AVL-Trees

Balanced binary tree

- The disadvantage of a binary search tree is that its height can be as large as N-1
- This means that the time needed to perform insertion and deletion and many other operations can be O(N) in the worst case
- We want a tree with small height
- A binary tree with N node has height at least Θ(log N)
- Thus, our goal is to keep the height of a binary search tree O(log N)
- Such trees are called balanced binary search trees. Examples are AVL tree, red-black tree.

AVL (Adelson Velskii and Landis) tree

- An AVL tree is a binary search tree in which
  - for *every* node in the tree, the height of the left and right subtrees differ by at most 1.

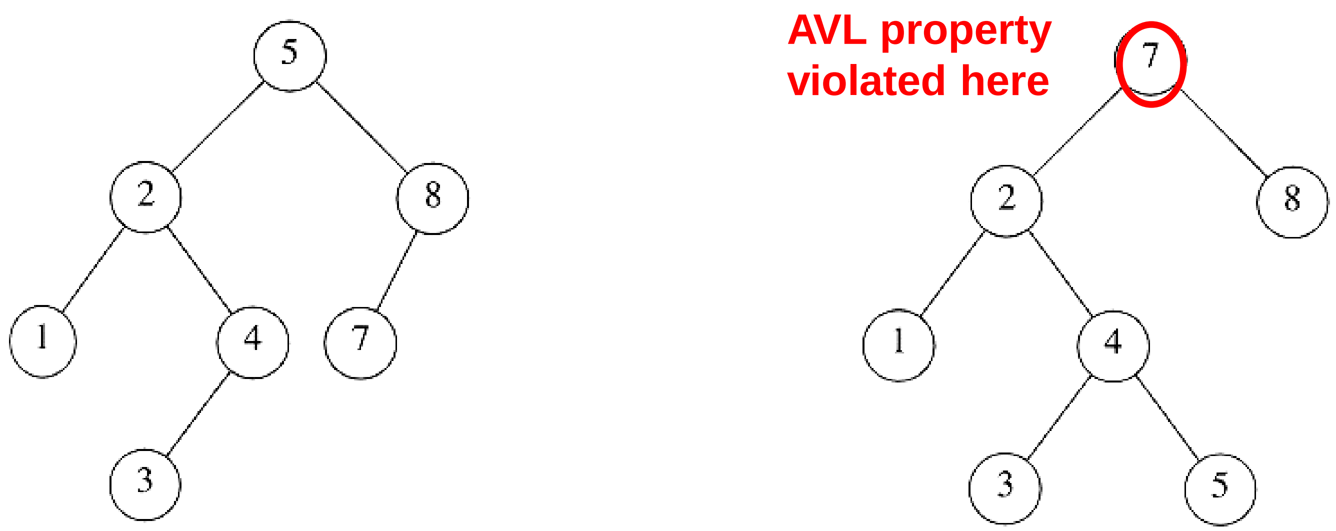


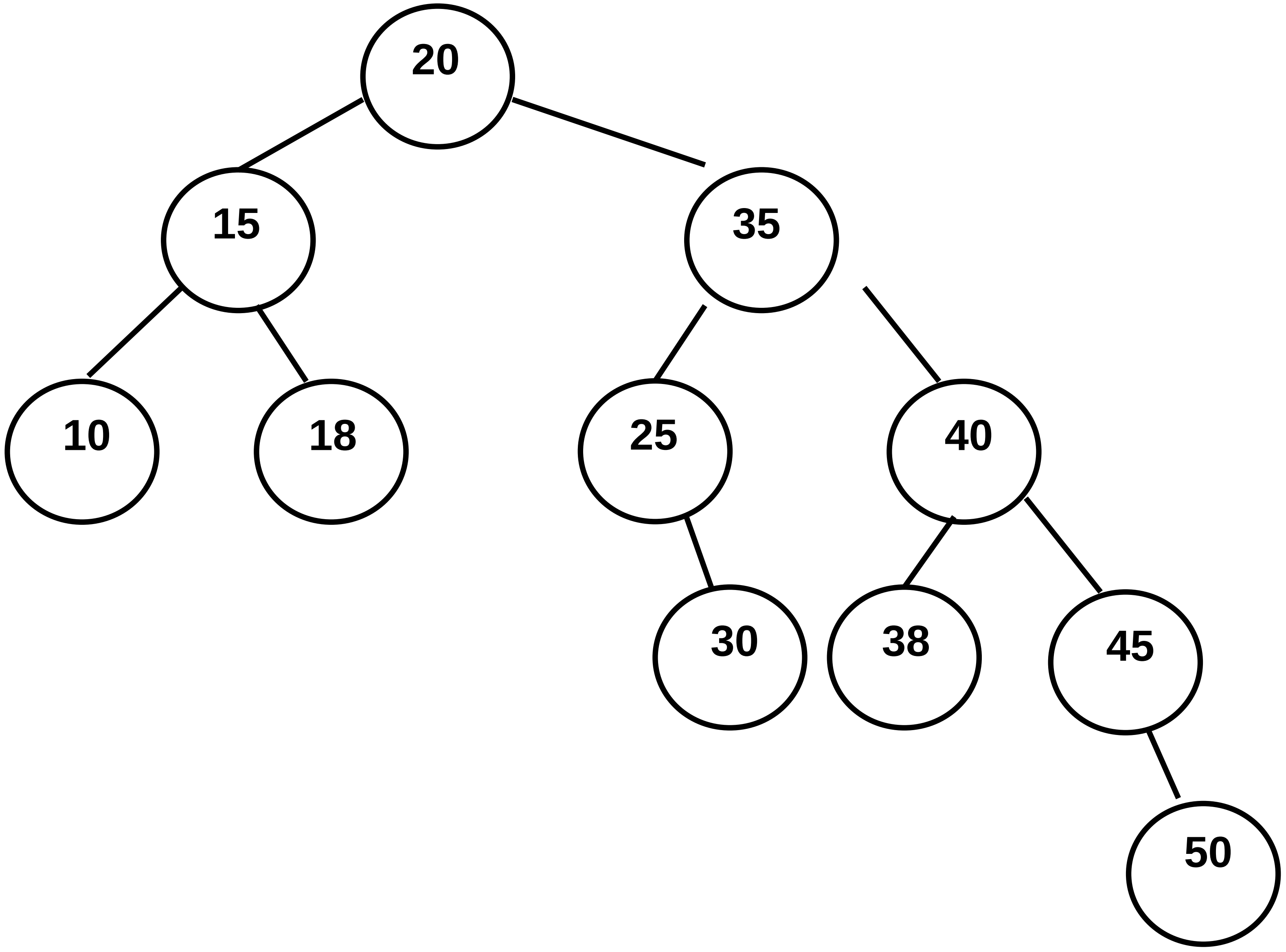
Figure 4.32 Two binary search trees. Only the left tree is AVL.

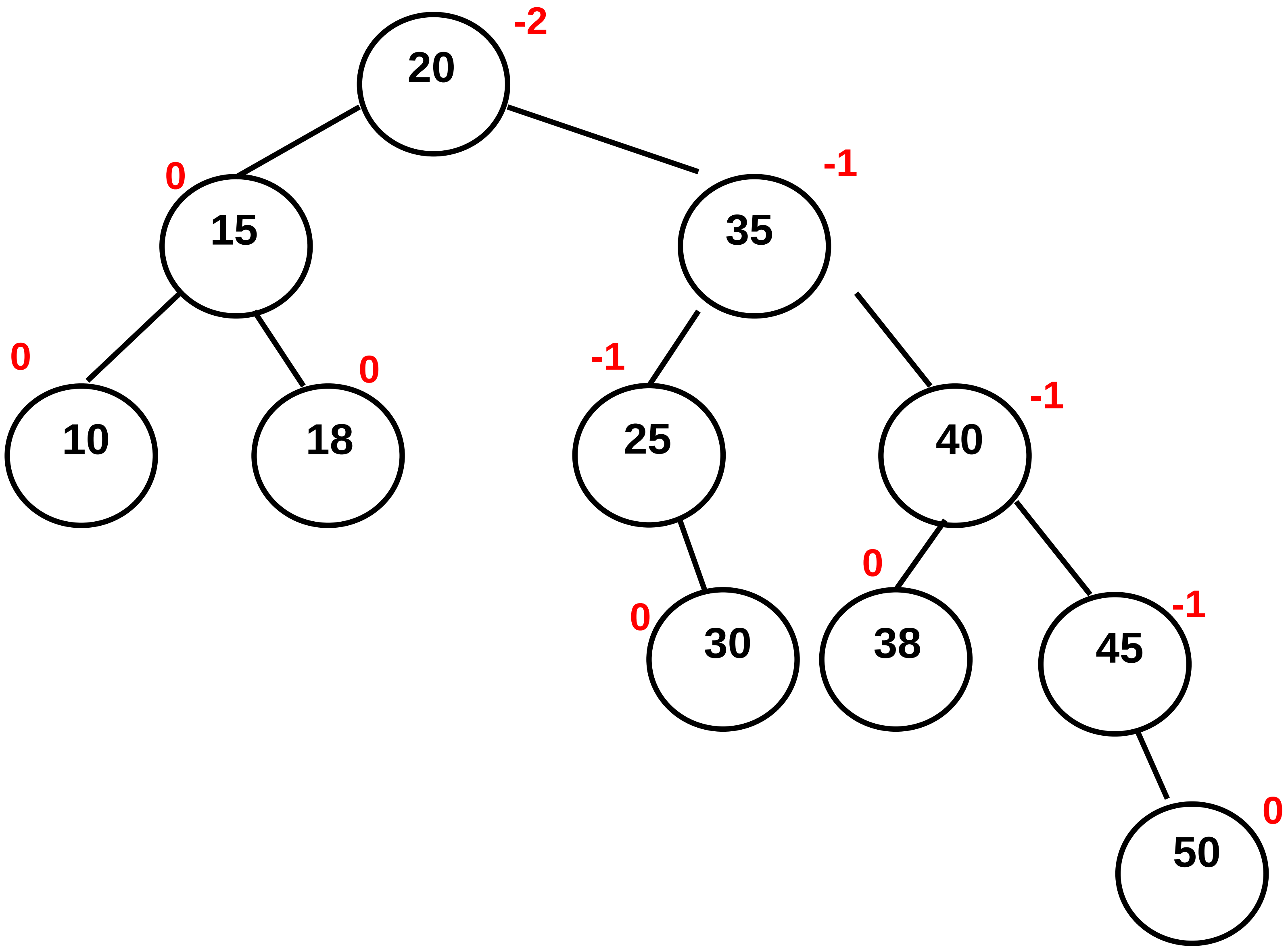
Balance Factor

- A Balance Factor of a node will be defined as height of left subtree of the node minus the height of right subtree of the node.
- B.F of a node

  = ht(leftsubtree) – ht(rightsubtree)

For AVL Trees , the B.F which is measure of imbalance , should be in the range of -1 , 0 , or 1.

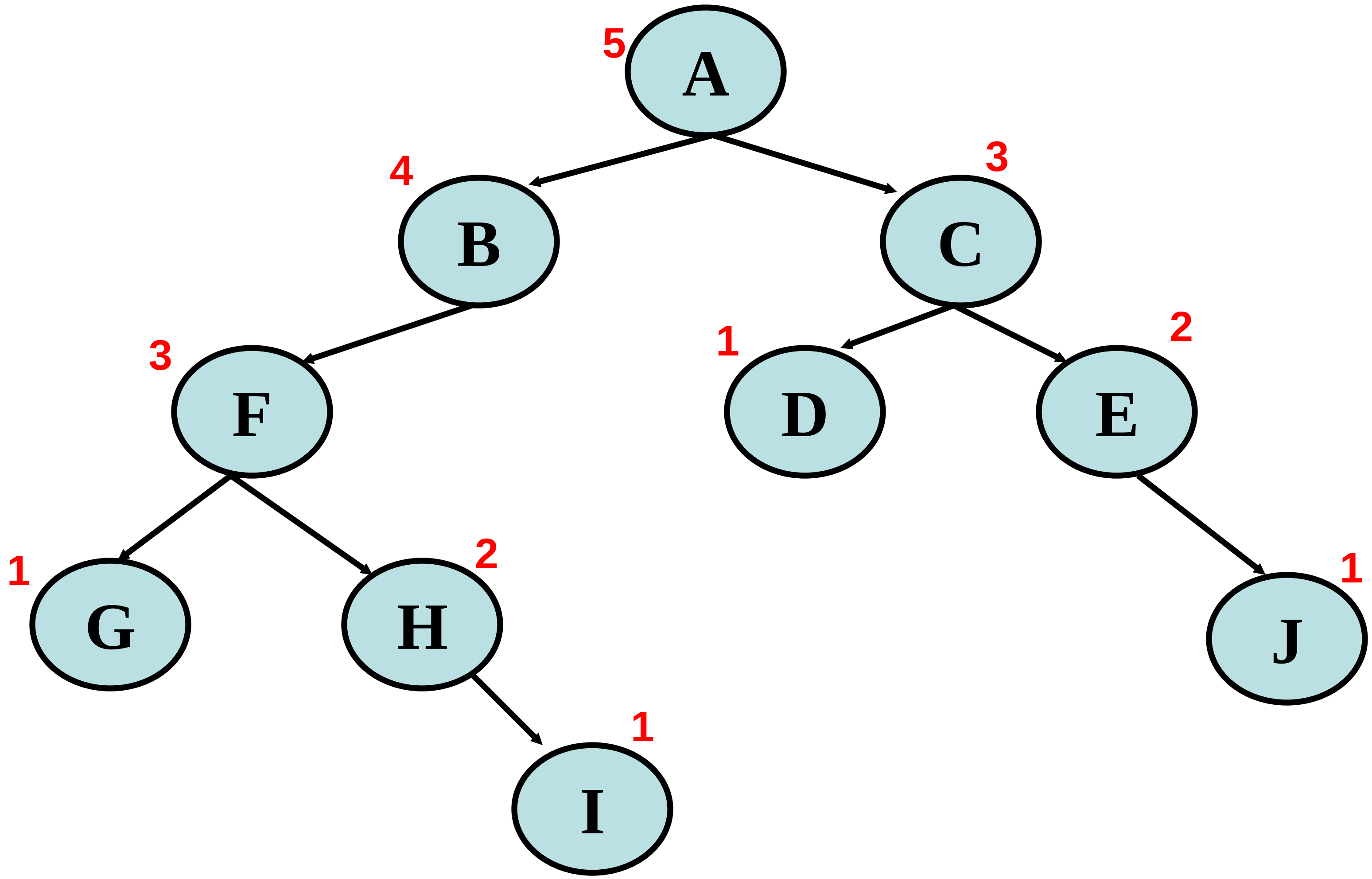

20
15
35
10
18
25
40
30
38
45
50


20
-2
15
0
35
-1
10
0
18
0
25
-1
40
-1
30
0
38
0
45
-1
50
0

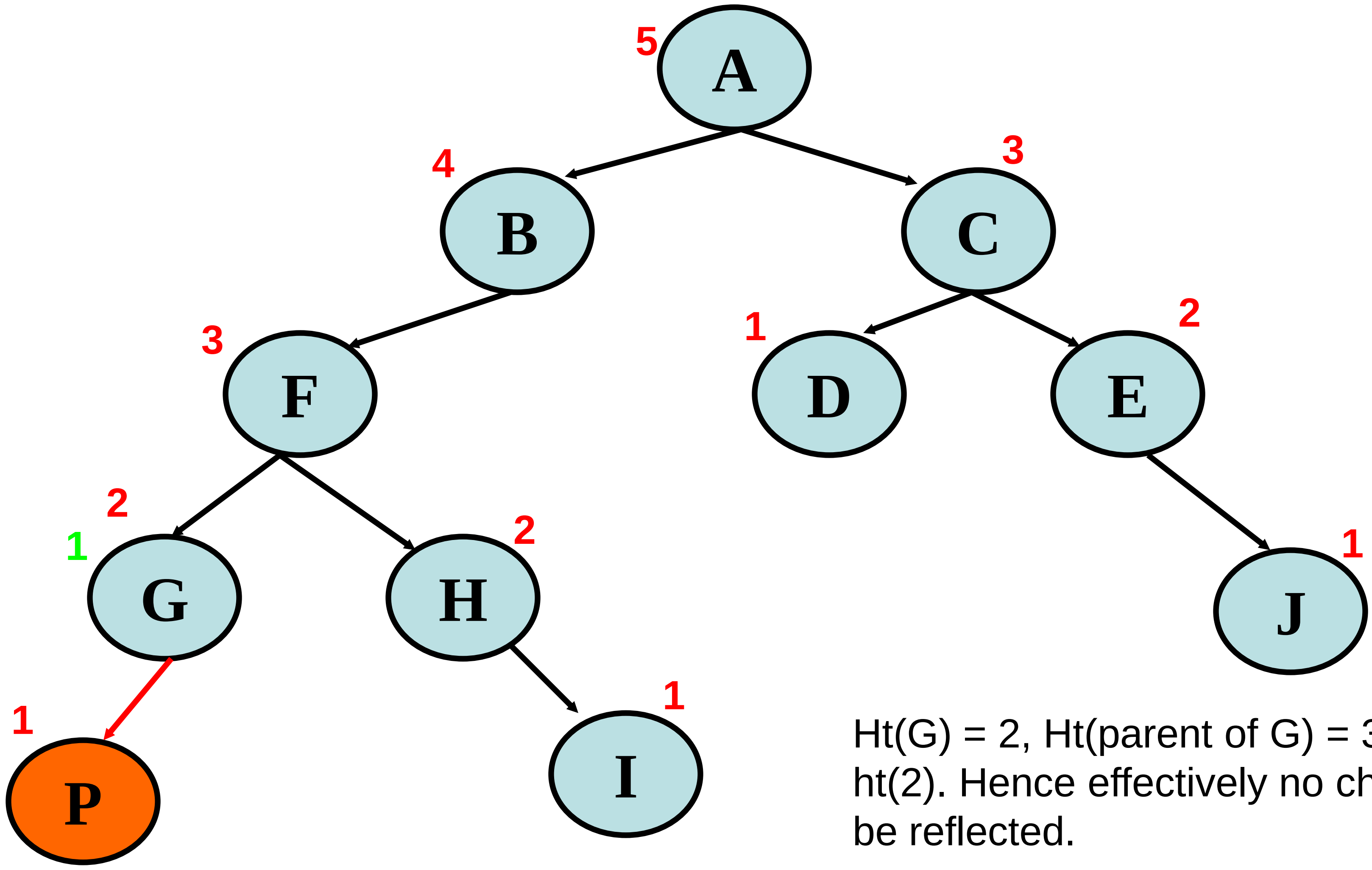
Implementation (Data Structure for AVL Tree)

- Extra Field in the structure of the tree itself for storing the height so that during creation only we can assign the value to that field and may increment it so that finally we get the height in each node.
- Check the feasibility of this idea.

* P is to be added as left child of G

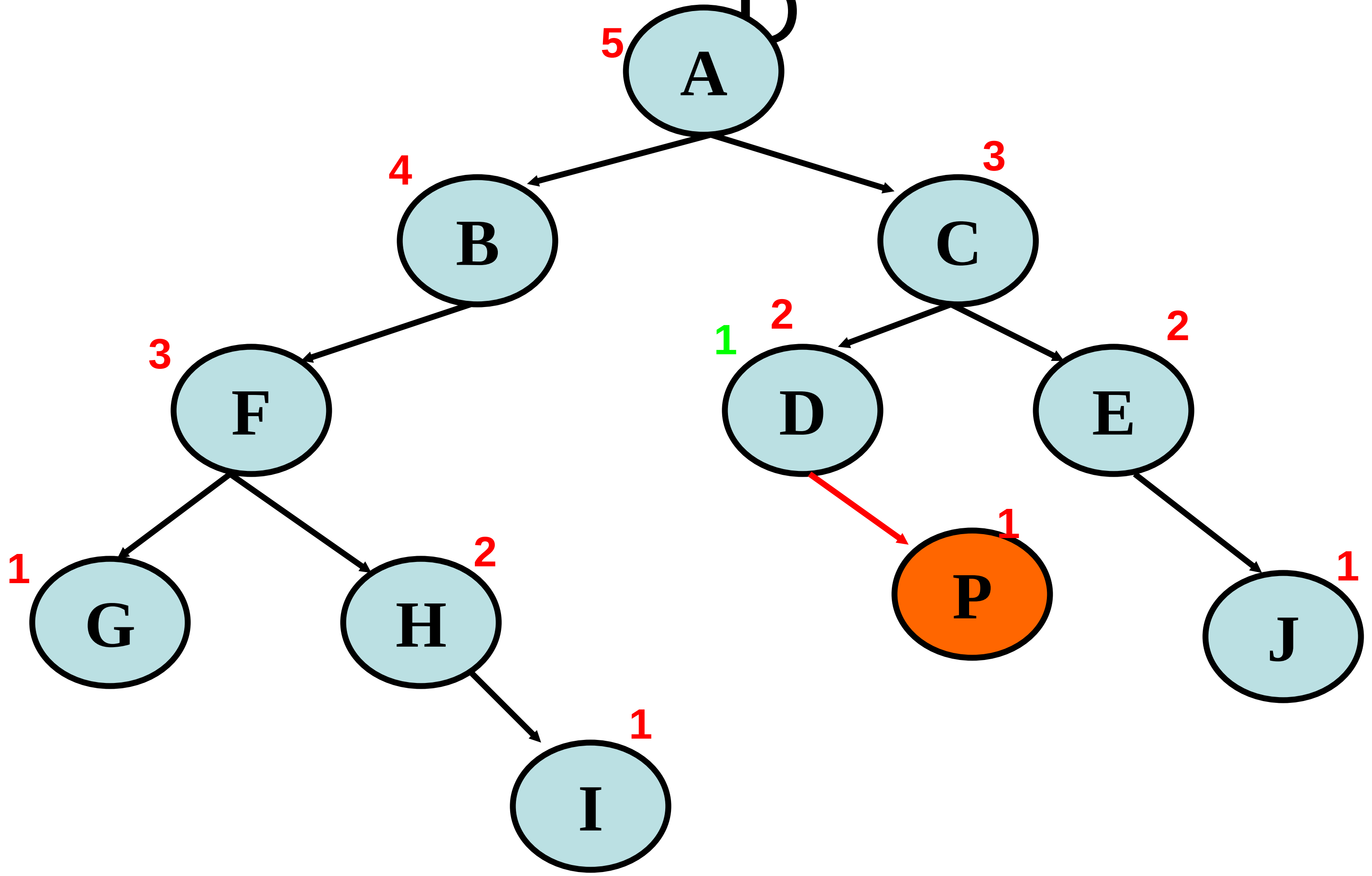


* P is to be added as left child of G



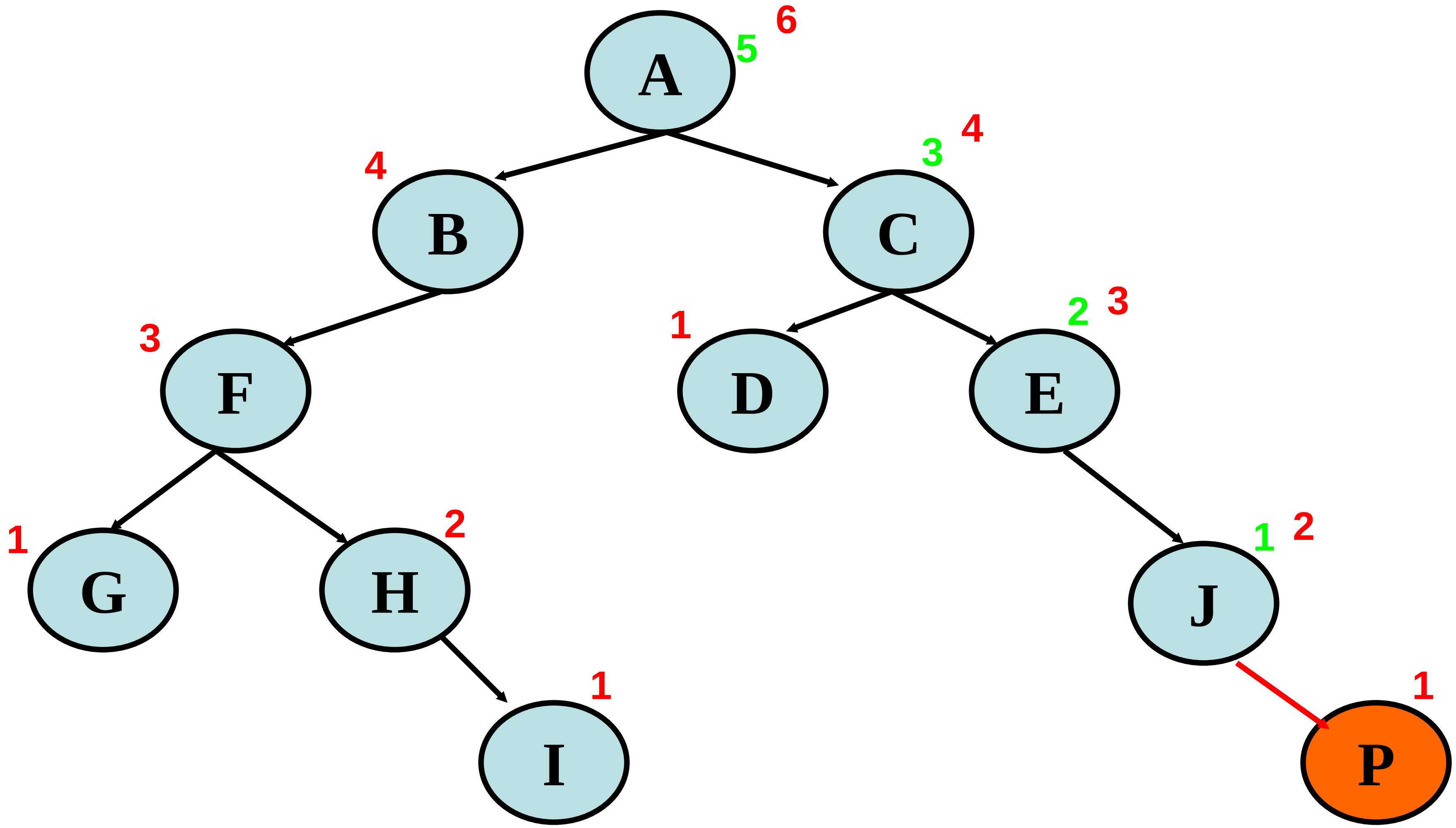
Ht(G) = 2, Ht(parent of G) = 3 > new ht(2). Hence effectively no change will be reflected.

* P is to be added as right child of D



Ht(D) = 2, As new ht of D is smaller than its parent no other change.

* P is to be added as left child of J

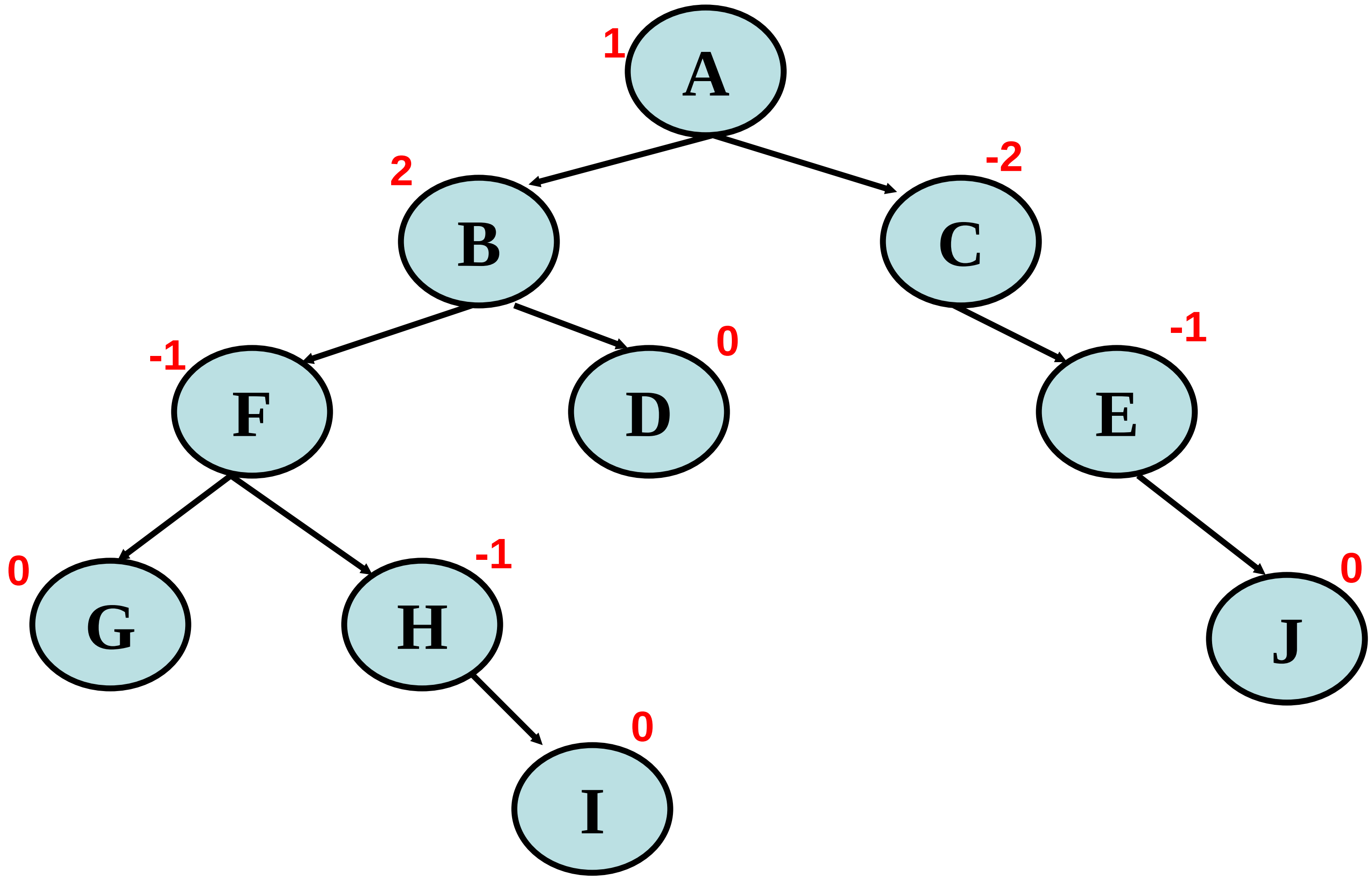


Parent height will be atleast 1 more than that of the child’s height..

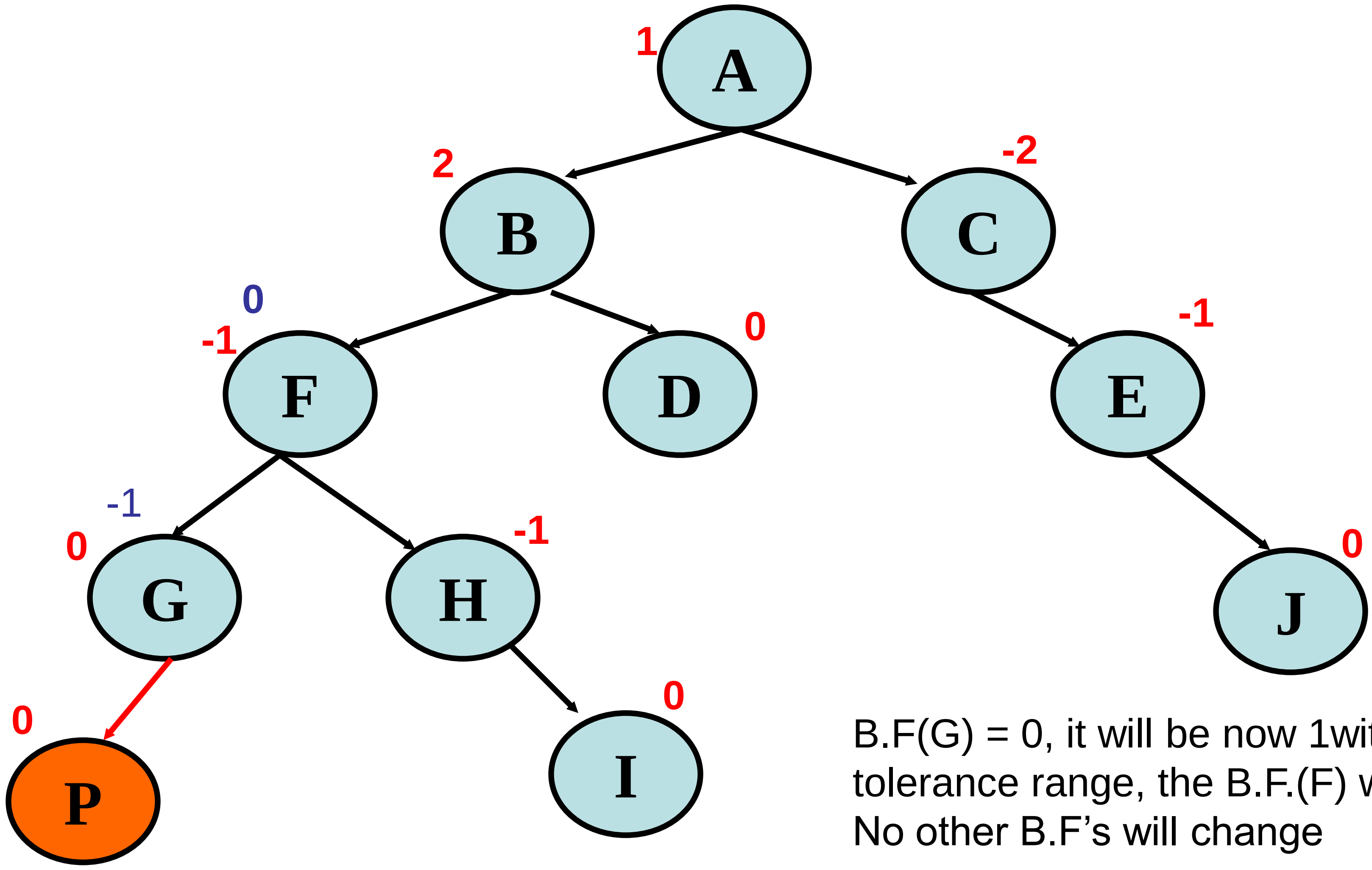
Observations

- Change in the height of the node will depend on
  - Where the node is added
  - What was the ht. of that subtree
  - The difference between the two heights
- We can conclude, that as we require the difference, let us store the balance factor with node rather than height.

Example

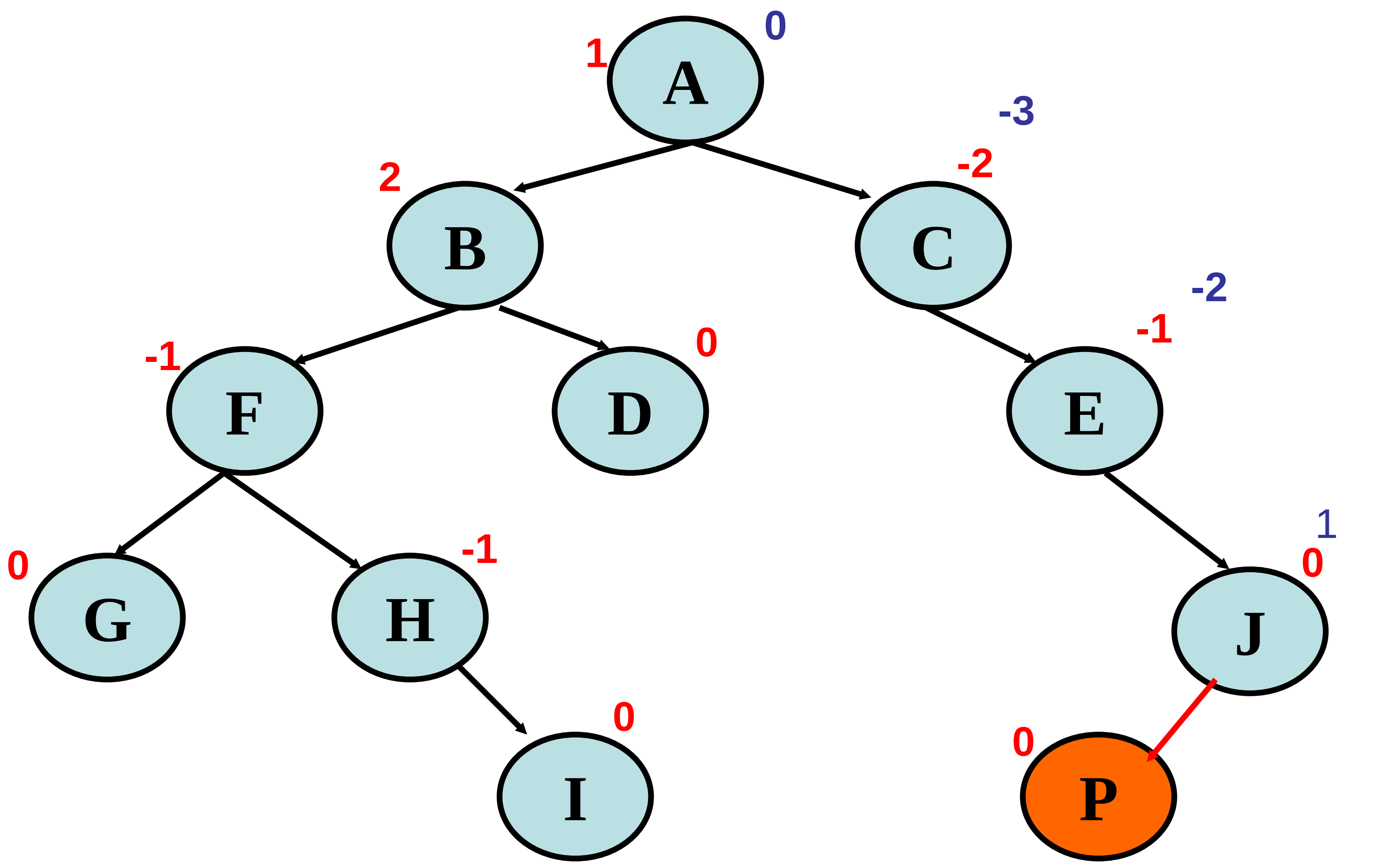


* P is to be added as left child of G



B.F(G) = 0, it will be now 1within the tolerance range, the B.F.(F) will be 0. No other B.F's will change

* P is to be added as left child of J



Observations

- B.F changes will be made only on increment and decrement basis , depending on whether the node is added on LHS or RHS.
- These changes are also reflected in the parents of the nodes in the similar manner and while making these changes if some B.F become 0 , then we should stop.
- Advantage : no need of Height function.
- We can assign the values directly during Creation.

Creation Algorithm

1. Create the root. Set B.F = 0
2. Create new node, and from root move till we get the par node to which it should be attached. But every time push the addresses of the nodes which are passed in the stack.
3. New node has B.F = 0
4. par = pop()
5. If par->lc == new node   increment b.f of par
6. If par->rc == new node   Decrement b.f of par
7. If B.F = 0 go to step 9
8. New node = par
9. If more nodes then go to step 2
10. stop

Implementation

Insert logic of BST with storing the par node on the stack.

Then do the following

```
Do
{
    if (par->lc == node)
          par->bf += 1
    else
         par->bf -= 1
    node = par
    par = pop();
}while(par != root && par->bf != 0)
```

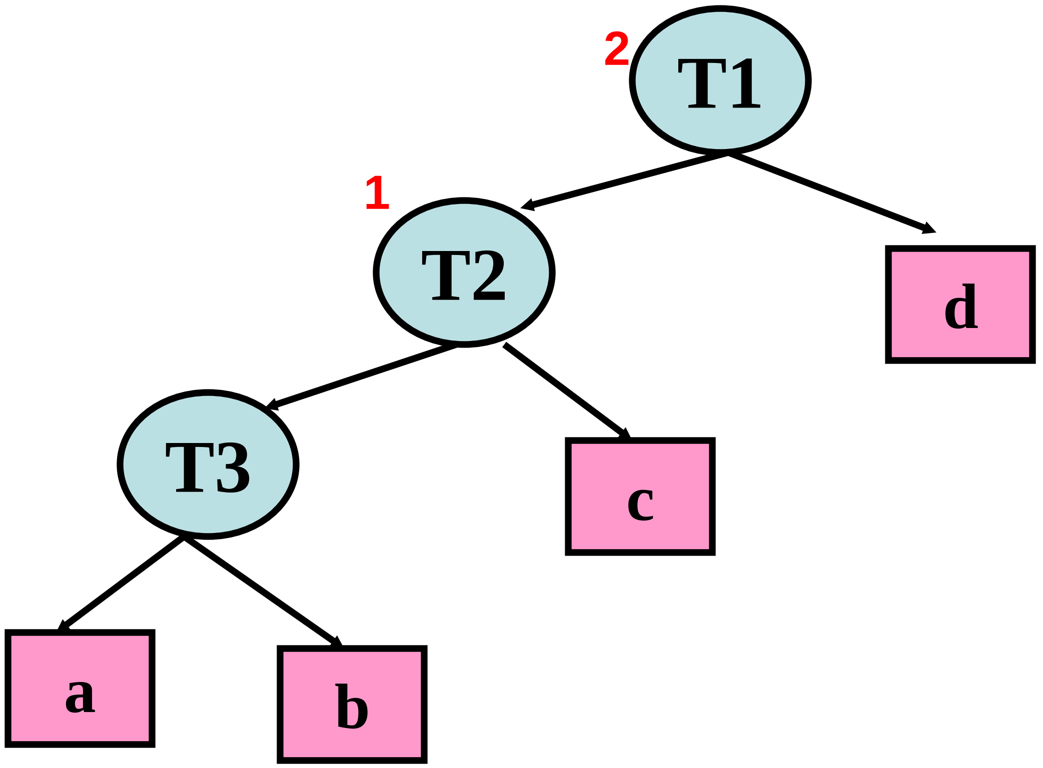
4 Types of Rotations to be applied when the tree is imbalanced

- Single rotations
  - LL Rotations
  - RR Rotations
- Double Rotations
  - LR Rotations
  - RL Rotations

LL Rotations

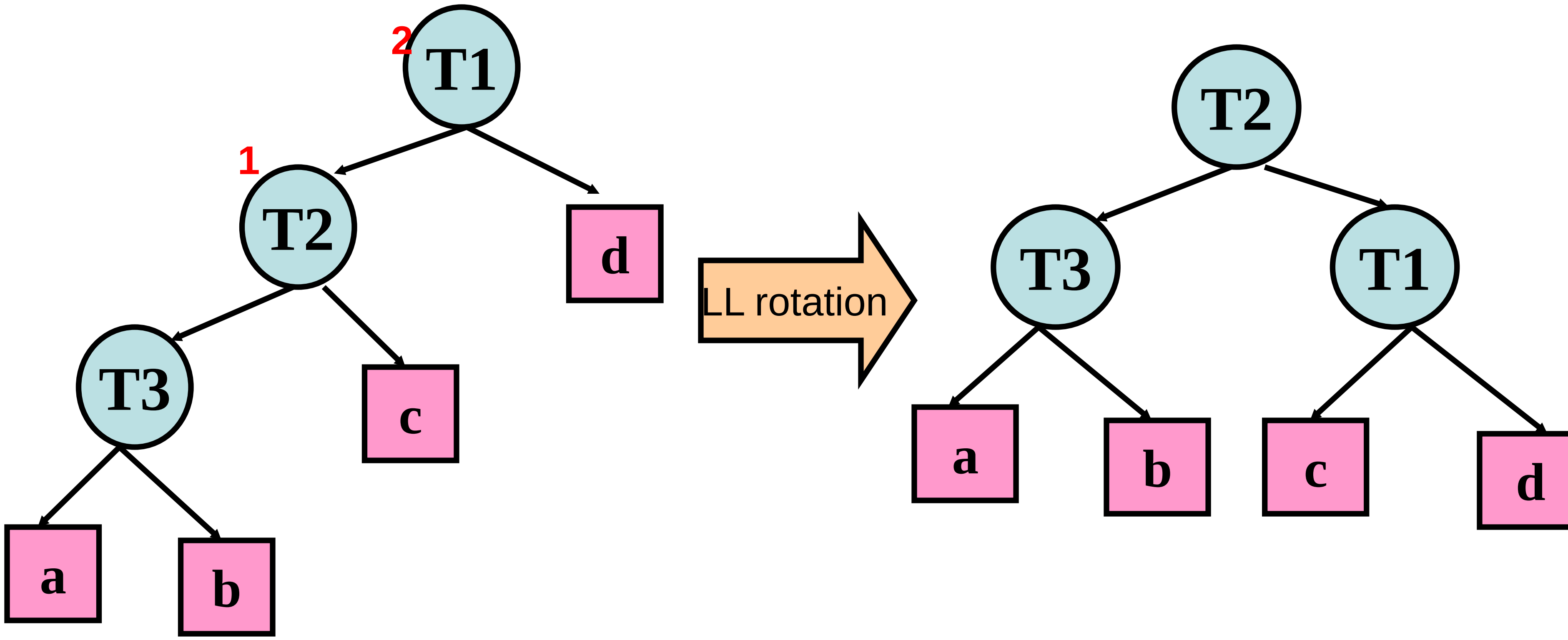
- When a node is added on the left of leftson, such that the balance factor of the current node becomes 2, balancing will be achieved by LL Rotations.

LL Rotations



New node is added any where in the subtree rooted at t3, and while modifying the balance factors in the parent nodes, if T1 b.f is 2, and T2 B.f is 1 then we need to apply LL rotation as new node is added to the left of T1 and left of T2.

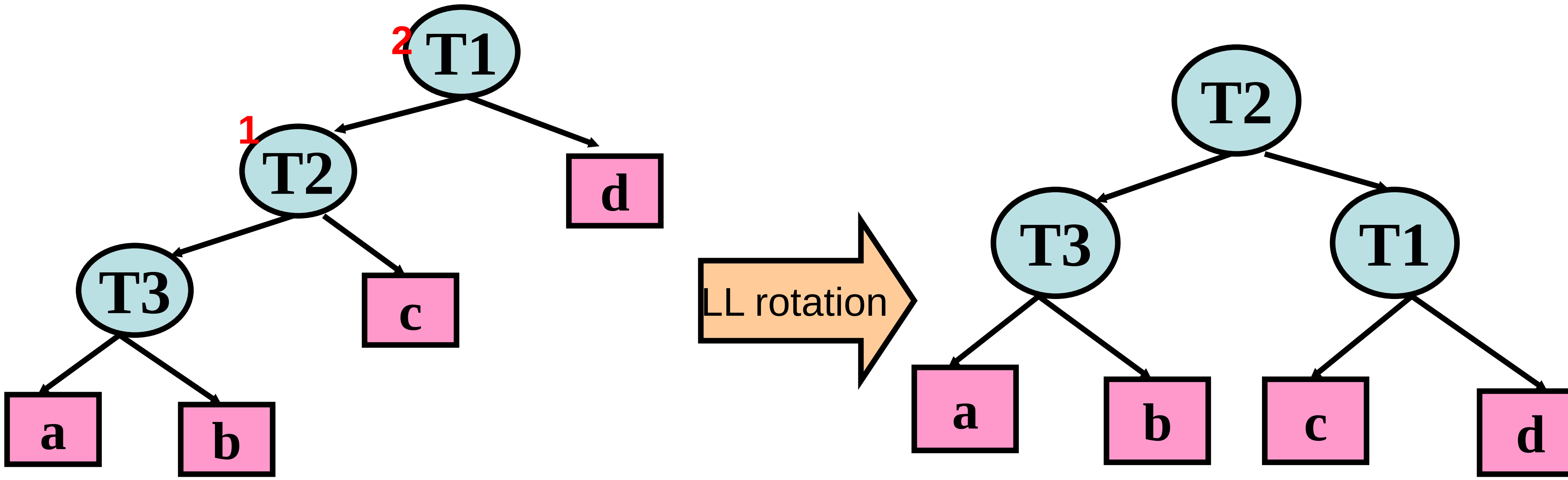
LL Rotations



Before Balancing

After Balancing

LL Rotations Proof



Before Balancing.

Assumptions :

Let ht(d) = x

So, ht(T2) = x + 2

So, Ht(T3) = x + 1

And So Ht(c) = x

After Balancing.

B.F (T1) = ht(c) – ht(d)

= x – x = 0

B.F (T3) = Original

B.F(T2) = Ht(T3) – Ht(T1)

= (x + 1) - (x + 1)

= 0

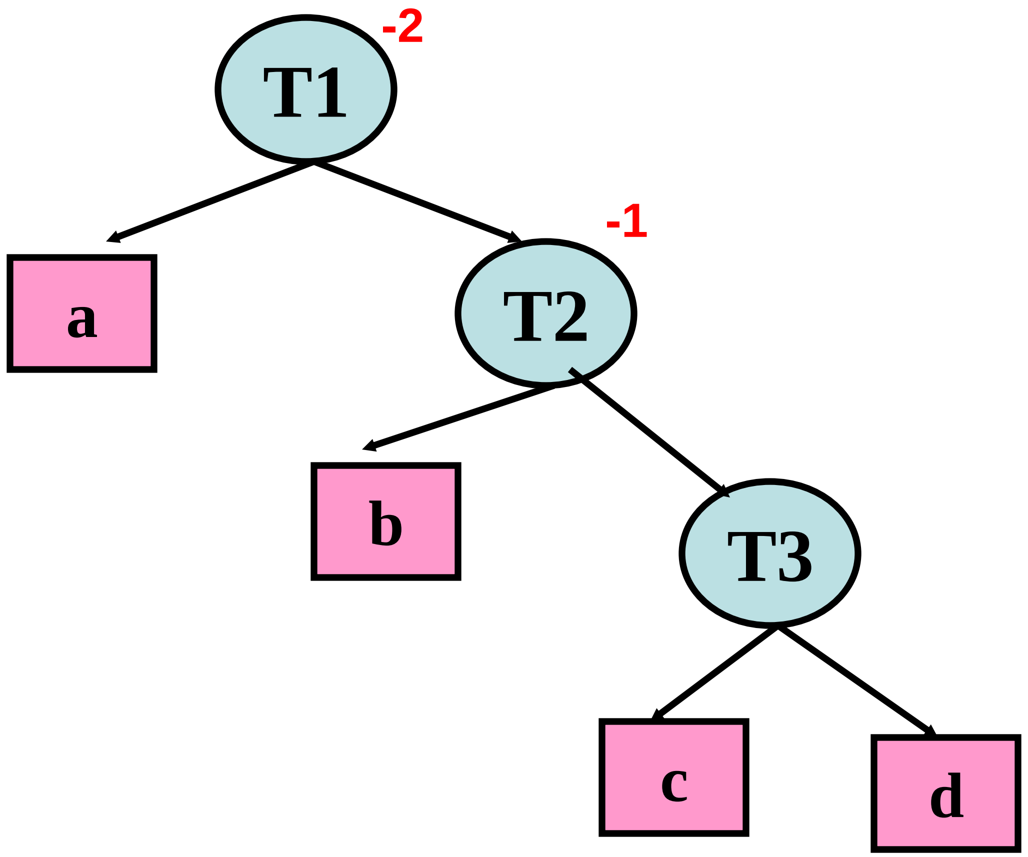
LL Rotations implementations

1. Let T1 be the node with bf 2 and its parent as par
2. T2 = T1->lc and has bf 1
3. T1->lc = T2->rc
4. T2->rc = T1
5. T1->bf = 0
6. T2->bf = 0
7. Attach T2 to parent node par

RR Rotations

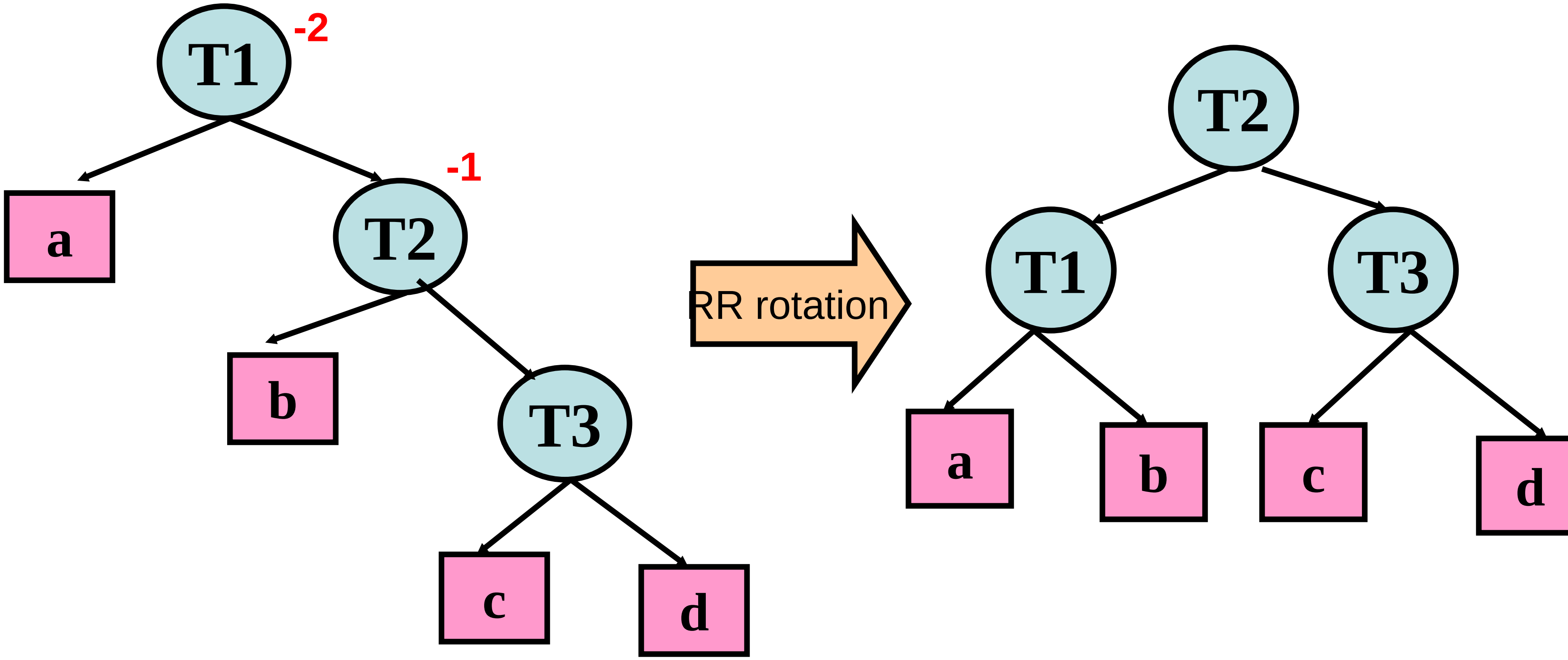
- When a node is added on the right of rightson, such that the balance factor of the current node becomes -2, balancing will be achieved by RR Rotations.

RR Rotations



New node is added any where in the subtree rooted at t3, and while modifying the balance factors in the parent nodes, if T1 b.f is -2, and T2 B.f is -1 then we need to apply RR rotation as new node is added to the right of T1 and right of T2.

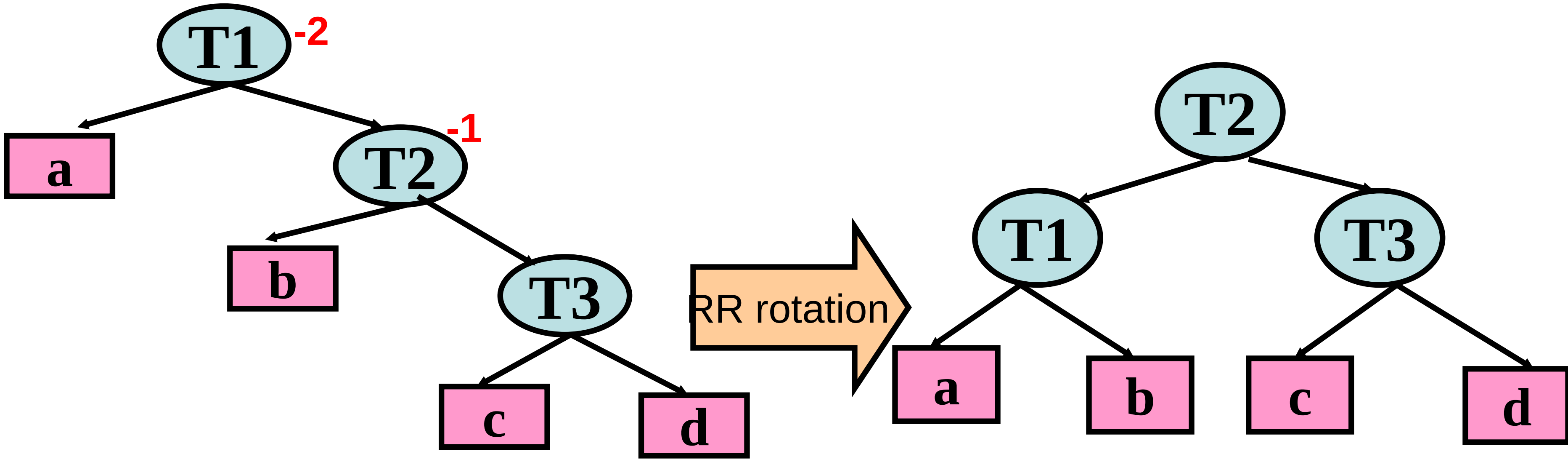
RR Rotations



Before Balancing

After Balancing

RR Rotations



Before Balancing.

Assumptions :

Let ht(a) = x

So, ht(T2) = x + 2

So, Ht(T3) = x + 1

And So Ht(b) = x

After Balancing.

B.F (T1) = ht(a) – ht(b)

= x – x = 0

B.F (T3) = Original

B.F(T2) = Ht(T1) – Ht(T3)

= (x + 1) - (x + 1)

= 0

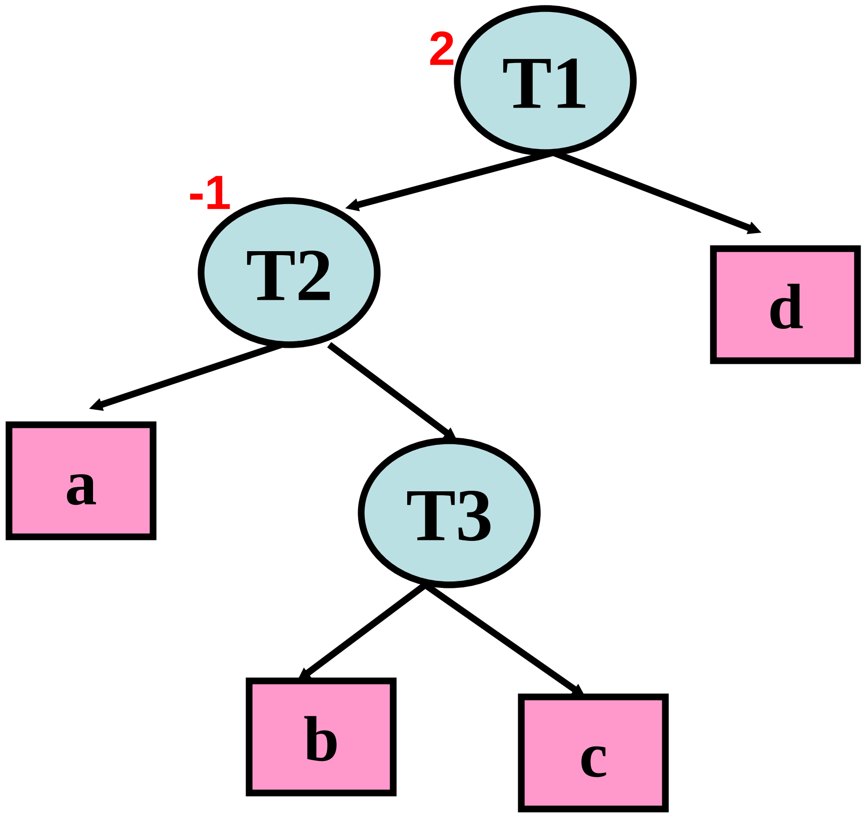
RR Rotations implementations

1. Let T1 be the node with bf -2 and its parent as par
2. T2 = T1->rc and has bf -1
3. T1->rc = T2->lc
4. T2->lc = T1
5. T1->bf = 0
6. T2->bf = 0
7. Attach T2 to parent node par

LR Rotations

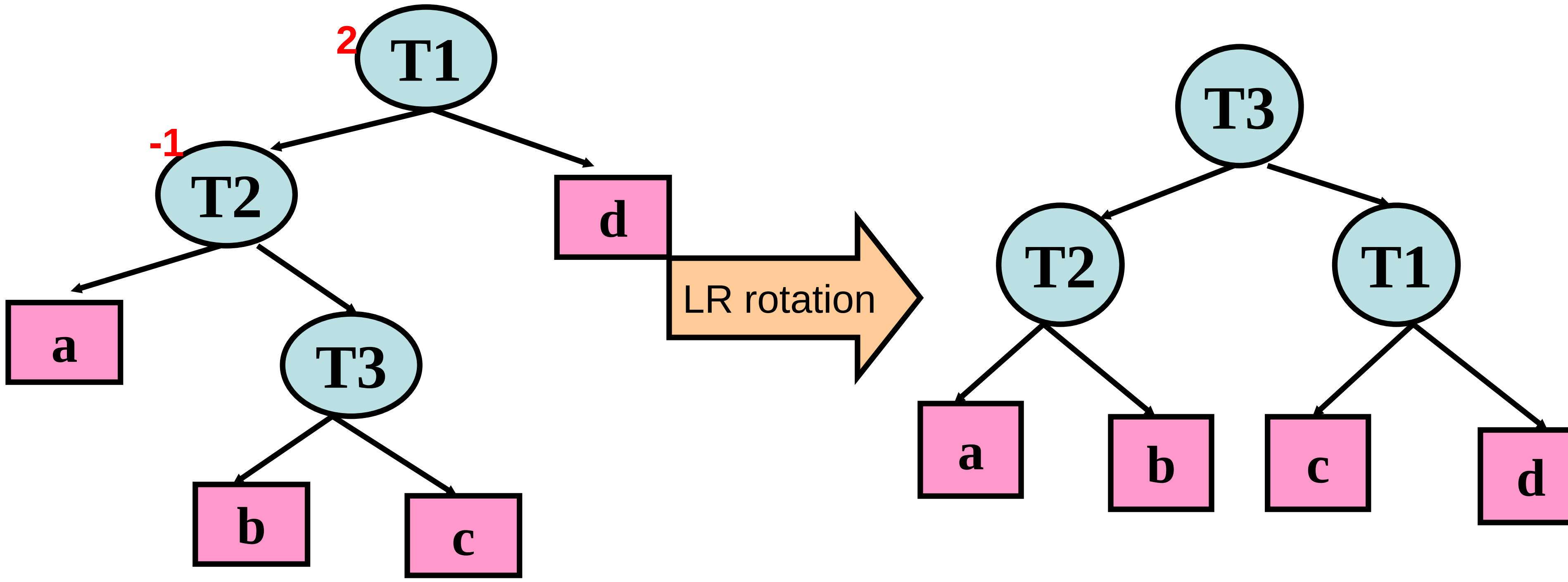
- When a node is added on the left of Rightson, such that the balance factor of the current node becomes 2 and that of its leftson becomes -1, balancing will be achieved by LR Rotations.

LR Rotations



New node is added any where in the subtree rooted at t3, and while modifying the balance factors in the parent nodes, if T1 b.f is 2, and T2 B.f is -1 then we need to apply LR rotation as new node is added to the left of T1 and right of T2.

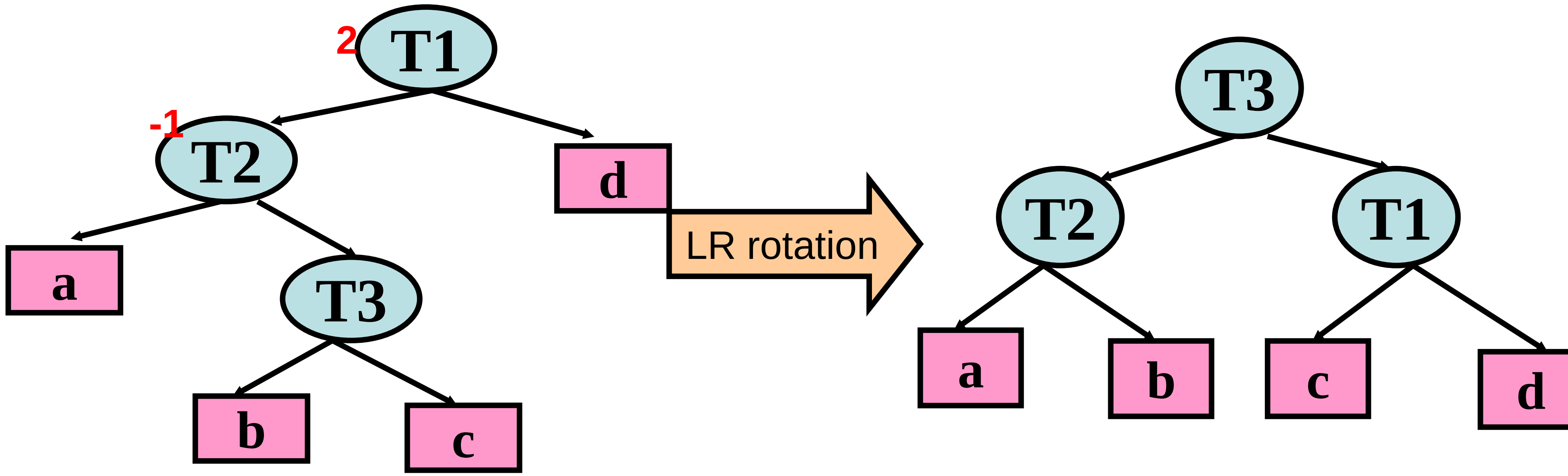
LR Rotations



Before Balancing

After Balancing

LR Rotations



After Balancing.

HT (T2) = max (Ht(a), Ht(b)) + 1

= max(x, will not be > x) + 1

= x + 1

HT (T1) = max (Ht(c), Ht(d)) + 1

= max(will not be > x , x) + 1

= x + 1

B.F (T3) = O

Before Balancing.

Assumptions :

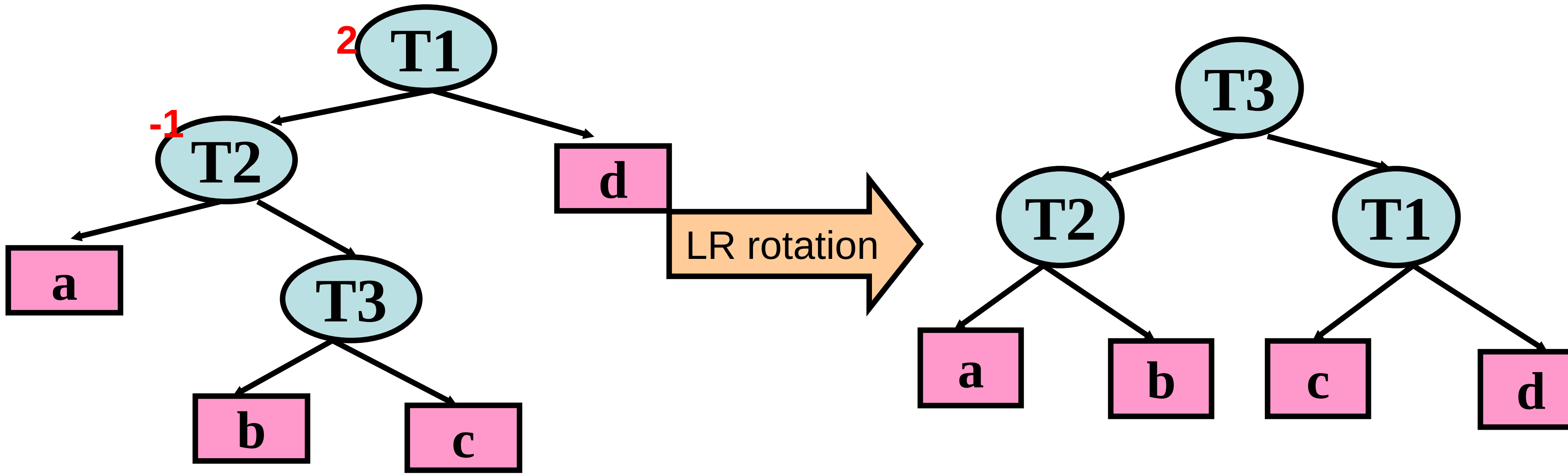
Let ht(d) = x

So, ht(T2) = x + 2

So, Ht(T3) = x + 1

And So Ht(a) = x

Actually B.F of T3 could be +1 or -1 after adding the node.
Case 1 : When B.F of T3 = 1



Before Balancing.

ht(b) = ht(c) + 1

But , ht(T3) = x + 1

So, ht(b) = x

So, Ht(c) = x - 1

After Balancing.

BT (T2) = Ht(a) - Ht(b)

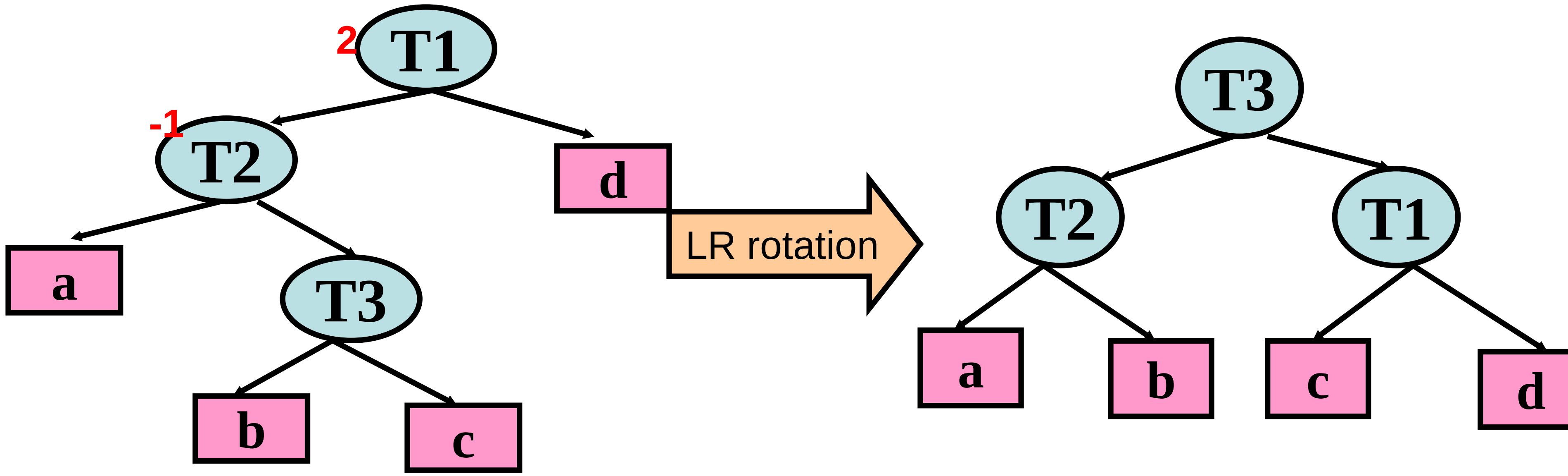
= x – x = 0

BF (T1) = Ht(c) - Ht(d)

= x – 1 - x

= -1

Actually B.F of T3 could be +1 or -1 after adding the node.
Case 2 : When B.F of T3 = -1



After Balancing.

BT (T2) = Ht(a) - Ht(b)

= x – (x – 1) = 1

BF (T1) = Ht(c) - Ht(d)

= x – x

= 0

Before Balancing.

ht(c) = ht(b) + 1

But , ht(T3) = x + 1

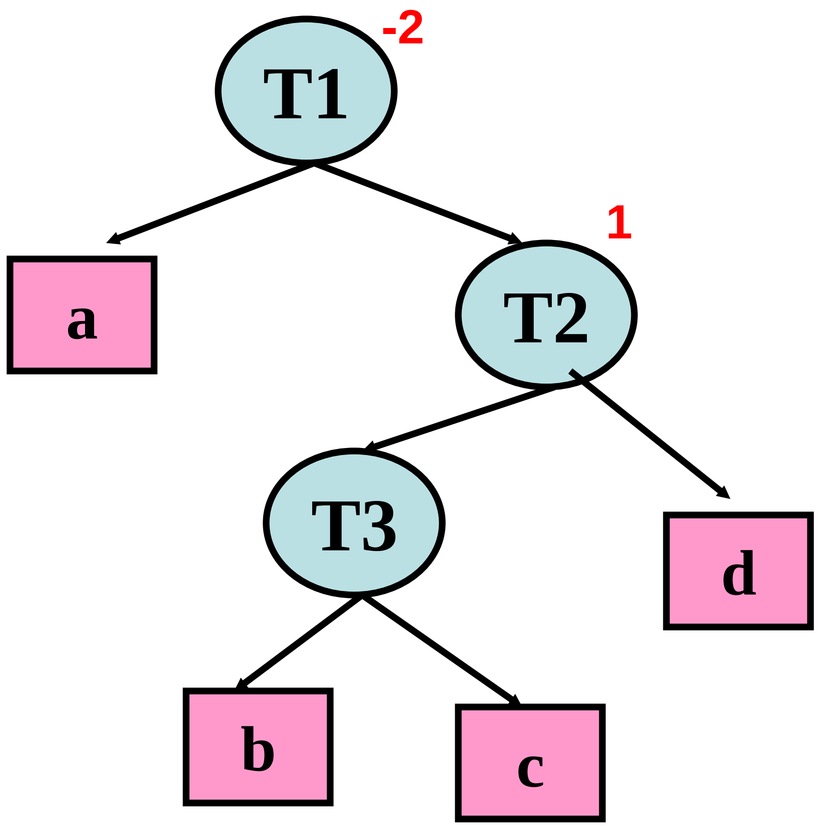
So, ht(c) = x

So, Ht(b) = x - 1

RL Rotations

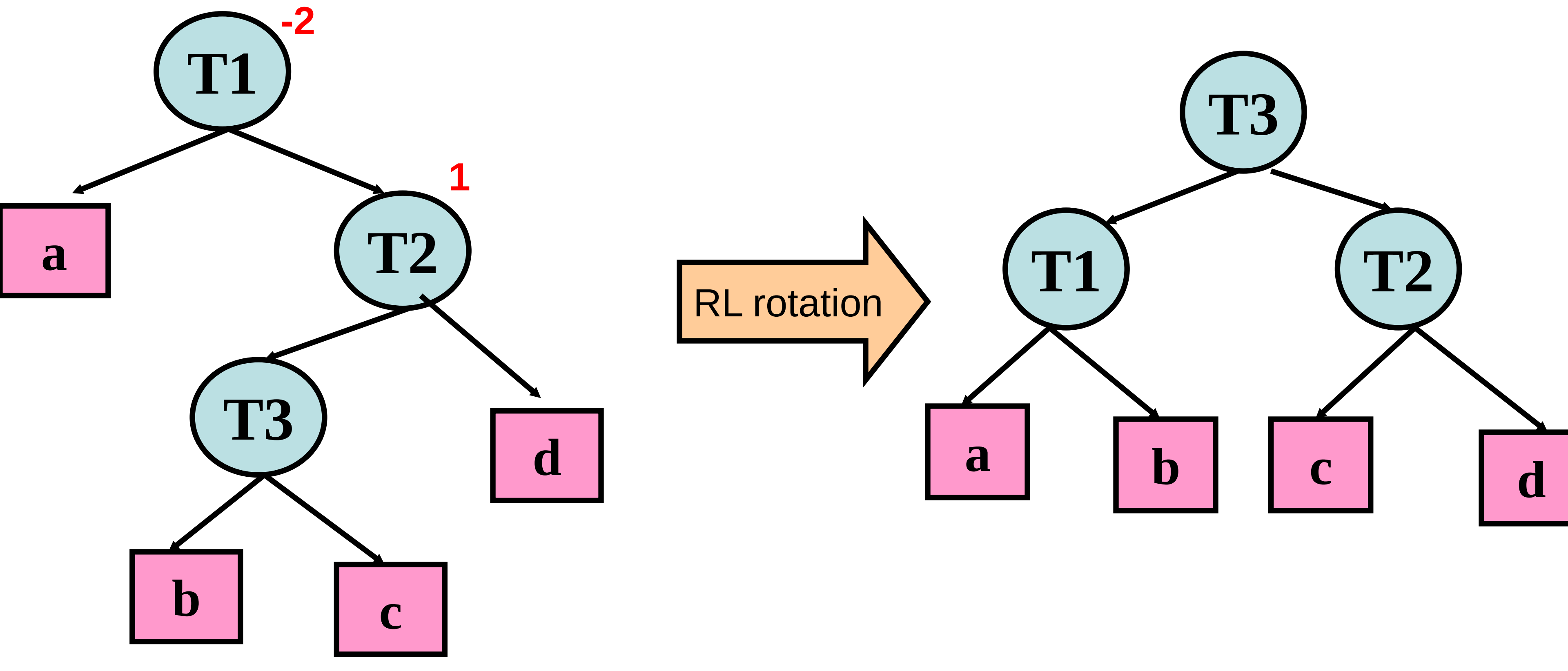
- When a node is added on the Right of Leftson, such that the balance factor of the current node becomes -2 and that of its Rightson becomes 1, balancing will be achieved by RL Rotations.

RL Rotations



New node is added any where in the subtree rooted at t3, and while modifying the balance factors in the parent nodes, if T1 b.f is -2, and T2 B.f is 1 then we need to apply RL rotation as new node is added to the right of T1 and left of T2.

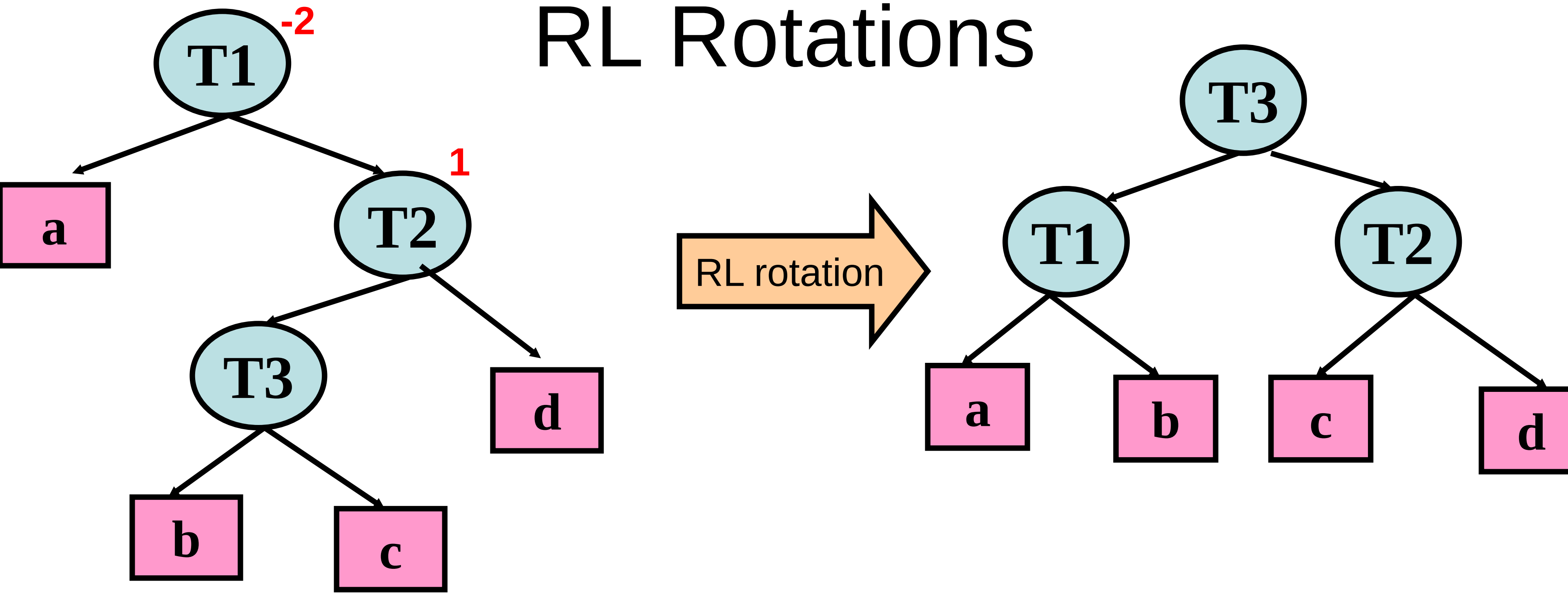
RL Rotations



Before Balancing

After Balancing

RL Rotations



Before Balancing.

Assumptions :

Let ht(a) = x

So, ht(T2) = x + 2

So, Ht(T3) = x + 1

And So Ht(d) = x

After Balancing.

HT (T2) = max (Ht(a), Ht(b)) + 1

= max(x, will not be > x) + 1

= x + 1

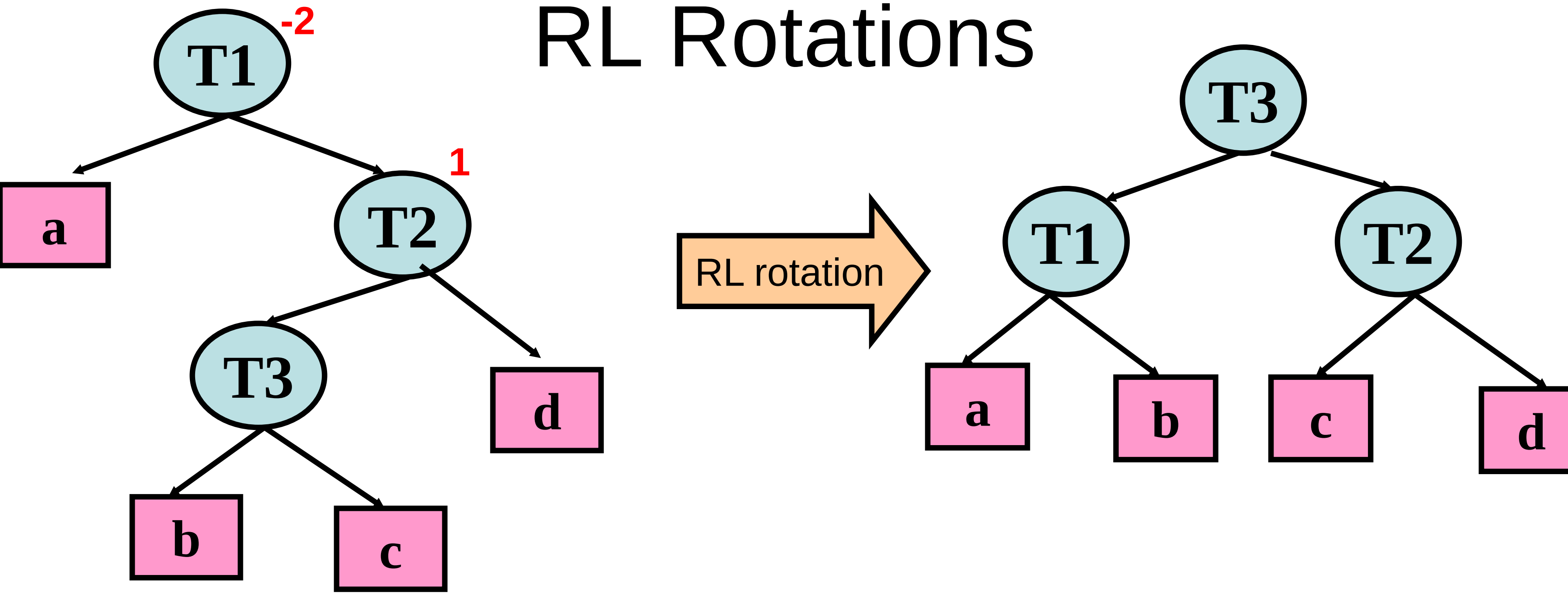
HT (T1) = max (Ht(c), Ht(d)) + 1

= max(will not be > x , x) + 1

= x + 1

B.F (T3) = O

RL Rotations

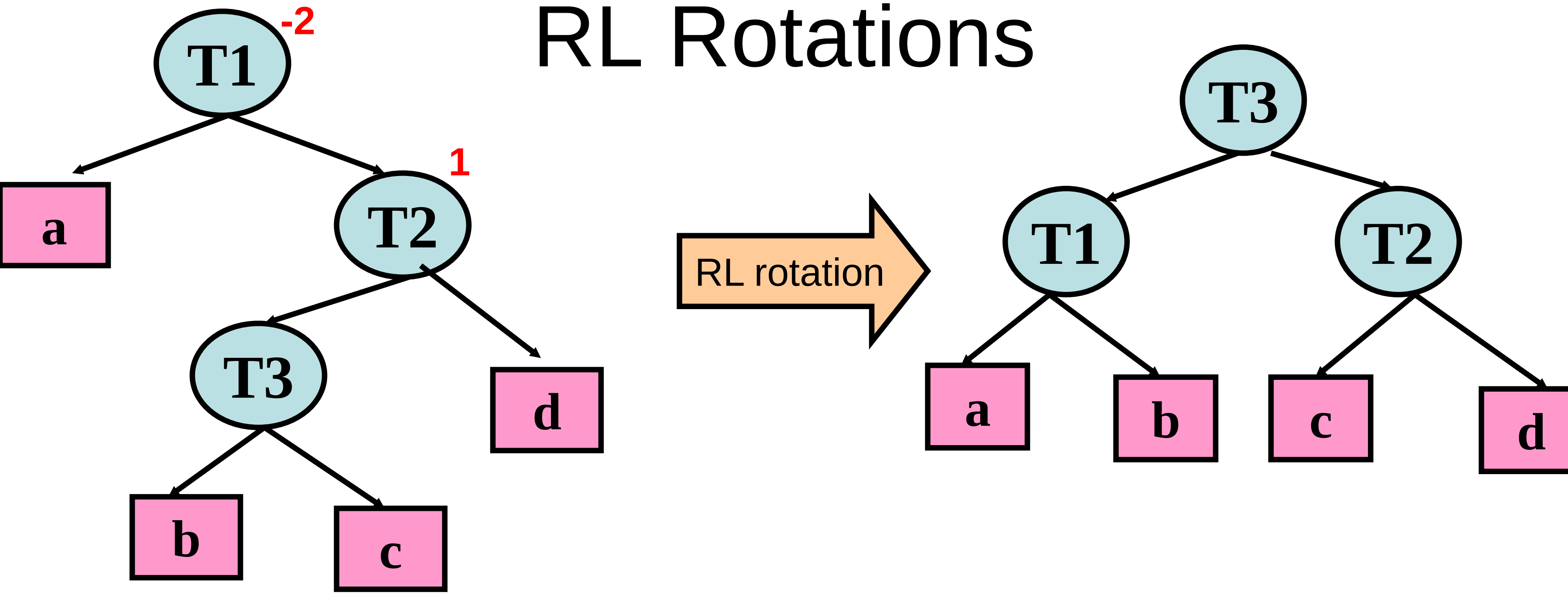


Actually B.F of T3 could be +1 or -1 after adding the node.
Case 1 : When B.F of T3 = 1

Before Balancing.
ht(b) = ht(c) + 1
But , ht(T3) = x + 1
So, ht(b) = x
So, Ht(c) = x - 1

After Balancing.
BT (T2) = Ht(c) - Ht(d)
= x – 1 – x = -1
BF (T1) = Ht(a) - Ht(b)
= x – x
= 0

RL Rotations



Actually B.F of T3 could be +1 or -1 after adding the node.

Case 2 : When B.F of T3 = -1

Before Balancing.

ht(b) = ht(c) + 1

But , ht(T3) = x + 1

So, ht(b) = x - 1

So, Ht(c) = x

After Balancing.

BT (T2) = Ht(c) - Ht(d)

= x – x = 0

BF (T1) = Ht(a) - Ht(b)

= x – (x-1)

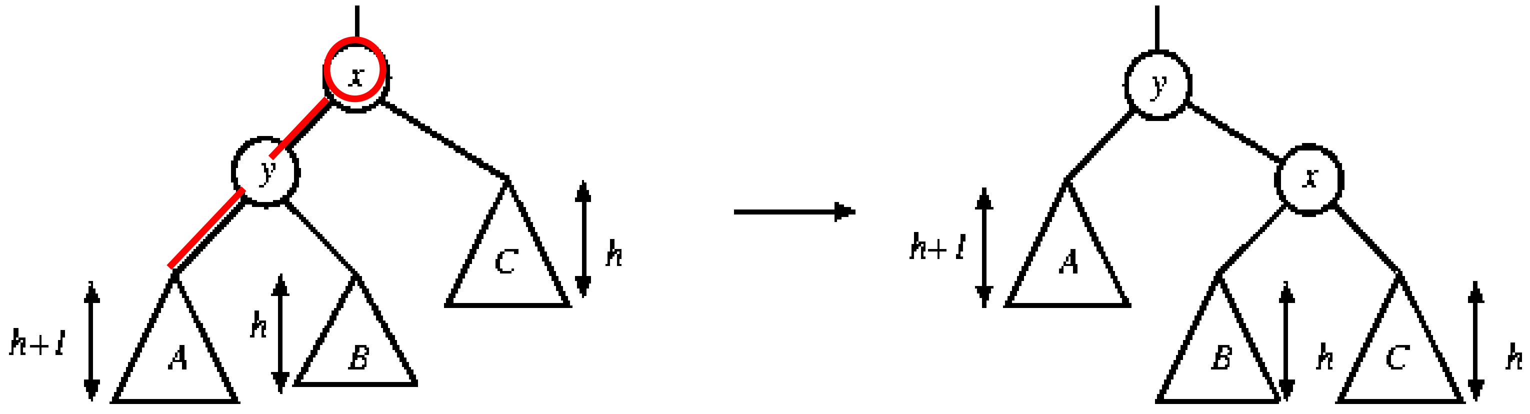
= 1

Single rotation

The new key is inserted in the subtree A.

The AVL-property is violated at x

- height of left(x) is h+2
- height of right(x) is h.

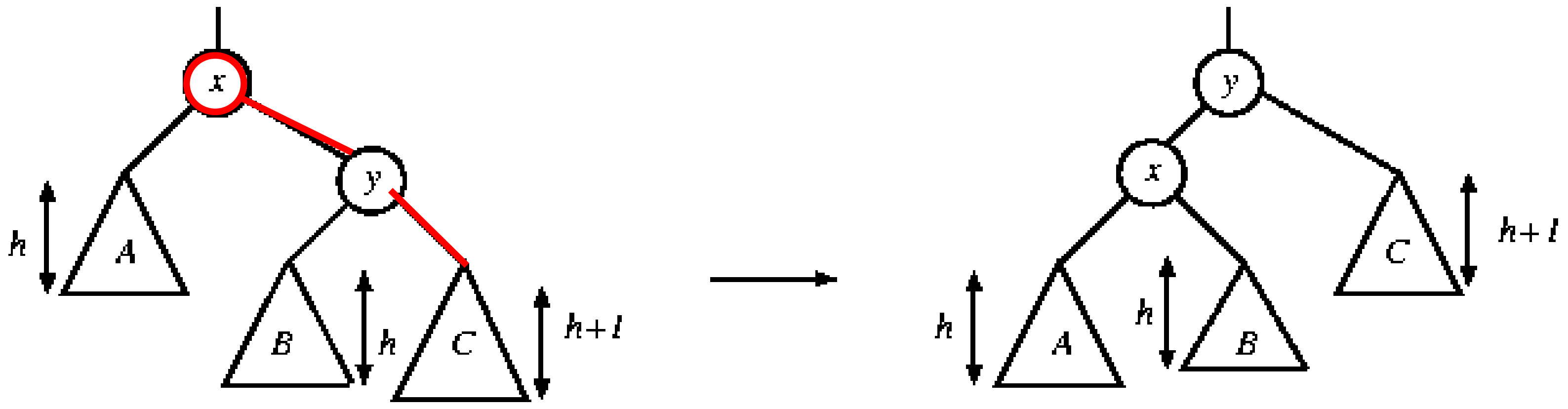


Rotate with left child

Single rotation

The new key is inserted in the subtree C.

The AVL-property is violated at x.

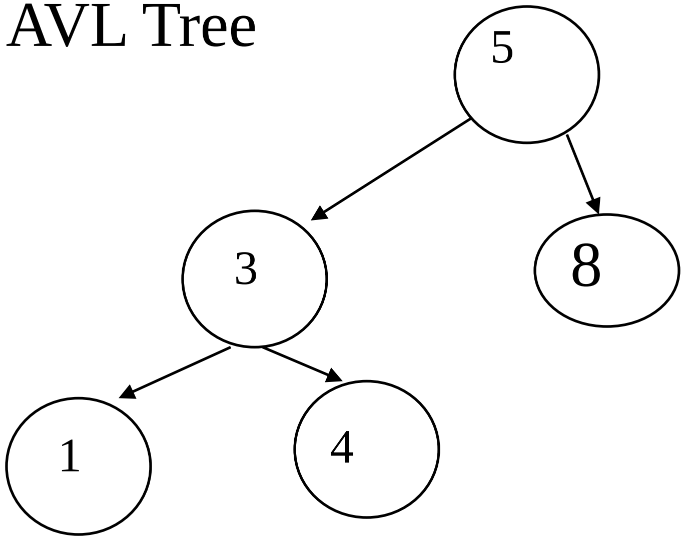


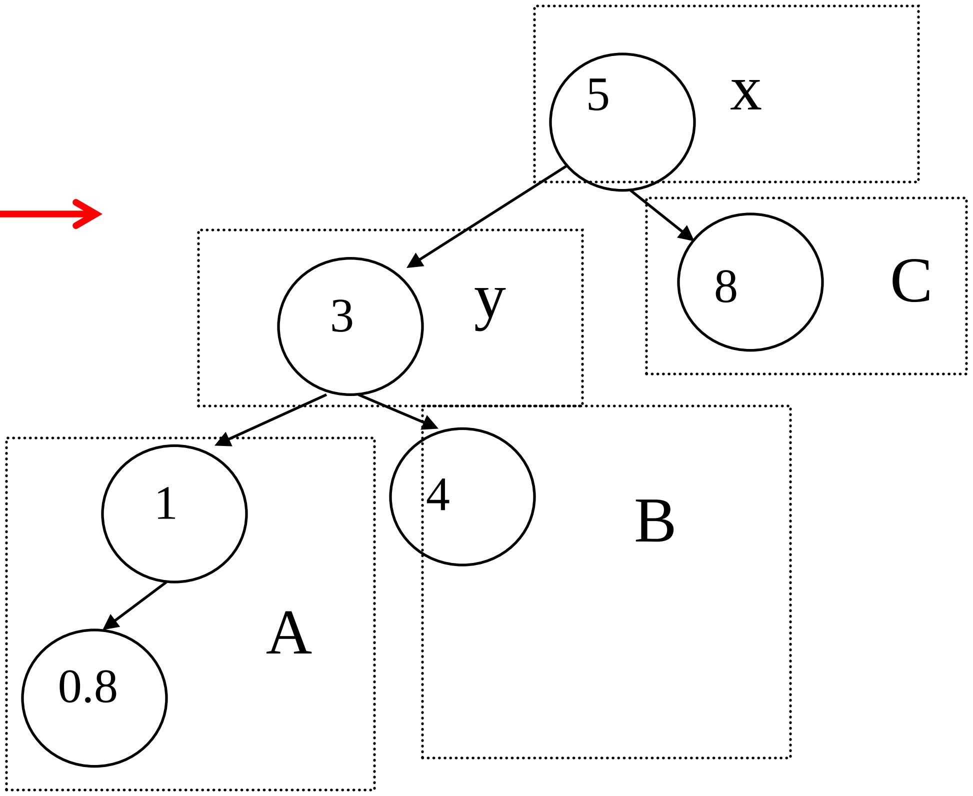
Rotate with right child

Single rotation takes O(1) time.

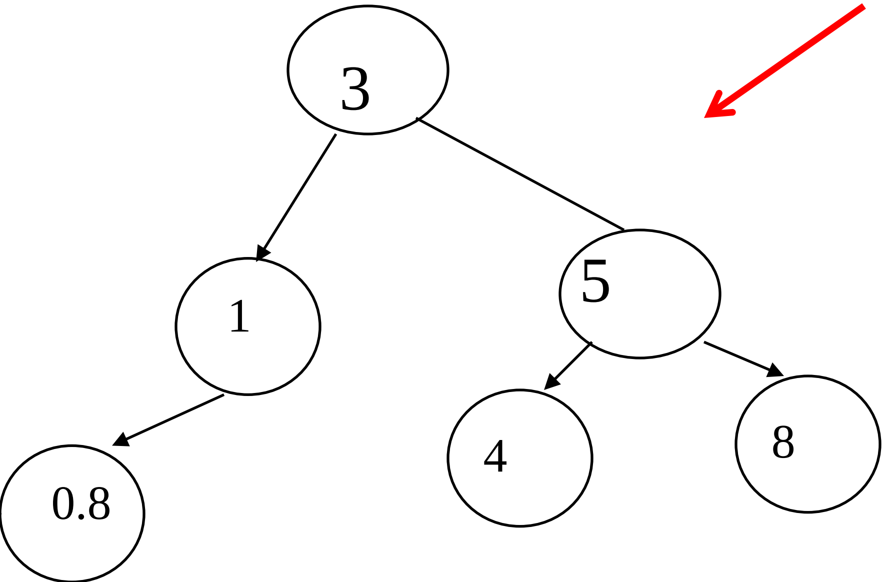
Insertion takes O(log N) time.

AVL Tree





Insert 0.8



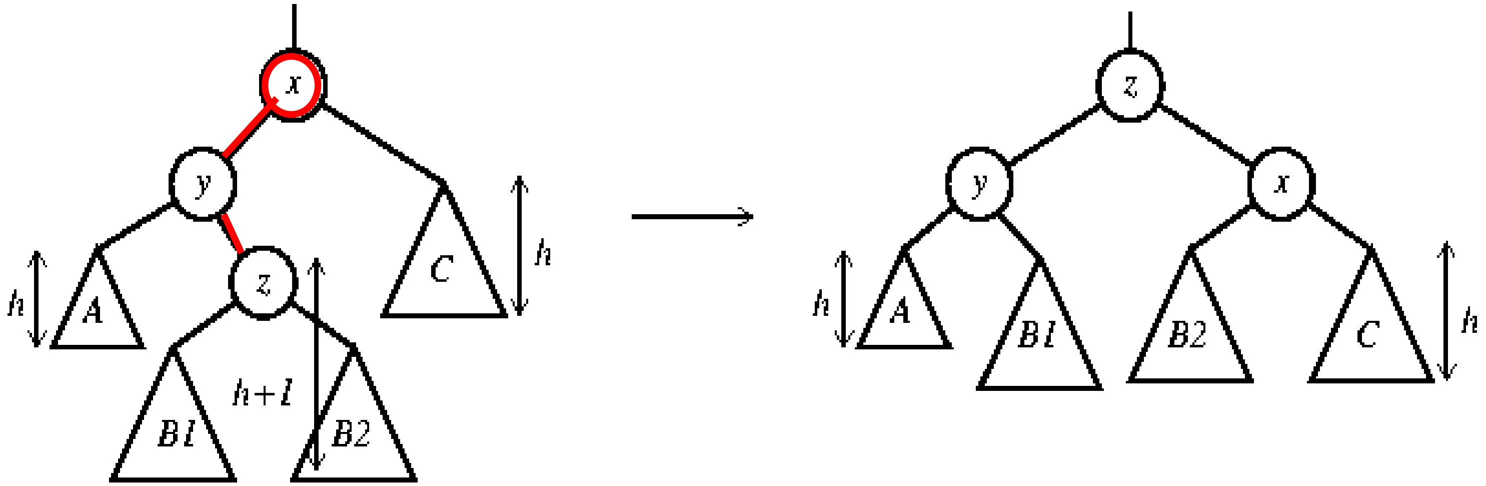
After rotation

Double rotation

The new key is inserted in the subtree B1 or B2.

The AVL-property is violated at x.

x-y-z forms a zig-zag shape



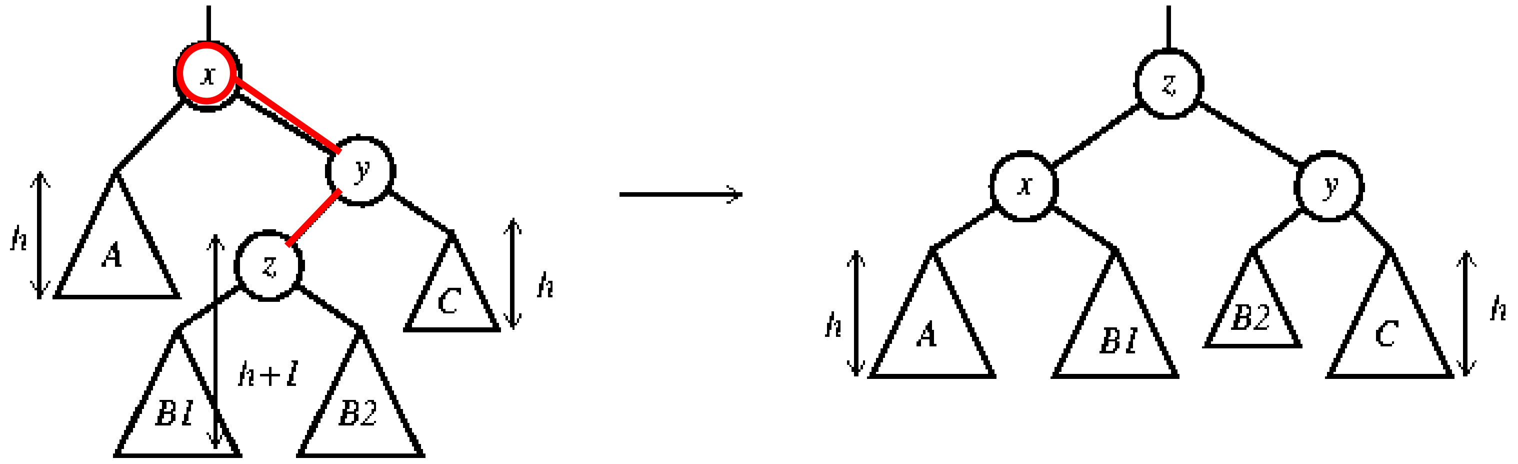
Double rotate with left child

also called left-right rotate

Double rotation

The new key is inserted in the subtree B1 or B2.

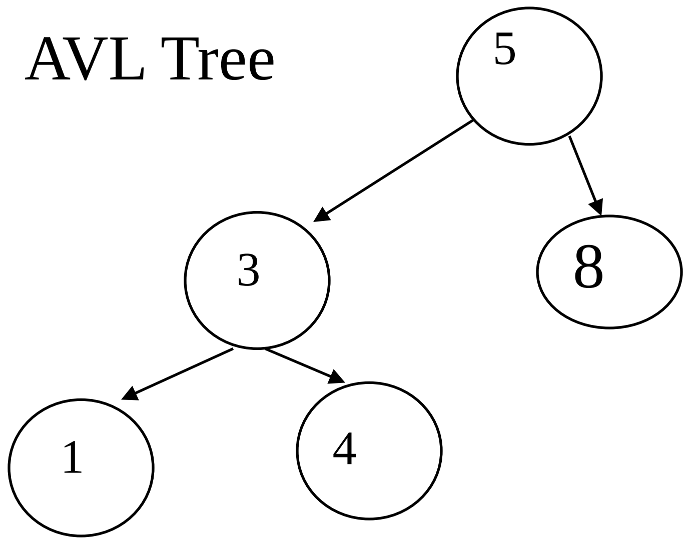
The AVL-property is violated at x.

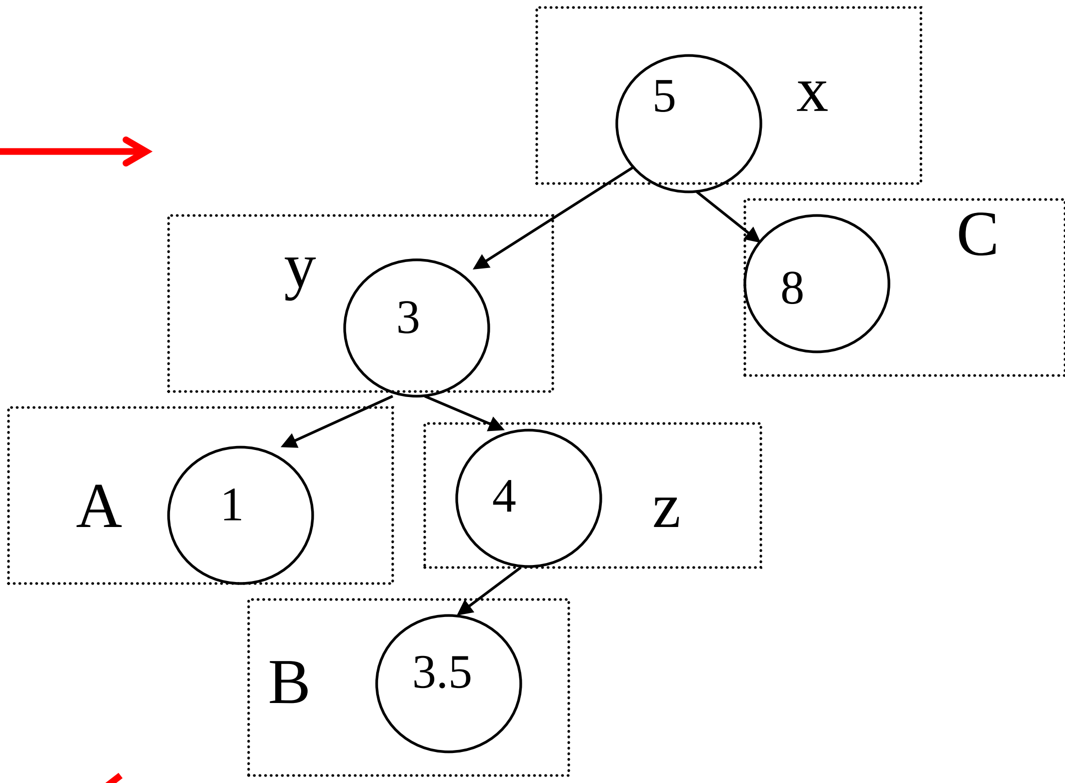


Double rotate with right child

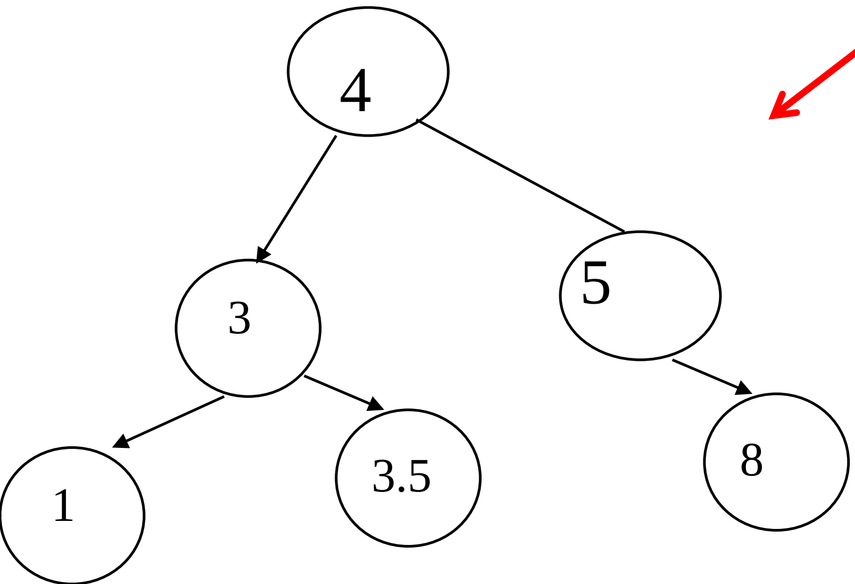
also called right-left rotate

AVL Tree





Insert 3.5



After Rotation

An Extended Example

Insert 3,2,1,4,5,6,7, 16,15,14

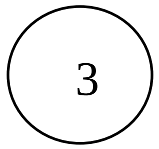


Fig 1

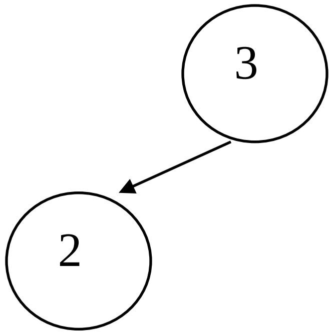


Fig 2

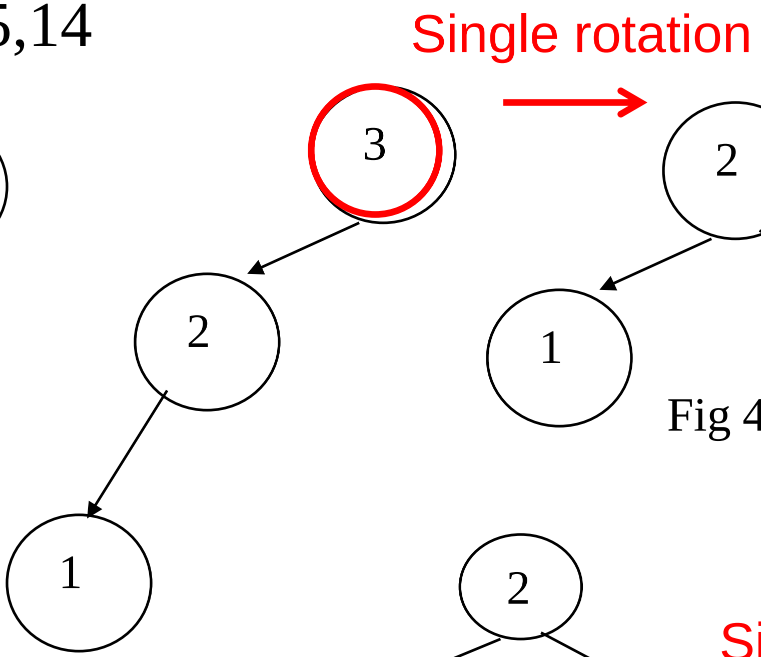


Fig 3

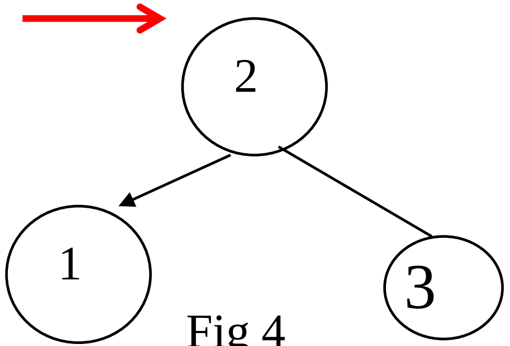


Fig 4

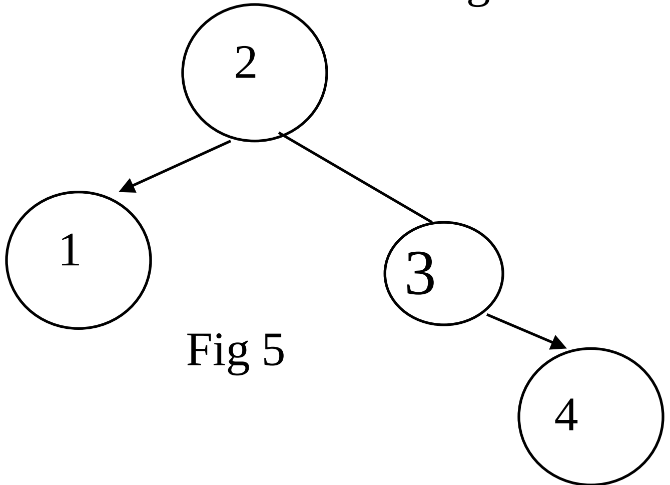


Fig 5

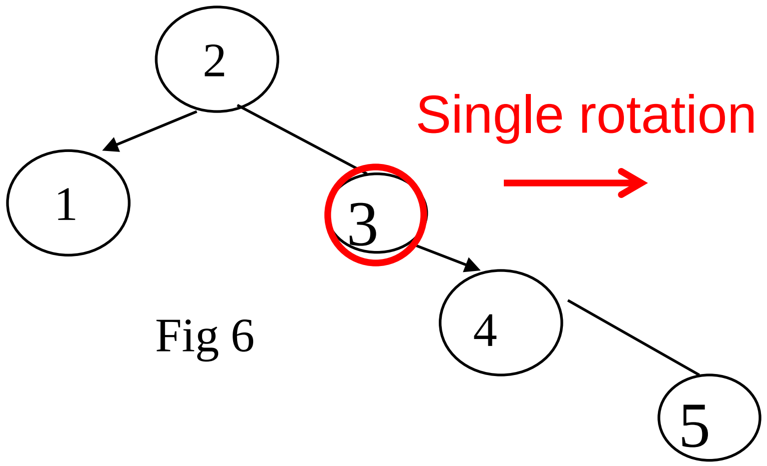


Fig 6

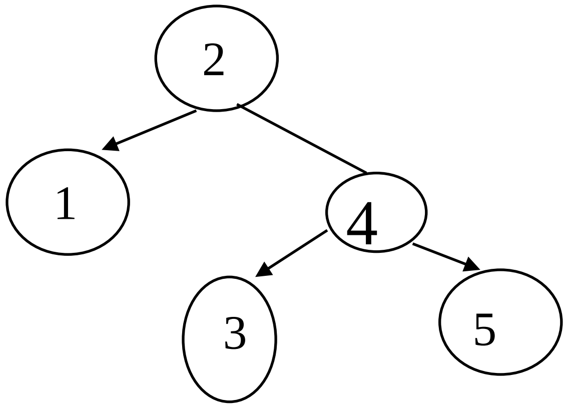


Fig 7

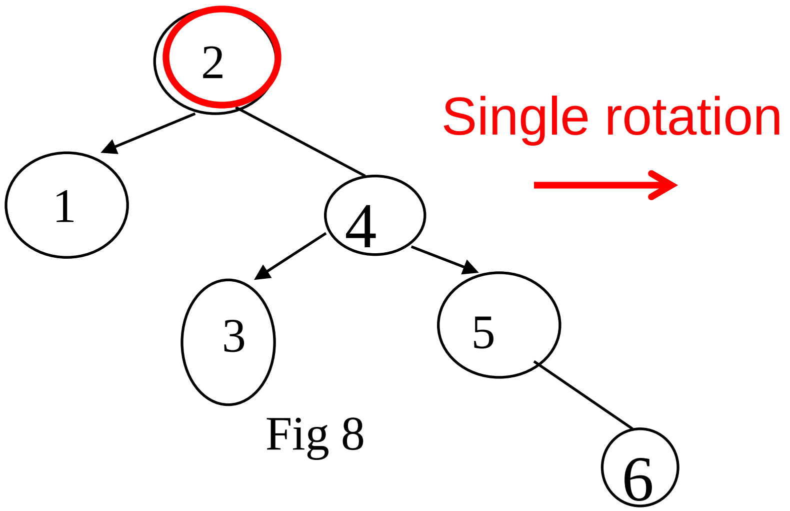


Fig 8

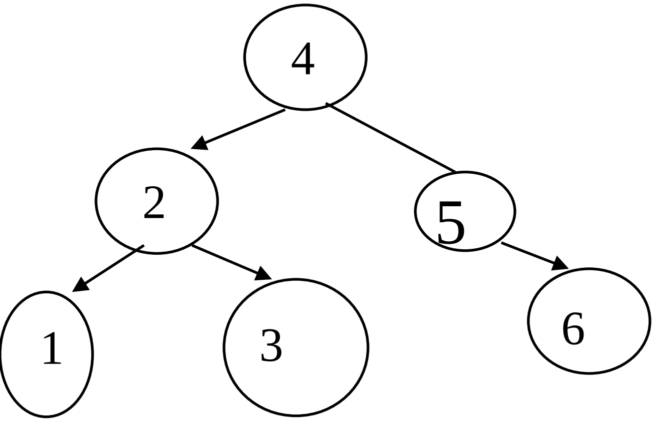


Fig 9

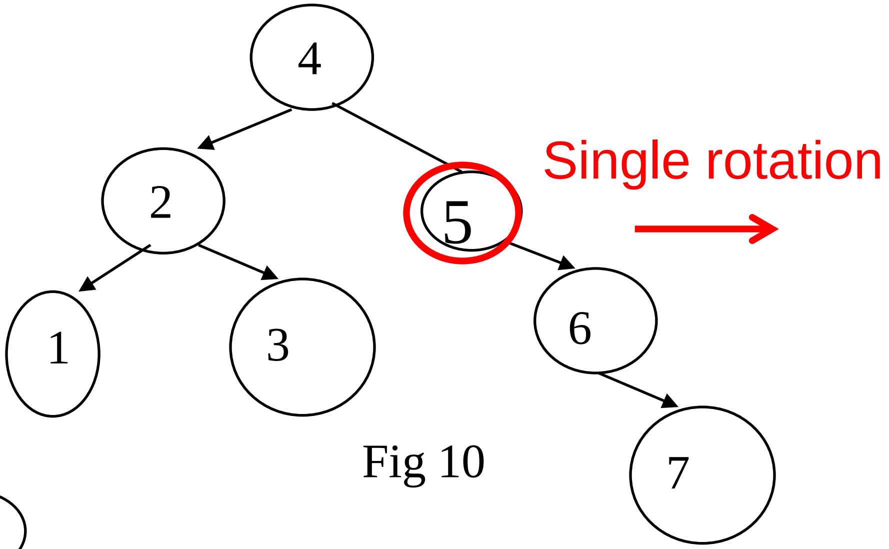


Fig 10

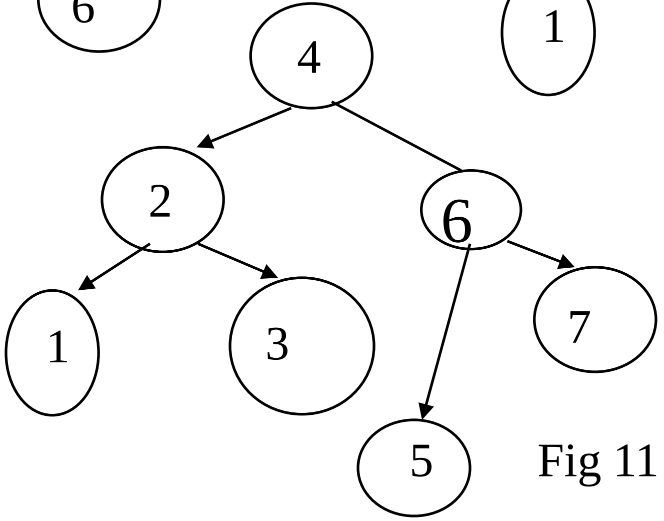


Fig 11

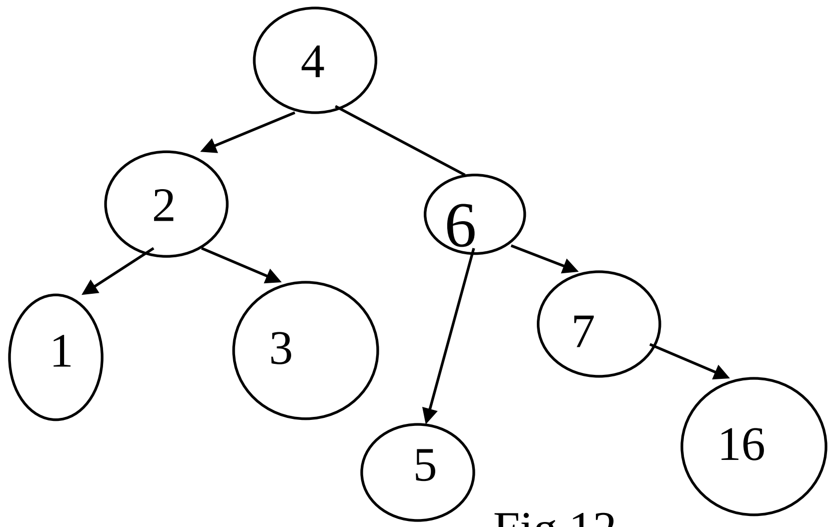


Fig 12

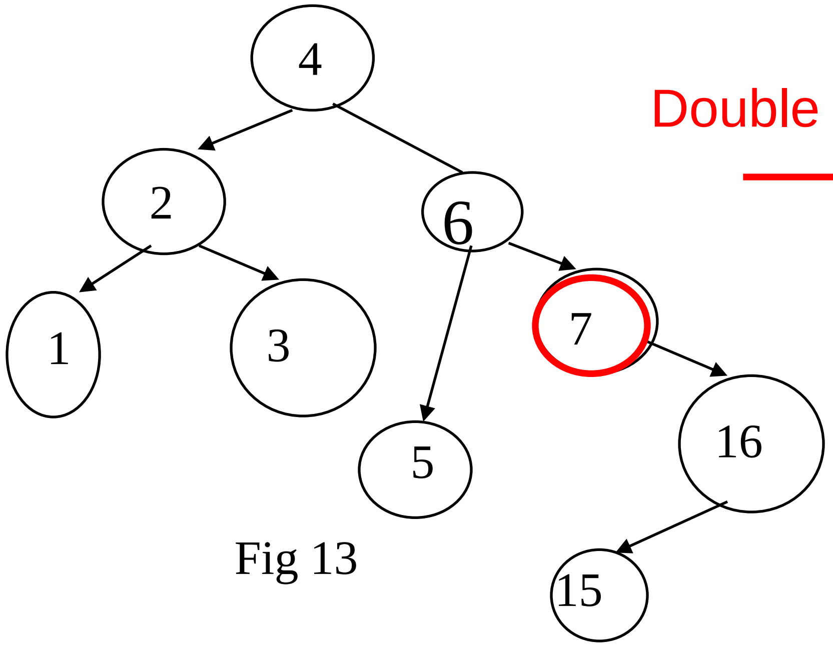


Fig 13

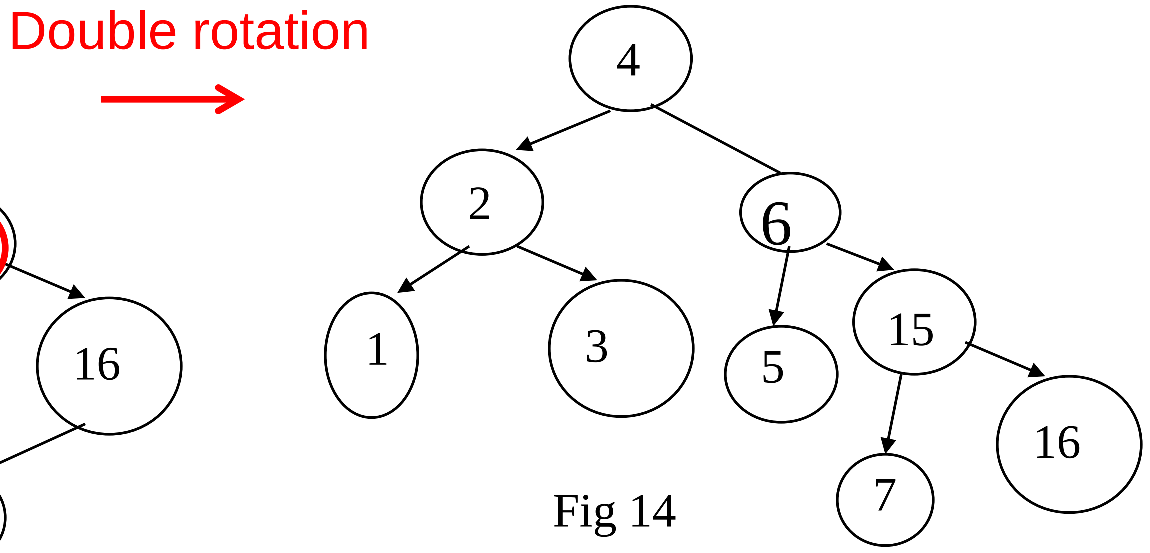


Fig 14

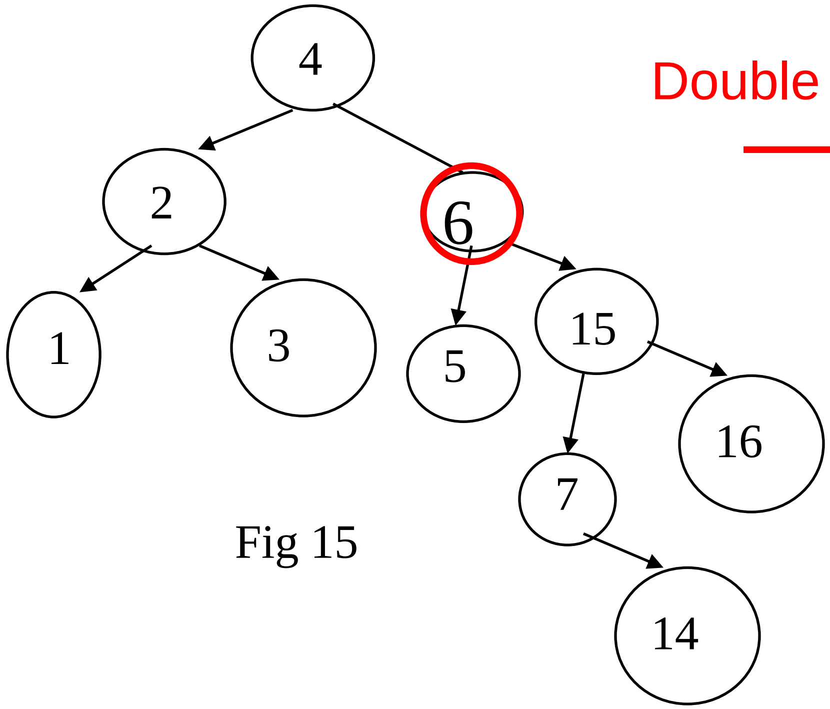


Fig 15

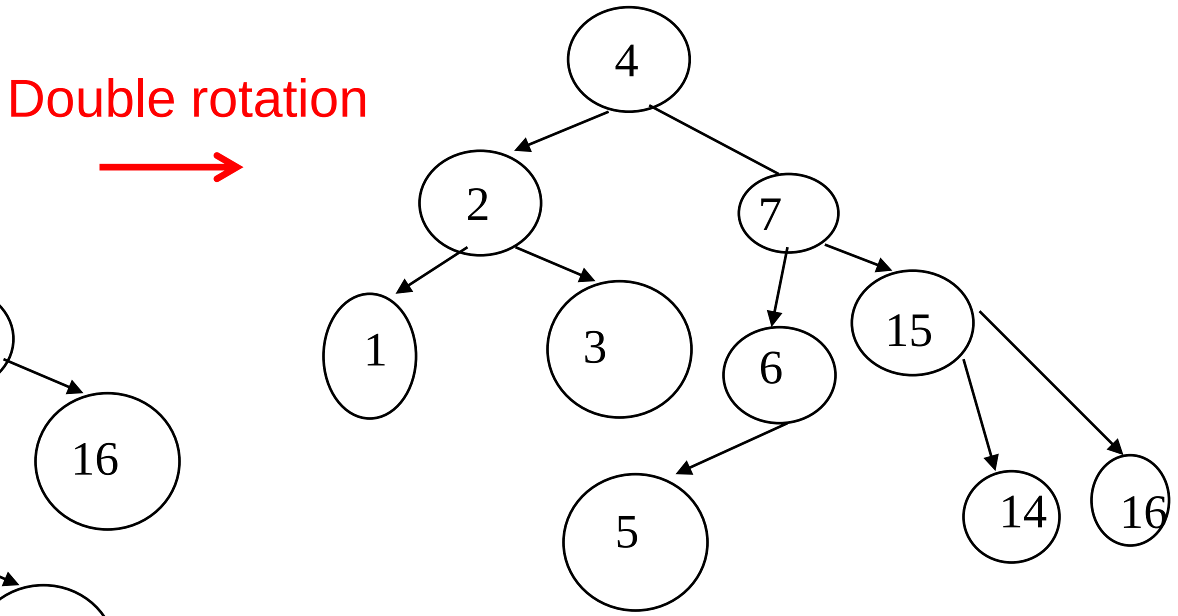


Fig 16

Deletion

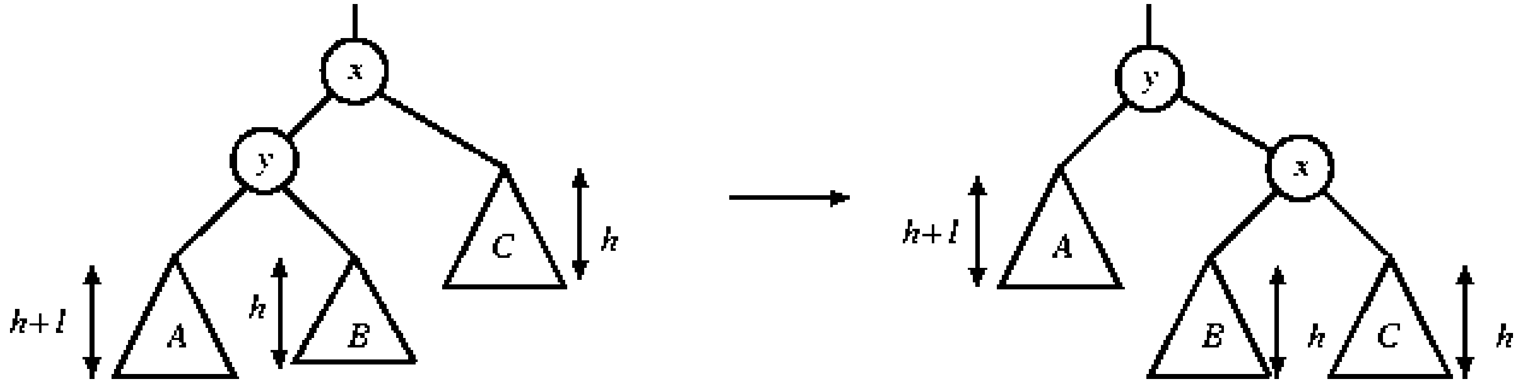
- Delete a node x as in ordinary binary search tree. Note that the last node deleted is a leaf.
- Then trace the path from the new leaf towards the root.
- For each node x encountered, check if heights of left(x) and right(x) differ by at most 1. If yes, proceed to parent(x). If not, perform an appropriate rotation at x. There are 4 cases as in the case of insertion.
- For deletion, after we perform a rotation at x, we may have to perform a rotation at some ancestor of x. Thus, we must continue to trace the path until we reach the root.

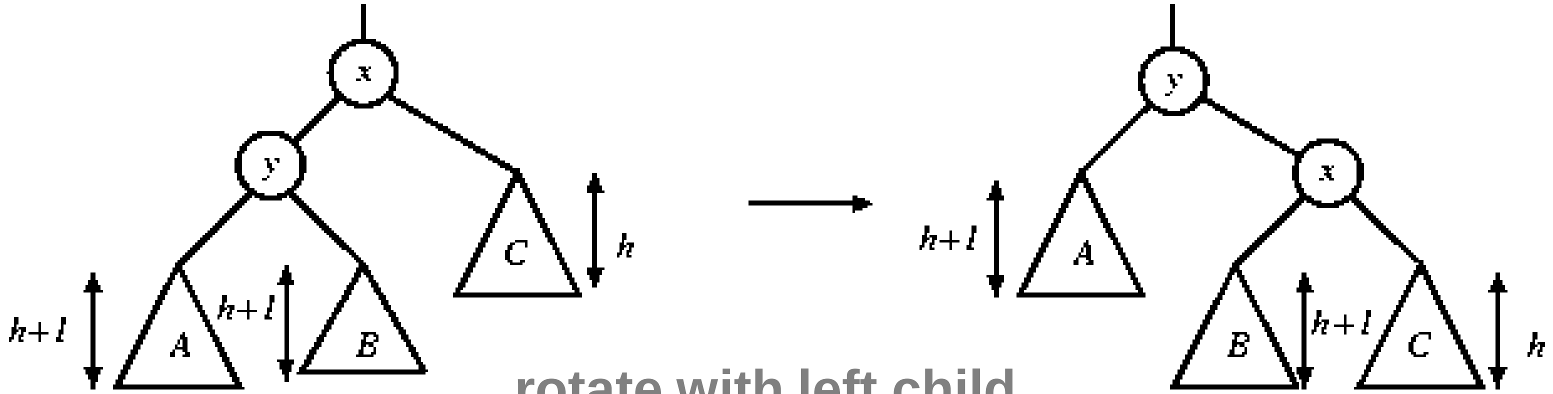
Deletion

- On closer examination: the single rotations for deletion can be divided into 4 cases (instead of 2 cases)
  - Two cases for rotate with left child
  - Two cases for rotate with right child

Single rotations in deletion

In both figures, a node is deleted in subtree C, causing the height to drop to h. The height of y is h+2. When the height of subtree A is h+1, the height of B can be h or h+1. Fortunately, the same single rotation can correct both cases.

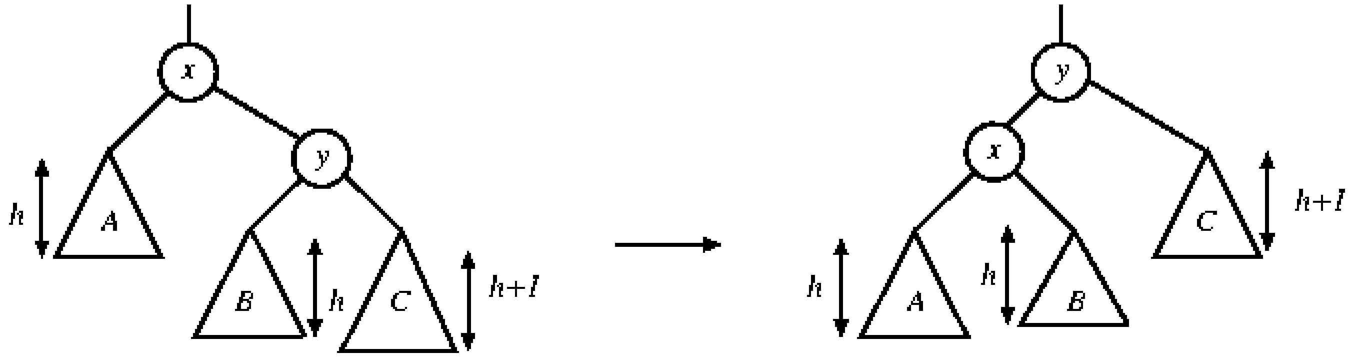


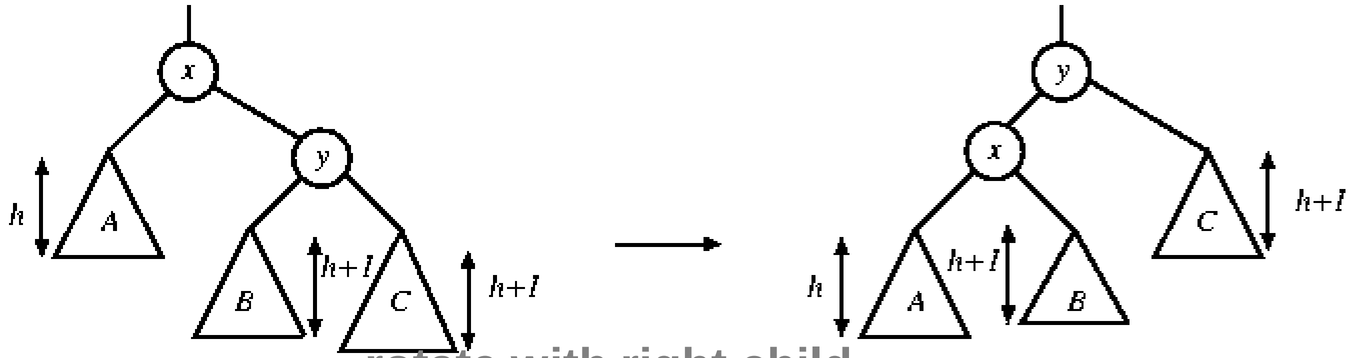


rotate with left child

Single rotations in deletion

In both figures, a node is deleted in subtree A, causing the height to drop to h. The height of y is h+2. When the height of subtree C is h+1, the height of B can be h or h+1. A single rotation can correct both cases.





rotate with right child

Rotations in deletion

- There are 4 cases for single rotations, but we do not need to distinguish among them.
- There are exactly two cases for double rotations (as in the case of insertion)
- Therefore, we can reuse exactly the same procedure for insertion to determine which rotation to perform

Insertion

- Trace from path of inserted leaf towards the root, and check if the AVL tree property is violated. Perform rotation if necessary.

- For insertion, once we perform (single or double) rotation at a node *x*, the AVL tree property is already restored. We need not to perform any rotation at any ancestor of x.

- Thus one rotation is enough to restore the AVL tree property.

- There are 4 different cases (actually 2), so don’t mix up them!

Insertion

- The time complexity to perform a rotation is O(1).
- The time complexity to insert, and find a node that violates the AVL property is dependent on the height of the tree, which is O(log(n)).
- So insertion takes O(log(n)).

Deletion

- Delete a node x as in ordinary BST (Note that x is either a leaf or x has exactly one child.).
- Check and restore the AVL tree property.

  Trace from path of deleted node towards the root, and check if the AVL tree property is violated.
- Similar to an insertion operation, there are four cases to restore the AVL tree property.

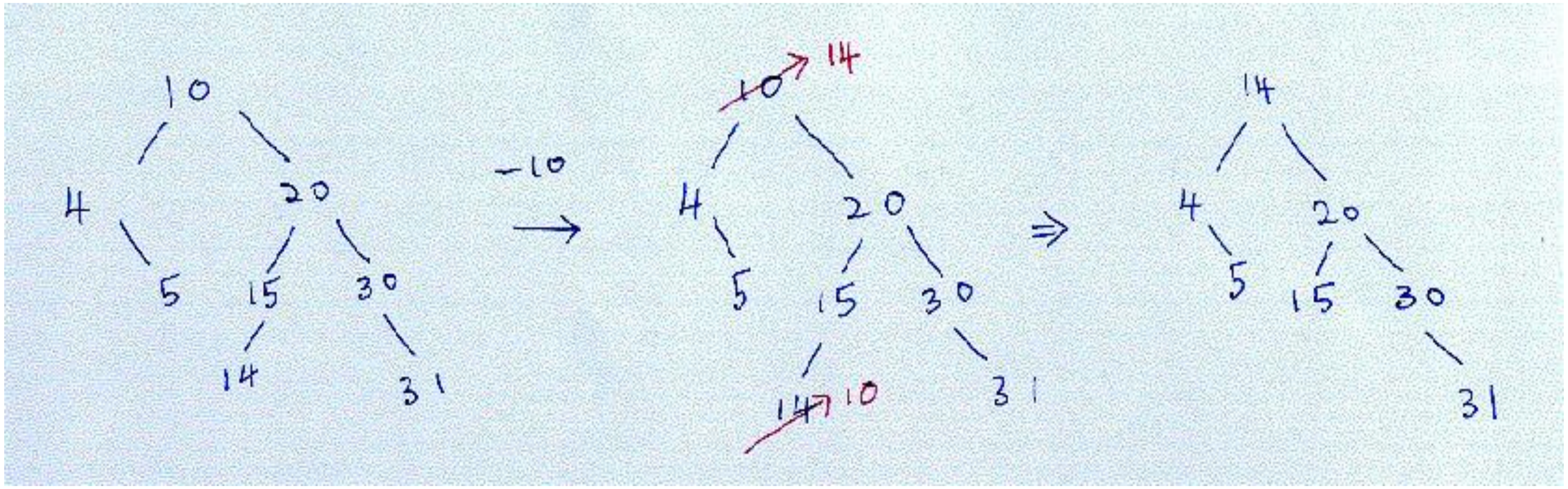
Deletion

- The only difference from insertion is that after we perform a rotation at x, we may have to perform a rotation at some ancestors of x. → It may involve several rotations.

- Therefore, we must continue to trace the path until we reach the root.

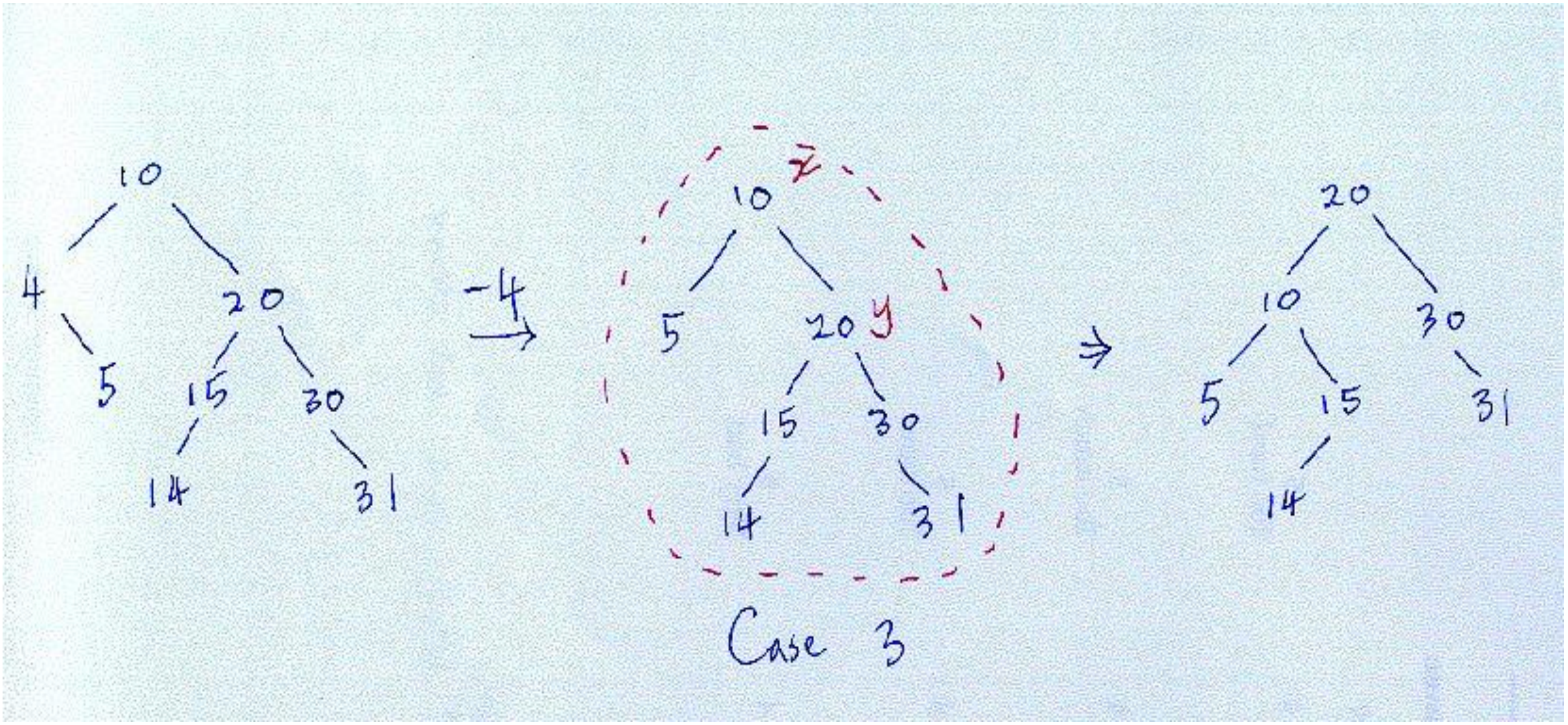
- The time complexity to delete a node is dependent on the height of the tree, which is also O(log(n)).

Deletion : no rotation



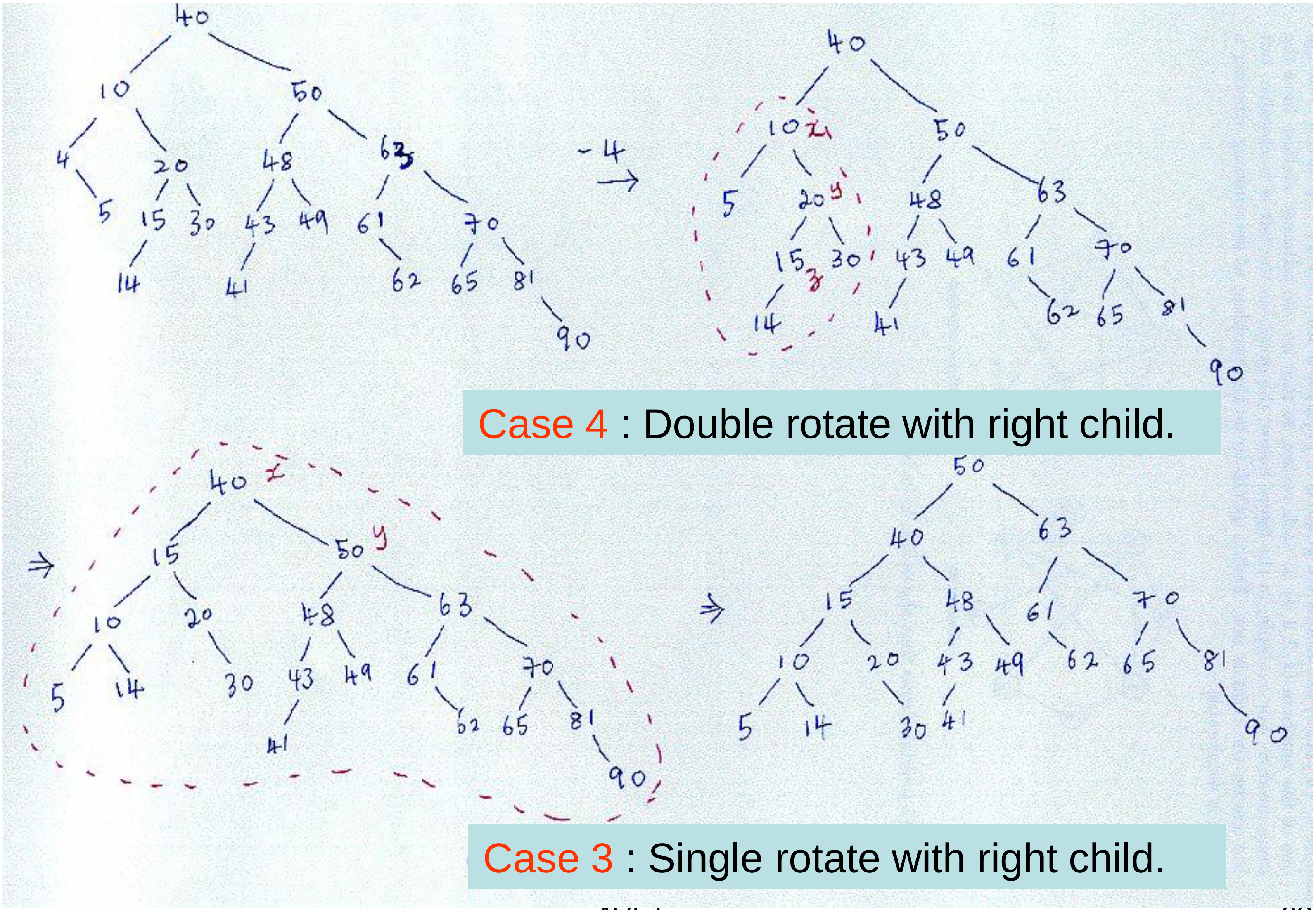
No need to rotate.

Deletion : one rotation



Case 3 : Single rotate with right child.

Deletion : several rotations



Case 4 : Double rotate with right child.

Case 3 : Single rotate with right child.

Summary of AVL Trees

- Maintains a balanced tree by posing an AVL tree property:
  - → guarantees the height of the AVL tree be O(log n)
  - → implies that functions `search`, `min`, and `max`, `insert`, and `delete` will be performed in O(logn)
- Modifies the insertion and deletion routine
  - ◆ Performs single or double rotation to restore the AVL tree property if necessary.
  - ◆ Requires a little more work for insertion and deletion.