Balanced Search Trees

Balanced Search Trees

- Balanced Binary Search Trees
  - AVL Trees
  - Red-Black Trees
- Balanced Multiway Trees
  - B-Trees

Binary Search Trees - Review

- Tree walks Θ(n)

- Queries
  - Search O(h)
  - Minimum O(h)
  - Maximum O(h)
  - Successor O(h)
  - Predecessor O(h)
- Modifying
  - Insert O(h)
  - Delete O(h)

Each of the basic operations on a binary search tree runs in O(h) time, where h is the ***height*** of the tree

The height of a binary search tree of n nodes depends on the order in which the keys are inserted

The height of a BST with n nodes:
Worst case: O(n) => **BST operations are also O(n) !!!**
Best case: O(log n)
Average case O(log n)

Binary Search Trees - Review

- Tree walks Θ(n)
- Queries
  - Search O(h)
  - Insert O(h)
  - Delete O(h)

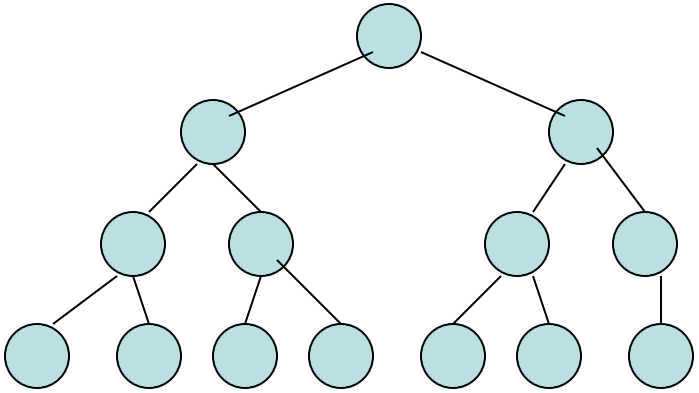
Each of the basic operations on a binary search tree runs in O(h)

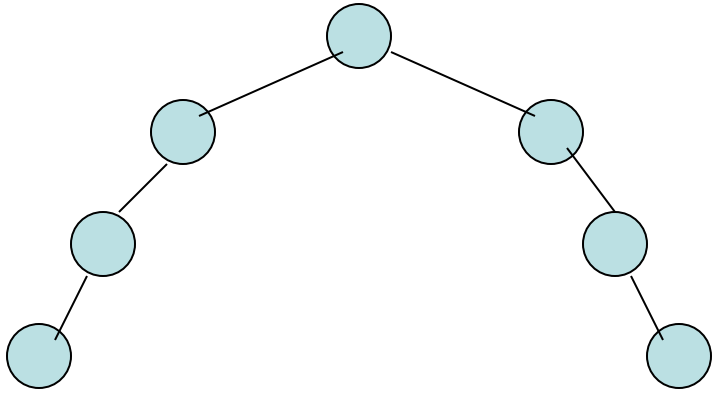
Conclusion: we must do *something* to make sure that the height is kept small !
The BST must be always (kind of) *balanced*

The height of a BST with n nodes:
Worst case: O(n) => **BST operations are also O(n) !!!**
Best case: O(log n)
Average case O(log n)

Balanced Binary Search Trees

- A BST is ***perfectly balanced*** if, ***for every node***, the difference between the ***number of nodes*** in its left subtree and the number of nodes in its right subtree is at most one
- Example: Balanced tree vs Not balanced tree



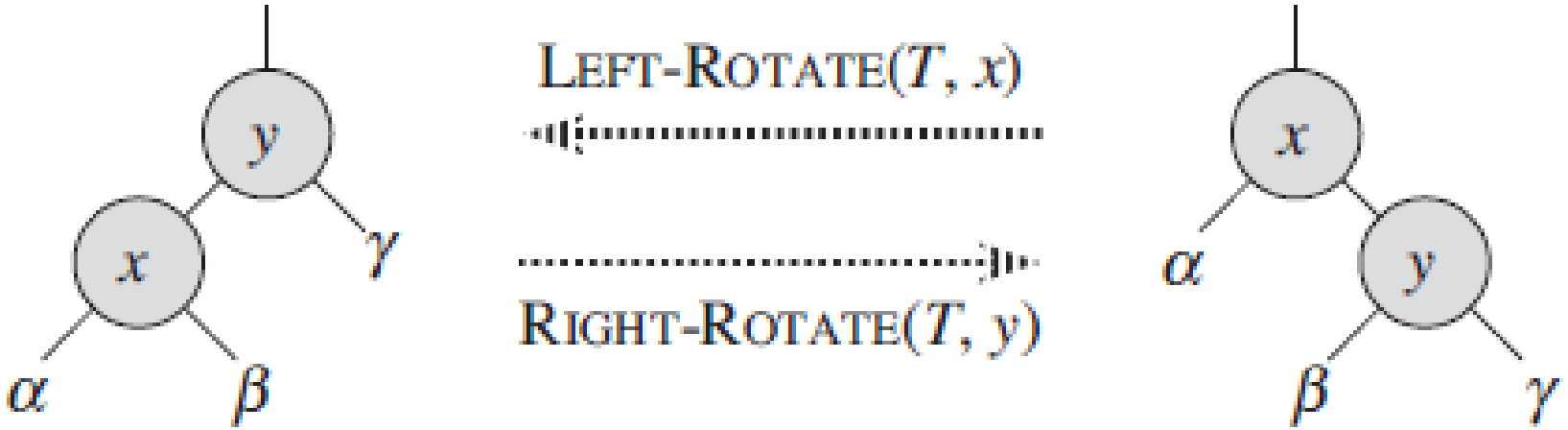


Balancing Binary Search Trees

- Inserting or deleting a node from a (balanced) binary search tree can lead to an unbalance
- In this case, we perform some operations to rearrange the binary search tree in a balanced form
  - These operations must be ***easy to perform*** and must require only a minimum number of links to be reassigned
  - Such kind of operations are ***Rotations***

Tree Rotations

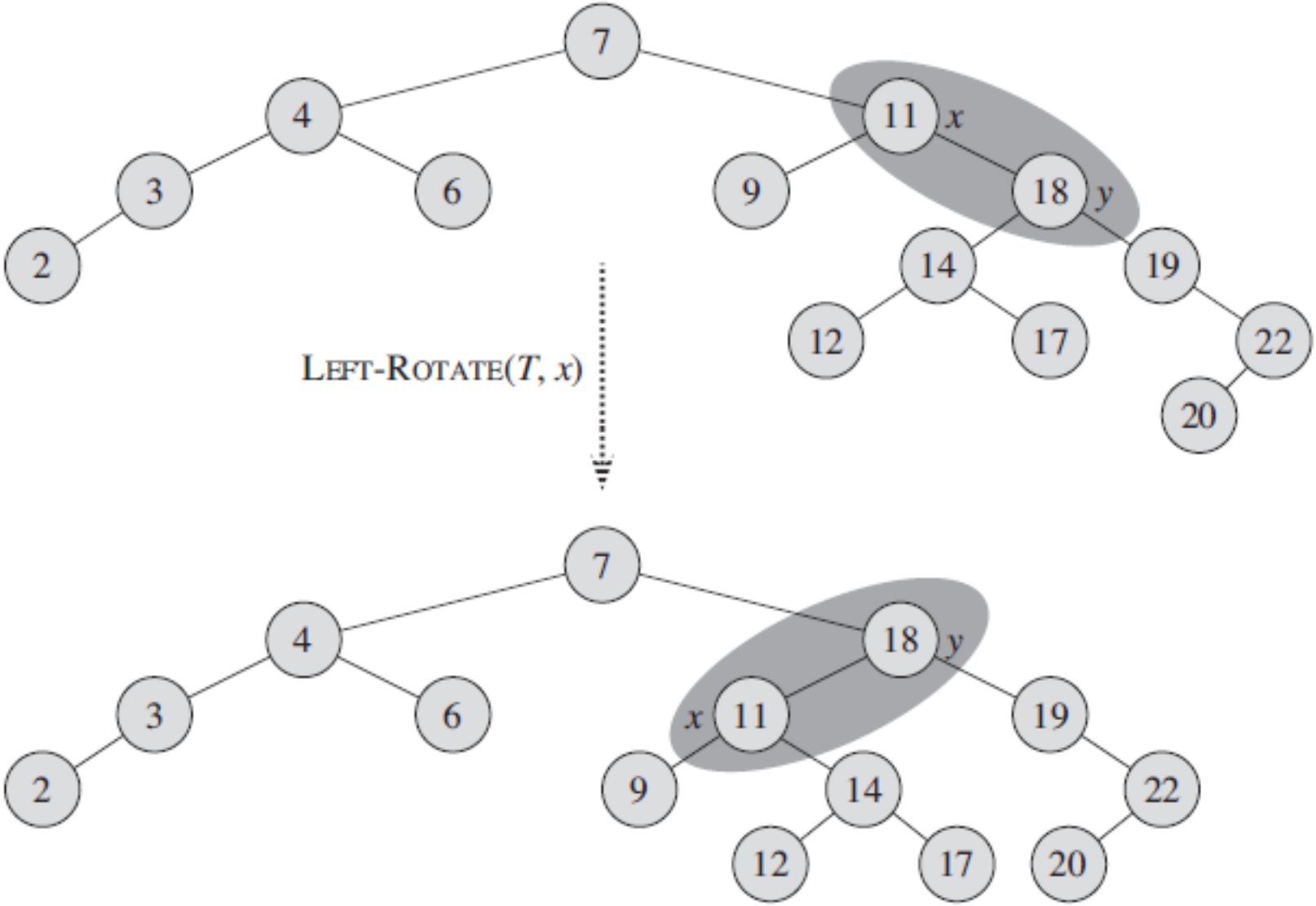
- The basic tree-restructuring operation
- There are left rotation and right rotation. They are inverses of each other



[CLRS Fig. 13.1]

Tree Rotations

- Changes the local pointer structure. (Only pointers are changed.)
- A rotation operation preserves the binary-search-tree property: the keys in α precede x.*key*, which precedes the keys in β, which precede y.*key*, which precedes the keys in γ .


7
4
11
x
3
6
9
18
y
2
14
19
12
17
22
20
LEFT-ROTATE(T, x)
7
4
18
y
3
6
x
11
19
2
9
14
22
12
17
20

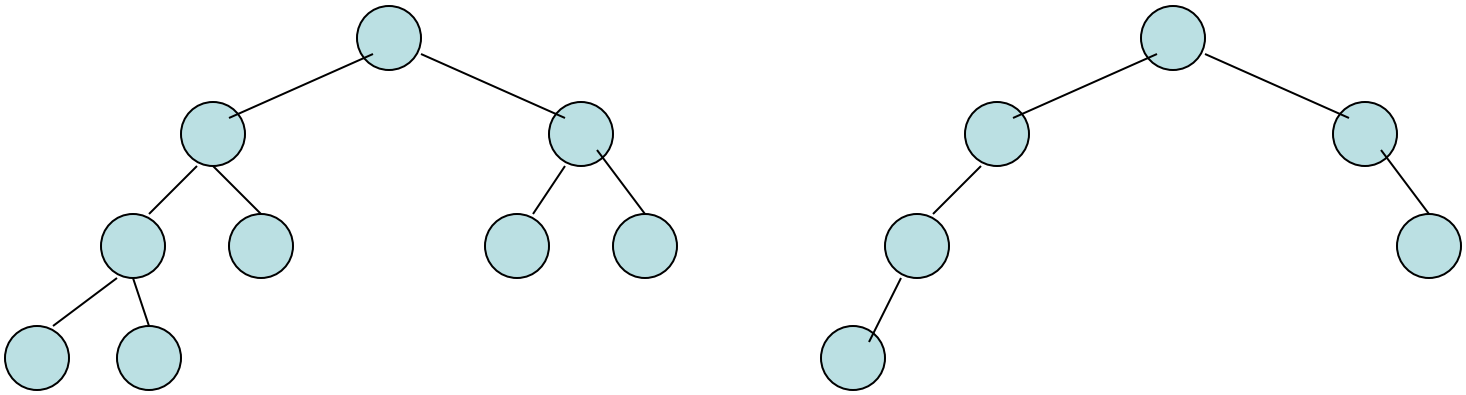
[CLRS Fig. 13.3]

Balancing Binary Search Trees

- Balancing a BST is done by applying simple transformations such as rotations to fix up after an insertion or a deletion
- Perfectly balanced BST are very difficult to maintain
- Different approximations are used for more relaxed definitions of “balanced”, for example:
  - AVL trees
  - Red-black trees

AVL trees

- Adelson Velskii and Landes
- An ***AVL tree*** is a binary search tree that is ***height balanced***: *for each node* x, the *heights* of the left and right subtrees of x differ by at most 1.
- AVL Tree vs Non-AVL Tree

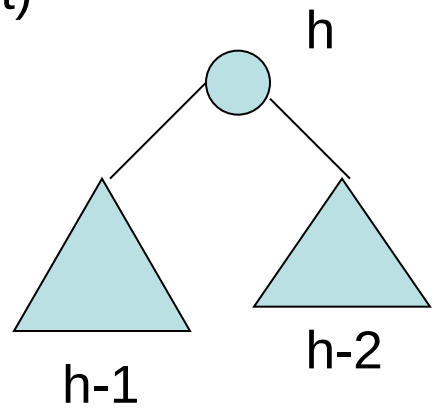


AVL Trees

- AVL trees are height-balanced binary search trees
- Balance factor of a node
  - height(left subtree) - height(right subtree)
- An AVL tree has balance factor calculated at every node
  - For every node, heights of left and right subtree can differ by no more than 1

Height of an AVL Tree

- How many nodes are there in an AVL tree of height h ?
- What is the *minimum* number of nodes that must be used in order to build an AVL tree of height h?
- N(h) = minimum number of nodes in an AVL tree of height h.
- Base Cases:
  - N(0) = 1 (need 1 node for h=0 = only the root)
  - N(1) = 2
- Induction Step:
  - N(h) = N(h-1) + N(h-2) + 1
- Solution: (it's the Fibonacci recurrence)
  - $N(h) = \phi^h \quad (\phi \approx 1.62)$



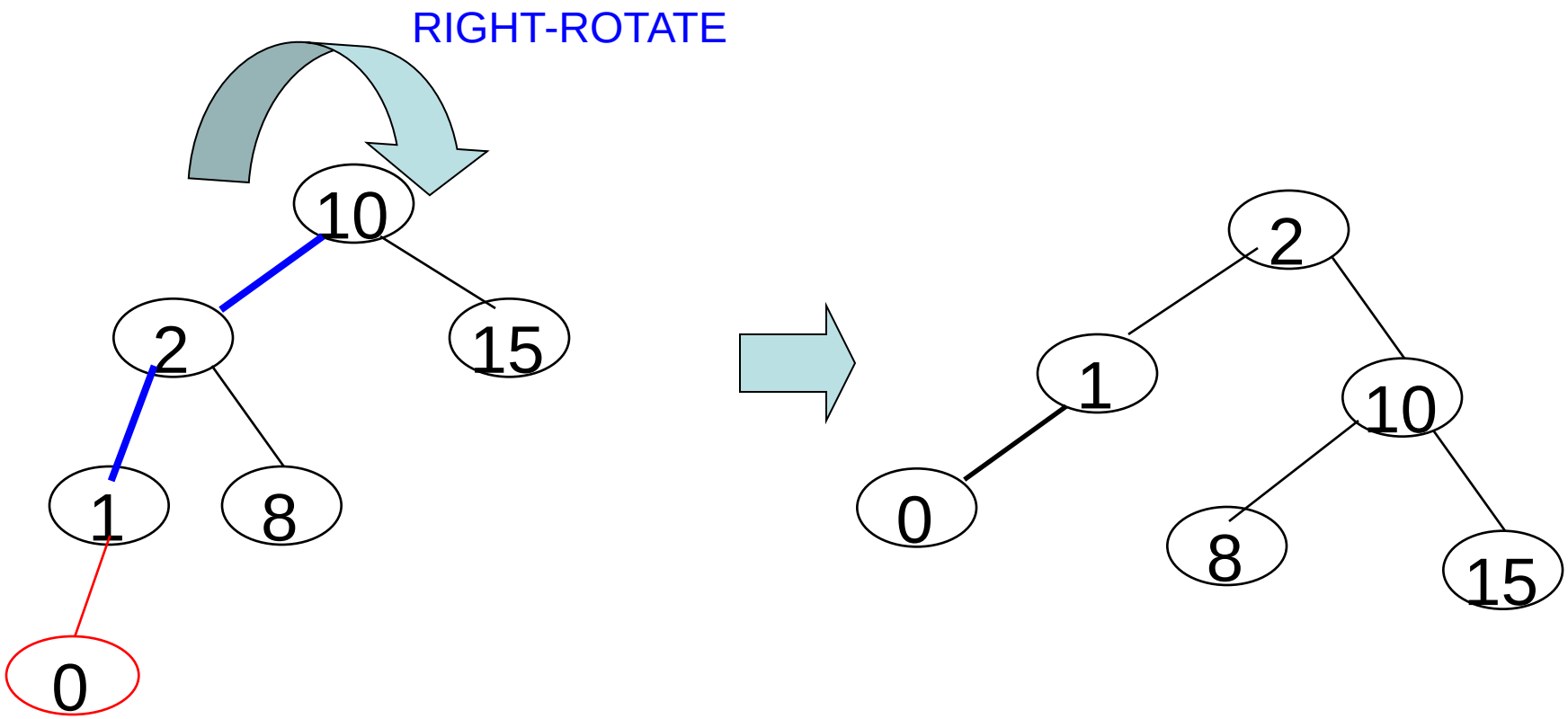
Height of an AVL Tree

- $N(h)=\phi^h \quad (\phi \approx 1.62)$
- What is the height of an AVL Tree with n nodes ?
- Suppose we have n nodes in an AVL tree of height h.
  - $n \geq N(h)$ (because N(h) was the minimum)
  - $n \geq \phi^h$ hence $\log_\phi n \geq h$
  - $h \leq \log_\phi n = \log_2 n / \log_2 \phi$
  - $h \leq 1.44 \log_2 n$ (h is O(logn))

Insertion into an AVL trees

1. place a node into the appropriate place in binary search tree order
2. examine height balancing on insertion path:
    1. Tree was balanced (balance=0) => increasing the height of a subtree will be in the tolerated interval +/-1
    2. Tree was not balanced, with a factor +/-1, and the node is inserted in the *smaller* subtree leading to its height increase => the tree will be balanced after insertion
    3. Tree was balanced, with a factor +/-1, and the node is inserted in the *taller* subtree leading to its height increase => the tree is no longer height balanced (the heights of the left and right children of some node x might differ by 2)
        - we have to balance the subtree rooted at x using rotations
        - How to rotate ? => see 4 cases according to the path to the new node

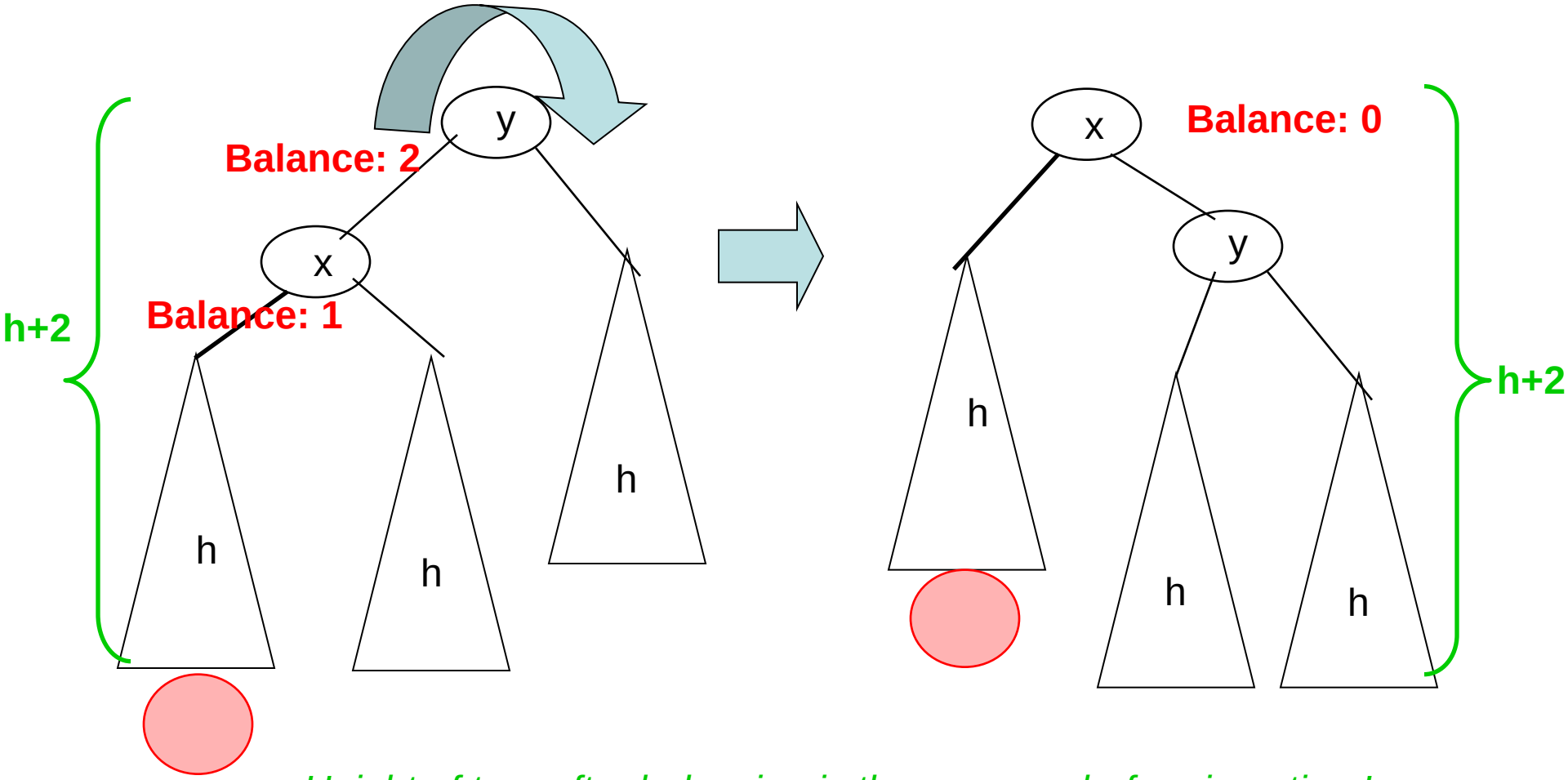
Example – AVL insertions



Case 1: Node's **Left – Left** grandchild is too tall

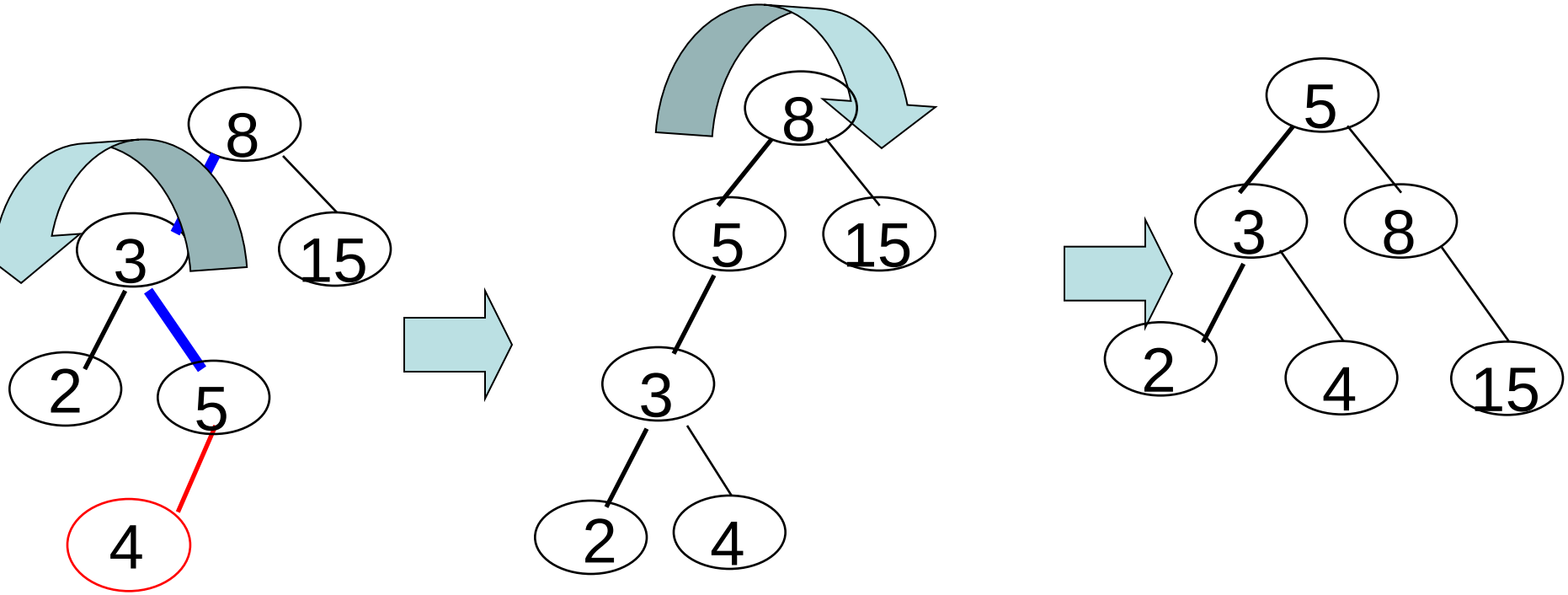
AVL insertions – Right Rotation

Case 1: Node's **Left – Left** grandchild is too tall



Height of tree after balancing is the same as before insertion !
=> there are NO upward propagations of the unbalance !

Example – AVL insertions

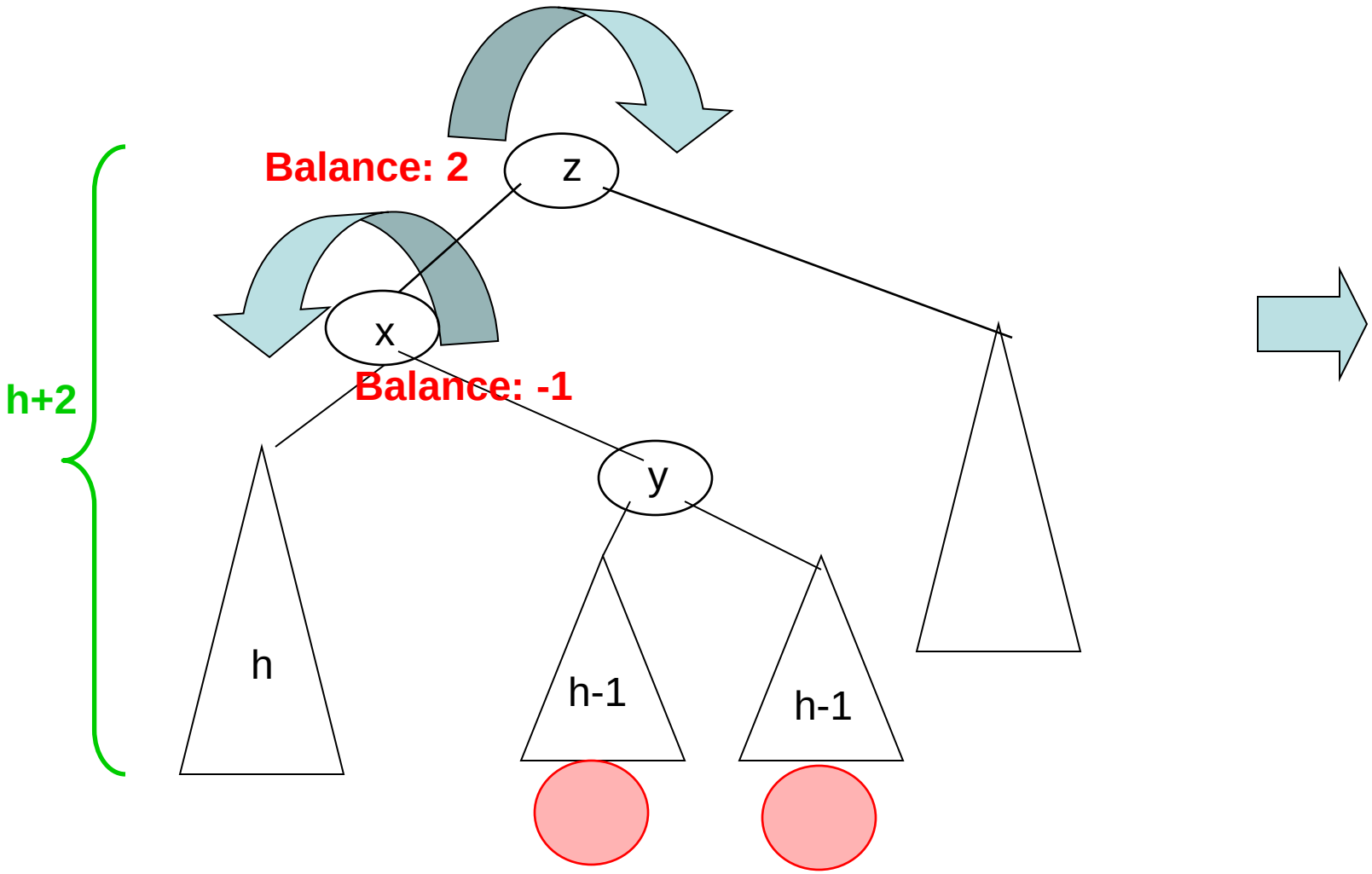


Solution: do a Double Rotation: LEFT-ROTATE and RIGHT-ROTATE

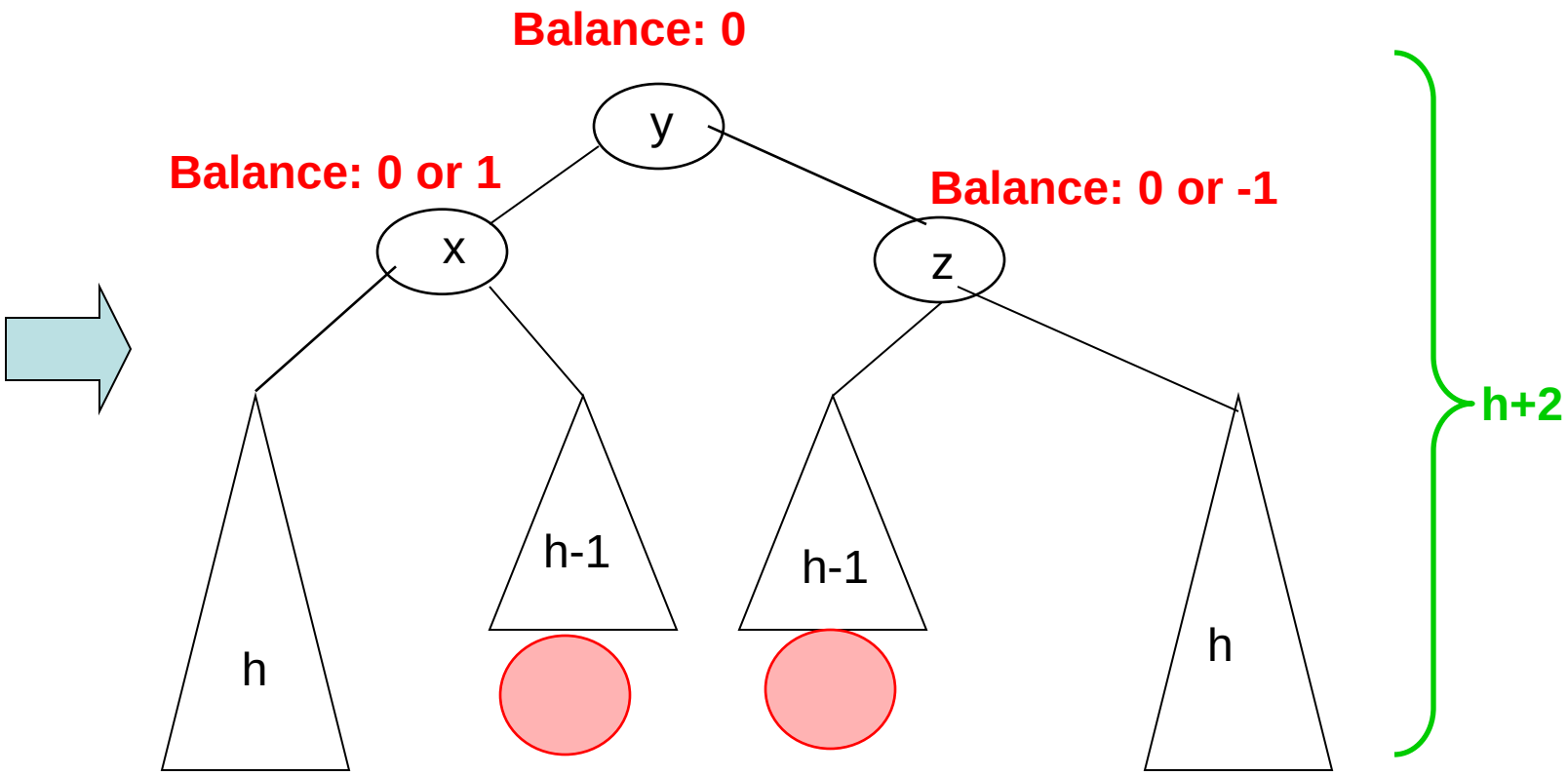
Case 2: Node's **Left-Right** grandchild is too tall

Double Rotation – Case Left-Right

Case 2: Node's **Left-Right** grandchild is too tall



Double Rotation – Case Left-Right

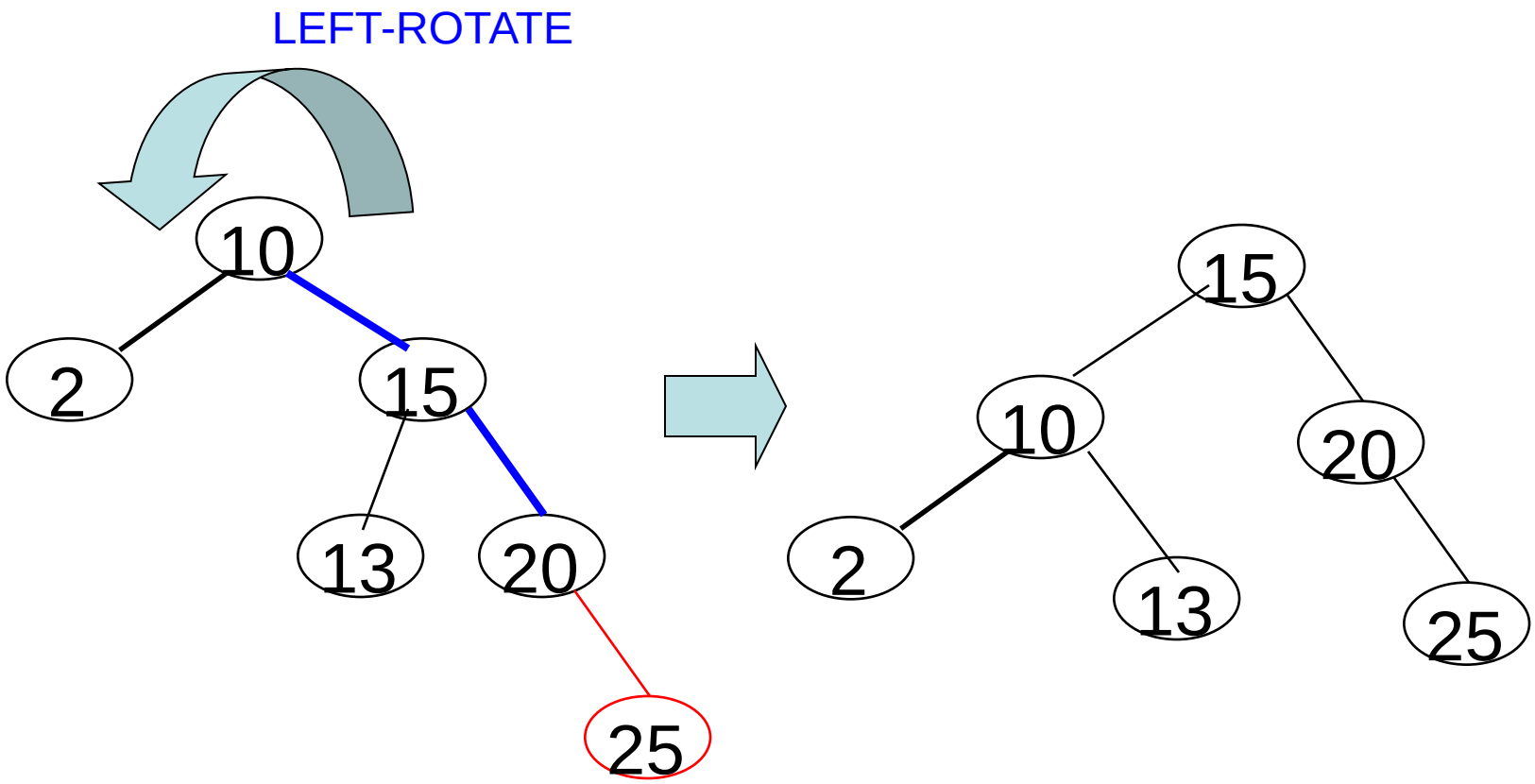


Height of tree after balancing is the same as before insertion !
=> there are NO upward propagations of the unbalance !

Exercise

- Insert following keys into an initially empty AVL tree. Indicate the rotation cases:
- 14, 17, 11, 7, 3, 13

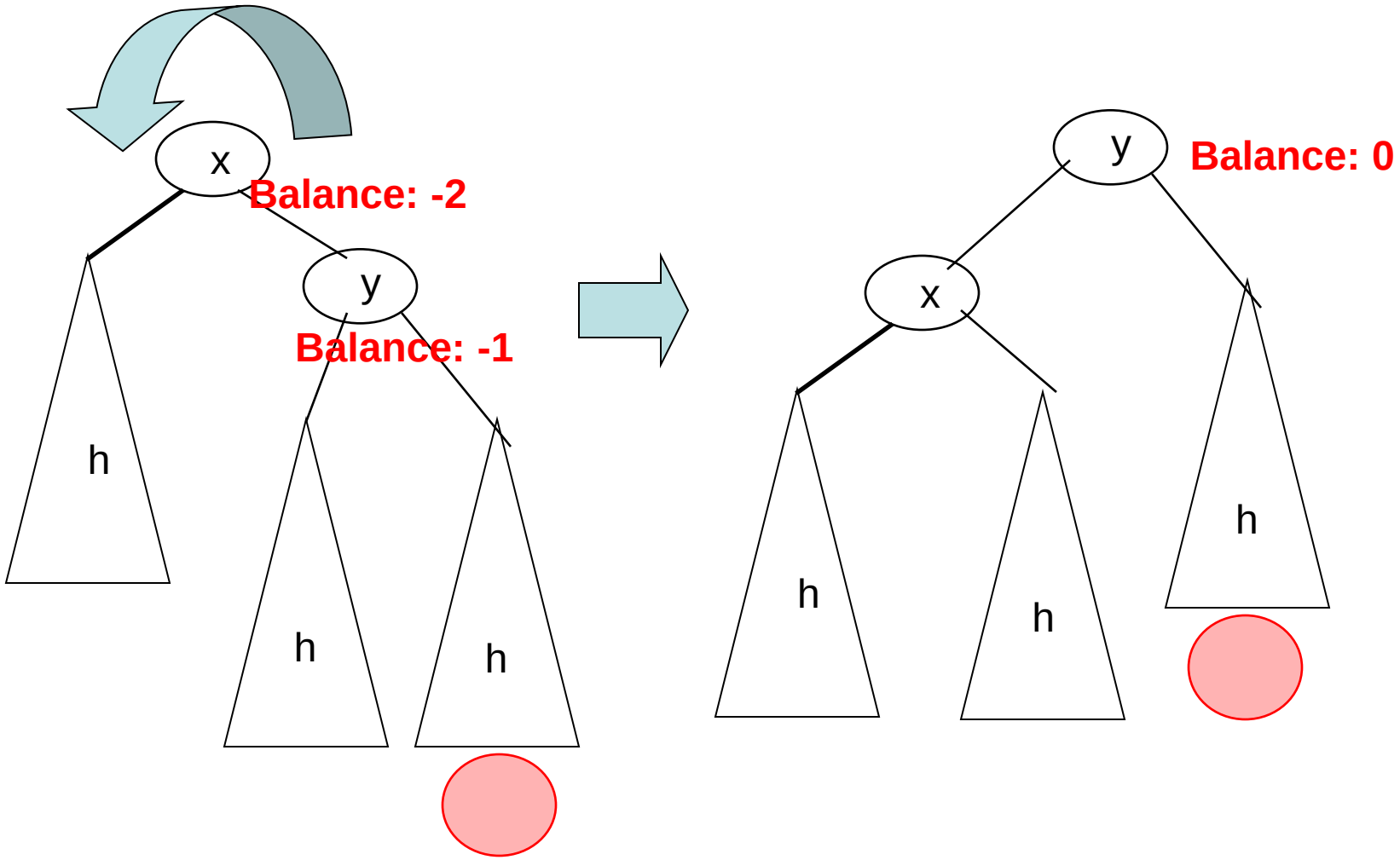
Example – AVL insertions



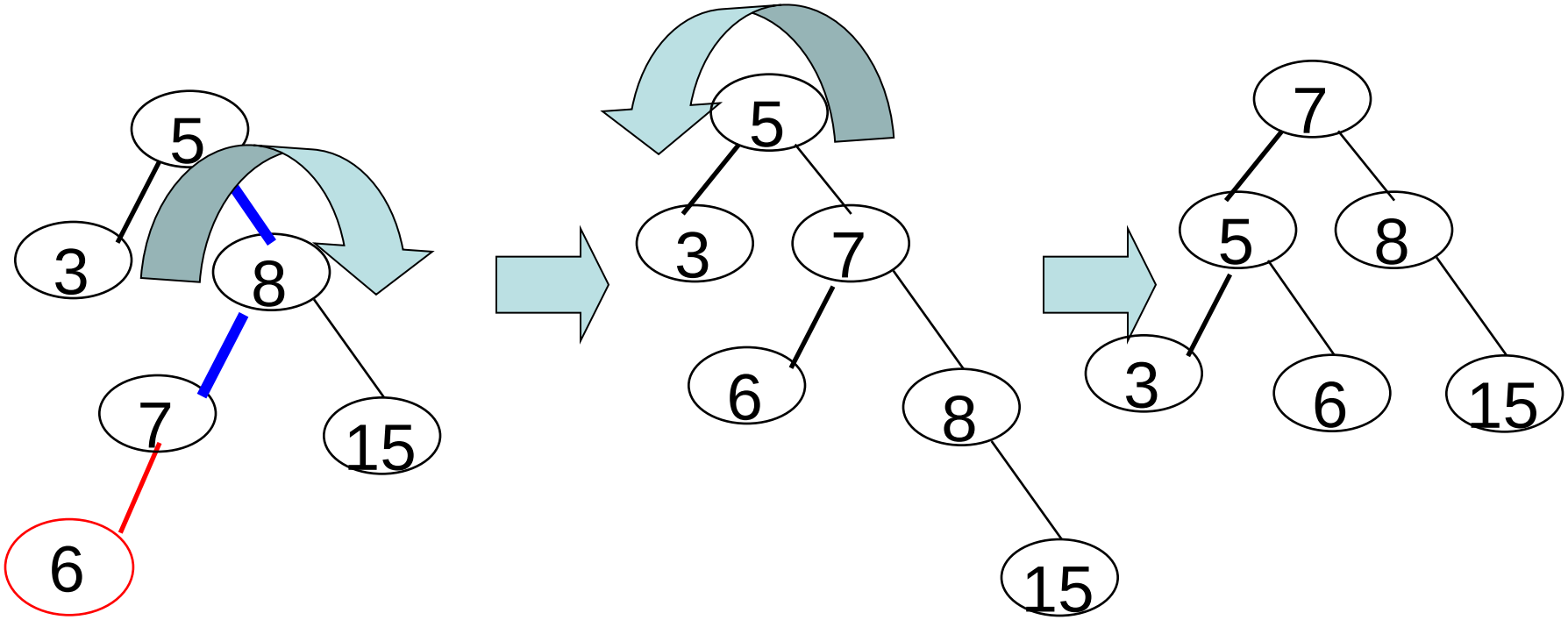
Case 3: Node's **Right – Right** grandchild is too tall

AVL insertions – Left Rotation

Case 3: Node's **Right – Right** grandchild is too tall



Example – AVL insertions

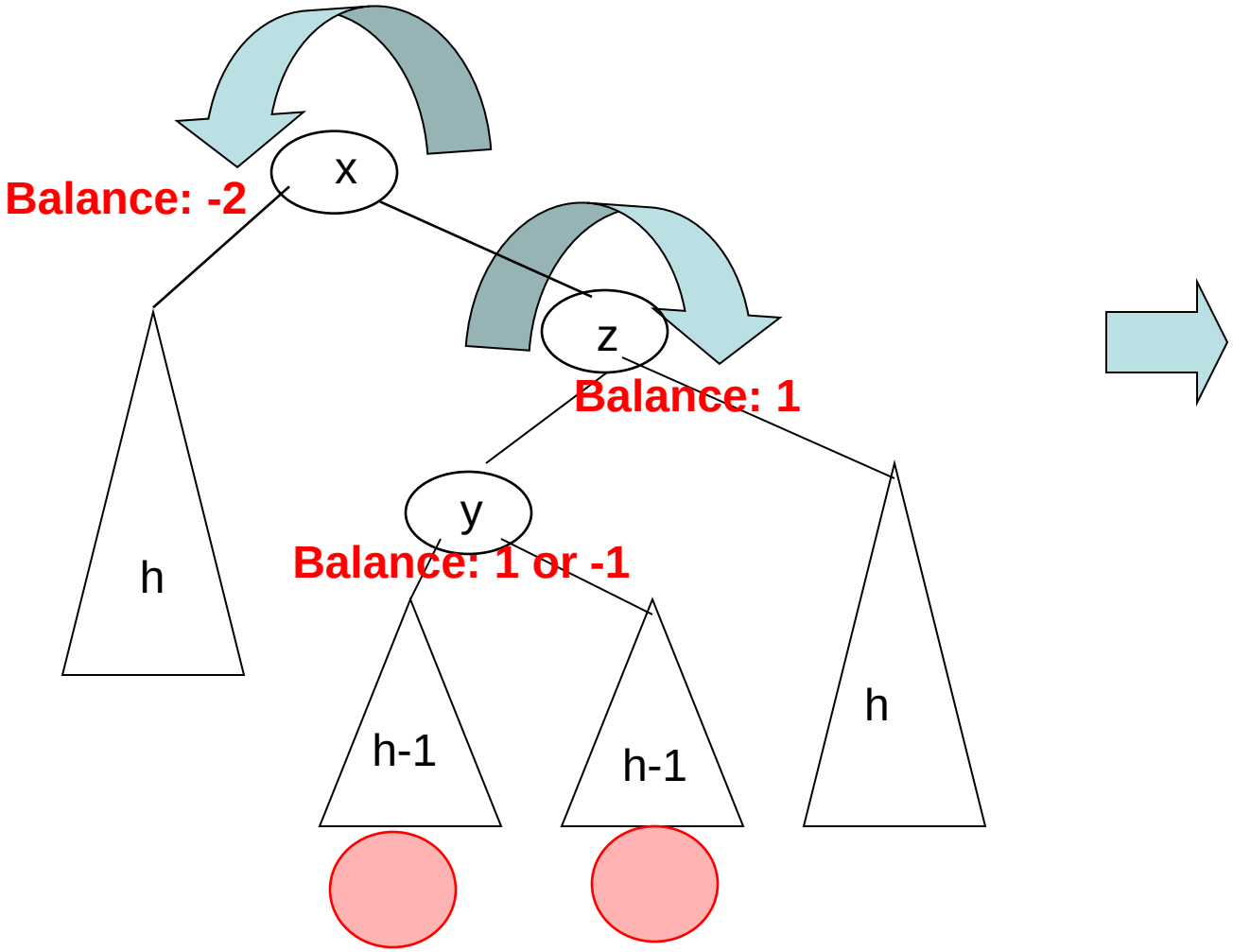


Solution: do a Double Rotation: RIGHT-ROTATE and LEFT-ROTATE

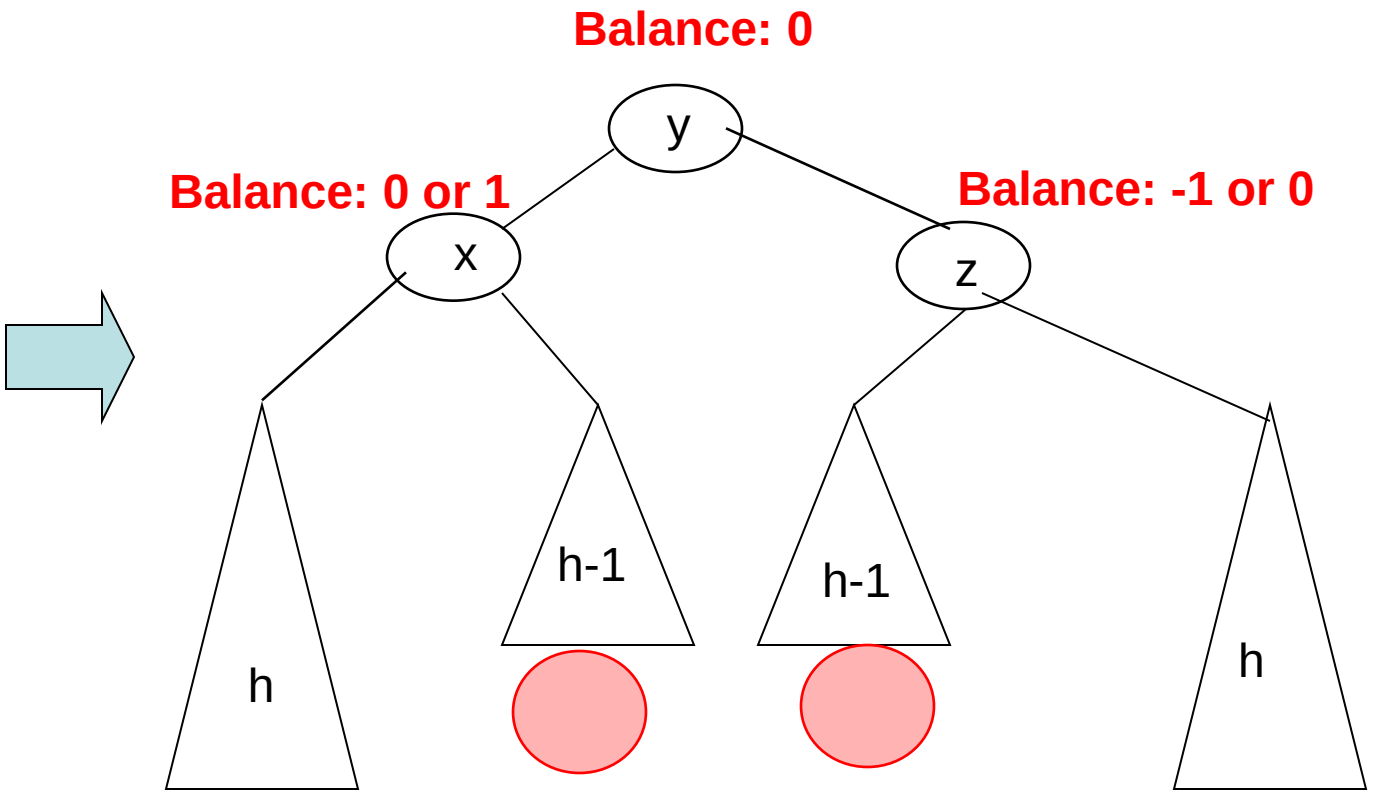
Case 4: Node's **Right – Left** grandchild is too tall

Double Rotation – Case Right-Left

Case 4: Node's **Right – Left** grandchild is too tall



Double Rotation – Case Right-Left



Implementing AVL Trees

- Insertion *needs* information about the height of each node
- It would be highly inefficient to calculate the height of a subtree every time this information is needed => *the tree structure is augmented with height information that is maintained during all operations (during insert and delete)*
- *An AVL Node contains the attributes:*
  - *Key*
  - *Left, right*
  - *Height (or Balance Factor)*

Simple AVL Tree Implementation

```
class AVLNode {
    Integer key;      // sorted by key
    Integer val;      // associated data
    AVLNode left;
    AVLNode right;    // left and right subtrees
    int height;       // height of subtree rooted in this node

    public AVLNode(Integer key, Integer val, int height) {
        this.key = key;
        this.val = val;
        this.height = height;
        this.left = null;
        this.right = null;
    }
}

public class IntegerAVL {
    private AVLNode root;


// ....
}
```

Simple example of AVL tree with Integer keys and Integer values.
In a general case, keys type could be any comparable type.
.

Height of a AVL node

```
// AVL nodes  record their height, so no need to compute!
private int height(AVLNode x) {
    if (x == null) return -1;
    return x.height;
}
```

Insert in AVL tree

```
public void put(Integer key, Integer val) {
    if (key == null) throw new IllegalArgumentException("key is null");
    root = put(root, key, val);
}

private AVLNode put(AVLNode x, Integer key, Integer val) {
    if (x == null) return new AVLNode(key, val, 0);
    if (key < x.key) {
        x.left = put(x.left, key, val);
    } else if (key > x.key) {
        x.right = put(x.right, key, val);
    } else {
        x.val = val;
        return x;
    }
  x.height = 1 + Math.max(height(x.left), height(x.right));

  return balance(x);
}
```

The balance() operation

```
private AVLNode balance(AVLNode x) {
    if (balanceFactor(x) < -1) {
        if (balanceFactor(x.right) > 0) {
            x.right = rotateRight(x.right);
        }
        x = rotateLeft(x);
    }
    else if (balanceFactor(x) > 1) {
        if (balanceFactor(x.left) < 0) {
            x.left = rotateLeft(x.left);
        }
        x = rotateRight(x);
    }
    return x;
}

private int balanceFactor(AVLNode x) {
     return height(x.left) - height(x.right);
}
```

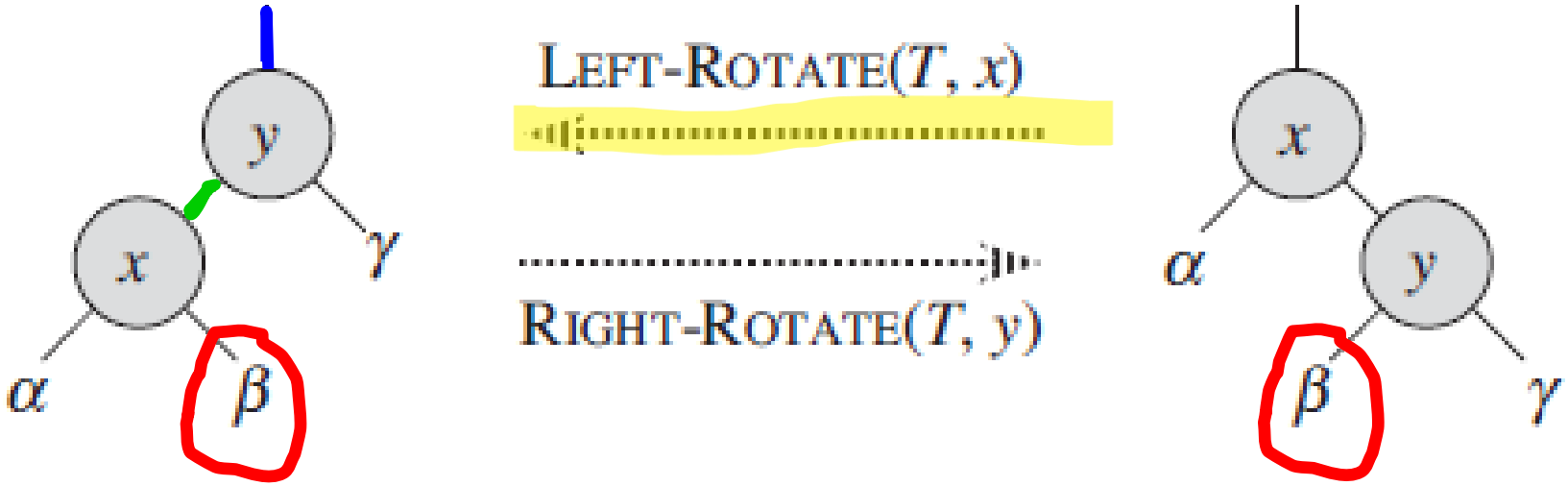
Case 4 – Right-Left

Case 3 – Right-Right

Case 2 – Left-Right

Case 1 – Left-Left

Tree Rotations



[CLRS Fig. 13.1]

Rotate Left, Rotate Right

```
private AVLNode rotateLeft(AVLNode x) {
    AVLNode y = x.right;
    x.right = y.left;      // move Beta
    y.left = x;            // old root x becomes child
    x.height = 1 + Math.max(height(x.left), height(x.right));
    y.height = 1 + Math.max(height(y.left), height(y.right));
    return y;              // return new root
}
```

```
private AVLNode rotateRight(AVLNode y) {
    AVLNode x = y.left;
    y.left = x.right;
    x.right = y;
    y.height = 1 + Math.max(height(y.left), height(y.right));
    x.height = 1 + Math.max(height(x.left), height(x.right));
    return x;
}
```

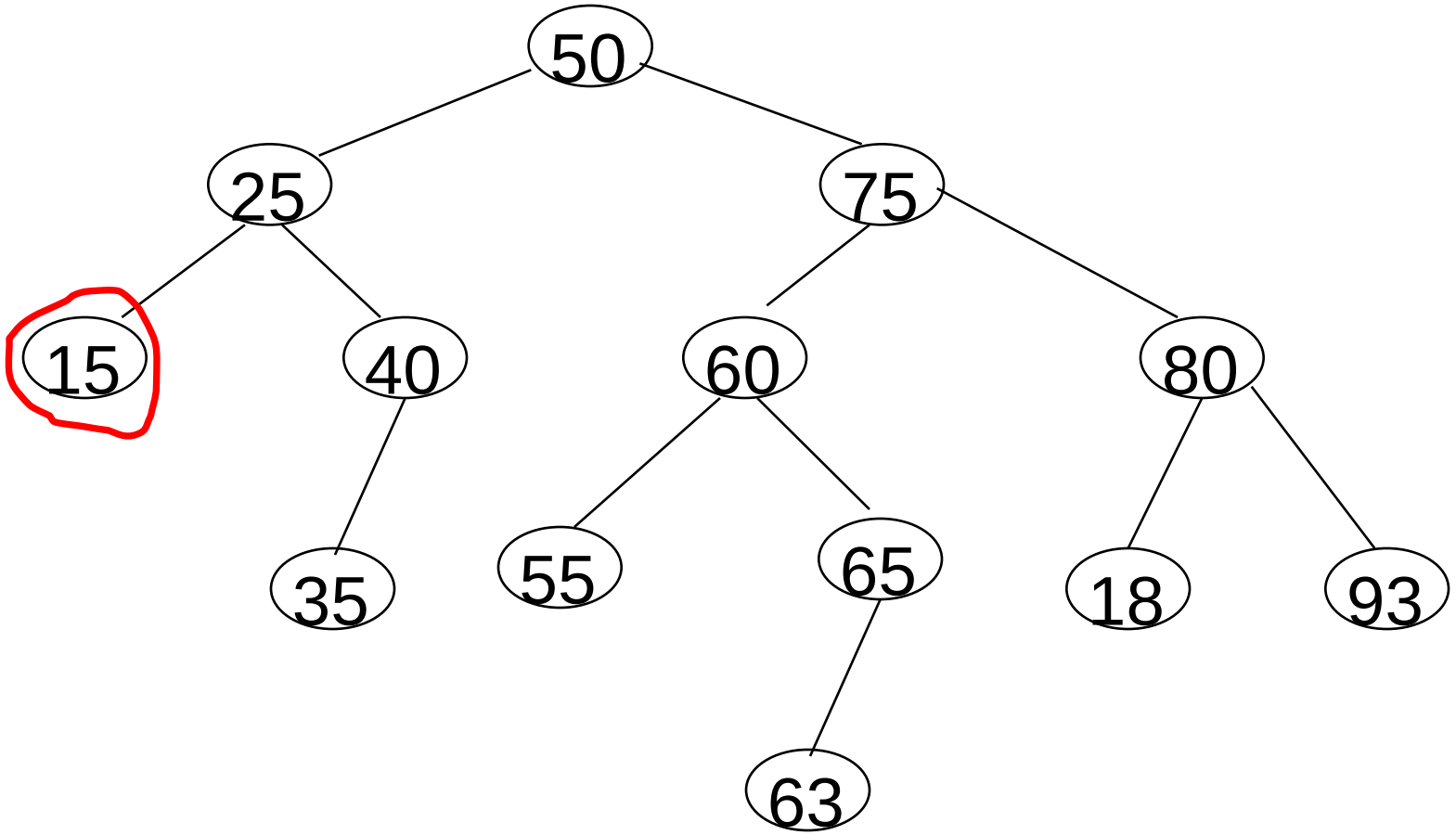
Analysis of AVL-INSERT

- Insertion makes O(h) steps, h is O(log n), thus Insertion makes O(log n) steps
- At every insertion step, there is a call to Balance, but *rotations will be performed only once for the insertion of a key*. It is not possible that after doing a balancing, unbalances are propagated , because the BALANCE operation restores the height of the subtree before insertion. => number of rotations for one insertion is O(1)
- AVL-INSERT is O(log n)

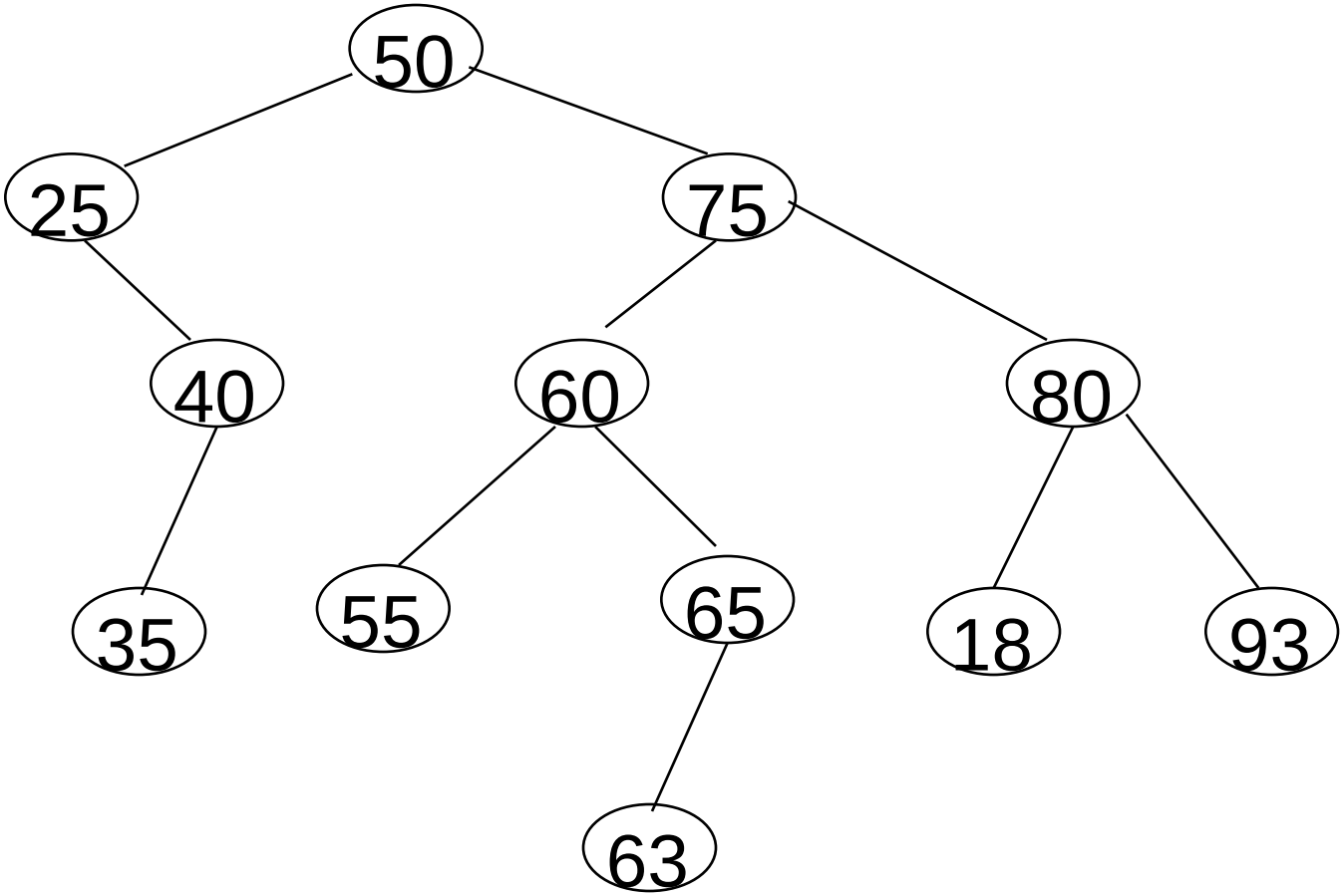
AVL Delete

- In principle: delete according to the BST delete algorithm, then do balance if needed as we return along the deletion path back to the root
- Balance has the same 4 cases as insert
- Different to AVL insert, in the case of delete the balance operation *can propagate* the whole way up to the root, because after deleting a node from a subtree the height of the subtree may shrink and unbalance may propagate to parent!

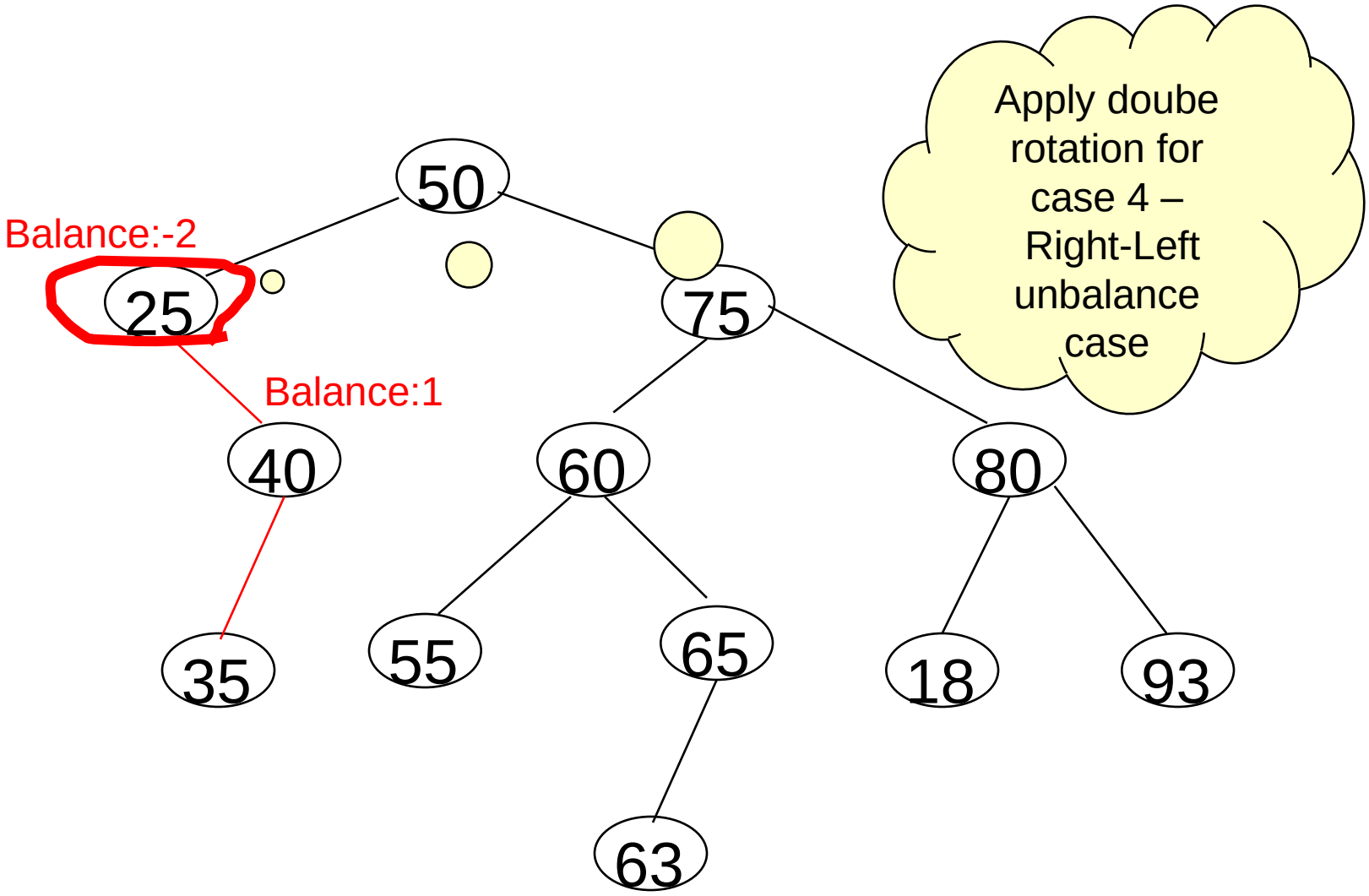
AVL Delete Example: Delete 15



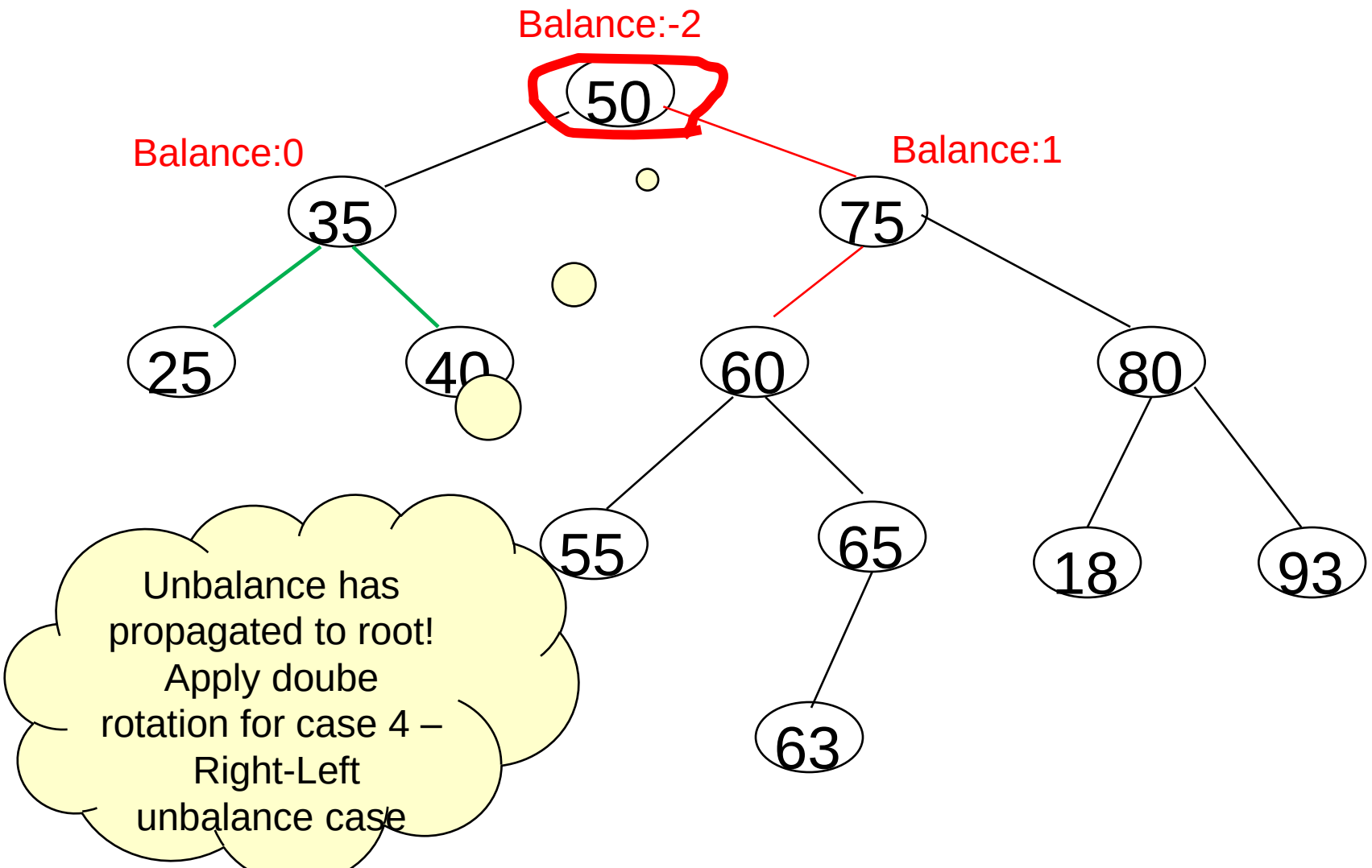
Example: Step 1: do BST delete



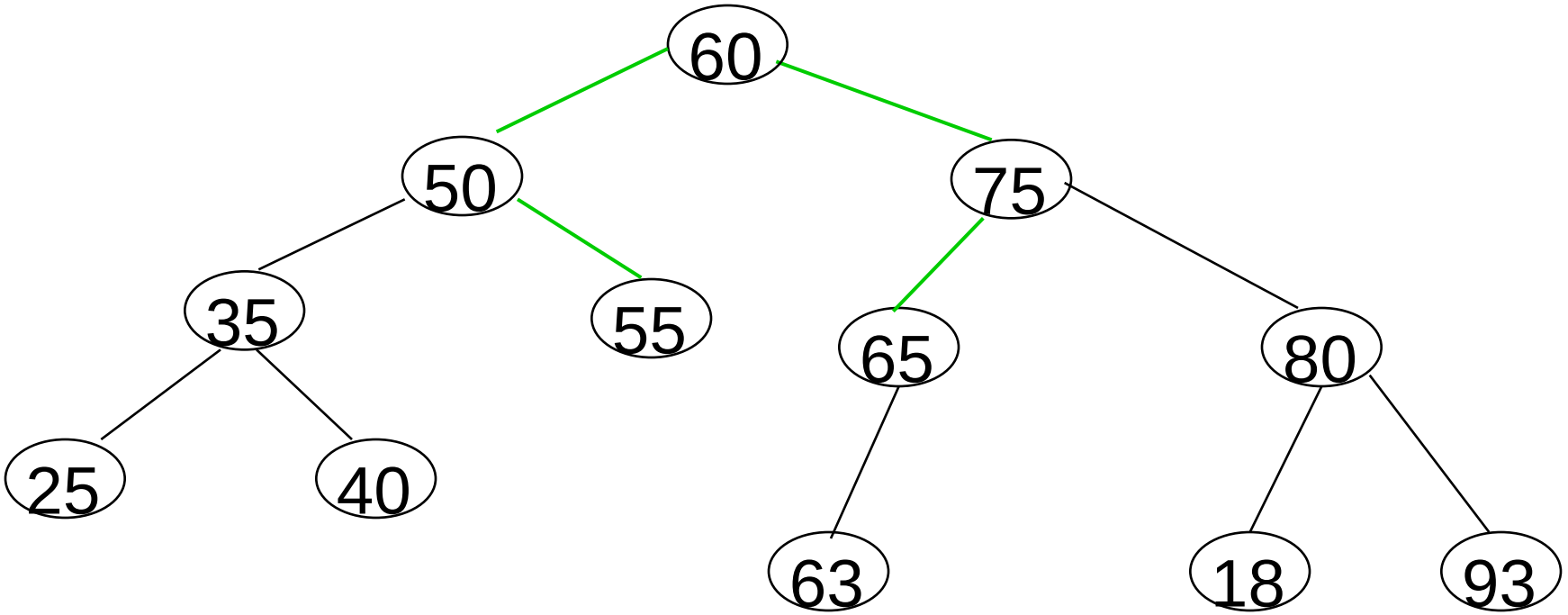
Example: Step 2: need to balance() at 25



Example: Step 3: need to balance() at 50



Example: Step 4: final result



AVL Trees - Summary

- AVL definition of balance: *for each node* x, the *heights* of the left and right subtrees of x differ by at most 1.
- Maximum height of an AVL tree with n nodes is $h \leq 1.44 \log_2 n$
- AVL-Insert: O(log n), Rotations: O(1) (For Insert, unbalances are not propagated after they are solved once)
- AVL-Delete: O(log n), Rotations: O(log n) (For Delete, unbalances may be propagated up to the root)

AVL vs Simple BST

	AVL	Simple BST
Max Height	1.44 log n	n
INSERT	O(log n)	O(n)
Rotations at Insert	O(1)	
DELETE	O(log n)	O(n)
Rotations at Delete	O(log n)	

Simple AVL Tree Implementation

- Source code:
- IntegerAVL.java

Using Balanced Binary Search Trees for Sorted Dictionary(Sorted Map)

- BST are well suited to implement Sorted Dictionary (Sorted Map) structures, but in order to be efficient, we must keep their height small by applying balancing techniques
  - AVL is one of these techniques
  - Red-Black trees is another similar technique -> to follow next lecture
- java.util.TreeMap is a SortedMap implementation based on balanced binary search trees (it particularly uses Red-Black)
- You could implement your own TreeMap with the same performance! You just need to:
  - Define a SortedMap interface with needed operations as an abstract data type because clients must not know internal implementation details
  - Use type parameters to make it reusable by different clients
  - Implement with AVL tree

Sorted Map (Sorted Dictionary) in Java Collections

- Interface **`java.util.SortedMap<K,V>`**
  - **`Extends interface Map<K,V>`**
- General purpose implementation: Class **`java.util.TreeMap<K,V>`**
  - **`TreeMap`** is implemented by *Balanced Binary Search Trees*

Using BST to implement our Sorted Dictionary(Sorted Map)

- The simple `GenericBST` or `GenericAVL` implementations are not yet fully useful as a SortedDictionary (SortedMap) abstract data type:
  - The inorder() operation should be replaced by operations getKeys() and getEntries() that return an iterator over all keys in ascending order.
    - In this way the client can decide what kind of processing they need (printing or something else)

- Our implementation:
  - `interface MySortedMap,`
  - `Implemented by:  class MyAVLMap`

Our Sorted Map Abstract Data Type

```
public interface MySortedMap <Key extends Comparable<Key>, Value>{

    boolean containsKey(Key key);

    Value get(Key key);

    void put(Key key, Value val);

    void remove(Key key);

    Iterable<Key> getKeys();

    Iterable<Pair<Key,  Value>> getEntries();

}
```

The SortedMap Abstract Data Type must not expose its implementation details, including the detail that it is implemented using Binary Search Trees

The SortedMap Abstract Data Type must provide methods to obtain all keys in sorted order and all entries in sorted order of their kys

```
public class MyAVLMap<Key extends Comparable<Key>, Value> implements
                                                    MySortedMap<Key, Value> {

    private Node root;            // root of BST

    private class Node {
        private Key key;      // sorted by key
        private Value val;           // associated data
        private Node left, right;  // left and right subtrees
        private int height;     //height of node


        public Node(Key key, Value val, int height ) {
            this.key = key;
            this.val = val;
            this.height = height;
        }
    }


public MyBSTMap() {
        root = null;
    }

//… implement methods
}
```

```
public Iterable<Key> getKeys() {
    Queue<Key> q= new LinkedList<Key>();
    keys(root, q);
    return q;
}

private void keys(Node x, Queue<Key> q) {
    if (x==null) return;
    keys(x.left, q);
    q.add(x.key);
    keys(x.right,q);
}
```

```
public class MyAVLMapClient {
…

public static void main(String[] args) {
    MySortedMap<Integer, Integer> map1= new MyAVLMap<Integer, Integer>();
    MySortedMap<Integer, String> map2 = new MyAVLMap<Integer, String>();
    MySortedMap<String, String> map3 = new MyAVLMap<String, String>();

   Random rand = new Random();

  //populate the maps with 20 random keys and values each

   for (int i=0; i<20; i++) {
      int rand_int1 = rand.nextInt(1000);
      map1.put(rand_int1, 2*rand_int1);
      map2.put(rand_int1, "XXX"+rand_int1);
      map3.put(getRandomAlphaString(4), getRandomAlphaString(10));
      }

   System.out.println("Keys of map1 in ascending order: ");
   for(Integer i:map1.getKeys()){
         System.out.print(" "+i);
    }

…
}
```

Source code

- The simple IntegerAVL:
- IntegerAVL.java

- The MySortedMap abstract data type:
- MySortedMap.java
- MyAVLMap.java
- MyAVLMapClient.java