CS 600.226: Data Structures

Michael Schatz

Oct 29, 2018

Lecture 26. BSTs and AVL trees

HW6

Assignment 6: Setting Priorities

Out on: October 26, 2018
Due by: November 2, 2018 before 10:00 pm
Collaboration: None
Grading:

- Packaging 10%,
- Style 10% (where applicable),
- Testing 10% (where applicable),
- Performance 10% (where applicable),
- Functionality 60% (where applicable)

Overview

The sixth assignment is all about sets, priority queues, and various forms of experimental analysis aka benchmarking. You'll work a lot with jaybee as well as with new incarnations of the old Unique program. Think of the former as "unit benchmarking" the individual operations of a data structure, think of the latter as "system benchmarking" a complete (albeit small) application.

Agenda

1. ***Recap on Maps***
2. ***BSTs***
3. ***AVL Trees***

Part 1:Maps

Maps

aka dictionaries

aka associative arrays

```
Mike      -> Malone 323
Peter     -> Malone 223
Joanne     -> Malone 225
Zack      -> Malone 160 suite
Debbie     -> Malone 160 suite
Randal     -> Malone 160 suite
Ron       -> Garland 242
```

Key (of Type K) -> Value (of Type V)

Note you can have multiple keys with the same value,
But not okay to have one key map to more than 1 value

How might you map to more than 1 value?

Maps, Sets, and Arrays

Sets as Map<T, Boolean>

```
Mike       -> True
Peter      -> True
Joanne      -> True
Zack       -> True
Debbie      -> True
Yair       -> True
Ron        -> True
```

Array as Map<Integer, T>

```
0   -> Mike
1   -> Peter
2   -> Joanne
3   ->  Zack
4   ->  Debbie
5   ->  Yair
6   ->  Ron
```

Maps are extremely flexible and powerful,
and therefore are extremely widely used

Built into many common languages: Awk, Python, Perl, JavaScript…

How could maps be used with sparse arrays?

How could maps be used with graphs?

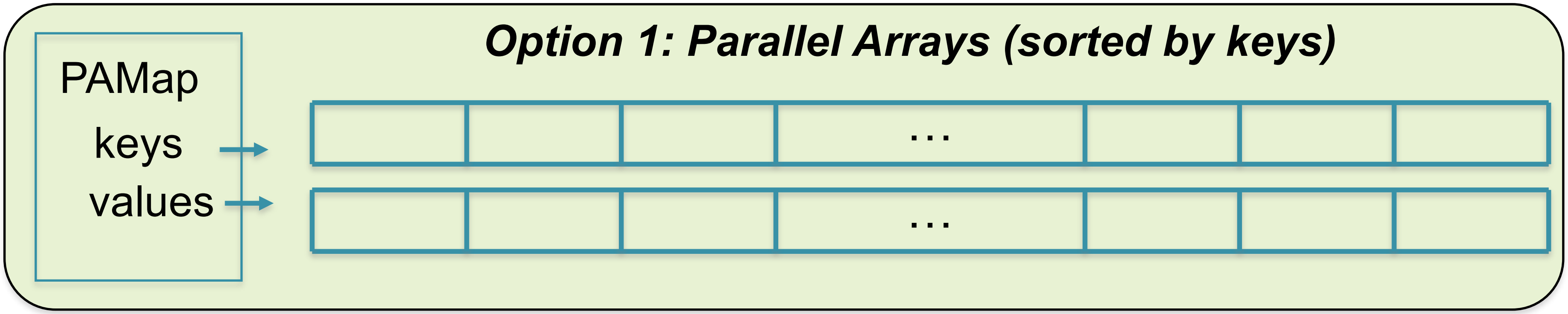
Map Interface v3

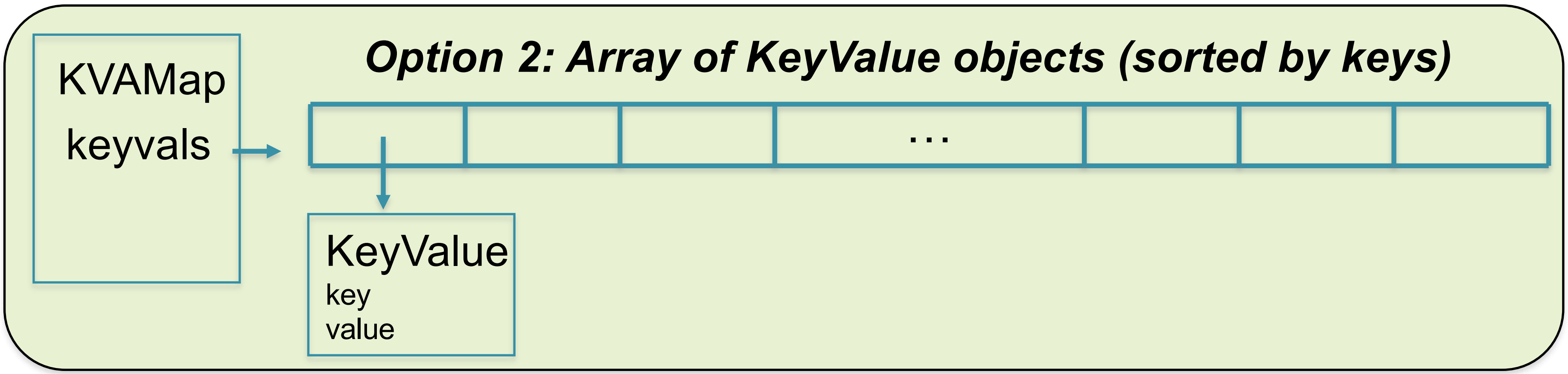
```
public interface OrderedMap<K extends Comparable<K>,V>
          extends Iterable<K>{

    void insert (K k, V v) throws DuplicateKeyException;
    V remove(K k) throws UnknownKeyException;
    void put(K k, V v) throws UnknownKeyException;
    V get(K k) throws UnknownKeyException;
    boolean has(K k);

}
```

Woohoo! Now keys can be compared to each other

How would you implement this interface?

Map Implementation





If the arrays are sorted, we can use binary search =>
get() and has() will run in O(lg n) time!

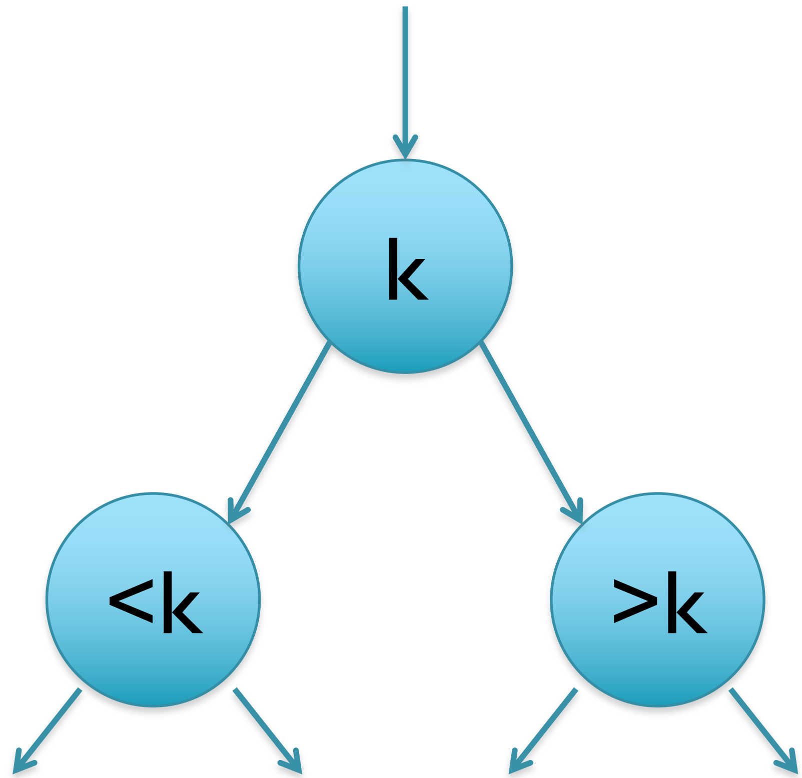
What about insert() or remove()?

O(n) time ☹

Could we do everything in O(lg n) time or faster?

Part 2: Binary Search Tree

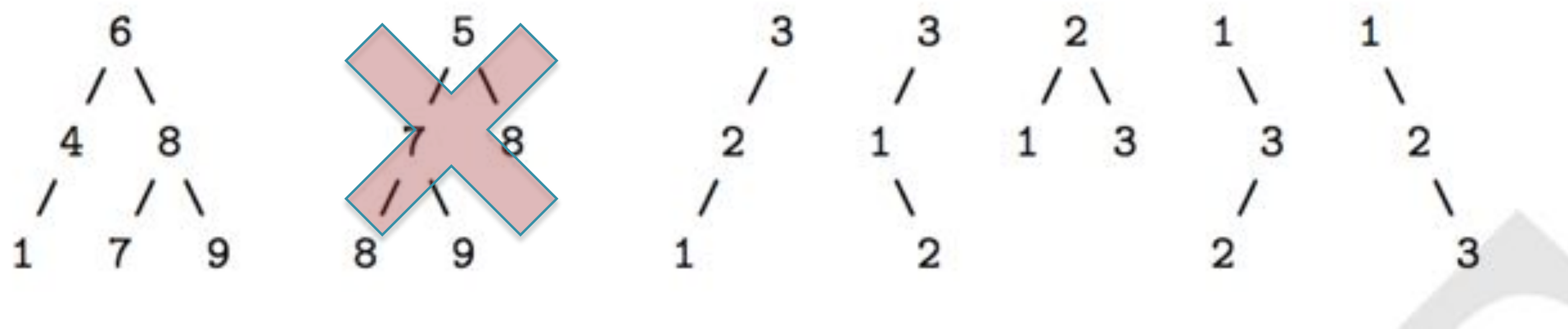
Binary Search Tree



A BST is a binary tree with a special ordering property:
If a node has value k, then the left child (and its descendants) will have values smaller than k; and the right child (and its descendants) will have values greater than k

Can a BST have duplicate values?

Examples



A *B* *C* *D* *E* *F* *G*

Which of these are valid BSTs? (And why?)

What is special about C through G?

Unlike heaps, the shape of the tree is not constrained, but …
What kind of shape would we like the BST to have?

Searching

```
      6
     / \
    4   8
   /   / \
  1   7   9
```

has(7):	compare 6 => compare 8 => found 7
has (2):	compare 6 => compare 4 => compare 1 => not found!

What is the runtime for has() ?

Searching

```
      6
     / \
    4   8
   /   / \
  1   7   9
```

Recursive

```
search(tree, key):

  if tree is empty:
    return false

  if key == tree.key:
    return true
  elif key < tree.key:
    return search(tree.left, key)
  else:
    return search(tree.right, key)

has(key):
   search(root, key)
```

Iterative

```
has(key):
  tree = root

  while tree is not empty

    if key == tree.key:
      return true
    elif key < tree.key:
      tree = tree.left
    else:
      tree = tree.right

  return false;
```

Which version do you like better? Why?

Constructing

```
      i(7)      i(5)       i(8)        i(1)          i(3)           i(4)            i(9)
empty ---->   7 ---->   7  ---->   7   ---->     7   ---->      7   ---->      7    ---->      7
                       /          / \           / \            / \            / \             / \
                      5          5   8         5   8          5   8          5   8           5   8
                                              /              /              /               /     \
                                             1              1              1               1       9
                                                             \              \               \
                                                              3              3               3
                                                                              \               \
                                                                               4               4
```

Note the shape of a general BST will depend on the order of insertions

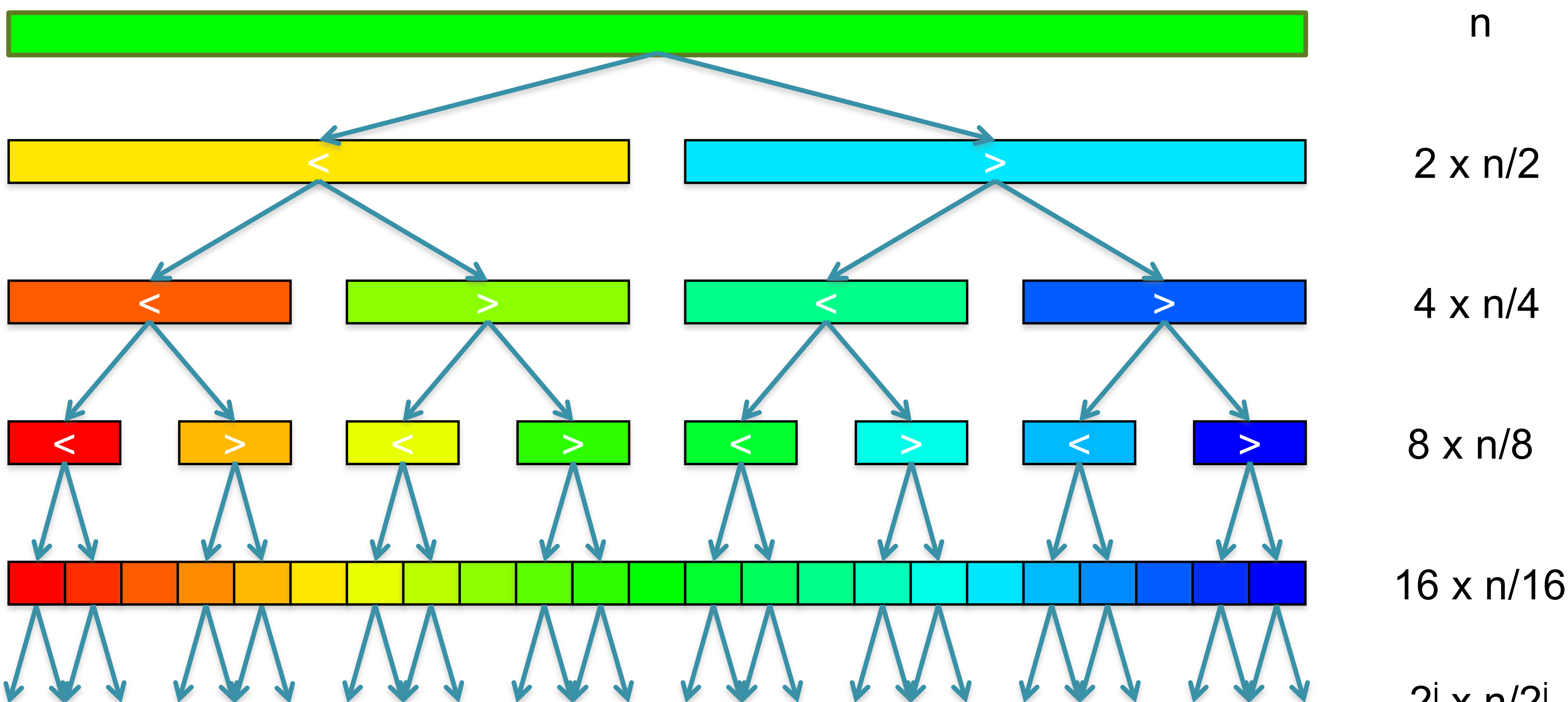
What is the “worst” order for constructing the BST?

What is the “best” order for constructing the BST?

What happens for a random ordering?

Binary Search

Binary Search (and balanced BSTs) are fast because we split the range in half each time.

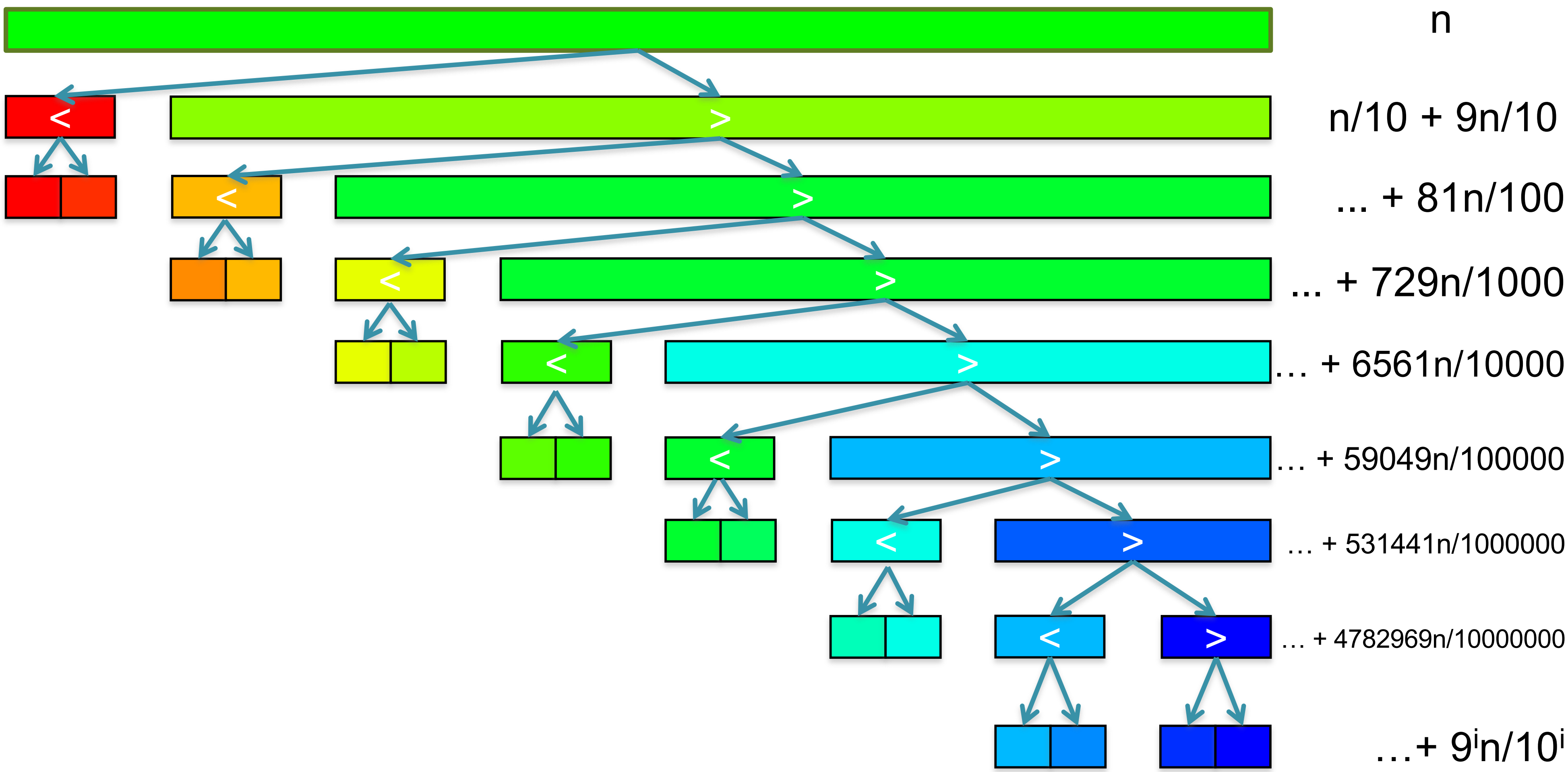


[How many times can we split a list in half?]

lg n ☺

Binary Search

What if we miss the median and do a 90/10 split instead?



[How many times can we cut 10% off a list?]

90% binary search

- 90/10 split runtime analysis

Find smallest x s.t.

$$(9/10)^x n \leq 1$$

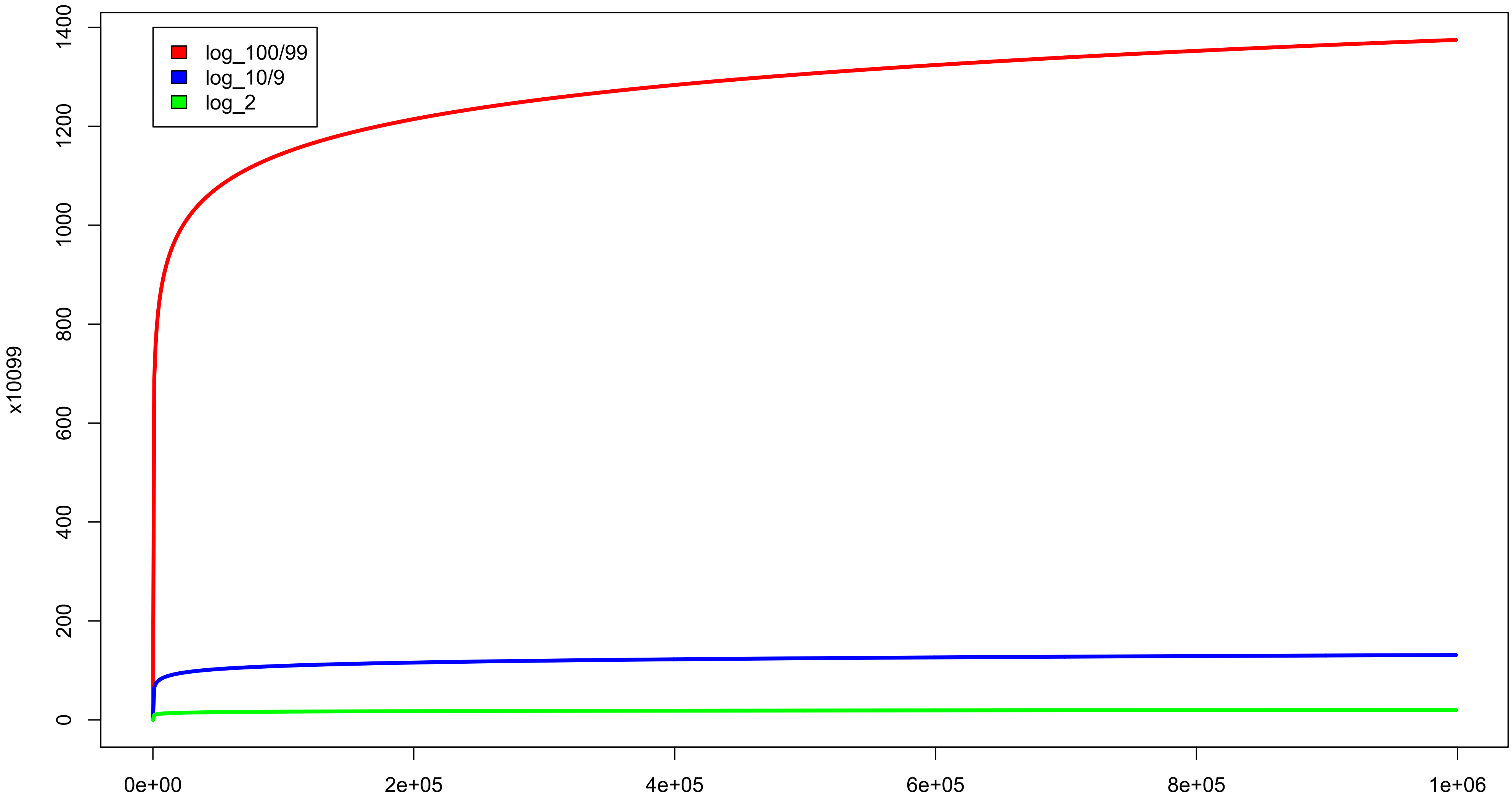
$$(10/9)^x \geq n$$

$$x \geq \log_{10/9} n$$

- If we randomly pick a pivot, we will get at least a 90/10 split with at least 90% probability => $O(\log_{10/9} n)$

- If we randomly pick a pivot, we will get at least a 99/100 split with at least 99% probability => $O(\log_{100/99} n)$

99% binary search



Everything is “okay” (but ~100x slower) as long as we always slice off 1% of the list

What happens if we only slice off 1 item?

How would this occur?

Removing

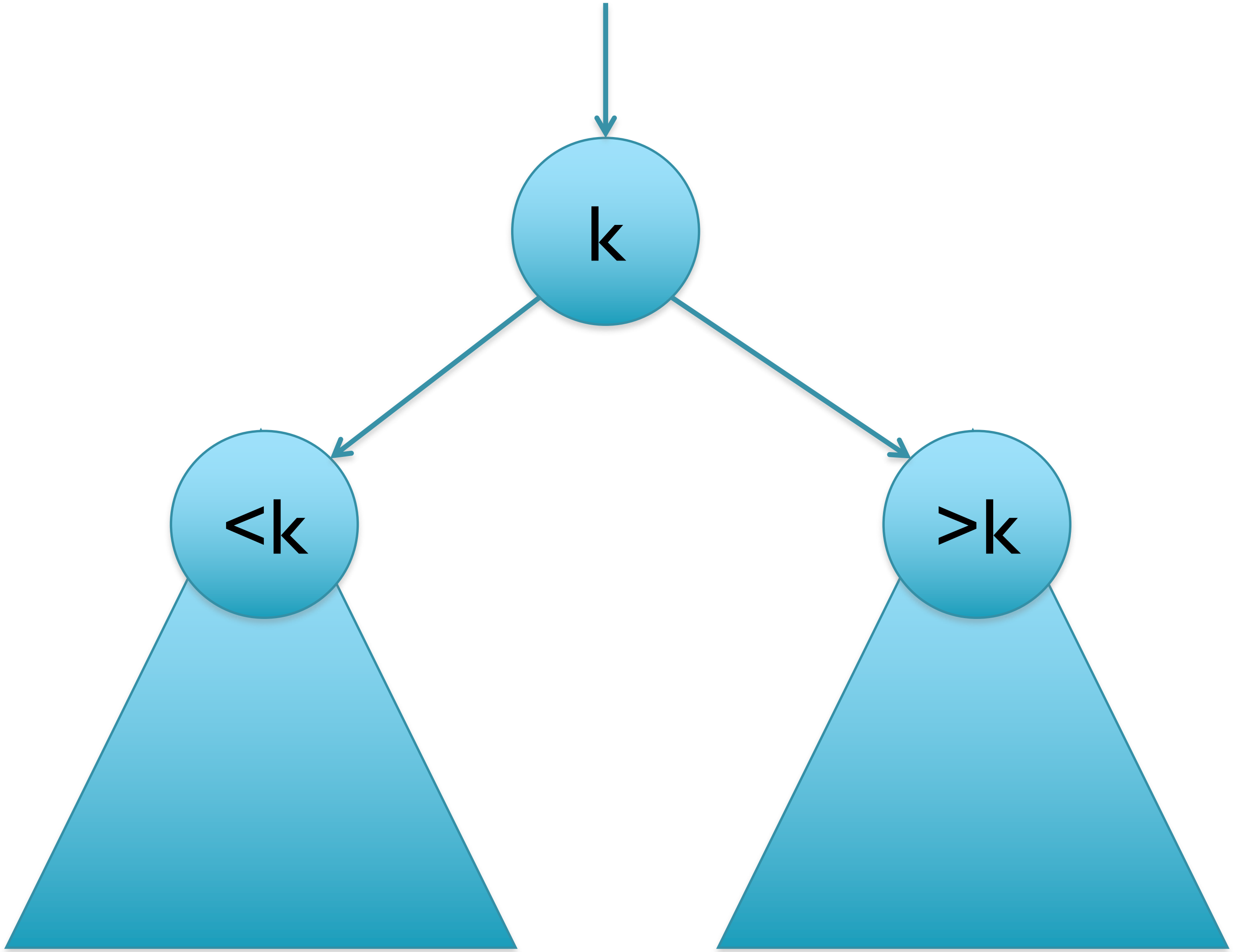
```
       6          r(7)       6          r(8)        6          r(6)
      / \         --->      / \         --->       / \         --->
     3   8                 3   8                  3   9
    / \ / \               / \   \                / \
   1  4 7  9             1  4    9              1  4
```

Remove leaf is easy

Remove with only one child Is easy

Remove internal node ?

Binary Search Tree

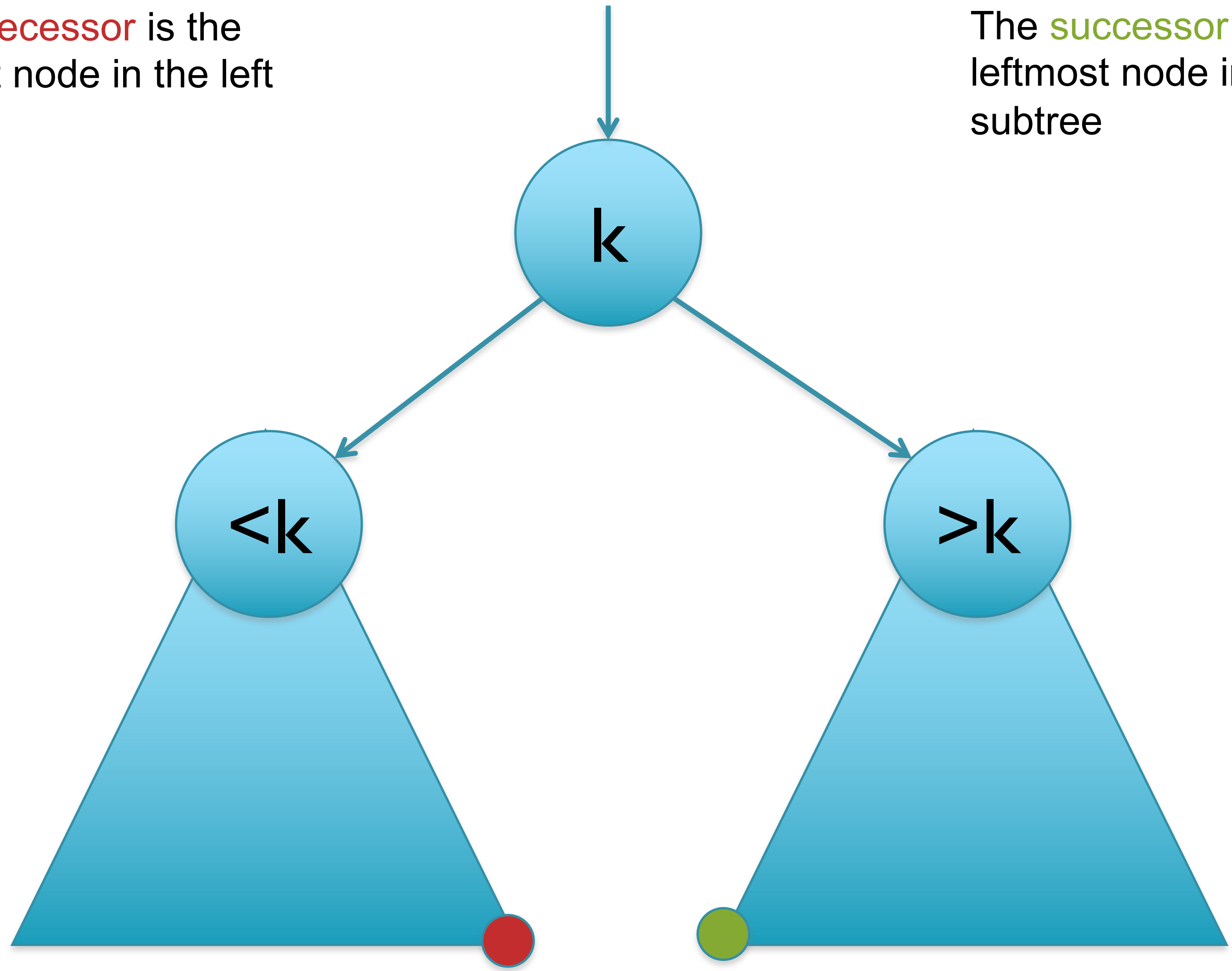


Where is k’s immediate predecessor?
Where is k’s immediate successor?

Binary Search Tree

The predecessor is the rightmost node in the left subtree

The successor is the leftmost node in the right subtree



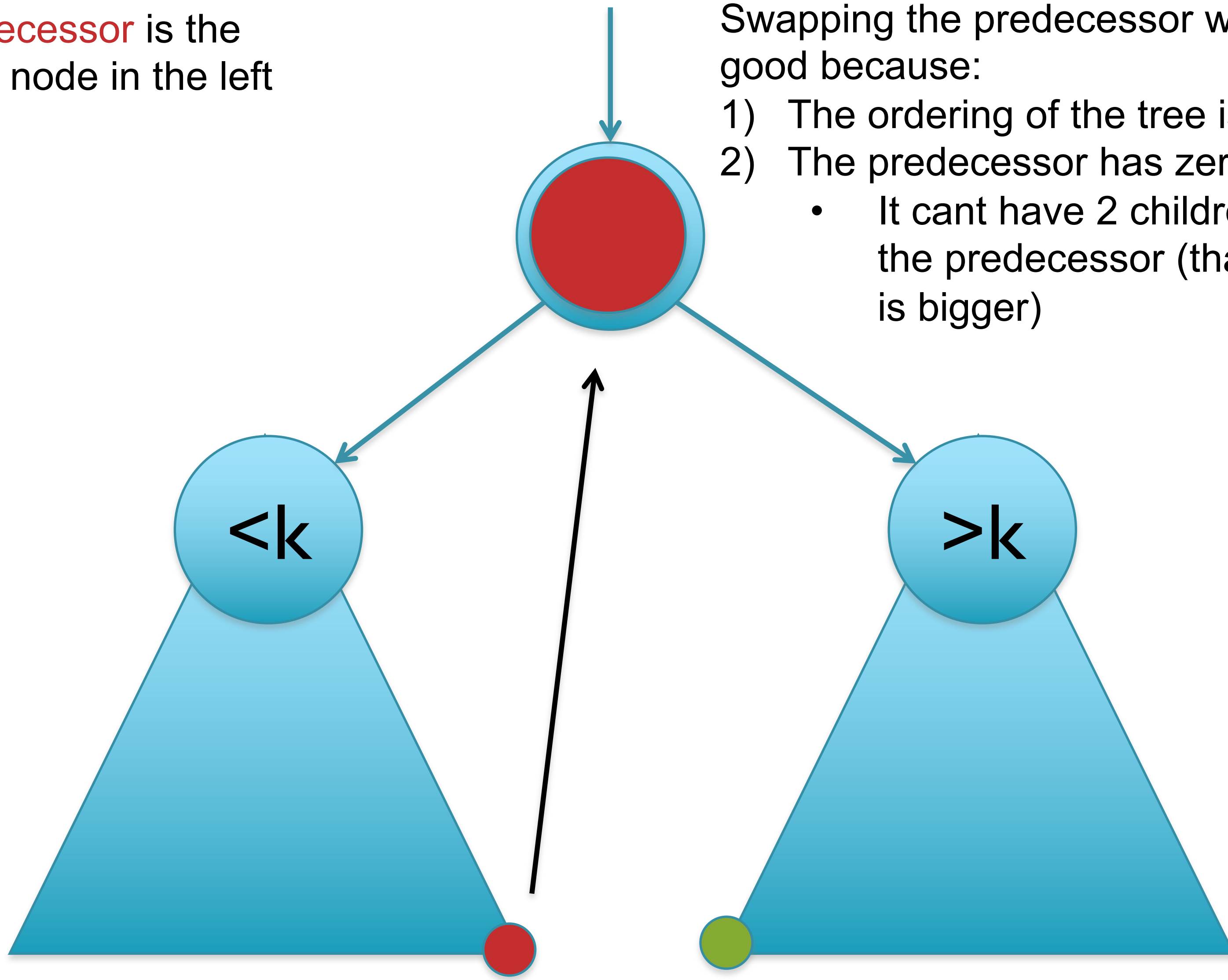
Where is k’s immediate predecessor?
Where is k’s immediate successor?

Binary Search Tree

The predecessor is the rightmost node in the left subtree

Swapping the predecessor with root is good because:

1) The ordering of the tree is preserved
2) The predecessor has zero or 1 child
   - It cant have 2 children or it is not the predecessor (that right child is bigger)



Where is k's immediate predecessor?
Where is k's immediate successor?

Removing

```
       6          7         6          8         6          6         4      or      9
     /   \       -->      /   \       -->      /   \       -->      /   \          /
    3     8              3     8              3     9              3     9        3
   / \   / \            / \     \            / \                  /              / \
  1   4 7   9          1   4     9          1   4                1              1   4
```

Remove
leaf
is easy

Remove
with only
one child
Is easy

Remove
internal
node
?

1. Find k’s successor (or predecessor) and swap values with k
2. Remove the node we got that key from (easy, since it has at most one child)

BinarySearchTreeMap (I)

```
      6
     / \
    3   9
   / \
  1   4
```

```java
import java.util.Iterator;

public class BinarySearchTreeMap<K extendsComparable<K>,V>
                      implements OrderedMap<K, V> {

    private static class Node<K, V> {
        Node<K, V> left, right;
        K key;
        V value ;
        Node(K k,V v){
            this.key = k;
            this.value =v;
        }
    }

    private Node<K, V> root;

    public boolean has(K k) {
        return this.find(k) != null;
    }
```

Static nested class
Sometimes convenient to use child[0] and child[1]

has() calls find()

BinarySearchTreeMap (2)

```
      6
     / \
    3   9
   / \
  1   4
```

```
private Node<K, V> find(K k) {
    Node<K, V> n = this.root;
    while (n != null) {
        int cmp = k.compareTo(n.key);
        if (cmp < 0){
            n = n.left;
        } else if (cmp > 0){
            n = n.right;
        } else {
            return n;
        }
    }
    return null;
}

public void put(K k, V v) throws UnknownKeyException {
    Node<K, V> n = this.findForSure(k);
    n.value = v;
}

public V get (K k) throws UnknownKeyException {
    Node<K, V> n = this.findForSure(k);
    return n.value;
}
```

find() iteratively walks the tree, returns null if not found

put()/get() use a special findForSure() method

BinarySearchTreeMap (3)

```
      6
     / \
    3   9
   / \
  1   4
```

```
private Node<K, V> findForSure(K k) throws UnknownKeyException {
    Node<K, V> n = this.find(k);
    if (n == null) {
        throw new UnknownKeyException();
    }
    return n;
}

public void insert (K k, V v) throws DuplicateKeyException{
    this.root = this.insert(this.root, k, v);
}

private Node<K, V> insert(Node<K, V> n, K k, V v) {
    if (n == null) {
        return new Node<K, V>(k, v);
    }
    int cmp = k.compareTo(n.key);
    if (cmp < 0){
        n.left = this.insert(n.left, k, v);
    } else if (cmp > 0){
        n.right = this.insert(n.right, k, v);
    } else {
        throw new DuplicateKeyException();
    }
    return n;
}
```

Just like find() but throws exception if not there

Recurse to right spot, add the new node, and return the modified tree after insert is complete

(n.left or n.right may be reset to same value for nodes that don't change)

BinarySearchTreeMap (4)

```
      6
     / \
    3   9
   / \
  1   4
```

```java
public V remove(K k) throws UnknownKeyException {
    V value = this.get(k);
    this.root = this.remove(this.root, k);
    return value;
}

private Node<K, V> remove(Node<K, V> n, K k) throws UnknownKeyException {
    if (n == null) {
        throw new UnknownKeyException();
    }

    int cmp = k.compareTo(n.key);

    if (cmp < 0){
        n.left = this.remove(n.left , k);
    } else if (cmp > 0){
        n.right = this.remove(n.right, k);
    } else {
        n = this.remove(n);
    }

    return n;
}
```

First get() it so we can return the value, then actually remove

Recurse to right spot, then call the overloaded private remove() function

BinarySearchTreeMap (5)

```
      6
     / \
    3   9
   / \
  1   4
```

```java
private Node<K, V> remove(Node<K, V> n) {
   // 0 and 1 child
   if (n.left == null) {
        return n.right;
   }

   if (n.right == null) {
        return n.left;
   }

   // 2 children
   Node<K, V> max = this.max(n.left);
   n.left = this.removeMax(n.left);
   n.key = max.key;
   n.value = max.value;
   return n;
}

private Node<K, V> max(Node<K, V> n) {
   while (n.right != null) {
        n = n.right ;
   }
   return n;
}
```

Easy cases

Find the max of the subtree rooted on the left child -> its predecessor

Just keep walking right as far as you can

BinarySearchTreeMap (6)

```
      6
    /   \
   3     9
  / \
 1   4
```

```java
  private Node<K, V> removeMax(Node<K, V> n) {
     if (n.right == null) {
         return n.left;
     }
     n.right = removeMax(n.right);
     return n;
  }

  public Iterator <K> iterator () {
     return null;
  }

  public String toString () {
     return this.toStringHelper(this.root);
  }

  private String toStringHelper (Node<K, V> n) {
     String s = "(";
     if (n != null) {
         s += this.toStringHelper(n.left);
         s += "" + n.key + ": " + n.value;
         s += this . toStringHelper (n.right);
     }
     return s + ")";
}
```

Fix the pointers to maintain BST invariant

Flush out rest of class: recursively traverse the tree to fill up an ArrayList<K> and return its iterator ☺

Part 3: AVL Trees

Constructing

```
        7      5      8         1           3           4           9
empty --->  7 ---> 7 --->  7   --->   7    --->   7    --->   7    --->   7
                  /       / \          / \         / \         / \         / \
                 5       5   8        5   8       5   8       5   8       5   8
                                     /           /           /           /     \
                                    1           1           1           1       9
                                                 \           \           \
                                                  3           3           3
                                                               \           \
                                                                4           4
```

Note the shape of a general BST will depend on the order of insertions

We hope for O(lg(n)) height, but can end up being O(n) for nearly-sorted values

We (probably) cant change the order that we see data, but what can we change?

AVL Tree

```
         7       5       8           1           3           4            9
empty  ---->  7 ----> 7 ----> 7   ---->     7   ---->     5   ---->     5    ---->     5
                     /       / \           / \           / \           / \           /     \
                    5       5   8         5   8         3   7         3   7         3       8
                                         /             /     \       / \   \       / \     / \
                                        1             1       8     1   4   8     1   4   7   9
```

*not an actual AVL tree construction

Self-balancing binary search tree

Named after the two Russian inventors: Adelson-Velskii and Landis

First published in 1962, one of the first “efficient data structures”

Balanced Trees

Note that we cannot require a BST to be perfectly balanced:

```
1        2    1           3          3       2       1       1
        /      \         /          /       / \       \       \
       1        2       2          1       1   3       3       2
                       /            \                 /         \
                      1              2               2           3
```

Balanced | Not Balanced | Not Balanced | Not Balanced | Not Balanced | Balanced | Not Balanced | Not Balanced

Since neither tree with 2 keys is perfectly balanced we cannot insist on it

AVL Condition:
For every node n, the height of n's left and right subtree's differ by at most 1

Bonus: Counting Binary Search Trees

How many valid binary search trees are there with n nodes?

```
 1
```

C(1) = 1

```
    2      1
   /        \
  1          2
```

C(2) = 2

```
      3          3         2         1         1
     /          /         / \         \         \
    2          1         1   3         3         2
   /            \                     /           \
  1              2                   2             3
```

C(3) = 5

C(4) = 14   C(5) = 132   C(6) = 429

The number of binary trees can be calculated using the Catalan number.

Recursive solution:

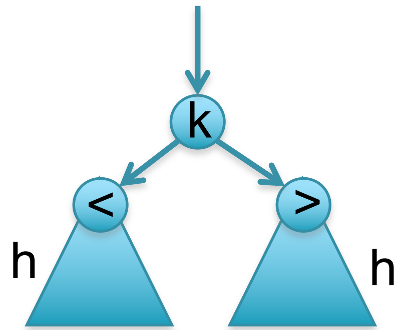
Number of binary search trees = (Number of Left binary search sub-trees) * (Number of Right binary search sub-trees) * (Ways to choose the root)

$$f(n) = \sum_{i=1}^{n} f(i-1)f(n-i) = \frac{(2n)!}{(n+1)!\,n!} \qquad C_n \sim \frac{4^n}{n^{3/2}\sqrt{\pi}}$$

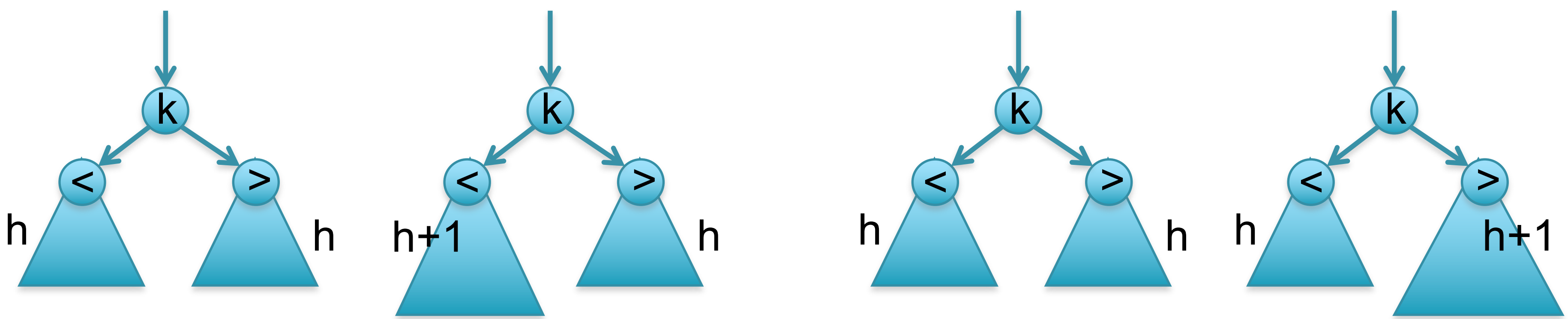
NOT ON FINAL EXAM ☺

Maintaining Balance

Assume that the tree starts in a balanced state:



Inserting a new value can lead to 1 of 4 possible outcomes:



Insert left,
Same height

Insert left,
Height + 1

Insert right,
Same height

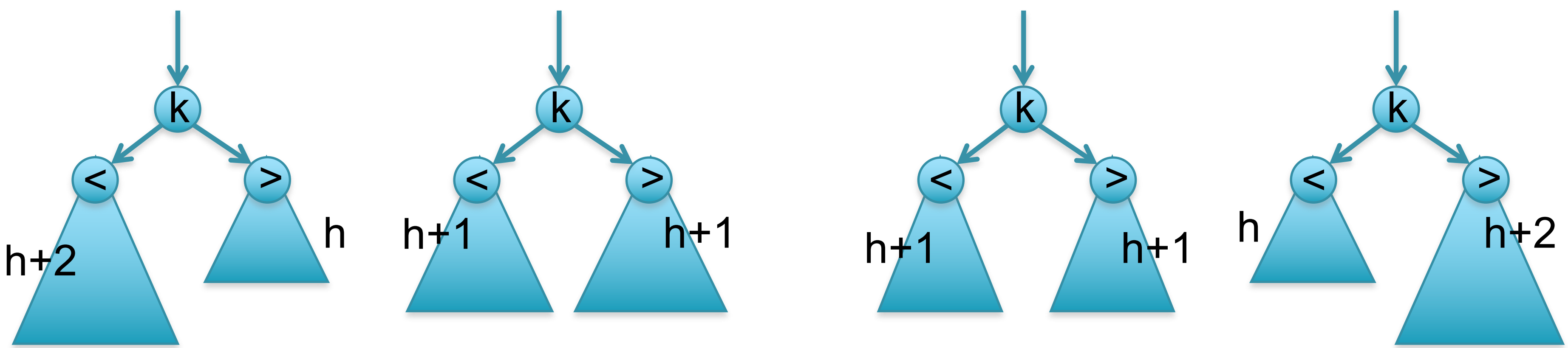
Insert right,
Height + 1

Meh, AVL property still holds

Maintaining Balance

Assume that the tree starts in a slightly unbalanced state:

Inserting a new value can maintain the subtree heights or 1 of 4 possible outcomes:



Insert left,
AVL Violation!

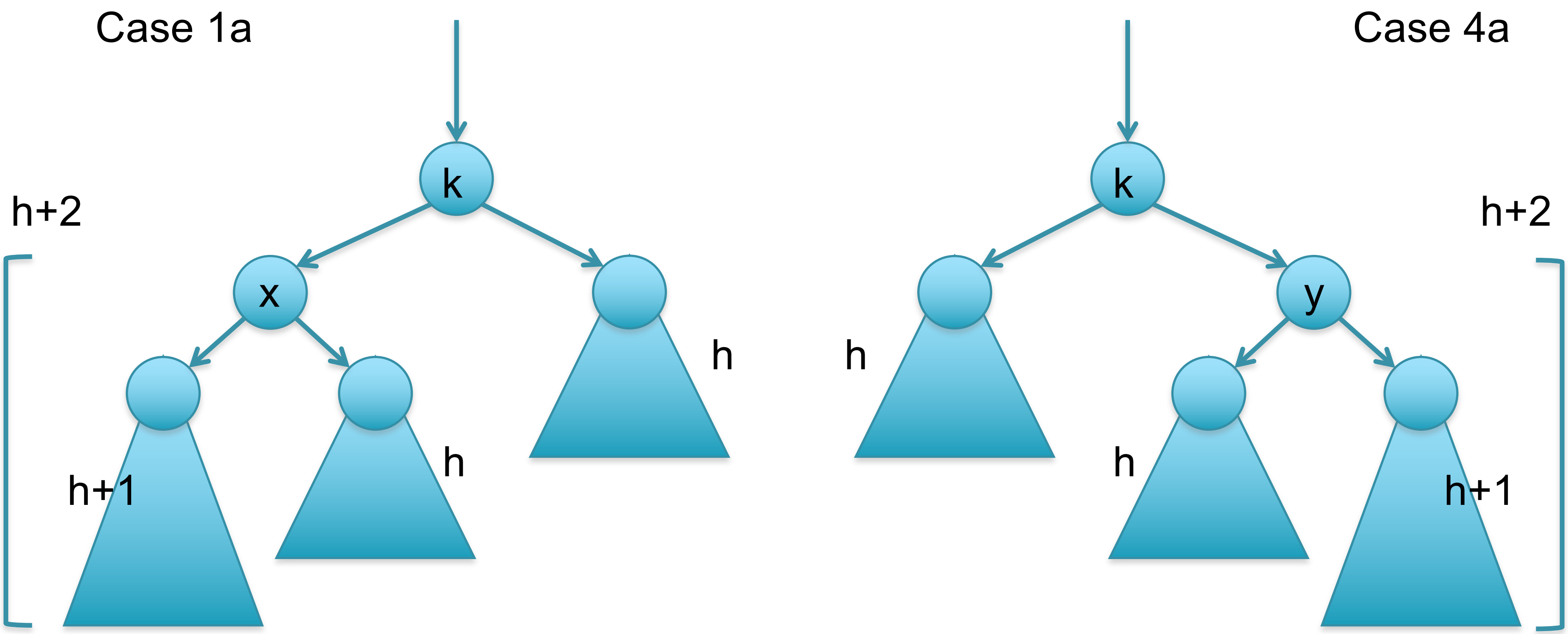
Insert right,
Rebalanced

Insert left,
Rebalanced

Insert right,
AVL Violation!

Woohoo! AVL Violations ;-)

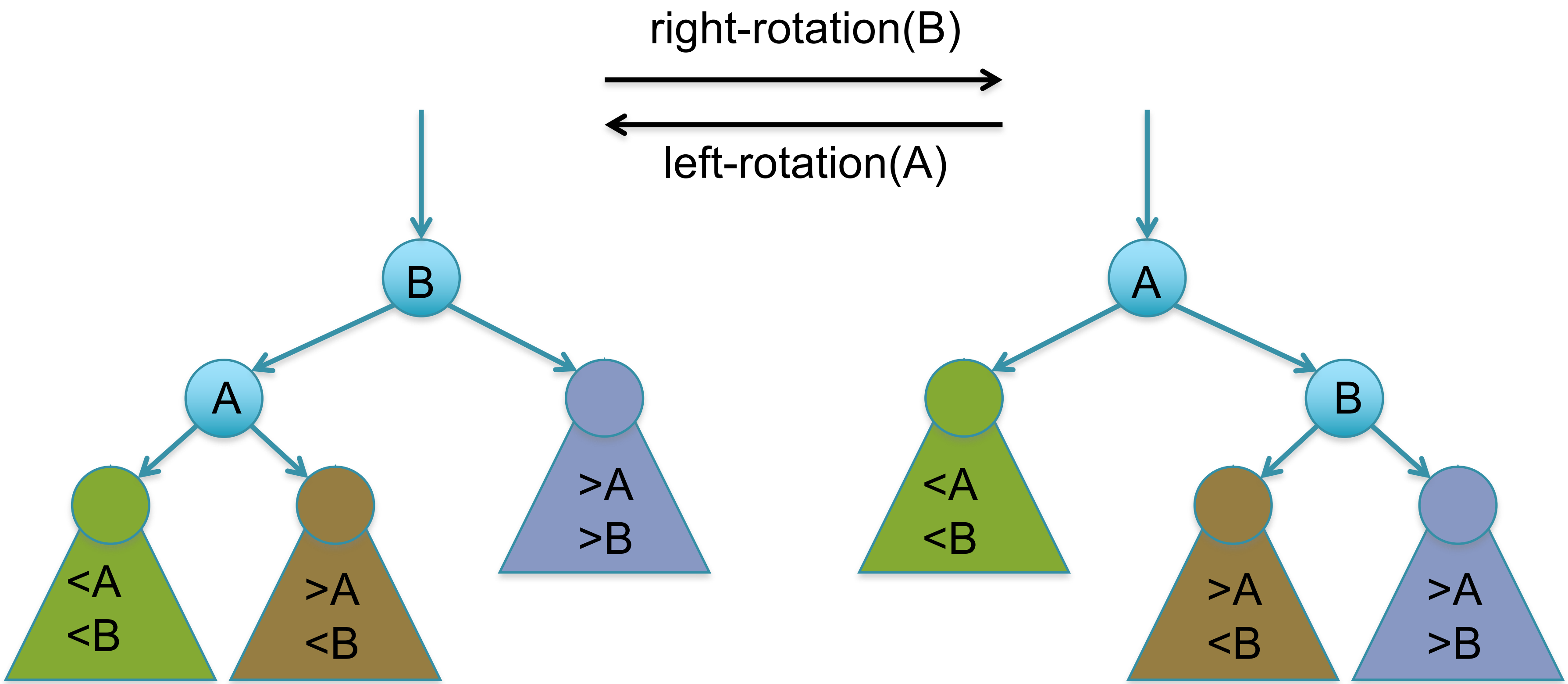
Maintaining Balance



Lets first assume the extra element goes into the leftmost or rightmost subtree

How can we restore balance to the trees?

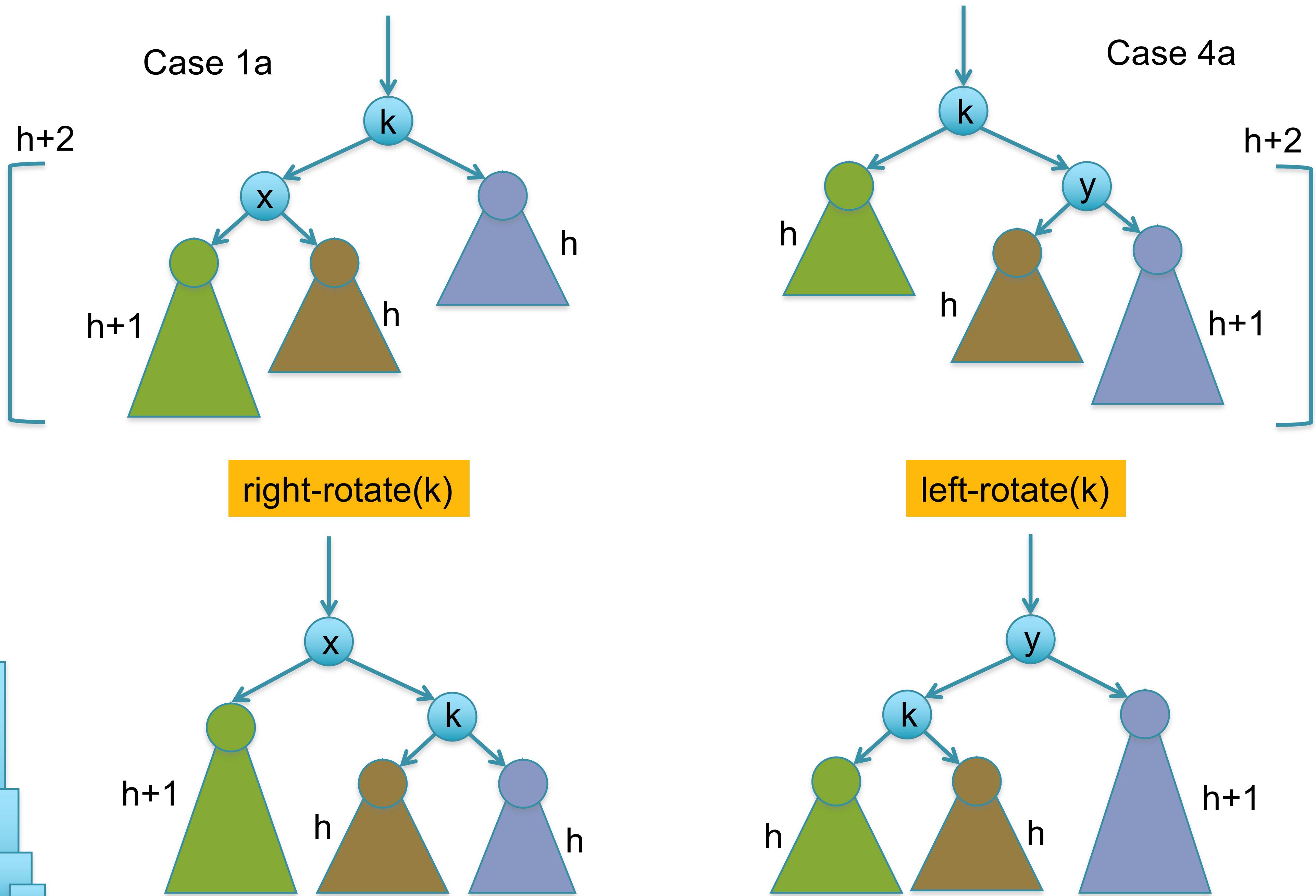
Tree Rotations



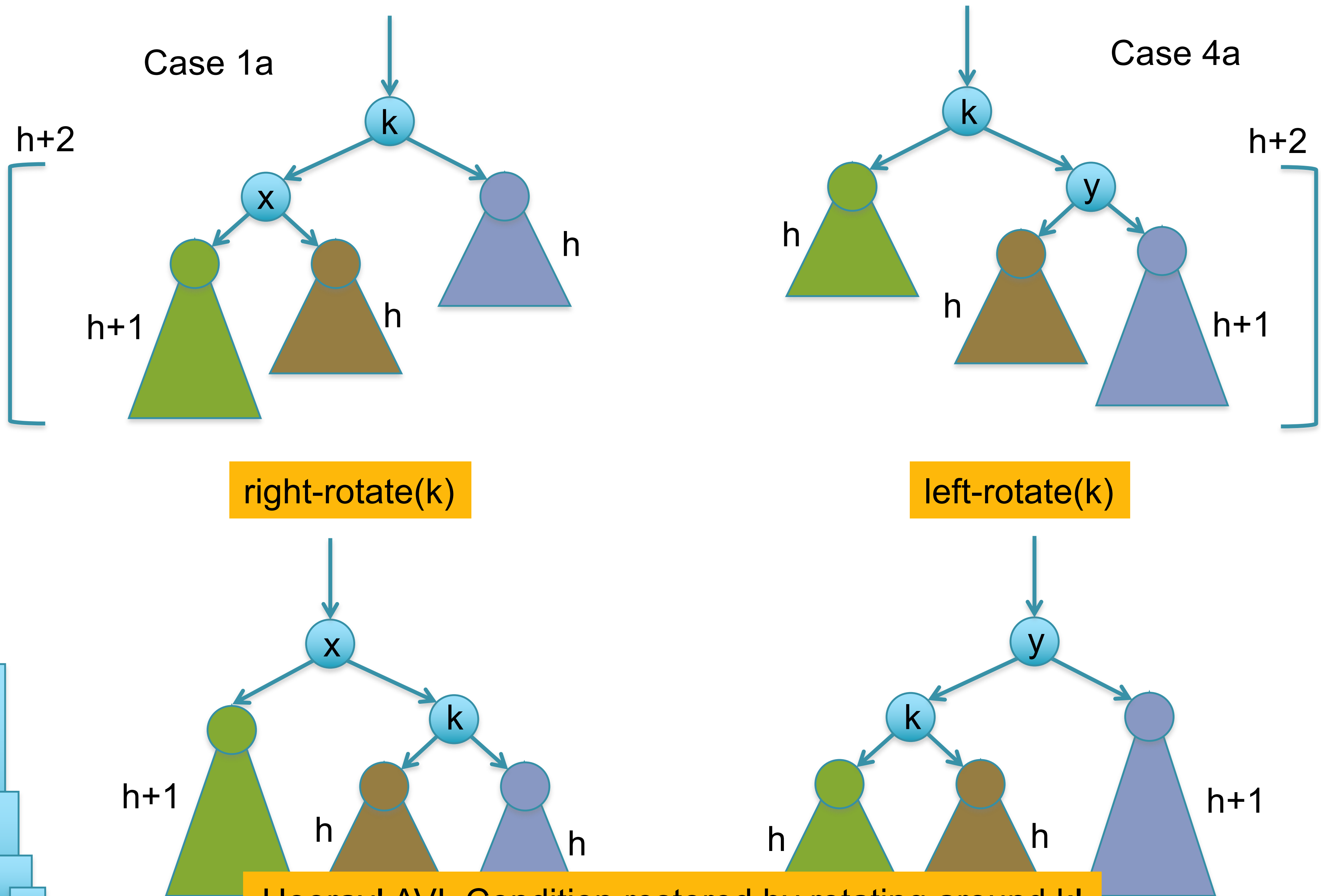
Note: Rotating a BST maintains the BST condition!

Green < A < Brown < B < Purple

Restoring Balance



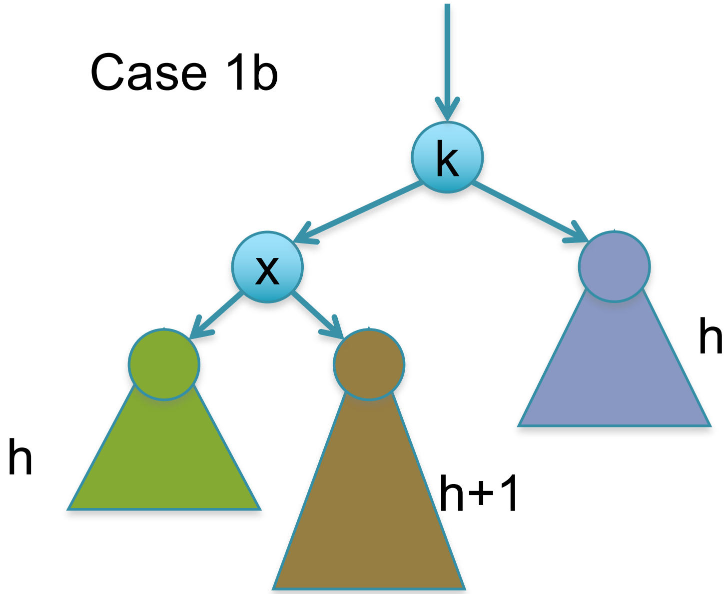
Restoring Balance



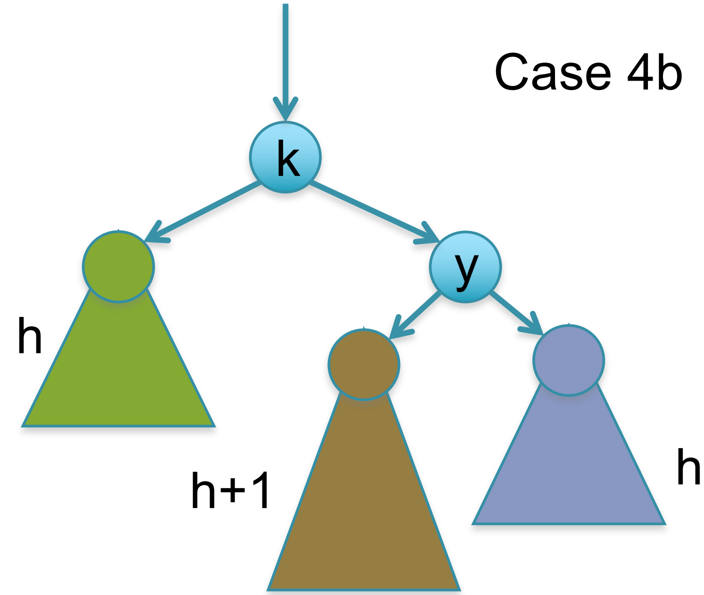
Hooray! AVL Condition restored by rotating around k!

Corner cases

Case 1b



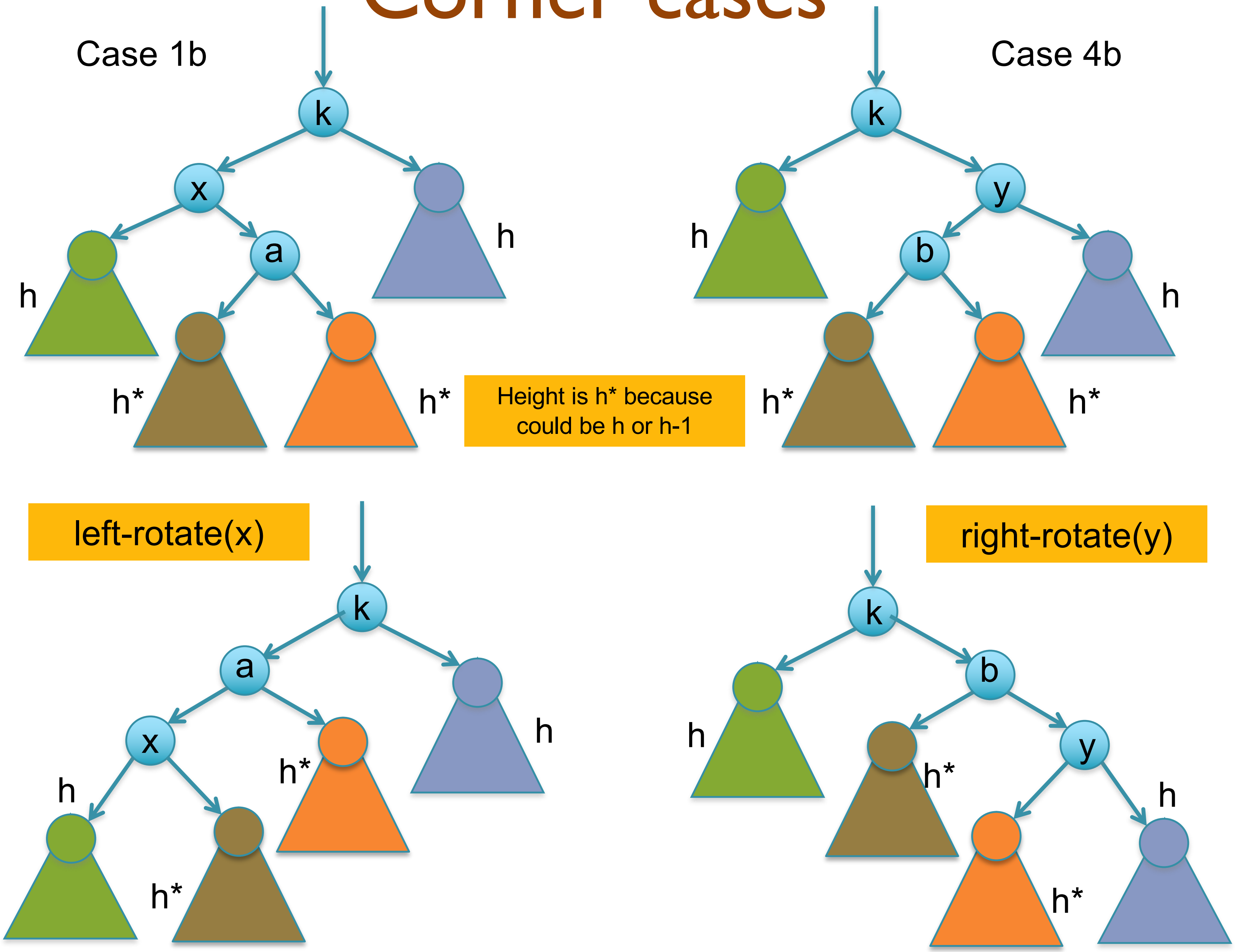
Case 4b



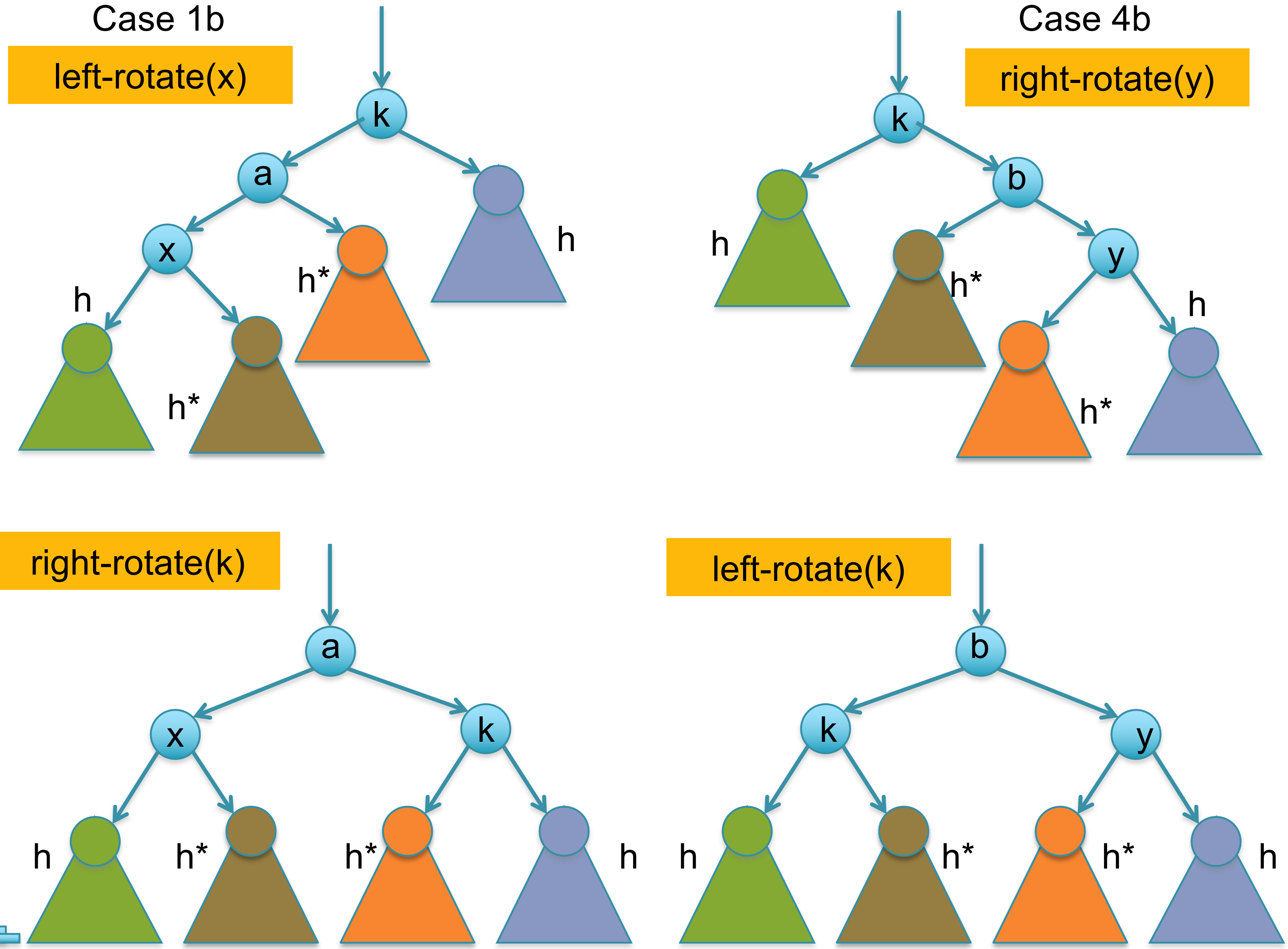
Will rotating around k fix the problem?

What else can we do?

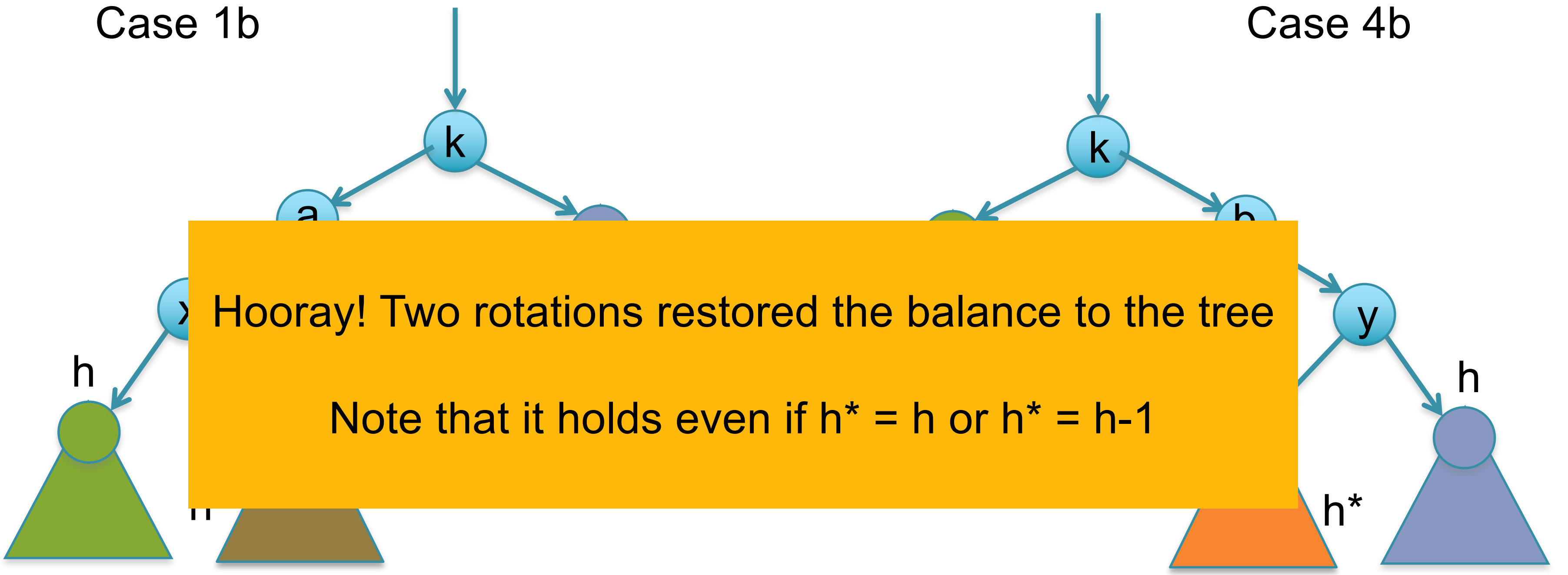
Corner cases



Corner cases

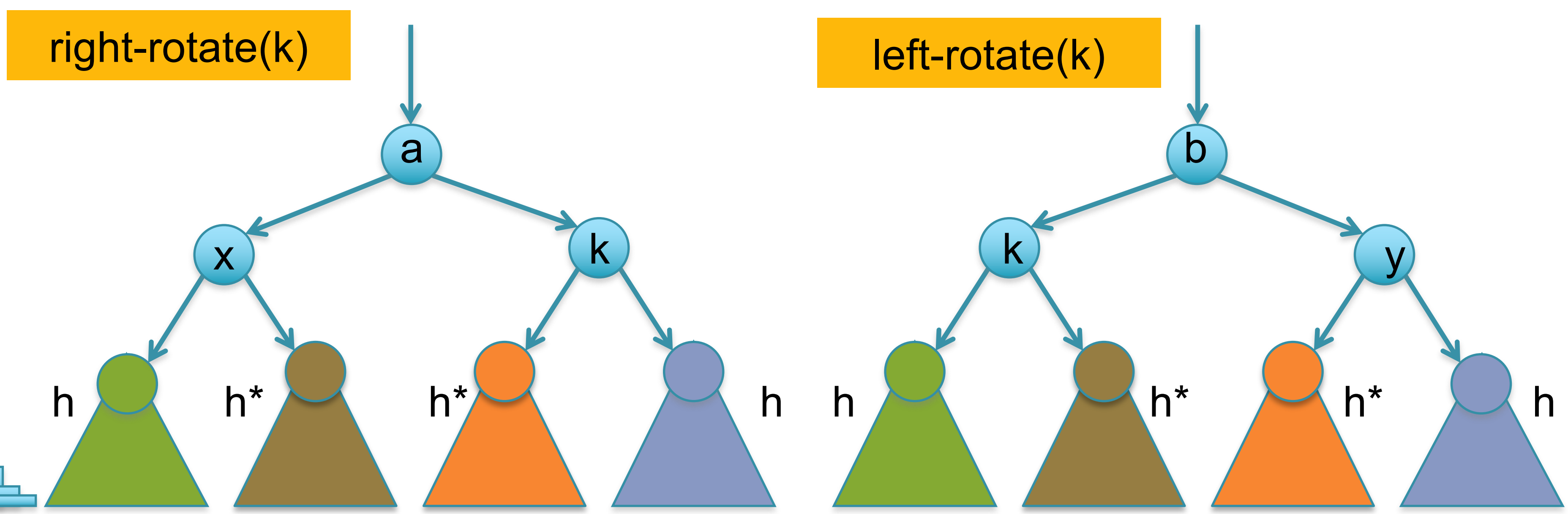


Corner cases



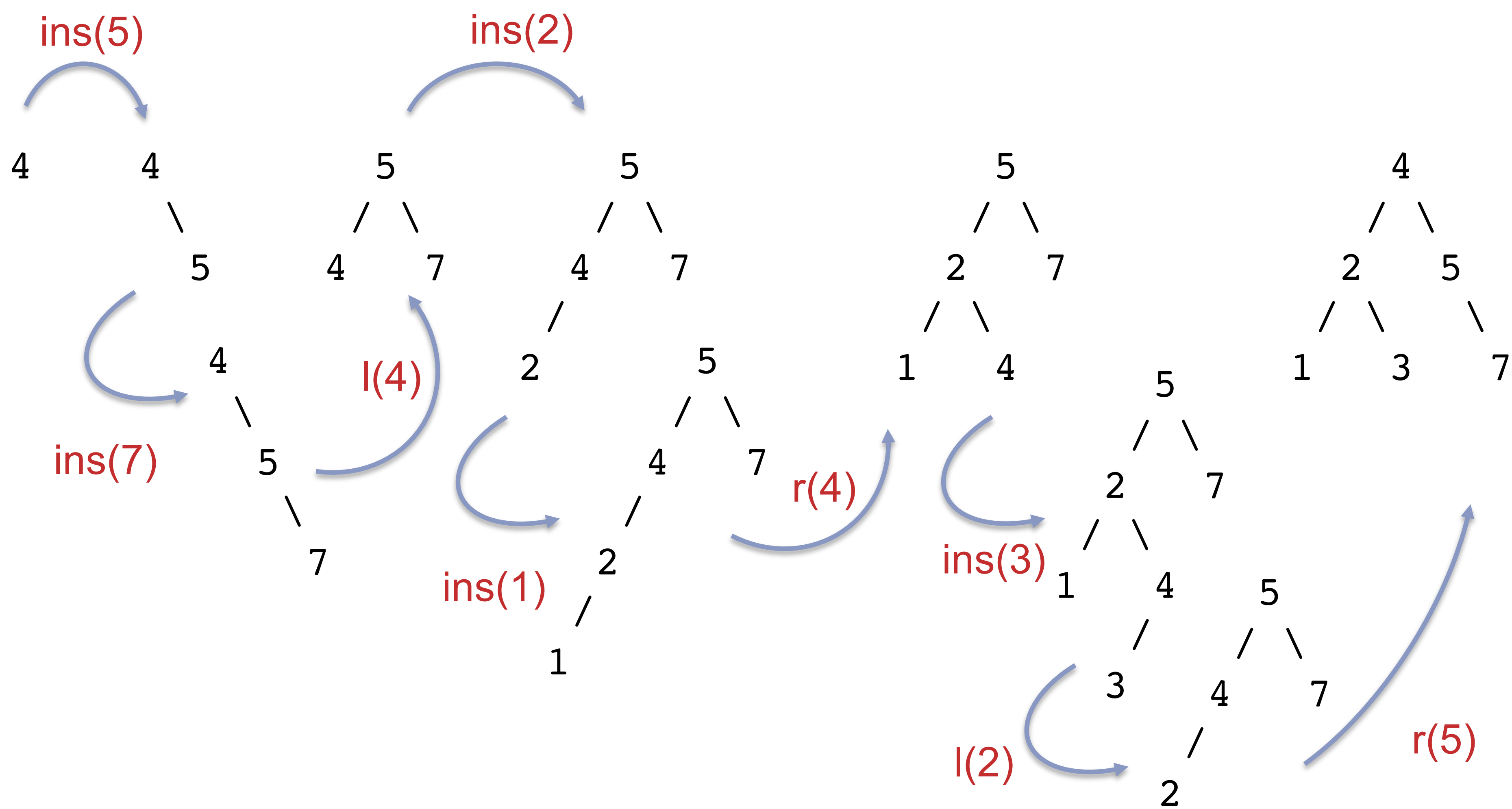
Hooray! Two rotations restored the balance to the tree

Note that it holds even if h* = h or h* = h-1



Complete Example

Insert these values: 4 5 7 2 1 3



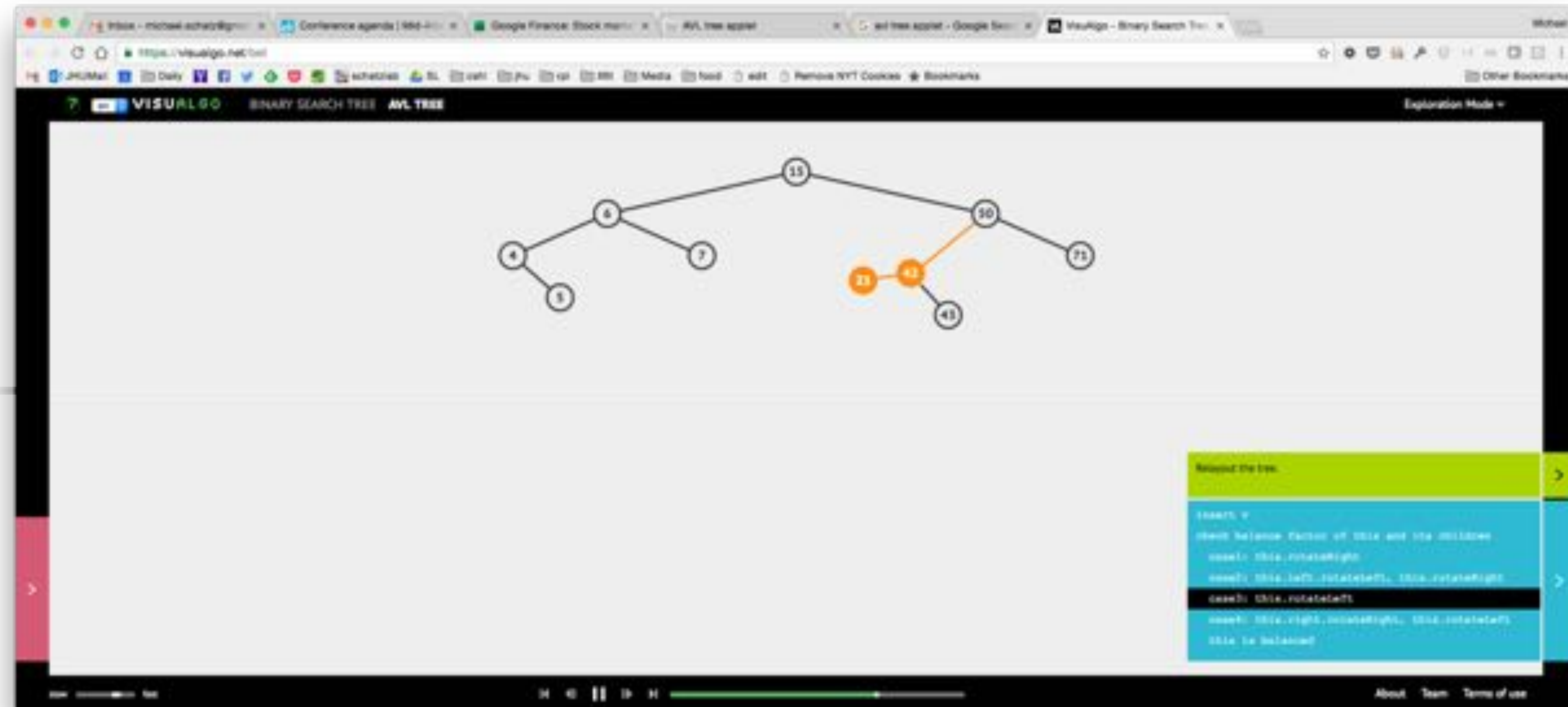
Note:
AVL Violations are fixed bottom up

Implementation Notes

- ***Rotations can be applied in constant time!***
  - Inserting a node into an AVL tree requires O(lg n) time and guarantees O(lg(n)) height
- ***Track the height of each node as a separate field***
  - The alternative is to track when the tree is lopsided, but just as hard and more error prone
  - Don't recompute the heights from scratch, it is easy to compute but requires O(n) time!
  - Since we are guaranteeing the tree will have height lg(n), just use an integer
  - Only update the affected nodes

Check out Appendix B for some very useful tips on hacking AVL trees!

Sample Application



https://visualgo.net/bst

Next Steps

1. Work on HW6
2. Check on Piazza for tips & corrections!