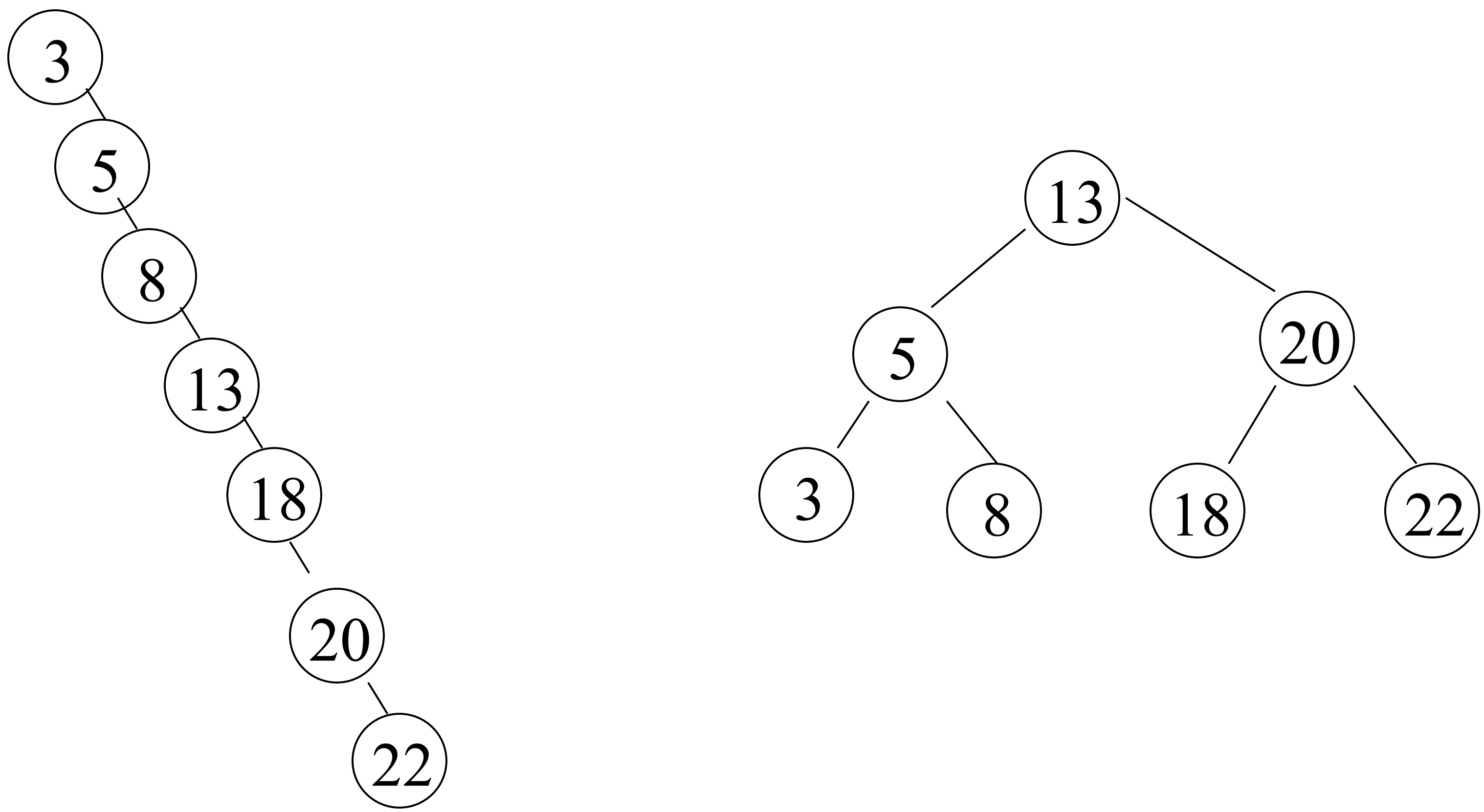
AVL trees

Motivation

- When building a binary search tree, what type of trees would we like? Example: 3, 5, 8, 20, 18, 13, 22



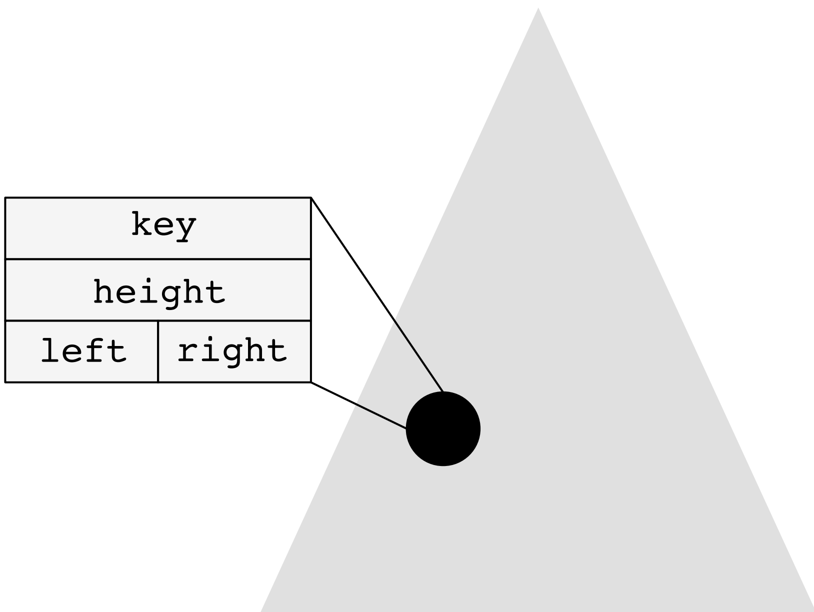
Motivation

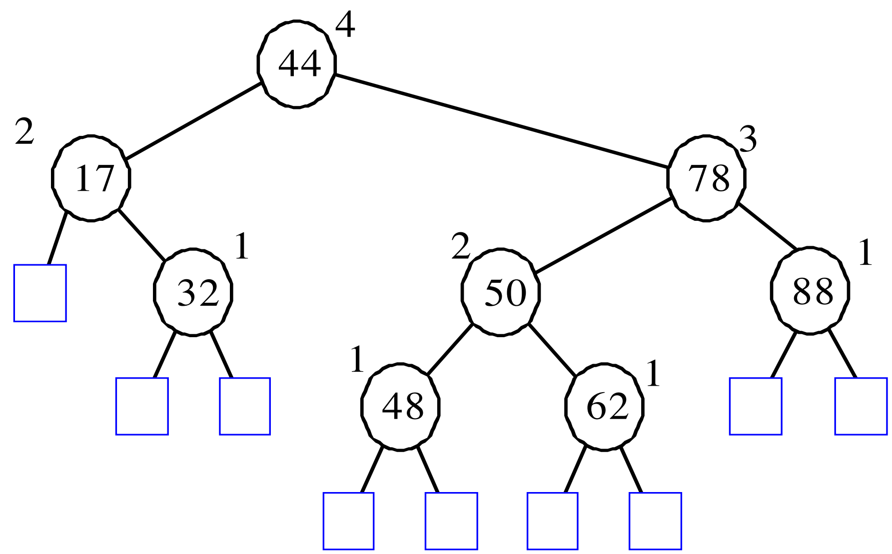
- Complete binary tree is hard to build when we allow dynamic insert and remove.
  - We want a tree that has the following properties
    - Tree height = O(log(N))
    - allows dynamic insert and remove with O(log(N)) time complexity.

The AVL tree is one of this kind of trees.

AVL (Adelson-Velskii and Landis) Trees

An AVL Tree is a ***binary search tree*** such that for every internal node v of T, **the *heights of the children of v can differ by at most 1*.**





An example of an AVL tree where the heights are shown next to the nodes

AVL Trees

- AVL tree is a binary search tree with balance condition
  - To ensure depth of the tree is O(log(N))
  - And consequently, search/insert/remove complexity bound O(log(N))
- Balance condition
  - For **every node** in the tree, height of left and right subtree can differ by at most 1

- The depth of a typical node in an AVL tree is very close to the optimal *log N*.
- Consequently, all searching operations in an AVL tree have logarithmic worst-case bounds.
- An update (insert or remove) in an AVL tree could destroy the balance. It must then be rebalanced before the operation can be considered complete.
- After an insertion, only nodes that are on the path from the insertion point to the root can have their balances altered.

AVL Tree: insert & remove

- Do binary search tree insert and remove
- The balance condition can be violated sometimes
  - Do something to fix it : **rotations**
  - After rotations, the balance of the whole tree is maintained

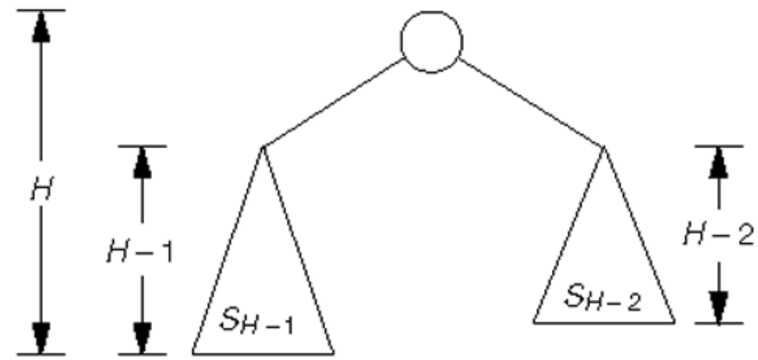
Balance Condition Violation

- If condition violated after a node insertion
  - Which nodes do we need to rotate?

  Only nodes on path from insertion point to root may have their balance altered

- Rebalance the tree through rotation at the **deepest node** with balance violated
  - The entire tree will be rebalanced

balance violation cases



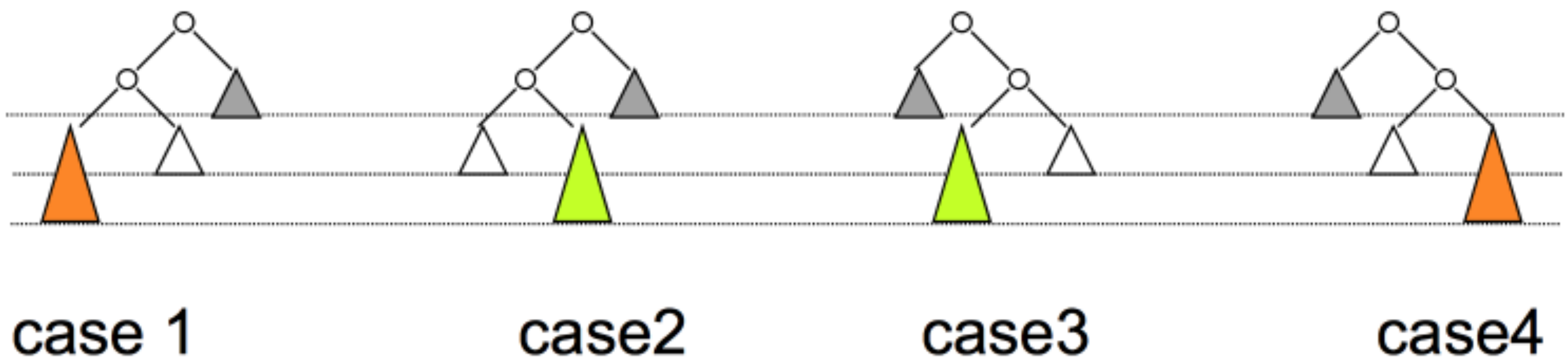
Violation cases at node k (deepest node)

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

– Cases 1 and 4 equivalent (symmetric)
  - Single rotation to rebalance

– Cases 2 and 3 equivalent (symmetric)
  - Double rotation to rebalance

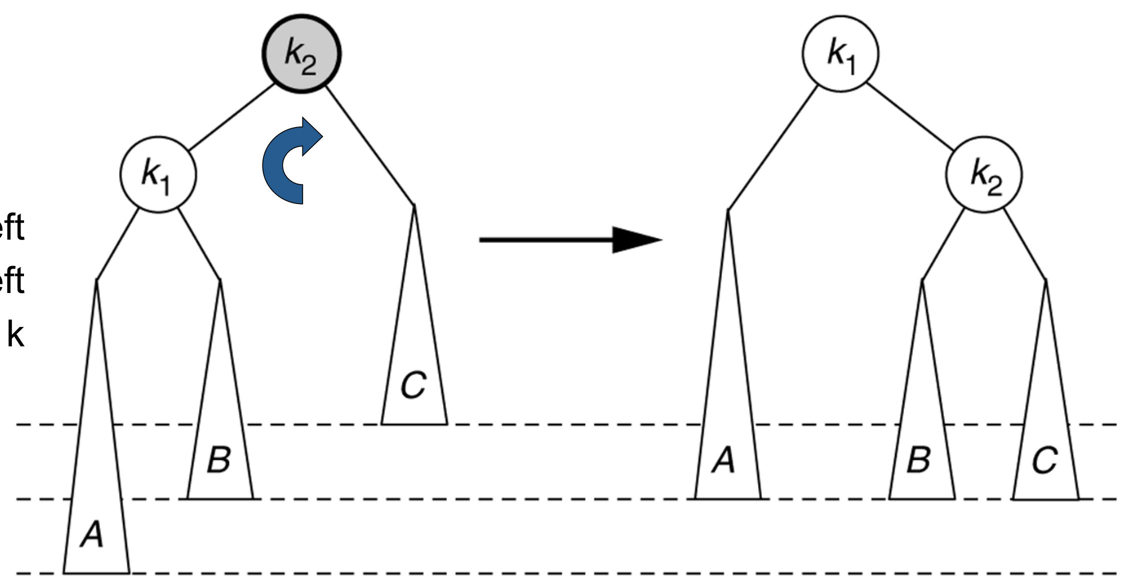
cases



single rotation

- A single rotation switches the roles of the parent and child while maintaining the search order.
- Single rotation handles the “outside” cases (i.e. 1 and 4).
- We rotate between a node and its child.
  - Child becomes parent. Parent becomes right child in case 1, left child in case 4.
- The result is a binary search tree that satisfies the AVL property.

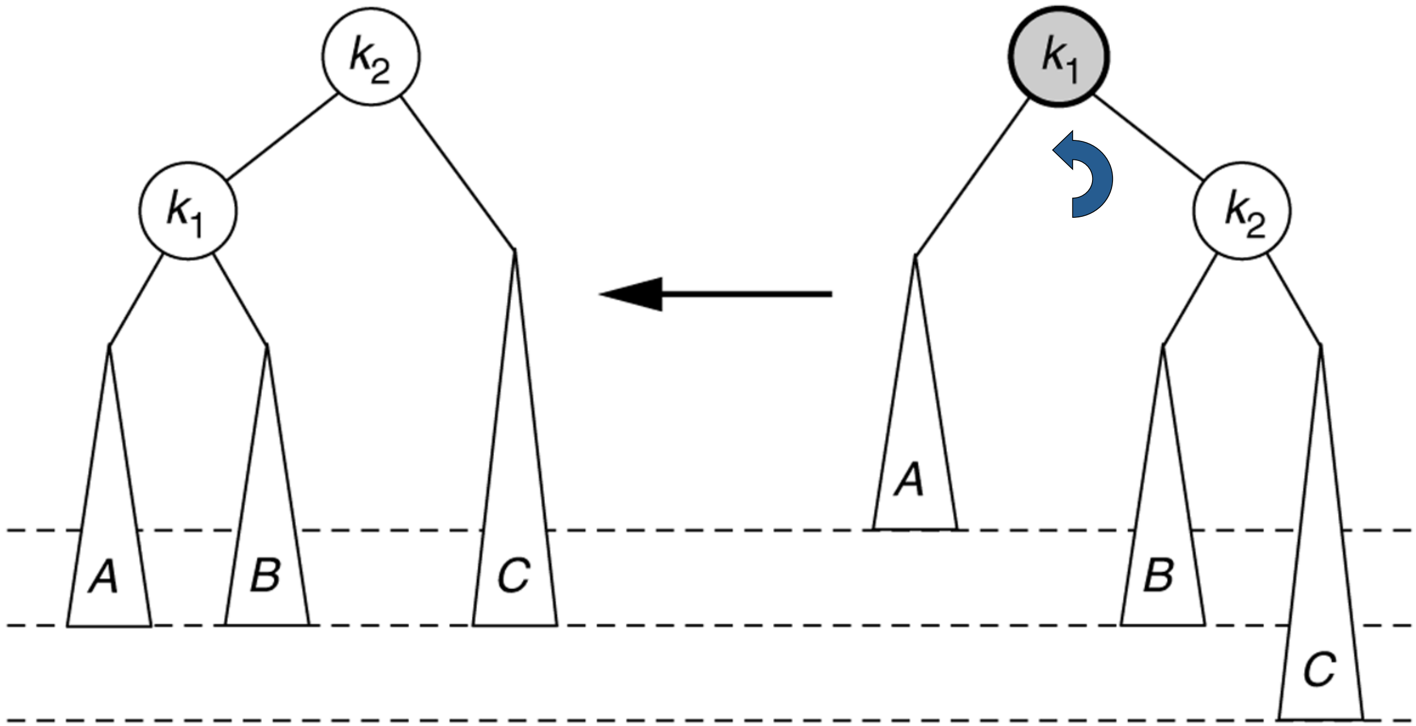
insertion into left subtree of left child of k



(a) Before rotation (b) After rotation

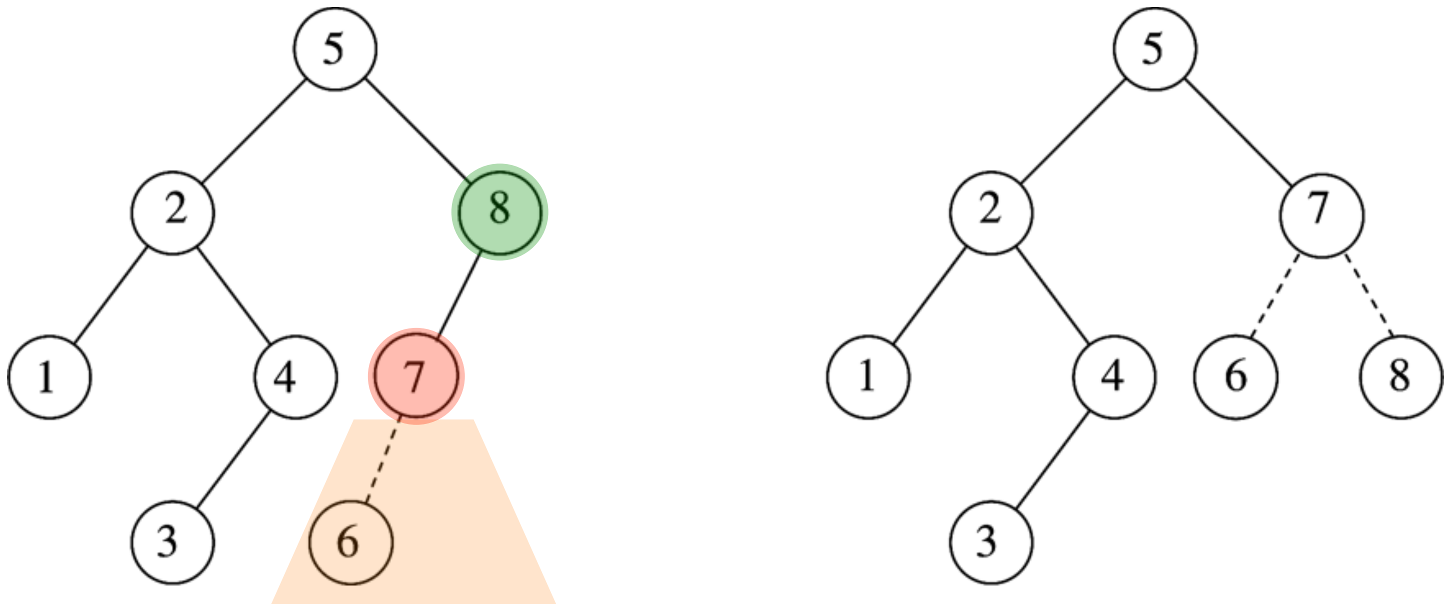
symmetric case: case 4

insertion into right subtree of right child of k



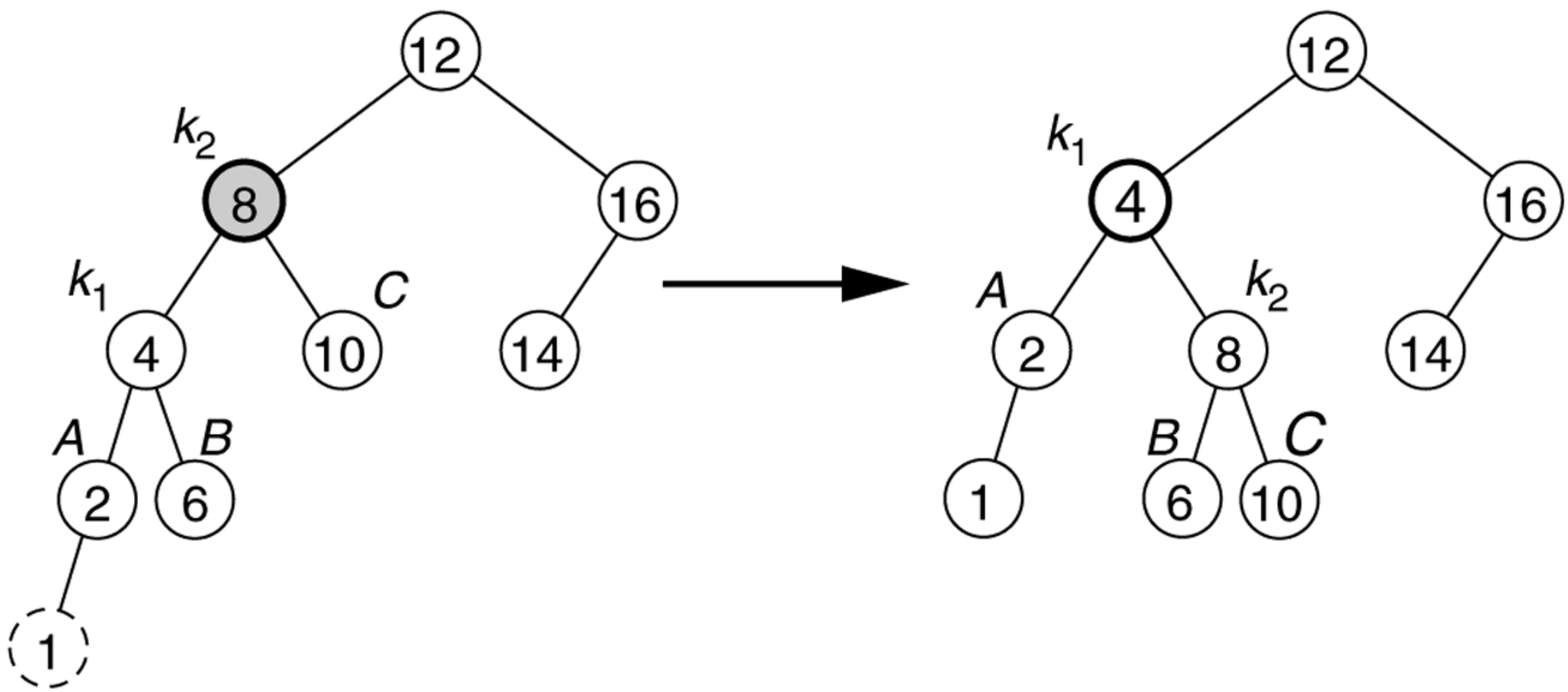
(a) After rotation (b) Before rotation

Example 1



- After inserting 6
  - Balance condition at node 8 is violated

Example 2

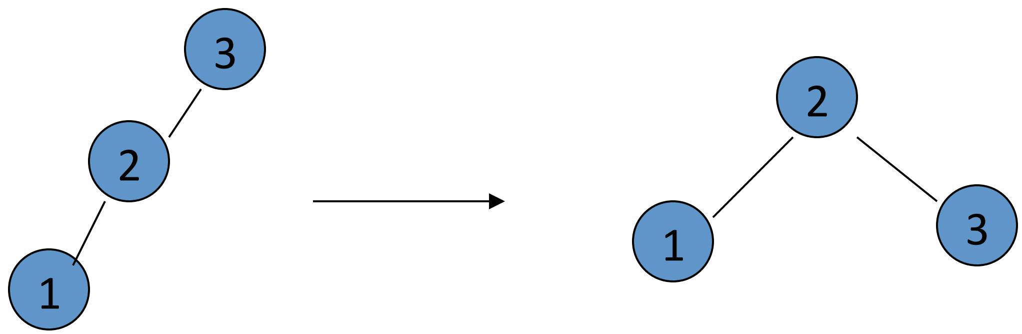


(a) Before rotation

(b) After rotation

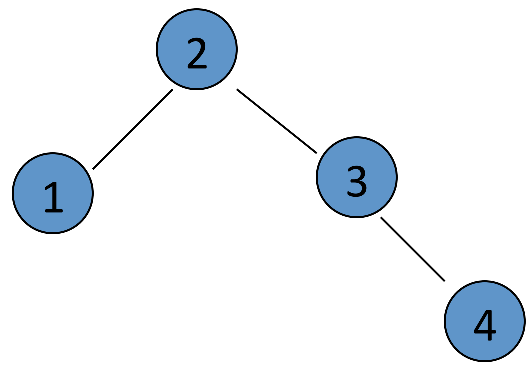
Example 3

- Inserting 3, 2, 1, and then 4 to 7 sequentially into empty AVL tree

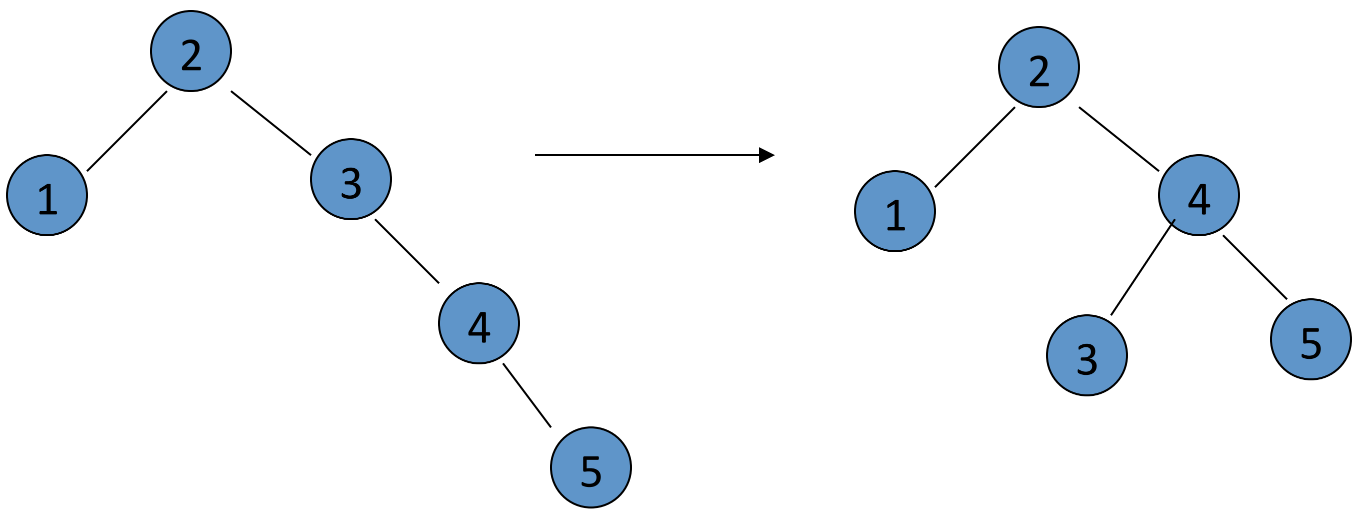


Example 3 (cnt' d)

- Inserting 4

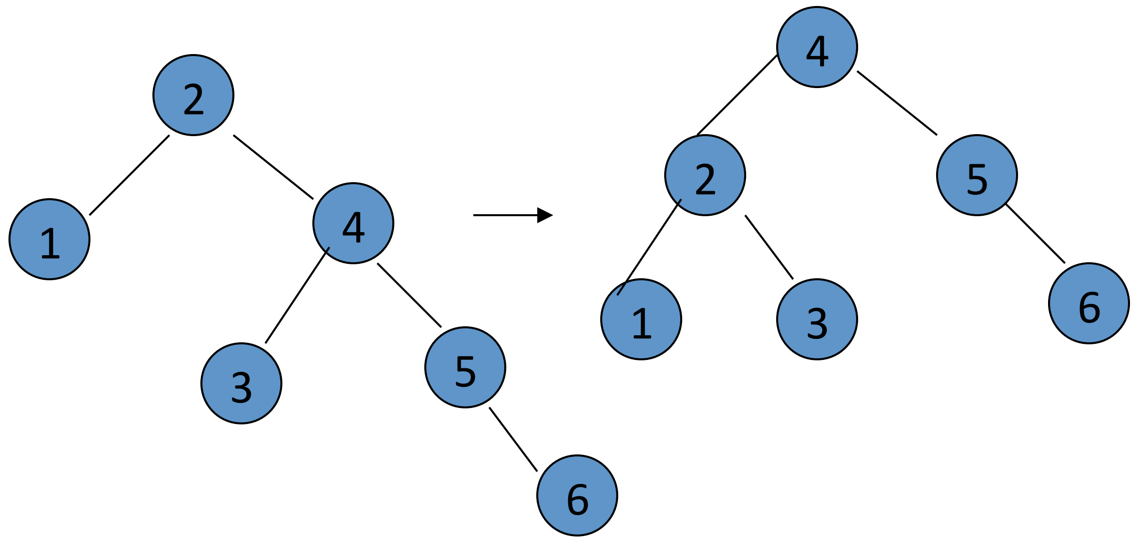


- Inserting 5

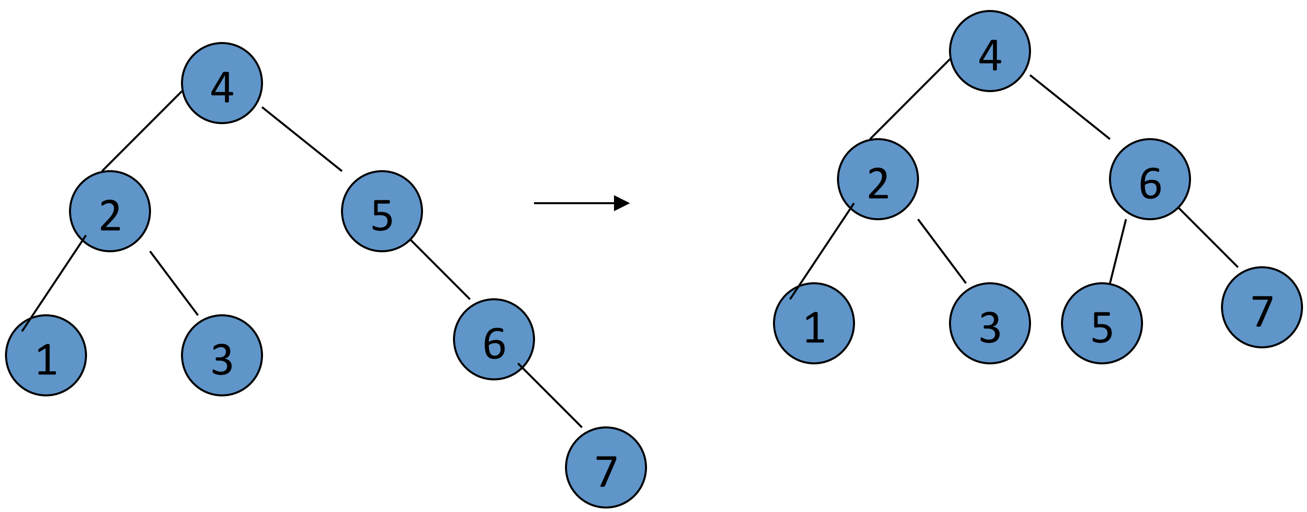


Example 3 (cnt' d)

- Inserting 6



- Inserting 7



Analysis

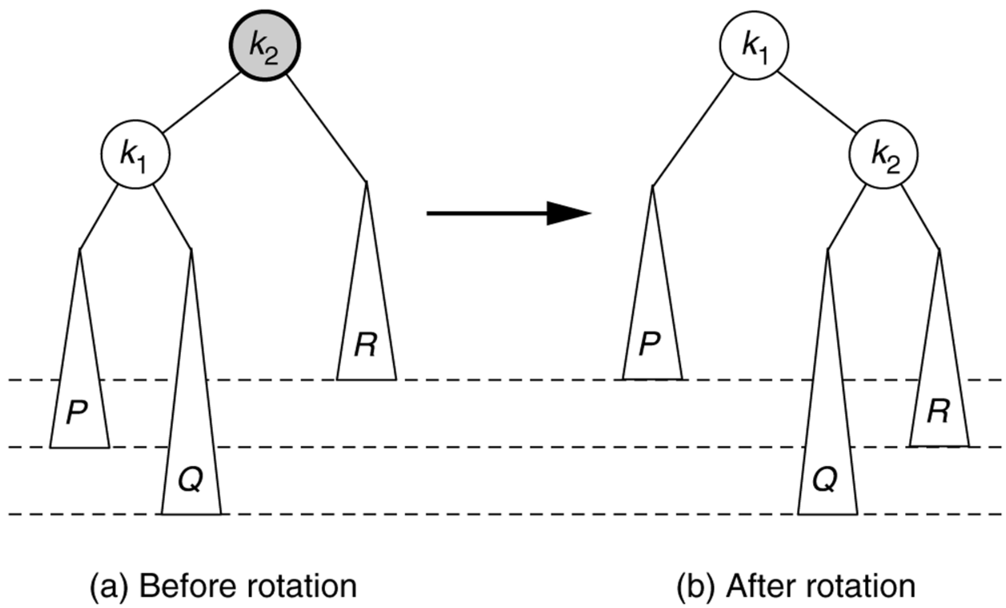
- One rotation suffices to fix cases 1 and 4.
- Single rotation preserves the original height:
  - The new height of the entire subtree is exactly the same as the height of the original subtree before the insertion.
- Therefore it is enough to do rotation only at the first node, where imbalance exists, on the path from inserted node to root.
- The rotation takes O(1) time.
- Hence insertion is O(logN)

Double rotation

- Single rotation does not fix cases 2 and 3:

  case 2: An insertion into right subtree of left child of k

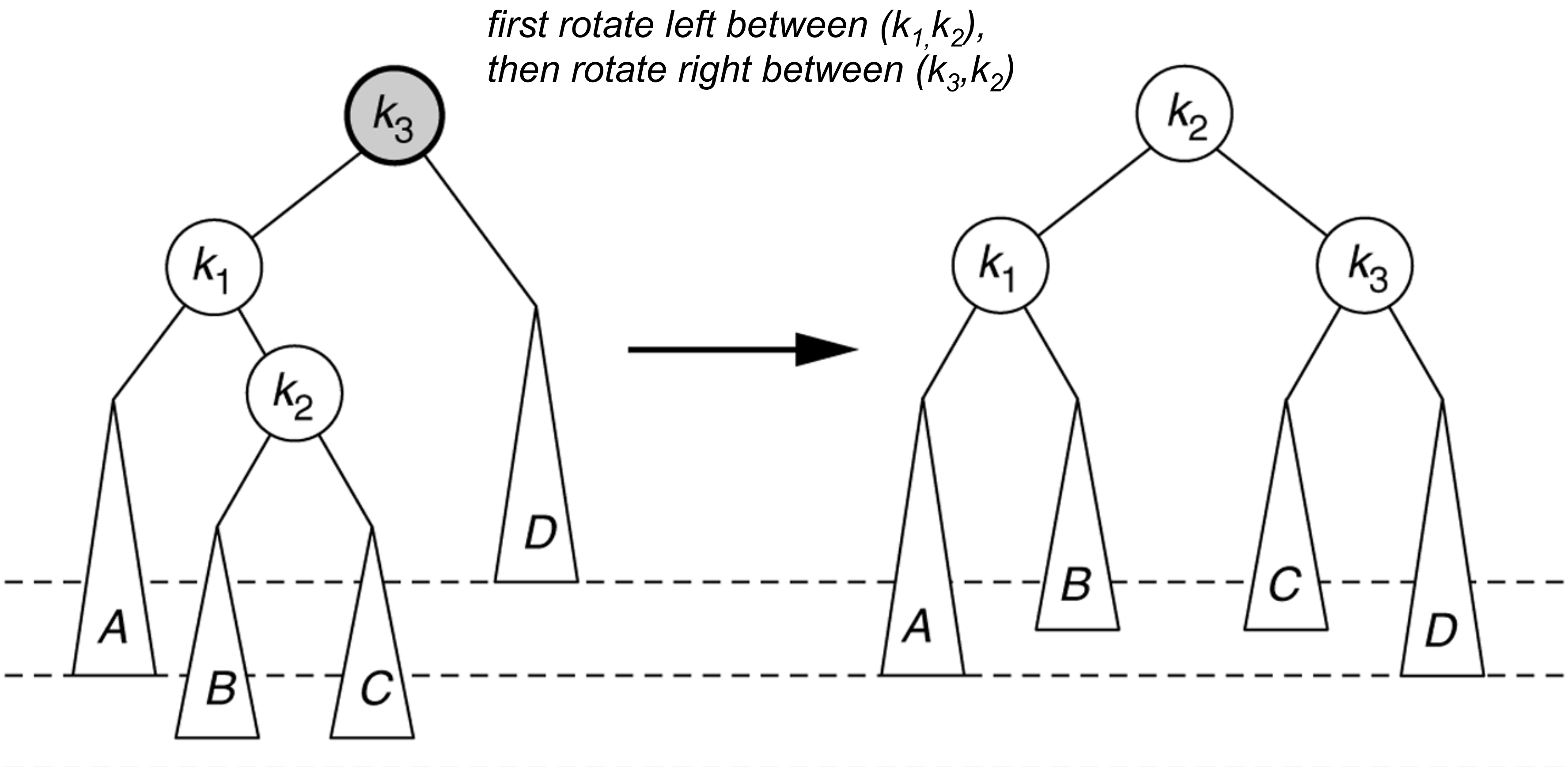
  case 3: An insertion into left subtree of right child of k



- These cases require a ***double* rotation**, involving three nodes and four subtrees.

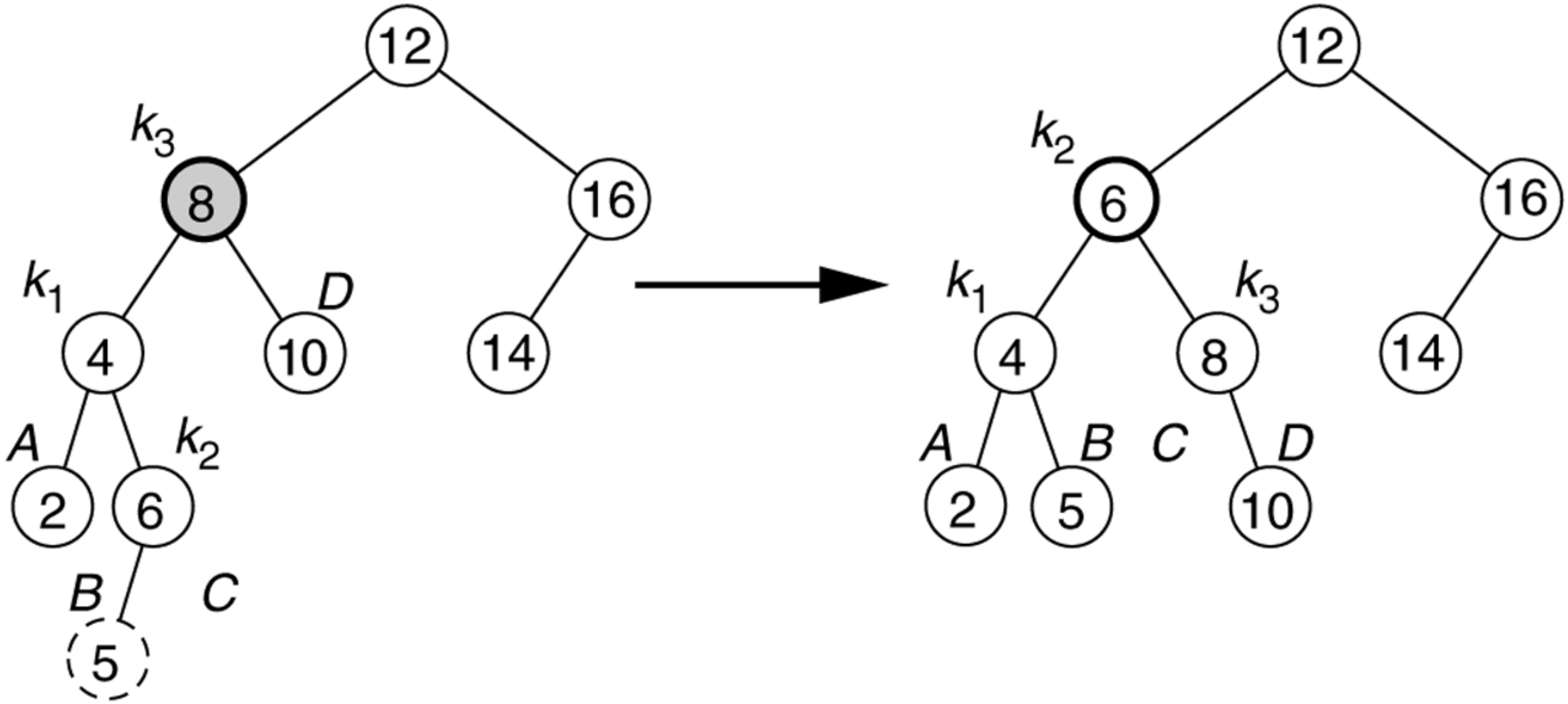
Left–right double rotation to fix case 2

insertion into right subtree of left child of k



(a) Before rotation (b) After rotation

first rotate left between (k_1,k_2),
then rotate right between (k_3,k_2)

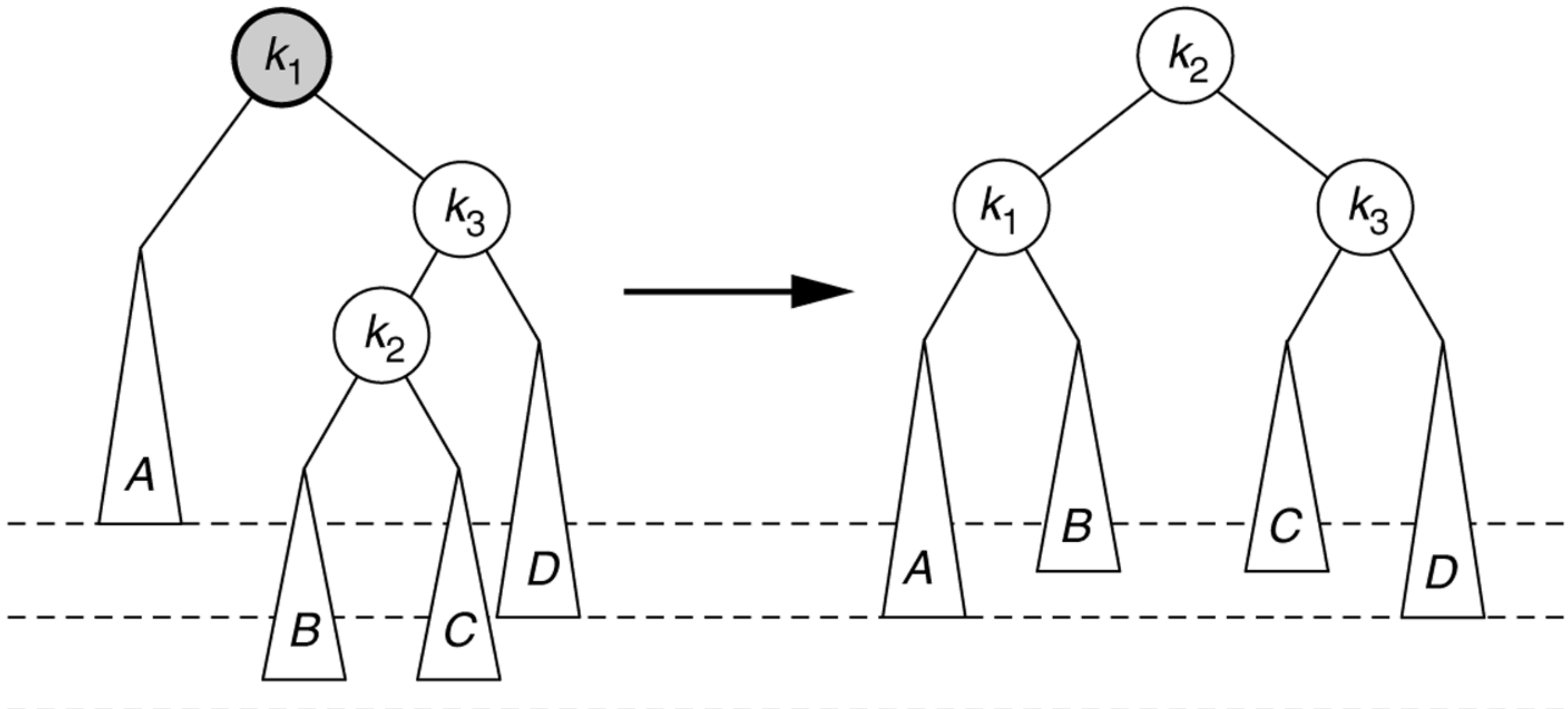


(a) Before rotation

(b) After rotation

Right–Left double rotation to fix case 3

insertion into left subtree of right child of k

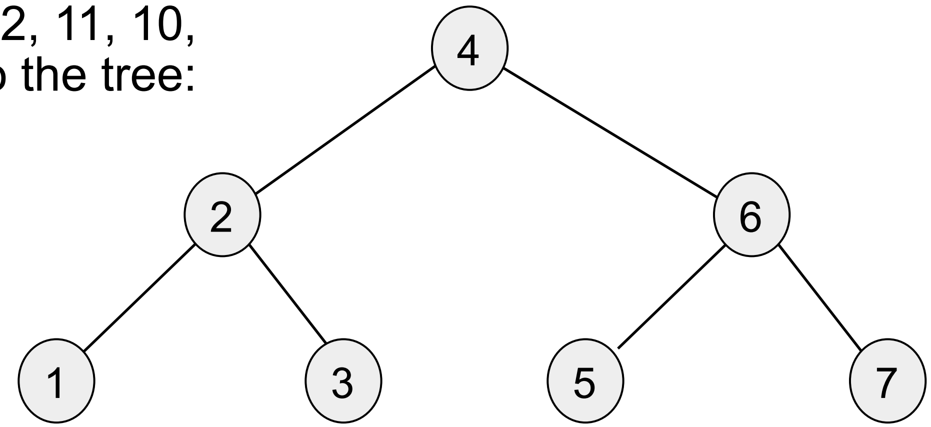


(a) Before rotation

(b) After rotation

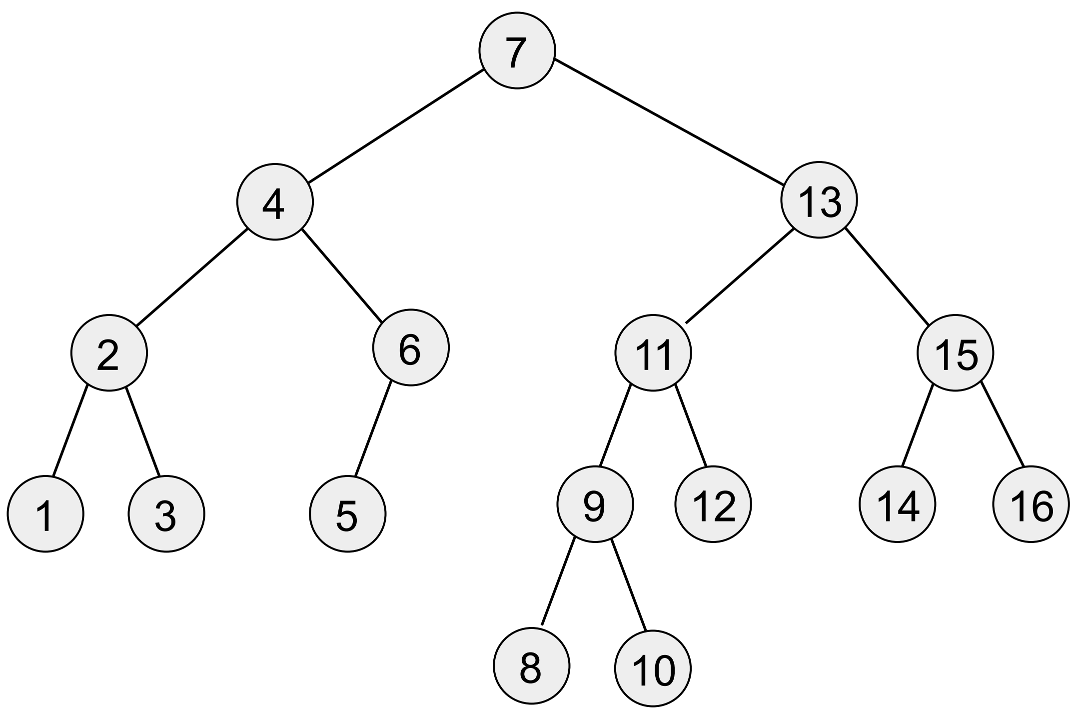
Example

Insert 16, 15, 14, 13, 12, 11, 10, and 8, and 9 to the tree:



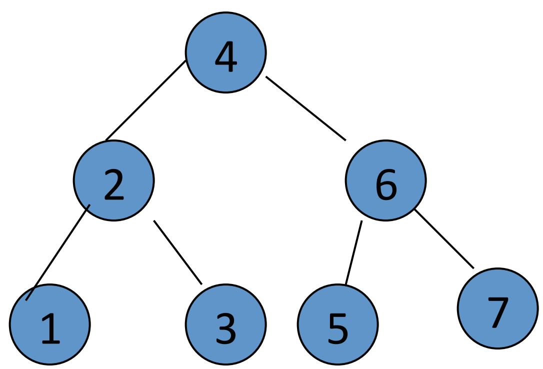
LEFT-RIGHT rotation: insertion into right subtree of left child of k

RIGHT-LEFT rotation: insertion into left subtree of right child of k

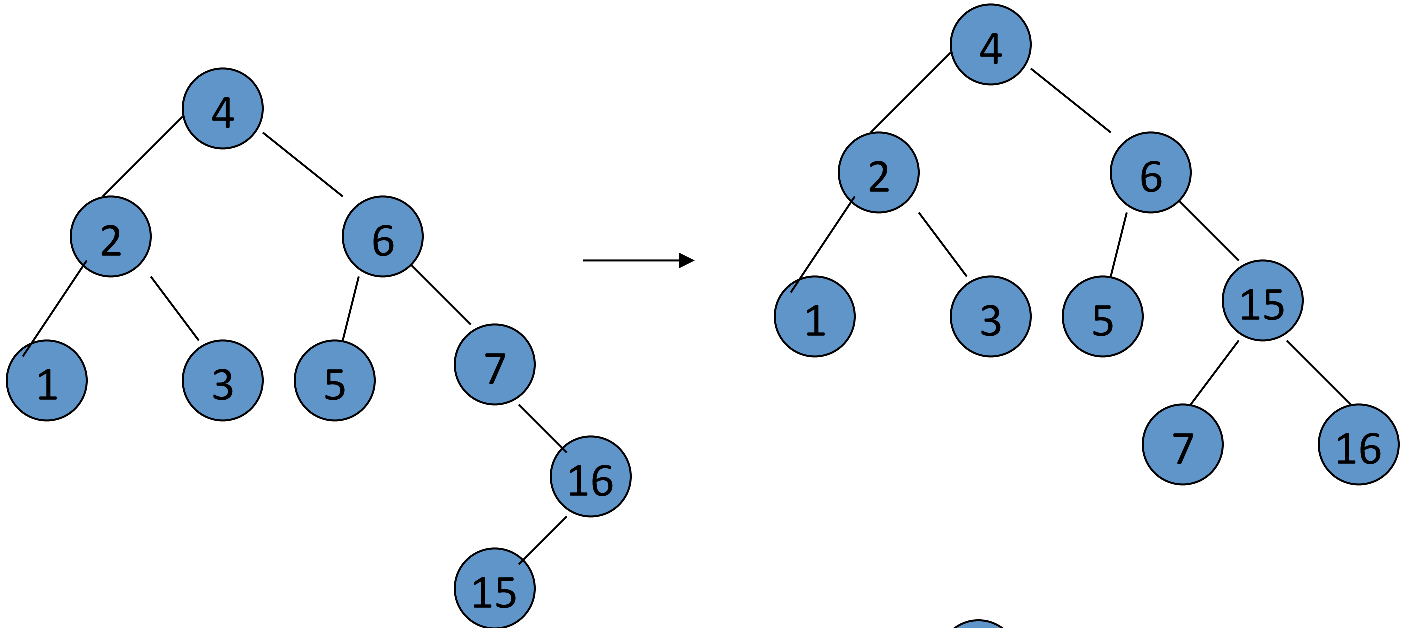


Example

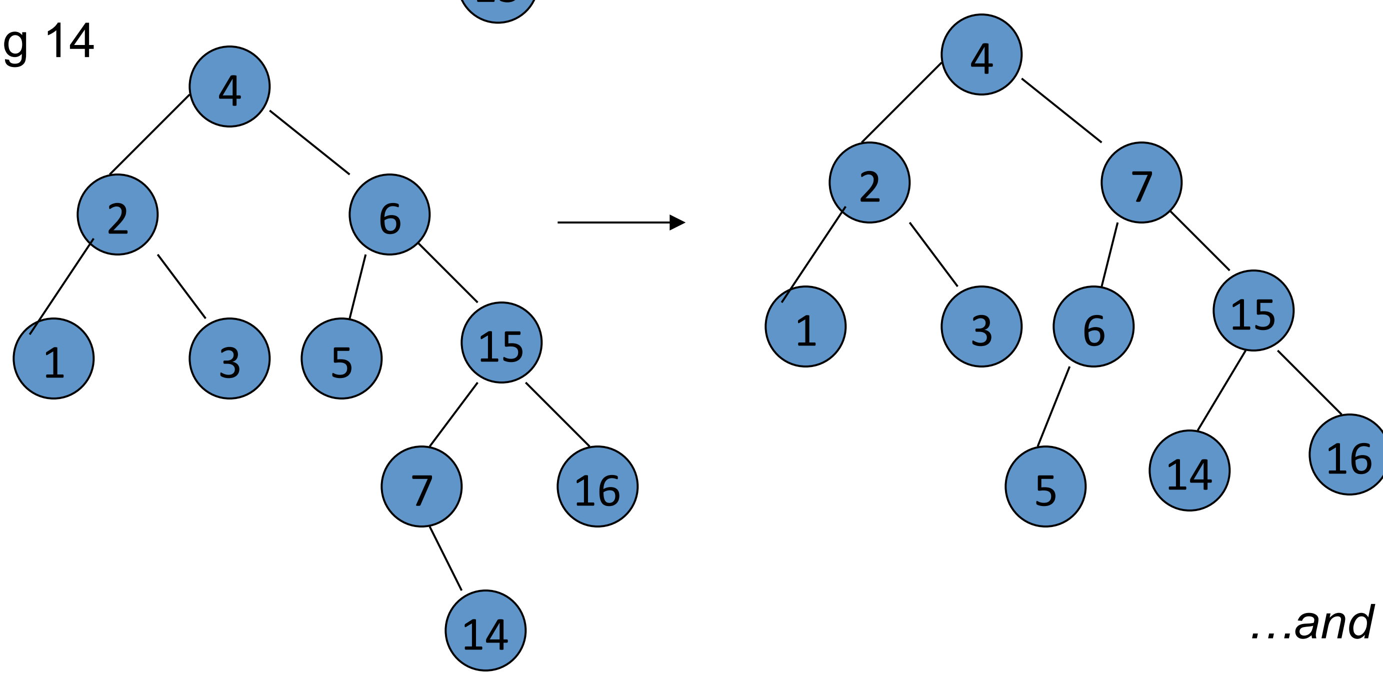
- Continuing the previous example (Example 3) by inserting
  - 16 down to 10, and then 8 and 9



Inserting 16 and 15



Inserting 14



…and so on…

Deletion

- Deletion is a bit more complicated.
- We may need more than one rebalance on the path from deleted node to root.
- Deletion is O(logN)

Deletion of a Node

- Deletion of a node x from an AVL tree requires the same basic ideas, including single and double rotations, that are used for insertion.

- With each node of the AVL tree is associated a ***balance factor*** that is left high, equal, or right high, respectively, as the left subtree has height greater than, equal to, or less than that of the right subtree.

Deletion method

1. Reduce the problem to the case when the node *x* to be deleted has at most one child.
   - If *x* has two children replace it with its immediate predecessor *y* under inorder traversal (the immediate successor would be just as good)
   - Delete *y* from its original position, by proceeding as follows, using *y* in place of *x* in each of the following steps:

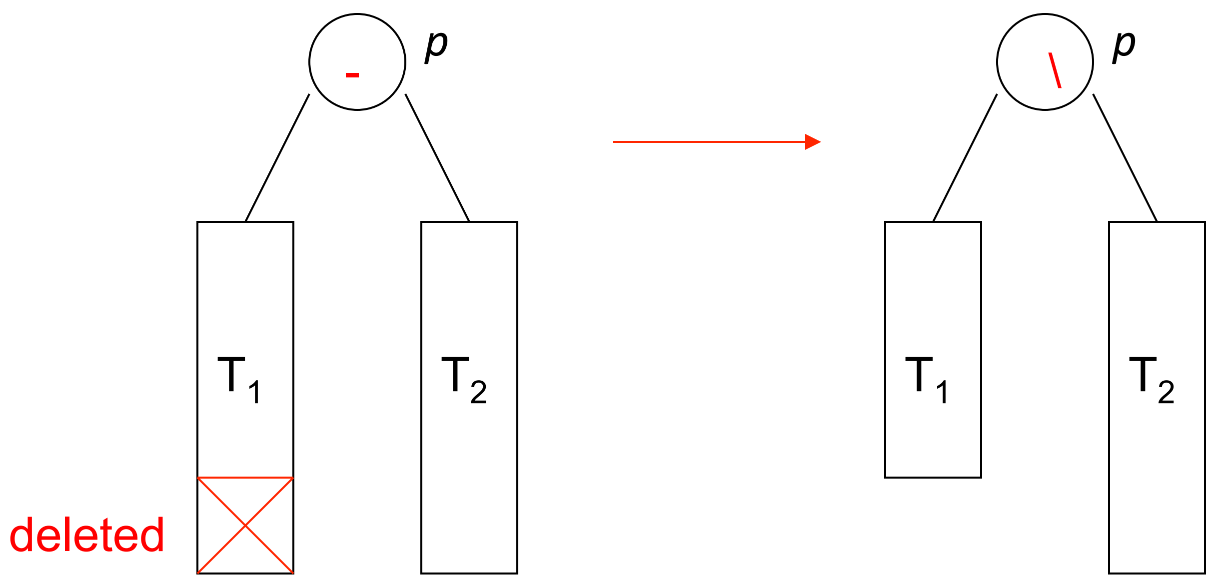
Deletion method (cnt'd)

2. Delete the node *x* from the tree.
   - We will trace the effects of this change on height through all the nodes on the path from *x* back to the root.
   - We use a Boolean variable **`shorter`** to show if the height of a subtree has been shortened.
   - The action to be taken at each node depends on
     - the value of `shorter`
     - balance factor of the node
     - sometimes the balance factor of a child of the node.
3. `shorter` is initially true. The following steps are to be done for each node *p* on the path from the parent of x to the root, provided shorter remains true. When shorter becomes false, the algorithm terminates.

Case 1

Case 1: The current node *p* has balance factor "equal".

- Change the balance factor of *p*
- shorter becomes false

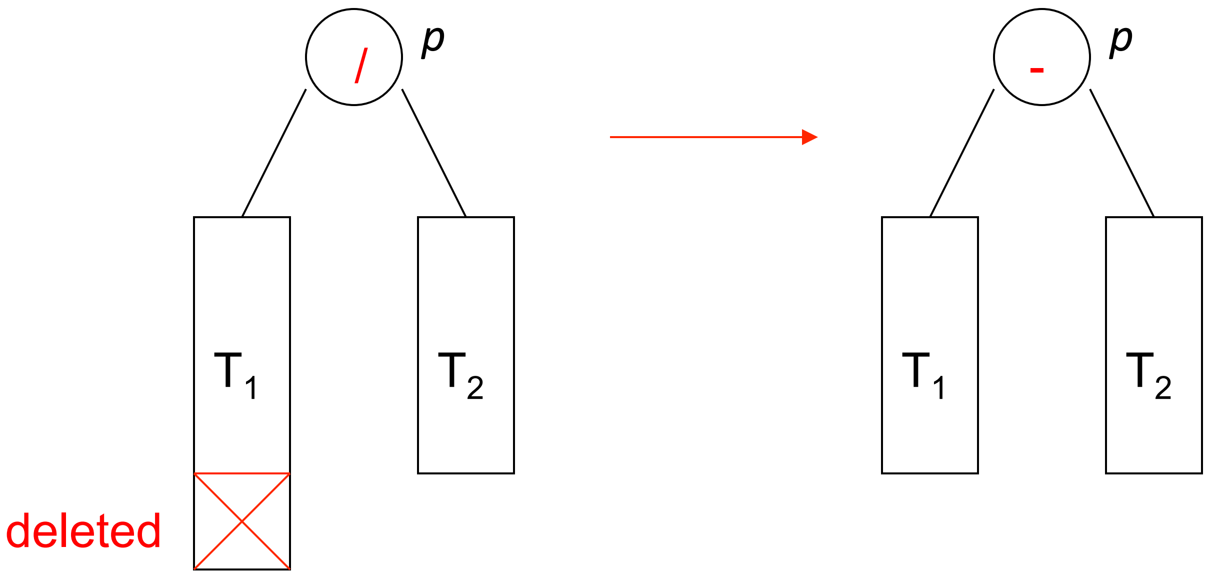


- No rotations
- Height unchanged

Case 2

Case 2: The balance factor of *p* is not equal and the taller subtree was shortened.

- Change the balance factor of *p* to equal
- Leave shorter true



- No rotations
- Height reduced

Case 3

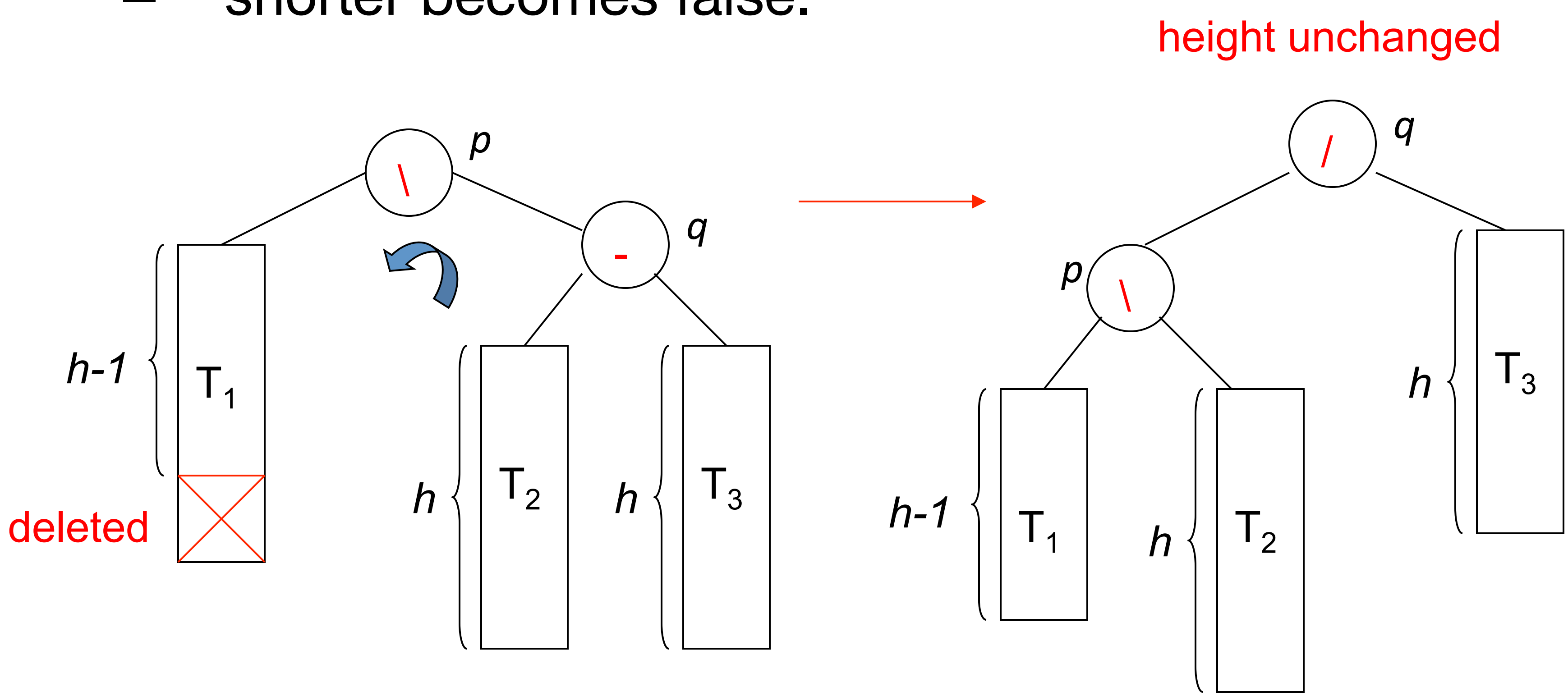
Case 3: The balance factor of *p* is not equal, and the shorter subtree was shortened.

- Rotation is needed (why?)
- Let *q* be the root of the taller subtree of *p*. We have three cases according to the balance factor of *q*:

Case 3a

Case 3a: The balance factor of *q* is equal.

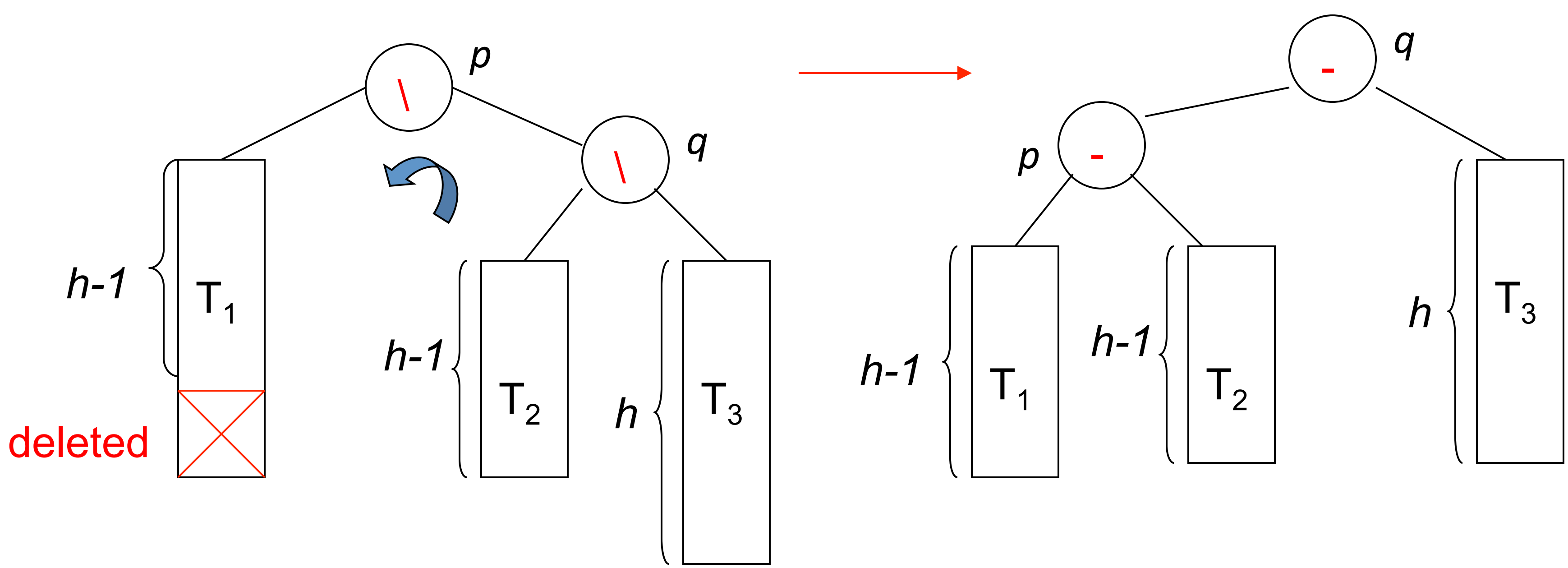
– Apply a single rotation
– shorter becomes false.



Case 3b

Case 3b: The balance factor of *q* is the same as that of p.

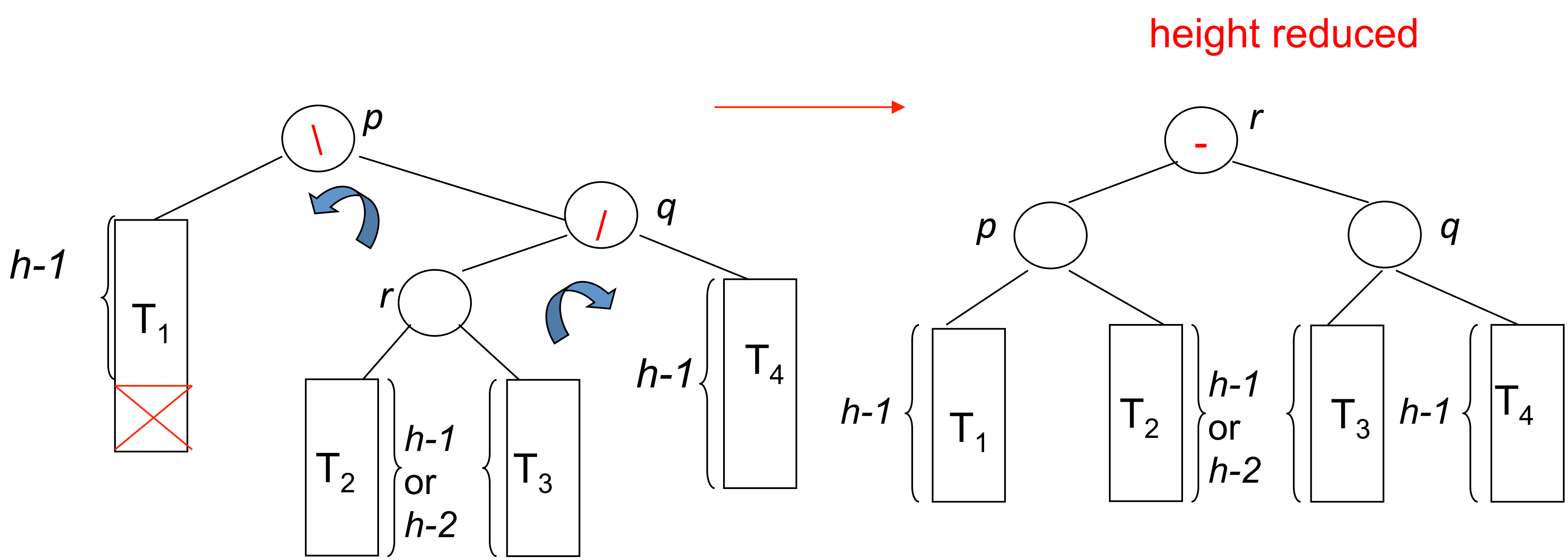
- Apply a single rotation
- Set the balance factors of *p* and *q* to equal
- leave shorter as true



Case 3c

Case 3c: The balance factors of *p* and *q* are opposite.

- Apply a double rotation
- set the balance factors of the new root to equal
- leave shorter as true



Example 1

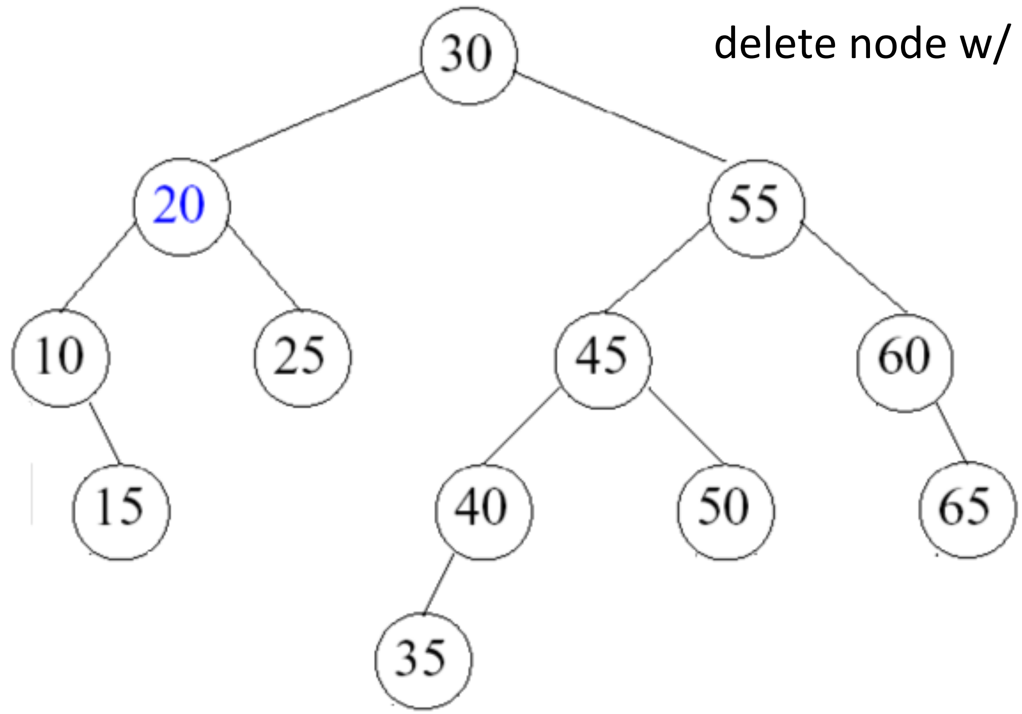
```
            9
          /   \
       6          15
     /   \      /     \
    2     7  13        20
   /        /  \      /  \
  1       12    14  18    30
         /
        11
```

delete node w/ key=2

?

```
              13
            /     \
          9         15
        /   \      /  \
      6      12  14    20
     / \    /         /  \
    1   7  11       18    30
```

Example 2



delete node w/ key=20

Arguments for AVL trees

1. Search is O(log N) since AVL trees are always balanced.
2. Insertion and deletions are also O(logn)
3. The height balancing adds no more than a constant factor to the speed of insertion.

Arguments against using AVL trees

1. Difficult to program & debug; more space for balance factor.
2. Asymptotically faster but rebalancing costs time.
3. Most large searches are done in database systems on disk and use other structures (e.g. B-trees).
4. May be OK to have O(N) for a single operation if total run time for many consecutive operations is fast.