes and performing rotation to preserve RB tree property.

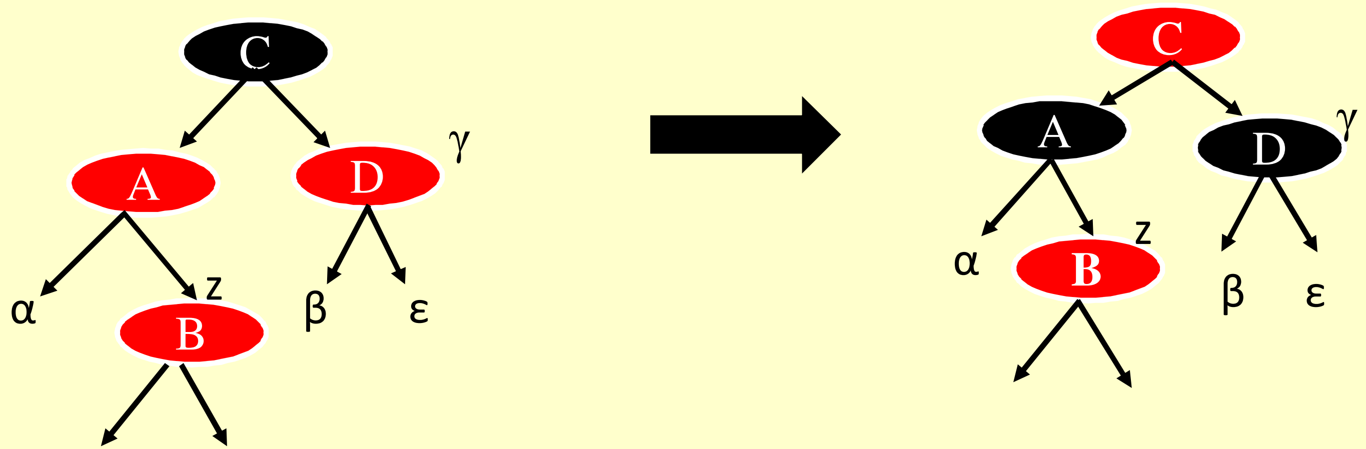
INSERTION IN RB TREES

- To restore the red-black tree properties, there are four cases discussed below.
- Case 1: If parent of node x is left child and uncle of node x is red
- then perform color flip.
- Grandparent of x becomes rγd and both the parent and uncle of x becomes black.



Cont.

- Case 2: If parent of node x is right child and uncle of node x is red then perform color flip.
- Grandparent becomes red and both the parent and uncle of x becomes black.



Cont.

- Case 3: If parent of node x is left child and uncle of node x is black then two sub cases arise:
- If x is right Child of its parent then perform the left rotation. After this rotation, the node x and its parent are interchanged and we take the new child as x which is now the left child.
- If x is left child of its parent then change the color of parent of node x and grandparent of node x and perform right rotation.

Cont.

- Case 4: If parent of node x is right child and uncle of node x is black then two sub cases arise:
  - If x is left Child of its parent then perform the right rotation. After this rotation, the node x and its parent are interchanged and we take the new child as x which is now the right child.
  - If x is right child of its parent then change the color of parent of node x and grandparent of node x and perform left rotation.

Insertion Algorithm Of RB Tree

Step 1: Insert the node x in Red- Black tree using BSTInsertion() algorithm and color the node x as Red.

Step 2: Repeat while(**x → Parent → Color = Red**)

Step 3: If (**x → Parent** = **x → Parent → Parent → Left**) Then

Step 4: If(**x → Parent → Parent → Right → Color = Red)** Then

//case1 when x's parent is left child and x's uncle is red.

x → Parent → Color = Black

x → Parent → Parent → Right → Color

x → Parent → Parent → Color =

x = x → Parent → Parent

Else

Cont.

//case3 when x's parent is left child and x's uncle is red.

If(x = x → Parent → Right) Then

//If x is right Child

x = x → Parent

Left Rotate(T,x)

[End If]

//If x is Left Child

x → Parent → Color = Black

x → Parent → Parent → Color = Red

Right Rotate(x → Parent → Parent)

[End If]

Else

Step 6: If(x → Parent → Parent → Left→ Color = Red) Then

Cont.

//case2: when x's parent is right child and uncle is red.

x → Parent → Color = Black

x → Parent → Parent → left→ Color = Black

x → Parent → Parent → Color = Red

x = x → Parent → Parent

Else

//case4: when x's parent is right child and its uncle is red.

If(**x** = **x → Parent → Left**) Then

x = x → Parent

Right Rotate(x)

[End If]

Cont.

x → Parent → Color = Black

x → Parent → Parent → Color = Red

Left Rotate(x → Parent → Parent)

[End If]

[End IF]

[End Loop]

Step 7:. **Root → Color = Black**

Step 8:. Exit

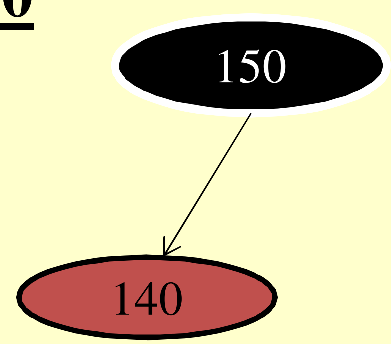
Example Of Insertion

- Let us make the red black trees using the following elements:

150, 140, 130, 120, 125, 122, 110, 100

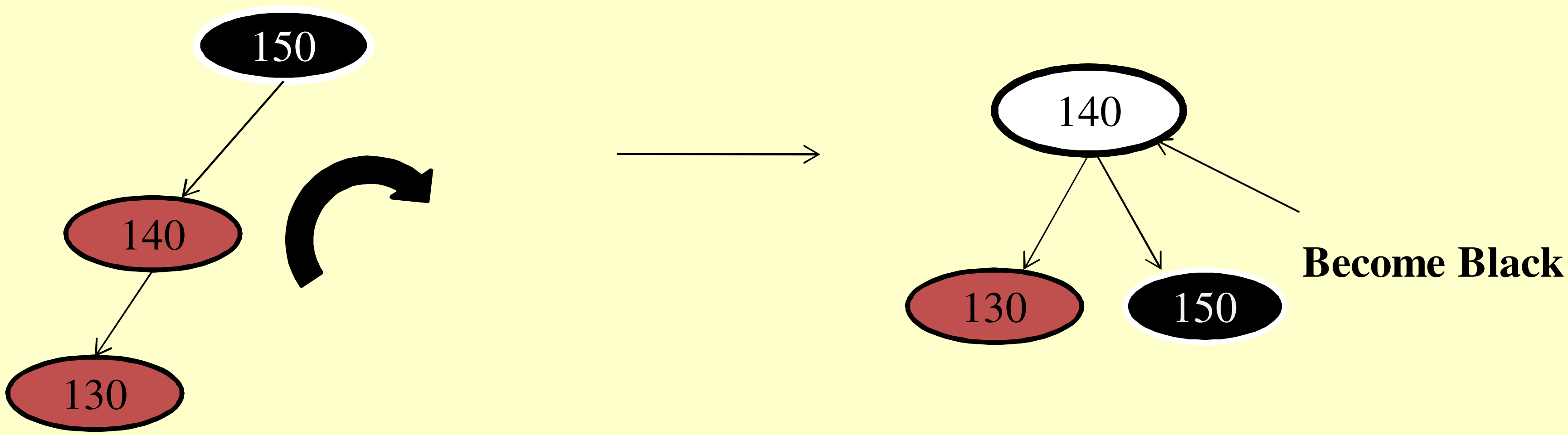
- **Insert 150**

- **Insert 140**

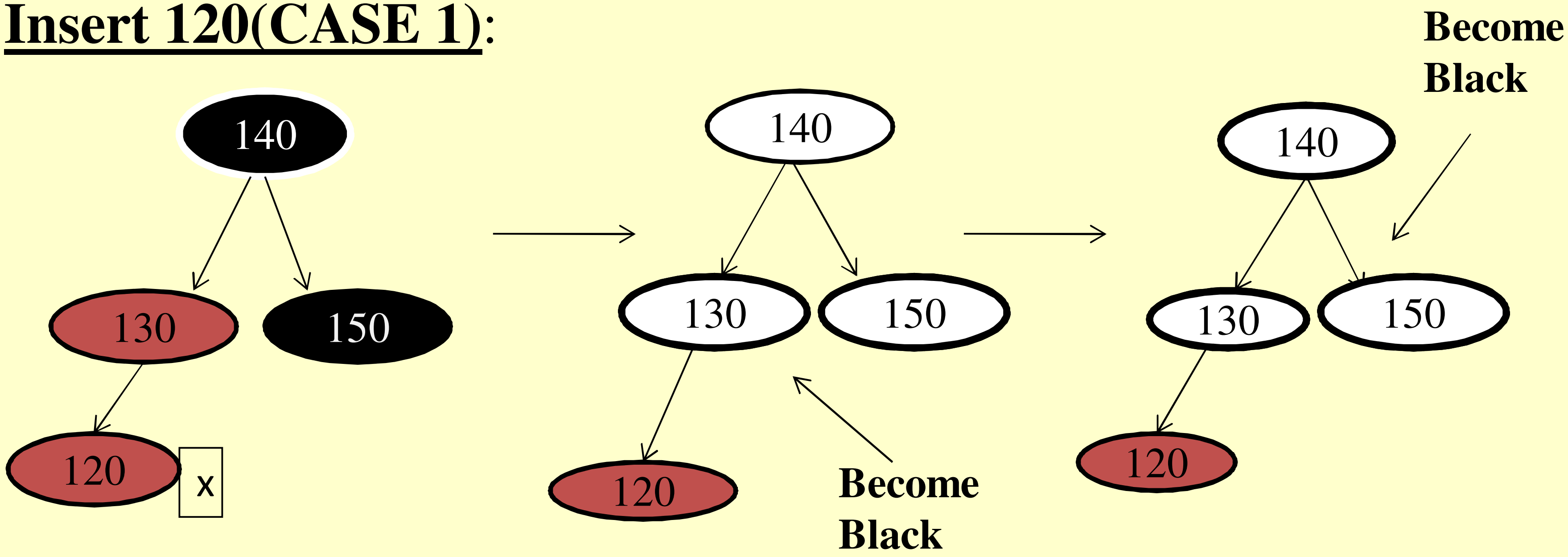


- **Insert 130 (CASE 3):**

Cont.

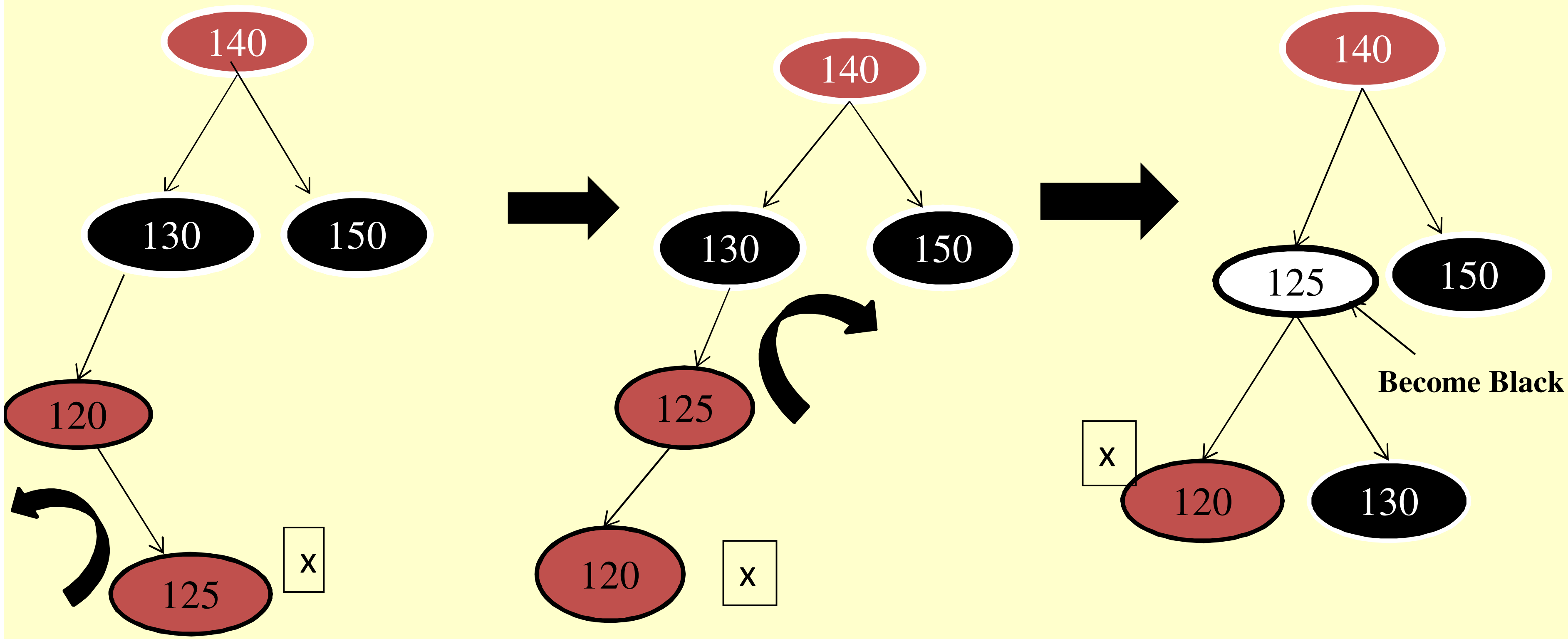


- **Insert 120(CASE 1):**

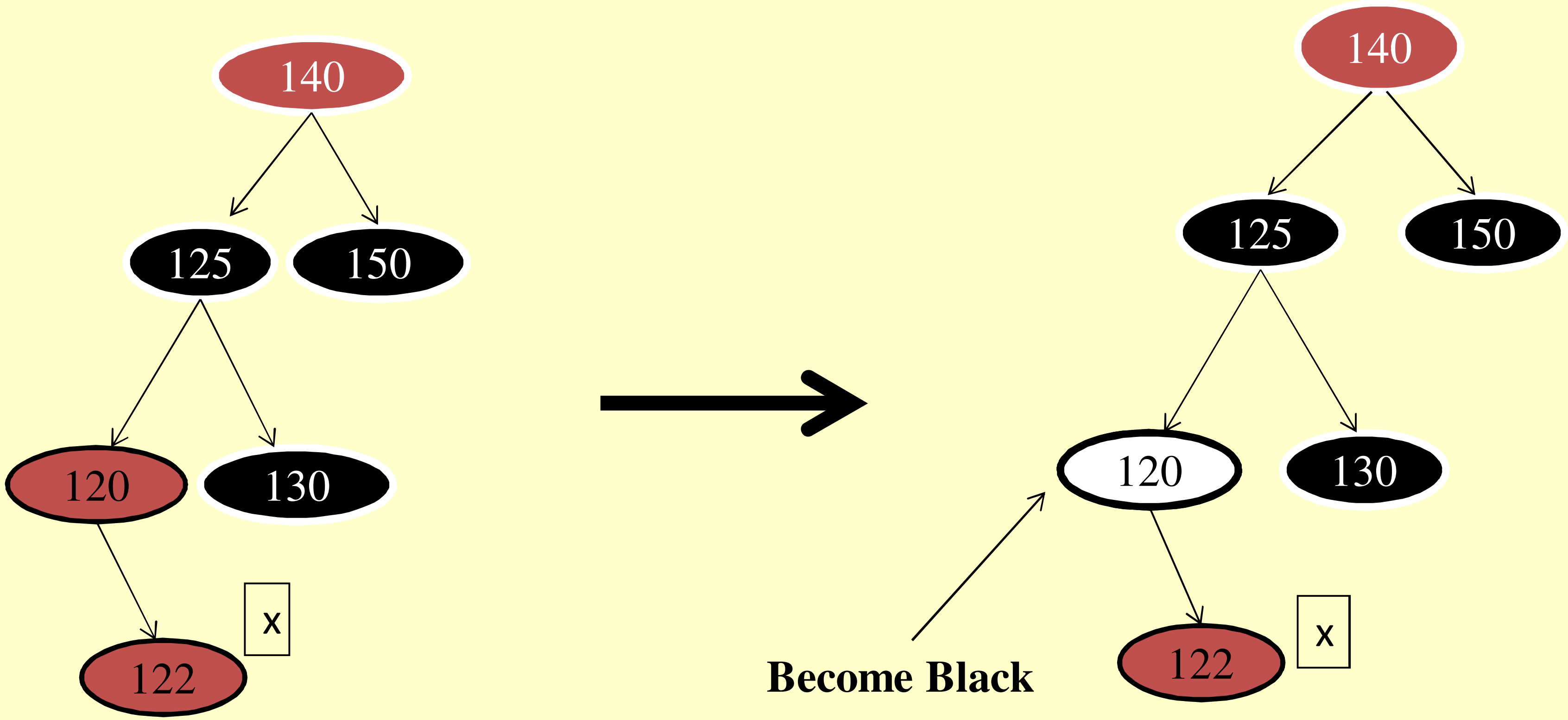


○ Insert 125(CASE 3):

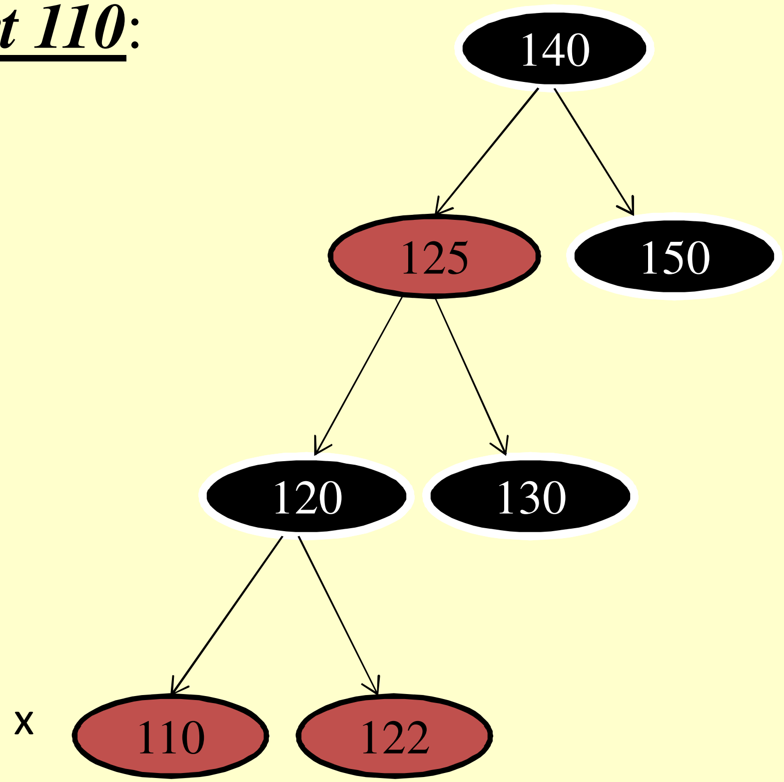
Cont.



○**Insert 122(CASE 3):**

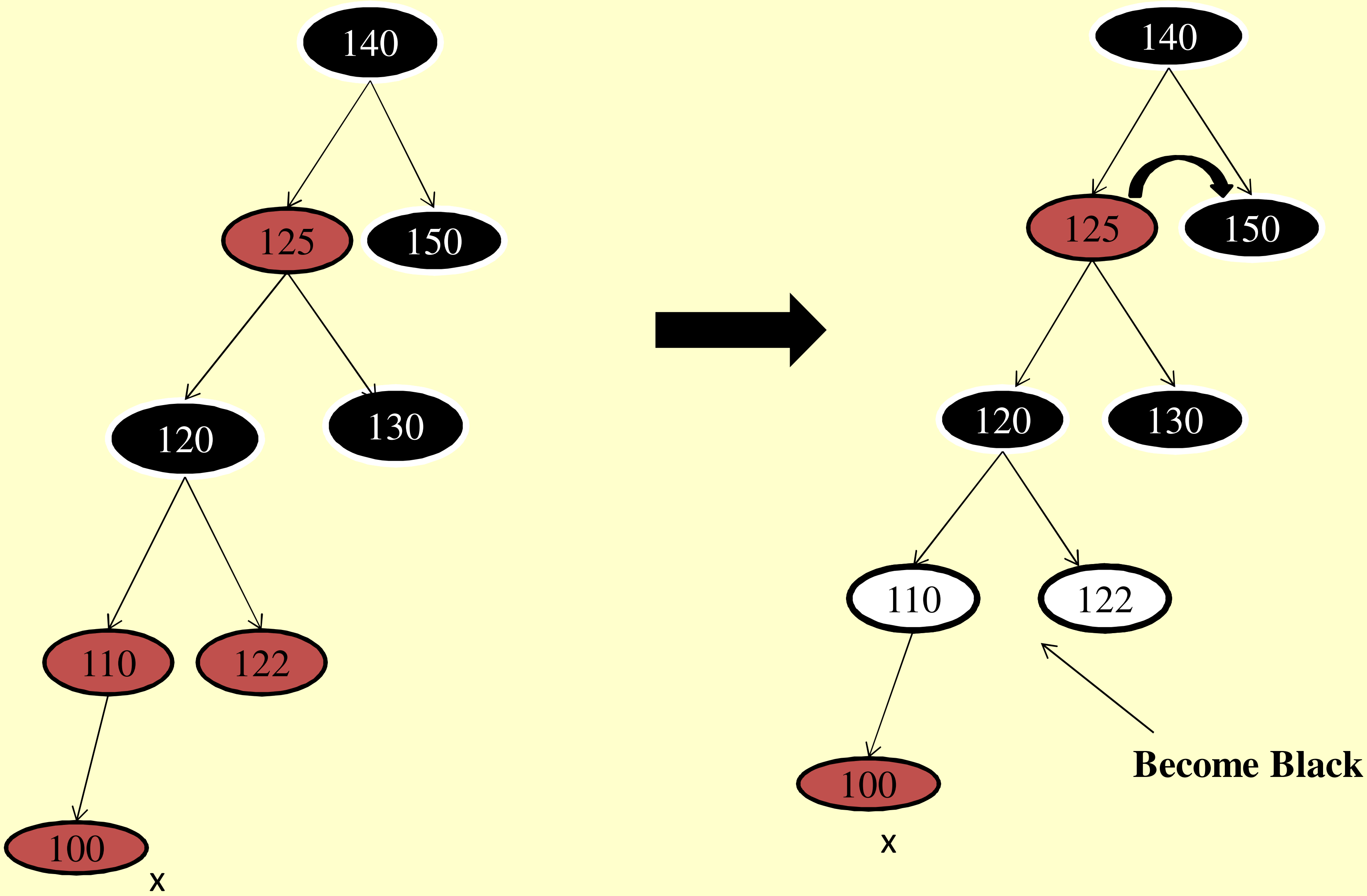


- ***Insert 110***:

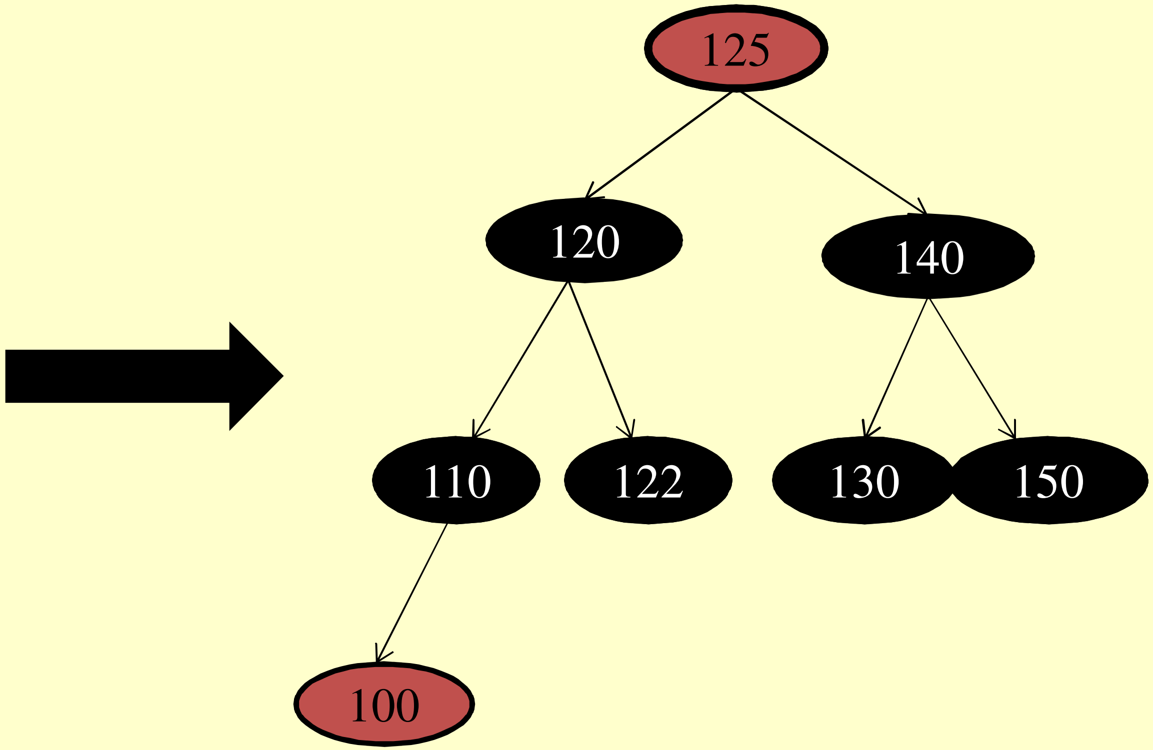


○ Insert 100(CASE 1 AND CASE 3):

Cont.



Cont.



FINAL RED BLACK TREE

Deletion Of Red Black Tree

Deletion of node from red black tree is composed of two steps:

In the **1st step** , the node is deleted just like the deletion Process in binary tree. After step 1 , the resulting the tree may not be the red black tree .

This is because the node to be deleted may be black and even this node may be replaced by its successor which may cause the red – red conflict or it may cause the number of back nodes in each path from root to leaf nodes of the tree to be different.

Cont.

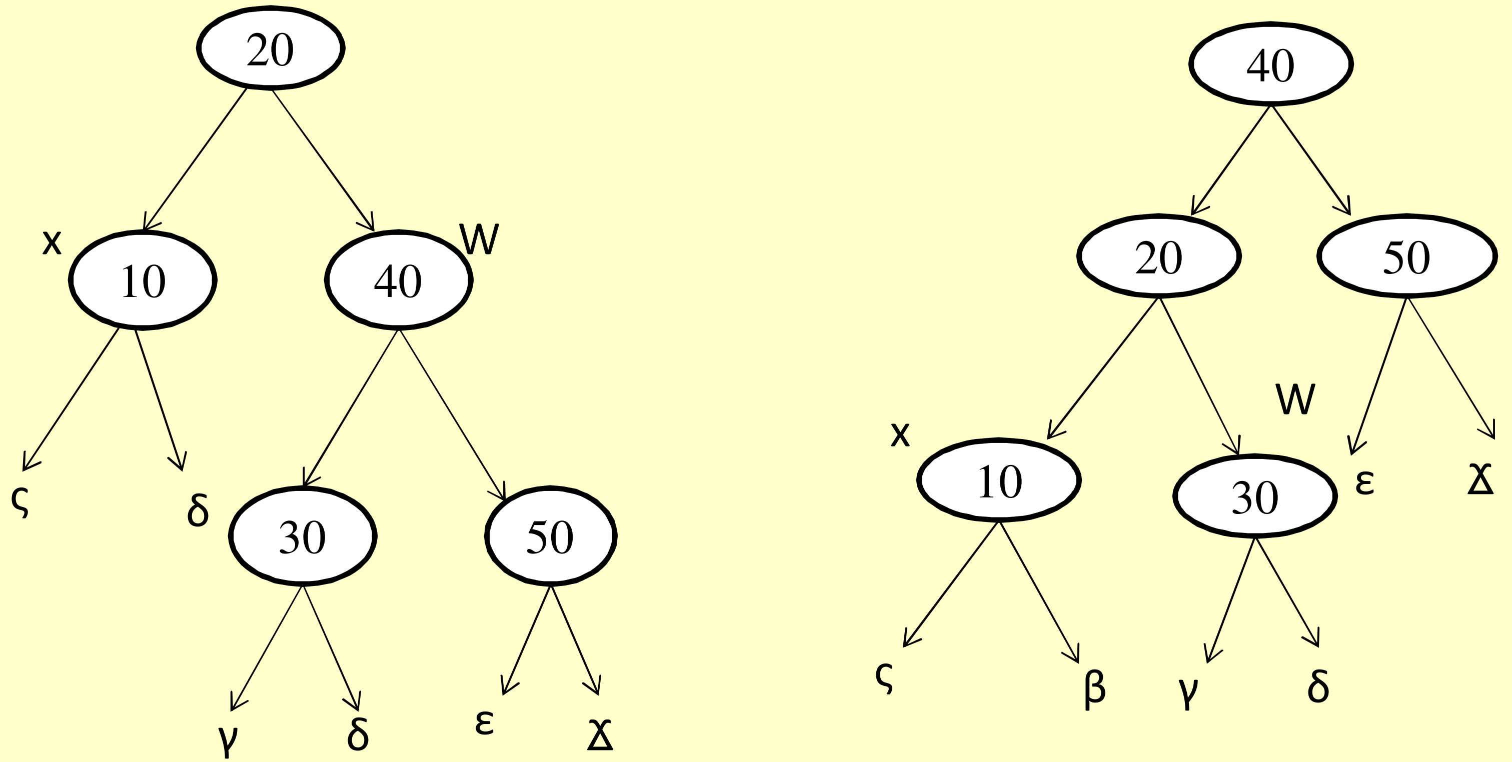
In **<u>the step 2</u>**: there are four deal with the 2^{nd} step of deletion process.

Before discussing the cases involved in deletion process , it must be remembered that x is the successor of node to be deleted .color of red nodes are indicted by empty circle , color of black nodes are indicted by dark filled circle and a light shaded color of nodes indicate either red or black nodes.

- **Case 1:** **Cont.**

If the node x’s sibling w is red then switch the color of w and x-> parent without violating the Properties of red –black tree.

After completion of this step, the procedure moves to case 2 or 3 or4.

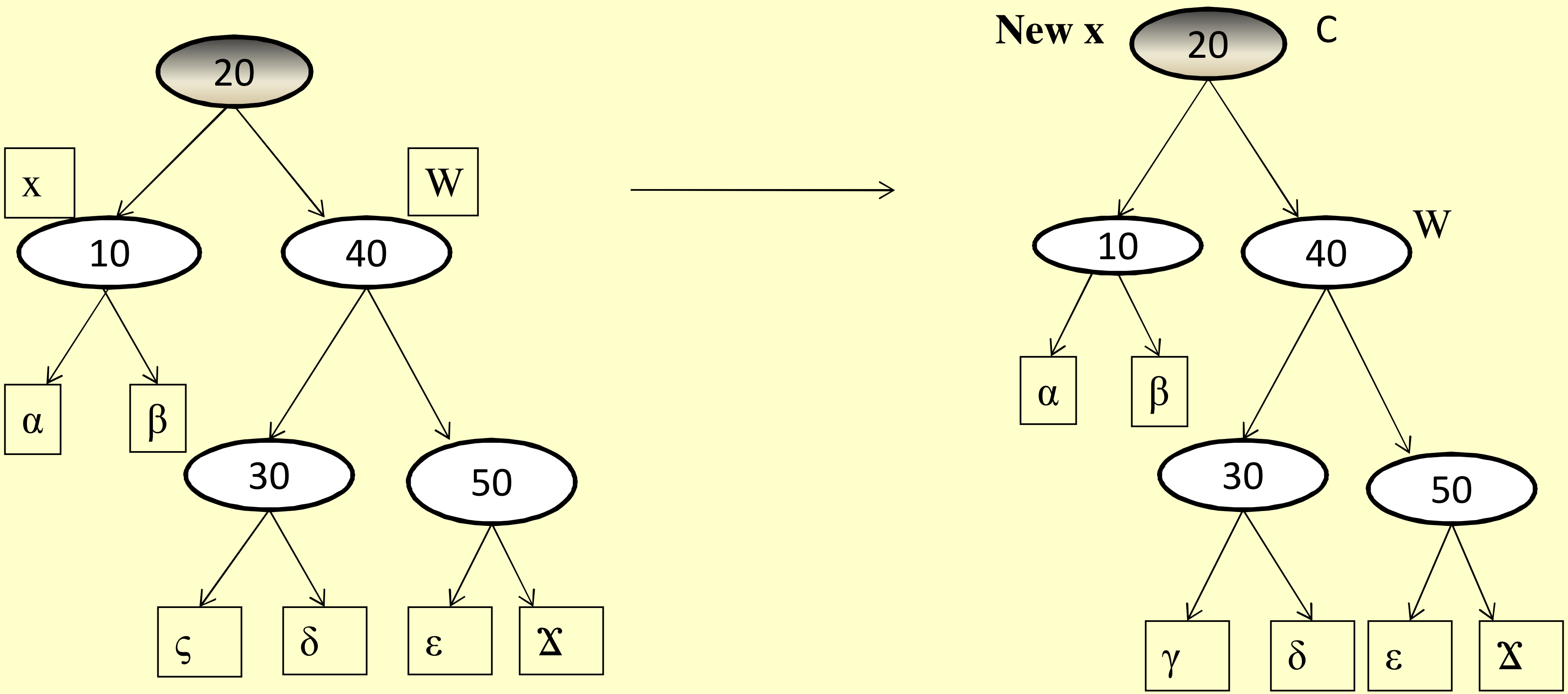


Case 2,3 and 4 occurs if x’s sibling w is black.

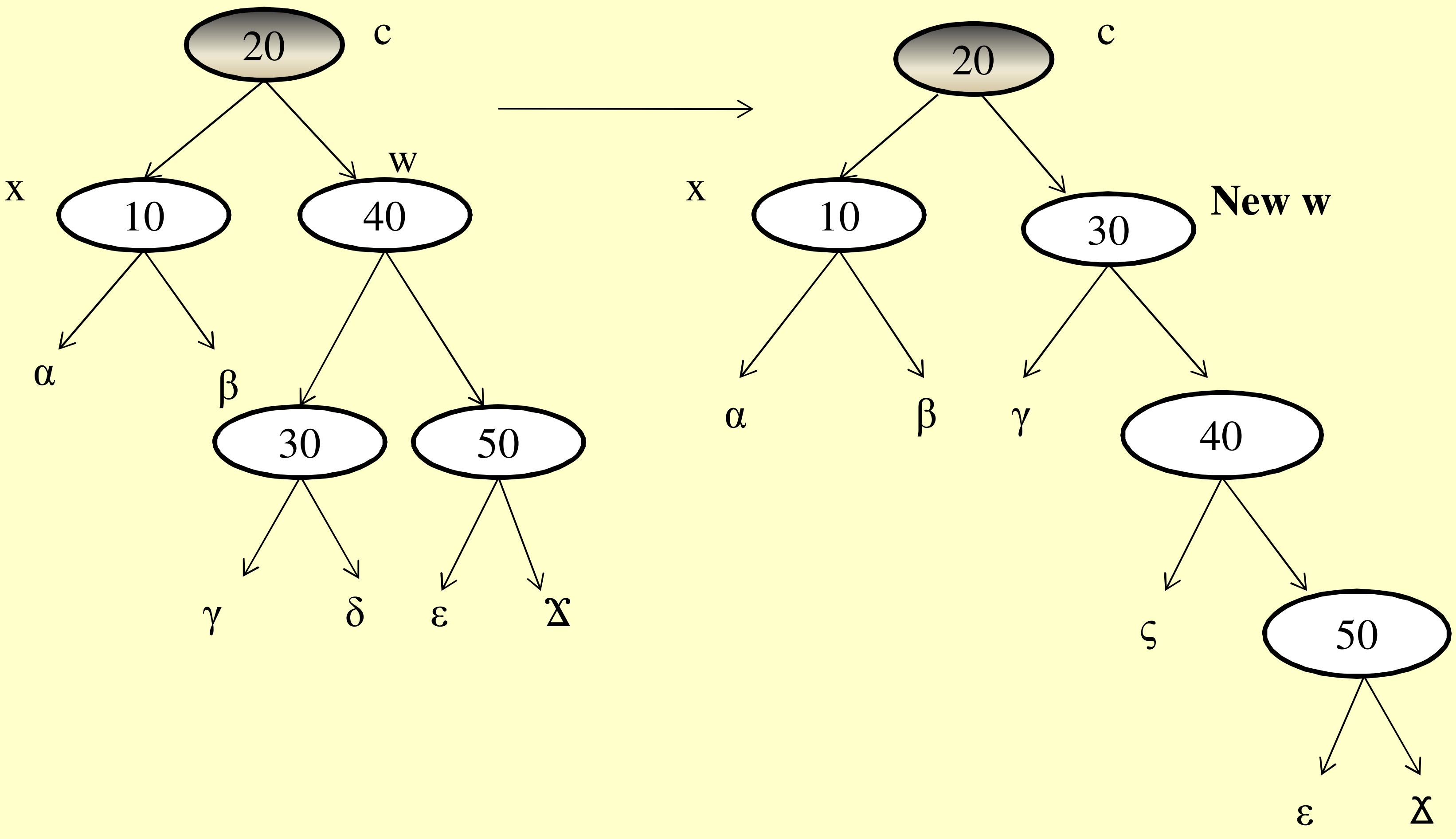
Note: a, b,c and so on indicates the arbitary subtrees

Cont.

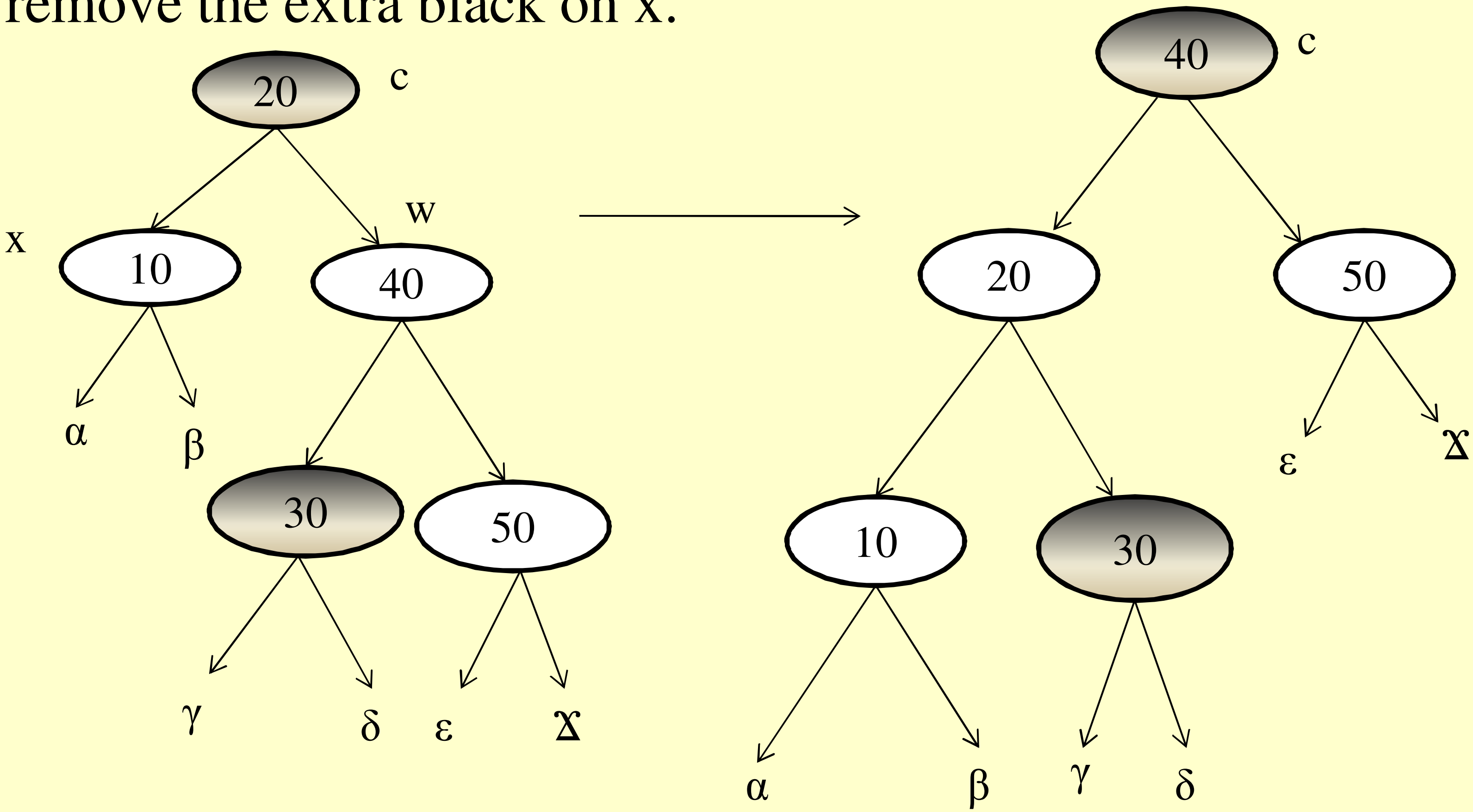
- **Case 2**: If the sibling w is black the child nodes of w are also black then sibling of x is changed to red and the parent of node x is made new x, which may be red or black.



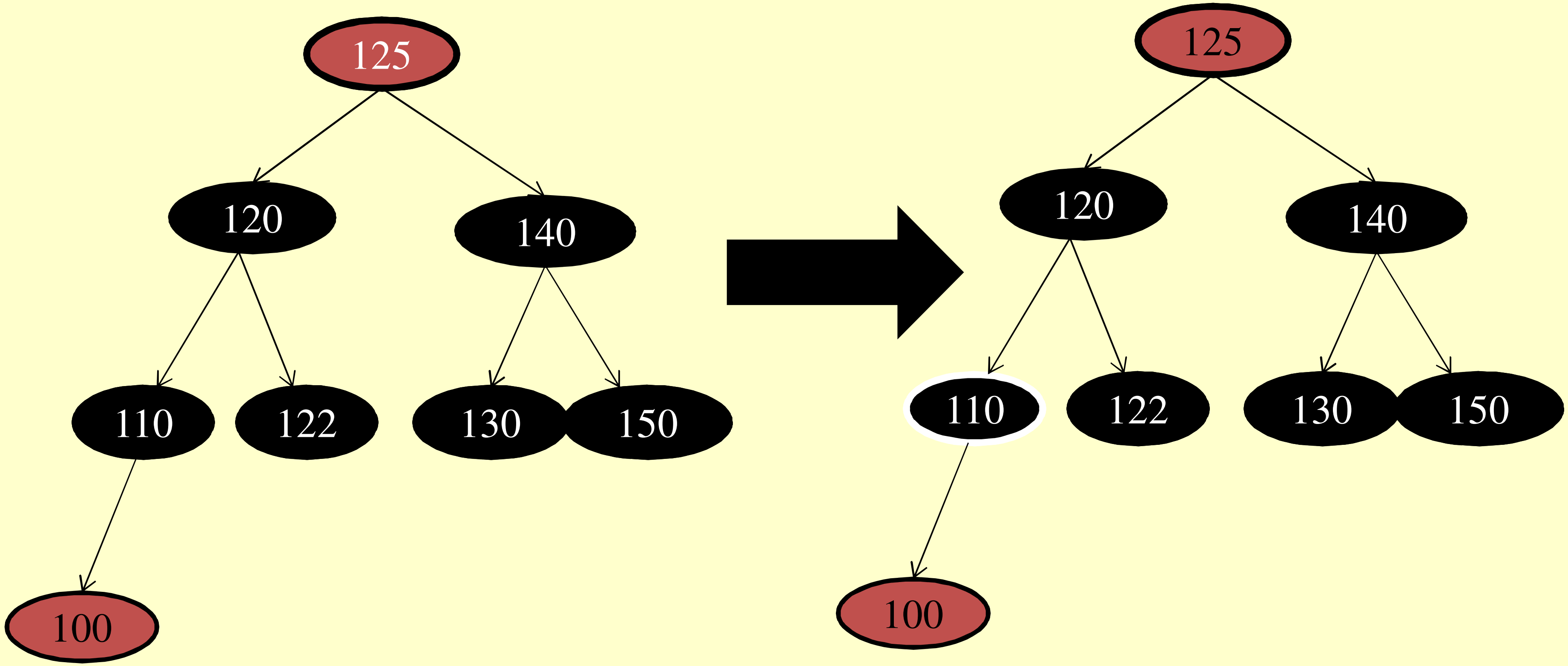
- **Case 3** :If x's sibling w is black and w's left child is red and w's right child is black then exchange the color of w and its left child . After charging the colors perform right rotation on w .



- **Case 4**: If x's sibling w is black and w's right child is red then make the color changes and then perform left rotation on x parent . Now remove the extra black on x.



Example Of Deletion of Root node



<<AVL TREE>>

Contents

- Introduction to AVL Tree
- Operation on AVL Tree

AVL Tree

- **Height Balanced :** A binary search tree is said to be height balanced tree if the nodes of the tree are organized in such a way that the difference in heights of the left subtree and right subtree of any node in the tree is less than or equal to one.

- **Unbalanced :** If the difference in heights of the left subtree and right subtree of any node in the Binary search tree becomes more than 1 then tree is said to be **unbalanced**.

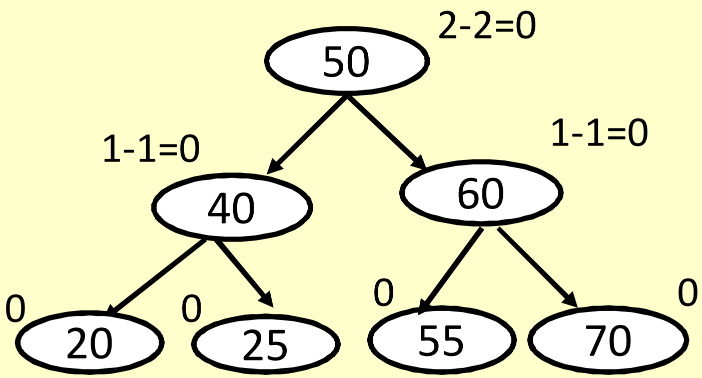
AVL Tree (cont.)

- The ***bf*** of a node will be -***ve*** if the height of its left subtree is less than the height of the right subtree.
- The ***bf*** of a node will be **0** if the height of its left subtree is equal to the height of its right subtree.
- The ***bf*** of a node will be +*ve* if the height of its left subtree is larger than the height of its right subtree.
- In the nutshell,

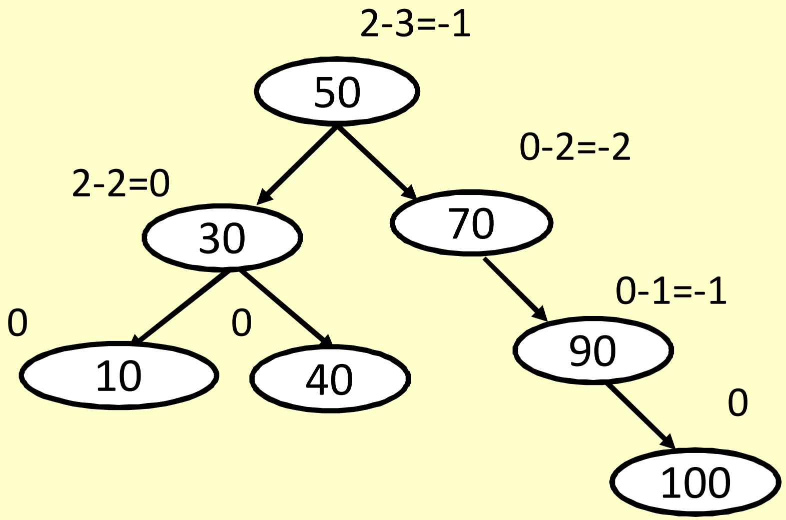
$$bf = \begin{cases} -ve \text{ if } H_L < H_R \\ 0 \text{ if } H_L = H_R \\ +ve \text{ if } H_L > H_R \end{cases}$$

cont…

•The following example shows some binary search tree which are balanced i.e. tree as all the nodes in these tree have ***bf*** = 1 or -1 or 0.



Balanced Tree



Unbalanced Tree

Various Operations on AVL Tree

- The main operations which are commonly applied on any data structure are also applied to AVL tree. These operations are:
  - Searching
  - Insertion
  - Deletion

Insertion

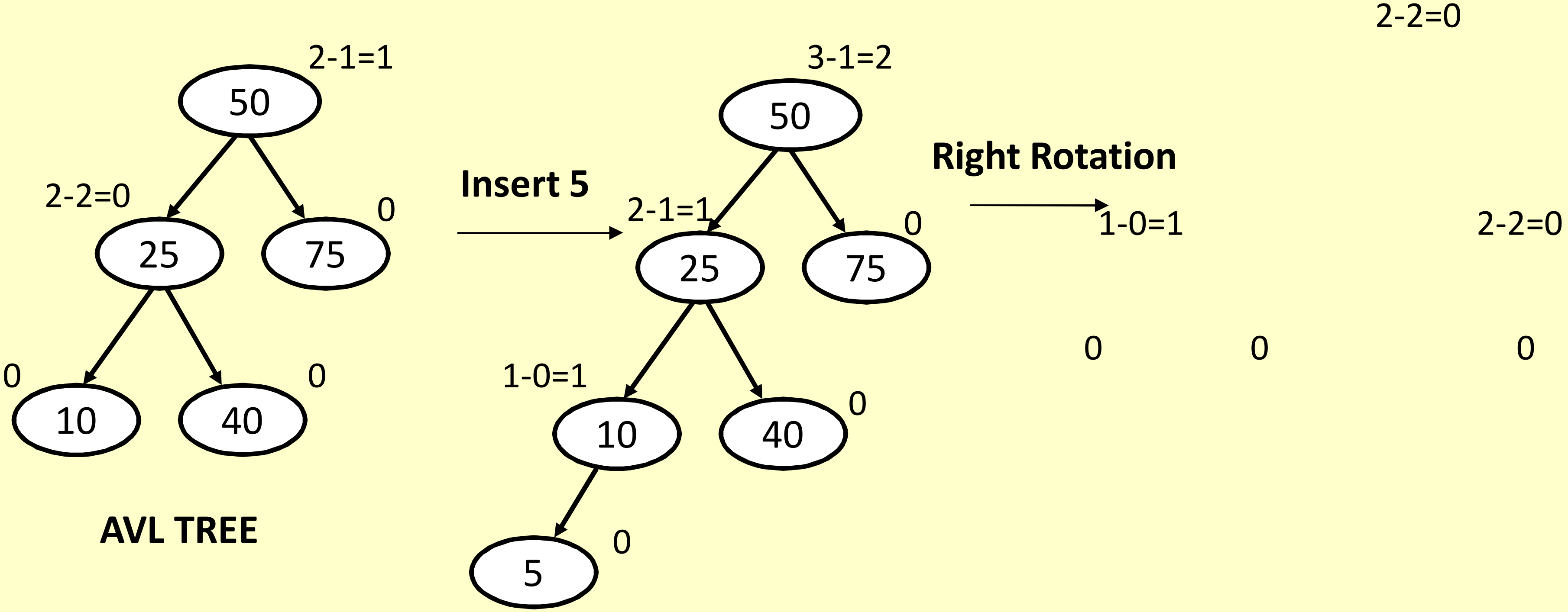
- Insertion of an element in AVL tree is performed in the similar fashion as in the case of BST(Binary search tree).
- That is if the new element is **smaller** than the root element, it is **inserted into the left subtree** else it is **inserted into the right subtree.**
- That is insertion of new node in AVL tree may cause the balance factor of a node in the tree to change to less than -1 or more than 1.

Insertion(cont.)

- In such case, there is a need to balance the tree so that no node in the tree has balance factor other than -1,0, or 1. This balancing is done using rotations.
- This marked node is known as **pivot node**.
- Based upon the position of the newly inserted node, there are types of rotations.
    - Left-Left Rotation
    - Right-Right Rotation
    - Left-Right Rotation
    - Right-Left Rotation

Left - Left Rotation

• When the new node is to be inserted in the left subtree of left child of pivot node P the left-left rotation is performed.



AVL TREE

Pivot Node P is 50
Not a AVL tree

AVL tree after Left Rotation

Left - Left Rotation(Algorithm)

LLRotation(Root , P)

Step 1: If **P → Parent = Null** Then *//If Pivot node is the root node*

Root = P →Left

Else If **P → Parent → Left = P** *//If Pivot node is left child*

P →Parent → Left = P →Left

Else *// If Pivot node is right child*

P → Parent → Right= P →Left

[End If]

Step 2: Temp = P → Left → Right

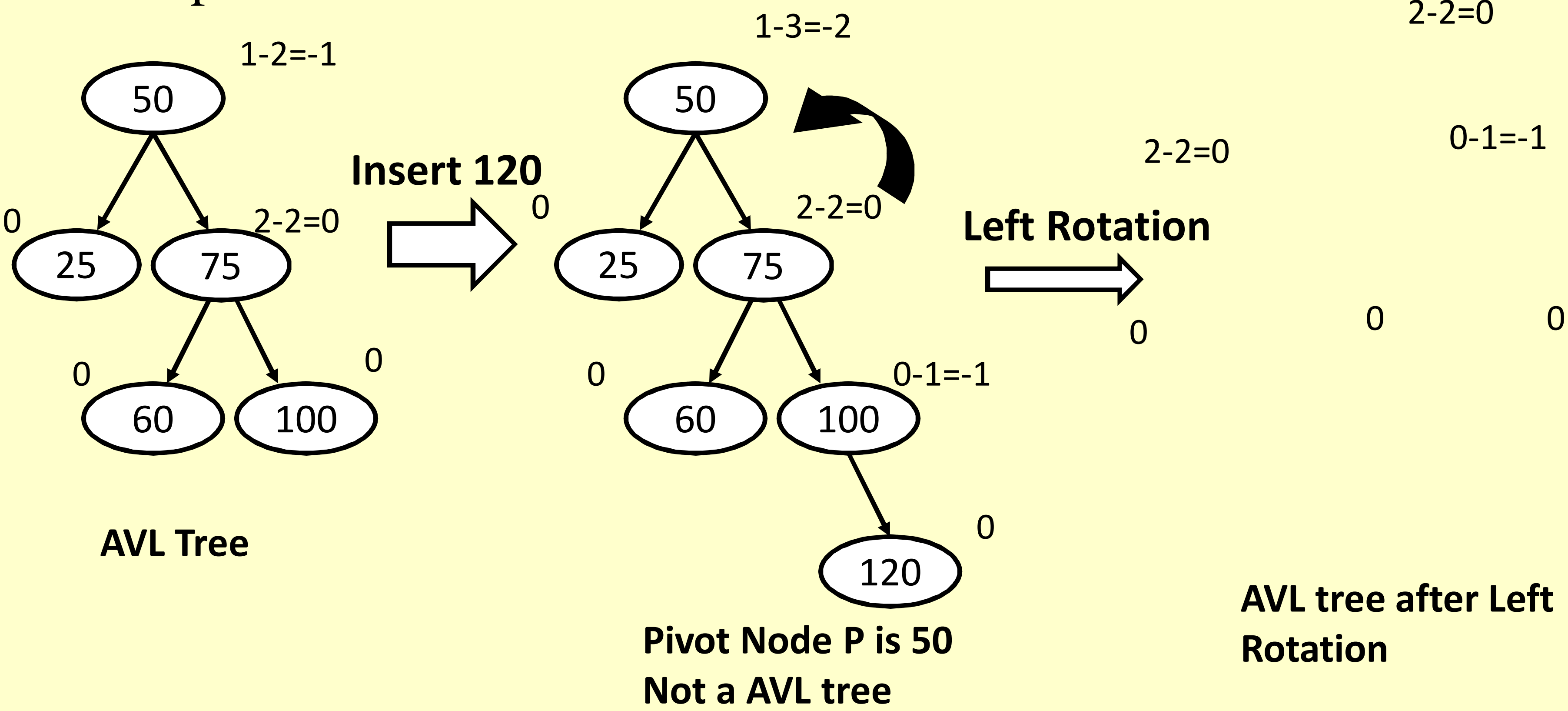
Step 3 : P → Left → Right = P

Step 4 : P → Left = Temp

Step 5 : Return

Right - Right Rotation

- When the new node is to be inserted in the **right subtree of right child of pivot node P** then right-right rotation is performed.



Right- Right Rotation(Algorithm)

RRRotation (Root,P)

Step 1: If **P → Parent = Null** Then // *If Pivot node is the root* : *node*

Root = P   Right

Else If **P → Parent → Left = P** *//if Pivot node is left child*

P → Parent->Left = P → Right

Else *//if Pivot node is right child*

P → Parent-> Right = P → Right

[End If]

Step 2: Temp = P → Right → Left

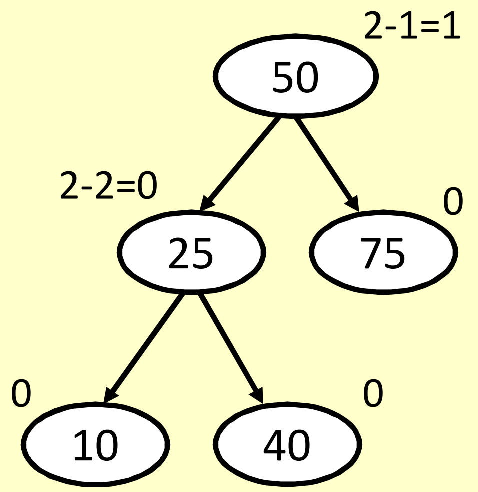
Step 3: P → Right → Left = P

Step 4: P → Right =Temp

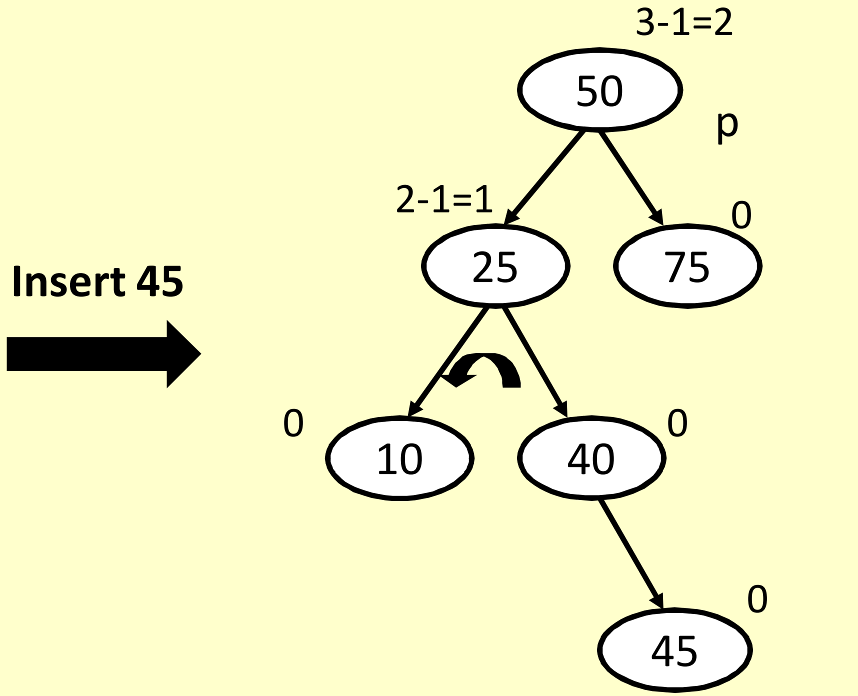
Step 5: Return

Left - Right Rotation

- When the new node is to be inserted in the **right subtree of the left child of pivot node P** then Left-Right rotation is performed.



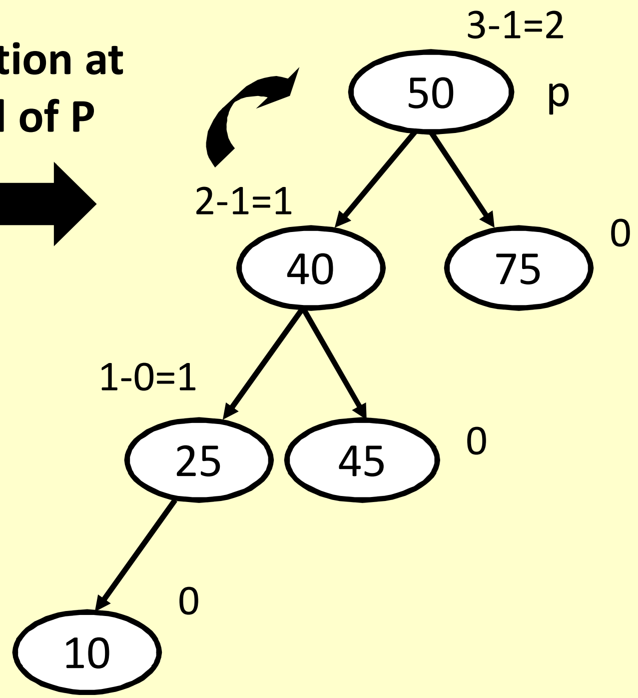
AVL Tree



Pivot Node P is 50
Not a AVL tree

cont..

Left rotation at Left child of P



Right rotation

2-2=0

1-0=1

2-2=0

0

0

0

Pivot Node P is 50
Not a AVL tree

AVL tree after right Rotation

Left –Right Rotation(Algorithm)

LRRotation(Root , P)

Step 1: If **P → Parent = Null** Then *//If Pivot node is the root node*

Root = P → Left → Right

Else If **P → Parent → Left = P** *//If Pivot node is left child*

P → Parent → Left = P → Left → Right

Else *//If Pivot node is right child*

P → Parent → Right = P → Left → Right

[End If]

Step 2: **P → Left = P → Left → Right → Right**

Step 3: P → Left → Right → Right = P

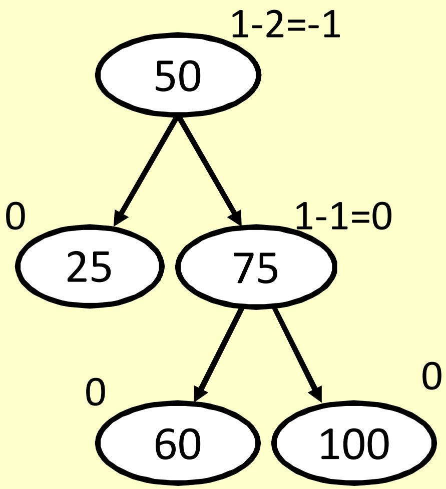
Step 4: P → Left → Right → Left = P → Left

Step 5: P → Left → Right = P → Left → Right->Left

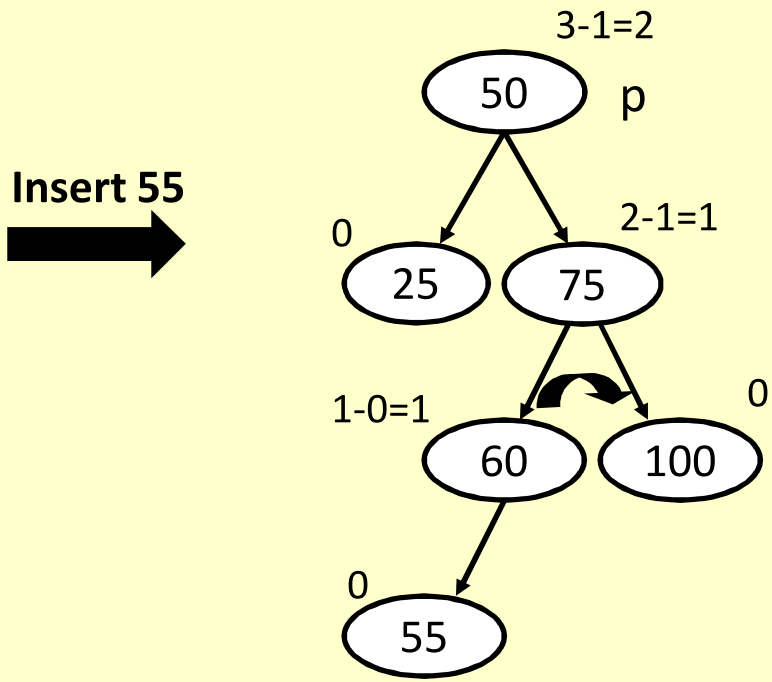
Step 6: Return

Right - Left Rotation

- When the new node is to be inserted in the **left subtree of the right child of pivot node P** the Right - Left rotation is performed.



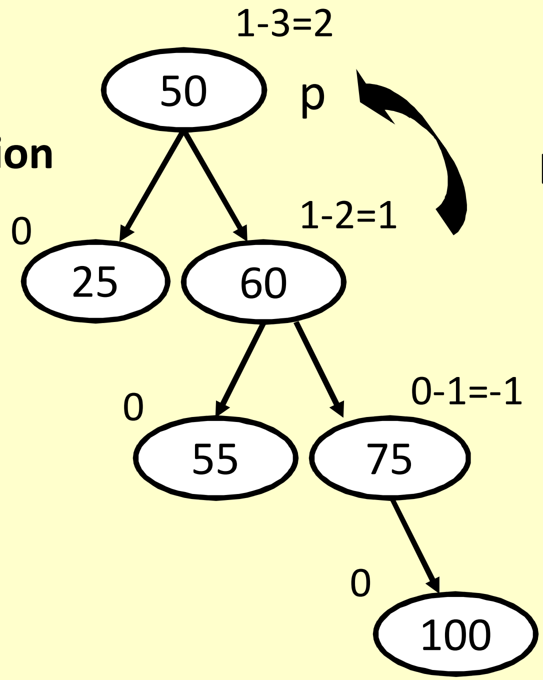
AVL Tree



Pivot Node P is 50
Not a AVL tree

cont..

Right Rotation



Pivot Node P is 50
Not a AVL tree

Left rotation

2-2=0

2-2=0

1-0=1

0

0

0

AVL tree after left Rotation

Right - Left Rotation(Algorithm)

RLRotation(Root,P)

Step 1: If **P → Parent = Null** Then *//If Pivot node is the root node*

Root = P → Right → Left

Else If **P → Parent → Left = P** *//If Pivot node is left child*

P → Parent → Left = P → Right → Left

Else *//If Pivot node is right child*

P → Parent->Right = P → Right → Left

[End If]

Step 2: Temp = P → Right → Left → Left

Step 3: P → Right → Left → Left = P

Right - Left Rotation (cont.)

Step 4: P → Right → Left → Right = P → Right
Step 5: P → Right → Left = P → Right → Left → Right
Step 6: P → Right = Temp
Step 7: Exit

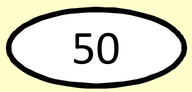
Example

Create AVL Tree with following 10 elements:

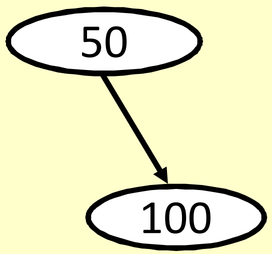
50, 100, 200, 35, 15, 20, 10, 300, 250, 150, 180, 5

In each step, one element will be inserted into AVL tree and at the end of 10th step, a final AVL tree is created.

<u>Step 1: Insert 50</u>

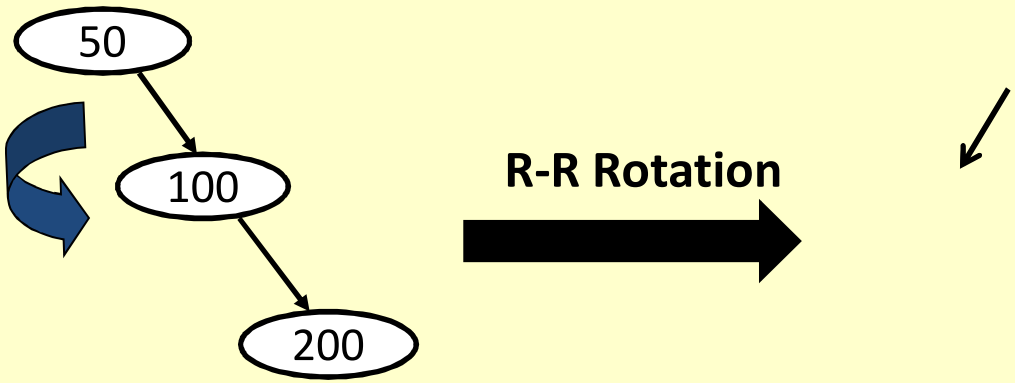


<u>Step 2 : Insert 100</u>

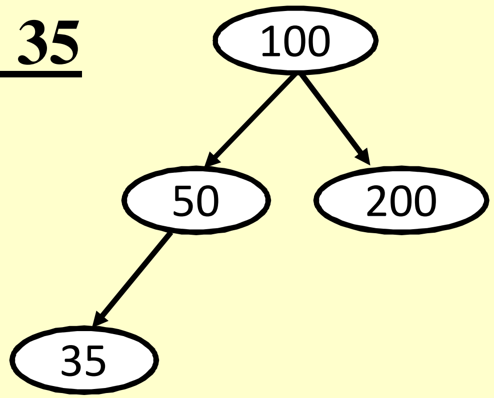


Cont.

<u>Step3:Insert200</u>

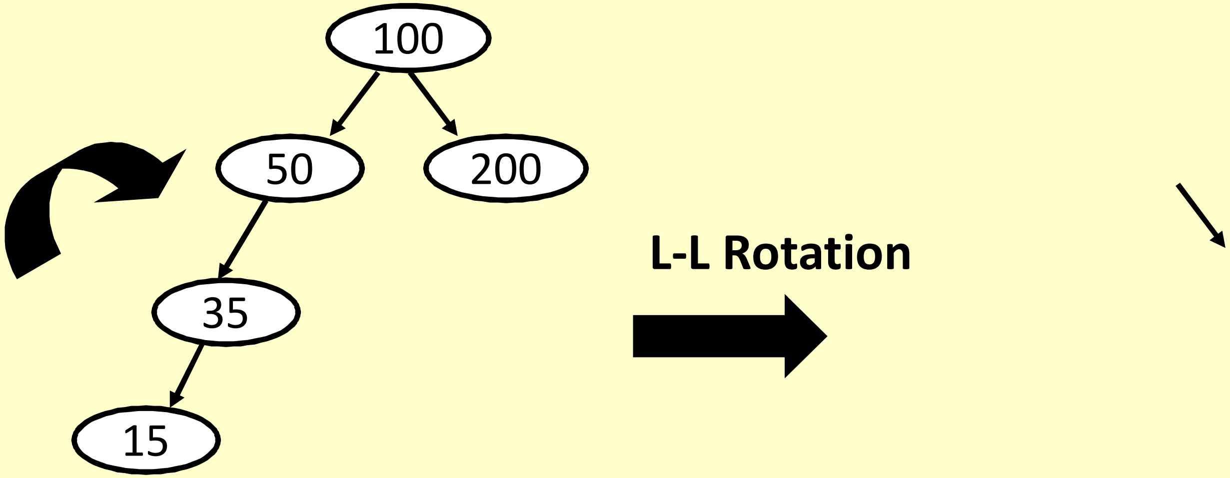


<u>Step4:Insert 35</u>

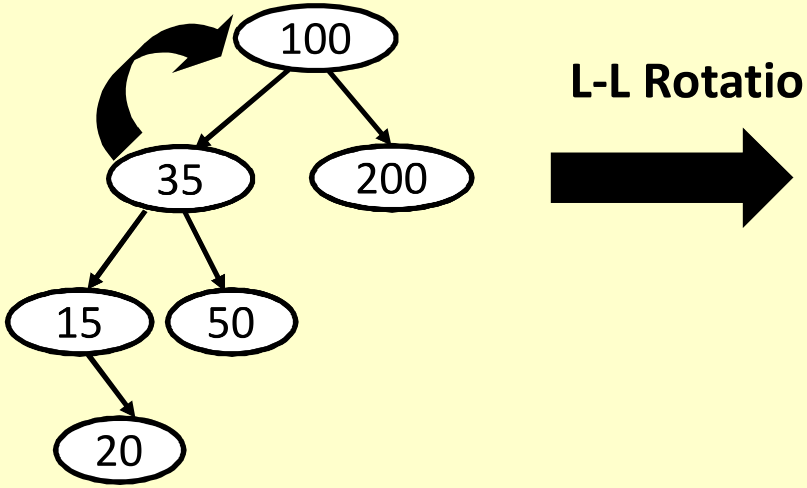


Cont.

Step5 :Insert 15

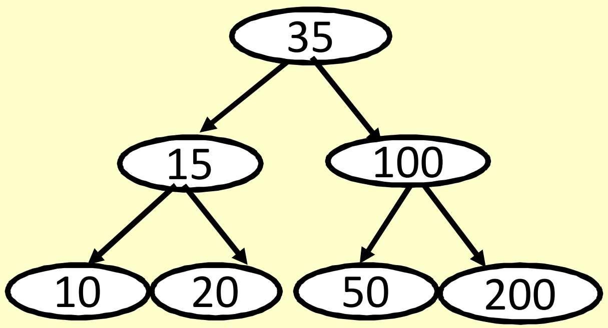


Step6 :Insert 20

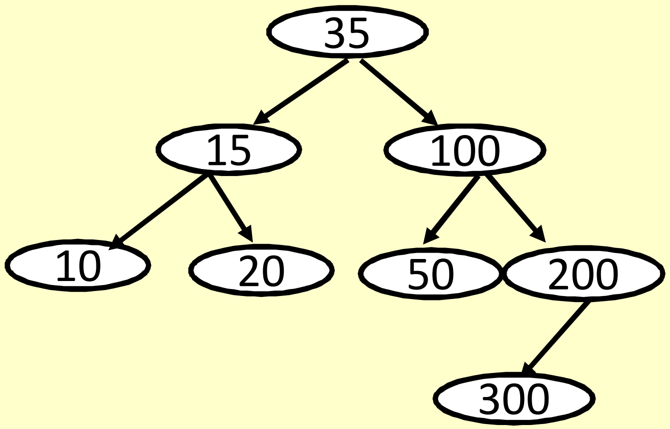


Cont.

Step7: Insert10

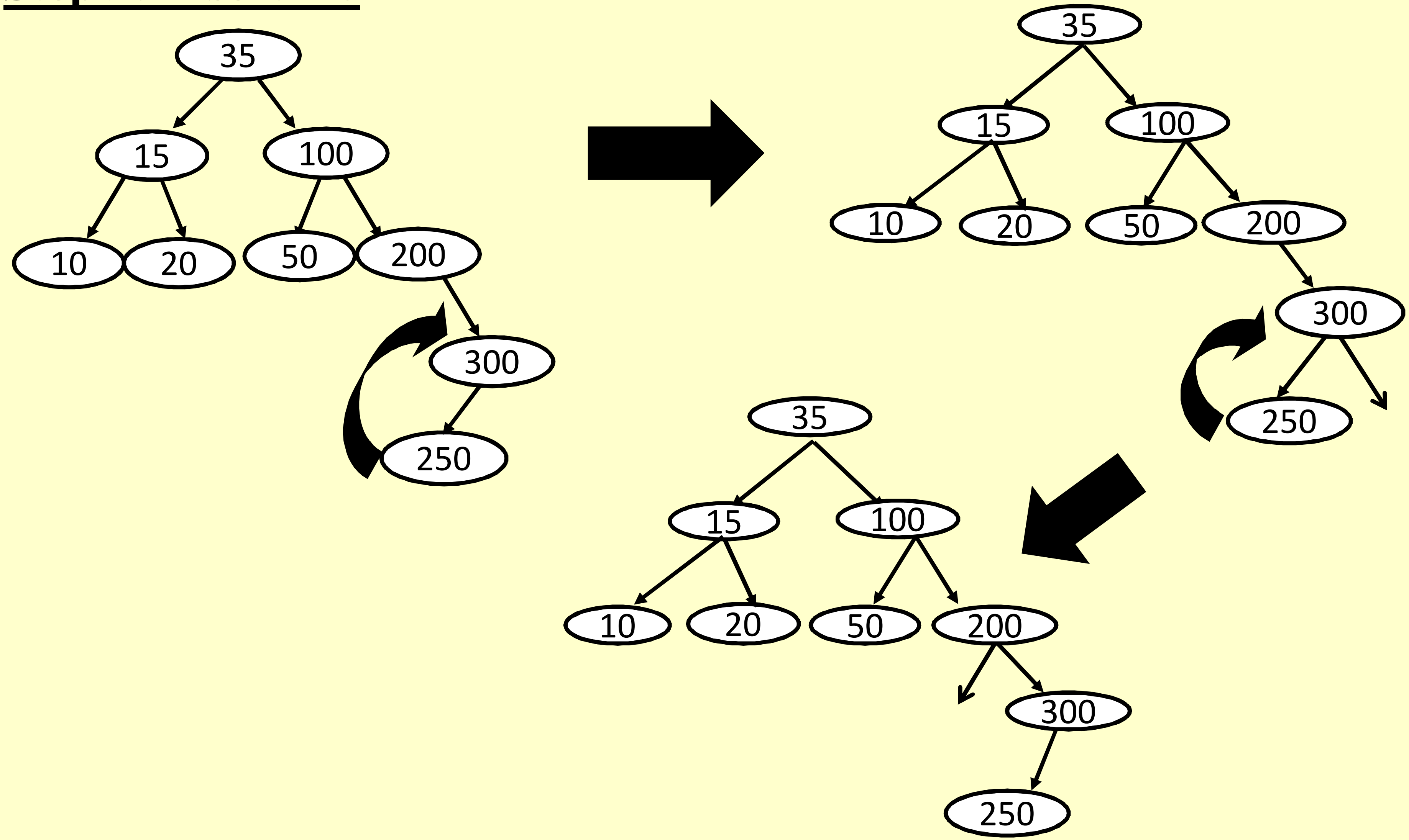


Step 8: Insert300



Cont.

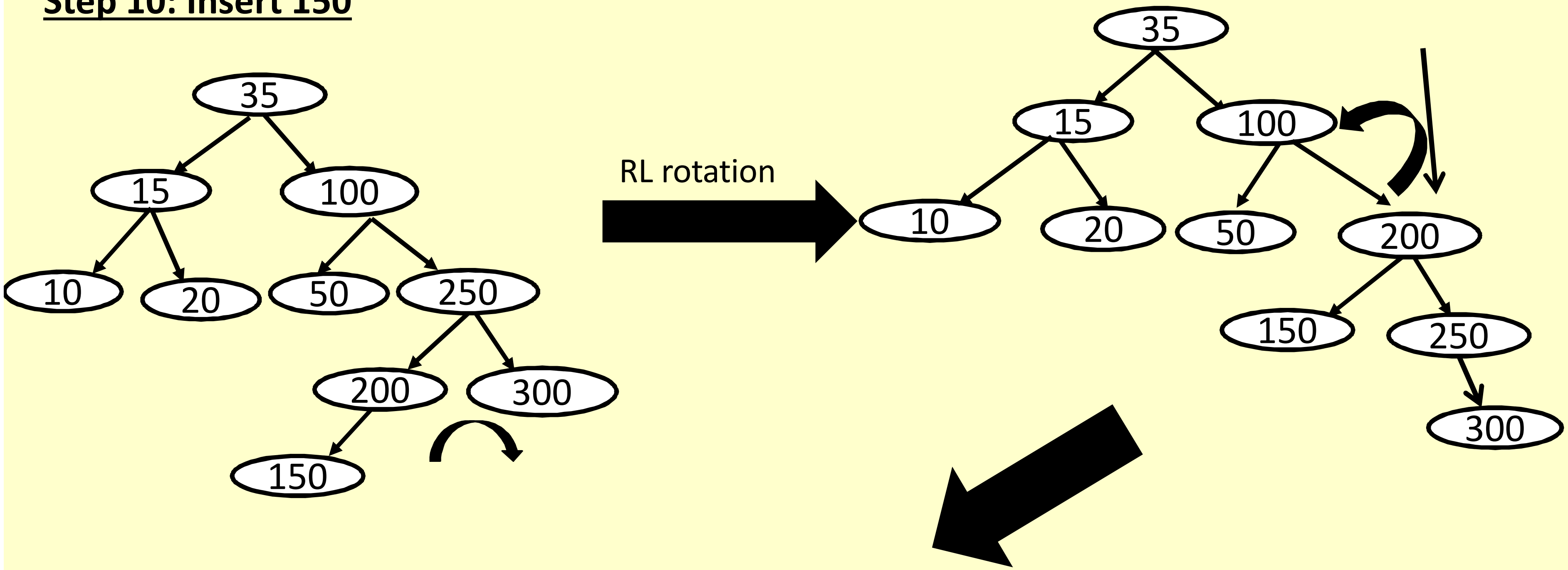
<u>Step 9: Insert250</u>



Cont.

Step 10: Insert 150

After rotation



FINAL AVL TREE

Searching

•The search operation is defined as finding the address of a node containing the desired element. The search operation on AVL tree is applied in the similar manner as it is applied on BST. This is because the AVL tree basically height balanced binary search tree.

• Therefore the complexity of the search operation on AVL tree is ***$o(\log_2 n)$.***

Searching Algorithm

BSTSearch(Root,Item,Position,Parent)

Step1: If **Root=Null** Then

set **Position = Null**

set **Parent = Null**

Return

[End If]

Step 2: **Pointer=Root** And **Pointer P = Null**

Step 3: Repeat Step 4 While **Pointer ≠ Null**

Step 4: If **Item = Pointer→Info** Then

set **Position = Pointer**

set **Parent = PointerP**

Return

Else If **Item<Pointer→Info** Then

Searching(cont.)

Set **PointerP=Pointer**
Set **pointer =Pointer → Left**

Else

Set **PointerP=Pointer**
Set **pointer =Pointer → Right**

[End of IF]

[End of Loop]

Step 5: Set **Pointer =Null and Parent=Null**

Step 6: Return

Deletion

The Deletion of element of an element in AVL tree proceed as in procedures for deletion of an element in a binary search tree. There are different case:

Case 1:"when node having Two child". In this case inorder successor of node replaced its position of the node to be deleted.

Case 2:"when node having 0 or 1 child". In this case deleted node is replaced by its only child node.

Deletion Algorithm

CASE A (INFO,LEFT, RIGHT, ROOT, LOC,PAR)

1. [Initializes **CHILD**]

    if **LEFT →LOC=NULL** and **RIGHT → LOC=NULL**, then

        Set **CHILD =NULL**;

    Else if **LEFT → LOC ≠ NULL** , then

        Set **CHILD= LEFT → LOC**

    Else

        Set **CHILD=RIGHT → LOC**

    [End of if structure]

2. If **PAR≠ NULL** ,then

        If **LOC = LEFT → PAR** then

            Set **LEFT → PAR =CHILD**

Cont.

Else:

Set **RIGHT → PAR=CHILD**.

[End of if structure]

Else:

Set **ROOT=CHILD**

[End of if structure]

3.RETURN

Cont.

CASE B(INFO,LEFT,RIGHT,ROOT,LOC,PAR)

1:[Find **SUC** and **PARSUC**]

(a)Set **PTR = RIGHT → LOC** and **SAVE=LOC**.

(b)Repeat while **LEFT → PTR ≠ NULL.**

Set **SAVE =PTR** and **PTR=LEFT → PTR**

[end of loop.]

2: [Delete inorder succesor]

Call **CASE A(INFO,LEFT,RIGHT,ROOT,SUC,PARSUC)**.

3: [Replace node **N** by its in order successor.]

(a)If **PAR ≠ NULL** , then

If **LOC =LEFT → PAR,** then

Set **LEFT → PAR=SUC**

Cont.

Else
Set **RIGHT → PAR=SUC**.
[End of If structure]
Else:
Set **ROOT=SUC.**
[End of If structure.]
(b) Set **LEFT → SUC=LEFT → LOC** and
RIGHT → SUC= RIGHT → LOC
4: RETURN.

Cont.

DEL(INFO,LEFT,RIGHT,ROOT,LOC,AVAIL,ITEM)

1: [Find the location of ITEM and its parent ,using procedure 7.4]
Call **BSTSearch (INFO,LEFT,RIGHT,ROOT,LOC,PAR,ITEM)**

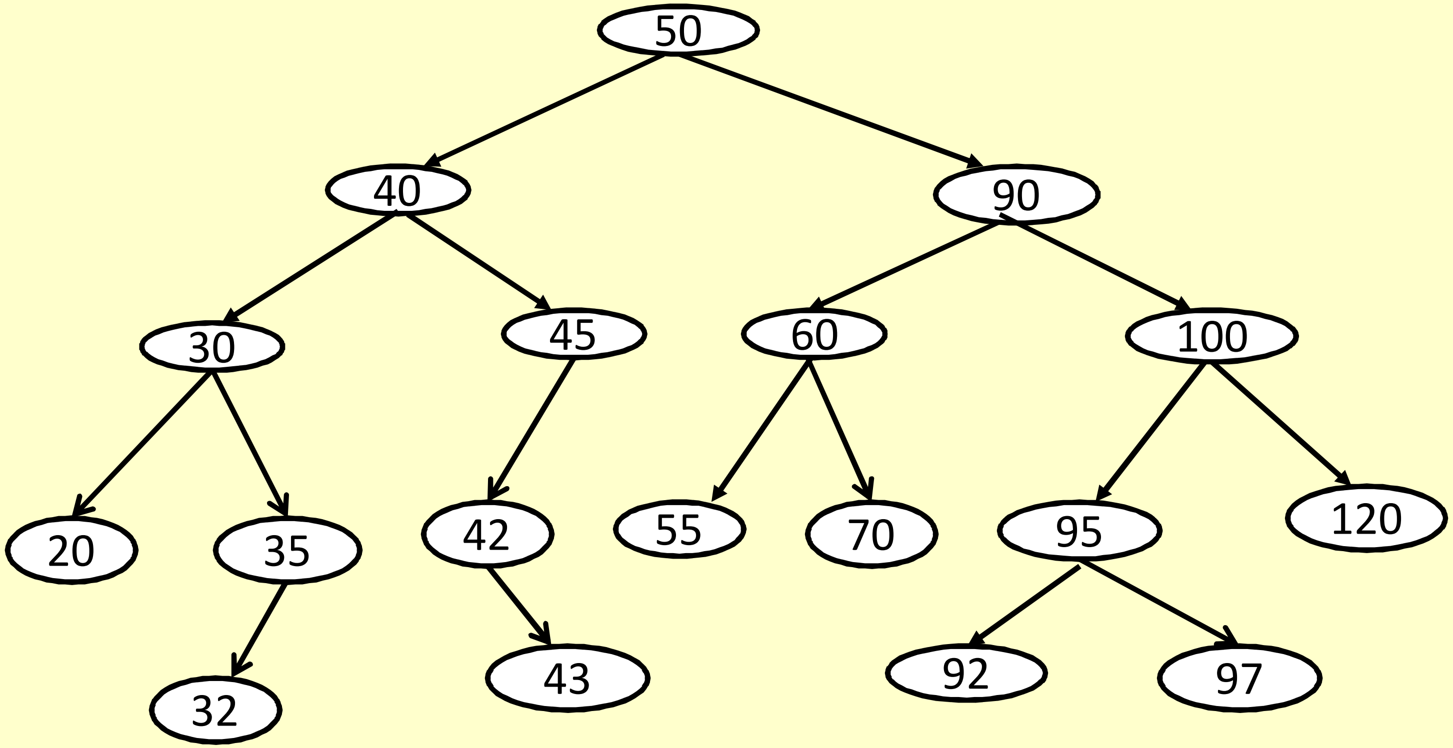
2: [**ITEM** in Tree ?]
If **LOC =NULL** , then,
write: **ITEM** not in tree
exit

3: [Delete node containing ITEM]
If **RIGHT → LOC ≠ NULL and LEFT → LOC ≠ NULL**, then
Call **CASE B(INFO,LEFT,RIGHT,ROOT,LOC,PAR)**:
Else:
Call **CASE A (INFO,LEFT, RIGHT, ROOT, LOC,PAR)**
[End of if structure]

4: [Return deleted node to the AVAIL list.]
Set **LEFT → LOC=AVAIL and AVAIL=LOC.**

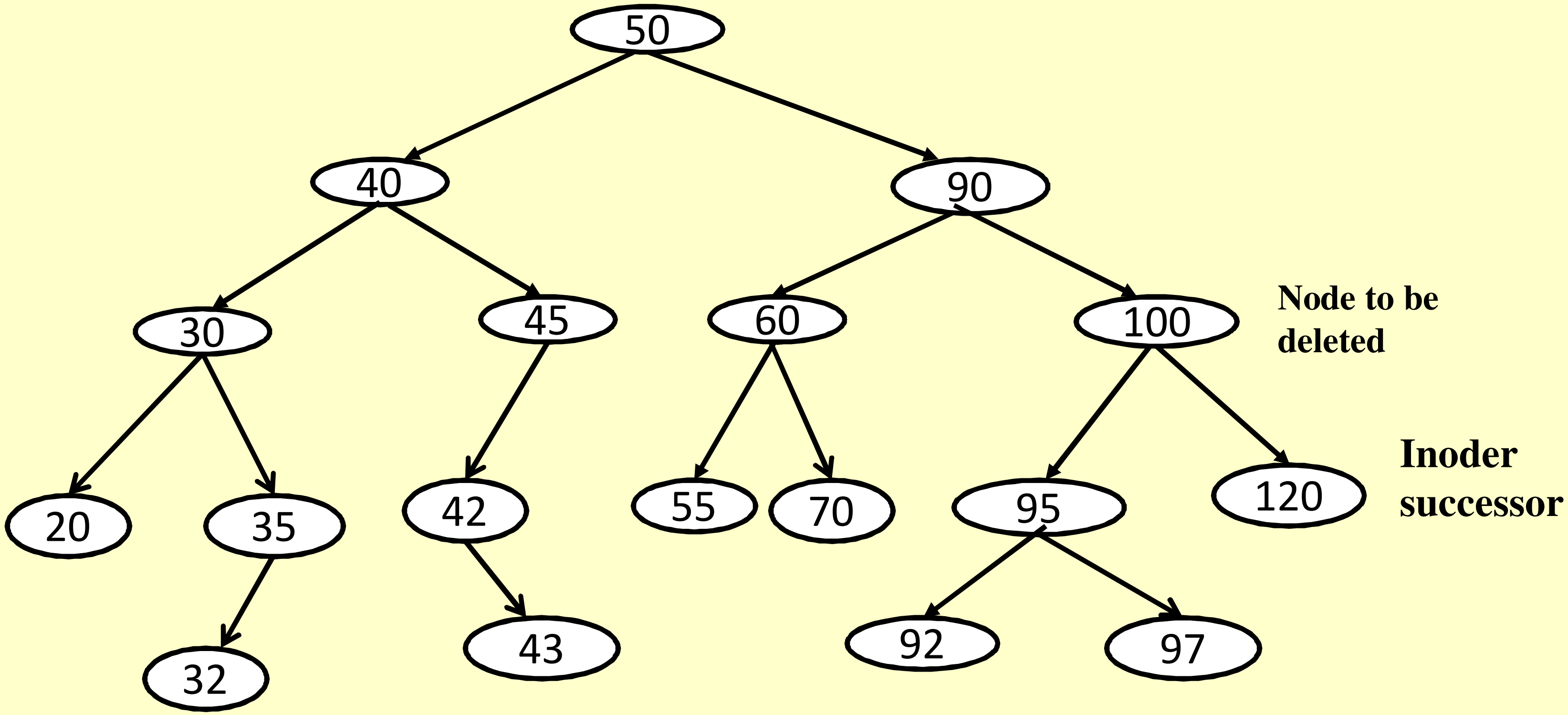
5:EXIT

Example



Deletion Of Node Having Two Child:

Case 1: Deletion Of Node “100” From Tree:



Deletion Of Node Having 0 or 1 Child:

Case 2: Deletion Of Node “45” From Tree: