Laboratory 7: AVL Tree Operations

CSC205A Data structures and Algorithms Laboratory B. Tech. 2015

Vaishali R Kulkarni

Department of Computer Science and Engineering
Faculty of Engineering and Technology
M. S. Ramaiah University of Applied Sciences
Email: vaishali.cs.et@msruas.ac.in
Tel: +91-80-4906-5555 (2212) WWW: www.msruas.ac.in

AVL Tree Opearations

Introduction and Purpose of Experiment

- Advanced data structures such as AVL trees help to overcome many limitations of binary search trees.
- In this experiment students will be able to implement and analyse the efficiency of AVL trees and its operations.

Aim:

- To design and develop algorithms for creating and demonstrating AVL tree operations

Binary Search Tree algorithms

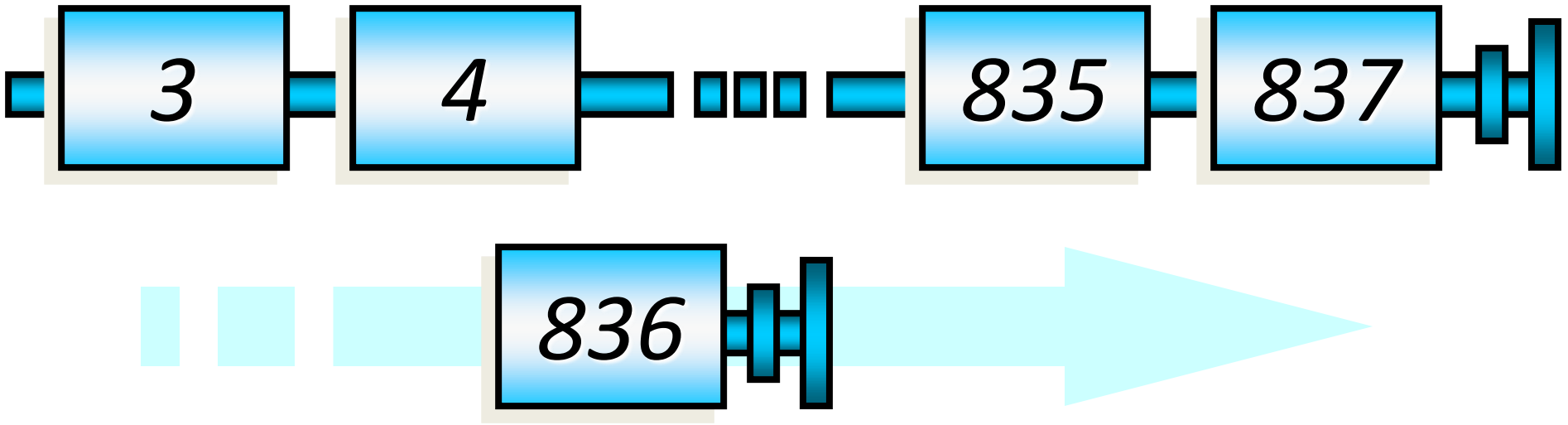
Objectives:

At the end of this lab, the student will be able to

- Create AVL tree
- Demonstrate inserting and deleting a node from AVL tree.

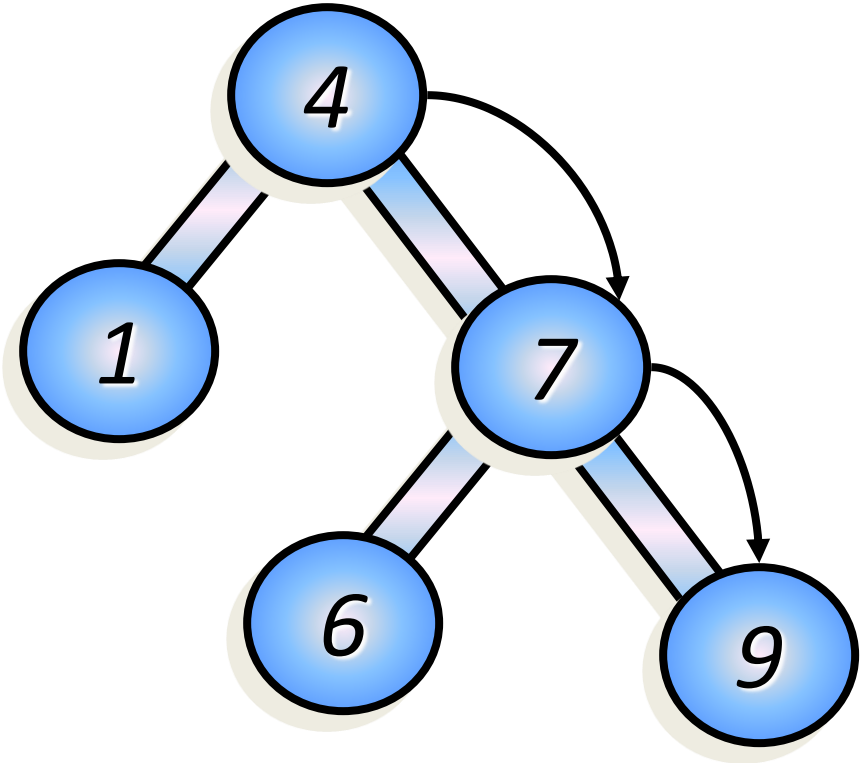
Theory behind the experiment

Motivation: Linked lists and queues work well enough for most applications, but provide slow service for large data sets.



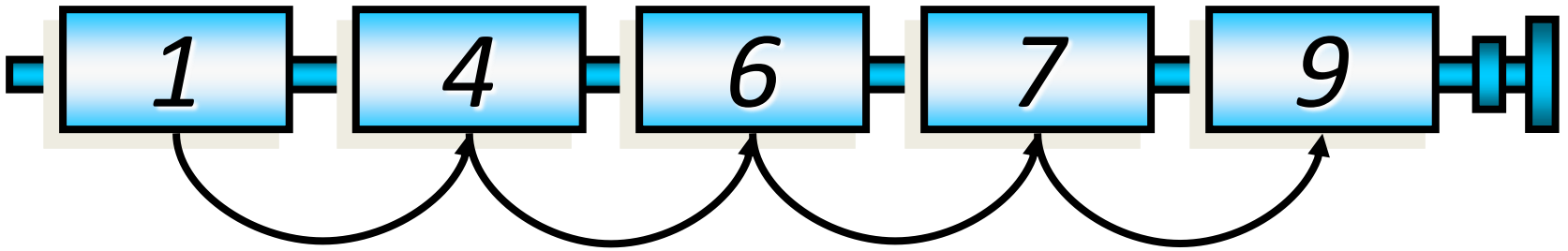
Ordered insertion takes too long for large sets.

Solution: Trees

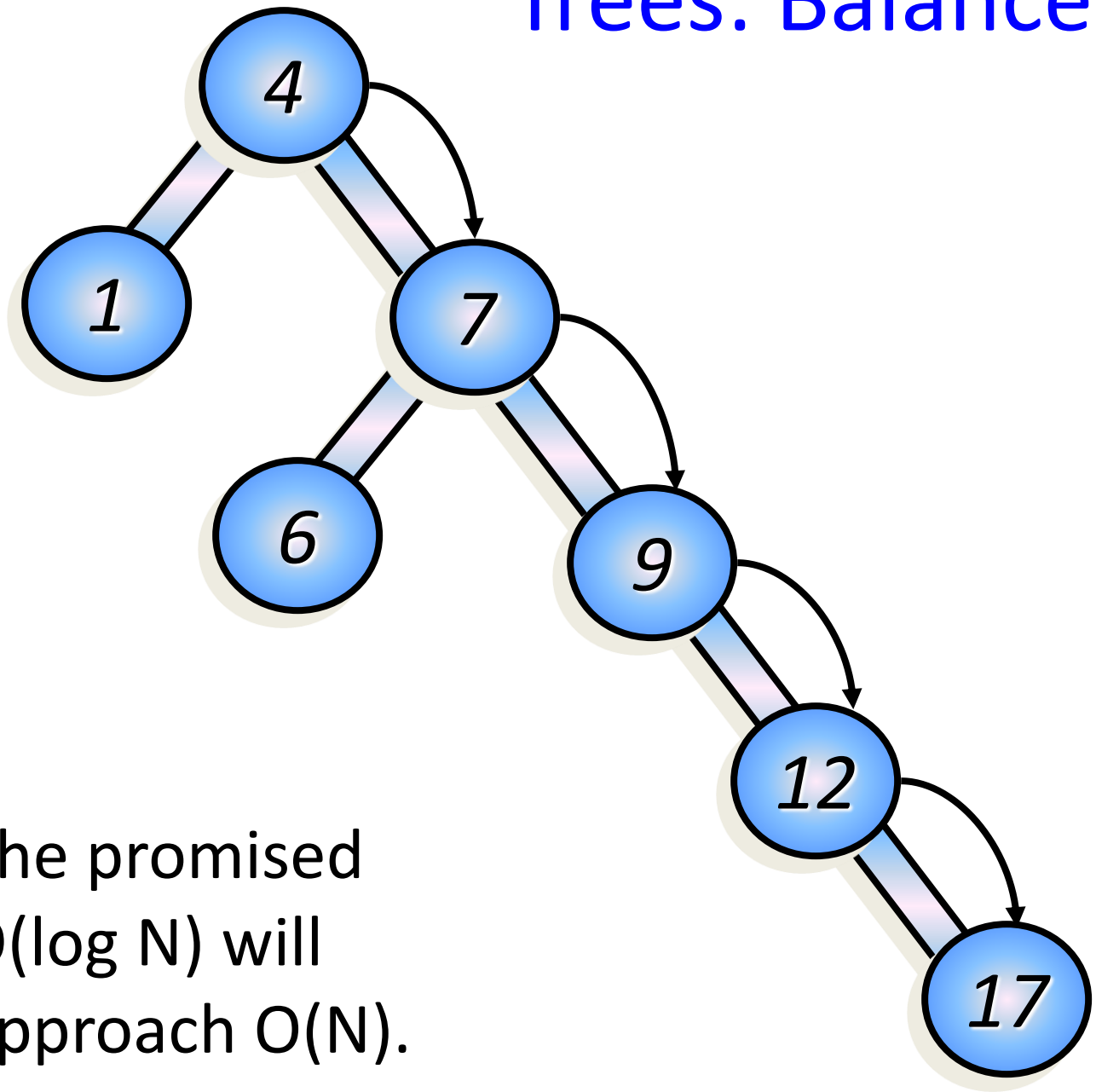


Binary Trees provide quick access to data

O (logN)
vs
O (N)



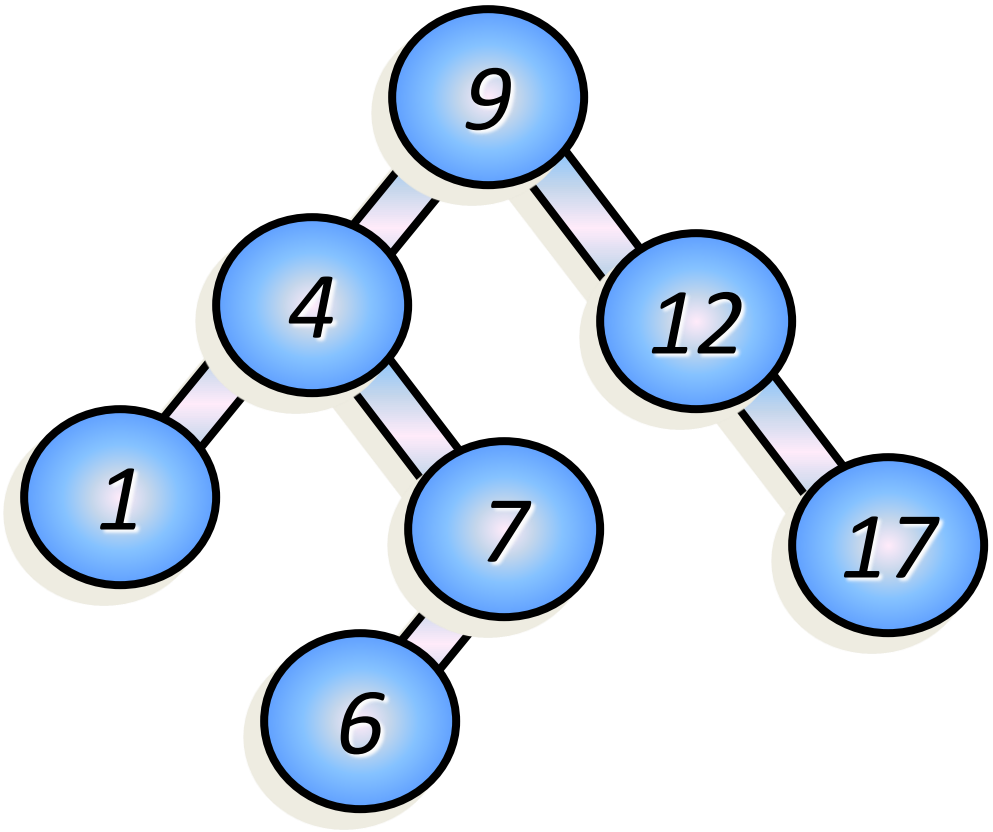
Trees: Balance



Consider an imbalanced Binary Search Tree. Search performance degrades back into a linked list.

The promised O(log N) will approach O(N).

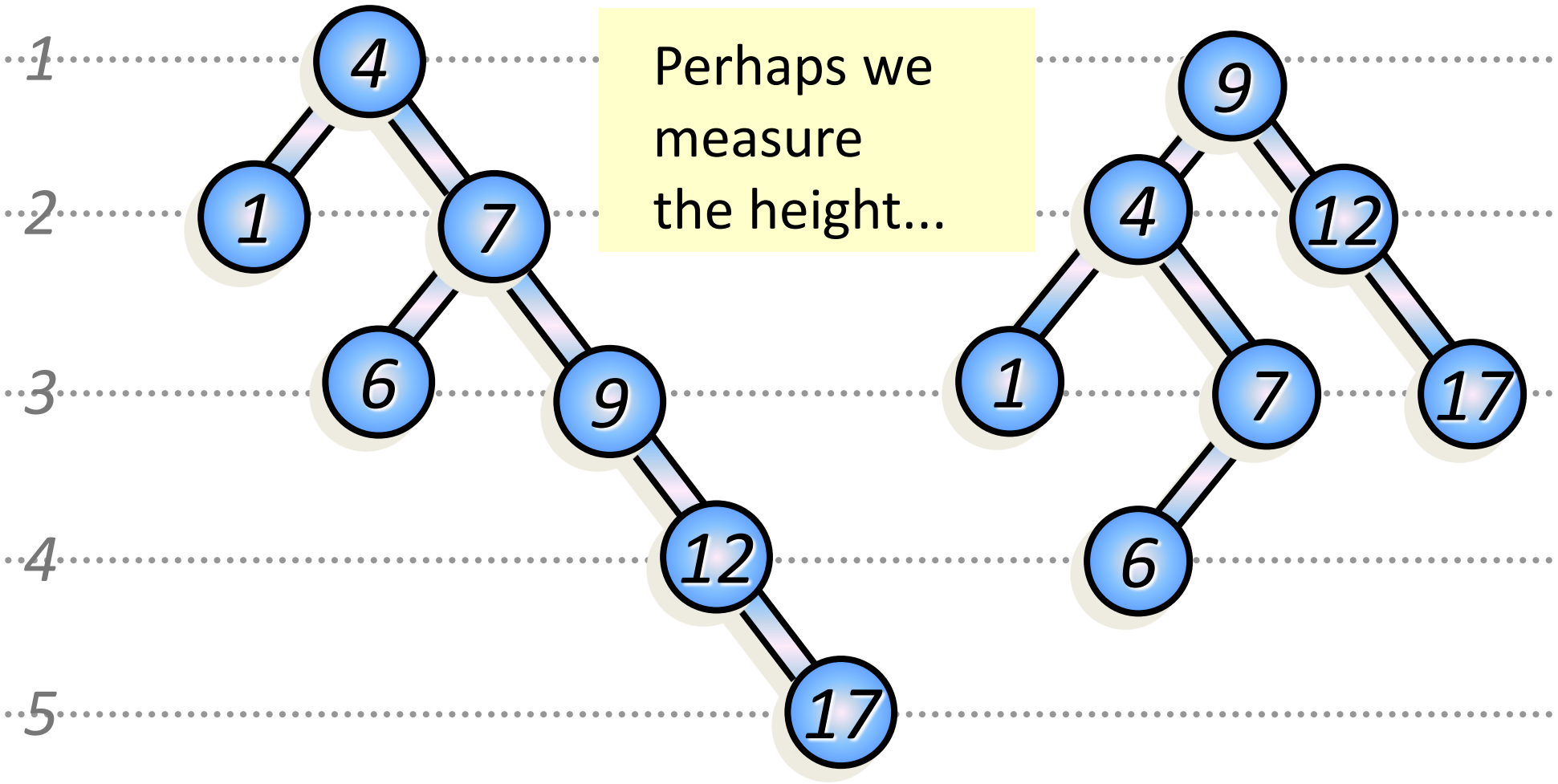
Balance



If we had some way of keeping the tree "in balance", our searches would be closer to O(log N)

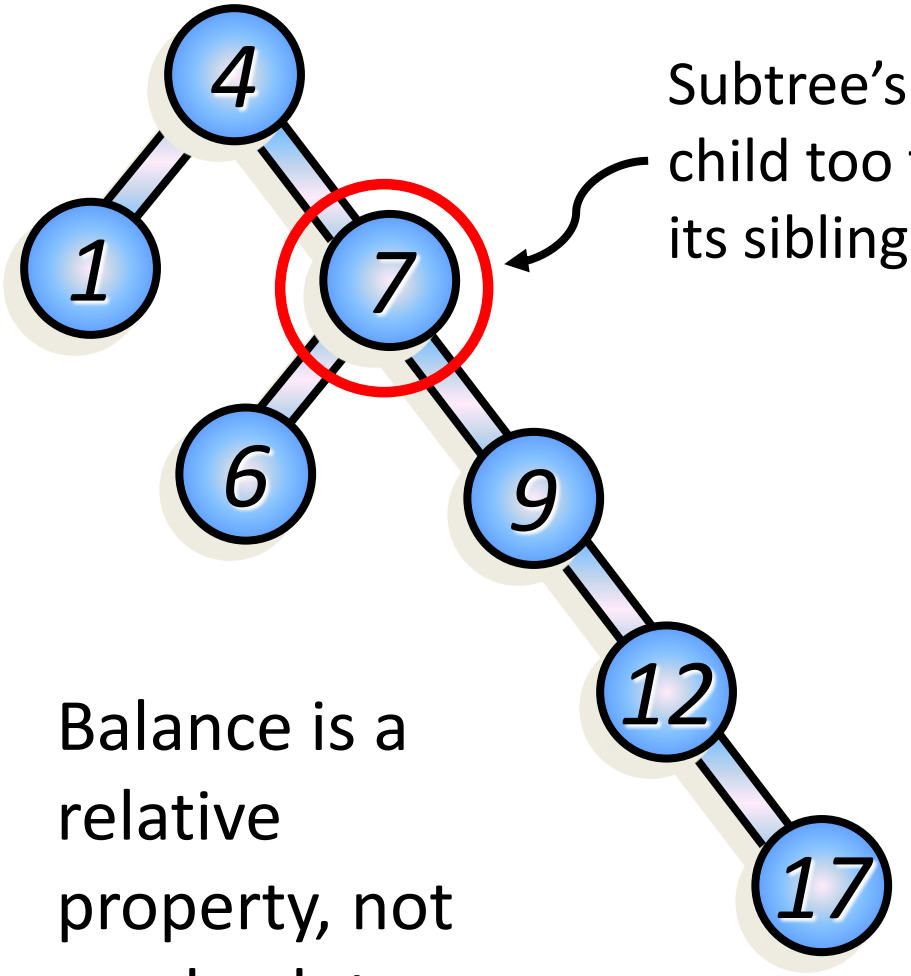
But how does one measure balance?

Measuring "Balance"

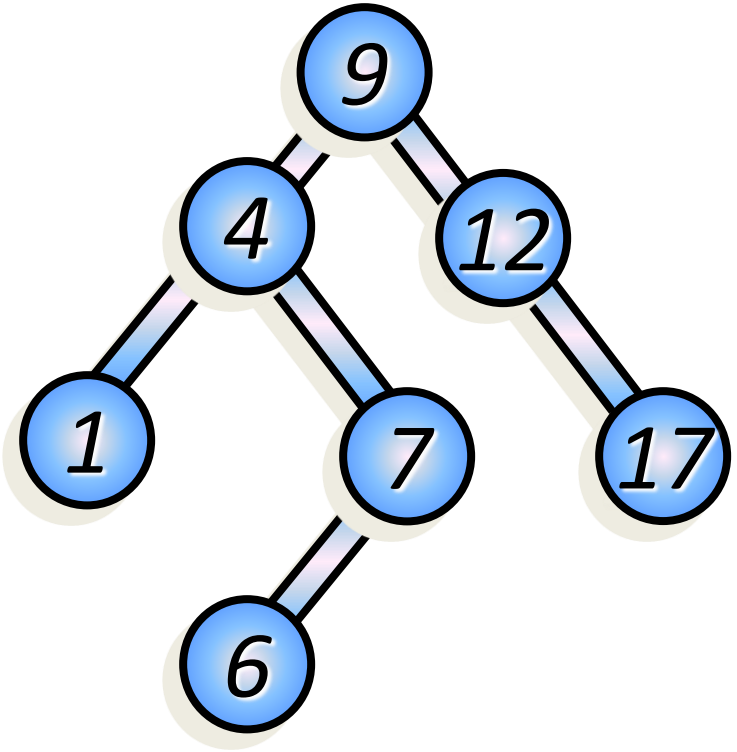


...but the height alone does not give us a indication of imbalance

Balance: The Key



Subtree's right child too tall for its sibling



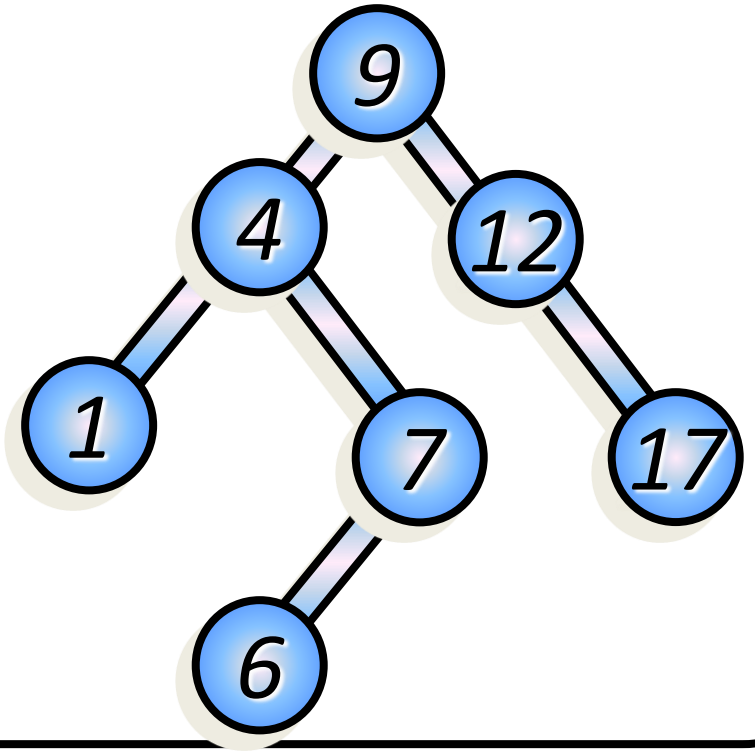
Balance is a relative property, not an absolute measure...

Measuring “Balance”

Adelson-Velskii and Landis suggested that, since trees may be defined recursively, balance may also be determined recursively.

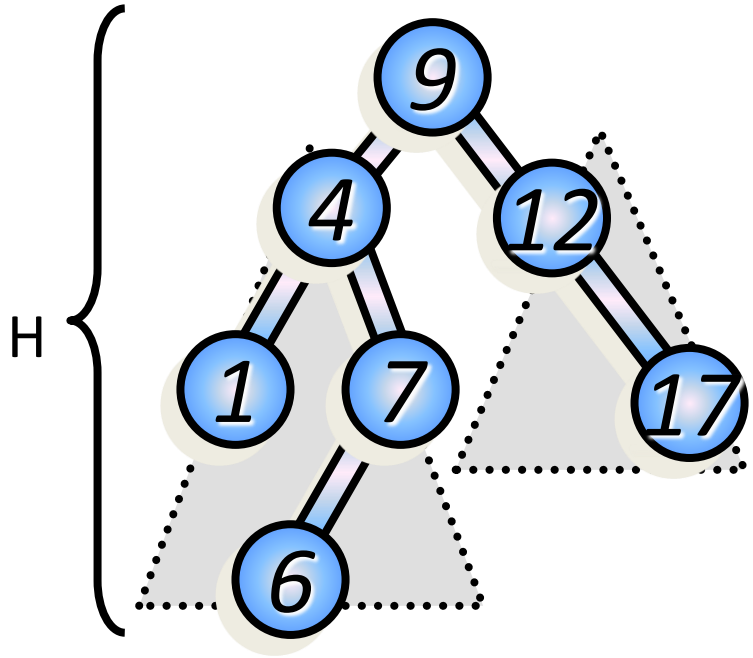
Internal nodes are also roots of subtrees.

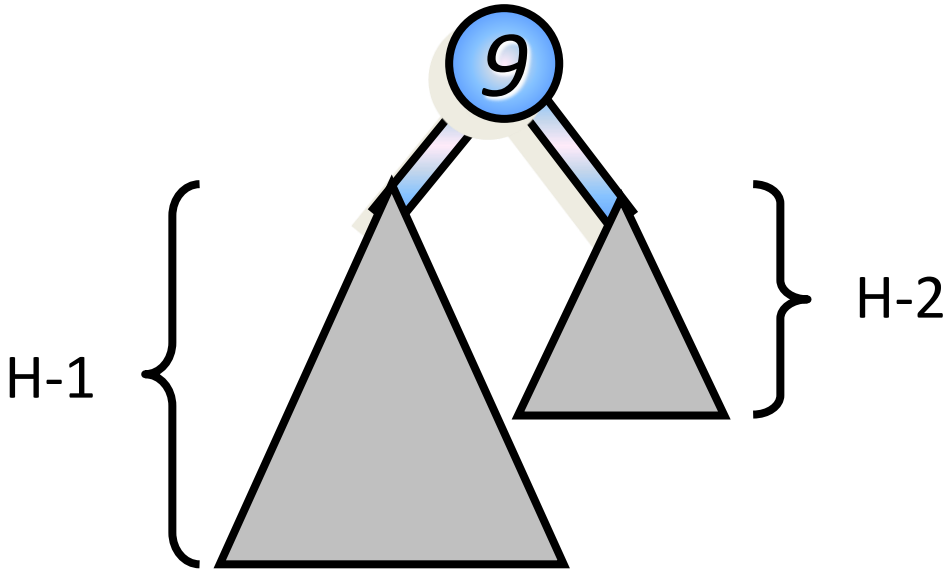
If each node’s subtree has a similar height, the overall tree is balanced



Measuring Balance

Thus, the actual numerical height does not determine balance; rather, it's the ***relative*** height of both subtrees.





Balanced Tree Test: subtrees differ in height by no more than one.

This recursively holds true for all nodes!

Balance Factor

To measure the balance of a node, there are two steps:

Record the *node height (NH)*
--an absolute numeric measure

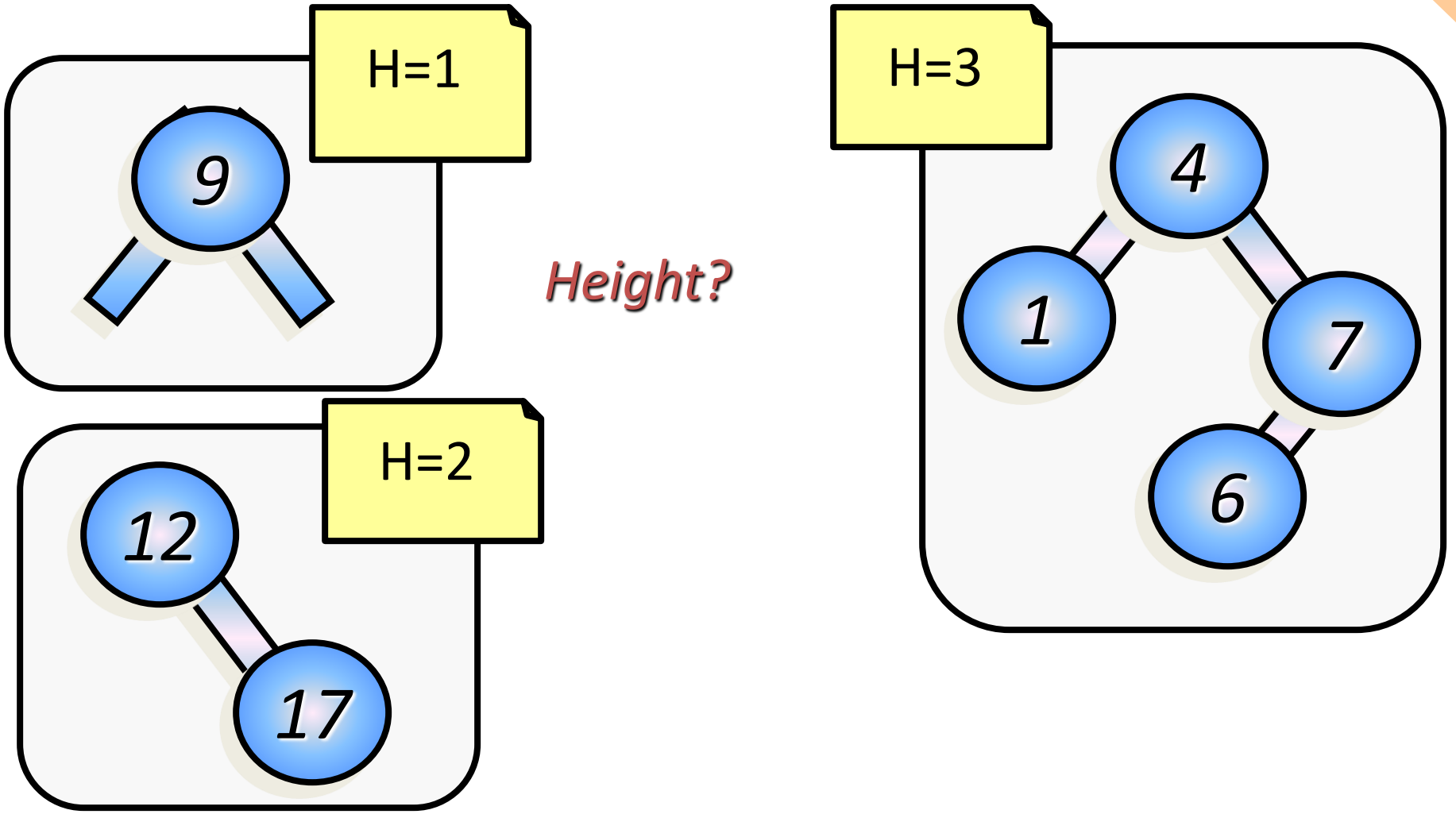
Record the node's *balance factor (BF)*
--a relative measure

The node height alone does not indicate balance: we have to compare it to the height of other nodes.

Node / Tree Height

STEP 1

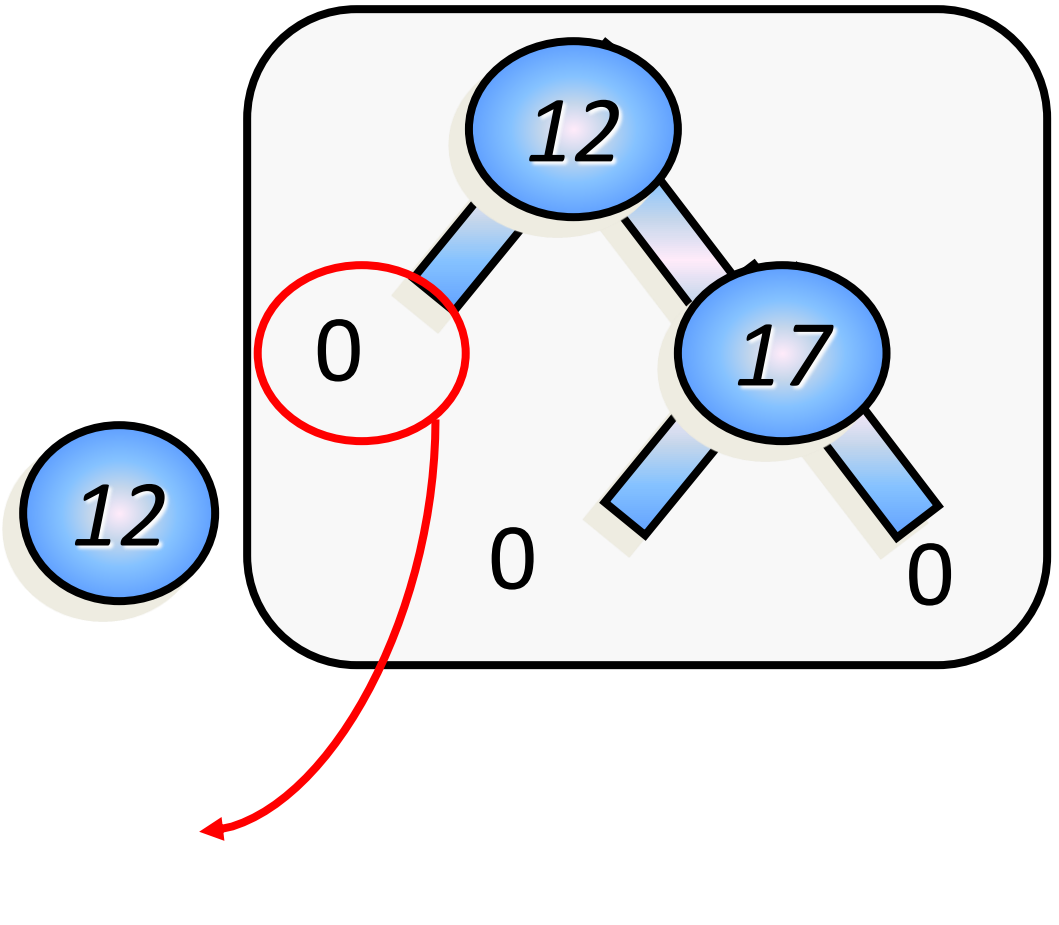
Tree height is easy, Here are three trees.



Node / Tree Heights for AVLs

By convention, a null reference will return a height of zero (0).

Therefore:
The Balance Factor of



is Ht of 17 inus (0)

= 1 – (0)

= +1

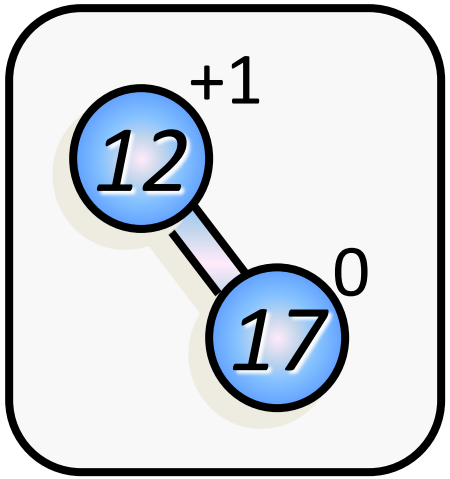
Balance Factor

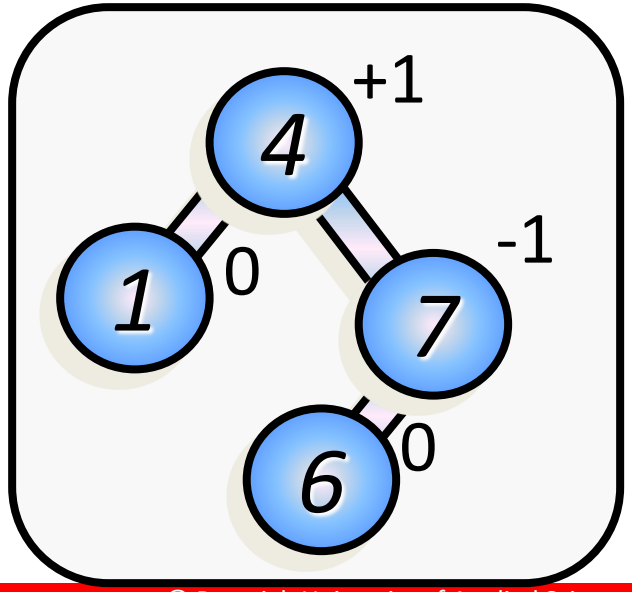
STEP 2

Remember that balance is ***relative***.

So we define the Balance Factor (BF) at:

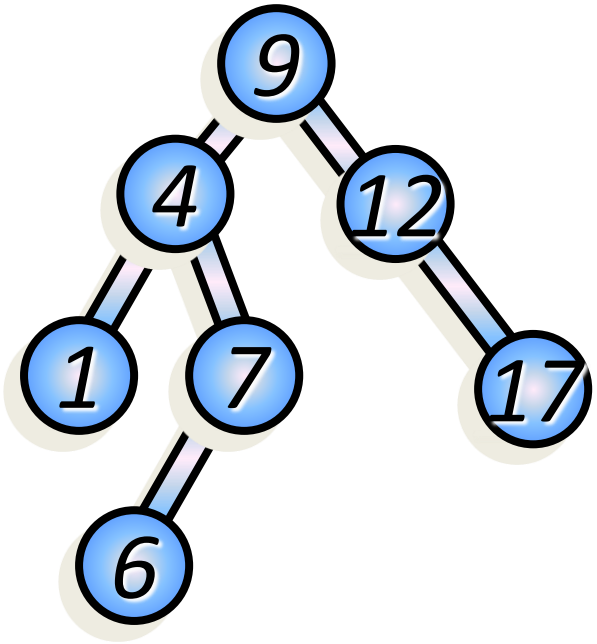
BF = (right subtree height - left subtree height)

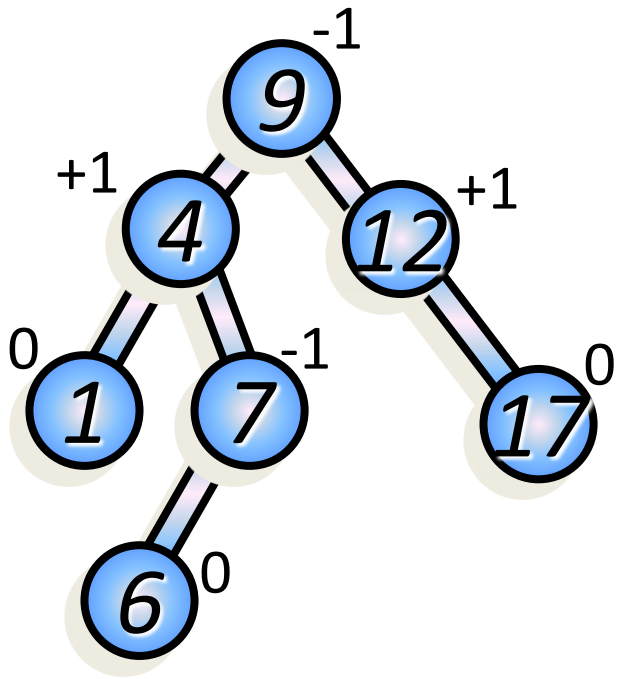




Balance Factor

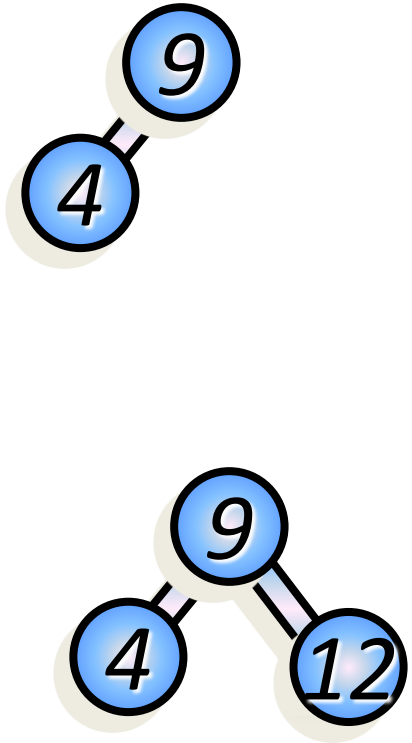
Each node has a balance factor. Remember, it's based on (right ht - left ht).

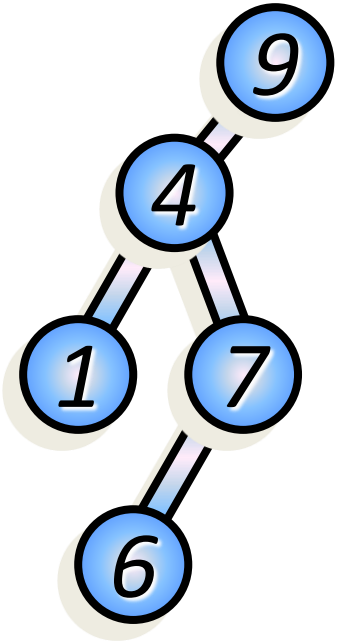


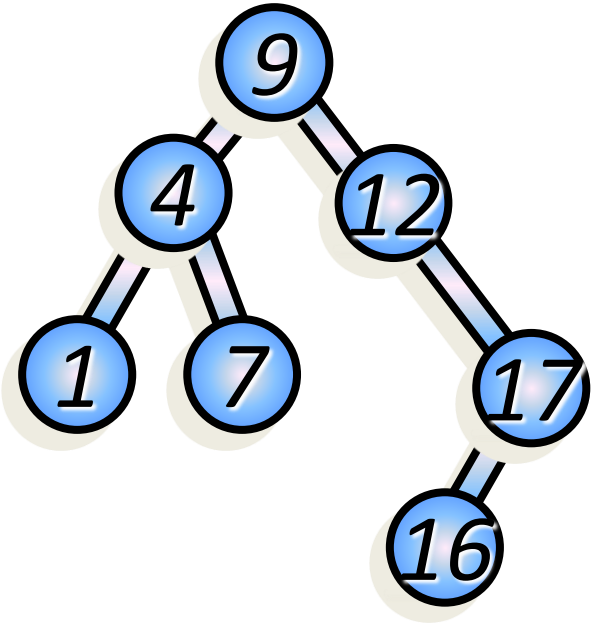


Balance Factor

Calculate the balance factor for each node in each tree:



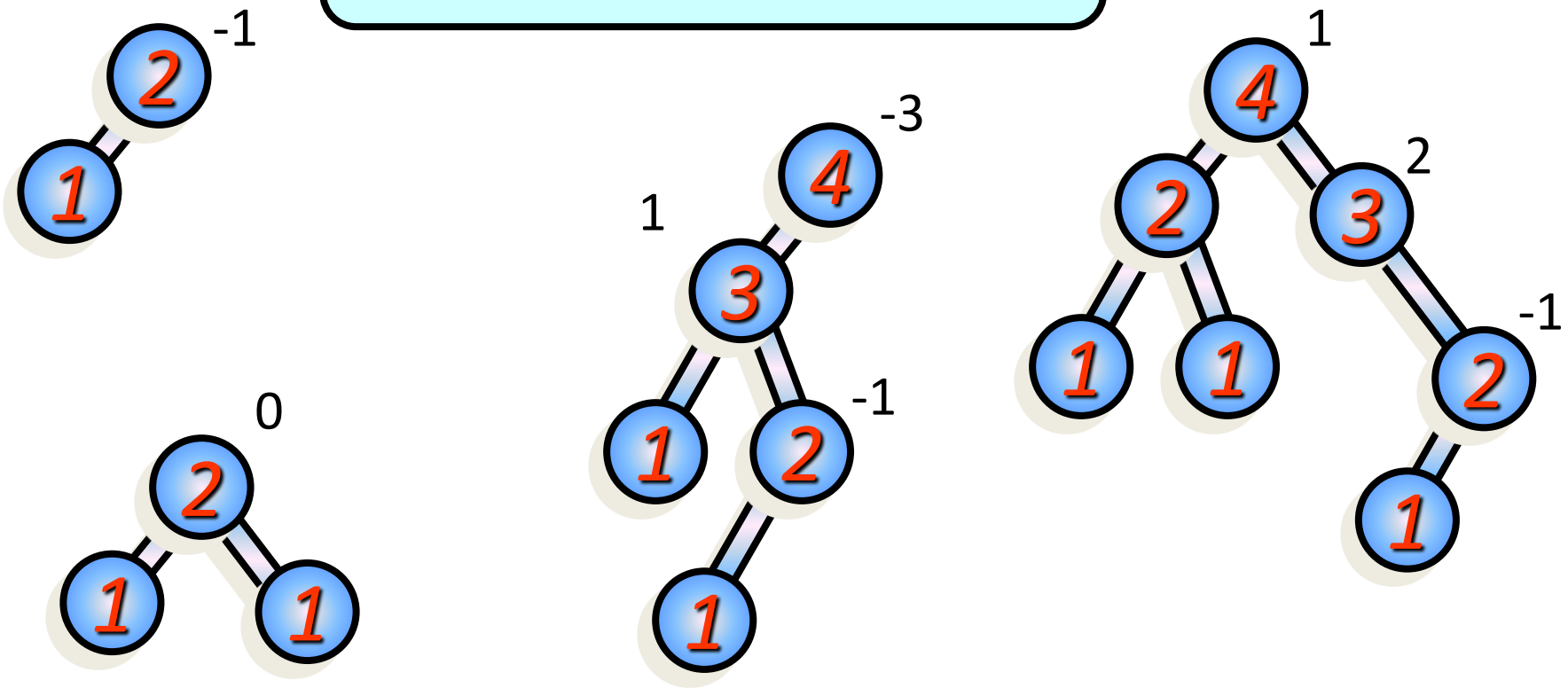




Balance factor

Calculate the balance factor for each node in each tree:

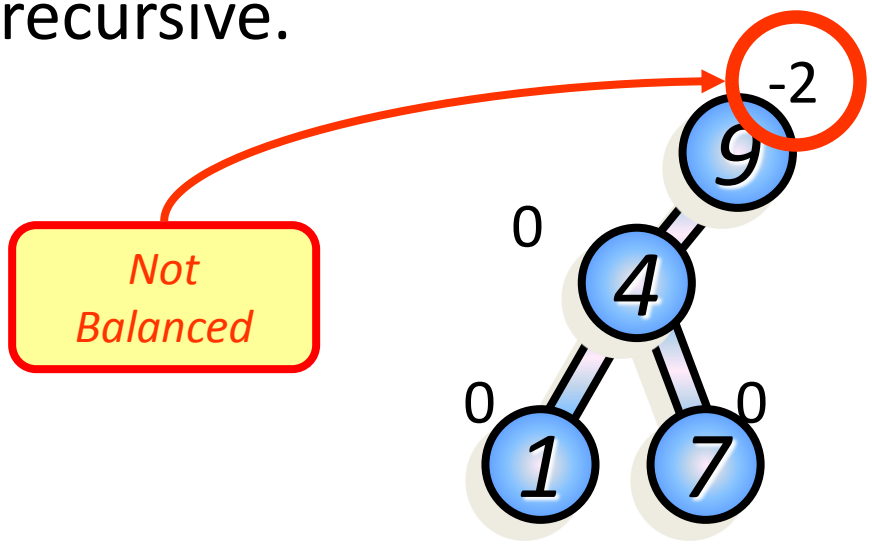
Here, the height is written inside the node, covering the key value it holds.

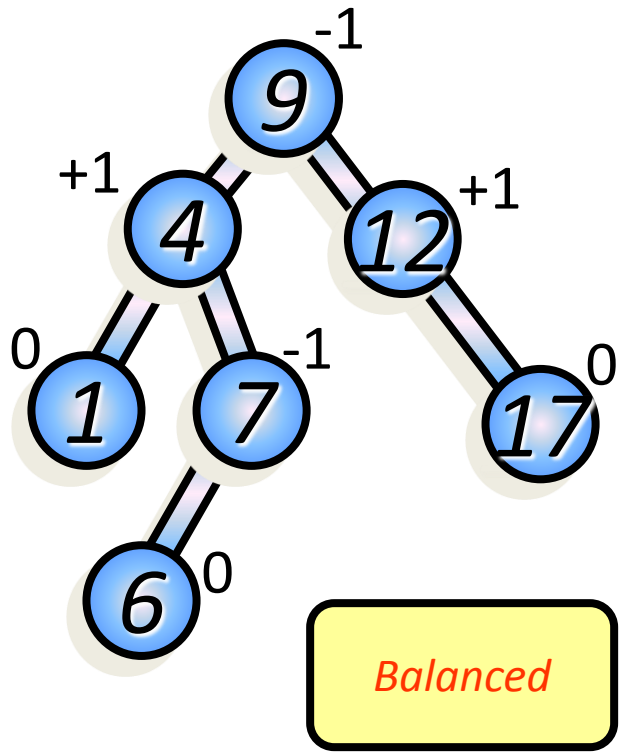


Using Balance Factors

How do the Balance Factors help?

Recall: Adelson-Velskii and Landis used a recursive definition of trees. Our BF measure is likewise recursive.





If any BF is > +1 or < -1, there's imbalance

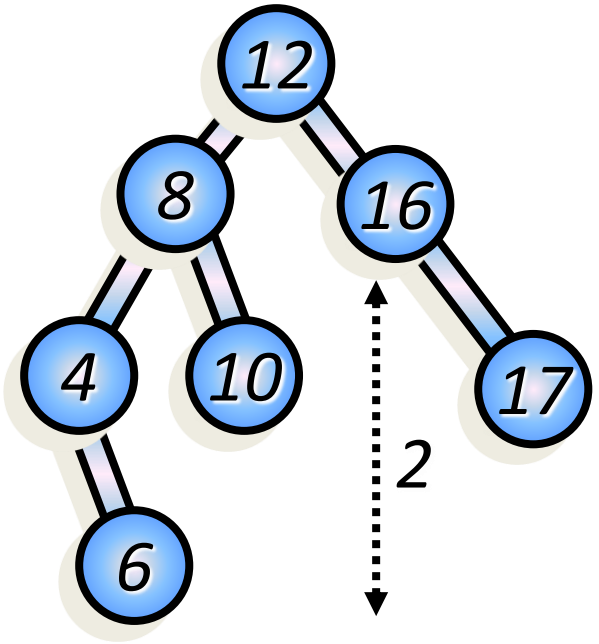
AVL's "Weak" Balancing

- **A *fully* balanced tree guarantees that, *for nodes on each level*, all left and right subtrees have the same height, and imposes this recursively.**

- **AVL trees use a weaker balance definition which still maintains the logarithmic depth requirement:**

  - **for any node in an AVL tree, the height of the left and right nodes differs by at most 1.**

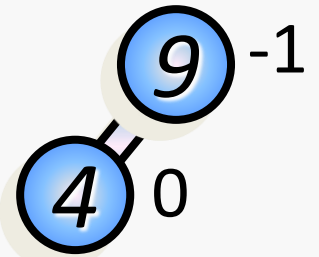
- **For example, this is a valid AVL tree, but not strictly balanced:**



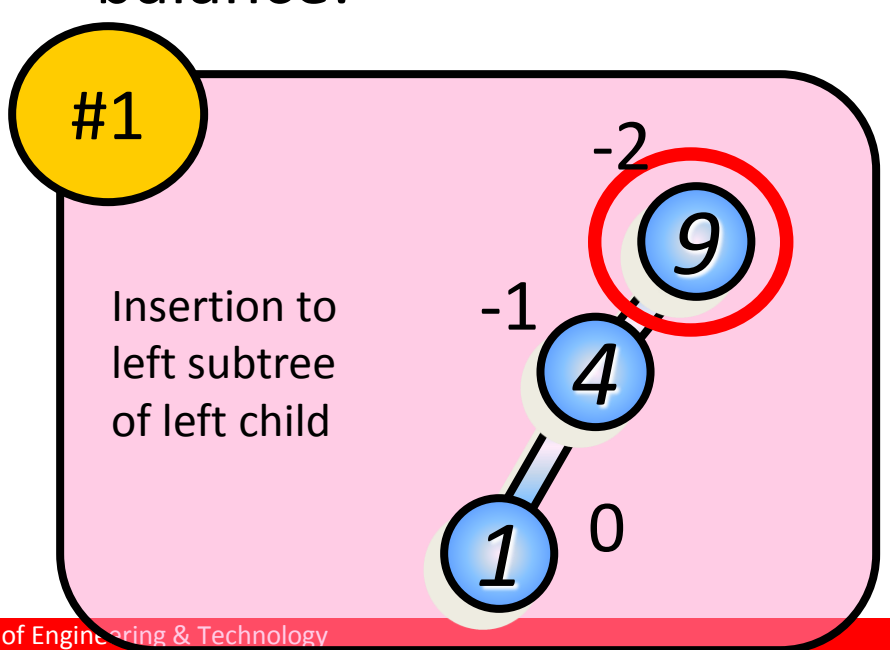
Identifying immbalance Scenarios

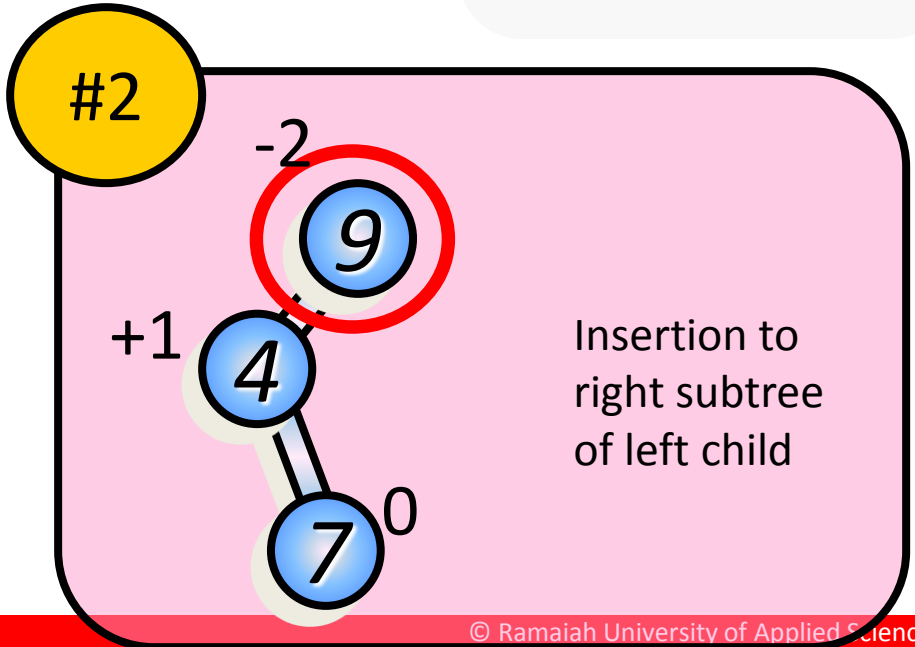
The key to fixing an unbalanced tree is knowing what caused it to break. Let's inductively look at what can cause a good tree to go bad.

Given an AVL balanced tree:



Two possible *left* insertions could break balance:

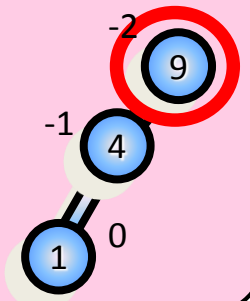




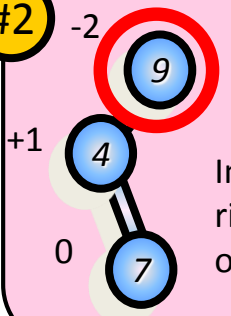
Patterns of Problems

#1

Insertion to left subtree of left child



#2

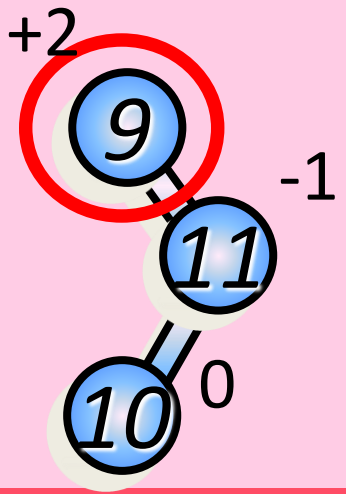


Insertion to right subtree of left child

… and the mirror of these insertions

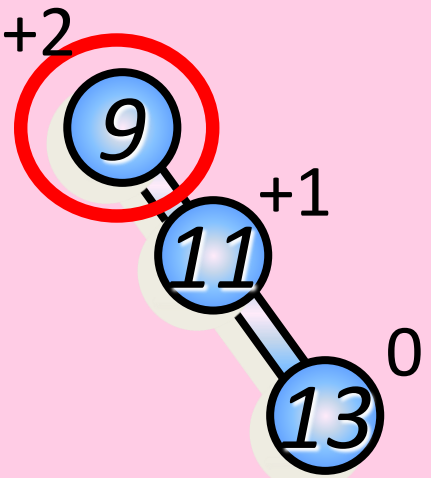
#3

Insertion to left subtree of right child



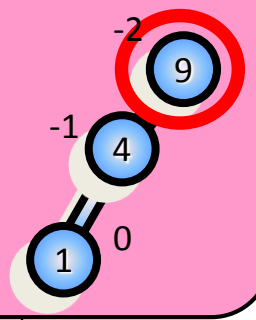
#4

Insertion to right subtree of right child

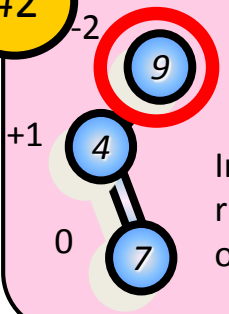


#1

Insertion to left subtree of left child



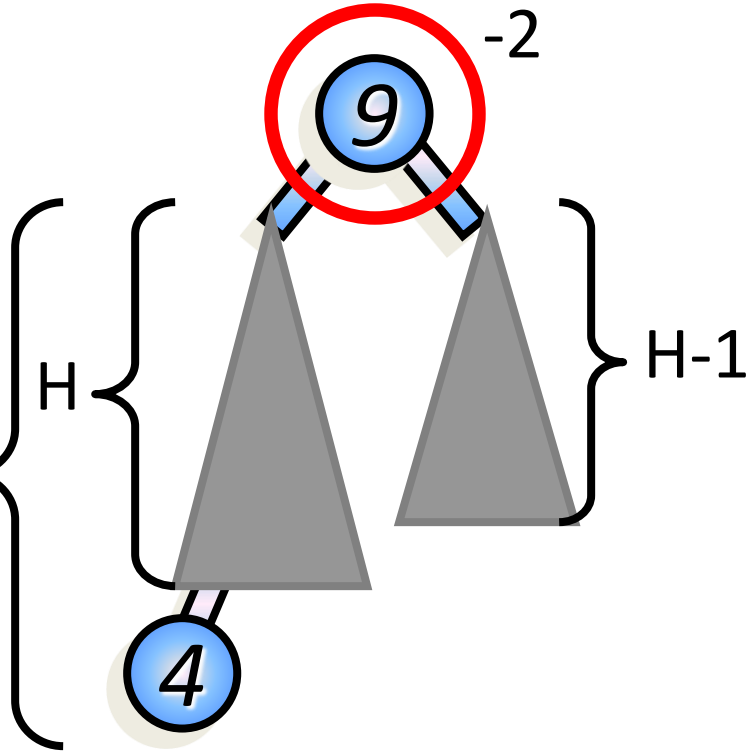
#2



Insertion to right subtree of left child

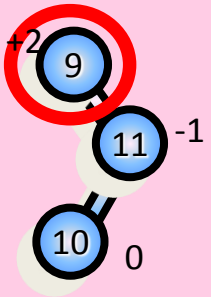
Imbalance found regardless of height

matches



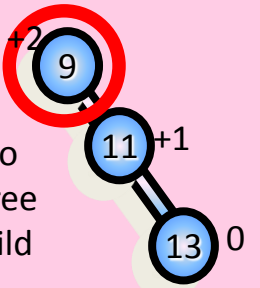
(H-1) - (H+1) = -2

#3



Insertion to left subtree of right child

#4

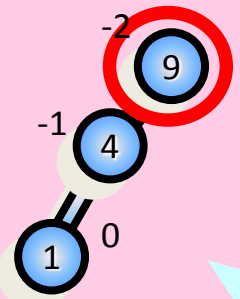


Insertion to right subtree of right child

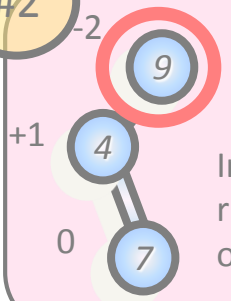
Given our recursive definition of trees, we know these four scenarios are complete. The same insertion into a larger AVL balanced tree does not create a new category.

#1

Insertion to left subtree of left child



#2

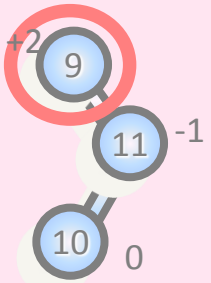


Insertion to right subtree of left child

Let's look at two insertions that cause imbalance. Recall they are mirrors.

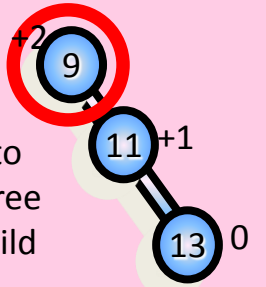
They both deal with "outside" insertions on the tree's exterior face: the left-left and right-right insertions.

#3



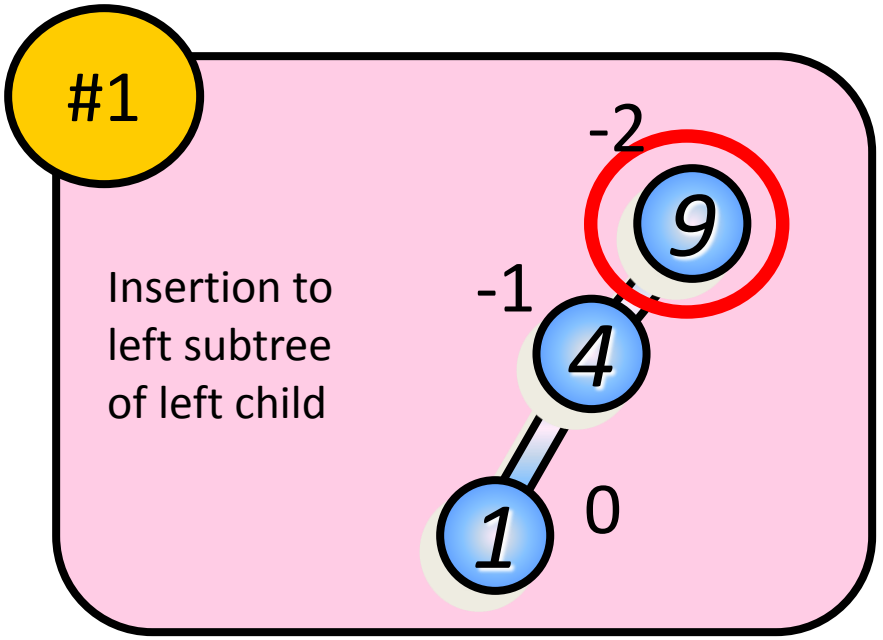
Insertion to left subtree of right child

#4



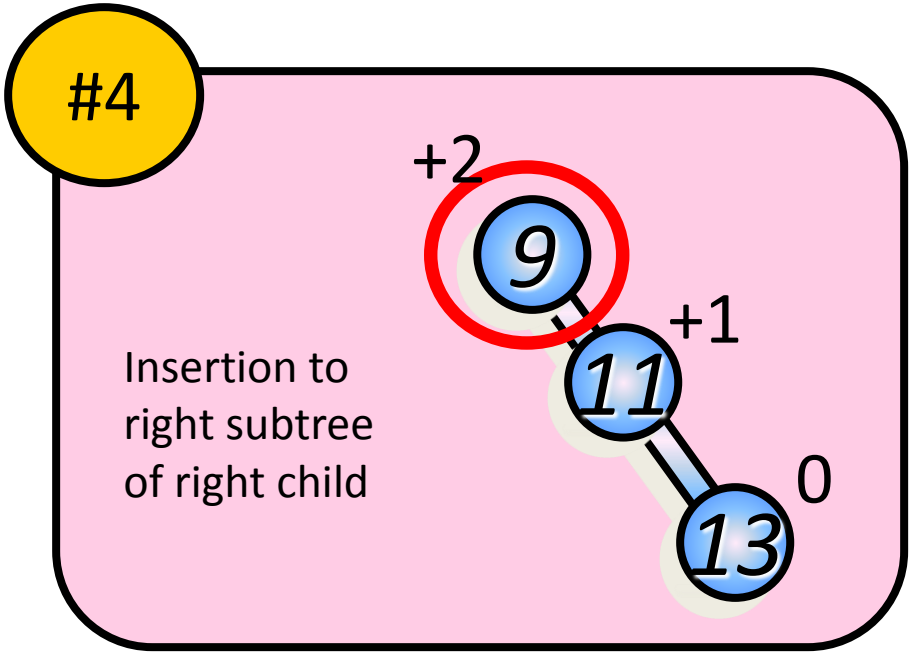
Insertion to right subtree of right child

Single Rotation



These insertions can be cured by swapping parent and child . . .

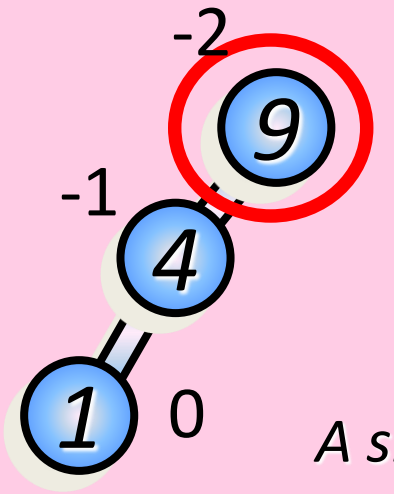
. . . provided you maintain search order. (An AVL tree is a BST)

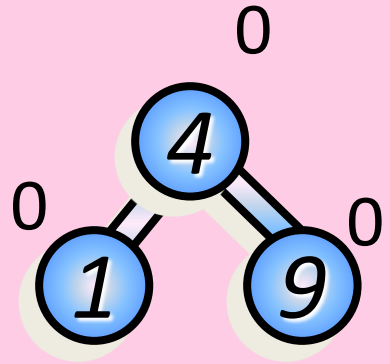


#1

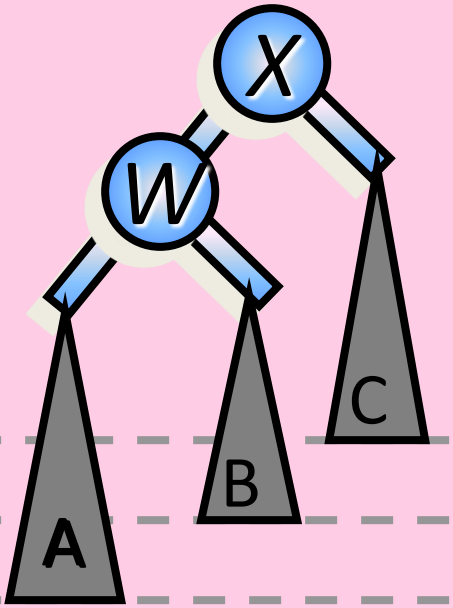
Single Rotation: Right

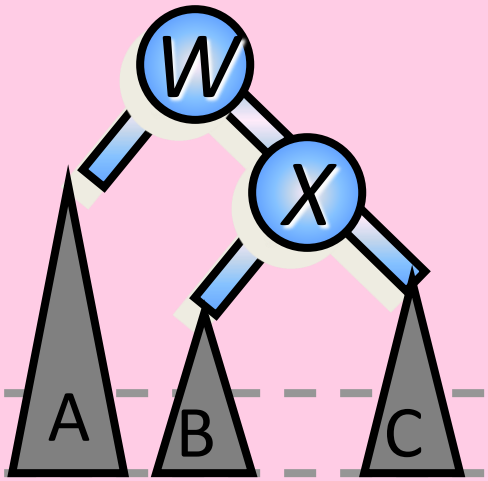
Insertion to left subtree of left child

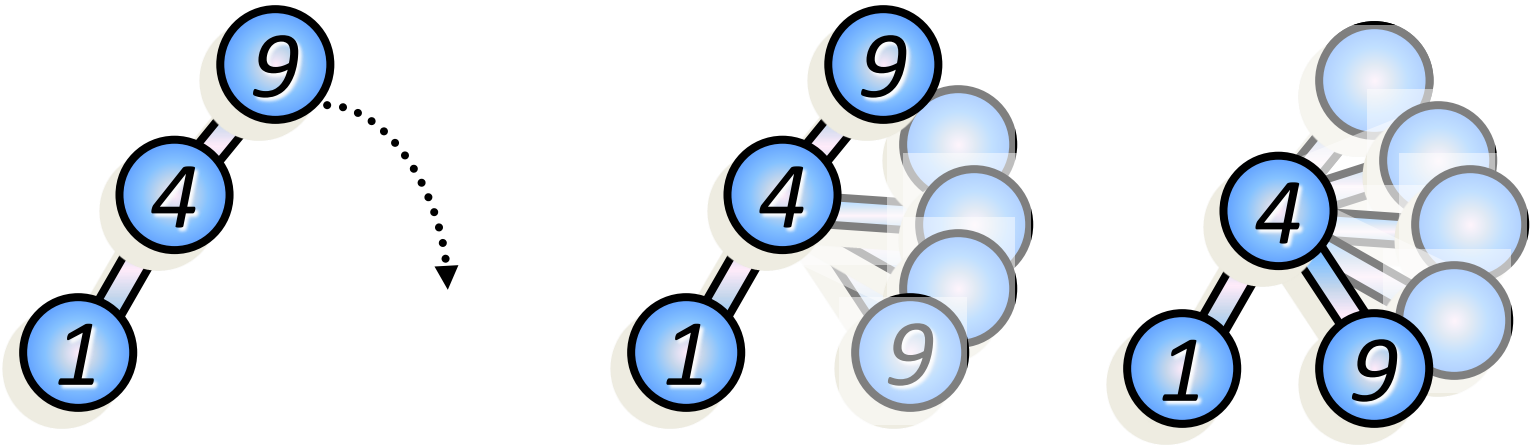




A single rotation round the unbalanced node restores AVL balance in all cases





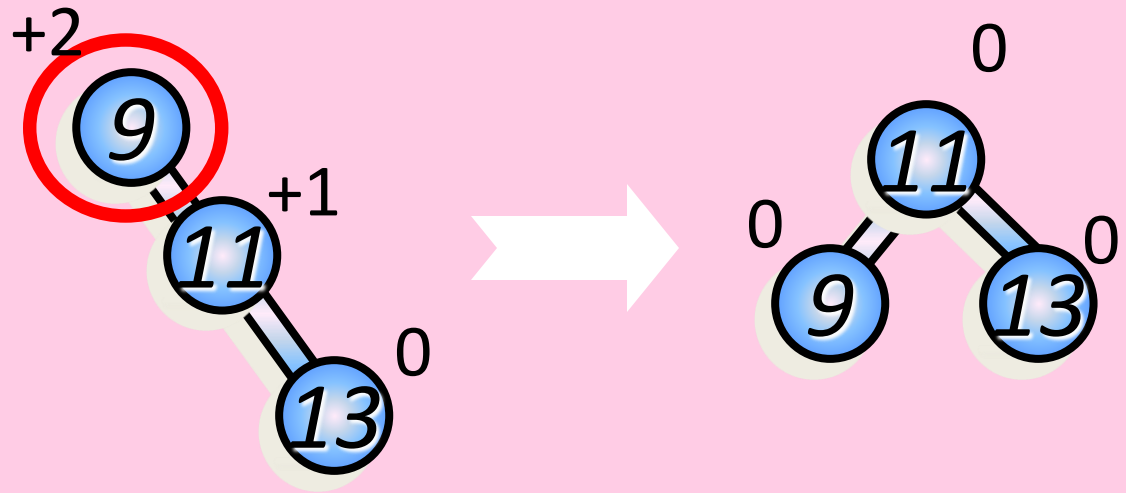


Think of rotations - Imagine the joints at the ‘4’ node starting to flex and articulate.

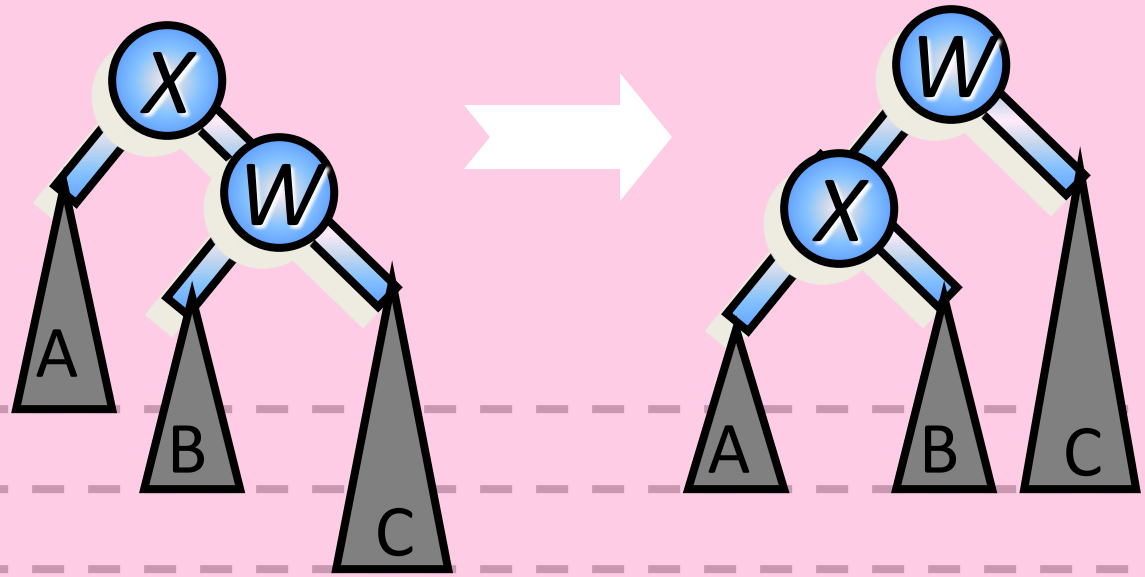
Single Rotation: Left

#4

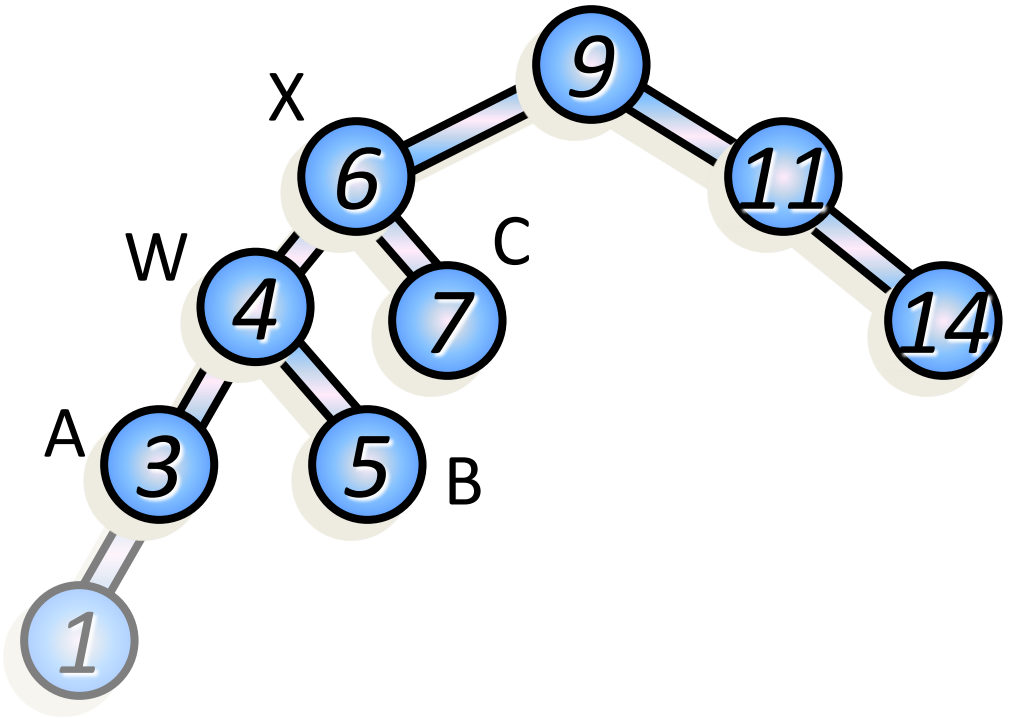
Insertion to right subtree of right child



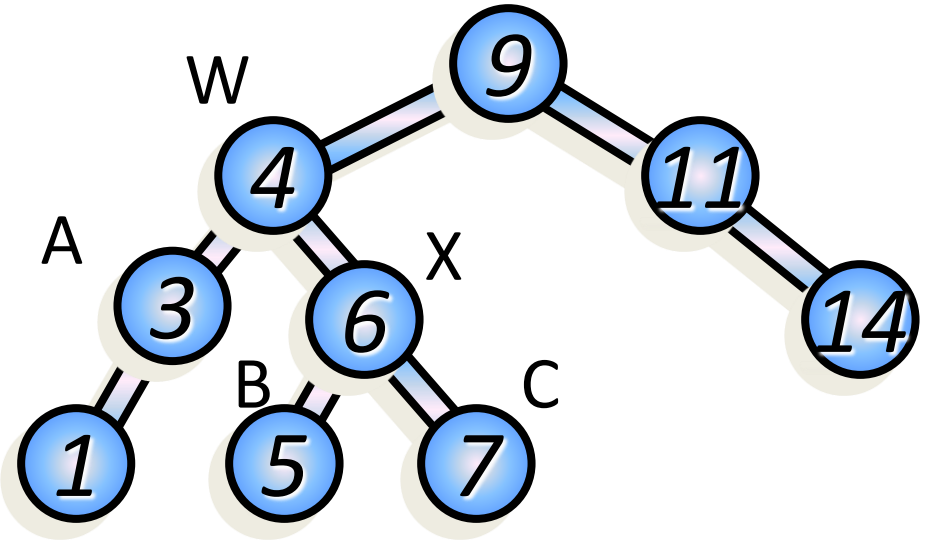
A similar rotation is used for right-right insertions that cause an imbalance



Word of caution

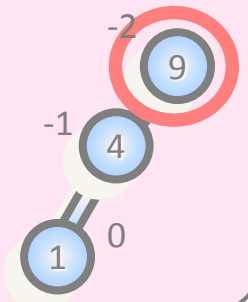


The single rotations are simple, but you have to keep track of child references.

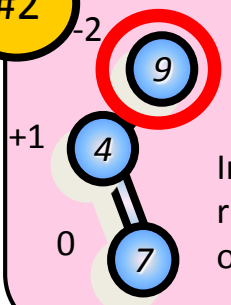


#1

Insertion to left subtree of left child



#2

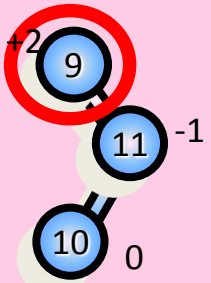


Insertion to right subtree of left child

Let's look at the other two insertions causing problems. Recall they are mirrors.

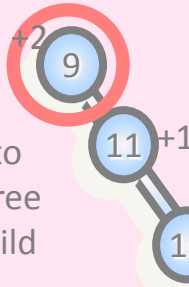
They both deal with "inside" insertions on the tree's interior: right-left and left-right

#3



Insertion to left subtree of right child

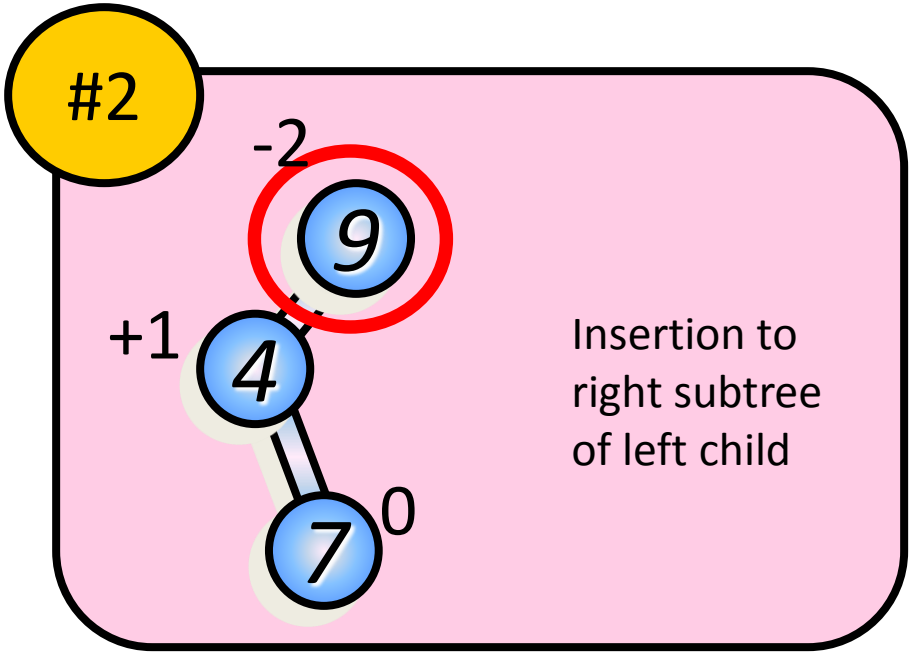
#4

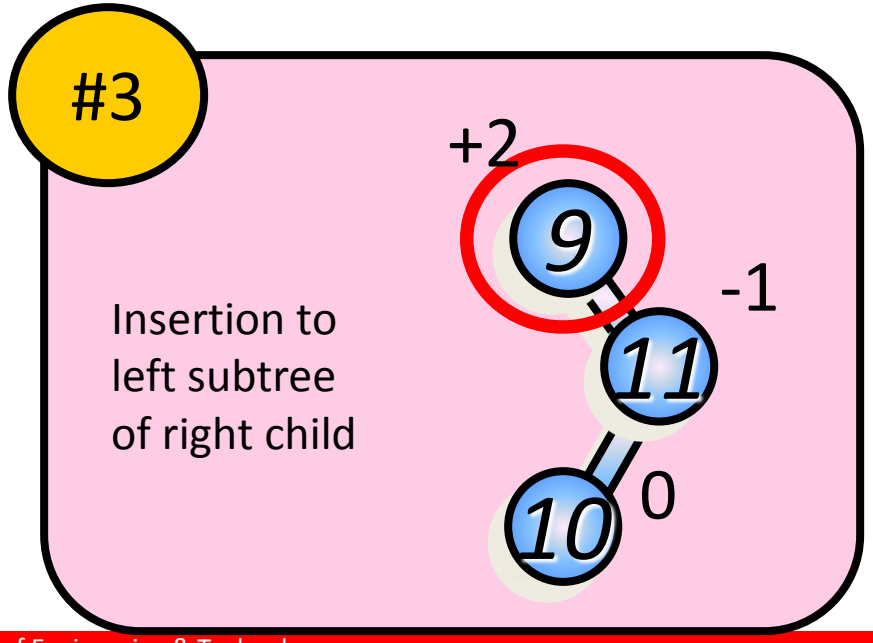


Insertion to right subtree of right child

Double Rotation

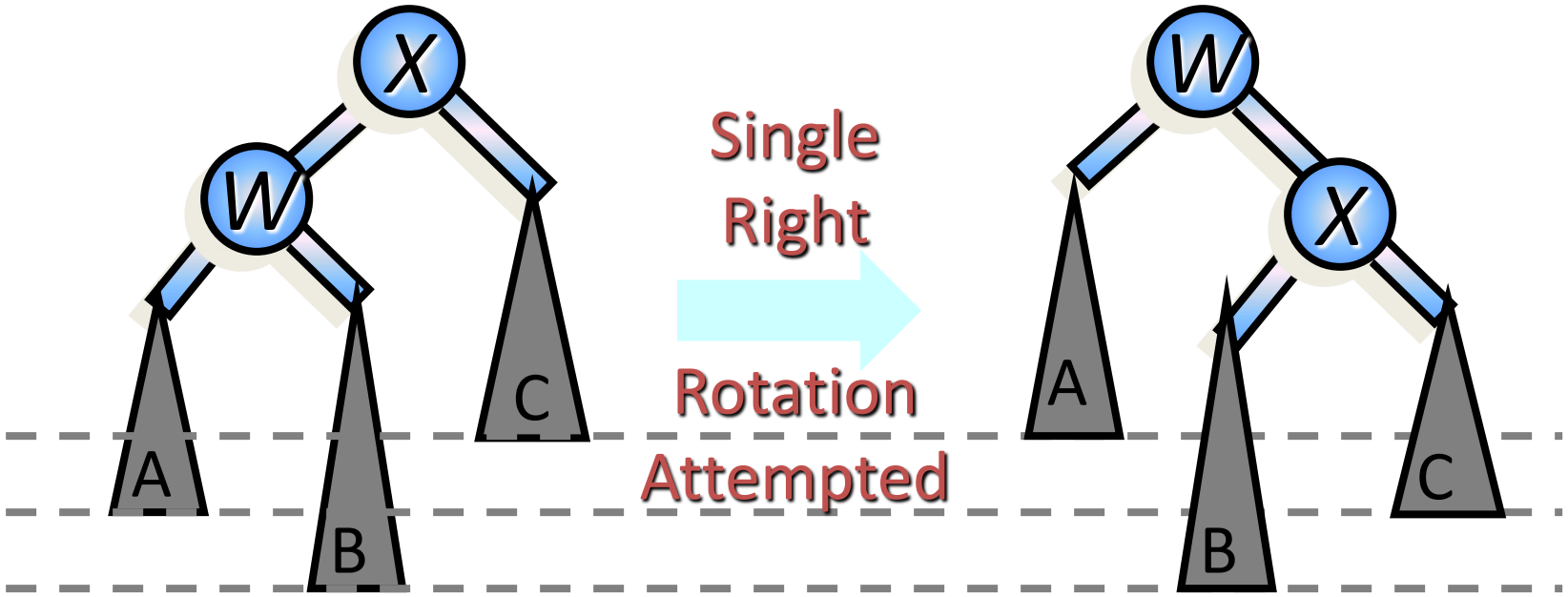
These insertions are not solved with a single rotation.





Instead, two single rotations, left-right and right-left are performed.

Why Doesn't Single Rotation Work?

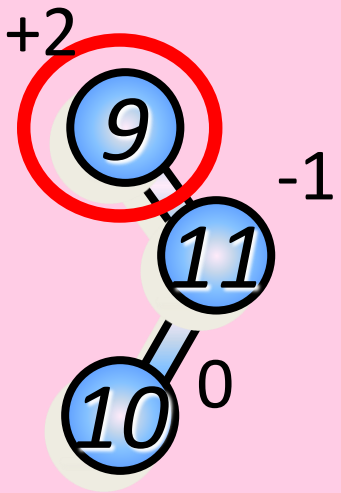


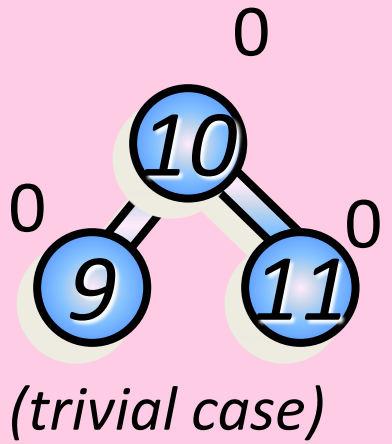
Unbalanced insertions into the "interior" of the tree remain unbalanced with only a single rotation

#3

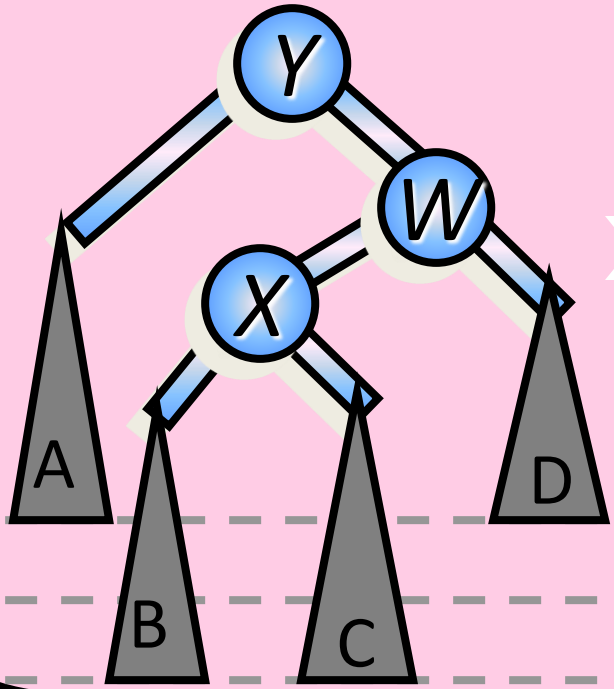
Double Rotation

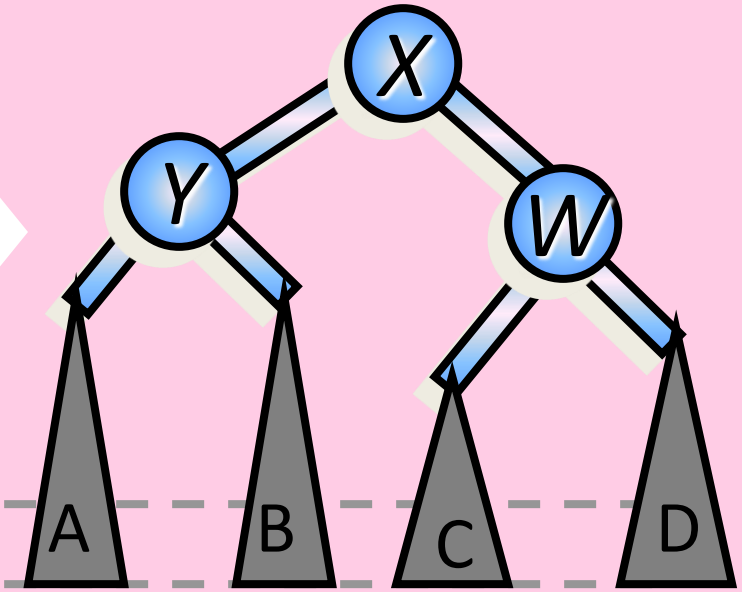
Insertion to left subtree of right child





(trivial case)

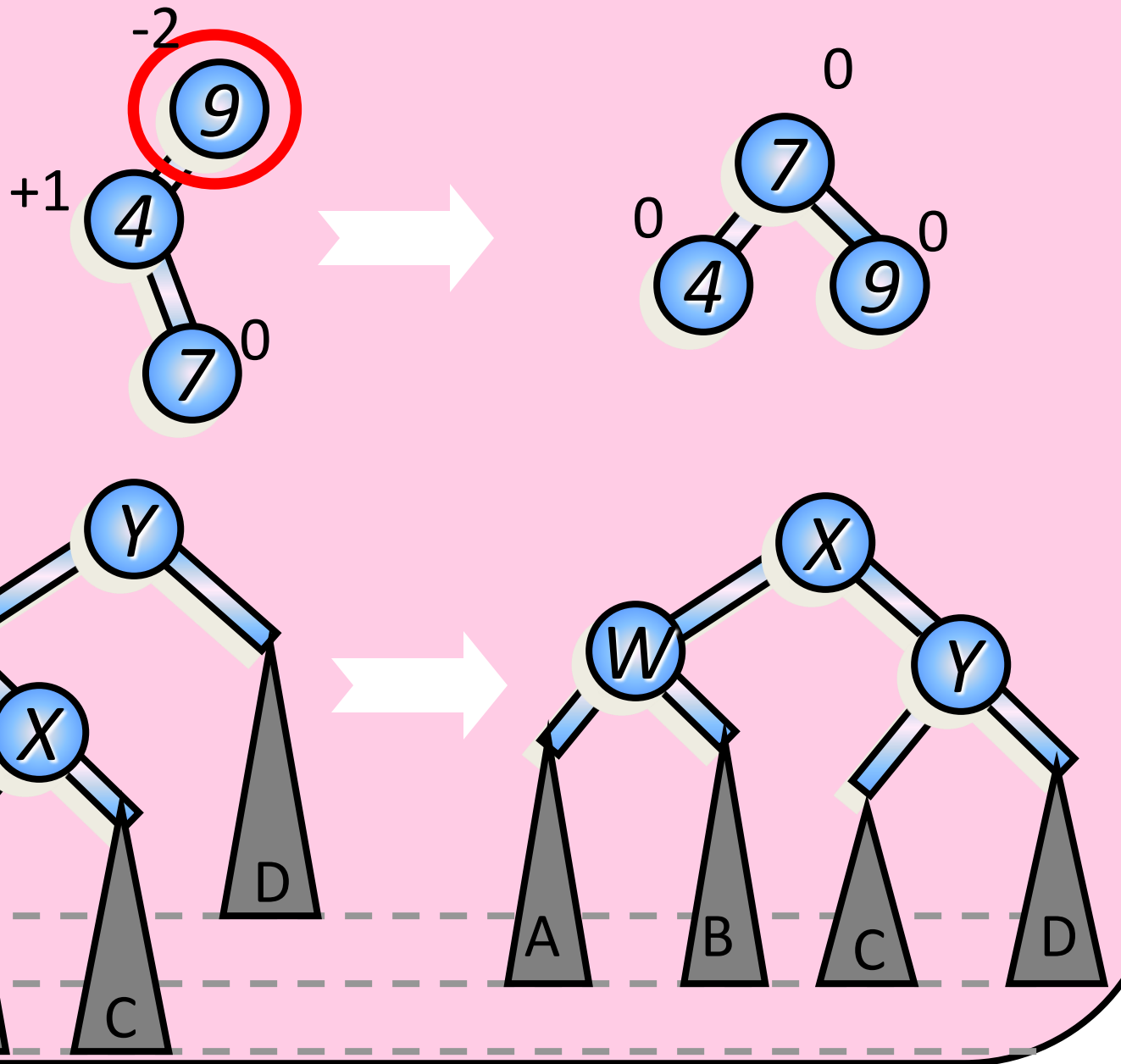




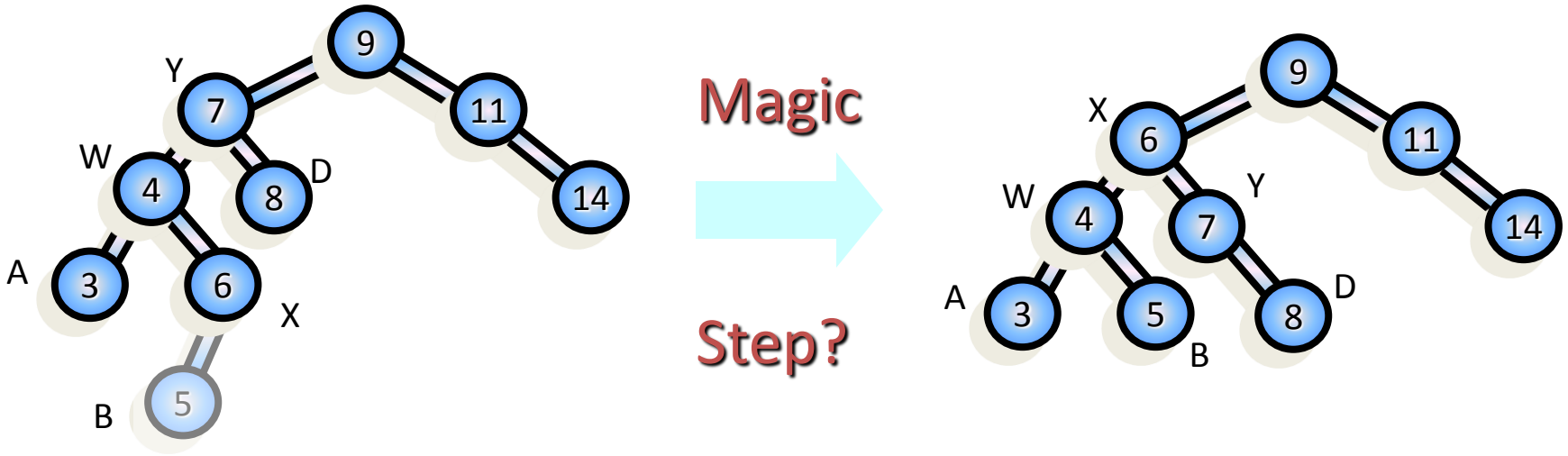
#4

Double Rotation

Insertion to right subtree of left child



Double Rotation



Solved:

Move up X!

W becomes X's left child

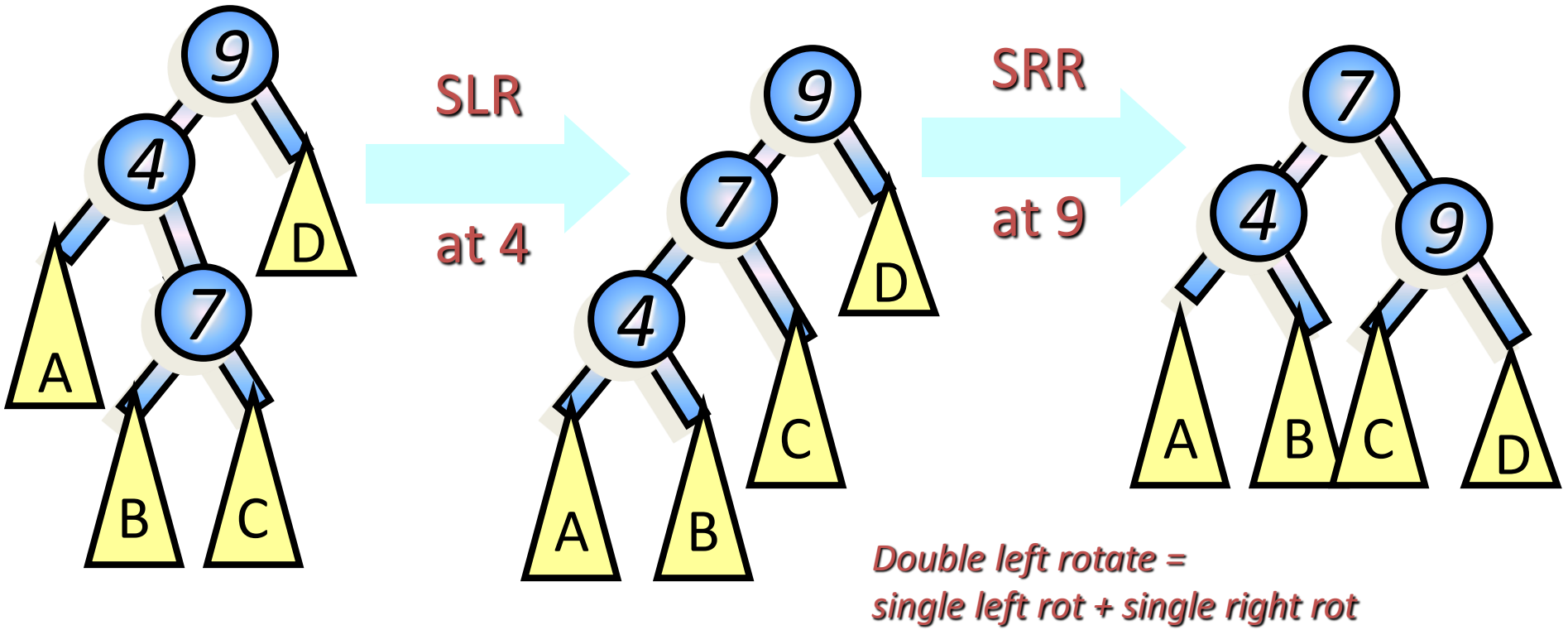
Y becomes X's right child

W's right child becomes X's left child

X's right child (if any) becomes Y's left child

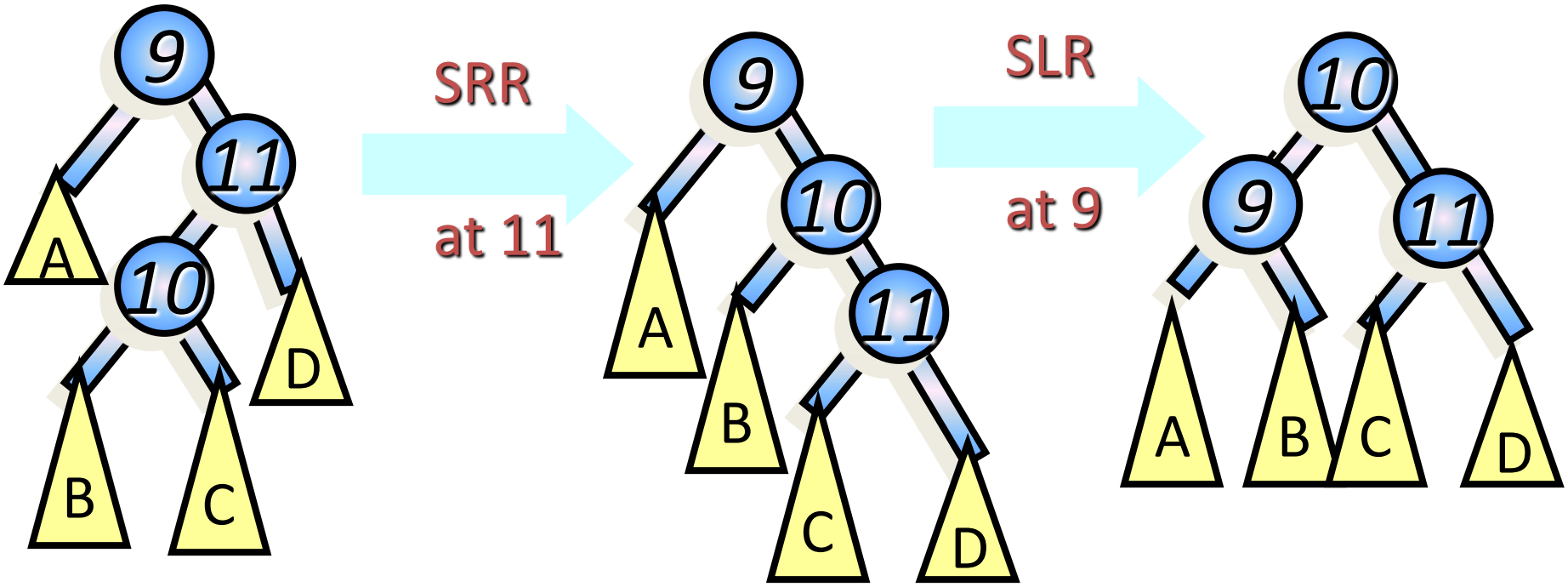
Double Rotation Strategy

On a left-right insert, we first rotate left



This places the tree in a position for which we already have a working solution. (Code reuse!)

Double Rotation Strategy



Similarly, we perform a right and left rotation on insertions that went left-right. The first rotation reforms the tree into an exterior imbalance. We already have a way of fixing this: single left rotations

AVL Tree Insert

```
Algorithm AVLInsert (root, newData)
Using recursion, insert a node into an AVL tree.
   Pre    root is pointer to first node in AVL tree/subtree
          newData is pointer to new node to be inserted
   Post   new node has been inserted
   Return root returned recursively up the tree
 1 if (subtree empty)
   Insert at root
   1   insert newData at root
   2   return root
 2 end if
 3 if (newData < root)
   1   AVLInsert (left subtree, newData)
   2   if (left subtree taller)
       1   leftBalance (root)
   3   end if
 4 else
   New data >= root data
   1   AVLInsert (right subtree, newPtr)
   2   if(right subtree taller)
       1   rightBalance (root)
   3   end if
 5 end if
 6 return root
end AVLInsert
```

AVL Tree Left Balance

```
Algorithm leftBalance (root)
This algorithm is entered when the root is left high (the
left subtree is higher than the right subtree).
   Pre    root is a pointer to the root of the [sub]tree
   Post   root has been updated (if necessary)
1  if (left subtree high)
   1   rotateRight (root)
2  else
   1   rotateLeft (left subtree)
   2   rotateRight (root)
3  end if
end leftBalance
```

Rotate AVL Tree Right and Left

```
Algorithm rotateRight (root)
This algorithm exchanges pointers to rotate the tree right.
   Pre    root points to tree to be rotated
   Post   node rotated and root updated
 1 exchange left subtree with right subtree of left subtree
 2 make left subtree new root
end rotateRight

Algorithm rotateLeft (root)
This algorithm exchanges pointers to rotate the tree left.
   Pre    root points to tree to be rotated
   Post   node rotated and root updated
 1 exchange right subtree with left subtree of right subtree
 2 make right subtree new root
end rotateLeft
```

AVL Tree Delete

```
Algorithm AVLDelete (root, dltKey, success)
This algorithm deletes a node from an AVL tree and
rebalances if necessary.
   Pre    root is a pointer to a [sub]tree
          dltKey is the key of node to be deleted
          success is reference to boolean variable
   Post   node deleted if found, tree unchanged if not
          success set true (key found and deleted)
             or false (key not found)
   Return pointer to root of [potential] new subtree
1  if Return (empty subtree)
       Not found
   1   set success to false
   2   return null
2  end if
3  if (dltKey < root)
   1   set left-subtree to AVLDelete(left subtree, dltKey,
                                     success)
```

continued

AVL Tree Delete *(continued)*

```
   2   if (tree shorter)
       1   set root to deleteRightBalance(root)
   3   end if
4  elseif (dltKey > root)
   1   set right subtree to AVLDelete(root->right, dltKey,
                                      success)
   2   if (tree shorter)
       1   set root to deleteLeftBalance (root)
   3   end if
5  else
       Delete node found--test for leaf node
   1   save root
```

AVL Tree Delete (continued)

```
   2   if (no right subtree)
       1   set success to true
       2   return left subtree
   3   elseif (no left subtree)
           Have right but no left subtree
       1   set success to true
       2   return right subtree
   4   else
           Deleted node has two subtrees
           Find substitute--largest node on left subtree
       1   find largest node on left subtree
       2   save largest key
       3   copy data in largest to root
       4   set left subtree to AVLDelete(left subtree,
                                         largest key, success)
       5   if (tree shorter)
           1  set root to dltRightBal (root)
       6   end if
   5   end if
 6  end if
 7  return root
end AVLDelete
```

AVL Tree Delete Right Balance

```
Algorithm deleteRightBalance (root)
The [sub]tree is shorter after a deletion on the left branch.
If necessary, balance the tree by rotating.
   Pre    tree is shorter
   Post   balance restored
   Return new root
 1 if (tree not balanced)
      No rotation required if tree left or even high
   1  set rightOfRight to right subtree
   2  if (rightOfRight left high)
         Double rotation required
      1  set leftOfRight to left subtree of rightOfRight
         Rotate right then left
      2  right subtree = rotateRight (rightOfRight)
      3  root          = rotateLeft (root)
   3  else
         Single rotation required
      1  set root to rotateLeft (root)
   4  end if
 2 end if
 3 return root
end deleteRightBalance
```

Binary Search Tree algorithms

Experimental Procedure:

- Analyse the problem statement
- Design an algorithm for the given problem statement and develop a flowchart/pseudo-code
- Implement the algorithm in C language
- Compile the C program
- Design test cases and test the implemented program
- Document the Results
- Analyse and discuss the outcomes of your experiment

Binary Search Tree algorithms

Exercises:

- Design, develop algorithms and write C program to traverse the given BST using DFS(Post order) and BFS(Level Order) using recursion and non-recursive way(Queues).
- Design the test cases to test the implemented C program and verify against expected values.
- Analyse the efficiency of both the algorithms.
- Describe your learning along with the limitations of both, if any. Suggest how these can be overcome.

Results and Presentations

- Calculations/Computations/Algorithms

  The calculations/computations/algorithms involved in each program has to be presented

- Presentation of Results

  The results for all the valid and invalid cases have to be presented

- Analysis and Discussions

  how the data is manipulated or transformed, what are the key operations involved. Errors encounters and how they are resolved.

- Conclusions

  Summary

Comments

- Limitations of Experiments

  Outline the loopholes in the program, data structures or solution approach.

- Limitations of Results

  Present the test cases; justify if the program is tested correctly considering all the outcomes. Mention what is not tested, if any.

- Learning happened

  What is the overall learning happened

- Conclusions

  Summary

References

- Gilberg, R. F., and Forouzan, B. A. (2007): A Pseudocode Approach With C, 2nd edn. Cengage Learning
- Graphical part of the avl is taken from