CPTS 223 Advanced Data Structure C/C++

Tree: AVL Tree

Overview

- Tree data structure
- Binary search trees
  - Support O(lg(n)) operations
  - Balanced trees
- STL set and map classes
- B-trees for accessing secondary storage
- Applications of Tree

Balanced BSTs

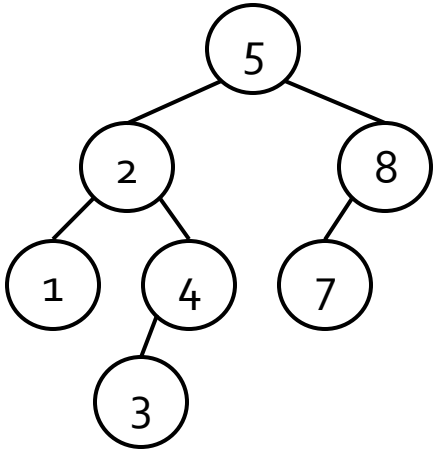
- AVL (Adelson-Velskii and Landis) trees
  - Height of left and right subtrees at every node in BST differ by at most 1
  - Balance forcefully maintained for every update (via rotations)
  - BST depth always O(lg(n))

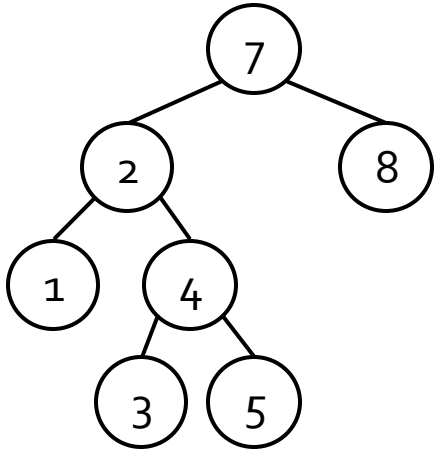
AVL trees

- Definition: every AVL tree is a BST such that
  - For every node in the BST, the heights of its left and right subtrees differ by at most 1
- Worst-case Height of AVL tree is Θ(lg(n))
  - Precisely, 1.44 log2(n+2) – 1.328
- Intuitively, it enforces that a tree is "sufficiently" populated before height is grown
  - Minimum #nodes S(h) in an AVL tree of height h :
  - S(h) = S(h-1) + S(h-2) + 1 (Similar to Fibonacci recurrence)
  - = Θ(2^h)

AVL trees

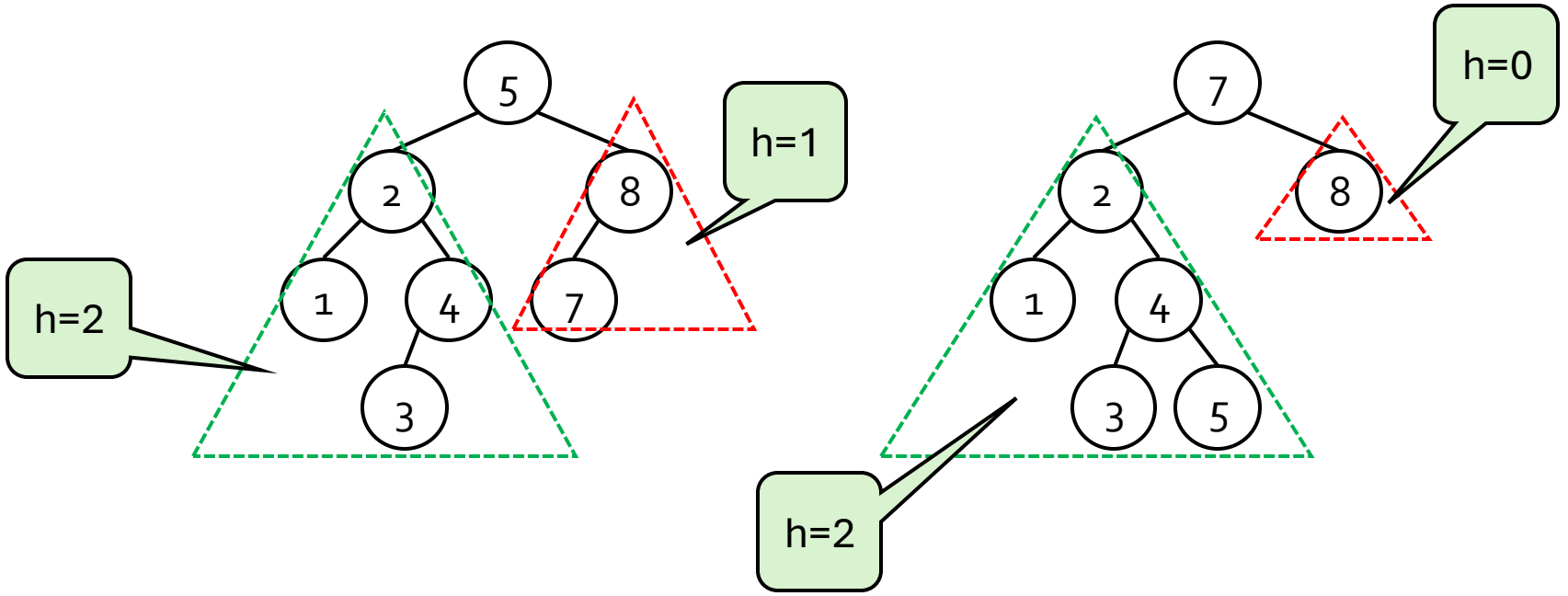
- Which of these is a valid AVL tree?





AVL trees

- Which of these is a valid AVL tree?

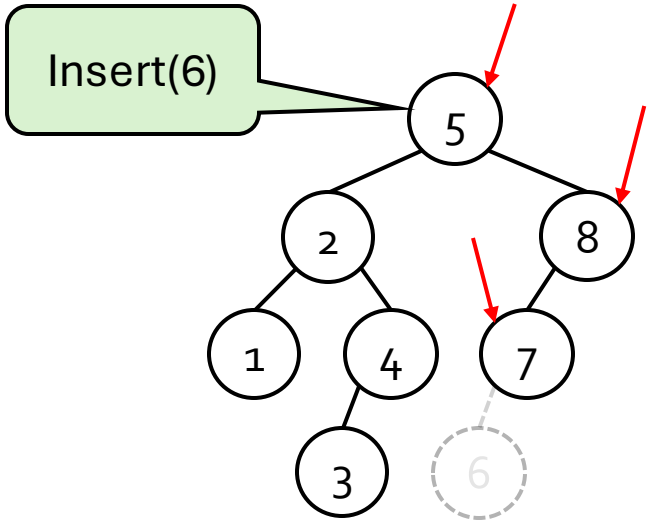


Maintaining balance condition

- If we can maintain balance condition
  - then insert, remove, find are O(lg(n))
- Why?
  - h = O(lg(n))
- Maintain height h(t) at each node t
  - h(t) := max{h(t->left), h(t->right)} + 1
  - h(empty tree) = -1
- Which operations can violate balance condition?
  - Find(X), insert(X), delete(X), etc.?

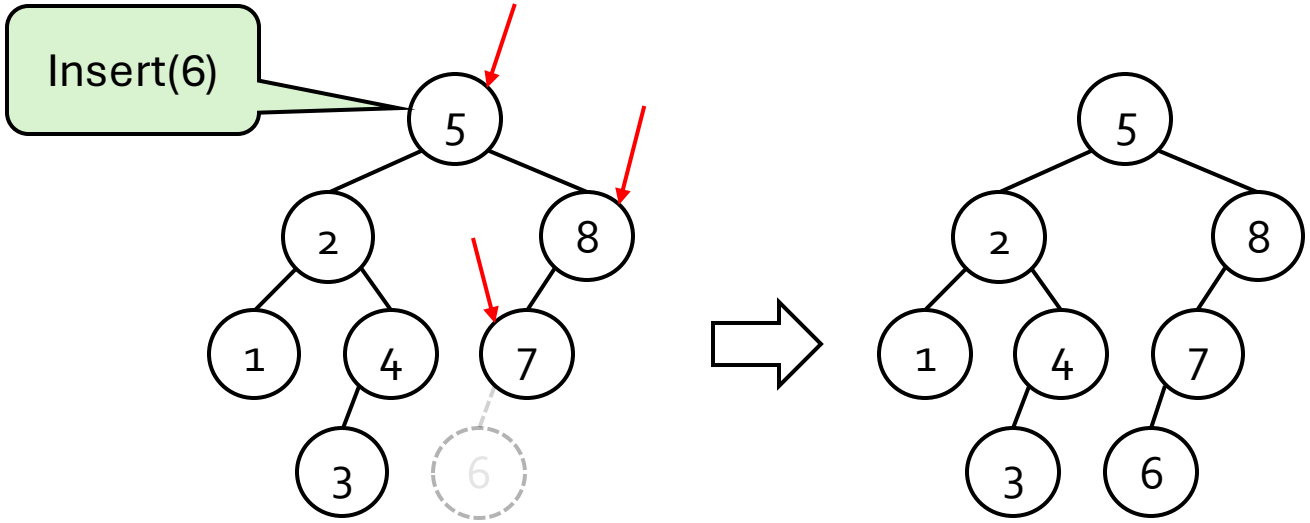
AVL insert

- Insert can violate AVL balance condition
- Can be fixed by a rotation



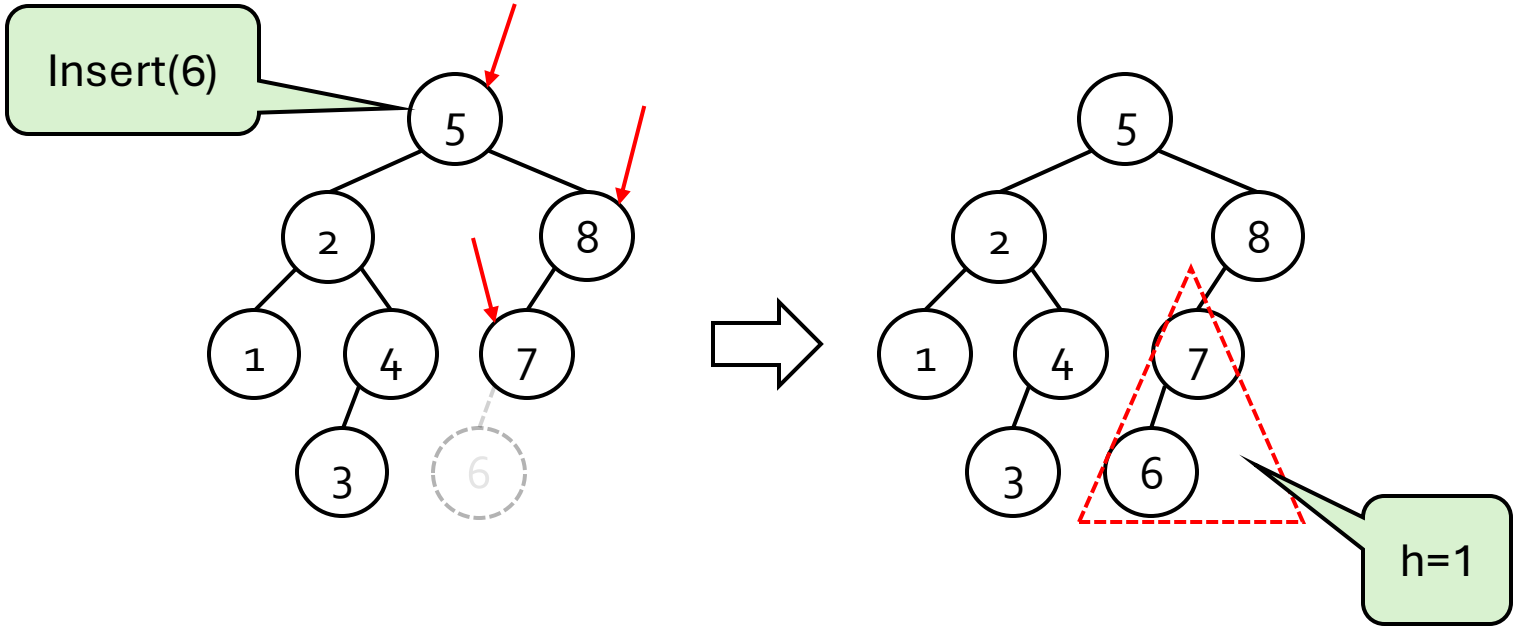
AVL insert

- Insert can violate AVL balance condition
- Can be fixed by a rotation



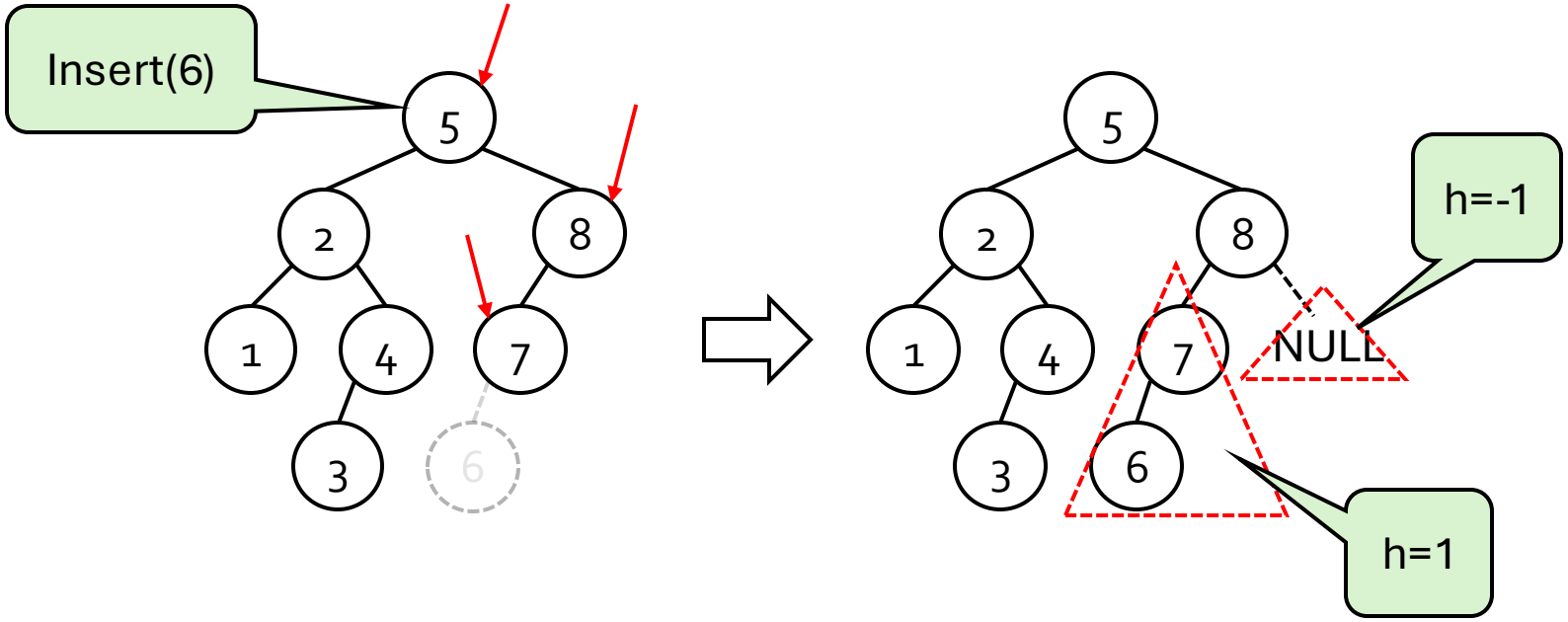
AVL insert

- Insert can violate AVL balance condition
- Can be fixed by a rotation



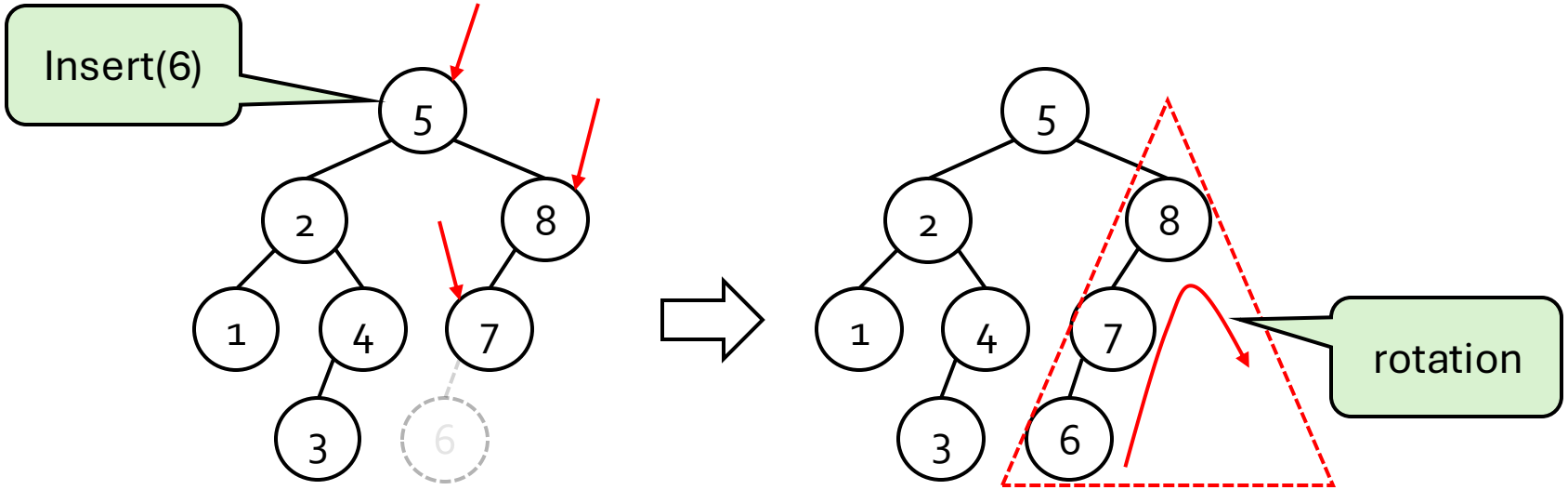
AVL insert

- Insert can violate AVL balance condition
- Can be fixed by a rotation



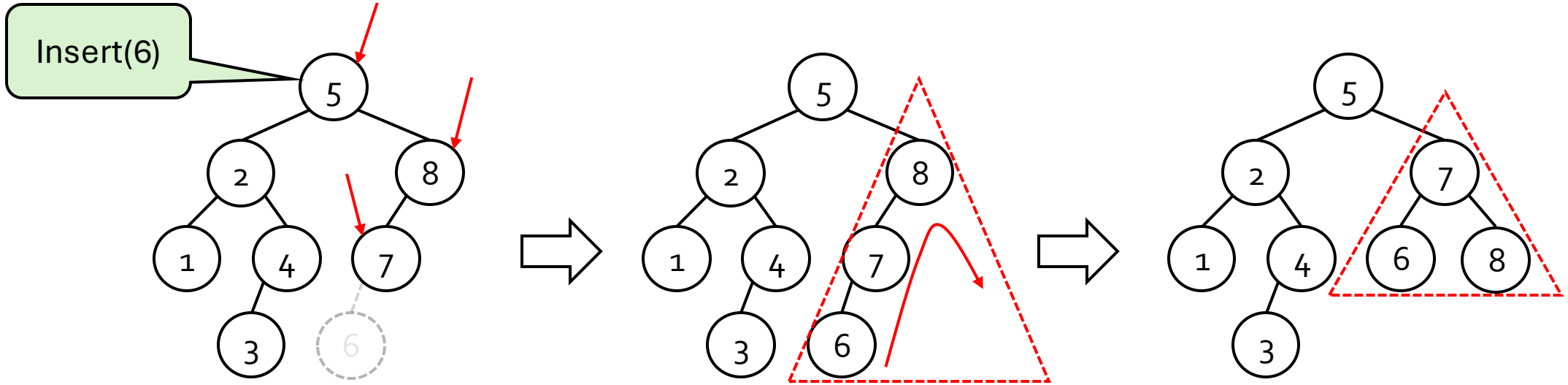
AVL insert

- Insert can violate AVL balance condition
- Can be fixed by a rotation



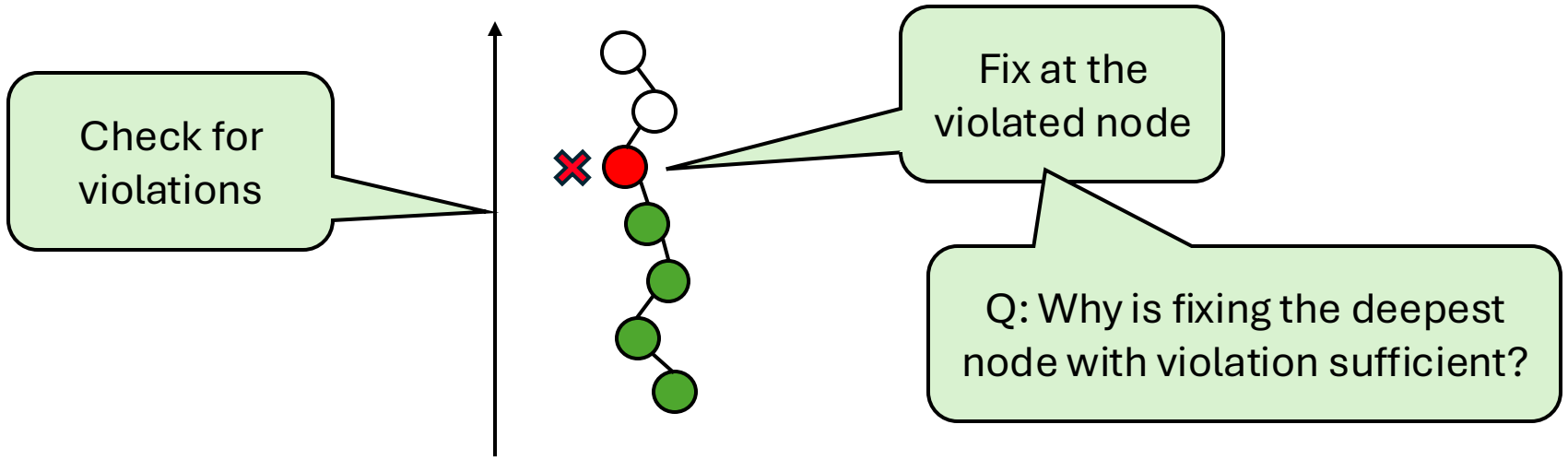
AVL insert

- Insert can violate AVL balance condition
- Can be fixed by a rotation



AVL insert

- Only nodes along path to insertion could have their balance altered
- Follow the path back to root, looking for violations
- Fix the deepest node with violation using single or double rotations



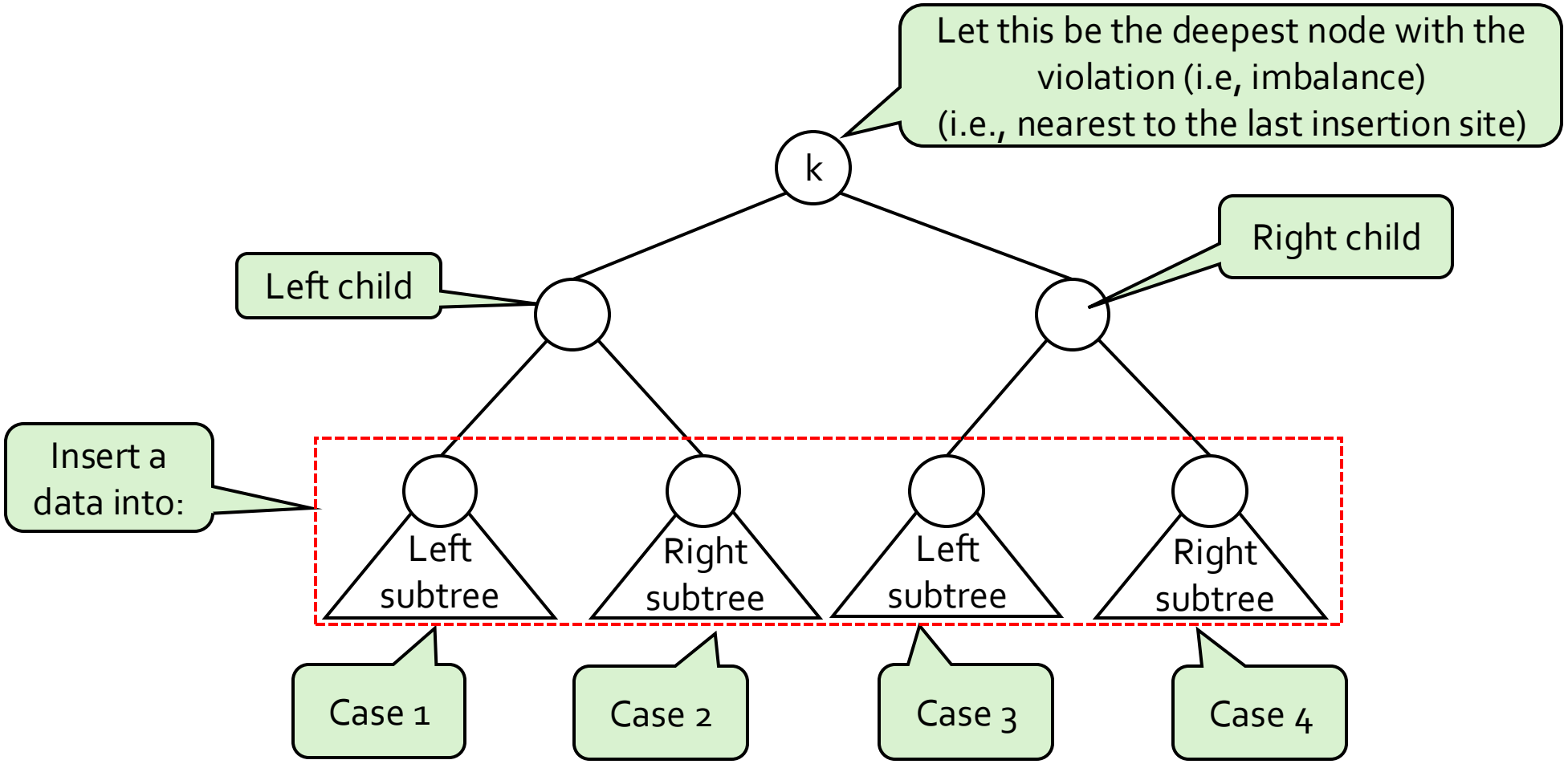
AVL insert: how to fix?

- Assume the violation after insert is at node k
- Four cases leading to violation:
  - CASE 1: Insert into the left subtree of the left child of k
  - CASE 2: Insert into the right subtree of the left child of k
  - CASE 3: Insert into the left subtree of the right child of k
  - CASE 4: Insert into the right subtree of the right child of k
- Cases 1 and 4 handled by "single rotation"
- Cases 2 and 3 handled by "double rotation"

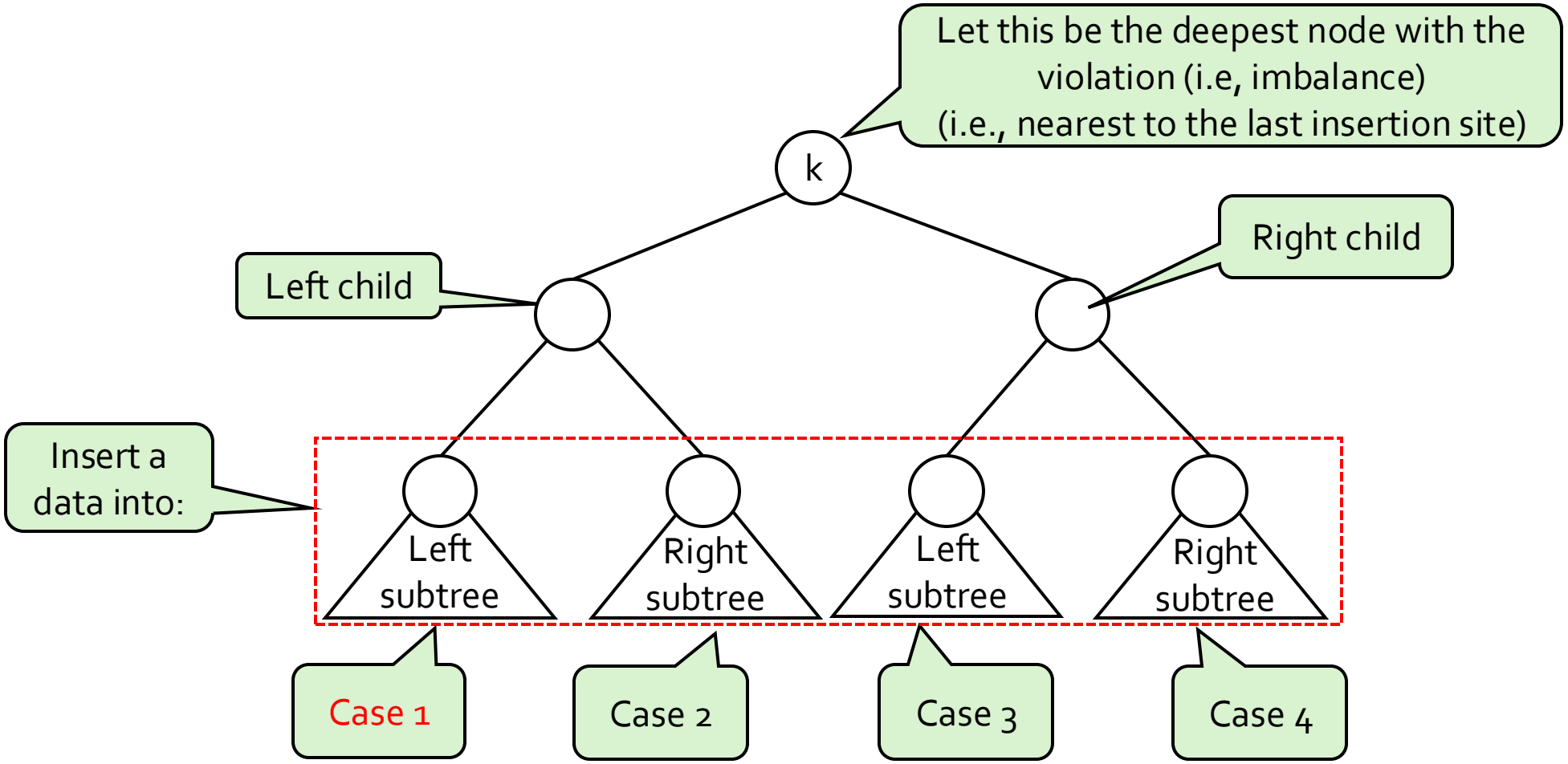
AVL insert: how to fix?

- A general approach
  - Locate the deepest node with the height imbalance
  - Locate which part of its subtree caused the imbalance
    - This will be same as locating the subtree site of insertion
  - Identify the case (1 or 2 or 3 or 4)
  - Do the corresponding rotation.

Identify cases for AVL insert



Identify cases for AVL insert



AVL insert (single rotation)

- Case 1: single rotation right

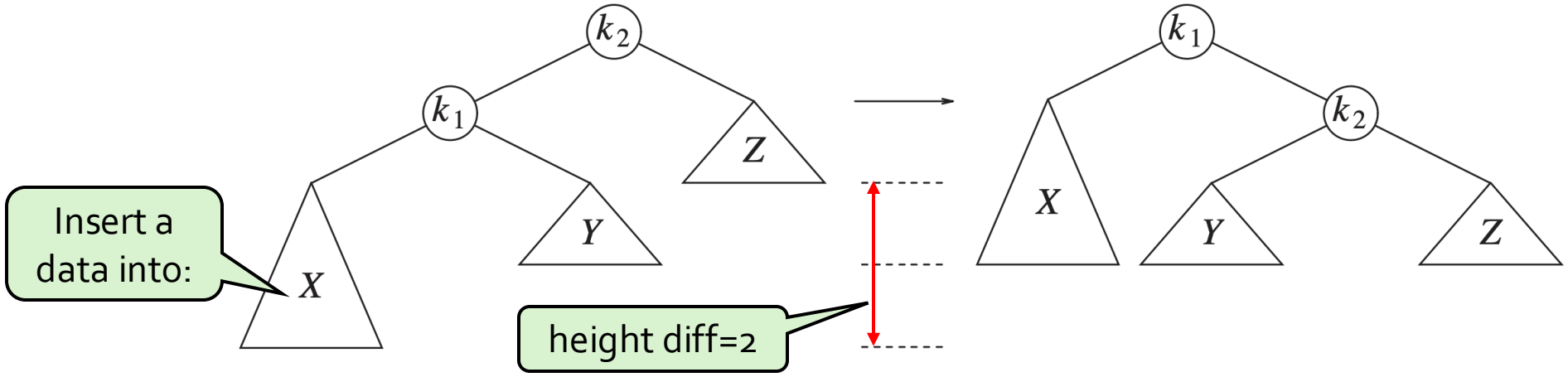


Figure 4.34 Single rotation to fix case 1

AVL insert (single rotation)

- Case 1: single rotation right

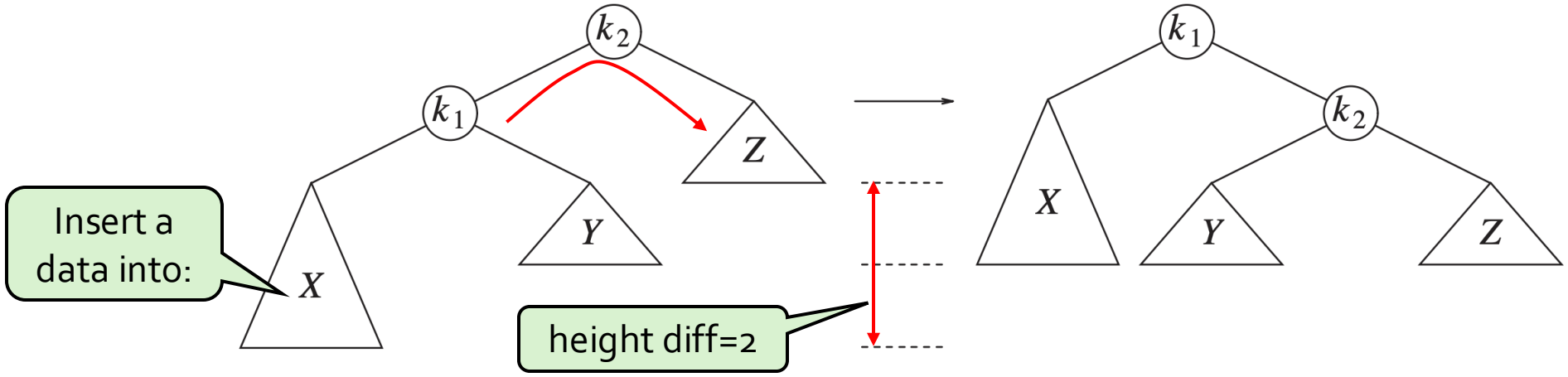


Figure 4.34 Single rotation to fix case 1

AVL insert (single rotation)

- Case 1: single rotation right

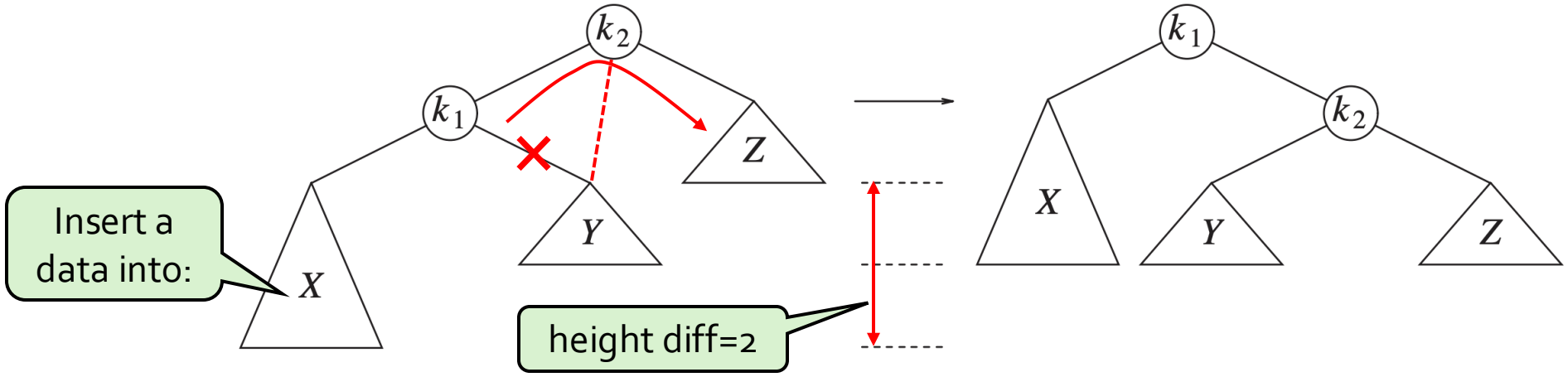


Figure 4.34 Single rotation to fix case 1

AVL insert (single rotation)

- Case 1: single rotation right

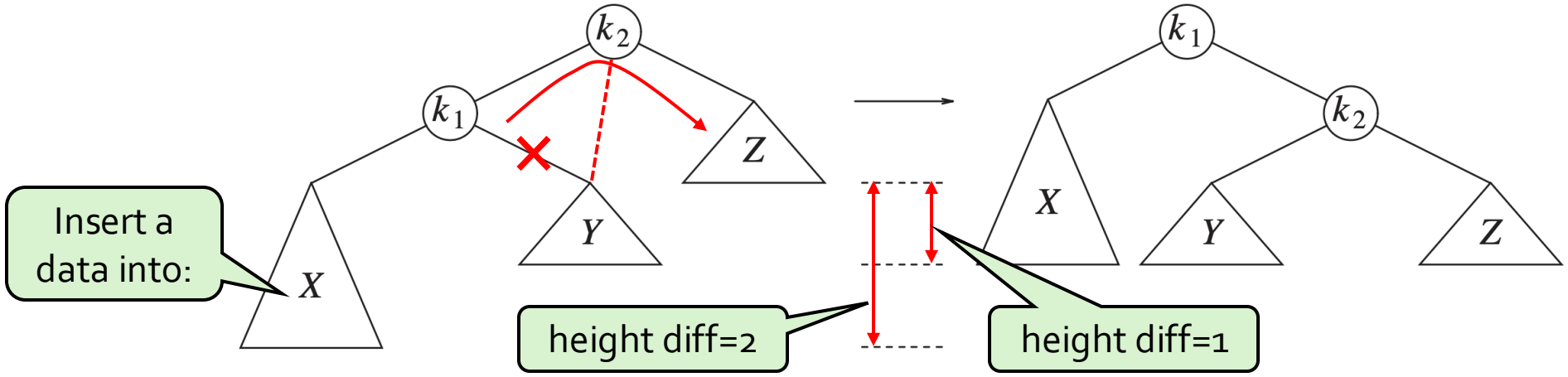


Figure 4.34 Single rotation to fix case 1

AVL insert (single rotation)

- Case 1: single rotation right

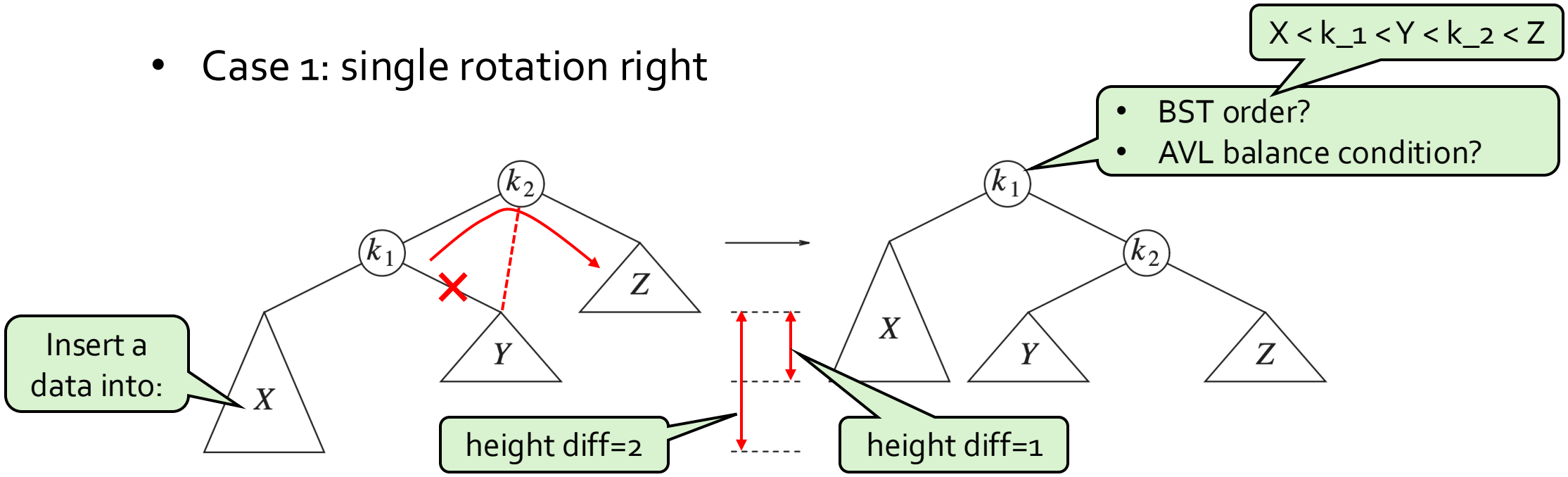
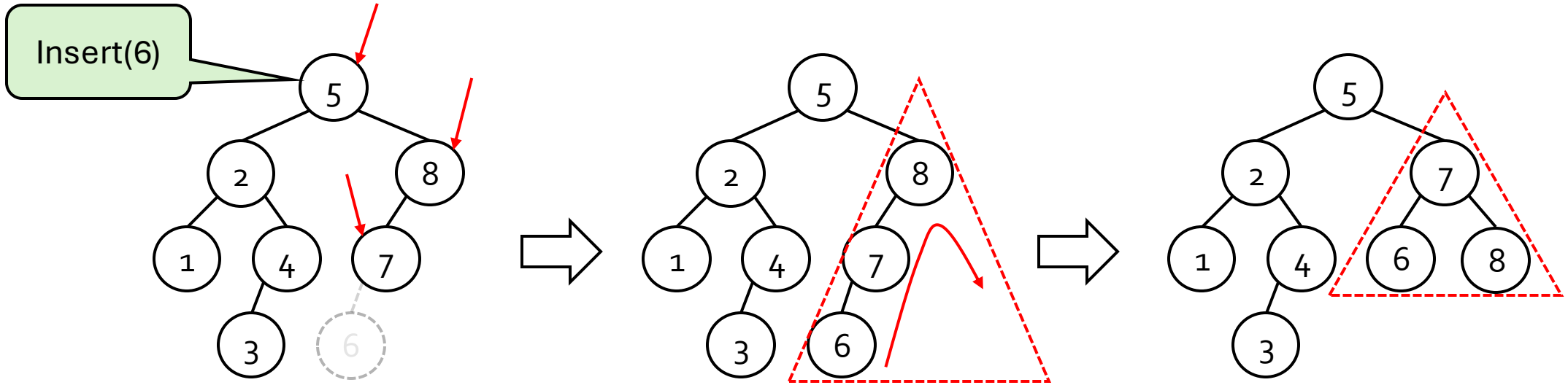


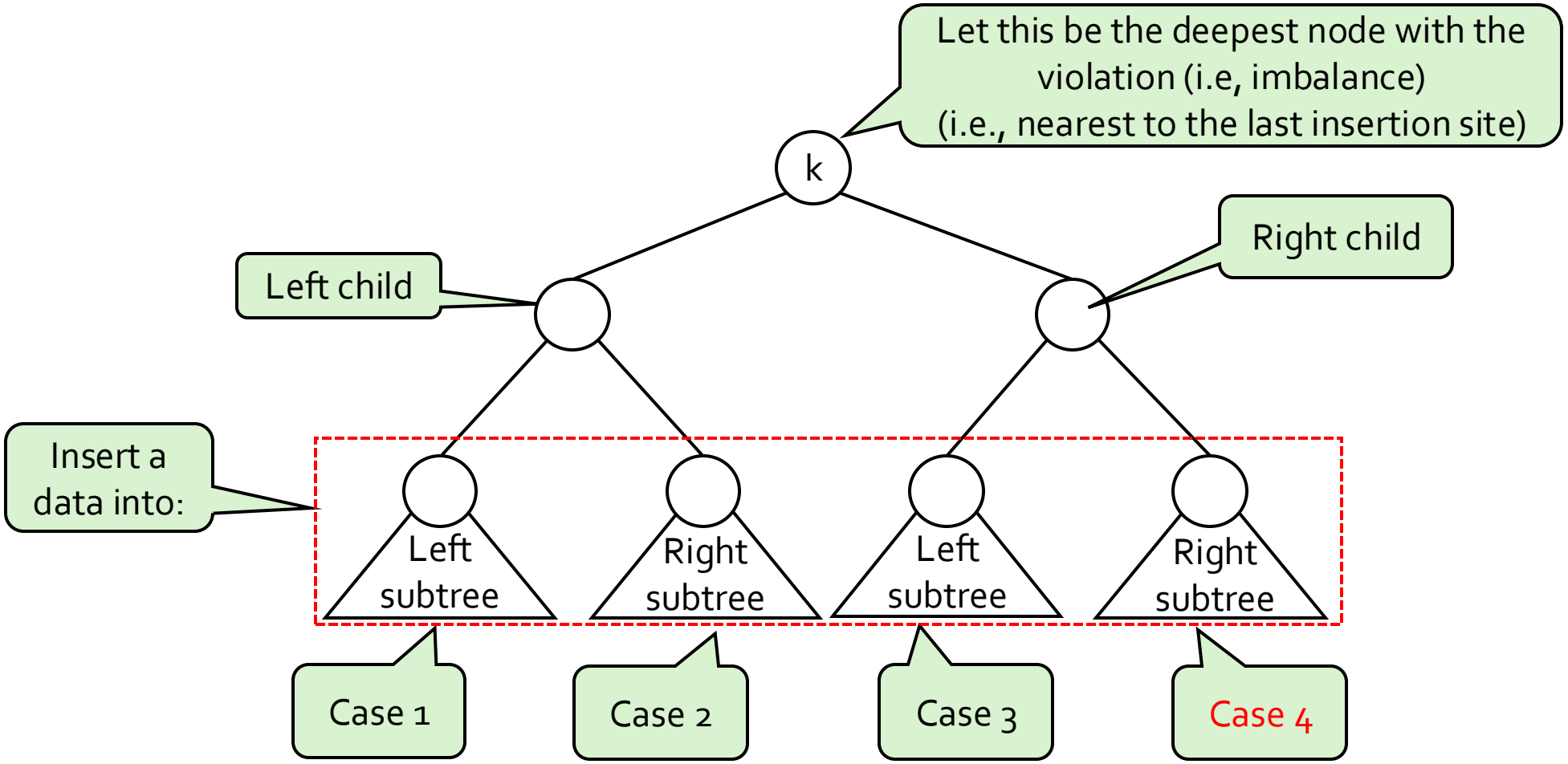
Figure 4.34 Single rotation to fix case 1

AVL insert (single rotation)

- Case 1 example



Identify cases for AVL insert



AVL insert (single rotation)

- Case 4: single rotation left

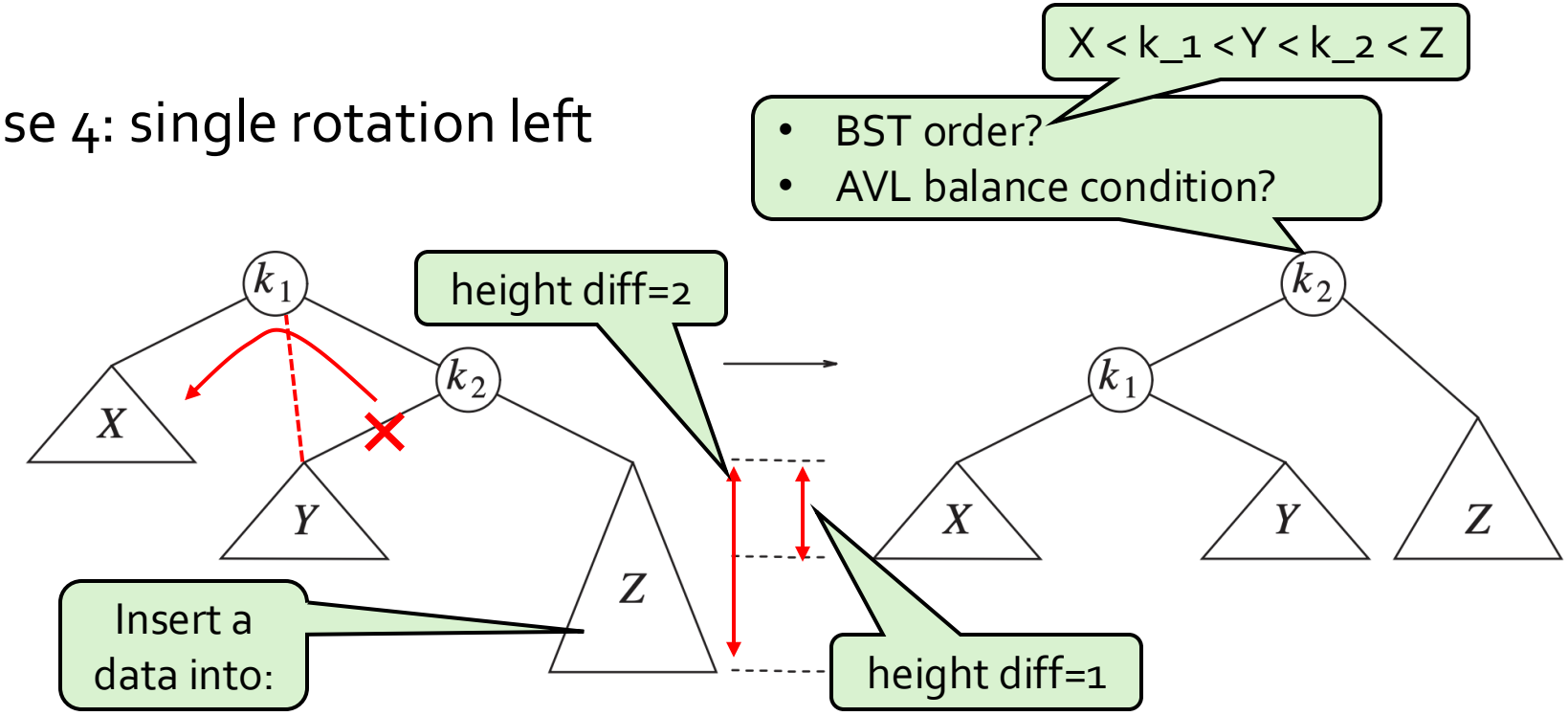
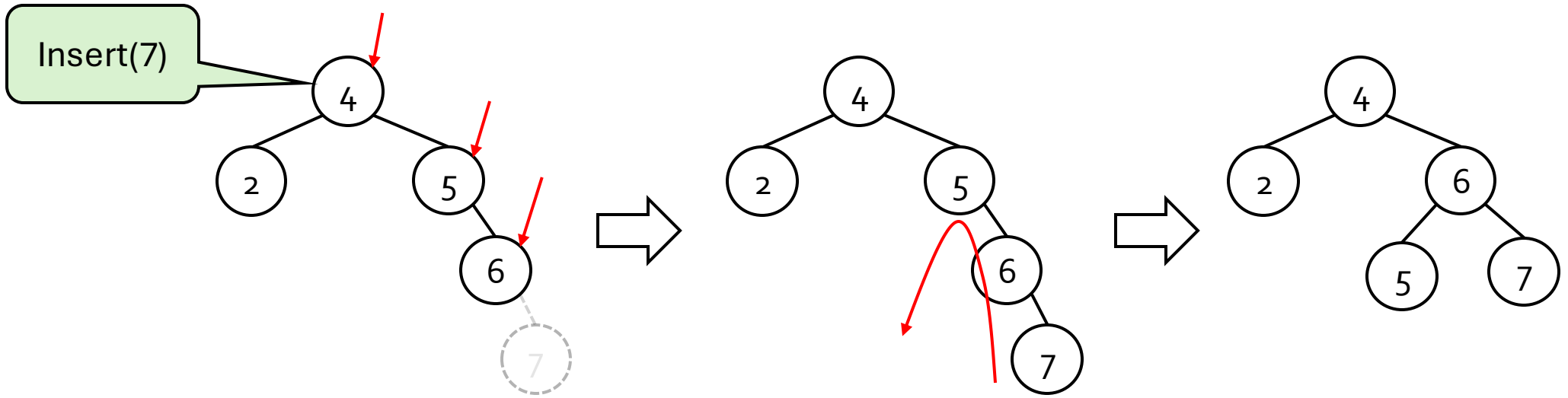


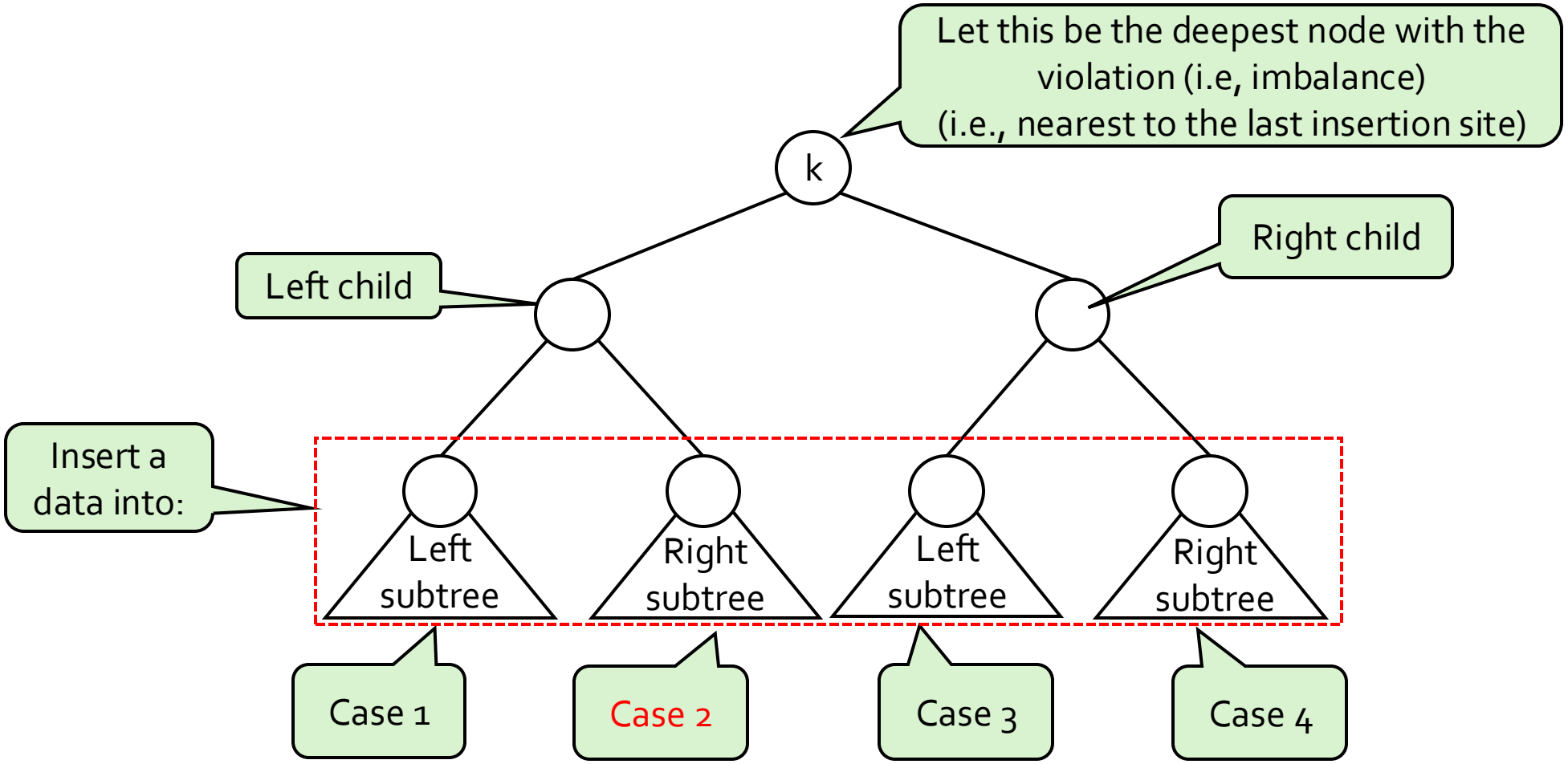
Figure 4.36 Single rotation fixes case 4

AVL insert (single rotation)

- Case 4 example



Identify cases for AVL insert



AVL insert (double rotation)

- Case 2: single rotation fails

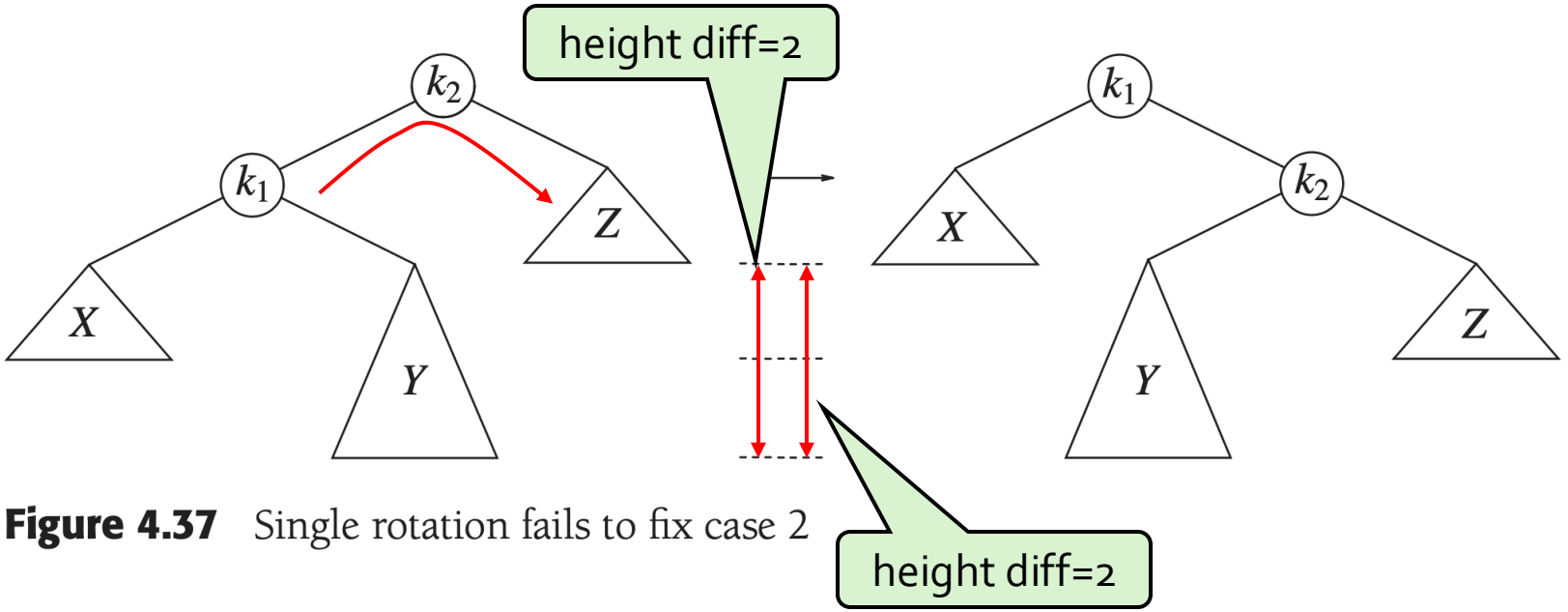


Figure 4.37 Single rotation fails to fix case 2

AVL insert (double rotation)

- Case 2: left-right double rotation

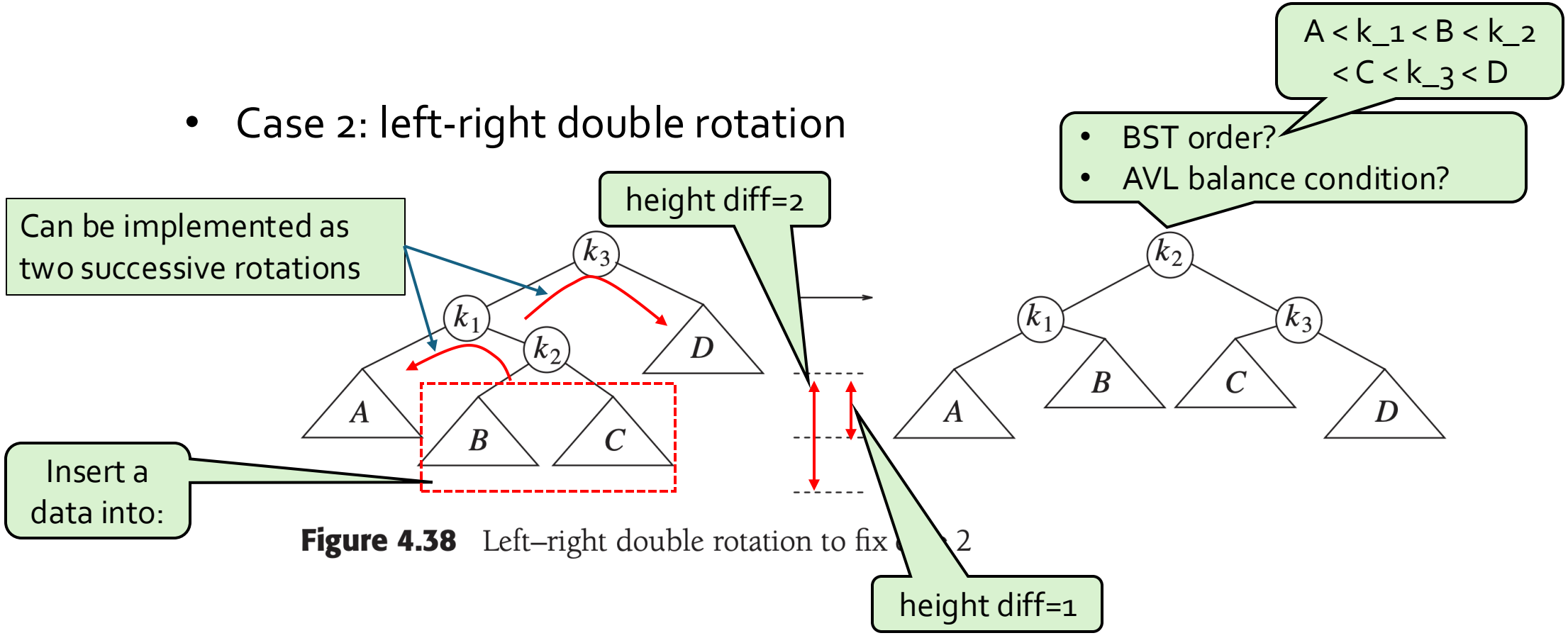
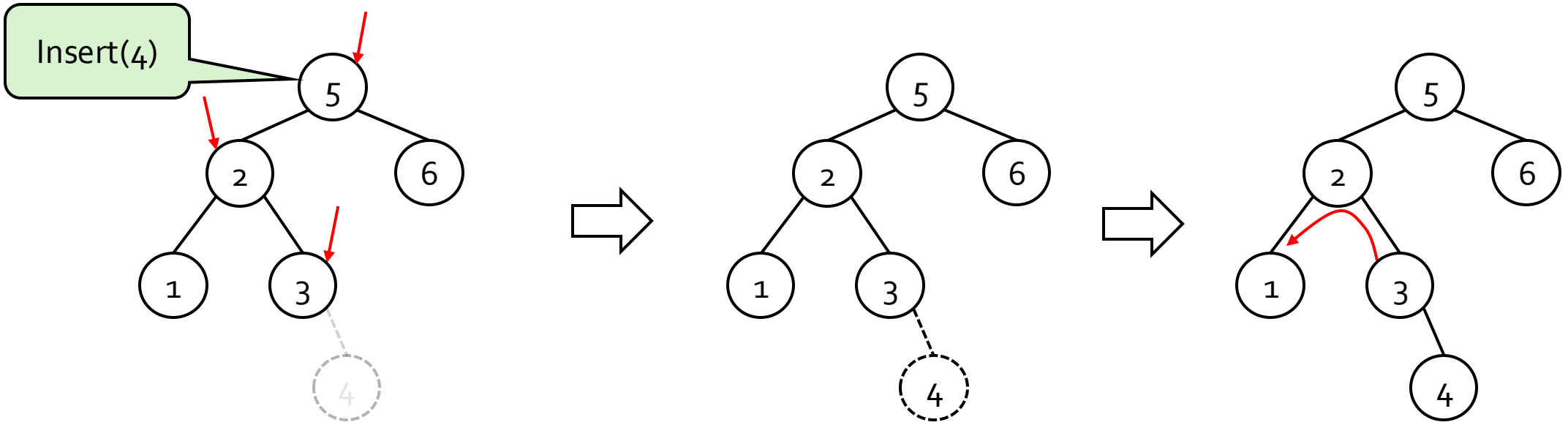


Figure 4.38 Left–right double rotation to fix

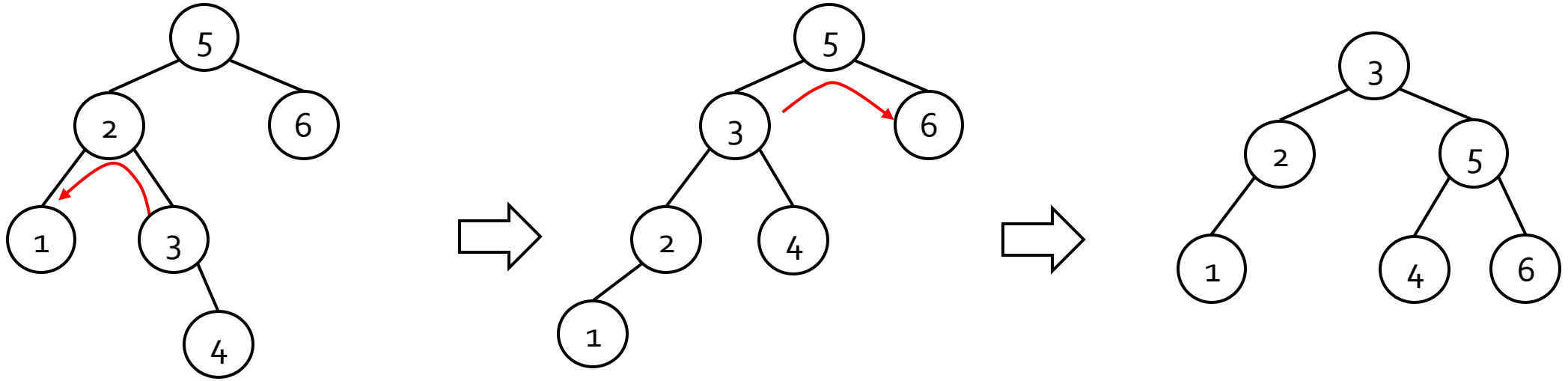
AVL insert (double rotation)

- Case 2 example

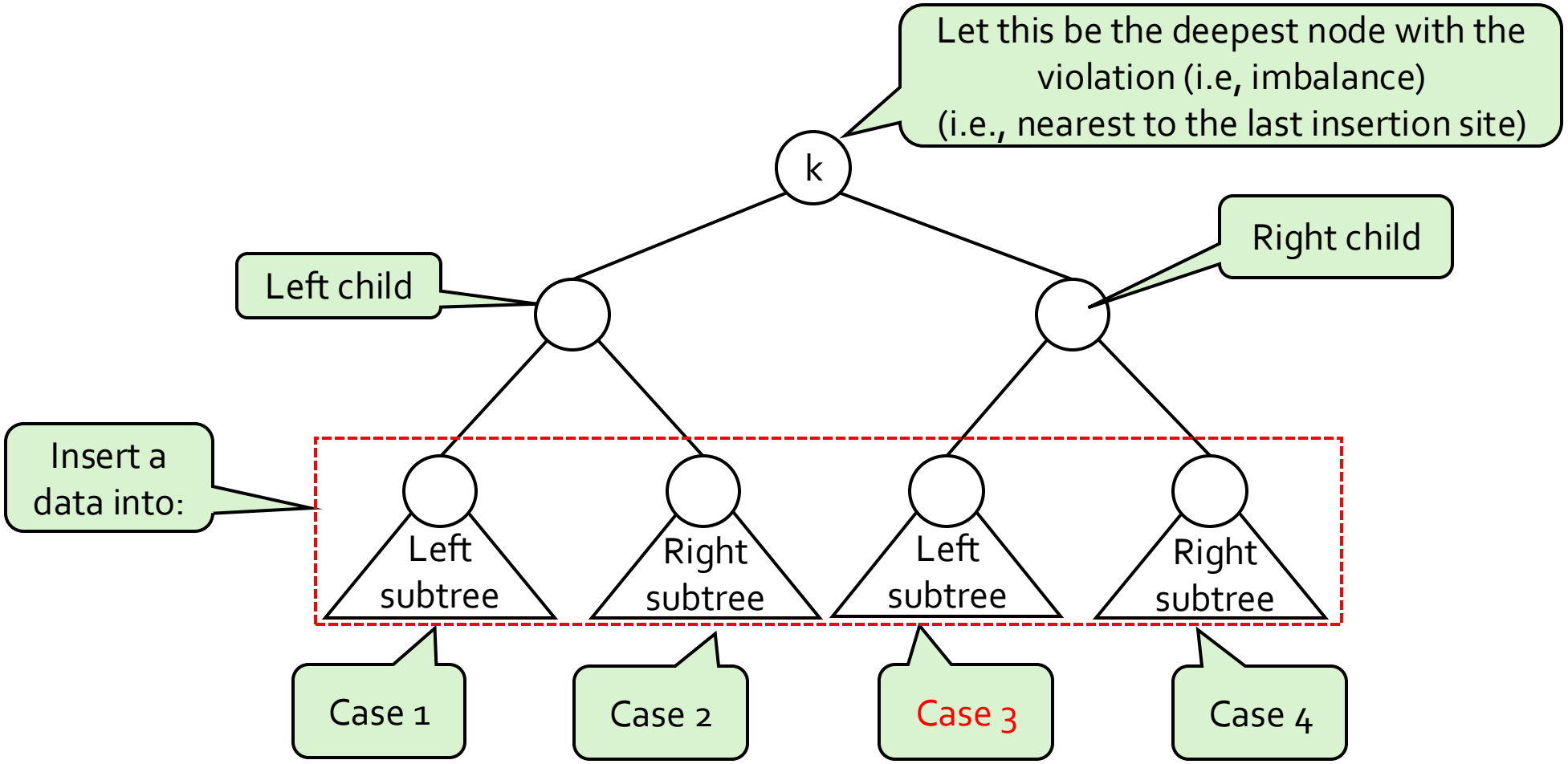


AVL insert (double rotation)

- Case 2 example



Identify cases for AVL insert



AVL insert (double rotation)

- Case 3: right-left double rotation

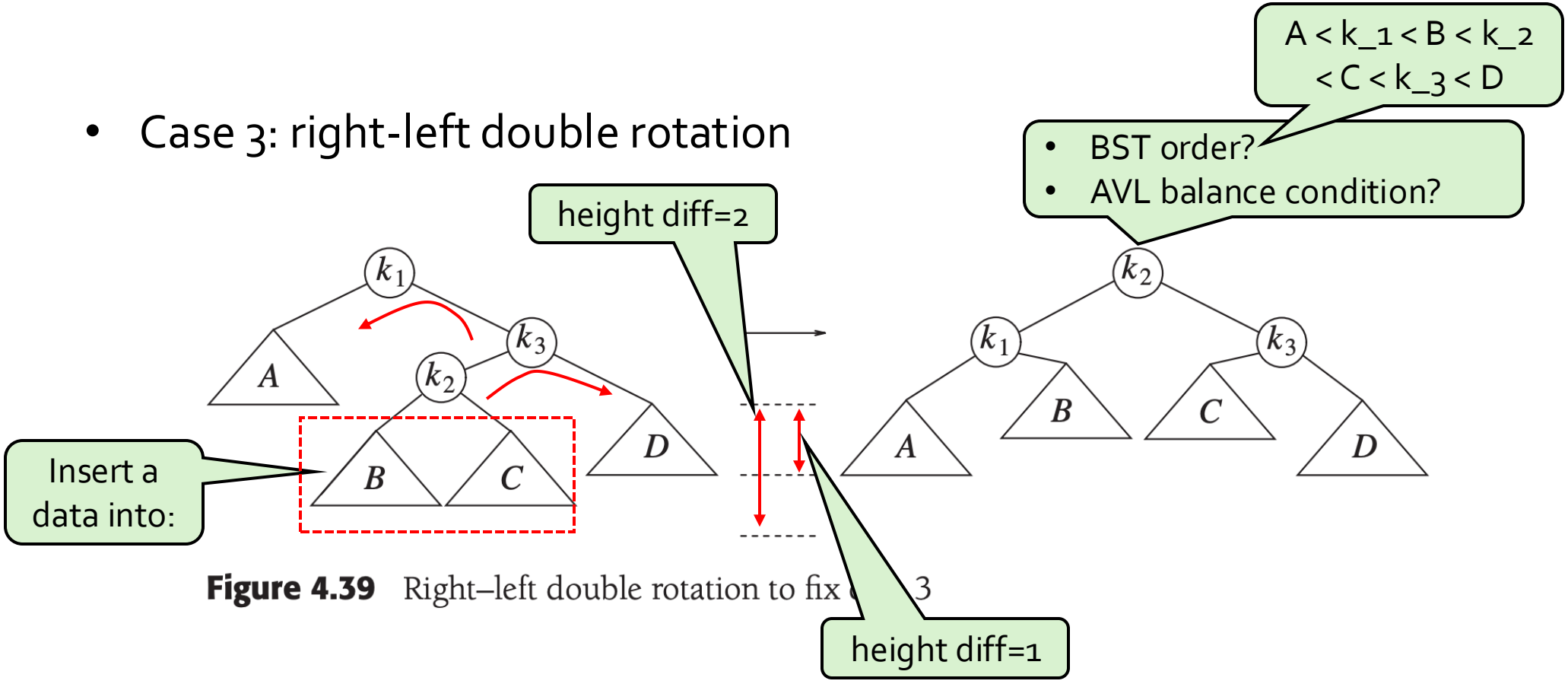
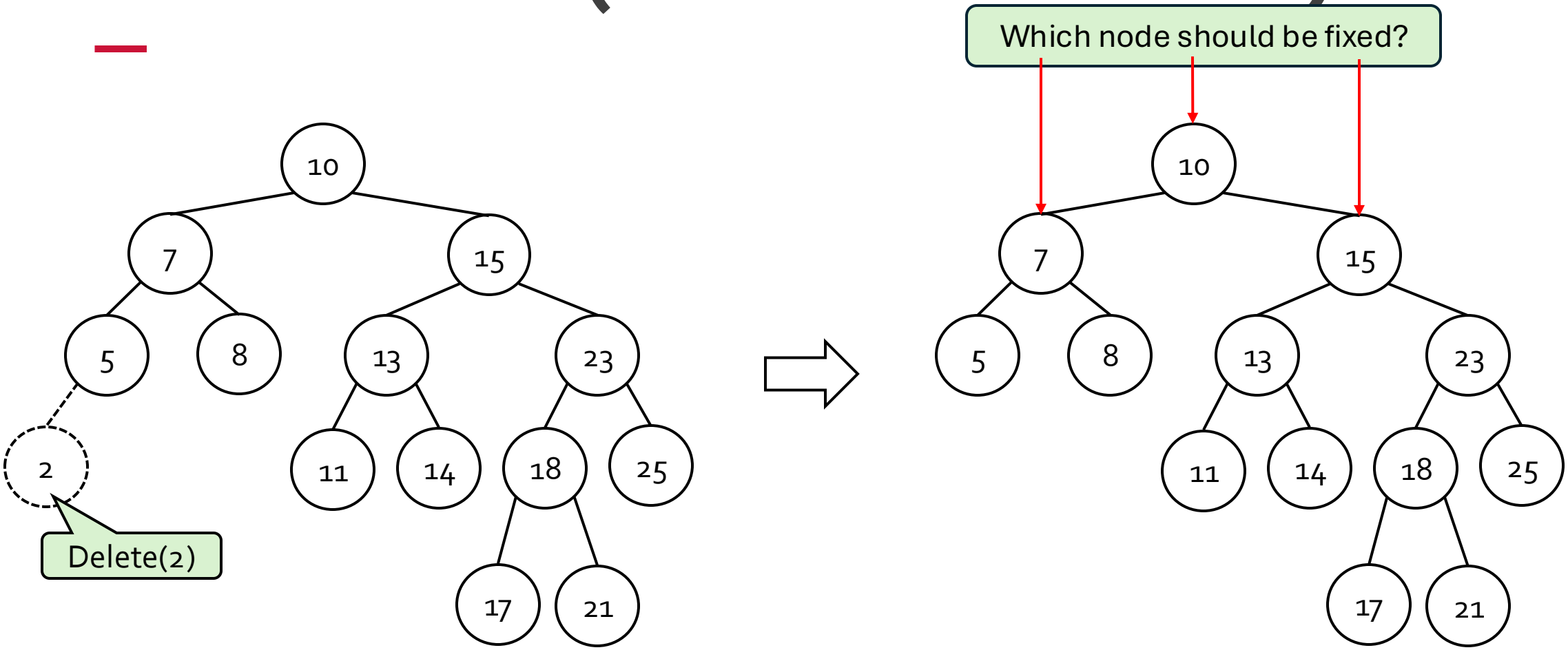
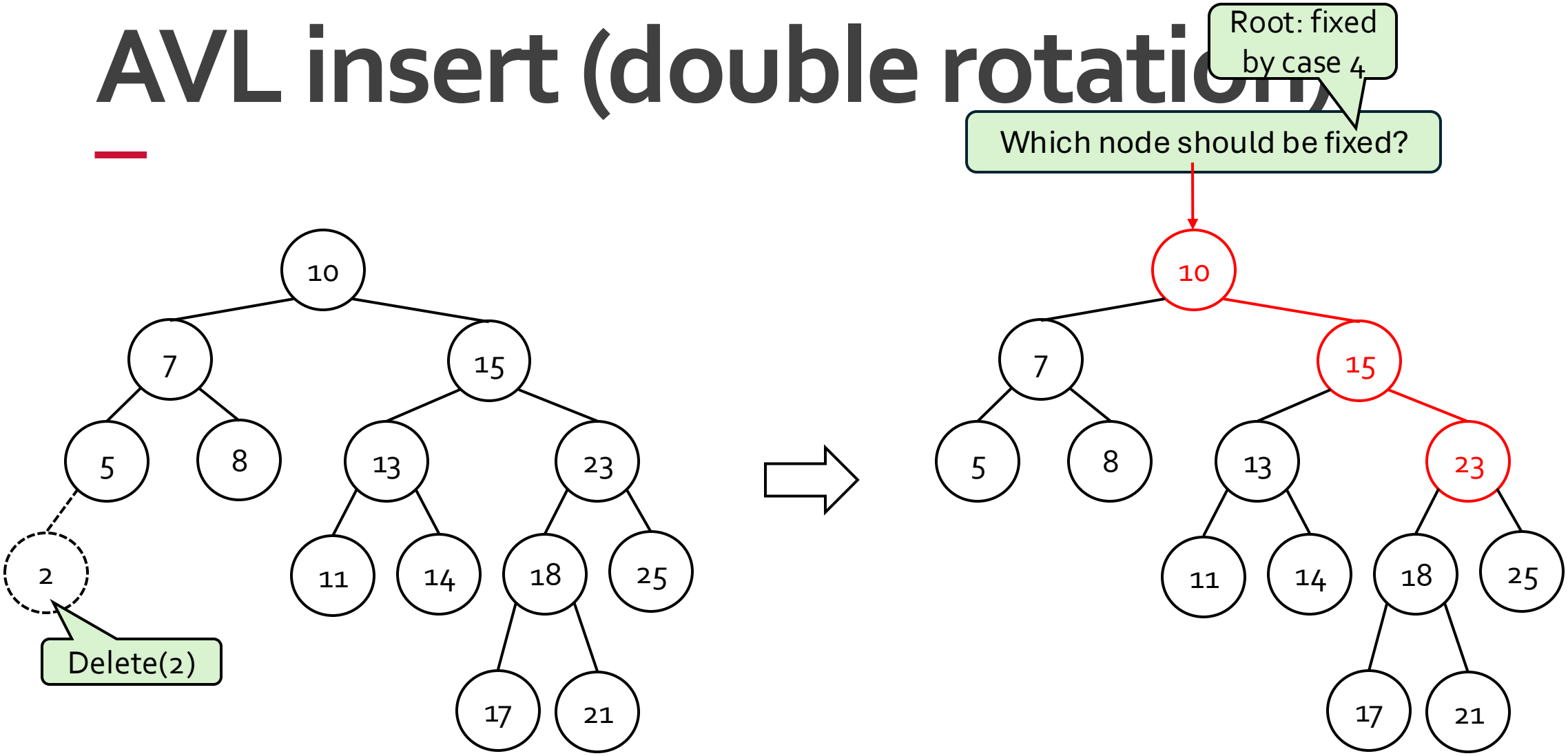


Figure 4.39 Right–left double rotation to fix 3

AVL insert (double rotation)



AVL insert (double rotation)



AVL deletion

- Locate the deepest node with the height imbalance (along the deletion path)
- Locate which part of its subtree caused the imbalance
  - The child-node that has higher height leads to the imbalance
- Identify the case (1 or 2 or 3 or 4)
- Do the corresponding rotation

AVL remove: lazy deletion

- Assume remove accomplished using lazy deletion
  - Removed nodes only marked as deleted, but not actually removed from BST until some cutoff is reached
  - Unmarked when same object re-inserted
  - Re-allocation time avoided
  - Does not affect O(log2 N) height as long as deleted nodes are not in the majority
  - Does require additional memory per node
- Can accomplish remove without lazy deletion

AVL heights

- Usually, how do we get the height of a node?
- DFS (Depth-first search) Tree traversal
  - Complexity O(n): all descendants of this node
- AVL Tree insertion / deletion frequently uses "height"
  - Locate the deepest node with the height imbalance
  - Locate which part of its subtree caused the imbalance
- This is extremely slow
- Maintain the height value in each AVL node to avoid tree traversal
  - Every time when an insertion/deletion finishes, update the height value of all nodes on the path, from the leaf to root. Complexity O(h), h = lg(n)
  - Insertion / deletion will use the height value of a node to perform Step 1 and 2

AVL tree implementation

```
1   struct AvlNode
2   {
3       Comparable element;
4       AvlNode   *left;
5       AvlNode   *right;
6       int       height;
7
8       AvlNode( const Comparable & ele, AvlNode *lt, AvlNode *rt, int h = 0 )
9         : element{ ele }, left{ lt }, right{ rt }, height{ h } { }
10
11      AvlNode( Comparable && ele, AvlNode *lt, AvlNode *rt, int h = 0 )
12        : element{ std::move( ele ) }, left{ lt }, right{ rt }, height{ h } { }
13  };
```

For each node

Figure 4.40 Node declaration for AVL trees

AVL tree implementation

```
1   /**
2    * Return the height of node t or -1 if nullptr.
3    */
4   int height( AvlNode *t ) const
5   {
6       return t == nullptr ? -1 : t->height;
7   }
```

Figure 4.41 Function to compute height of an AVL node

AVL

```
1   /**
2    * Internal method to insert into a subtree.
3    * x is the item to insert.
4    * t is the node that roots the subtree.
5    * Set the new root of the subtree.
6    */
7   void insert( const Comparable & x, AvlNode * & t )
8   {
9       if( t == nullptr )
10          t = new AvlNode{ x, nullptr, nullptr };
11      else if( x < t->element )
12          insert( x, t->left );
13      else if( t->element < x )
14          insert( x, t->right );
15
16      balance( t );
17  }
```

BST insert

AVL


```
19  static const int ALLOWED_IMBALANCE = 1;
20
21  // Assume t is balanced or within one of being balanced
22  void balance( AvlNode * & t )
23  {
24      if( t == nullptr )
25          return;
26
27      if( height( t->left ) - height( t->right ) > ALLOWED_IMBALANCE )
28          if( height( t->left->left ) >= height( t->left->right ) )
29              rotateWithLeftChild( t );
30          else
31              doubleWithLeftChild( t );
32      else
33      if( height( t->right ) - height( t->left ) > ALLOWED_IMBALANCE )
34          if( height( t->right->right ) >= height( t->right->left ) )
35              rotateWithRightChild( t );
36          else
37              doubleWithRightChild( t );
38
39      t->height = max( height( t->left ), height( t->right ) ) + 1;
40  }
```

Figure 4.42 Insertion into an AVL tree

AVL


```
19  static const int ALLOWED_IMBALANCE = 1;
20
21  // Assume t is balanced or within one of being balanced
22  void balance( AvlNode * & t )
23  {
24      if( t == nullptr )
25          return;
26
27      if( height( t->left ) - height( t->right ) > ALLOWED_IMBALANCE )
28          if( height( t->left->left ) >= height( t->left->right ) )
29              rotateWithLeftChild( t );
30          else
31              doubleWithLeftChild( t );
32      else
33      if( height( t->right ) - height( t->left ) > ALLOWED_IMBALANCE )
34          if( height( t->right->right ) >= height( t->right->left ) )
35              rotateWithRightChild( t );
36          else
37              doubleWithRightChild( t );
38
39      t->height = max( height( t->left ), height( t->right ) ) + 1;
40  }
```

Figure 4.42 Insertion into an AVL tree

AVL

```
19  static const int ALLOWED_IMBALANCE = 1;
20
21  // Assume t is balanced or within one of being balanced
22  void balance( AvlNode * & t )
23  {
24      if( t == nullptr )
25          return;
26
27      if( height( t->left ) - height( t->right ) > ALLOWED_IMBALANCE )
28          if( height( t->left->left ) >= height( t->left->right ) )
29              rotateWithLeftChild( t );
30          else
31              doubleWithLeftChild( t );
32      else
33      if( height( t->right ) - height( t->left ) > ALLOWED_IMBALANCE )
34          if( height( t->right->right ) >= height( t->right->left ) )
35              rotateWithRightChild( t );
36          else
37              doubleWithRightChild( t );
38
39      t->height = max( height( t->left ), height( t->right ) ) + 1;
40  }
```

Adjust height

Figure 4.42 Insertion into an AVL tree

AVL tree implementation

```
1   /**
2    * Rotate binary tree node with left child.
3    * For AVL trees, this is a single rotation for case 1.
4    * Update heights, then set new root.
5    */
6   void rotateWithLeftChild( AvlNode * & k2 )
7   {
8       AvlNode *k1 = k2->left;
9       k2->left = k1->right;
10      k1->right = k2;
11      k2->height = max( height( k2->left ), height( k2->right ) ) + 1;
12      k1->height = max( height( k1->left ), k2->height ) + 1;
13      k2 = k1;
14  }
```

Figure 4.44 Routine to perform single rotation

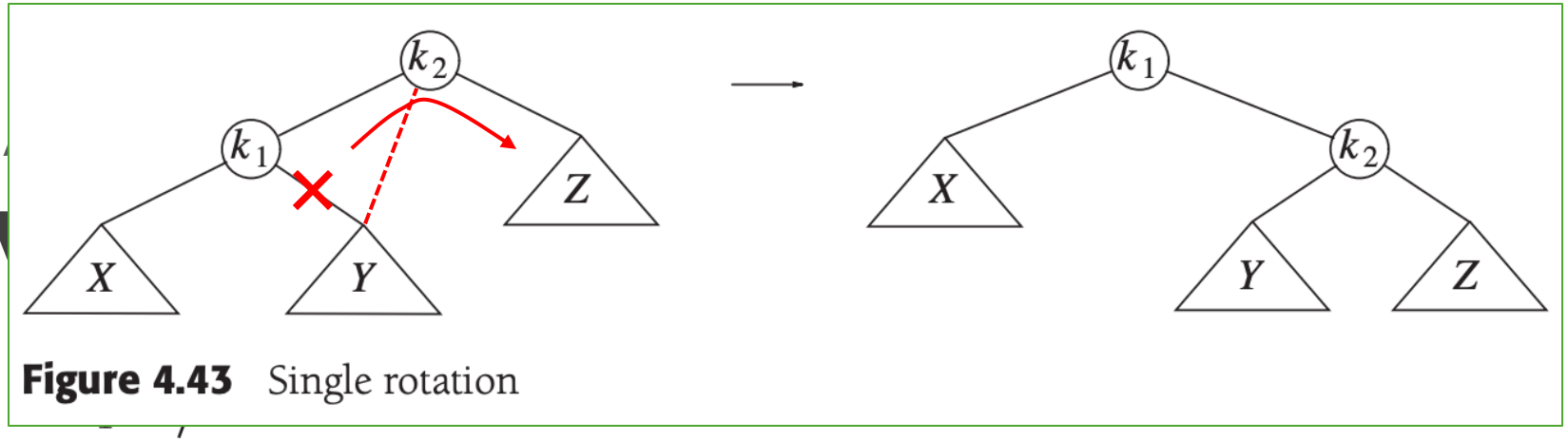


Figure 4.43 Single rotation

rotateWithRightChild is similar

```
2    * Rotate binary tree node with left child.
3    * For AVL trees, this is a single rotation for case 1.
     * Update heights, then set new root.
     */
6   void rotateWithLeftChild( AvlNode * & k2 )
7   {
8       AvlNode *k1 = k2->left;
9       k2->left = k1->right;
10      k1->right = k2;
11      k2->height = max( height( k2->left ), height( k2->right ) ) + 1;
12      k1->height = max( height( k1->left ), k2->height ) + 1;
13      k2 = k1;
14  }
```

Figure 4.44 Routine to perform single rotation

AVL tree implementation

```
1   /**
2    * Double rotate binary tree node: first left child
3    * with its right child; then node k3 with new left child.
4    * For AVL trees, this is a double rotation for case 2.
5    * Update heights, then set new root.
6    */
7   void doubleWithLeftChild( AvlNode * & k3 )
8   {
9       rotateWithRightChild( k3->left );
10      rotateWithLeftChild( k3 );
11  }
```

Figure 4.46 Routine to perform double rotation

#1: rotateWithRightChild

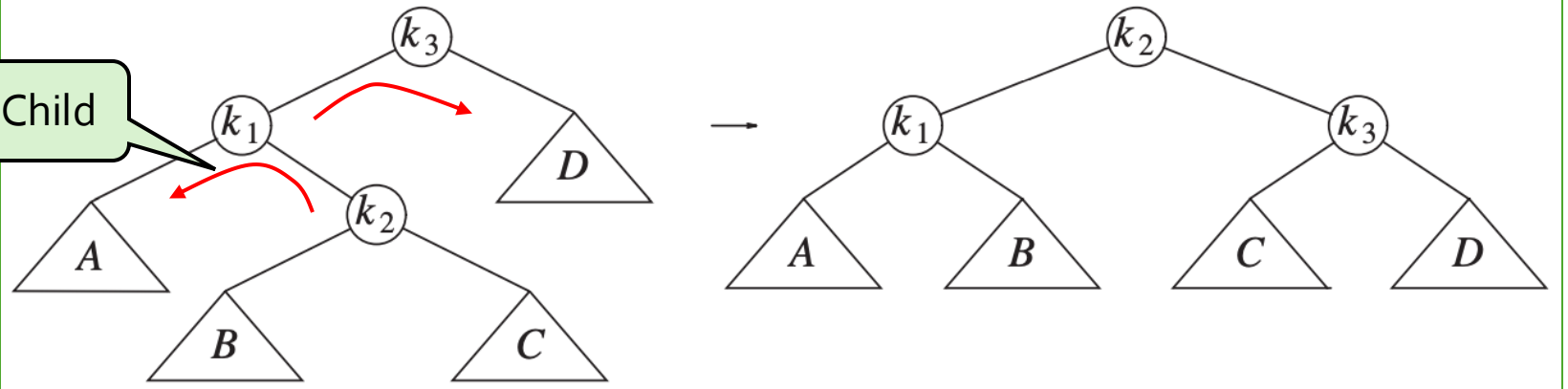


Figure 4.45 Double rotation

```
3     * with its right child; then node k3 with new left child.
4     * For AVL trees, this is a double rotation for case 2.
5     * Update heights, then set new root.
6     */
7    void doubleWithLeftChild( AvlNode * & k3 )
8    {
9        rotateWithRightChild( k3->left );
10       rotateWithLeftChild( k3 );
11   }
```

#1: rotateWithRightChild

Figure 4.46 Routine to perform double rotation

#2: rotateWithLeftChild

AV

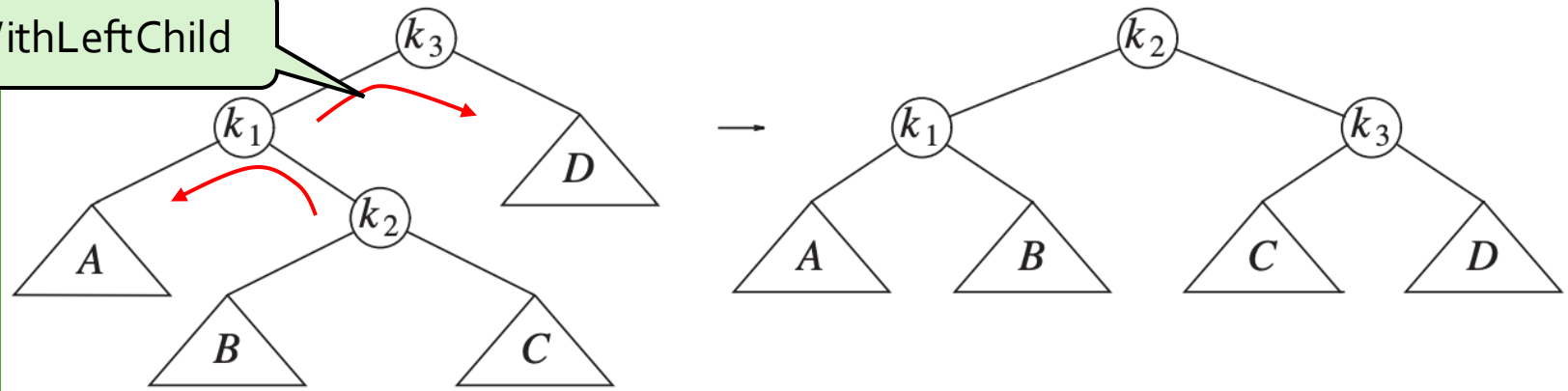


Figure 4.45 Double rotation

```
3     * with its right child; then node k3 with new left child.
4     * For AVL trees, this is a double rotation for case 2.
5     * Update heights, then set new root.
6     */
7    void doubleWithLeftChild( AvlNode * & k3 )
8    {
         rotateWithRightChild( k3->left );
         rotateWithLeftChild( k3 );
11   }
```

#1: rotateWithLeftChild

Figure 4.46 Routine to perform double rotation