CS2351
Data Structures

Lecture 14:
AVL Tree

About this lecture

- A general binary search tree (BST) does not have good worst-case performance since its height can be $\Theta(n)$
- In this lecture, we discuss a balanced BST called AVL tree, whose height = $O(\log n)$

  ➔ Query is done in $O(\log n)$ time

  - More involved updates due to balancing
  - invented by Adelson-Velskii and Landis

AVL Tree

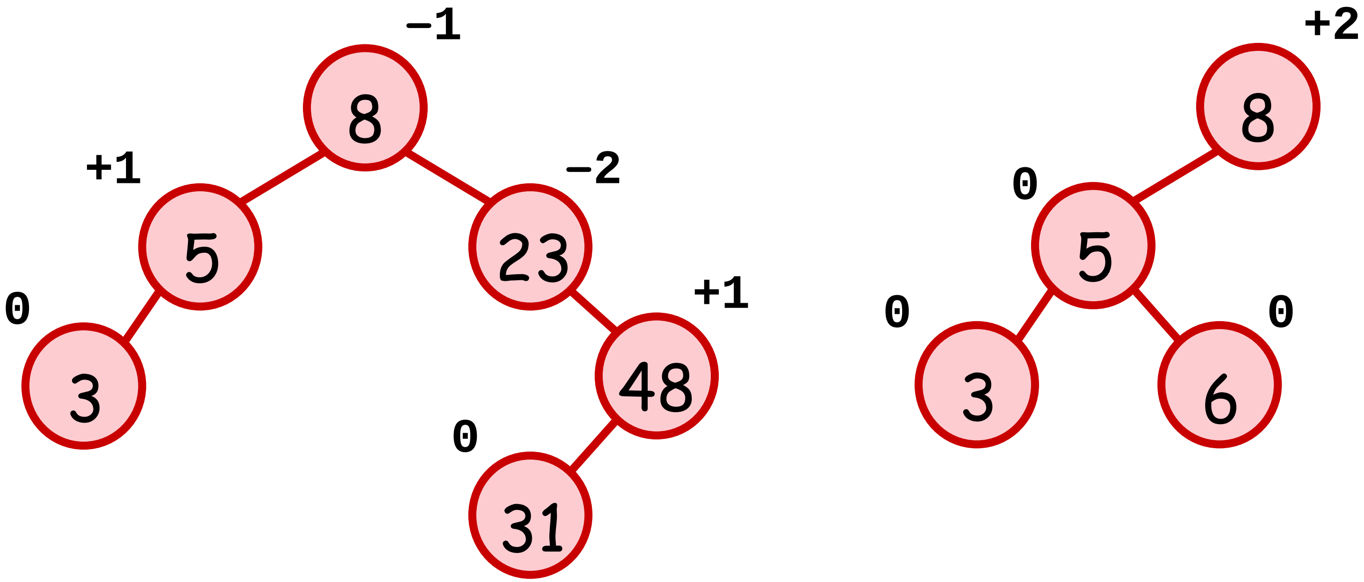
- Let x be a node.
- Let L and R be its left and right subtrees.
- We define balance factor of x to be :

  bf(x) = Height of L - Height of R

- An AVL tree is a BST with the property :

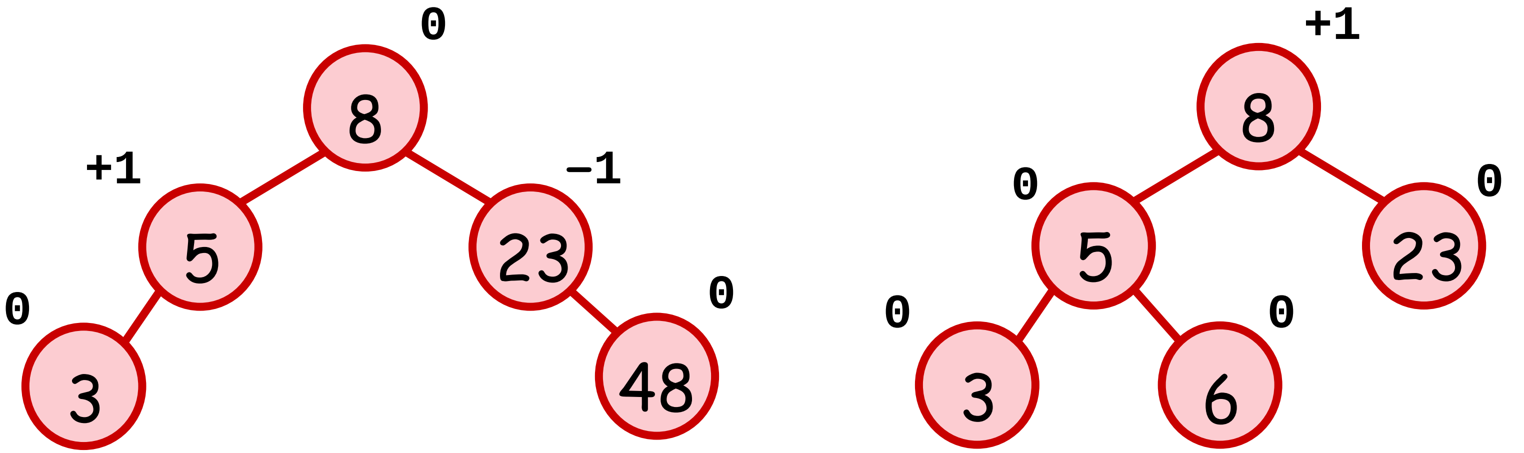
Each node has a balance factor of either 1, 0, or -1

Example of AVL Tree



Are these AVL trees?

Example of AVL Tree



Are these AVL trees?

Height of an AVL Tree

- Let h be the node-height of an AVL tree.
- Then we have :

Theorem : $h \leq 1.4405 \log n + O(1)$

- The idea of the proof is that :

  If an AVL tree has node-height h, then it must have a lot of nodes so that it cannot be too "skewed"

Proof

- Let N_h be the number of nodes in the smallest AVL tree with height h

➔ $N_1 = 1, \; N_2 = 2$

➔ $N_h = N_{h-1} + N_{h-2} + 1$ (why?)

- Indeed, we can show that (how?)

$$N_h = F_{h+1} - 1$$

where $F_k = k^{th}$ Fibonacci number ($F_0 = F_1 = 1$)

Proof (cont)

- It is known that for Fibonacci number F_k :

$$F_k \approx \Theta(\varphi^k)$$

where $\varphi = (1+\sqrt{5})/2 = 1.61803\ldots$

- Thus, if n is the number of nodes in an AVL tree with node-height h

$$n \geq N_h \geq c \times \varphi^{h+1} \qquad \text{[c is a constant]}$$

➔ $h \leq \log_\varphi n + O(1) \leq 1.4405 \log n + O(1)$

Query Performance

Corollary :

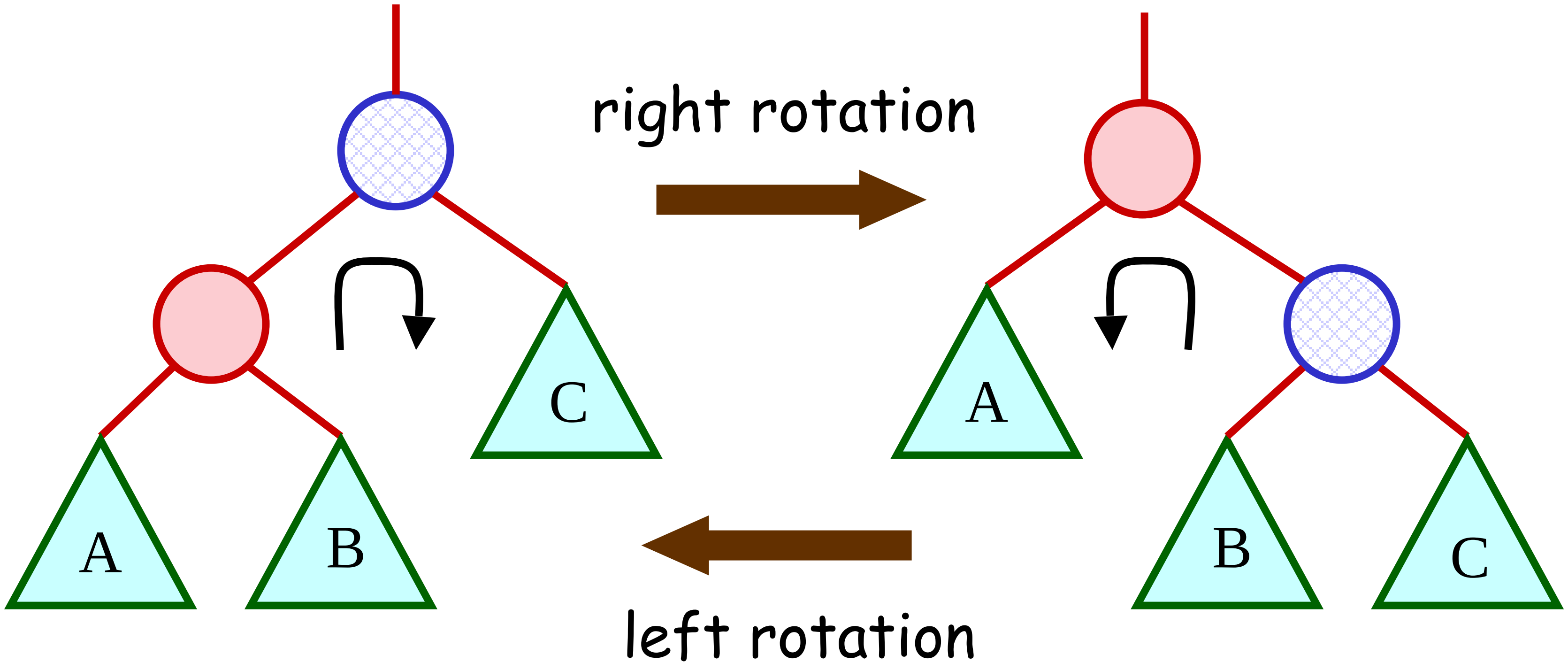
The queries minimum, maximum, search, predecessor, and successor can each be performed in O(log n) time in an AVL tree

Updates in an AVL Tree

Updates in an AVL Tree

- Updates are performed in the same way as in a general BST, except that we need balancing if the tree shape is too “skewed”
- The balancing is based on a powerful operation called “rotation”
  - also used in other balanced BST, such as Red-Black tree or Splay tree

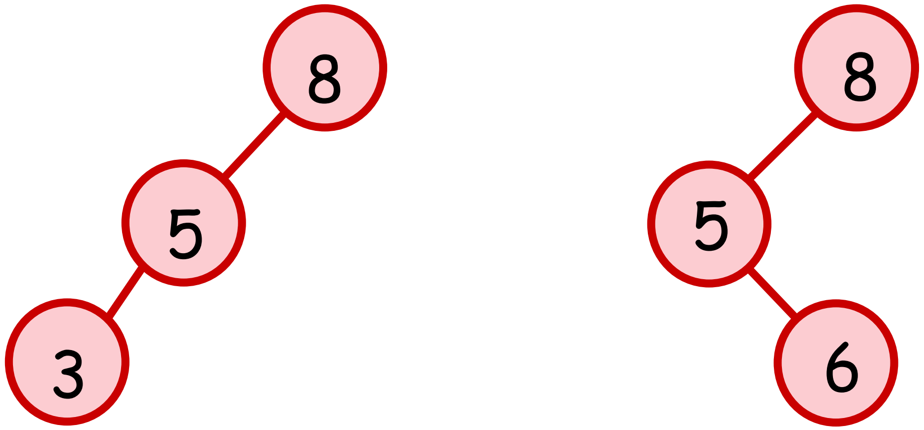
Rotation



Observation : *After* rotation, the inorder traversal ordering remains unchanged.

Remark

- If one subtree is too tall, we may use some rotations to balance the tree
- Ex : How to balance the following cases ?



In fact, we can always transform one BST to another just by rotations (how to show?)

Implementation in C

- We can define R_rotate as follows using Transplant from the previous lecture :

```
// Assume *x has left child
Node * R_rotate( Node *x ) {
  y = x->left ;
  Transplant( y, y->right );
  Transplant( x, y );
  y->right = x ;
  x->parent = y ;
}
```

Implementation in C

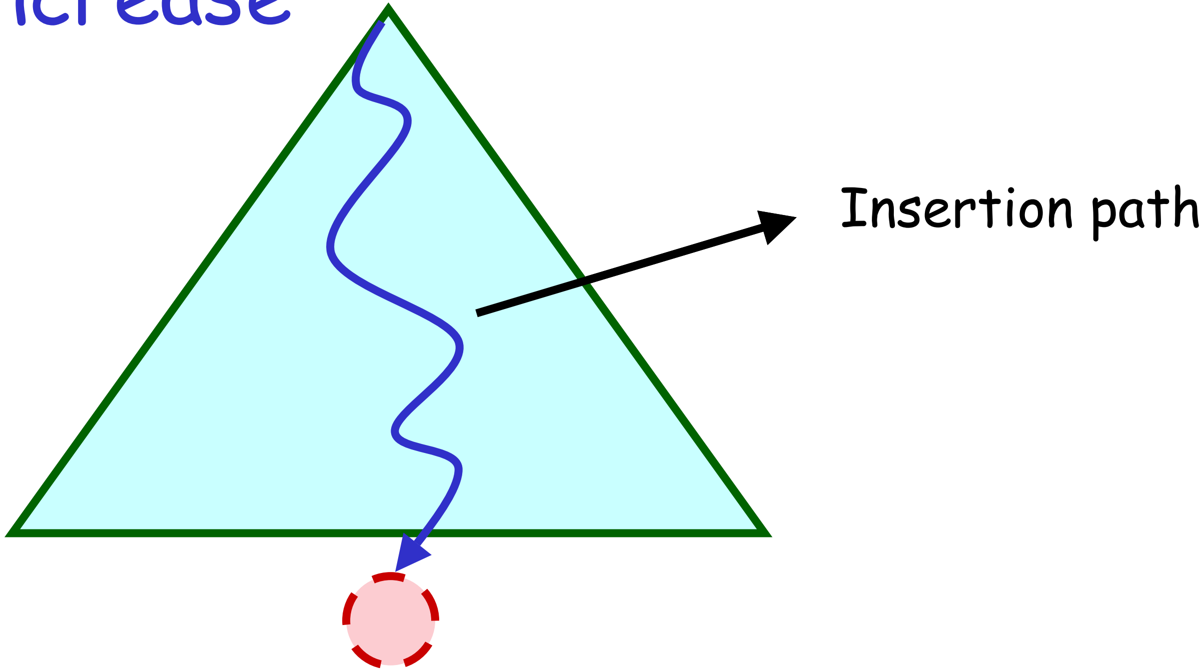
- Similarly, we can define L_rotate, and then LR_rotate or RL_rotate :

```
// Assume *x has left child
// and *(x->left) has right child
Node * LR_rotate( Node *x ) {
  L_rotate( x->left ) ;
  R_rotate( x );
}
```

Insertion in an AVL Tree

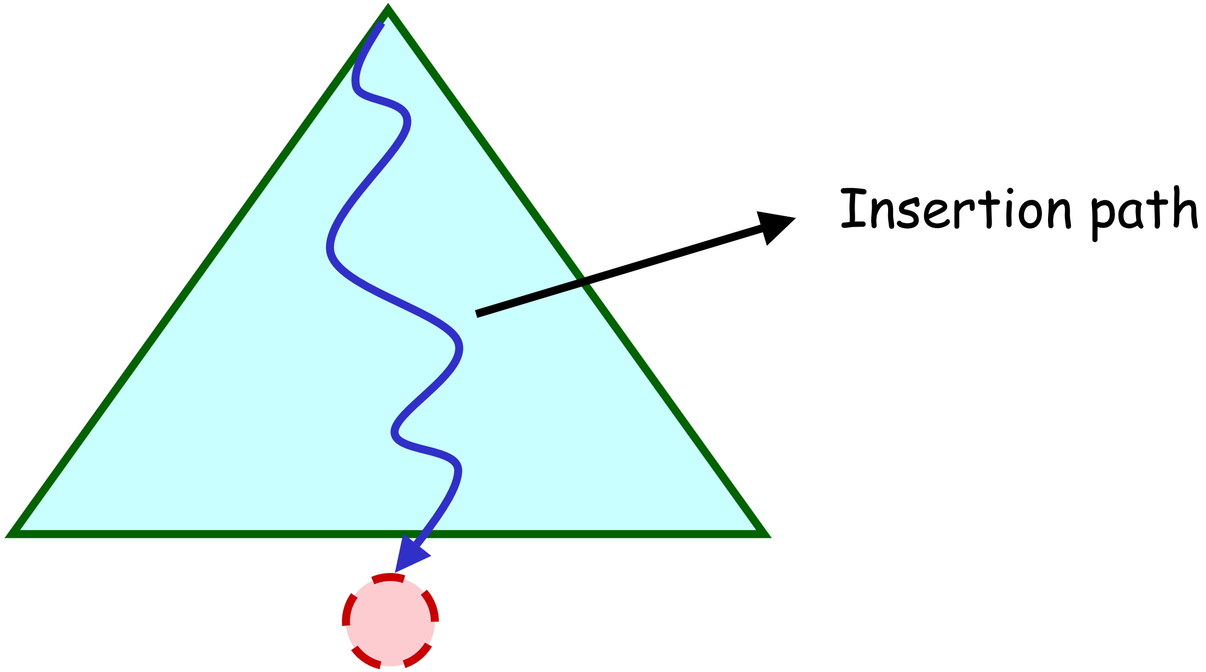
Insertion

- Insertion is the same way as before, except that after insertion, the balance factor of some nodes (along the insertion path) may increase



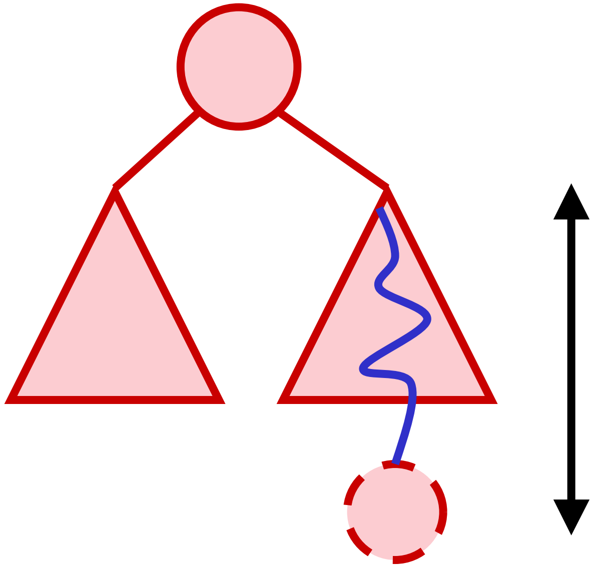
Insertion

- Consequently, we need to balance these nodes so that AVL property is maintained
- This is done by a bottom-up fashion



Case 1 (No Height Change)

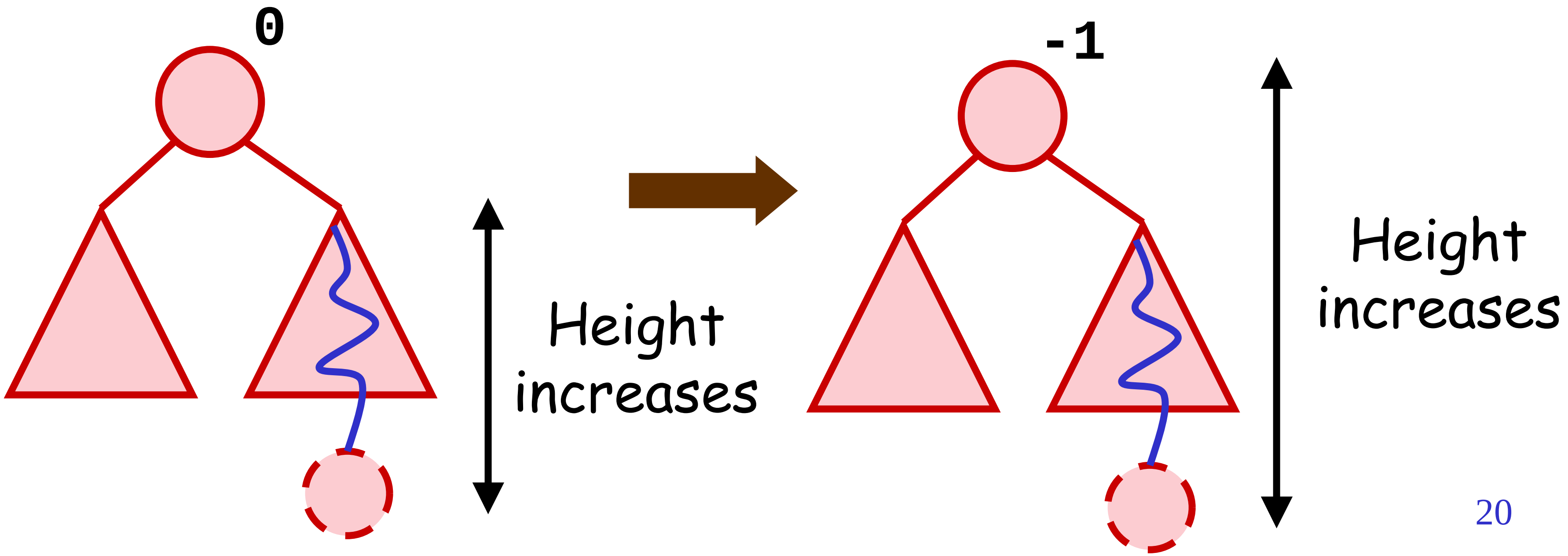
- If no height change in the subtree
  ➔ balance factor of a node (and its ancestors) is not changed ➔ done !



Height not changed

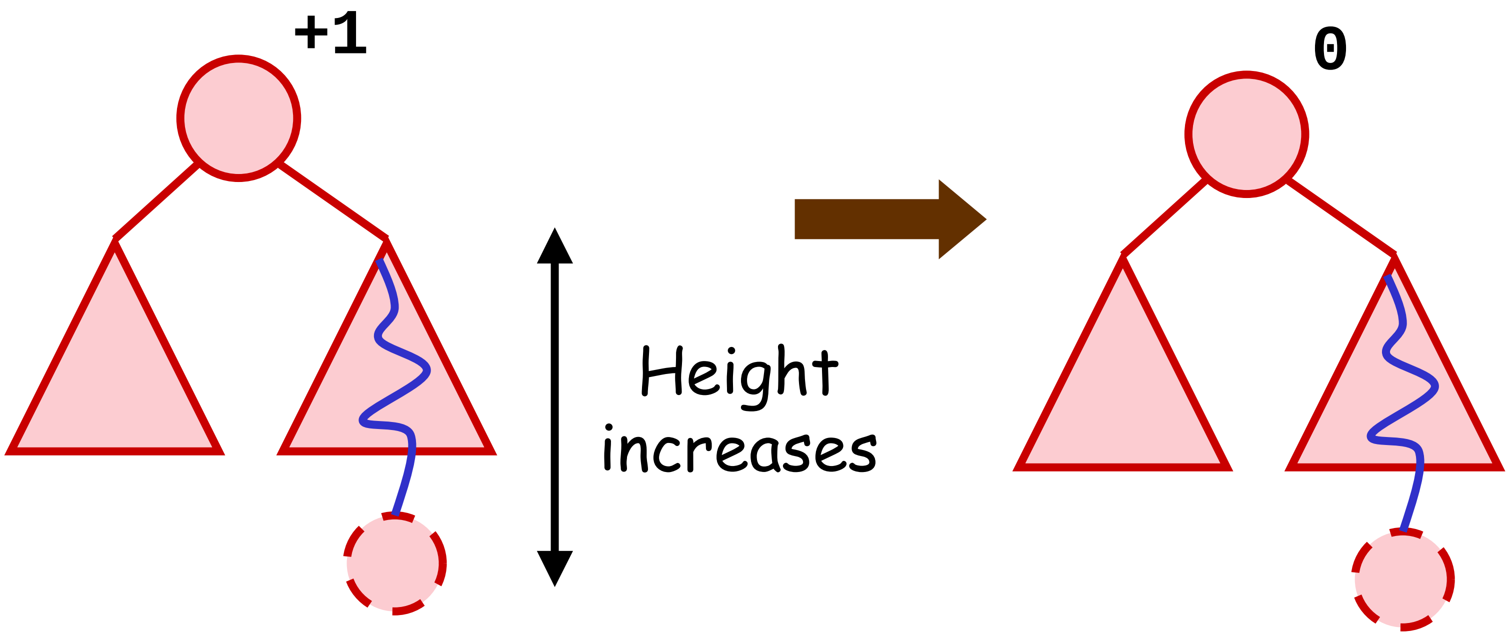
Case 2.1 (Height Increases)

- If height of the subtree increases (by 1)
  - ➔ If balance factor was 0 originally
  - ➔ Update balance factor and continue to balance the ancestors



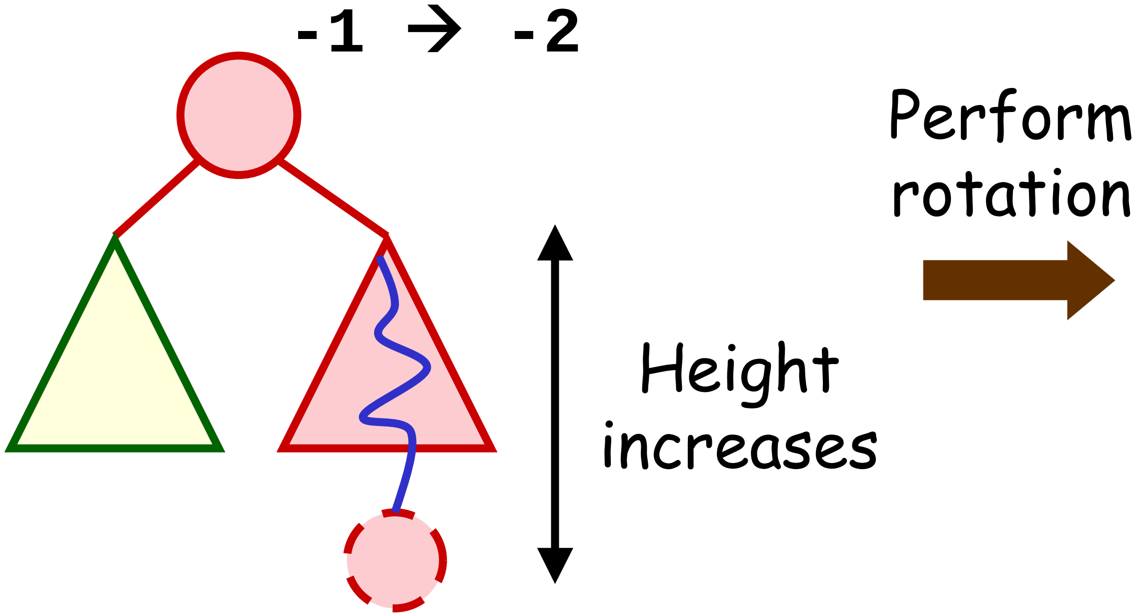
Case 2.2 (Height Increases)

- If height of the subtree increases (by 1)
  - ➔ If other subtree was taller originally
  - ➔ Set balance factor to 0, and done !



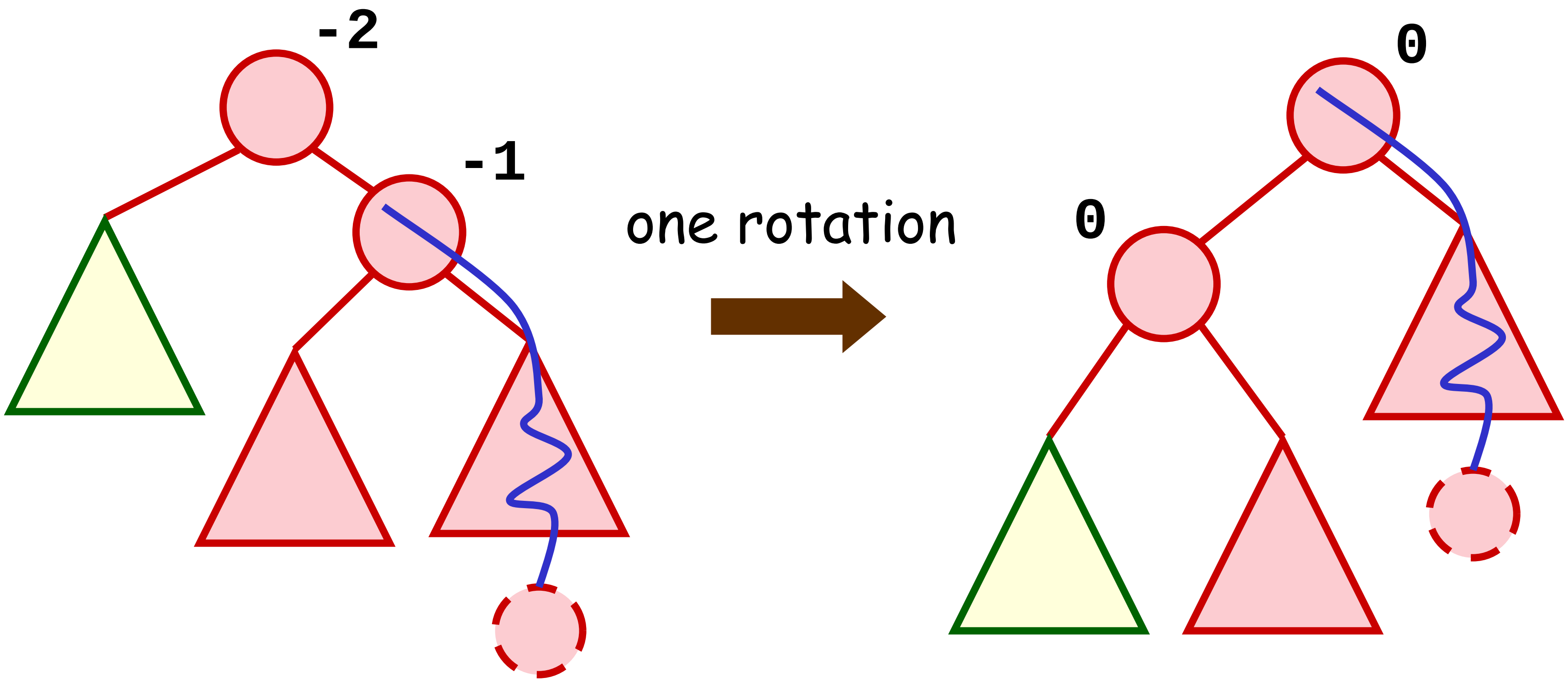
Case 2.3 (Height Increases)

- If height of the subtree increases (by 1)
  - ➔ If other subtree was shorter originally
  - ➔ Perform rotation



Case 2.3 (Height Increases)

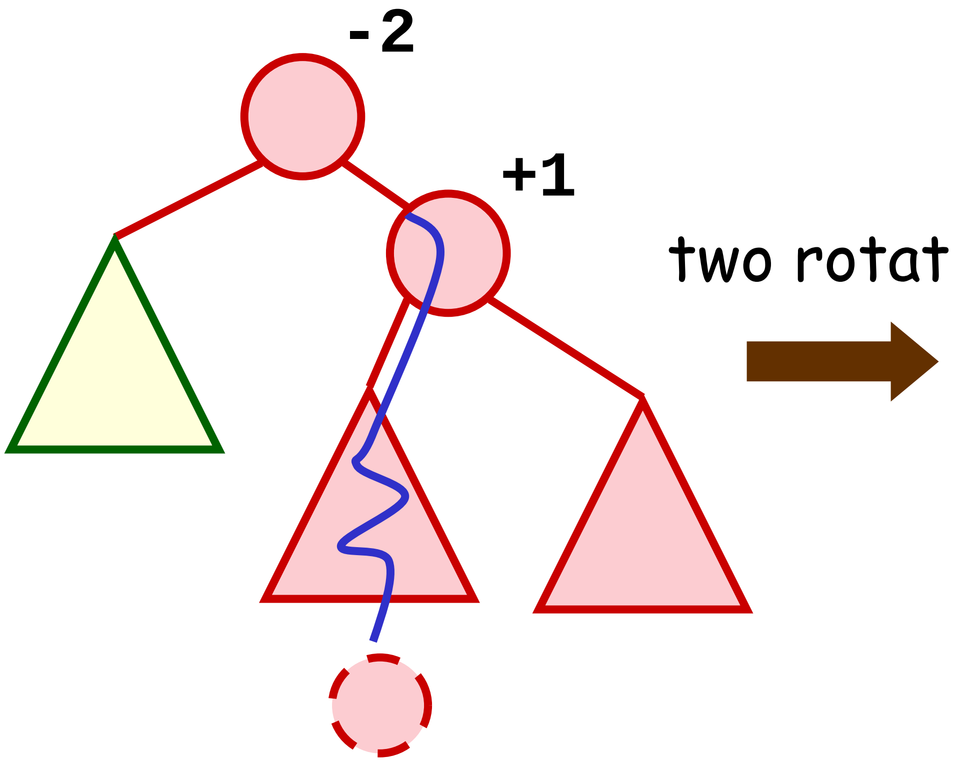
- There are two subcases for Case 2.3 :



First subcase

Case 2.3 (Height Increases)

- There are two subcases for Case 2.3 :

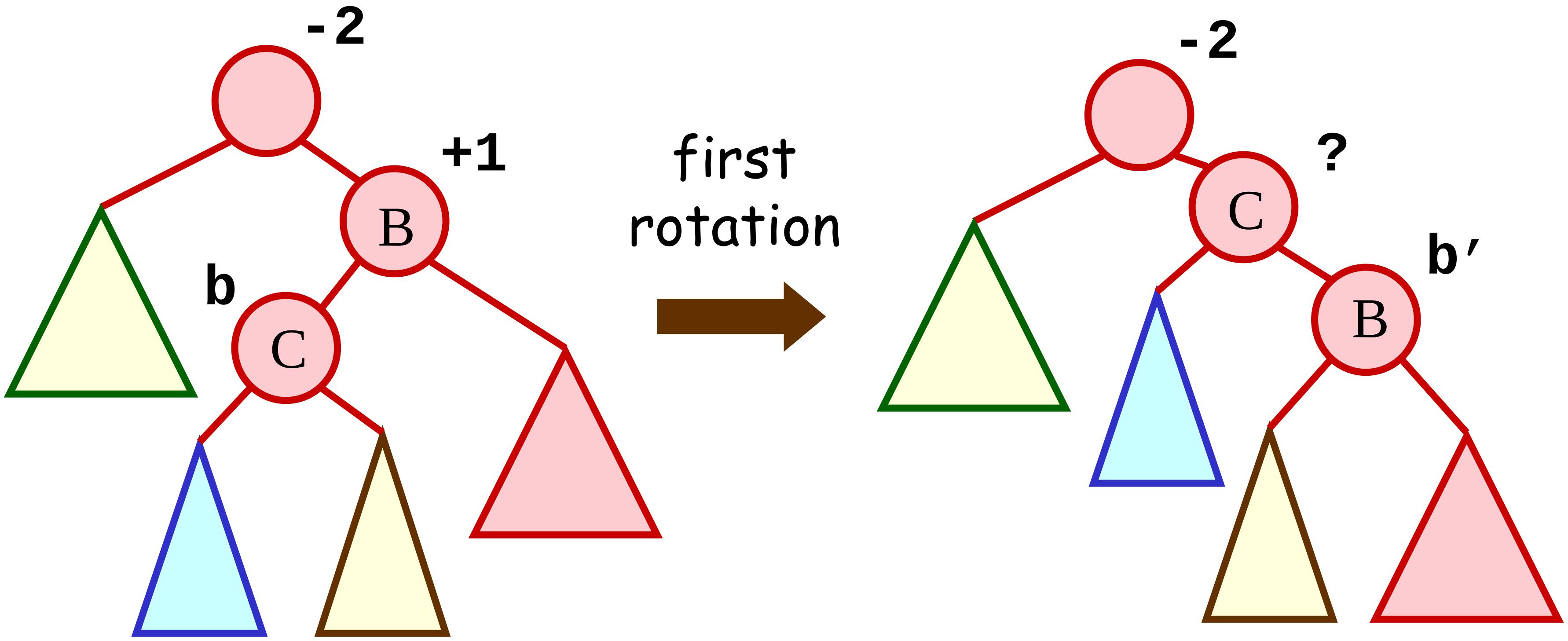


What is the resulting tree ?

Second subcase

Case 2.3 (Height Increases)

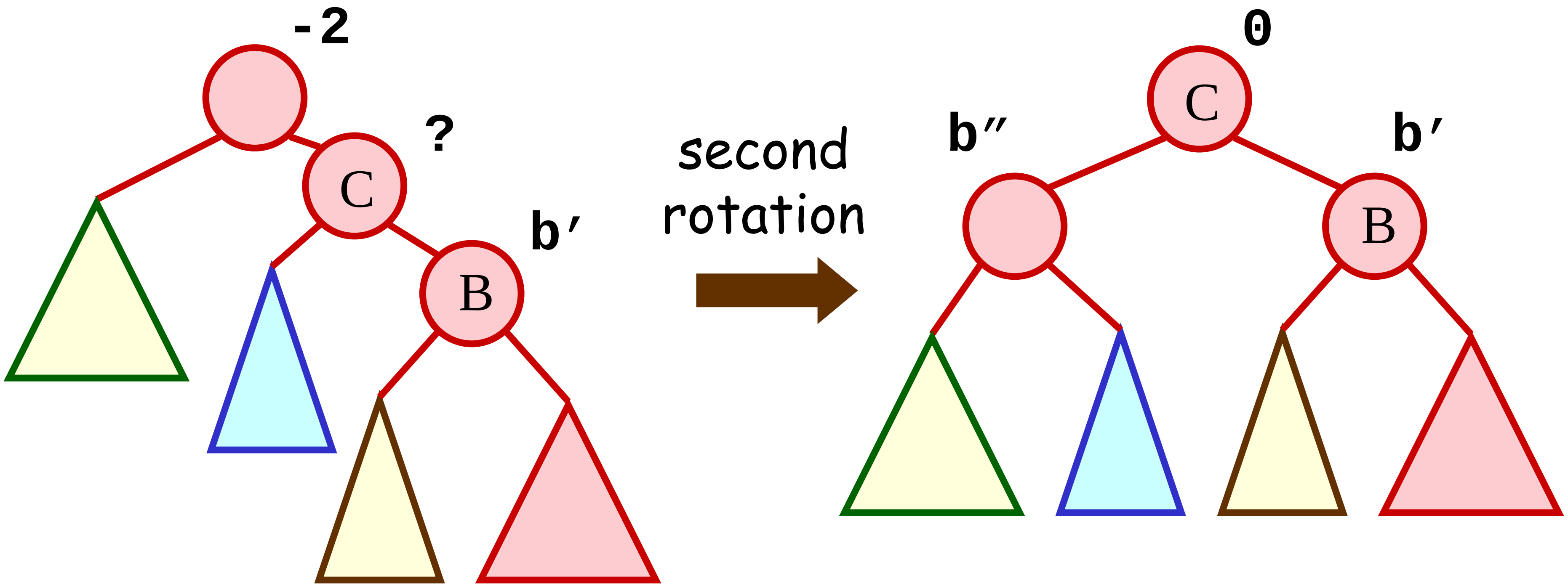
- First Rotation :



Second subcase

Case 2.3 (Height Increases)

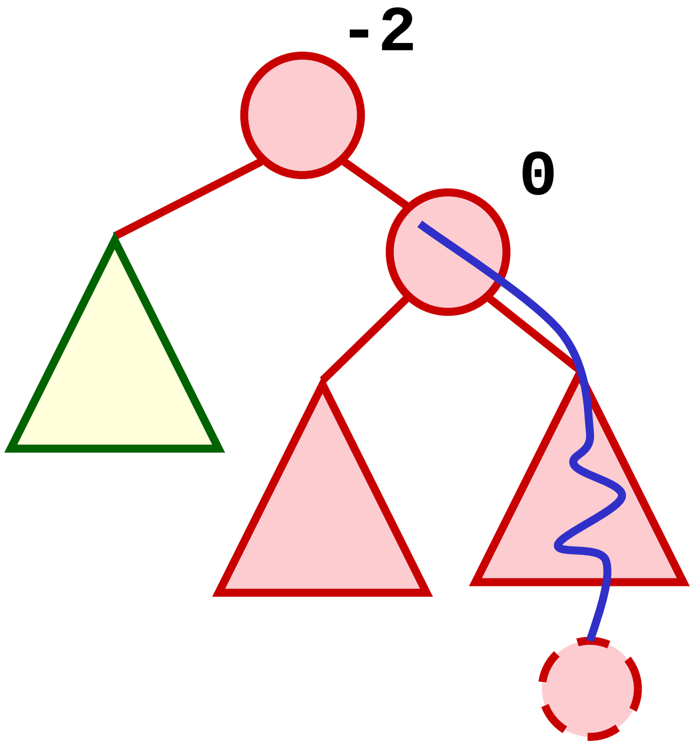
- Second Rotation :



Second subcase

Case 2.3 (Height Increases)

- What if the child node on the insertion path has balance factor 0 ?



Third subcase ?

We can prove that using our insertion scheme, if child node has balance factor 0, height cannot increase

Case 2.3 (Height Increases)

- After perform rotations in either subcases, the node becomes balanced

  → No change is needed for the ancestor

  → Done !

Implementation in C

- Recall the Insert function in the BST :

```
void Insert( Node *x, Node *z ) {
  if ( x->key < z->key ) {
    if ( x->right )  Insert( x->right, z );
    else x->right = z ;
  }
  else ...
}
```

- We now modify it to handle balancing …

Implementation in C

```
void Insert( Node *x, Node *z ) {
  if ( x->key < z->key ) {
    if ( x->right ) Insert( x->right, z );
    else {  x->right = z ;  z->bf = 0 ;
            height_inc = TRUE ; }
    if ( height_inc )
    {  /* Handle Cases 2.1, 2.2 & 2.3 */ }
  }
  else ...
}
```

height_inc is a global variable to indicate if height of subtree of x has increased during insertion

Handling Cases 2.1 and 2.2

```
/* Case 2.1 */
if ( x->bf == 0 )
   x->bf = -1 ;

/* Case 2.2 */
else if ( x->bf == 1 )
{   x->bf = 0 ;   height_inc = FALSE ; }

/* Case 2.3 */
else ...
```

Handling Case 2.3

```
/* Case 2.3 */
else {
  /* First Subcase */
  if ( x->right->bf == -1 ) {
     L_rotate( x );
     x->bf = x->parent->bf = 0 ;
     height_inc = FALSE ;
  }
  /* Second Subcase */ else ...
} ...
```

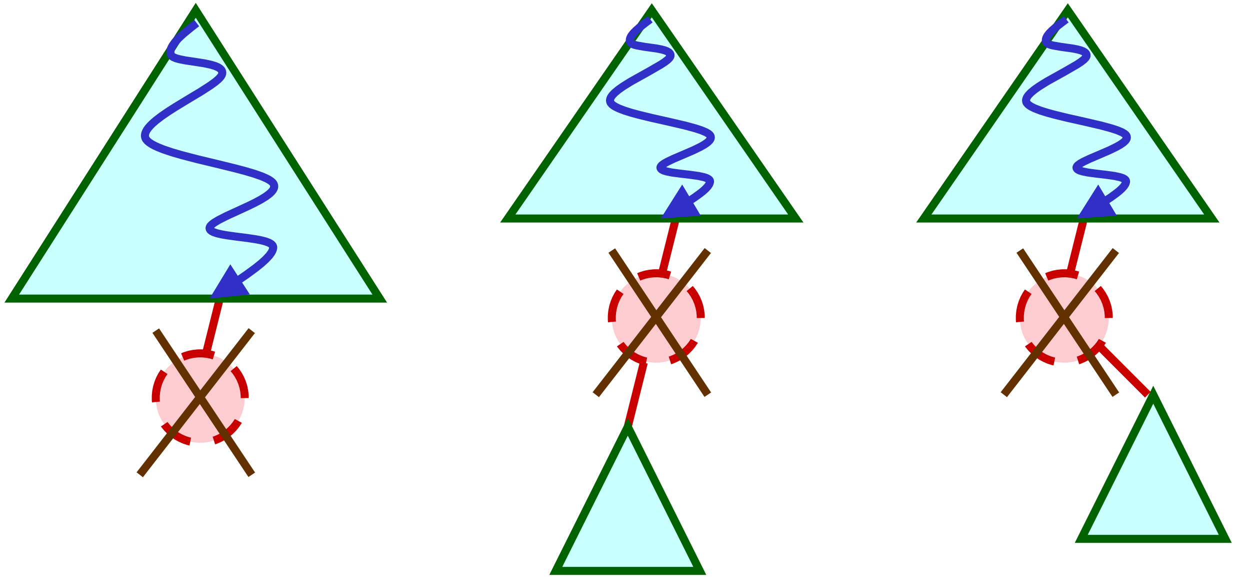
Handling Case 2.3

```
/* Second Subcase */
else if ( x->right->bf == 1 ) {
  int b = x->right->left->bf ;
  LR_rotate( x );  x->parent->bf = 0;
  if ( b == 0 )
  { x->bf = x->parent->right->bf = 0; }
  else if ( b == 1 )
  { x->bf = 0; x->parent->right->bf = -1; }
  else if ( b == -1 )
  { x->bf = 1; x->parent->right->bf = 0; }
  height_inc = FALSE ;
}
```

Deletion in an AVL Tree

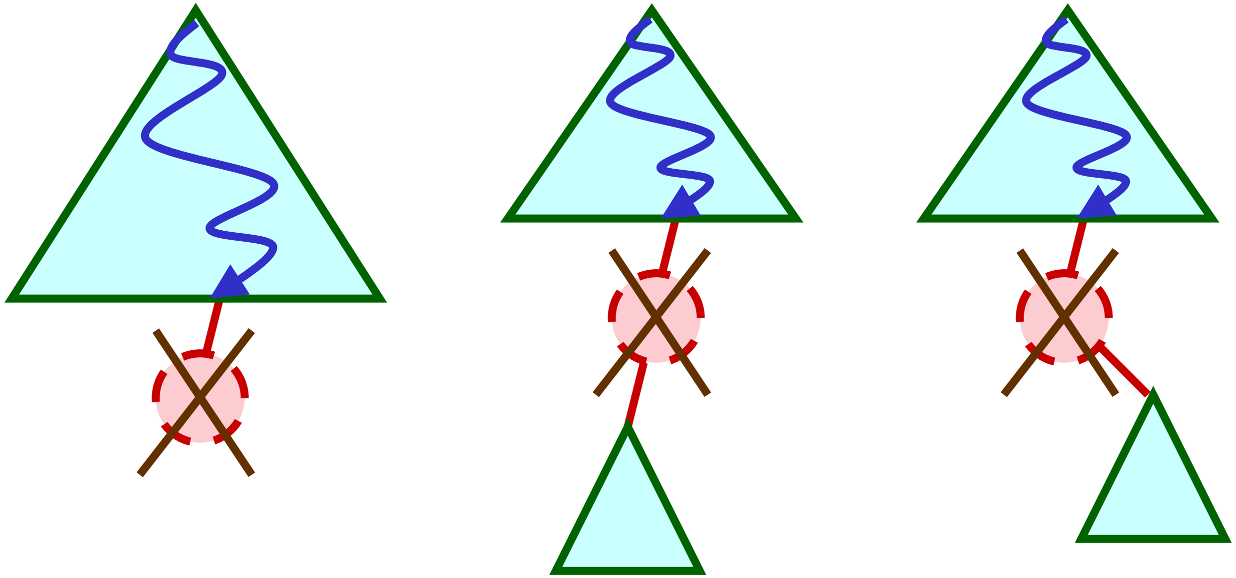
Deletion

- Deletion is the same way as before, except that after deletion, balance factor of the ancestors of the node "actually" deleted (ex: successor) may decrease



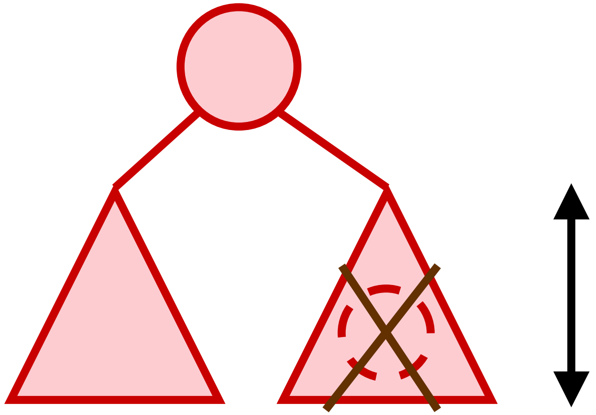
Deletion

- Consequently, we need to balance these nodes so that AVL property is maintained
- Again, this is done by a bottom-up fashion



Case 1 (No Height Change)

- If no height change in the subtree
  ➔ balance factor of a node (and its ancestors) is not changed ➔ done !



Height not changed

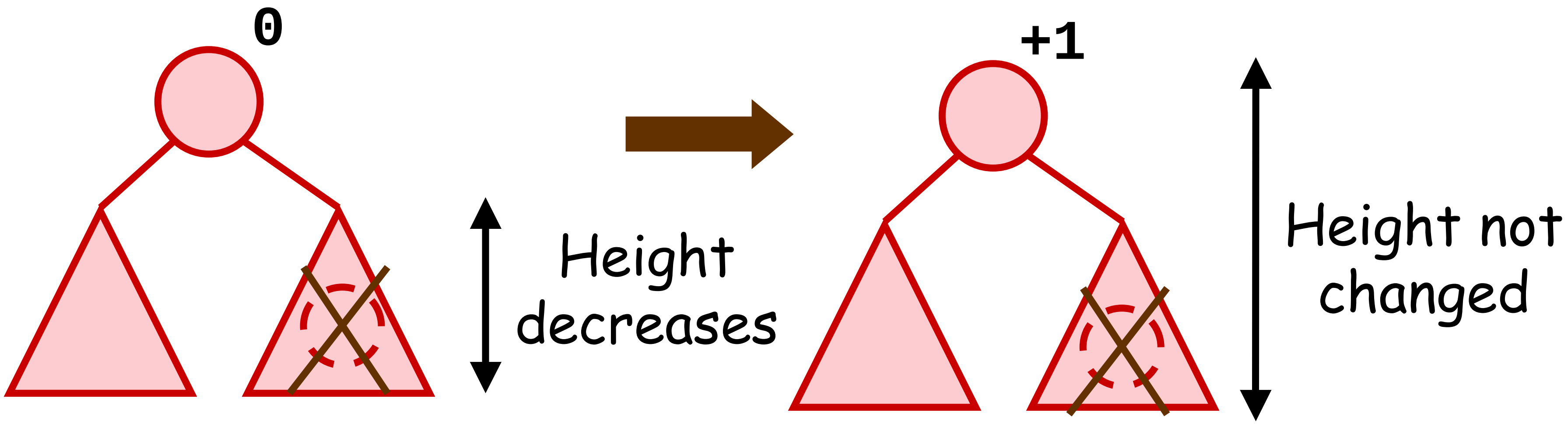
Case 2.1 (Height Decreases)

- If height of the subtree decreases (by 1)

  ➔ If balance factor was 0 originally

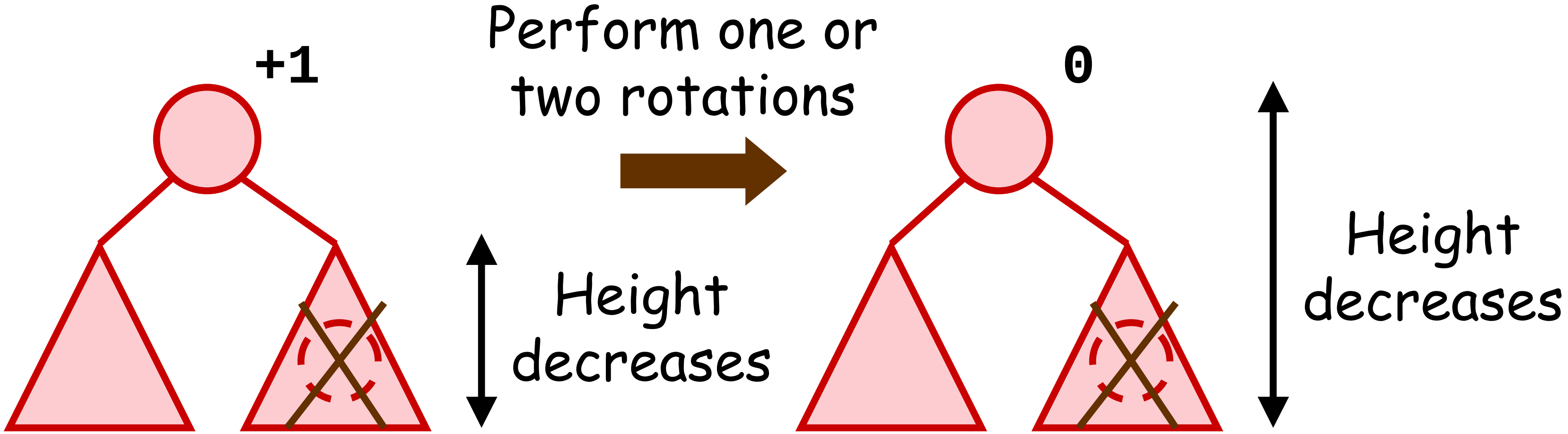
  ➔ Update balance factor and done !

  (handle differently from insertion)



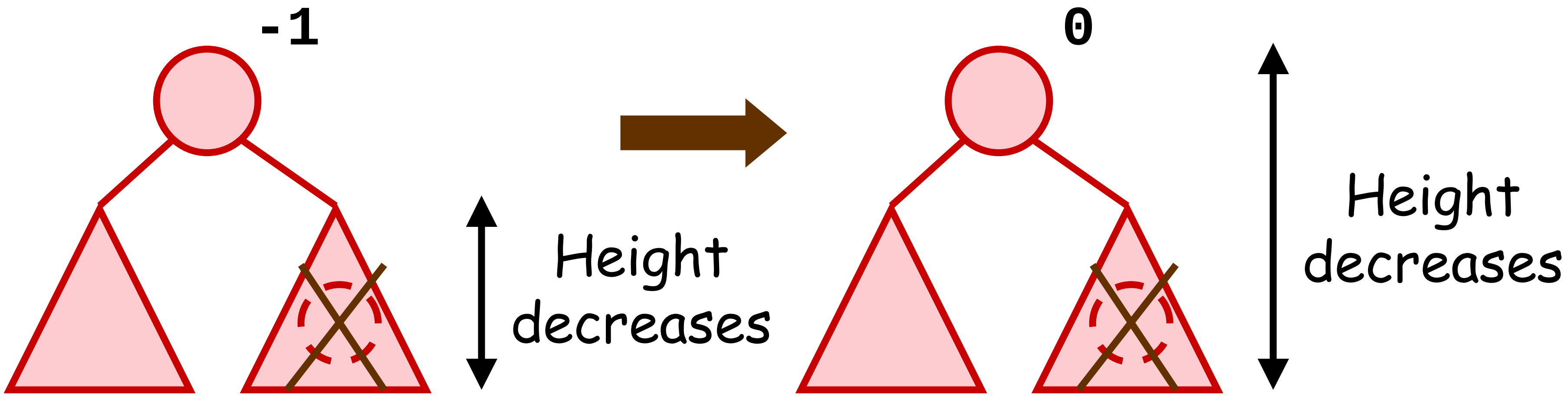
Case 2.2 (Height Decreases)

- If height of the subtree decreases (by 1)
  - ➔ If other subtree was taller originally
  - ➔ Rotations, set balance factor to 0
  - ➔ Continue to balance ancestors (why?)



Case 2.3 (Height Decreases)

- If height of the subtree decreases (by 1)
  - ➔ If other subtree was shorter originally
  - ➔ Set balance factor to 0
  - ➔ Continue to balance ancestors



Update Performance

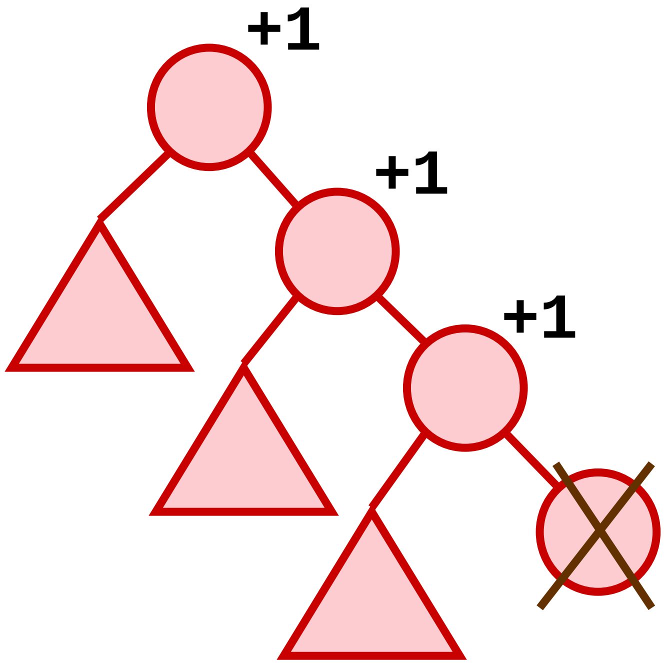
Corollary :

Insertion or deletion can each be performed in O(log n) time in an AVL tree

Remarks : Each insertion requires at most 2 rotations, but may update O(log n) nodes

Q: How about deletion ?

of Rotations in Deletion



In the worst case, we need to rotate in each level

➔ O(log n) rotations !