AVL Trees

Outline

- AVL Tree
- Implementation of AVL Tree

AVL Trees

- An AVL (Adelson-Velskii and Landis) tree is a binary search tree with a balance condition.
- The balance condition must be easy to maintain, and it ensures that the depth of the tree is O(log N).
- The simplest idea is to require that the left and right subtrees have the same height.

AVL Trees

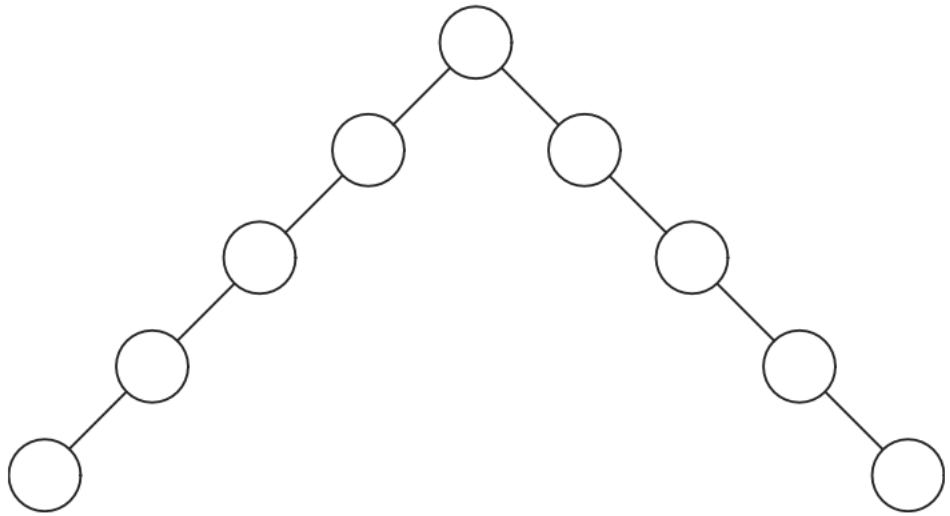


Figure 4.31 A bad binary tree. Requiring balance at the root is not enough.

AVL Trees

- An AVL tree is identical to a binary search tree, except that for every node in the tree, the height of the left and right subtrees can differ by at most 1. (The height of an empty tree is defined to be −1.)

AVL Trees

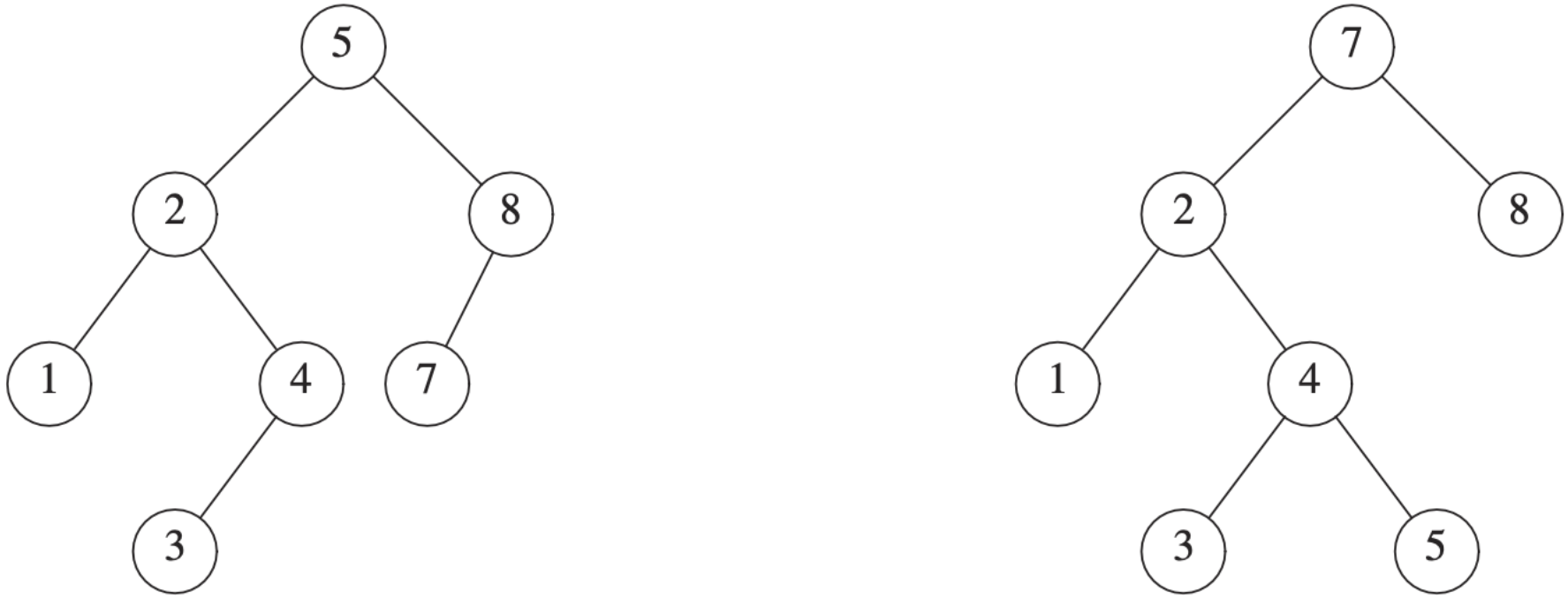


Figure 4.32 Two binary search trees. Only the left tree is AVL.

AVL Trees

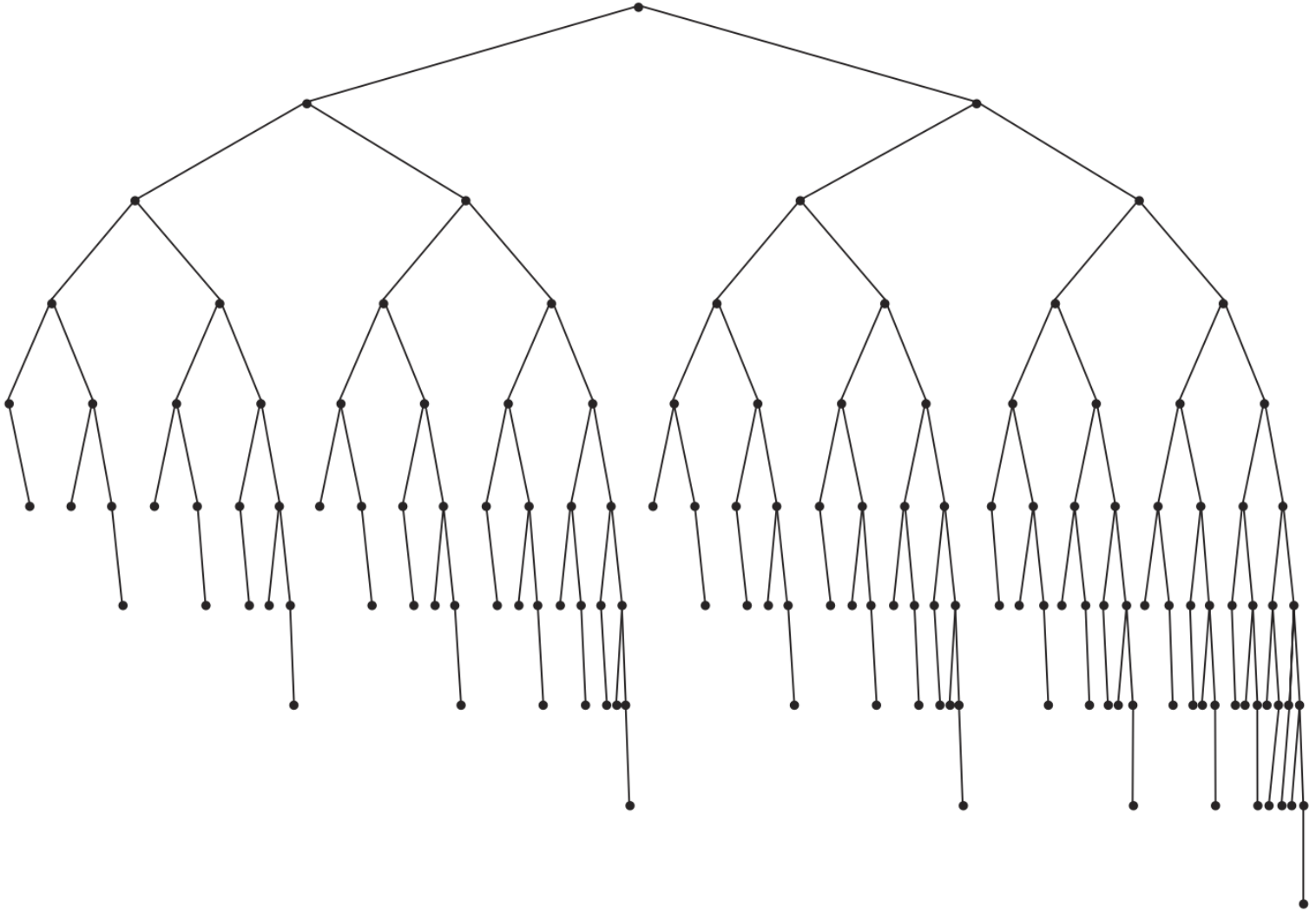


Figure 4.33 Smallest AVL tree of height 9

AVL Trees

- Let us call the node that must be rebalanced α.
- Since any node has at most two children, and a height imbalance requires that α's two subtrees' heights differ by two, it is easy to see that a violation might occur in four cases:

1. An insertion into the left subtree of the left child of α
2. An insertion into the right subtree of the left child of α
3. An insertion into the left subtree of the right child of α
4. An insertion into the right subtree of the right child of α

AVL Trees

- The first case, in which the insertion occurs on the “outside” (i.e., left–left or right– right), is fixed by a **single rotation** of the tree.
- The second case, in which the insertion occurs on the “inside” (i.e., left–right or right–left) is handled by the slightly more complex **double rotation**.

Single Rotation

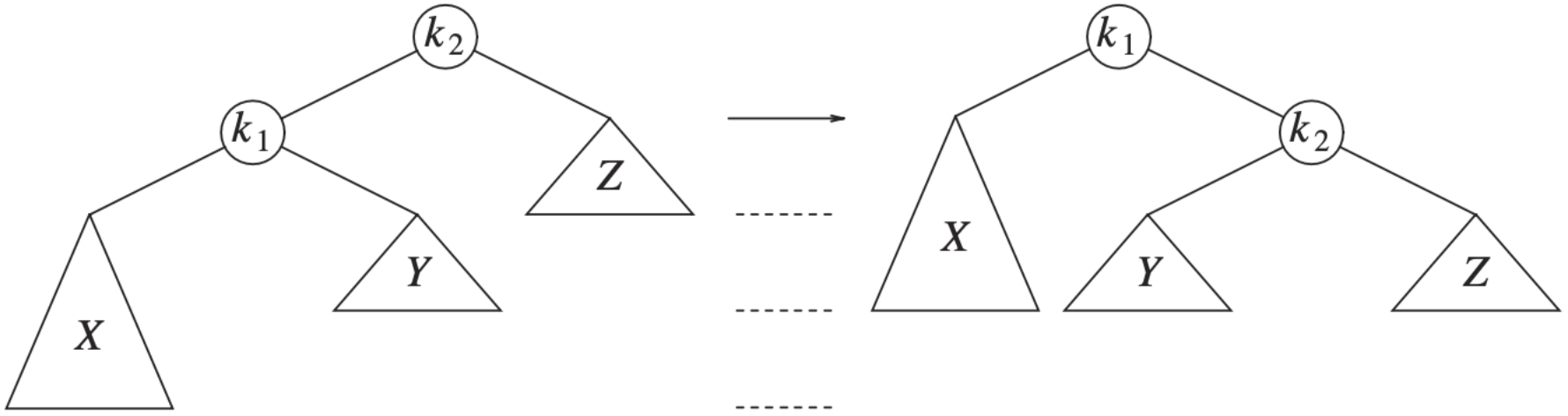


Figure 4.34 Single rotation to fix case 1

Single Rotation

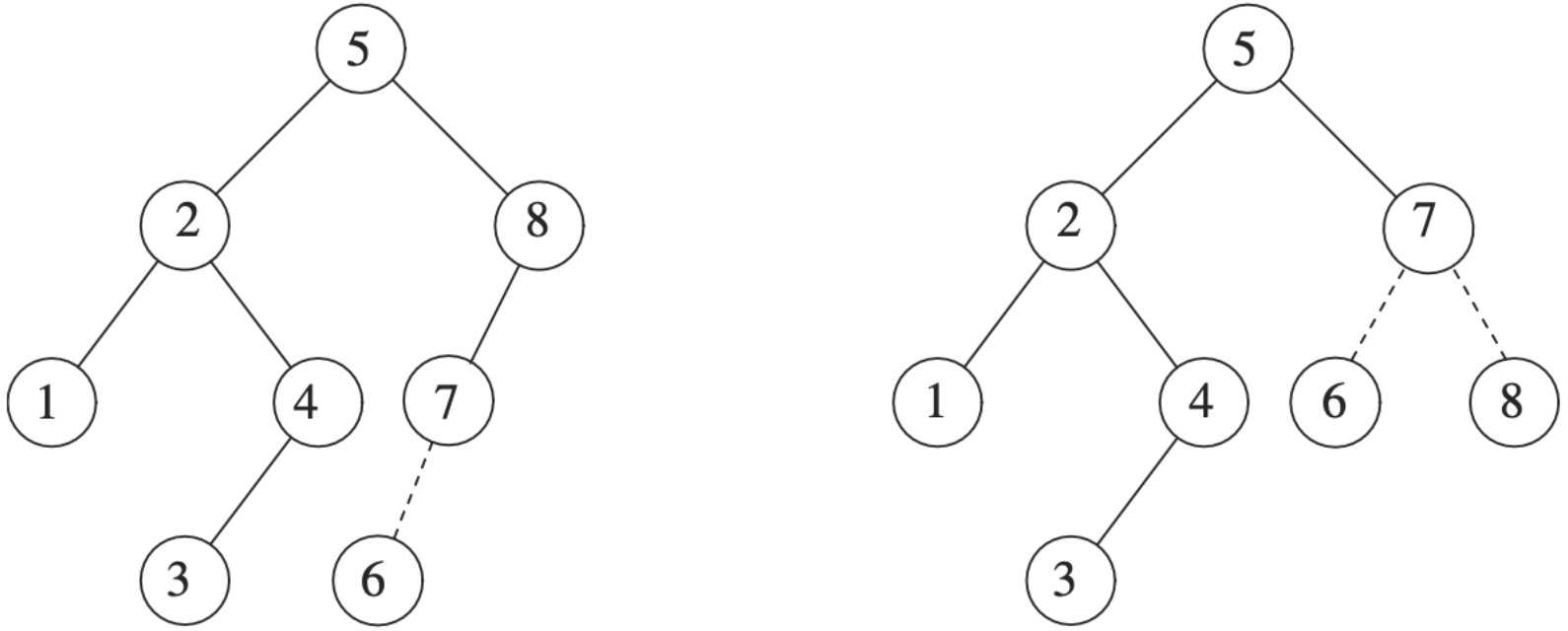


Figure 4.35 AVL property destroyed by insertion of 6, then fixed by a single rotation

Single Rotation

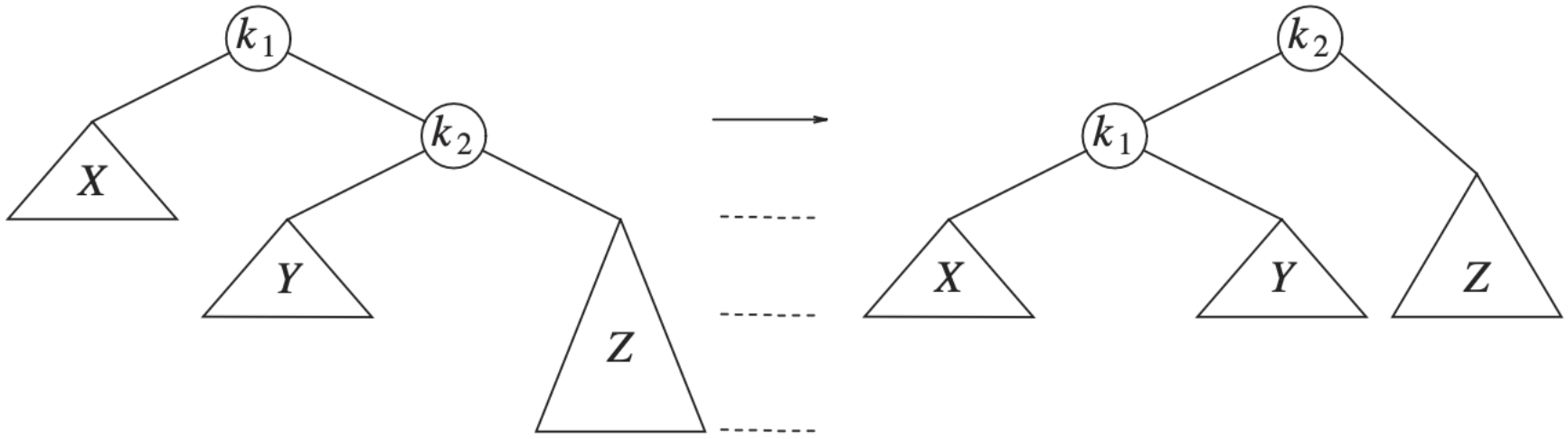
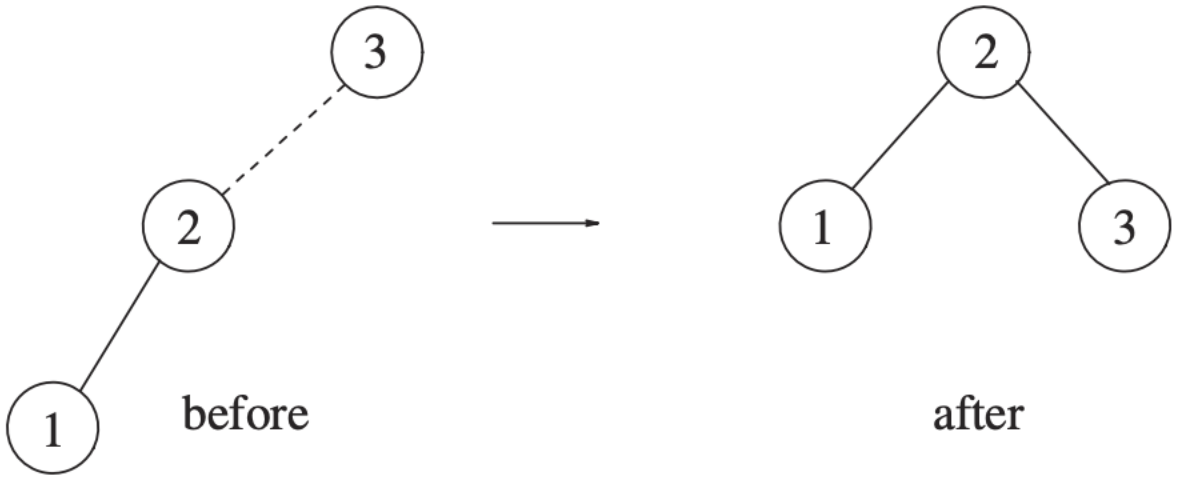
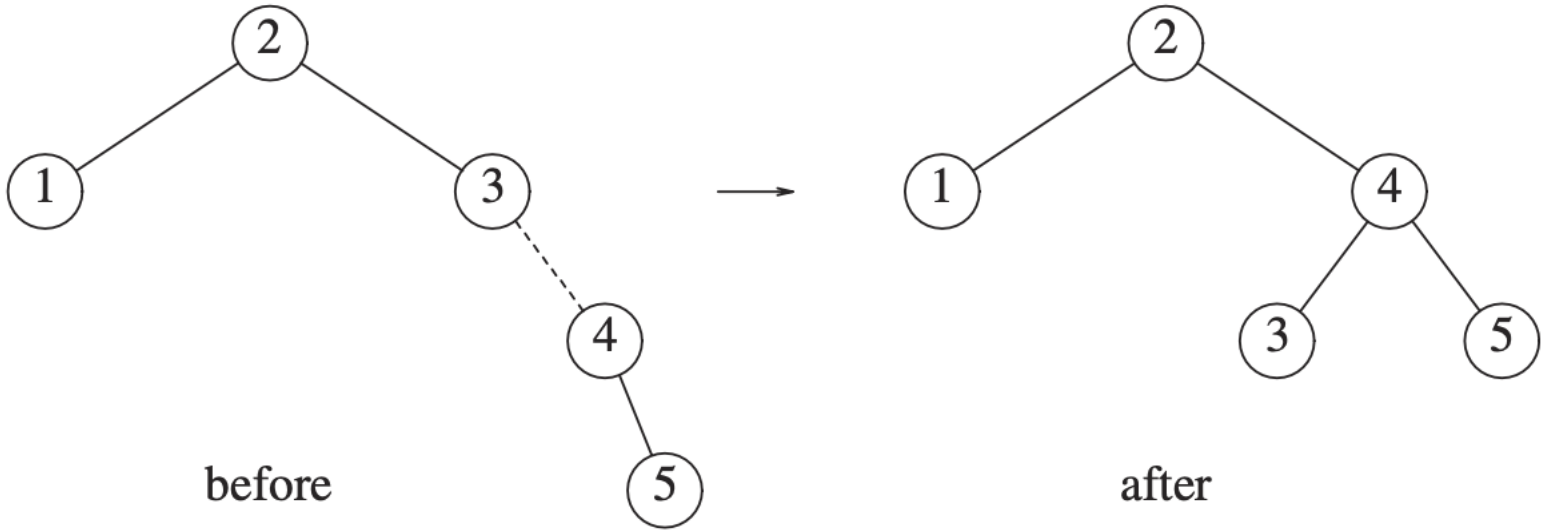


Figure 4.36 Single rotation fixes case 4

Single Rotation

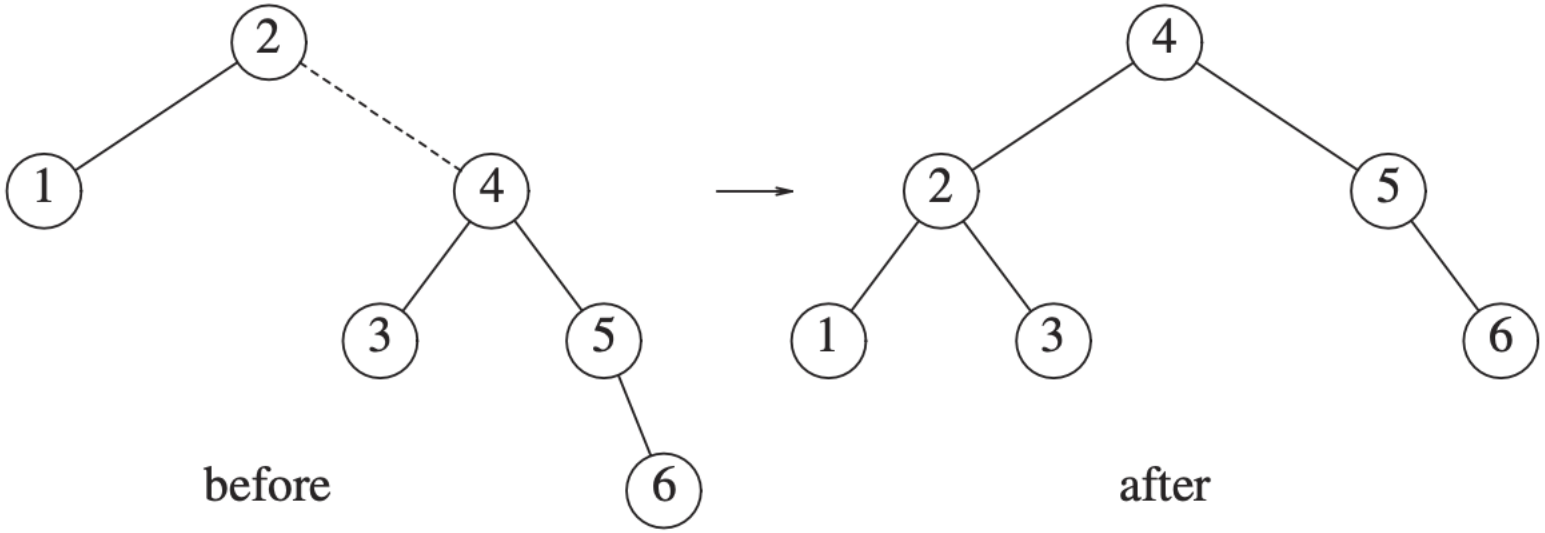


Single Rotation



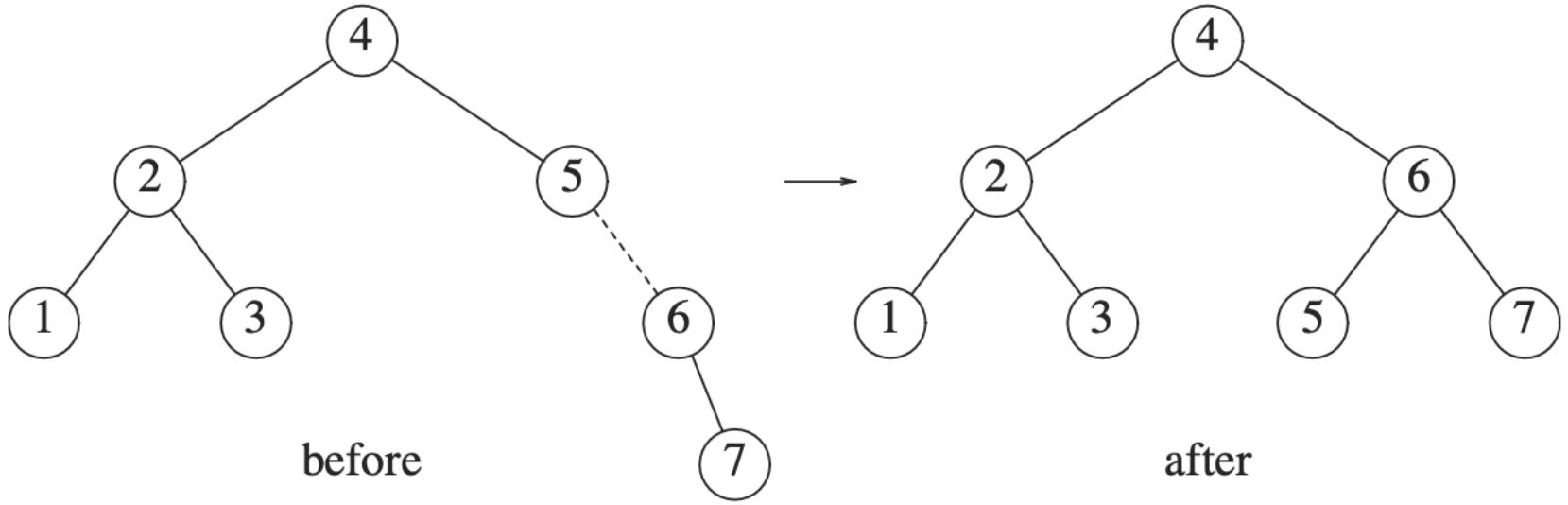
- A dashed line joins the two nodes that are the subject of the rotation.
- Next we insert 4, which causes no problems, but the insertion of 5 creates a violation at node 3 that is fixed by a single rotation.
- Besides the local change caused by the rotation, the programmer must remember that the rest of the tree has to be informed of this change.
- Here this means that 2's right child must be reset to link to 4 instead of 3.
- Forgetting to do so is easy and would destroy the tree (4 would be inaccessible).

Single Rotation



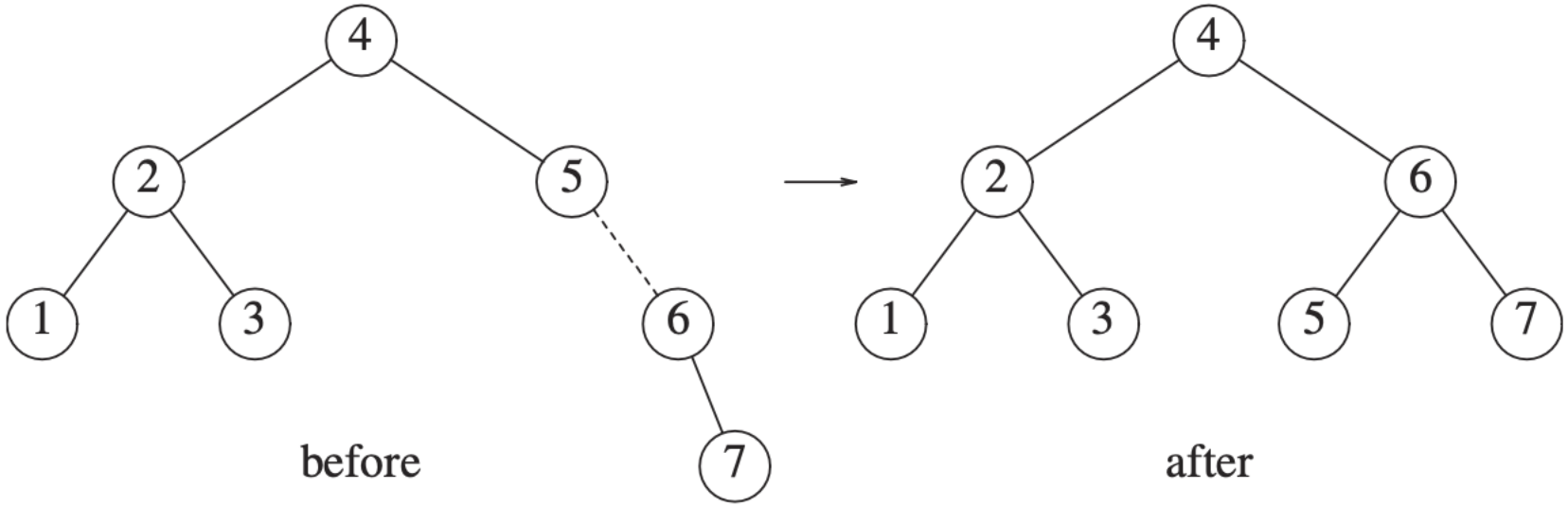
- The rotation is performed by making 2 a child of 4 and 4's original left subtree the new right subtree of 2.
- Every item in this subtree must lie between 2 and 4, so this transformation makes sense.
- The next item we insert is 7, which causes another rotation:

Single Rotation



- Next we insert 6.
- This causes a balance problem at the root, since its left subtree is of height 0 and its right subtree would be height 2.
- Therefore, we perform a single rotation at the root between 2 and 4.

Single Rotation



Double Rotation

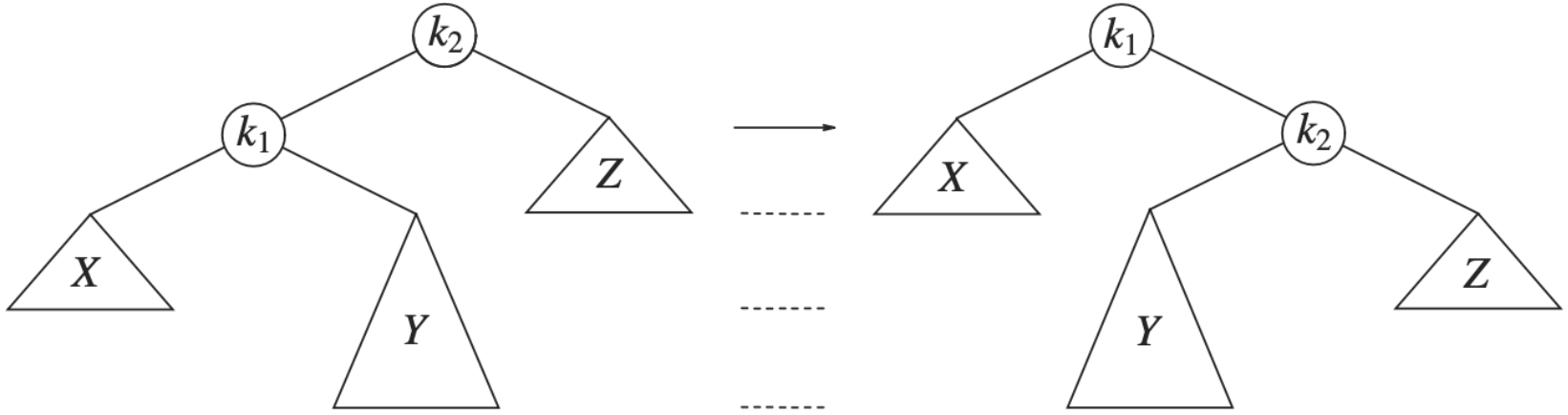


Figure 4.37 Single rotation fails to fix case 2

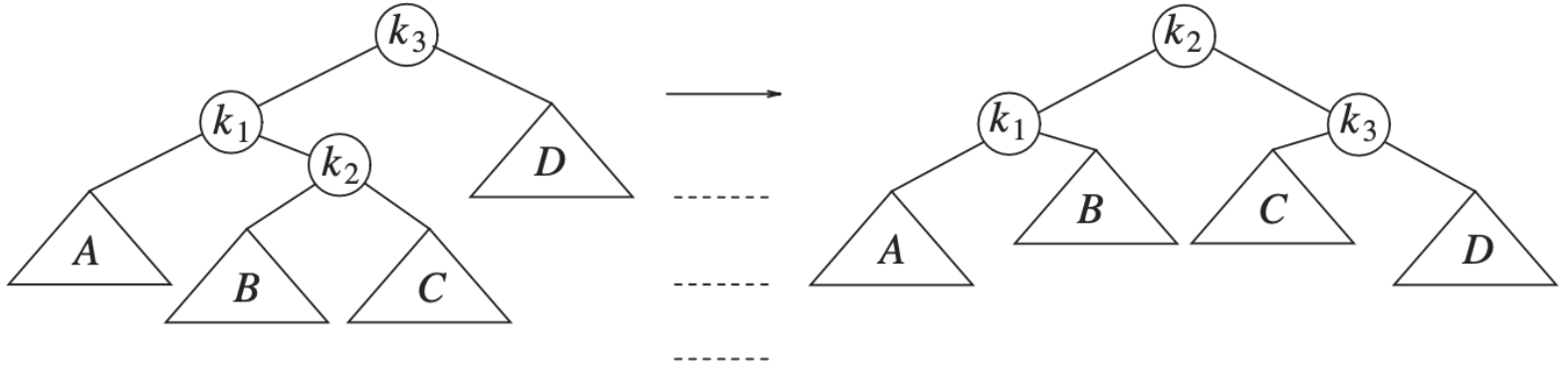
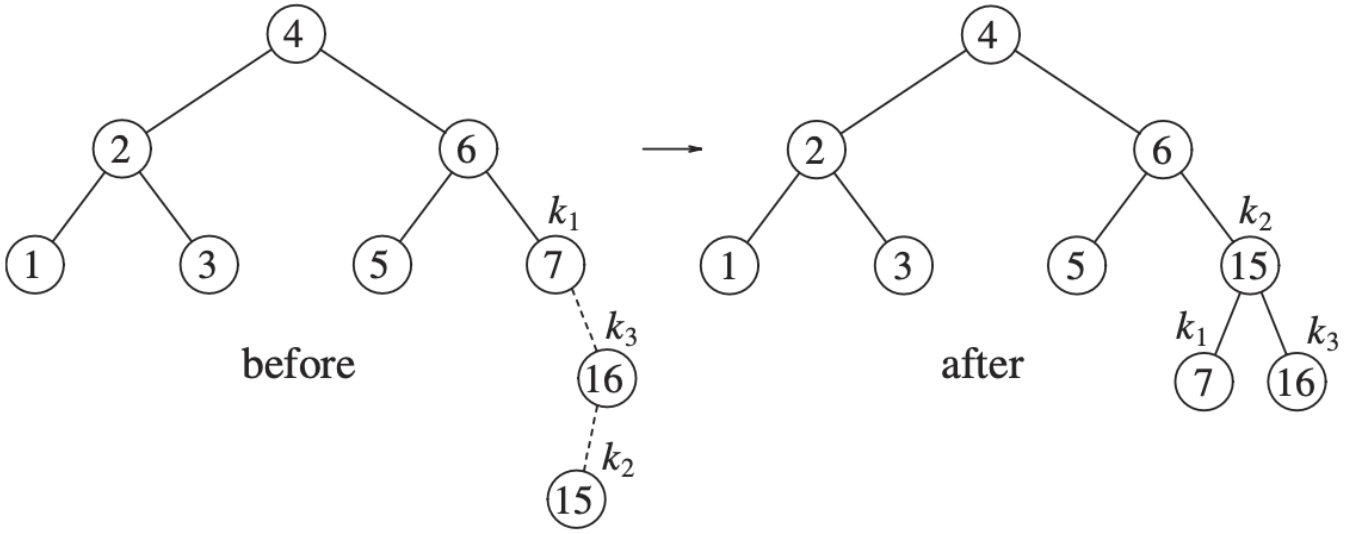


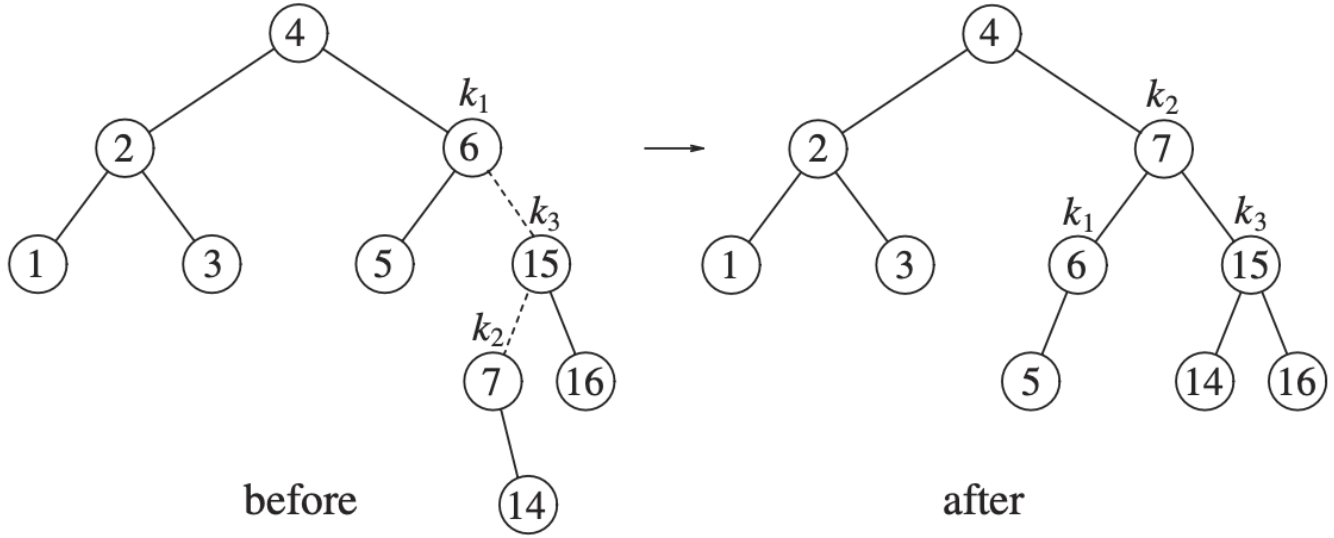
Figure 4.38 Left–right double rotation to fix case 2

Double Rotation



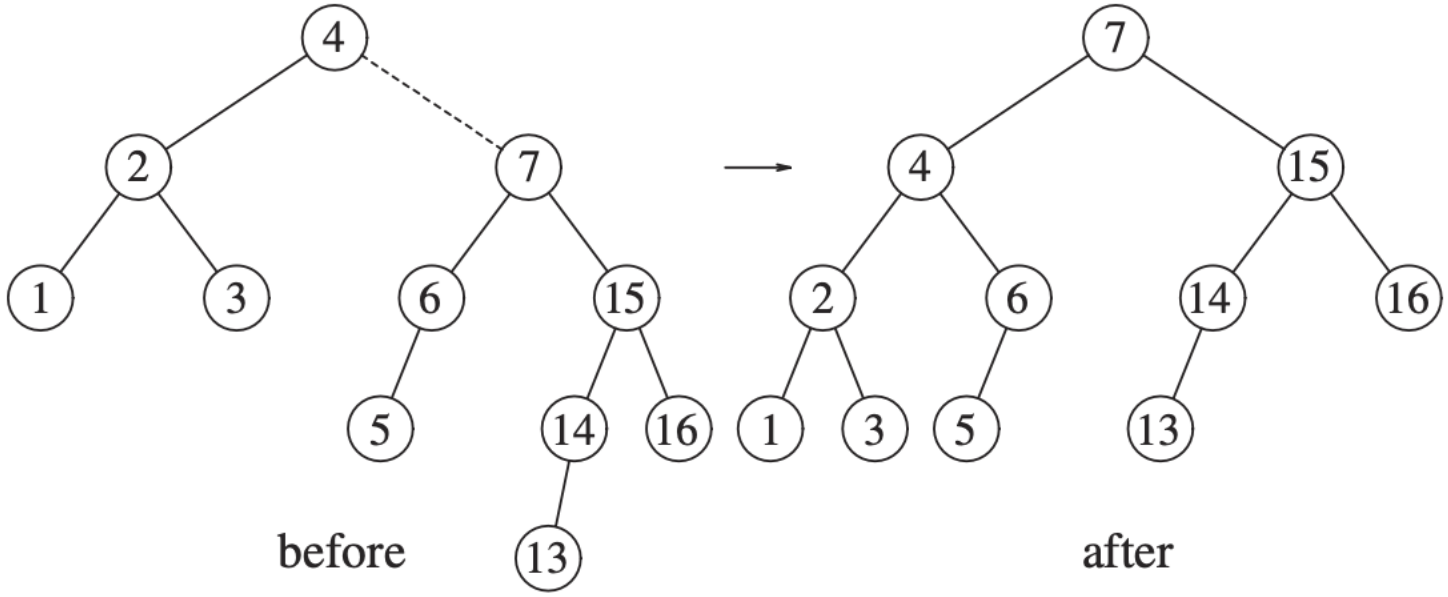
- inserting 10 through 16 in reverse order, followed by 8 and then 9.
- Inserting 16 is easy, since it does not destroy the balance property, but inserting 15 causes a height imbalance at node 7.
- This is case 3, which is solved by a right–left double rotation.
- In our example, the right–left double rotation will involve 7, 16, and 15. In this case, k_1 is the node with item 7, k_3 is the node with item 16, and k_2 is the node with item 15.
- Subtrees *A, B, C,* and *D* are empty.

Double Rotation



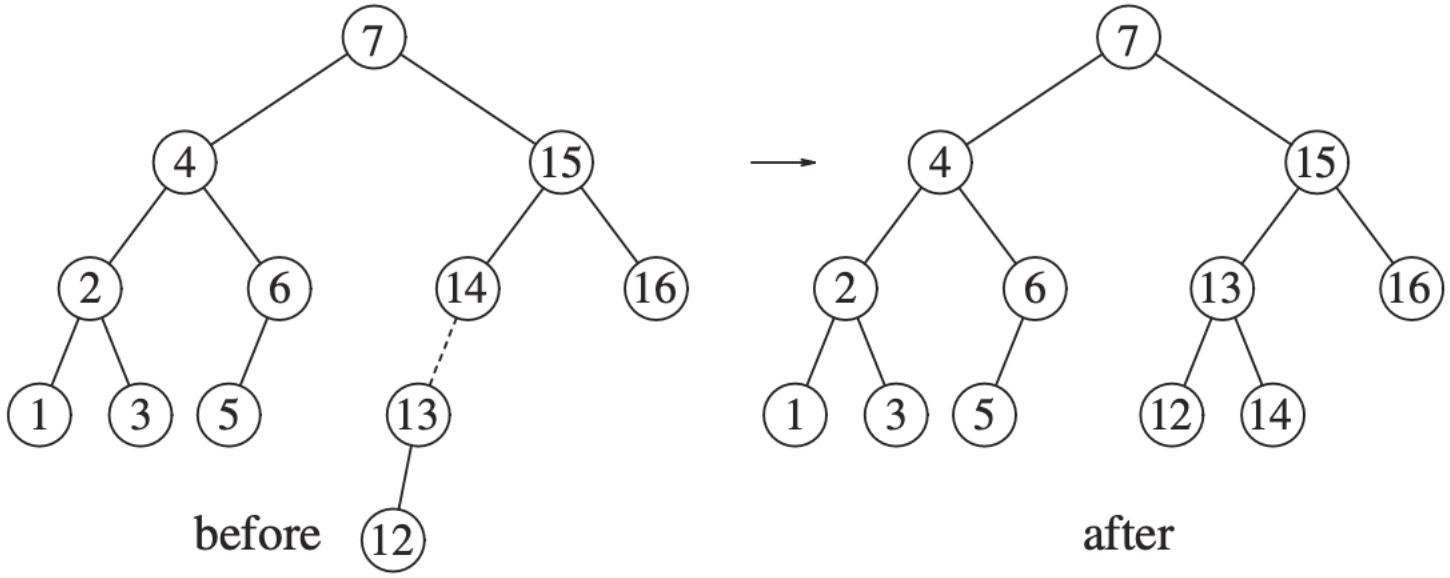
- Next we insert 14, which also requires a double rotation.
- Here the double rotation that will restore the tree is again a right–left double rotation that will involve 6, 15, and 7.
- In this case, *k*1 is the node with item 6, *k*2 is the node with item 7, and *k*3 is the node with item 15.
- Subtree *A* is the tree rooted at the node with item 5; subtree *B* is the empty subtree that was originally the left child of the node with item 7, subtree *C* is the tree rooted at the node with item 14, and finally, subtree *D* is the tree rooted at the node with item 16.

Double Rotation



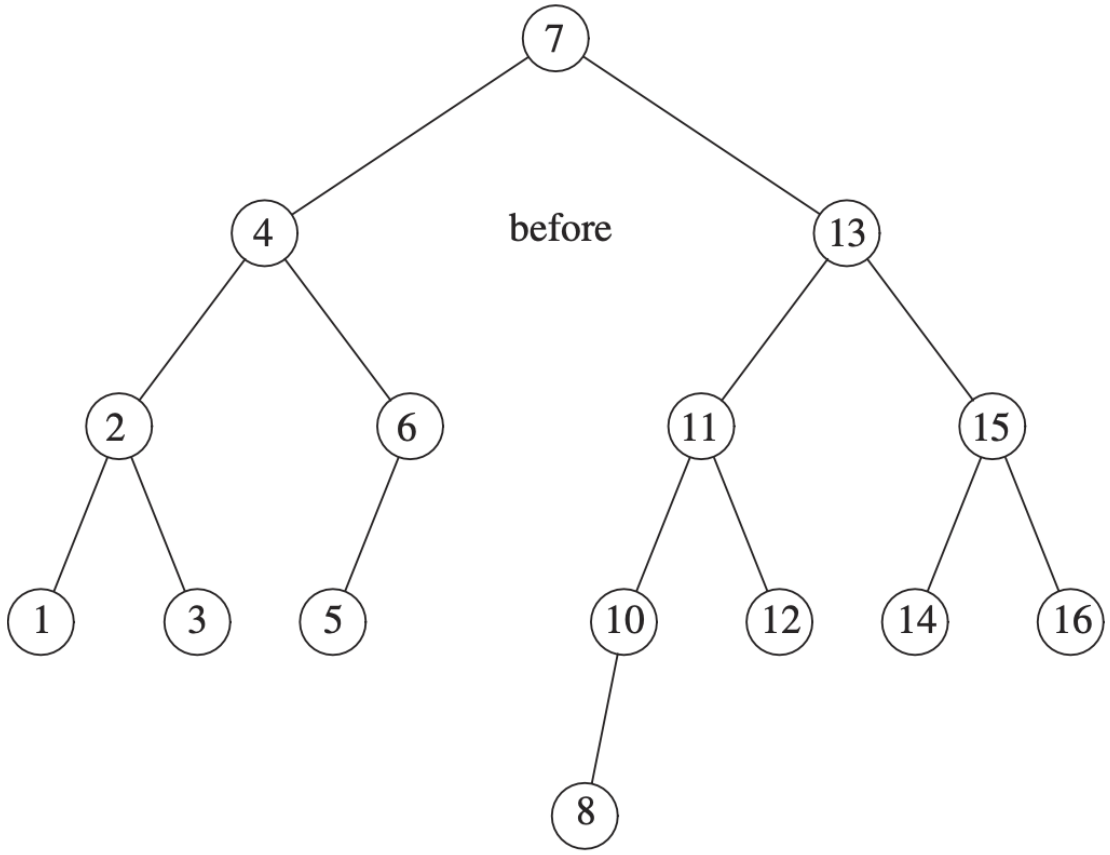
- If 13 is now inserted, there is an imbalance at the root.
- Since 13 is not between 4 and 7, we know that the single rotation will work.

Double Rotation



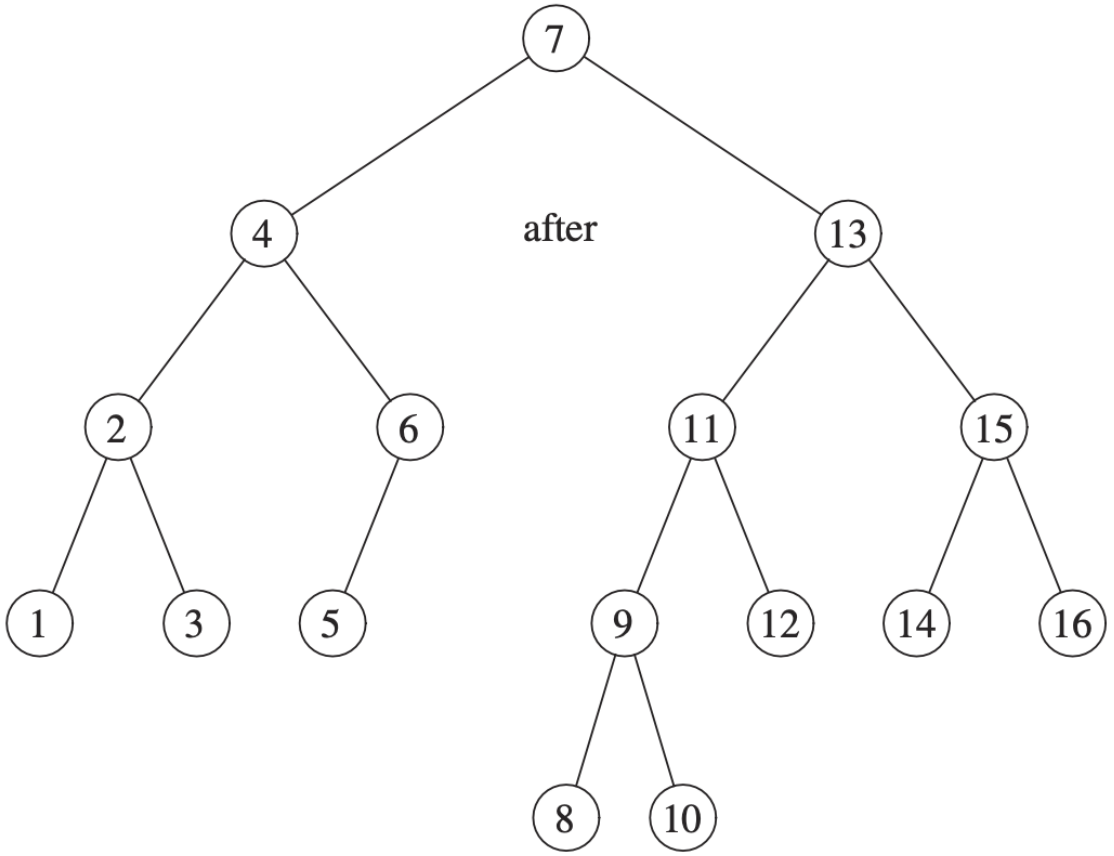
- Insertion of 12 will also require a single rotation:

Double Rotation



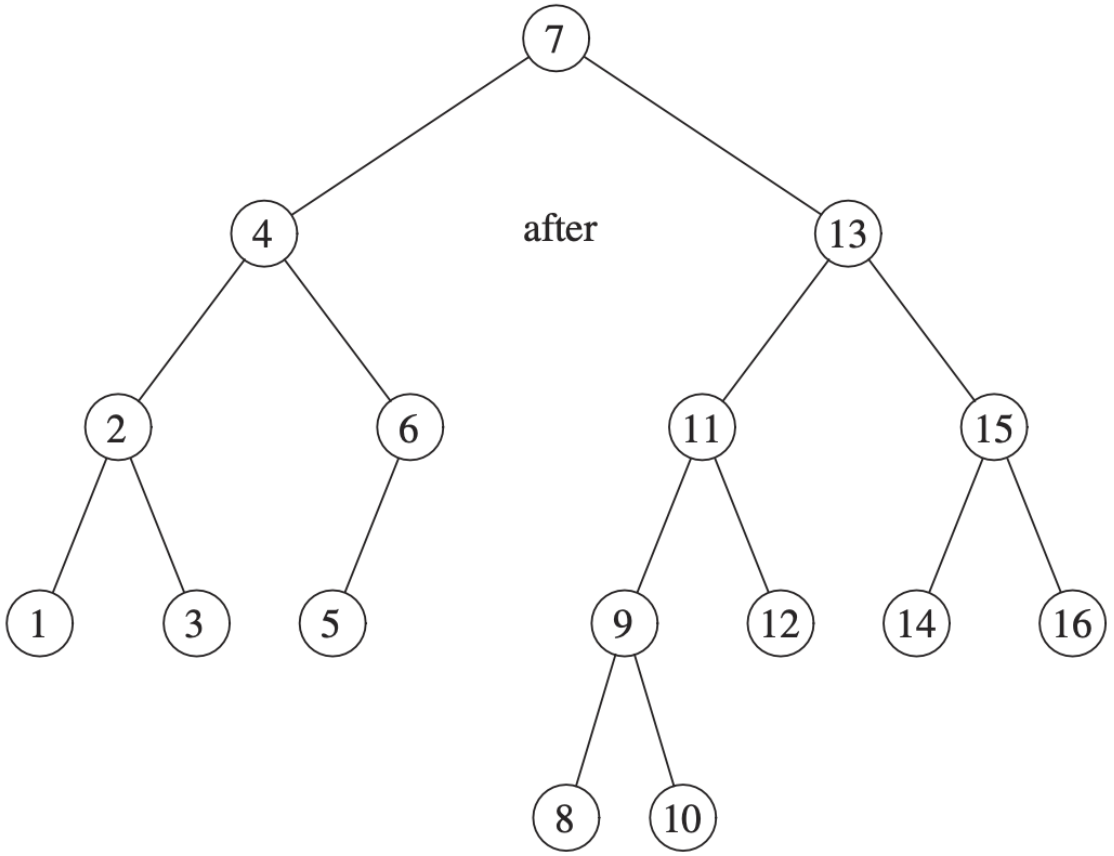
- To insert 11, a single rotation needs to be performed, and the same is true for the subsequent insertion of 10.
- We insert 8 without a rotation, creating an almost perfectly balanced tree:

Double Rotation



- Finally, we will insert 9 to show the symmetric case of the double rotation.
- Notice that 9 causes the node containing 10 to become unbalanced.
- Since 9 is between 10 and 8 (which is 10's child on the path to 9), a double rotation needs to be performed, yielding the following tree:

Double Rotation



- Finally, we will insert 9 to show the symmetric case of the double rotation.
- Notice that 9 causes the node containing 10 to become unbalanced.
- Since 9 is between 10 and 8 (which is 10's child on the path to 9), a double rotation needs to be performed, yielding the following tree:

height

```
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

Insert&Balance

```
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  }
18
```

```
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

Rotate

```
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

Rotate

```
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

Deletion

- Since deletion in a binary search tree is somewhat more complicated than insertion, one can assume that deletion in an AVL tree is also more complicated.
- In a perfect world, one would hope that the deletion of BST could easily be modified by changing the last line to return after calling the balance method, as was done for insertion.

Deletion

```
1  /**
2   * Internal method to remove from a subtree.
3   * x is the item to remove.
4   * t is the node that roots the subtree.
5   * Set the new root of the subtree.
6   */
7  void remove( const Comparable & x, BinaryNode * & t )
8  {
9      if( t == nullptr )
10         return;   // Item not found; do nothing
11     if( x < t->element )
12         remove( x, t->left );
13     else if( t->element < x )
14         remove( x, t->right );
15     else if( t->left != nullptr && t->right != nullptr ) // Two children
16     {
17         t->element = findMin( t->right )->element;
18         remove( t->element, t->right );
19     }
20     else
21     {
22         BinaryNode *oldNode = t;
23         t = ( t->left != nullptr ) ? t->left : t->right;
24         delete oldNode;
25     }
26 }
```

Figure 4.26 Deletion routine for binary search trees

```
1  /**
2   * Internal method to remove from a subtree.
3   * x is the item to remove.
4   * t is the node that roots the subtree.
5   * Set the new root of the subtree.
6   */
7  void remove( const Comparable & x, AvlNode * & t )
8  {
9      if( t == nullptr )
10         return;   // Item not found; do nothing
11
12     if( x < t->element )
13         remove( x, t->left );
14     else if( t->element < x )
15         remove( x, t->right );
16     else if( t->left != nullptr && t->right != nullptr )  // Two children
17     {
18         t->element = findMin( t->right  )->element;
19         remove( t->element, t->right );
20     }
21     else
22     {
23         AvlNode *oldNode = t;
24         t = ( t->left != nullptr ) ? t->left : t->right;
25         delete oldNode;
26     }
27
28      balance( t );
29 }
```

Figure 4.47 Deletion in an AVL tree

Reference

Allen, W. M. (2007). *Data structures and algorithm analysis in C++*. Pearson Education India.

Nell B. Dale. (2003). *C++ plus data structures*. Jones & Bartlett Learning.

เฉียบวุฒิ รัตนวิลัยสกุล. (2023). โครงสร้างข้อมูล. มหาวิทยาลัยเทคโนโลยีพระจอมเกล้าพระนครเหนือ

https://ece.uwaterloo.ca/~dwharder/aads/Lecture_materials/