AVL Trees

Cost of the BST Operations

Our Goal

- Develop a data structure that has **guaranteed** O(log n) worst-case complexity for lookup, insert and find_min
  - **always!**

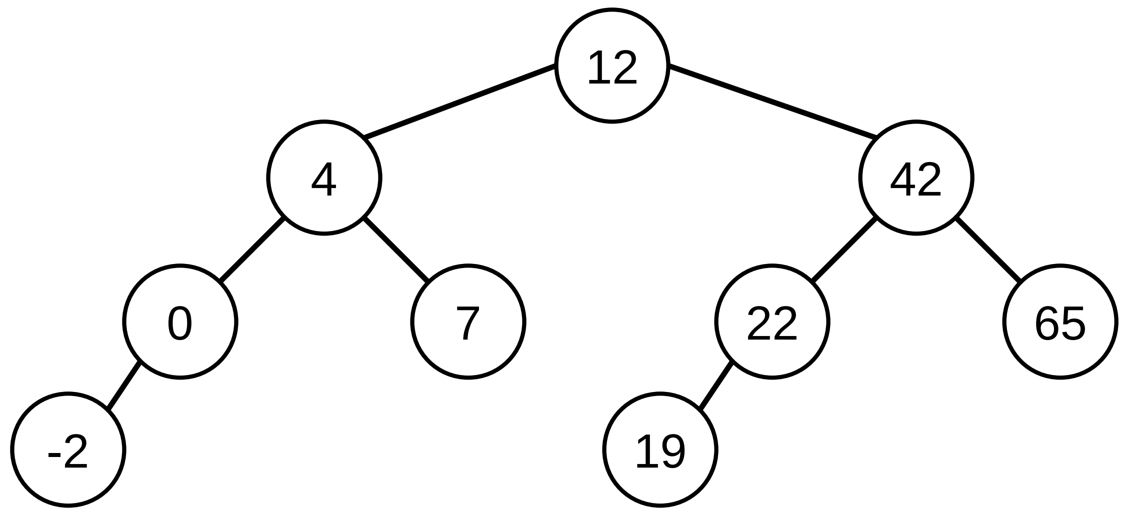
BST?

	Unsorted array	***Array sorted by key***	***Linked list***	***Hash Table***	
lookup	*O(n)*	*O(log n)*	*O(n)*	*O(1)* *average*	*O(log n)*
insert	*O(1)* *amortized*	*O(n)*	*O(1)*	*O(1)* *average and amortized*	*O(log n)*
find_min	*O(n)*	*O(1)*	*O(n)*	*O(n)*	*O(log n)*

- Do binary search trees achieve this?

Complexity

- Do lookup, insert and find_min have O(log n) complexity?
  - Yes, in this tree



  - But we are interested in the **worst-case** complexity

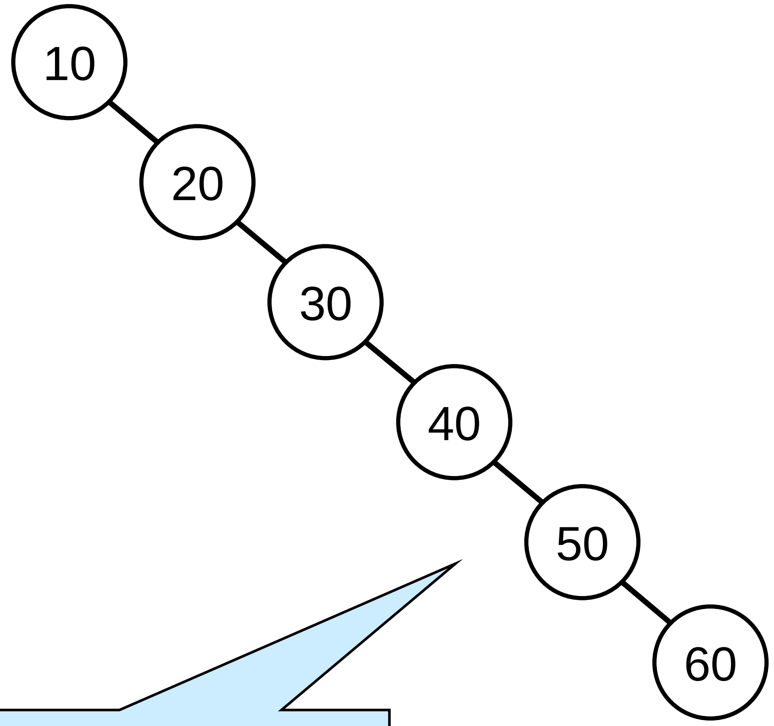
- Do lookup, insert and find_min have *O(log n)* complexity for *every* BST?

Complexity

- *Do lookup, insert and find_min have O(log n) complexity for every BST?*
  - Consider this sequence of insertions into an initially empty BST

```
insert 10
insert 20
insert 30
insert 40
insert 50
insert 60
```

  - It produces this tree:



This tree has degenerated into a linked list!

  - Then to lookup 70, we have to go through all the nodes
    - This is O(n)

- If the insertion sequence is sorted, lookup costs O(n)

Inserting 70 would also cost O(n)

Exercise: find a sequence that yields O(n) cost for find_min

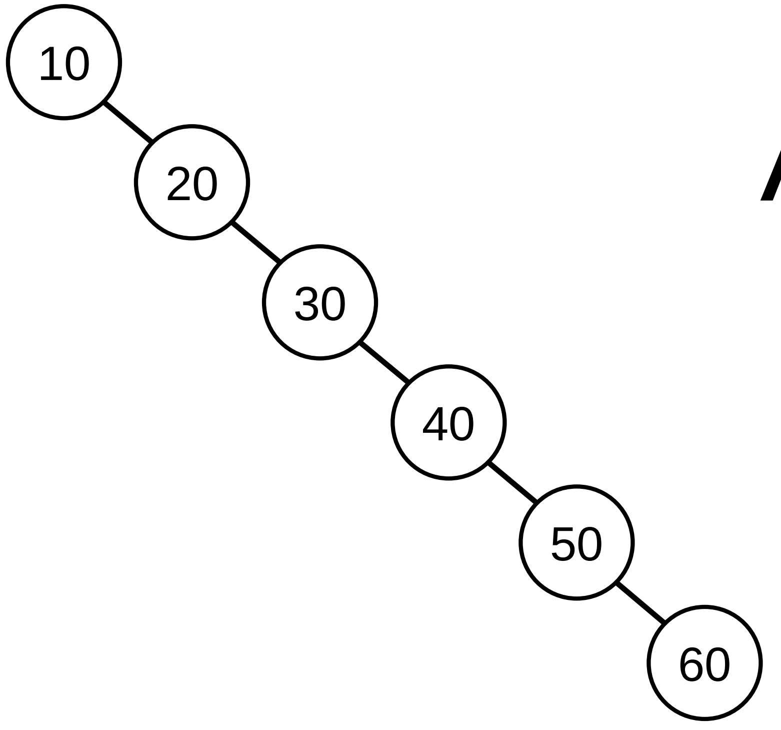
Back to Square One

- Develop a data structure that has **guaranteed** O(log n) worst-case complexity for lookup, insert and find_min
  - **always!**

	Unsorted array	*Array sorted by key*	*Linked list*	*Hash Table*	*BST*	Something else …
lookup	*O(n)*	*O(log n)*	*O(n)*	*O(1)* *average*	*O(n)*	*O(log n)*
insert	*O(1)* *amortized*	*O(n)*	*O(1)*	*O(1)* *average and amortized*	*O(n)*	*O(log n)*
find_min	*O(n)*	*O(1)*	*O(n)*	*O(n)*	*O(n)*	*O(log n)*

- BSTs are **not** the data structure we were looking for
  - What else?

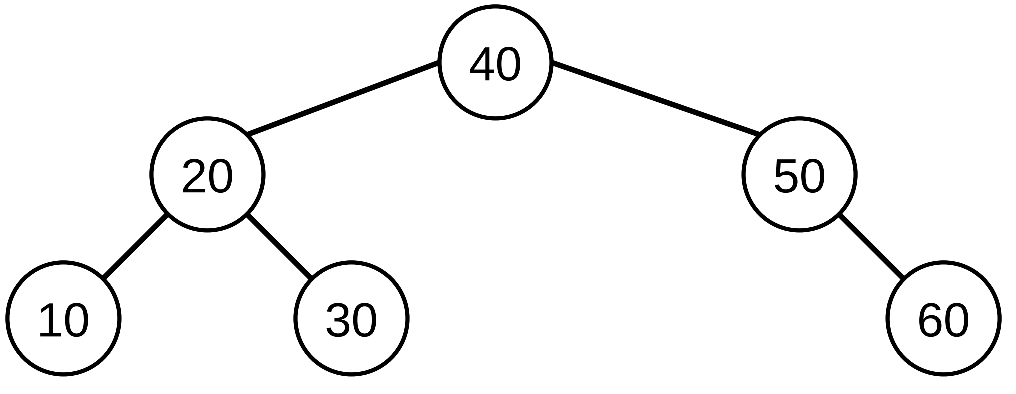
Balanced Trees



An Equivalent Tree

- Is there a BST with the same elements that yields O(log n) cost?

- How about this one?

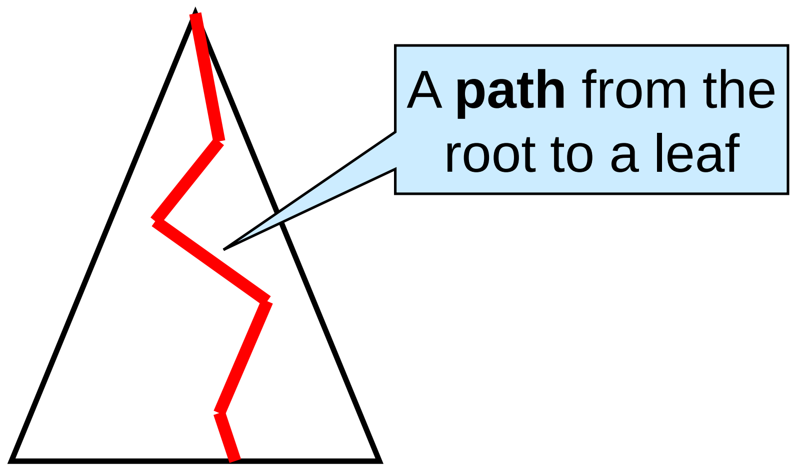


  - It contains the same elements,
  - it is sorted,
  - but the nodes are arranged differently

Reframing the Problem

- Depending on the tree, BST lookup can cost
  - O(log n) *or*
  - O(n)

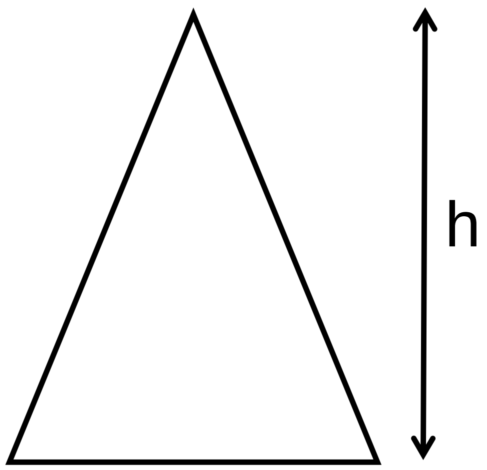
- Is there something that remains the same cost-wise?
  - ➢ Can we come up with a cost parameter that gives the same complexity in every case?
  - The cost of lookup is determined by how far down the tree we need to go
    - ➢ if the key is in the tree, the worst case is when it is in a leaf
    - ➢ if it is not in the tree, we have to reach a leaf to say so



  - The number of nodes on the longest path from the root to a leaf is called the **height** of the tree

Reframing the Problem

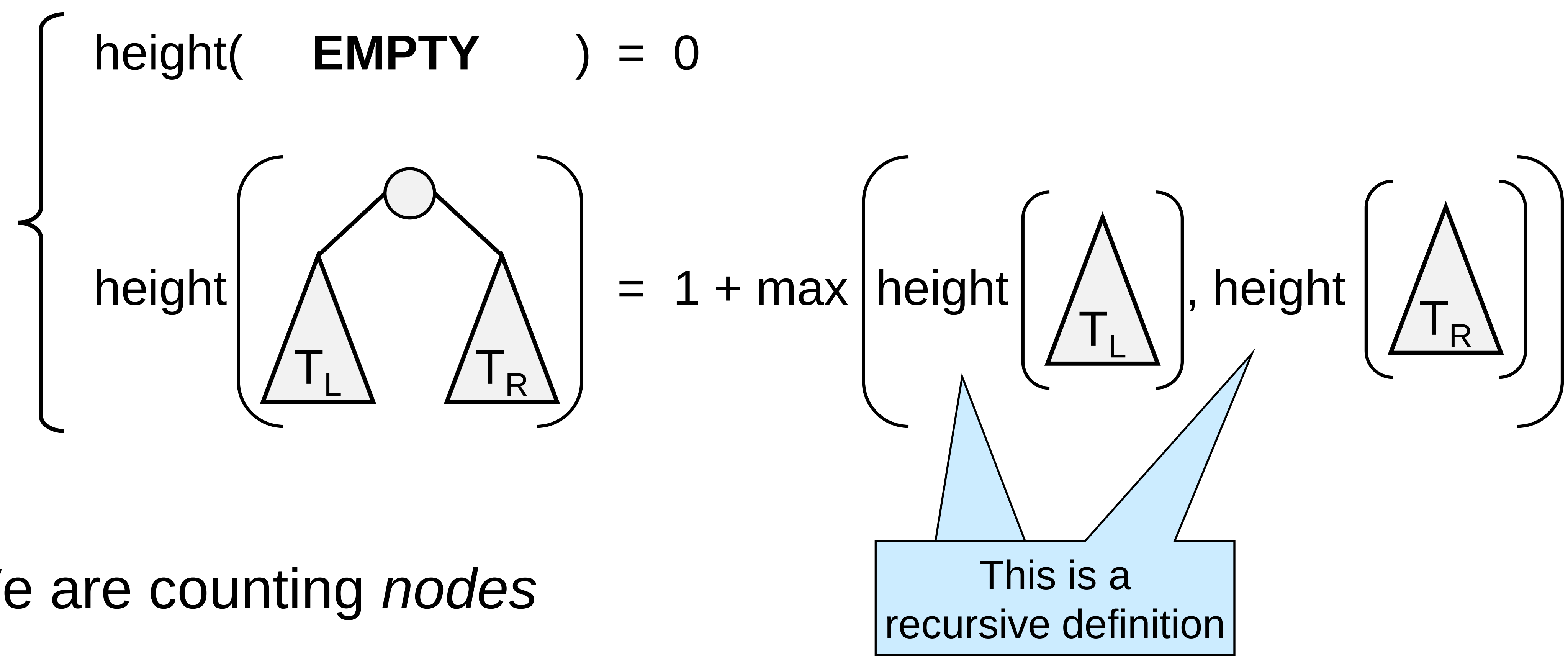
- lookup for a tree of height *h* has complexity *O(h)*
  - always!
  - same for insert and find_min



- But …
  - *h* can be in *O(n)* or in *O(log n)*
    - where *n* is the number of nodes in the tree

The Height of a Tree

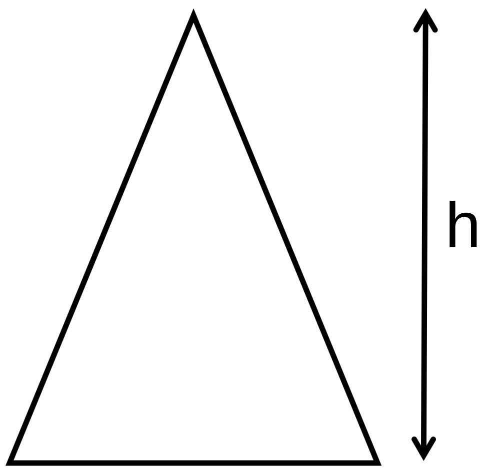
- The length of the longest path from the root to a leaf
- Let's define it mathematically

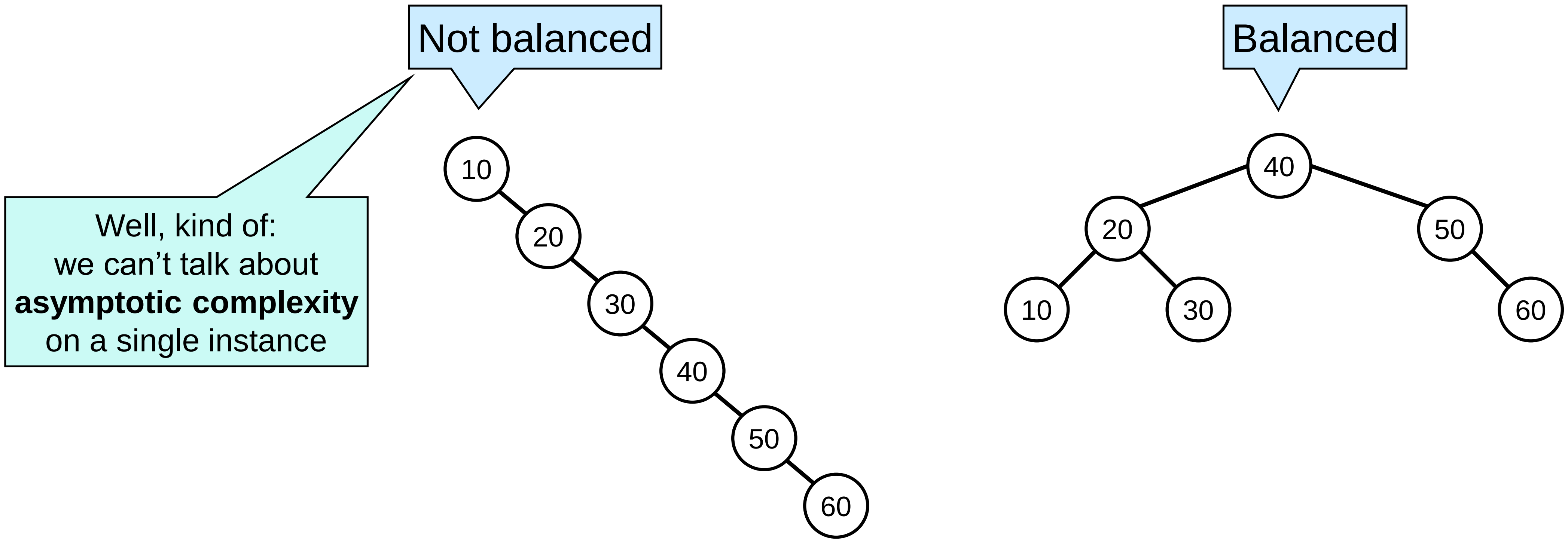


  - ○ We are counting *nodes*

Balanced Trees

- A tree is **balanced** if *h* $\in$ *O(log n)*
  - where *h* is its height and *n* is the number of nodes





- On a balanced tree, lookup, insert and find_min cost *O(log n)*

Self-balancing Trees

New goal:

- make sure that a tree remains balanced as we insert new nodes

… and continues to be a valid BST

- Trees with this property are called **self-balancing**
  - There are lots of them
    - AVL trees
    - Red-black trees
    - B-trees
    - …

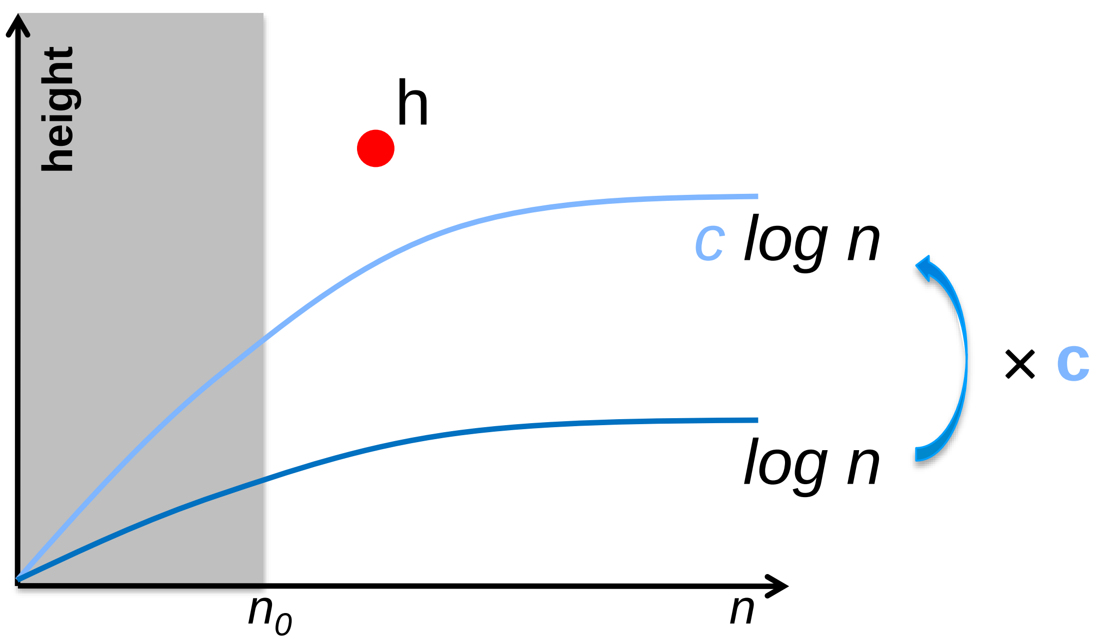
We will study this one

Why so many?

- there are many ways to guarantee that the tree remains balanced after each insertion
- some of these tree types have other properties of interest

Self-balancing Trees

- *"the tree stays balanced after each insertion"* is too vague
  - $h \in O(\log n)$ is an asymptotic behavior
    - we can't check it on any given tree
- Recall the definition



- We can fit any given h by
  - picking a bigger n_0
  - picking a bigger c

We can't say a given tree is **not** balanced

- More fundamentally, h needs to be a function in $h \in O(\log n)$

Self-balancing Trees

- *"the tree stays balanced after each insertion"* is too vague
  - $h \in O(\log n)$ is an asymptotic behavior
    - we can't check it on any given tree

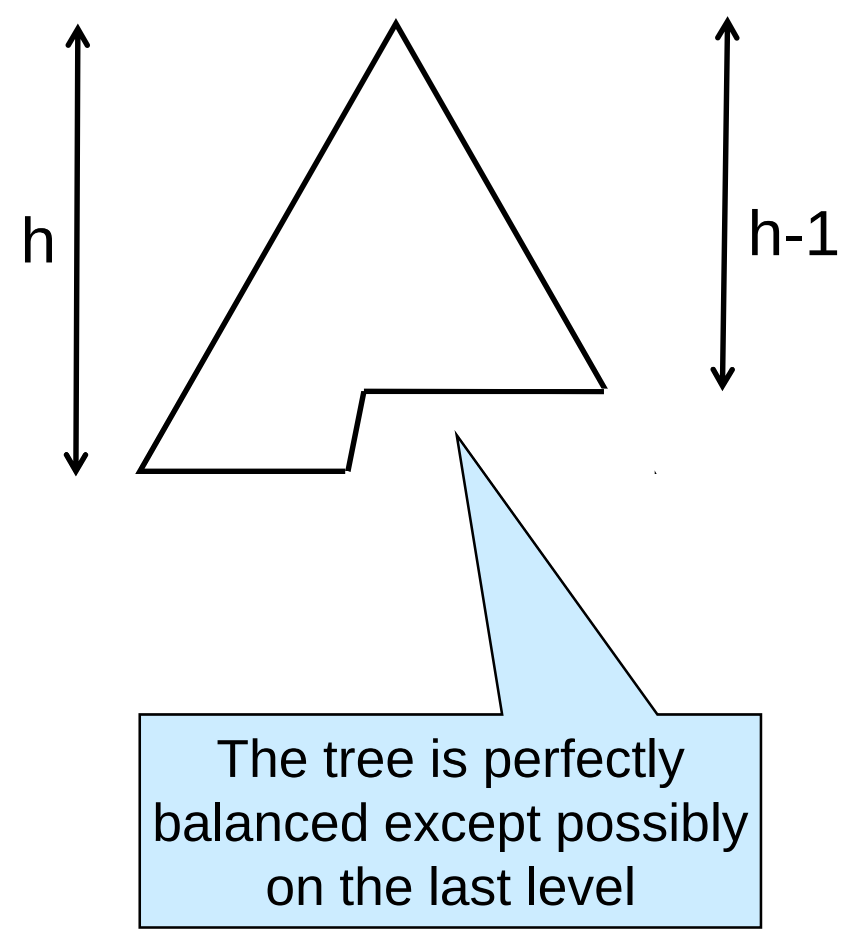
Specifically, we can't say a given tree is **not** balanced

- We want **algorithmically-checkable** constraints that
  1. guarantee that $h \in O(\log n)$
  2. are cheap to maintain
     - at most $O(\log n)$

- We do so by imposing an **additional representation invariants** on trees
    - on top of the ordering invariant
  - this *balance invariant*, when valid, ensures that $h \in O(\log n)$

A Bad Balance Invariant

- Require that
  - (the tree be a BST)
  - all the paths from the root to a leaf have height either *h* or *h-1*
  - the leaves at height *h* be on the left-hand side of the tree

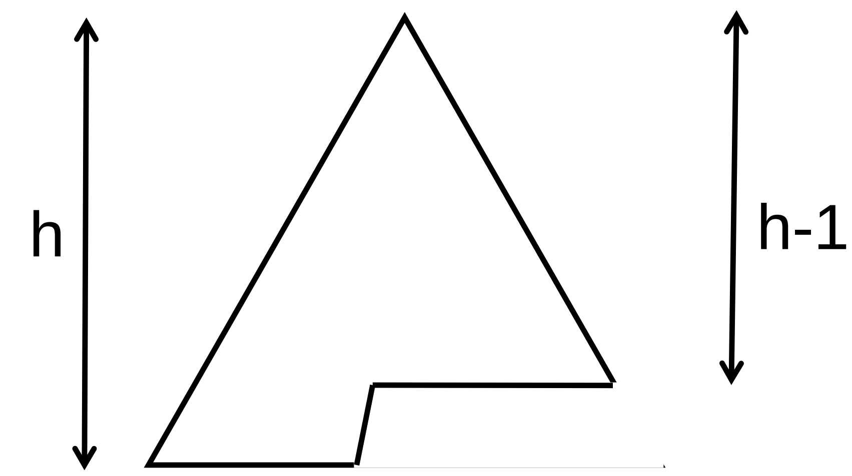


- Does it satisfy our requirements?
  1. guarantees that $h \in O(\log n)$ ✓
     - Definitely!
  2. cheap to maintain — at most $O(\log n)$
     - Let’s see

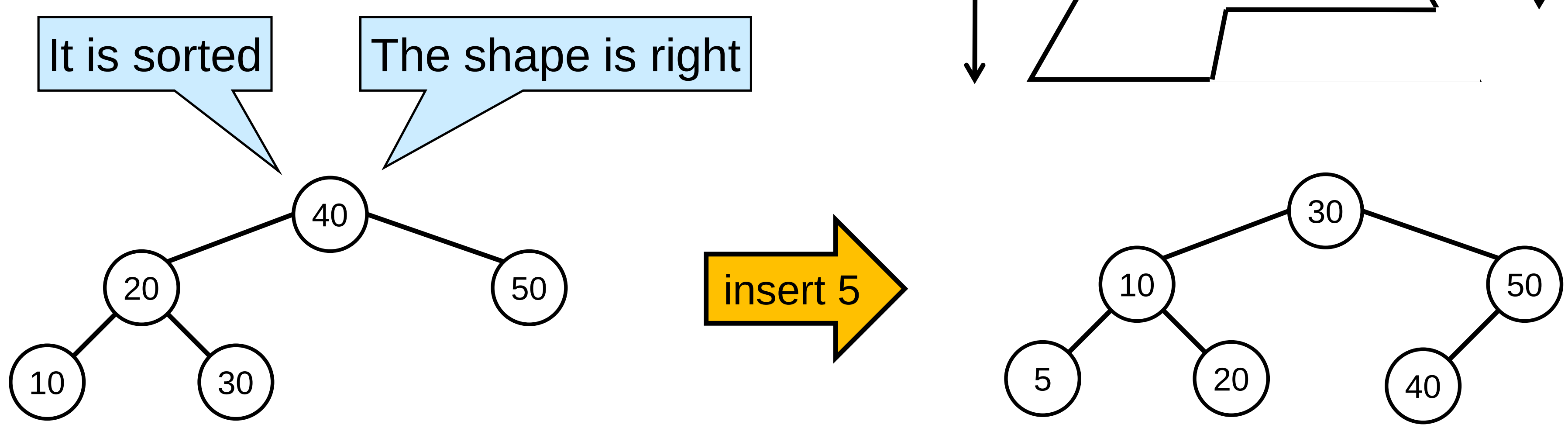
A Bad Balance Invariant

- Does it satisfy our requirements?

  1. guarantees that $h \in O(\log n)$



- Let's insert 5 in this tree



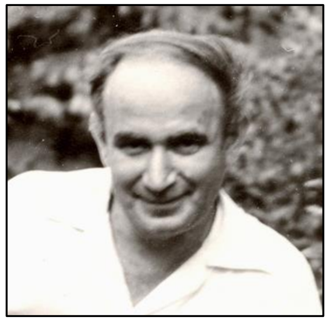
  - We changed all the pointers to maintain the balance invariant!
    - O(n)

  2. cheap to maintain — at most $O(\log n)$

AVL Trees

Adelson-**V**elsky

AVL Trees



Landis

The first self-balancing trees (1962)

That's what the balance invariant of AVL trees is called

- **Height invariant**

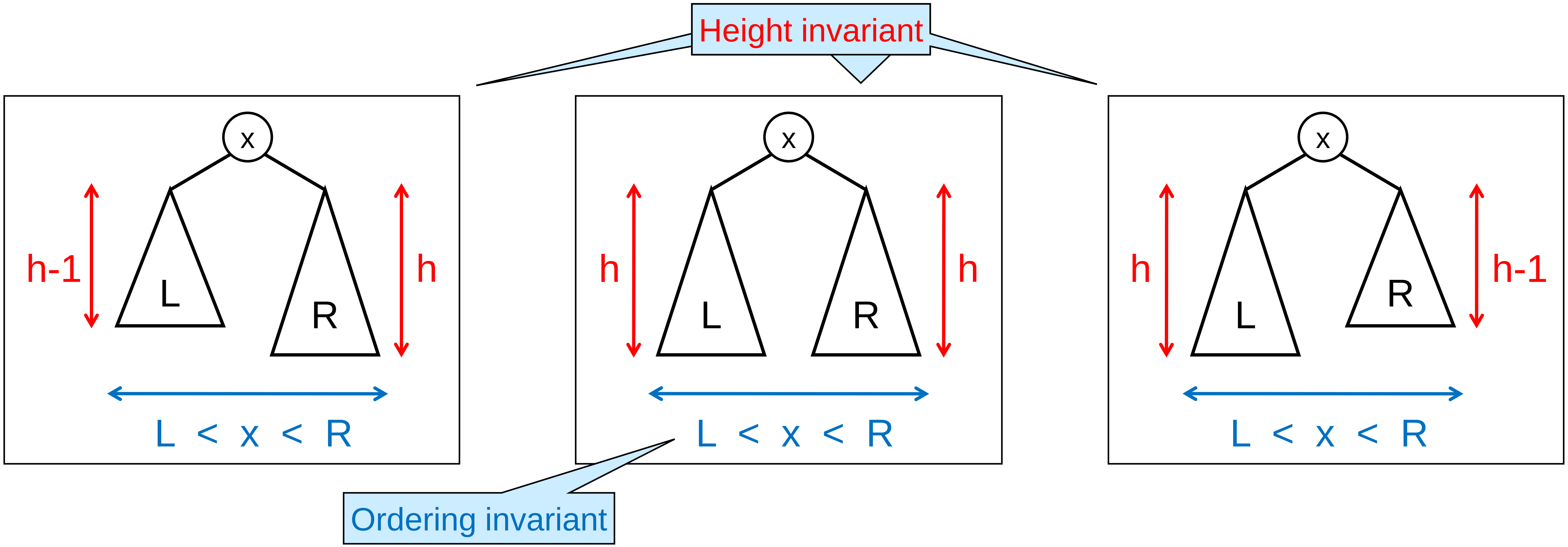
  At every node, the heights of the left and right subtrees differ by at most 1

- An AVL tree satisfies two invariants
  - the ordering invariant
  - the height invariant

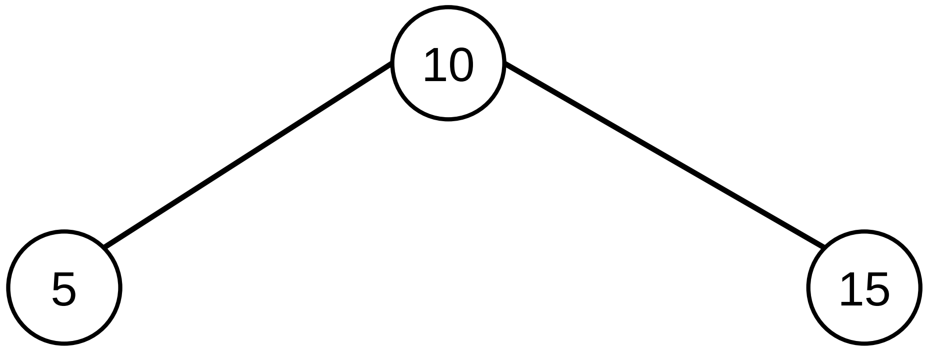
The Invariants of AVL Trees

- *The nodes are ordered*
- *At every node, the heights of the left and right subtrees differ by at most 1*

- At any node, there are 3 possibilities



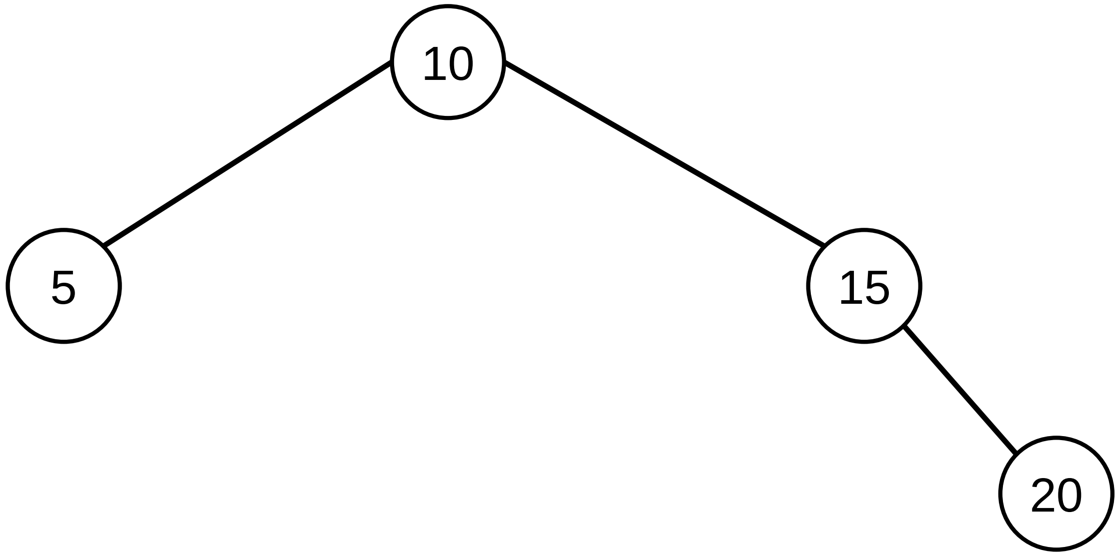
Is this an AVL Tree?



- Is it sorted? ✓
- Do the heights of the two subtrees of every node differ by at most 1? ✓

YES

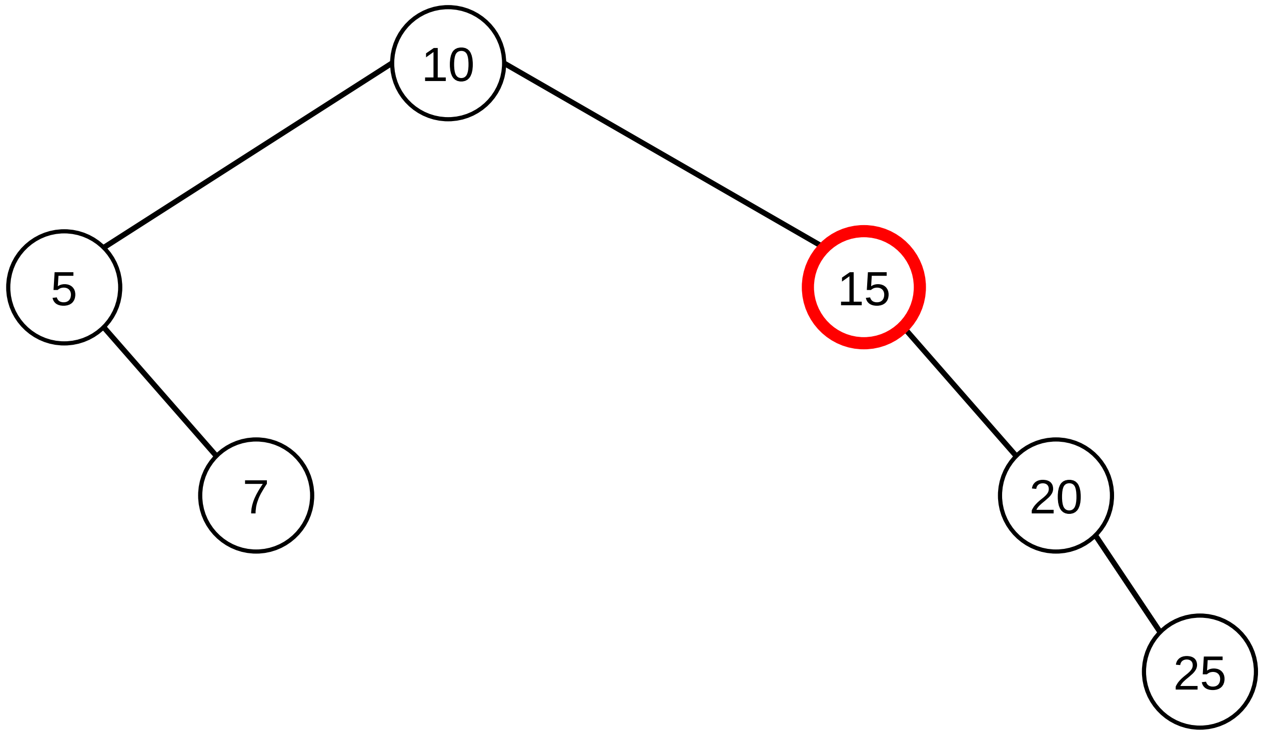
Is this an AVL Tree?



- Is it sorted? ✓
- Do the heights of the two subtrees of every node differ by at most 1? ✓

YES

Is this an AVL Tree?

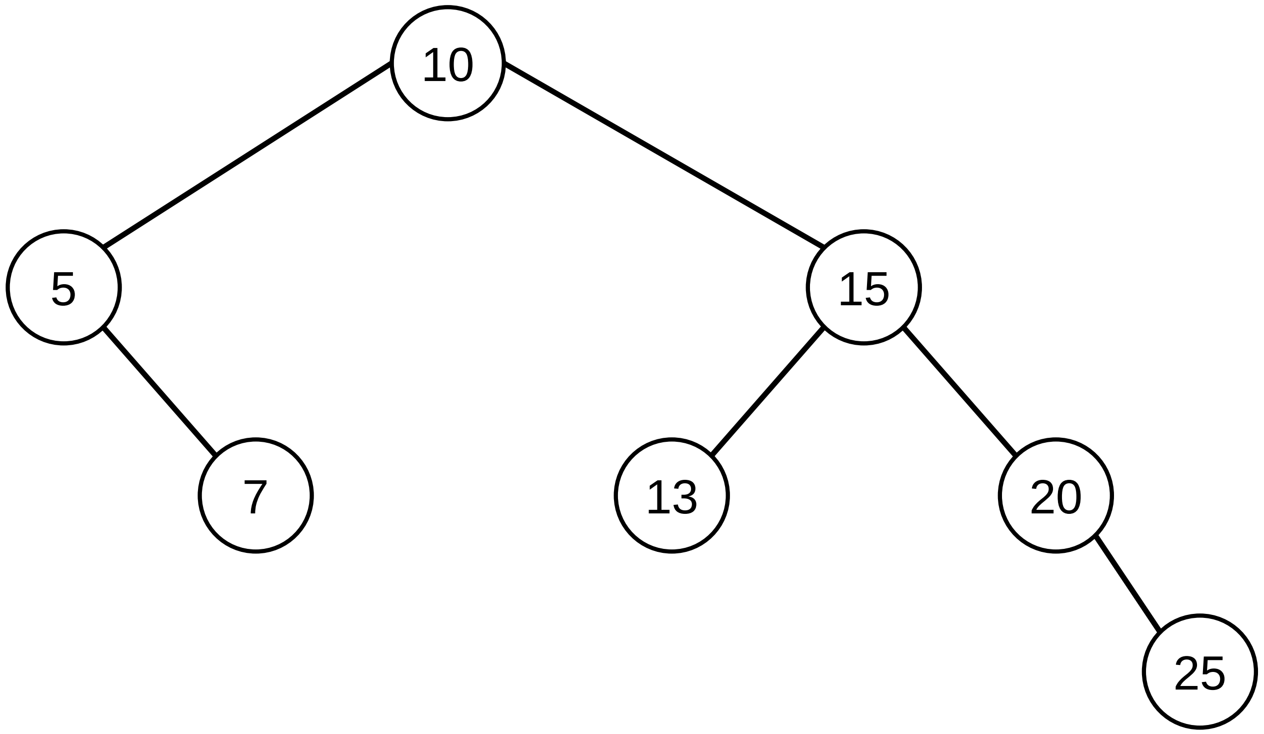


- Is it sorted? ✓
- Do the heights of the two subtrees of every node differ by at most 1? ✗
    - It doesn't hold at node 15

NO

- We say there is a **violation** at node 15

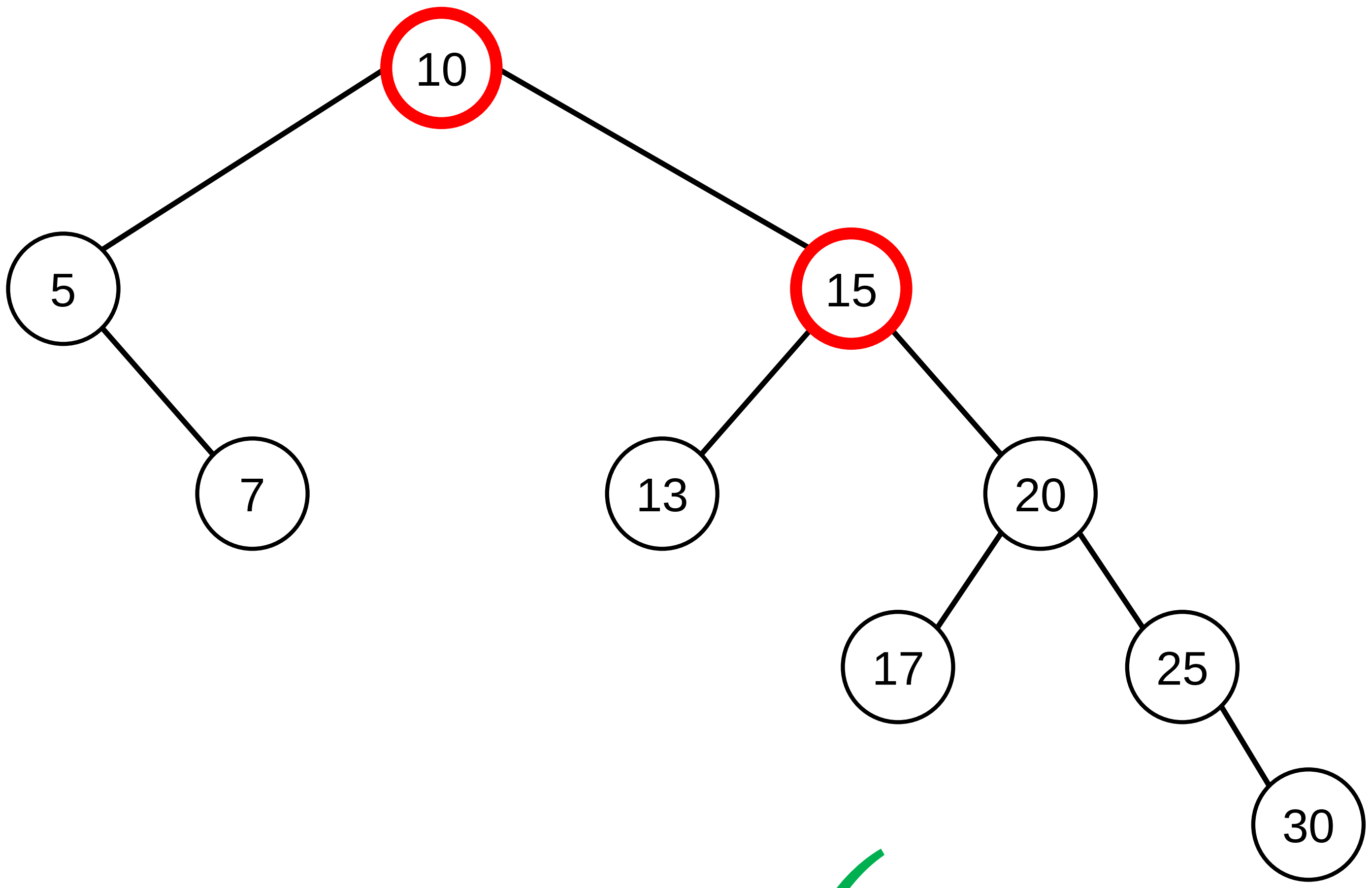
Is this an AVL Tree?



- Is it sorted? ✓
- Do the heights of the two subtrees of every node differ by at most 1? ✓

YES

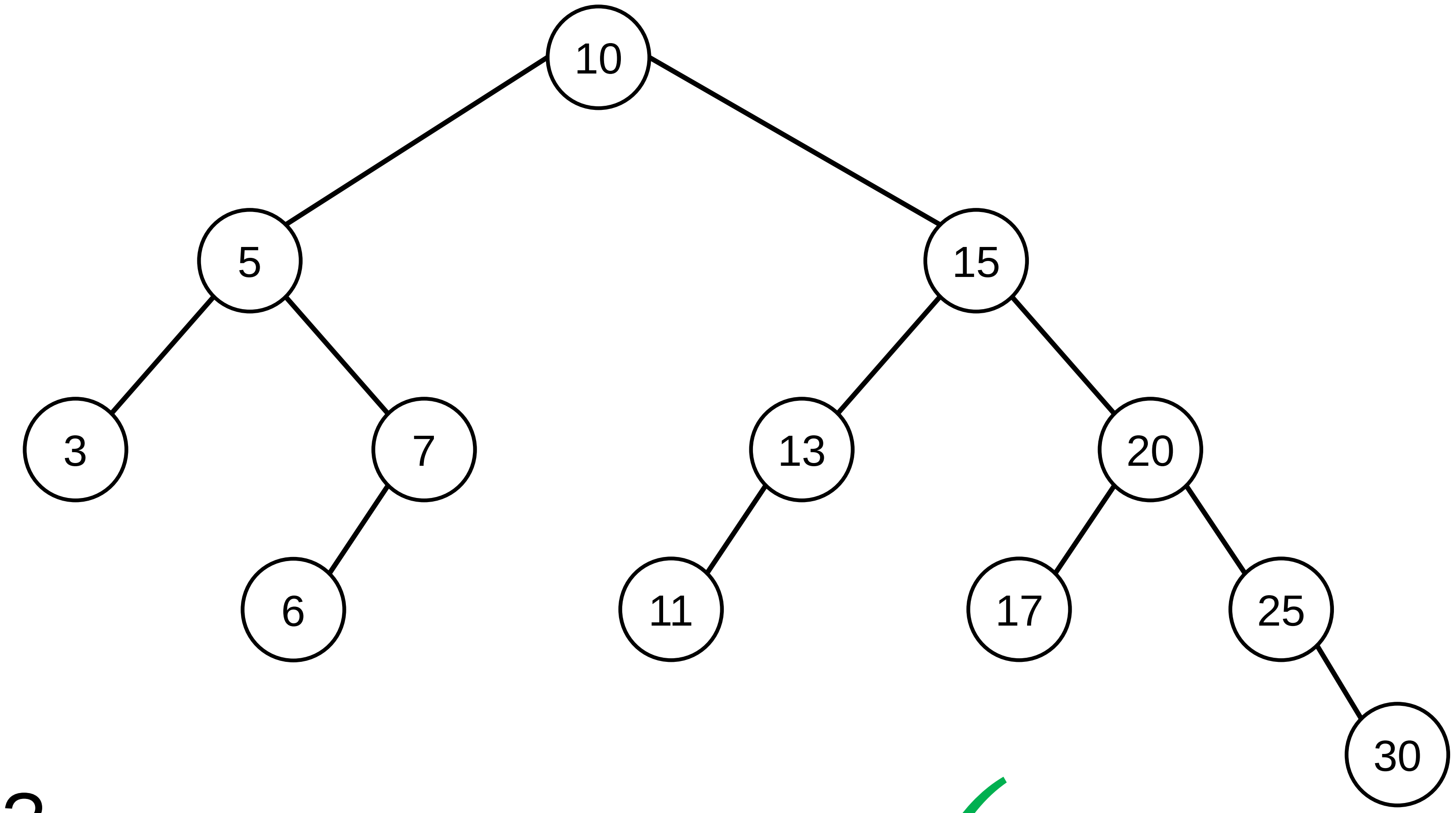
Is this an AVL Tree?



- Is it sorted?
- Do the heights of the two subtrees of every node differ by at most 1?
  - There is a violation at node 15 and another violation at node 10

NO

Is this an AVL Tree?



- Is it sorted? ✓
- Do the heights of the two subtrees of every node differ by at most 1? ✓

The height invariant does **not** imply that the length of every path from the root to a leaf differ by at most 1

YES

Rotations

Insertion Strategy

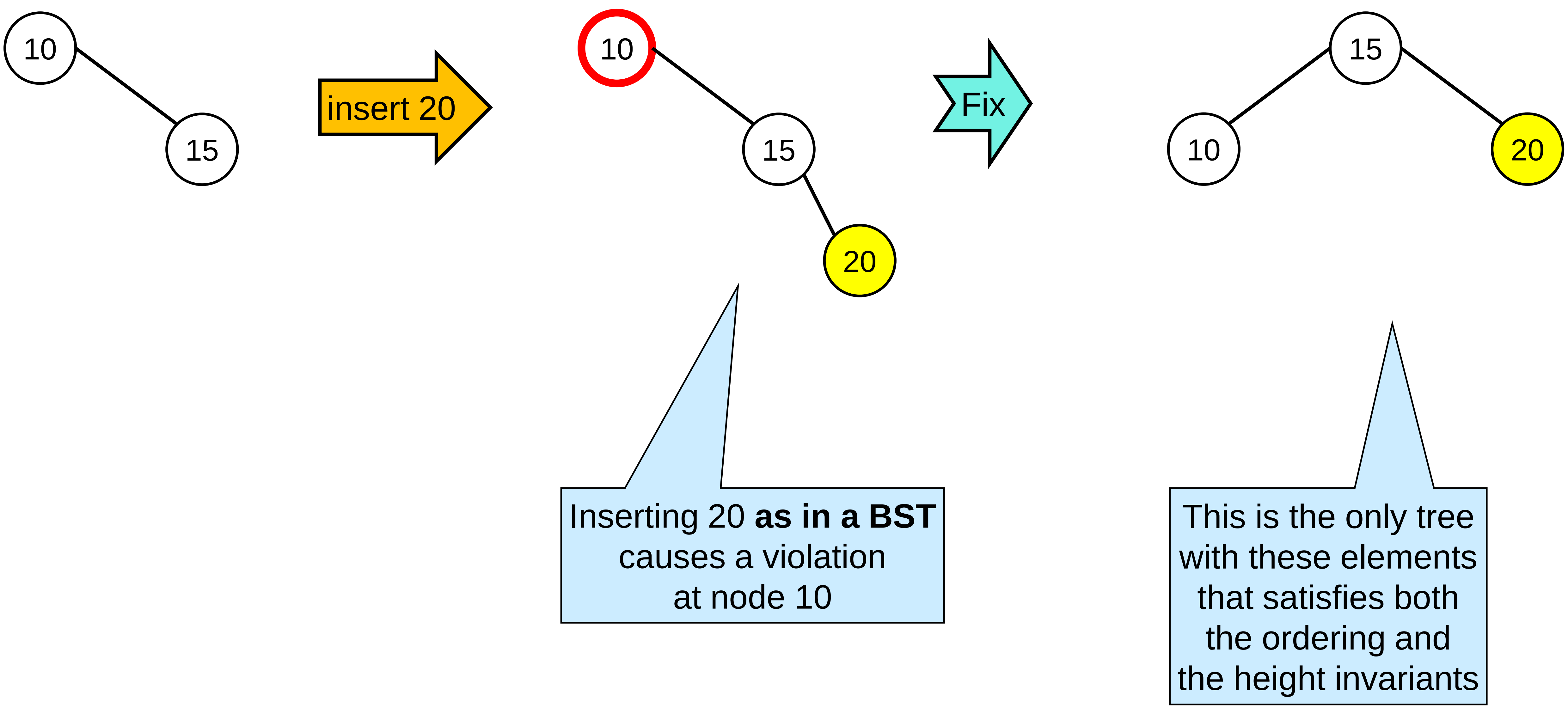
1. Insert the new node as in a BST
   - this preserves the ordering invariant
   - but it **may break the height invariant**

2. Fix any height invariant violations
   - fix the **lowest** violation
     - this will take care of all other violations

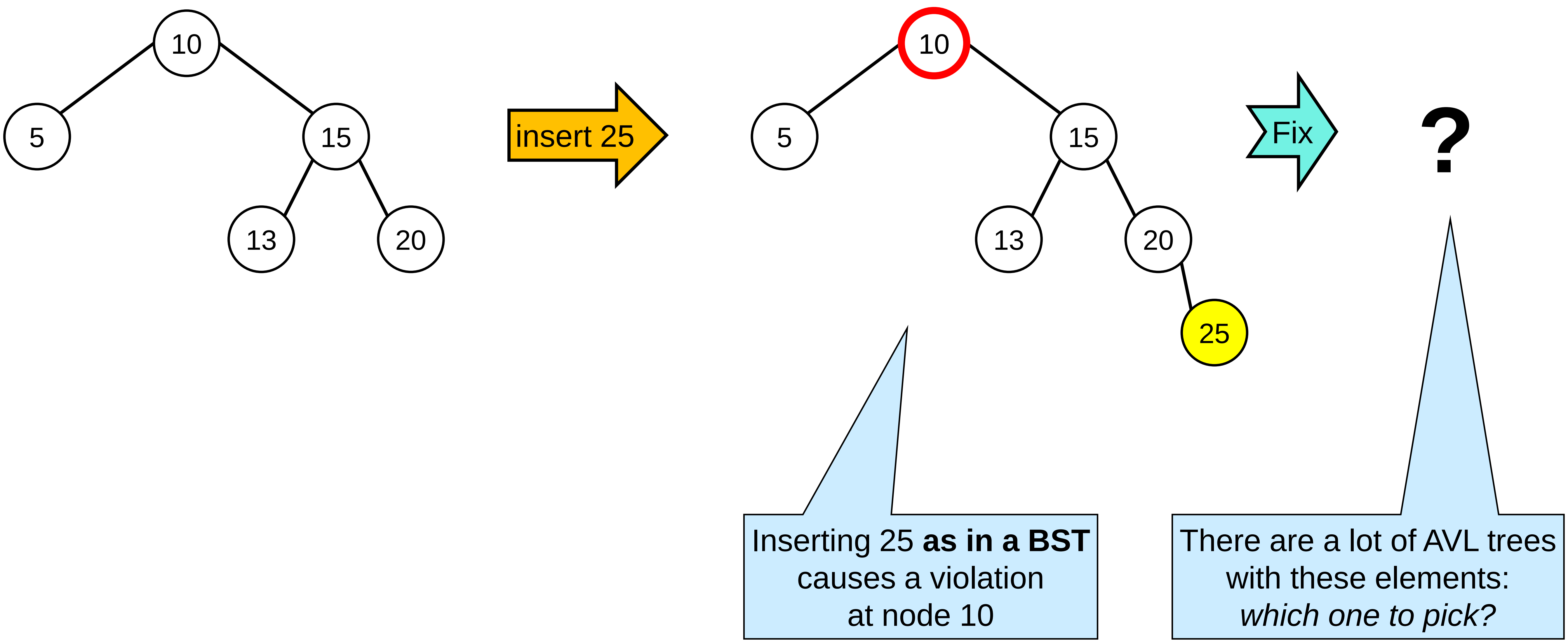
We will see why later

- This is a common approach
  - of two invariants, preserve one and temporarily break the other
  - then, patch the broken invariant
    - cheaply

Example 1

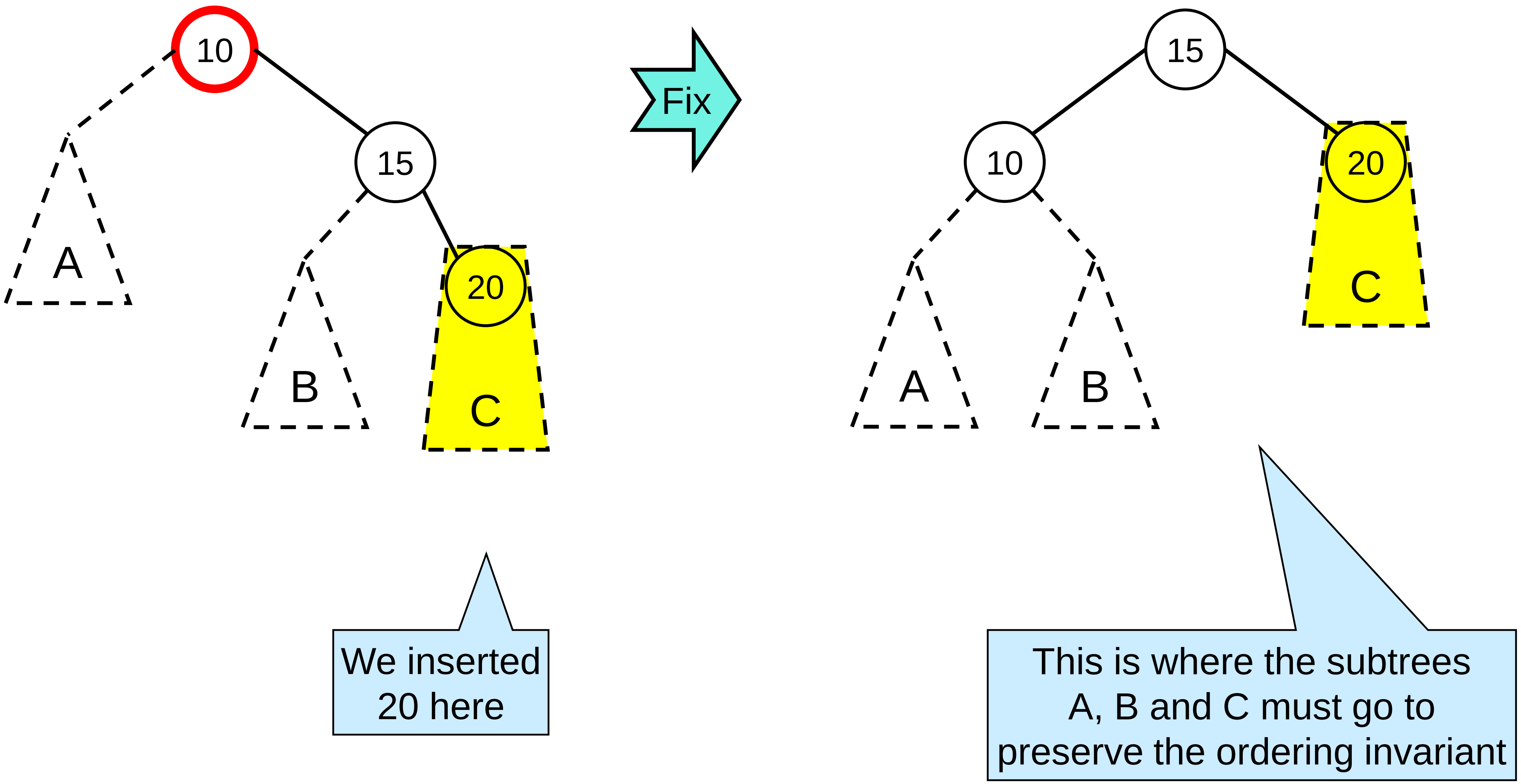

10
15
insert 20
10
15
20
Fix
15
10
20
Inserting 20 **as in a BST** causes a violation at node 10
This is the only tree with these elements that satisfies both the ordering and the height invariants

Example 2

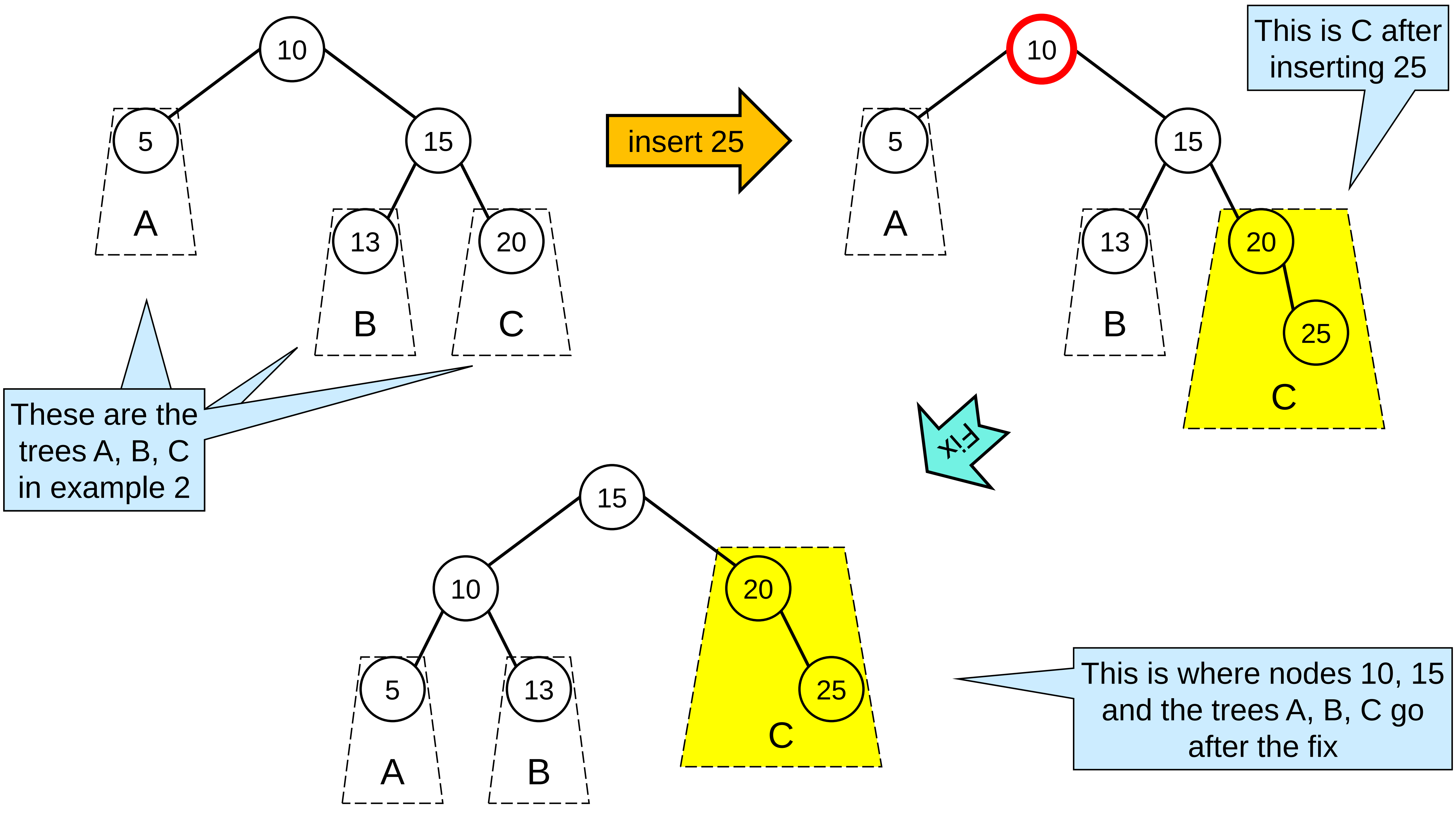

10
5
15
13
20
insert 25
10
5
15
13
20
25
Fix
?
Inserting 25 as in a BST
causes a violation
at node 10
There are a lot of AVL trees
with these elements:
which one to pick?

Example 1 Revisited

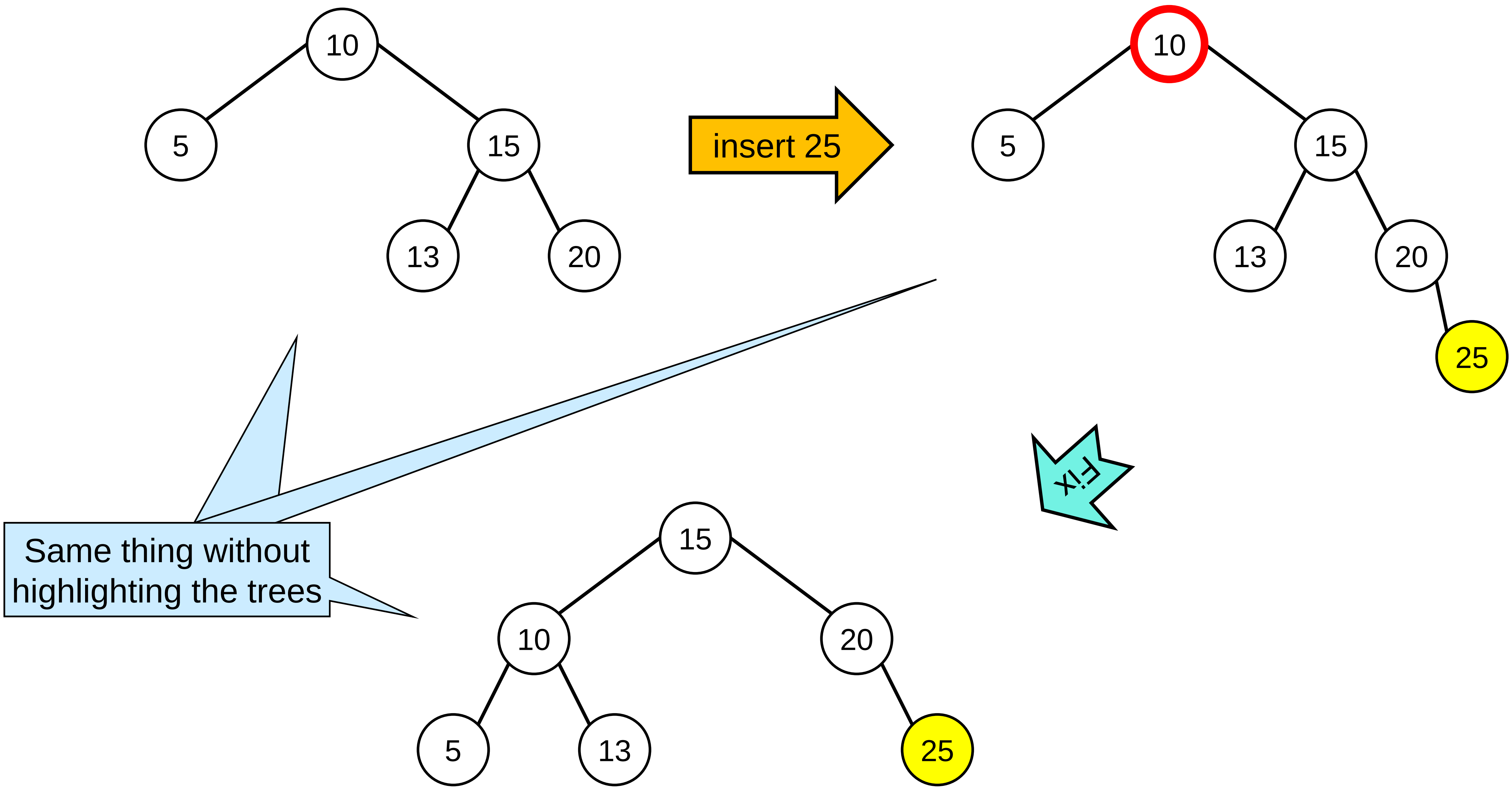
- If this example was part of a bigger tree, what would it look like?



Example 2

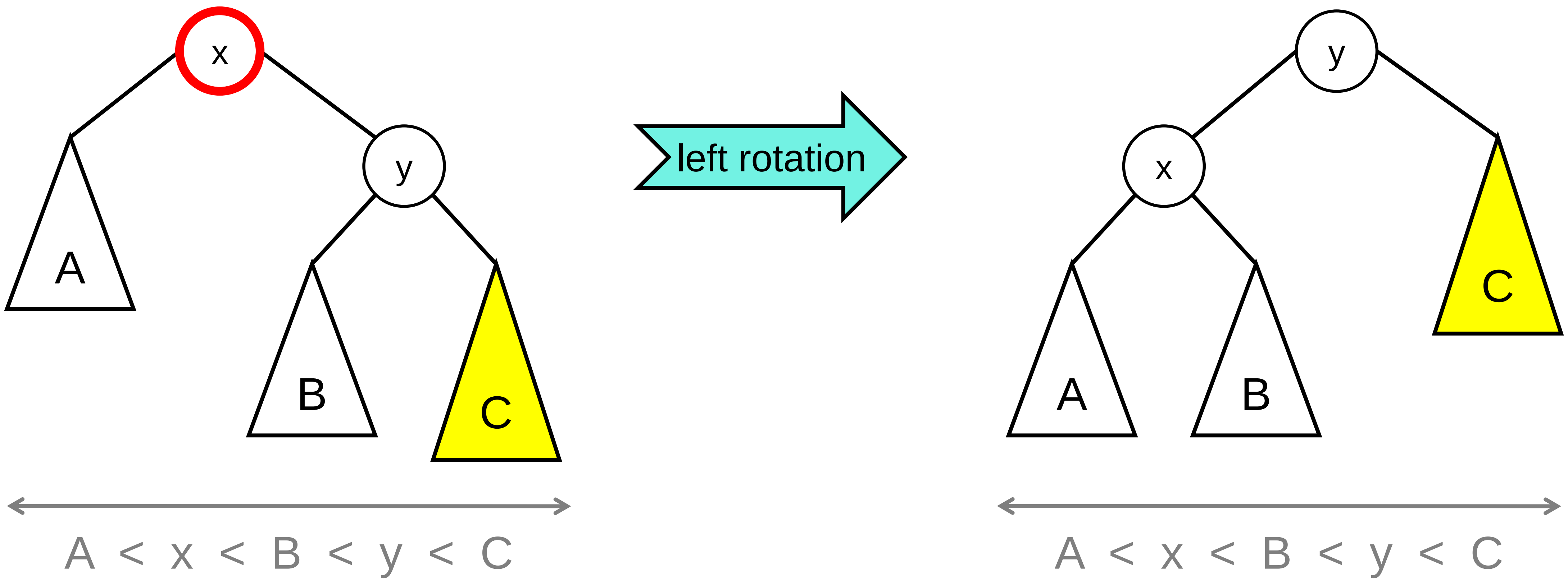

10
5
15
A
13
20
B
C
These are the trees A, B, C in example 2
insert 25
10
5
15
A
13
20
25
B
C
This is C after inserting 25
Fix
15
10
20
5
13
25
A
B
C
This is where nodes 10, 15 and the trees A, B, C go after the fix

Example 2


10
5
15
13
20
insert 25
10
5
15
13
20
25
Fix
15
10
20
5
13
25
Same thing without
highlighting the trees

Left Rotation

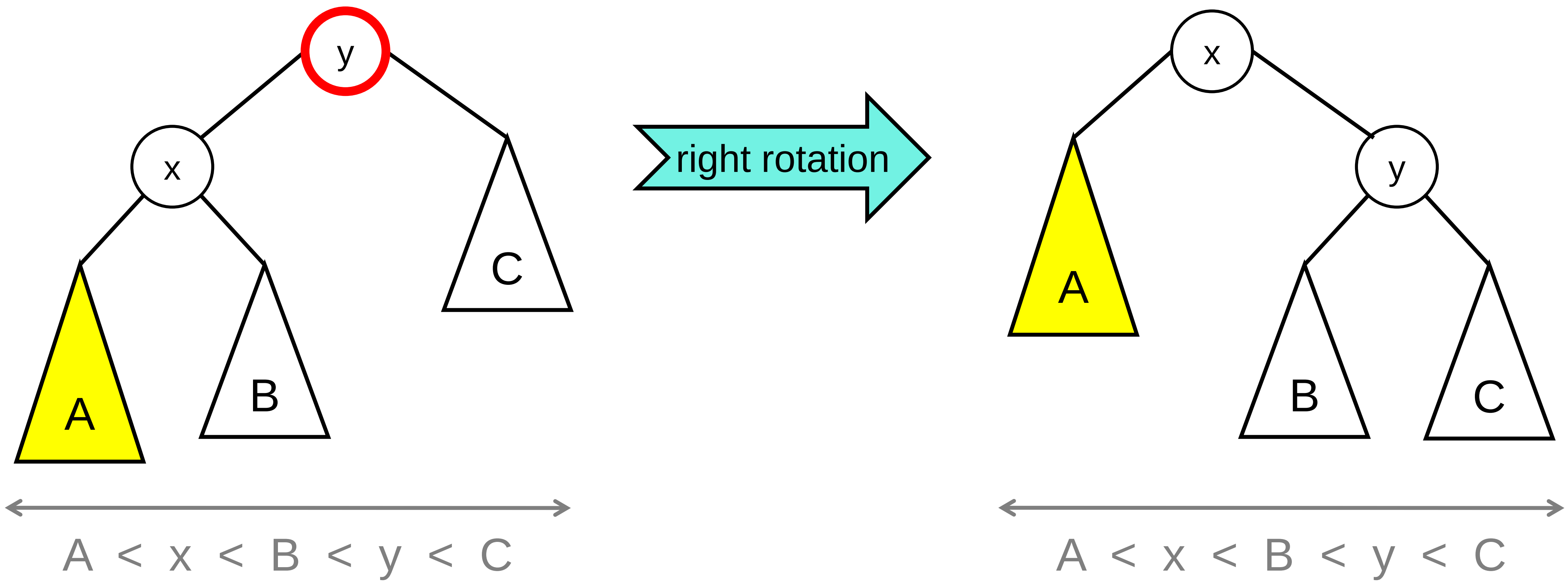
- This transformation is called a **left rotation**



  - Note that it maintains the ordering invariant
- We do a left rotation when C has become too tall after an insertion

Right Rotation

- The symmetric situation is called a **right rotation**

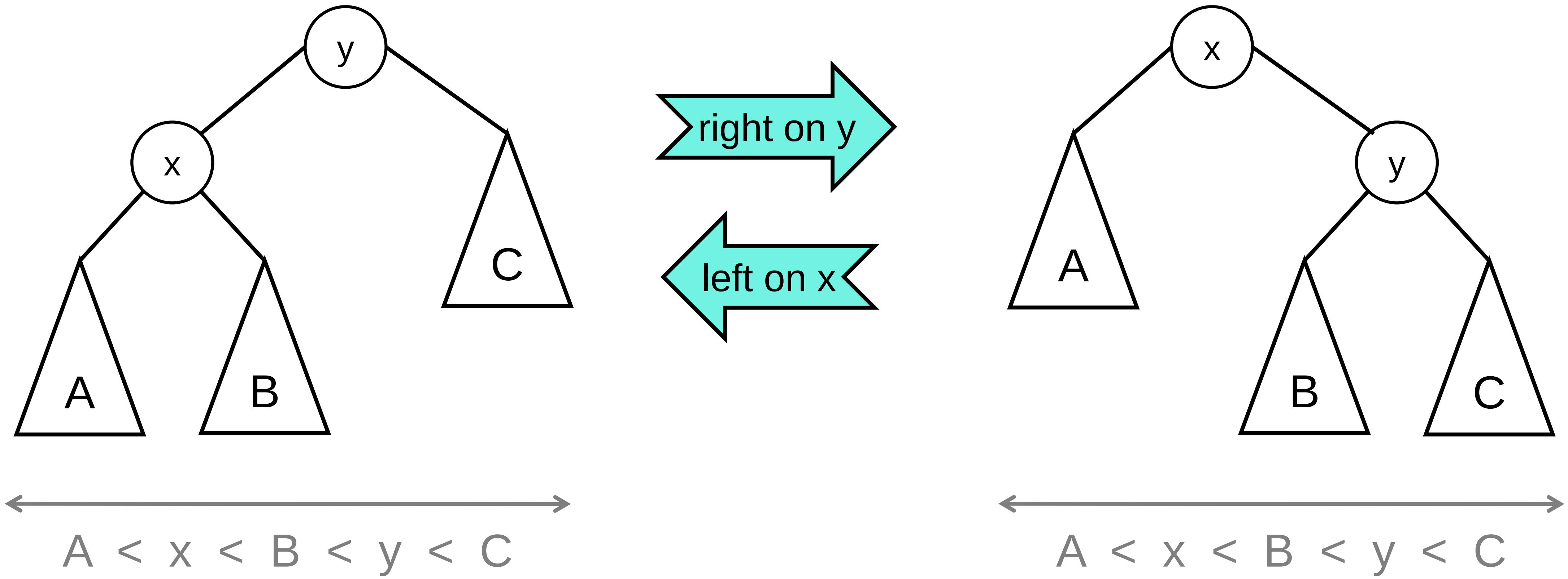


  - It too maintains the ordering invariant

- We do a right rotation when A has become too tall after an insertion

Single Rotations Summary

- Right and left rotations are **single rotations**



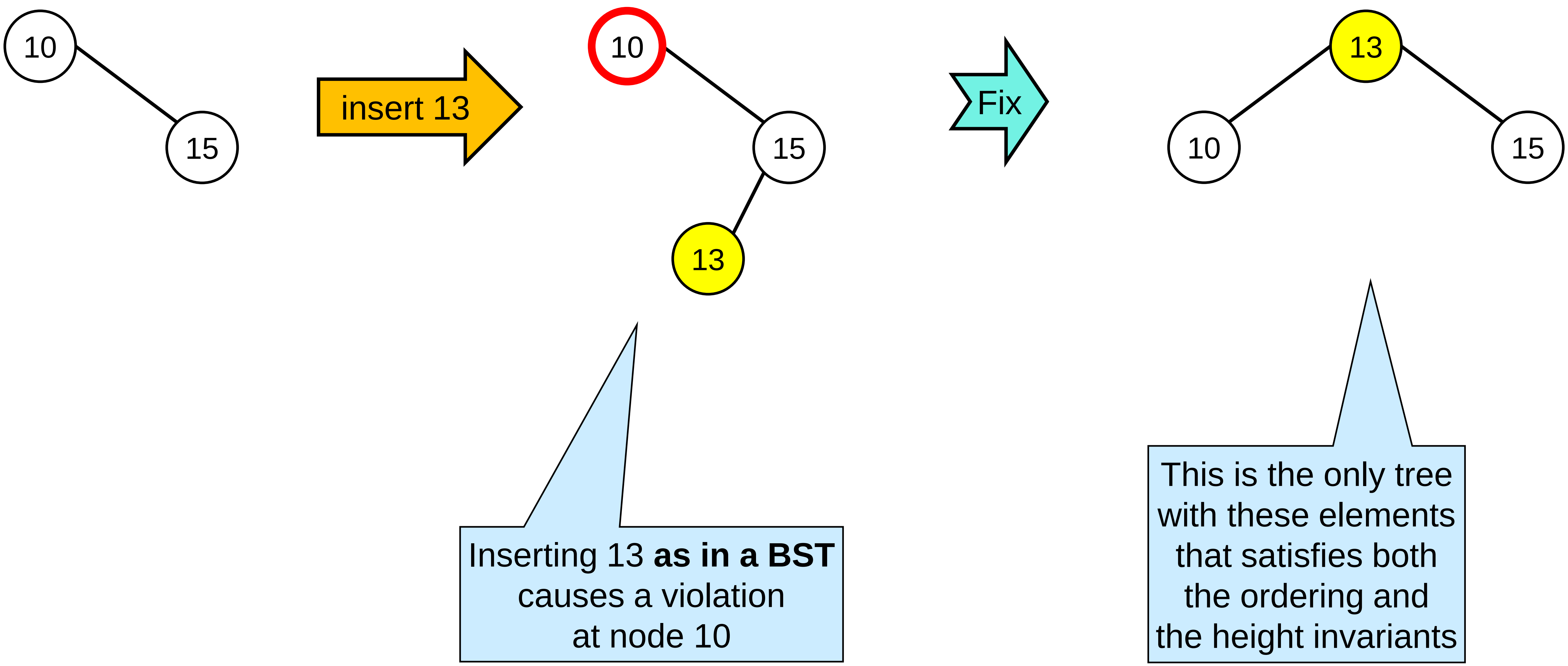
  - They maintain the ordering invariant

- We do one of them when
  - the **lowest violation** is at the root
  - one of the **outer subtrees** has become too tall

That's either y or x

That's either A or C respectively

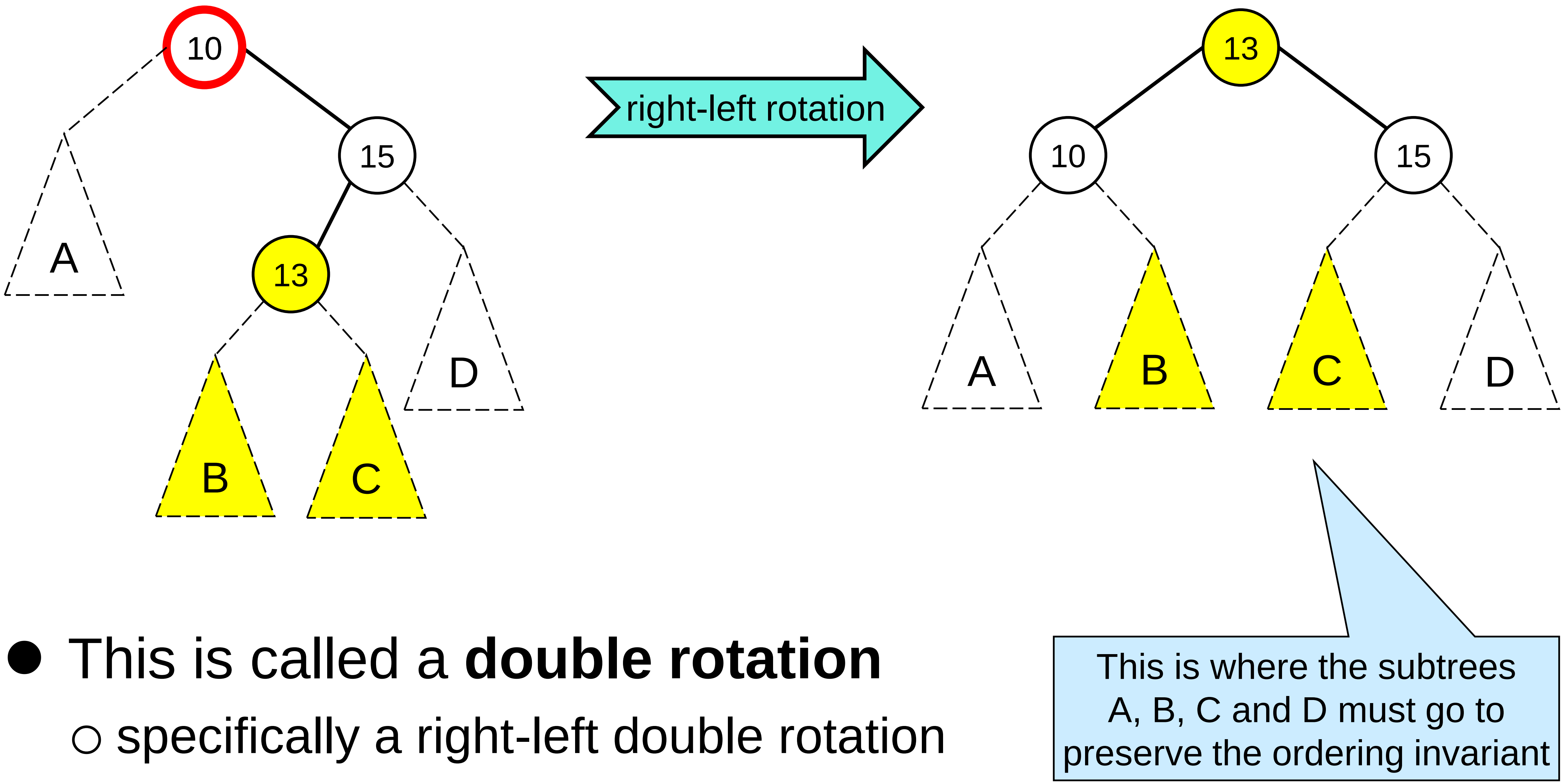
Example 3



- The fix is **not** a single rotation at 10

Double Rotations

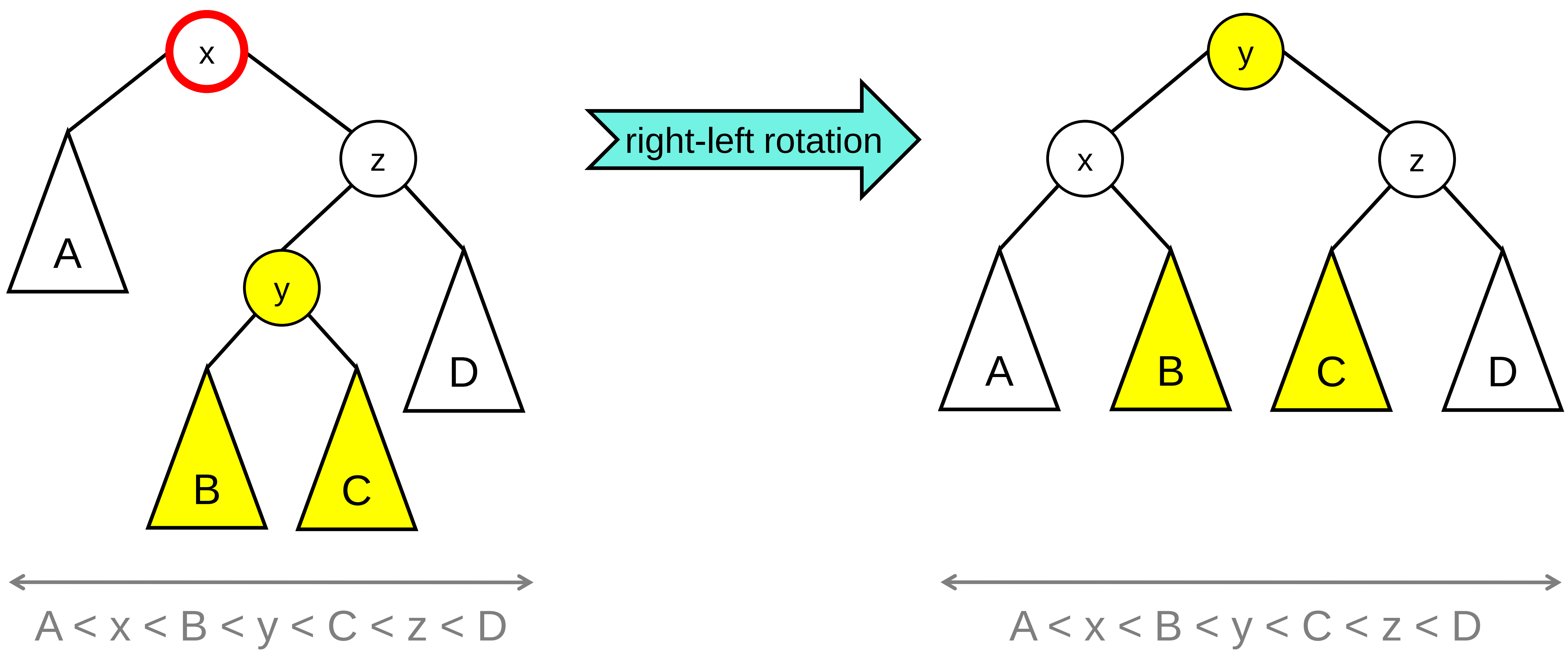
- We can generalize this example to the case where the nodes have subtrees



- This is called a **double rotation**
  - specifically a right-left double rotation

Right-left Double Rotation

- Here’s the general pattern

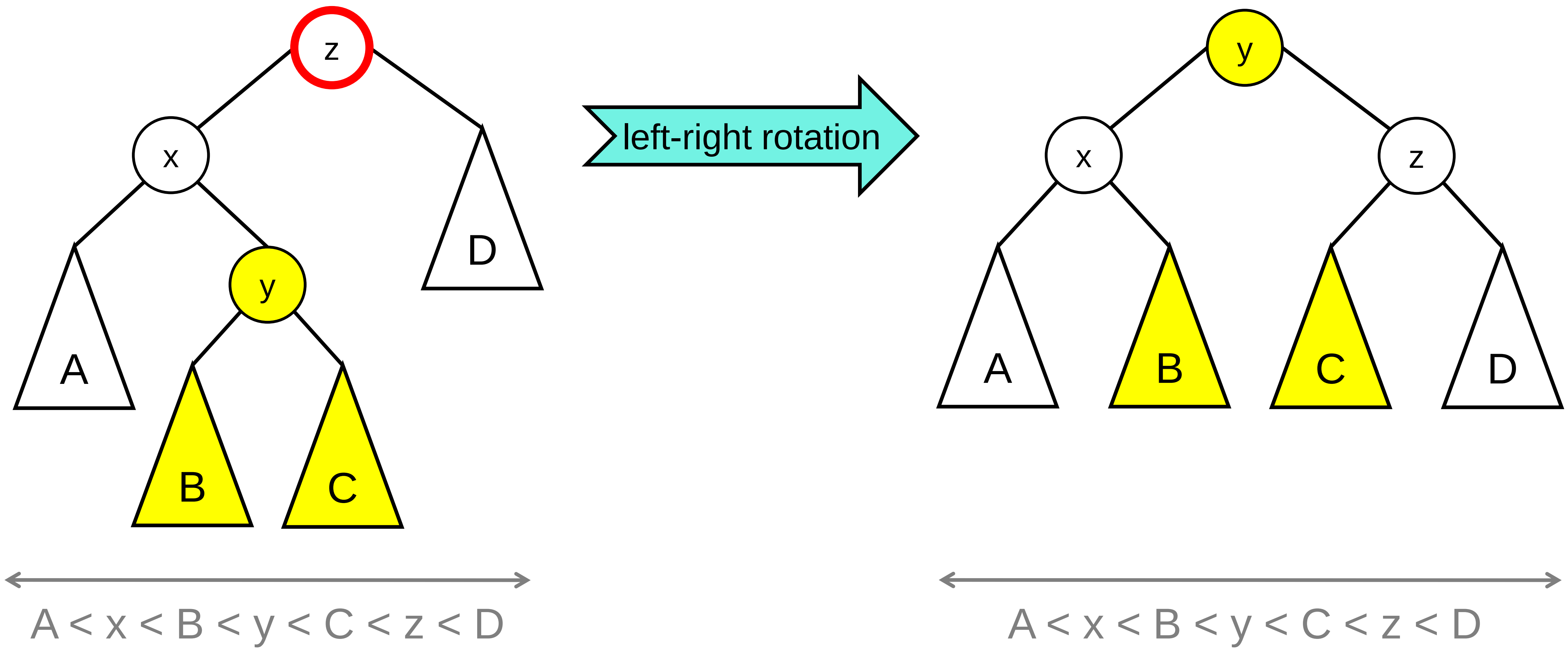


- We do this double rotation when the subtree rooted at y has become too tall after an insertion

The ordering invariant is maintained

Left-right Double Rotation

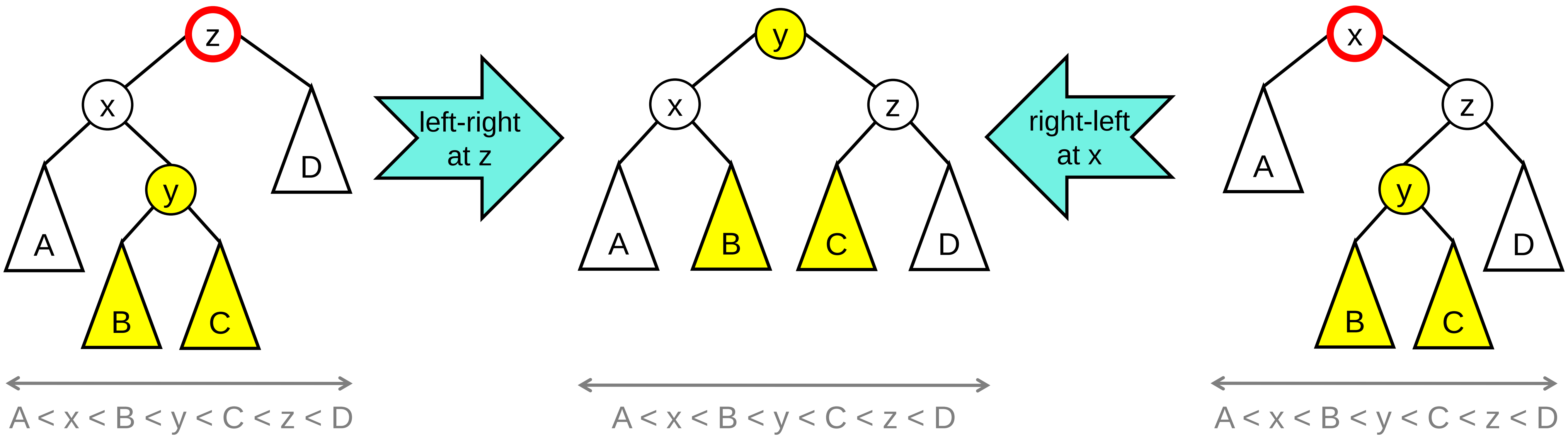
- The symmetric transformation is a left-right double rotation



- We do this double rotation when the subtree rooted at y has become too tall after an insertion

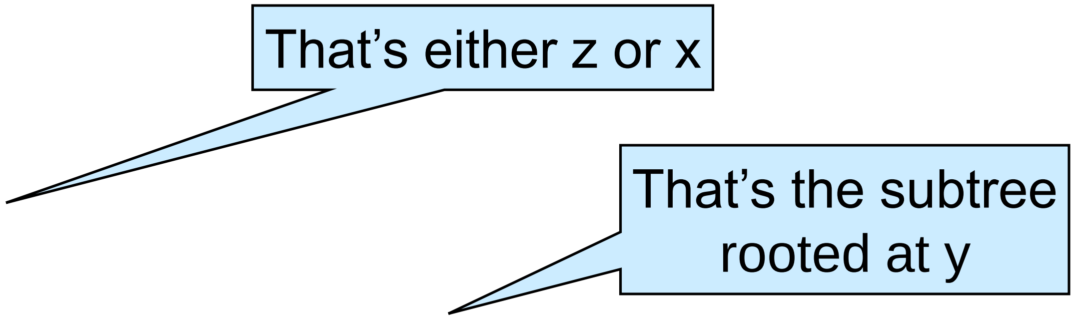
The ordering invariant is maintained

Double Rotations Summary

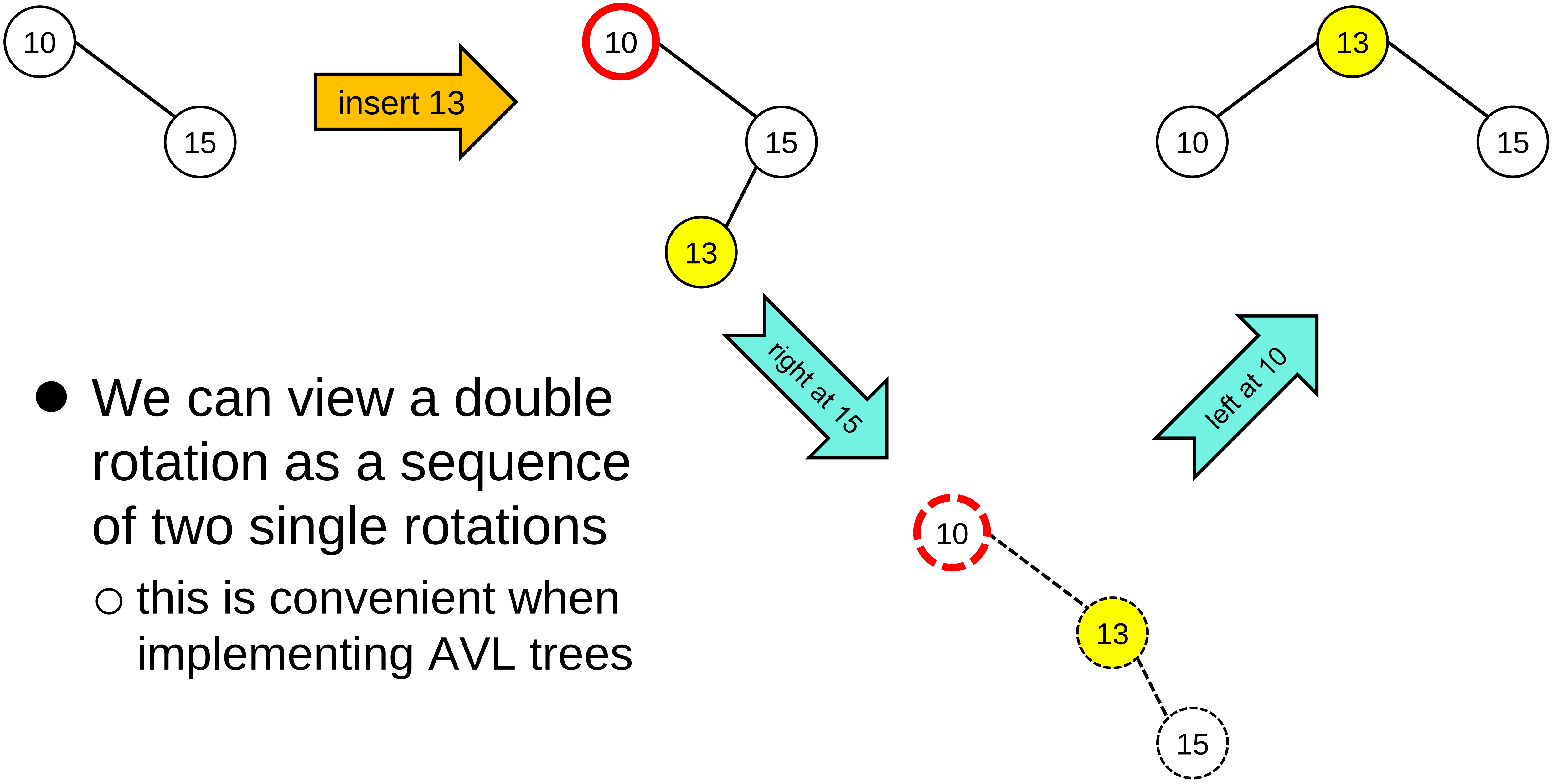


- Double rotations maintain the ordering invariant

- We do one of them when
  - the **lowest violation** is at the root
  - one of the **inner subtrees** has become too tall



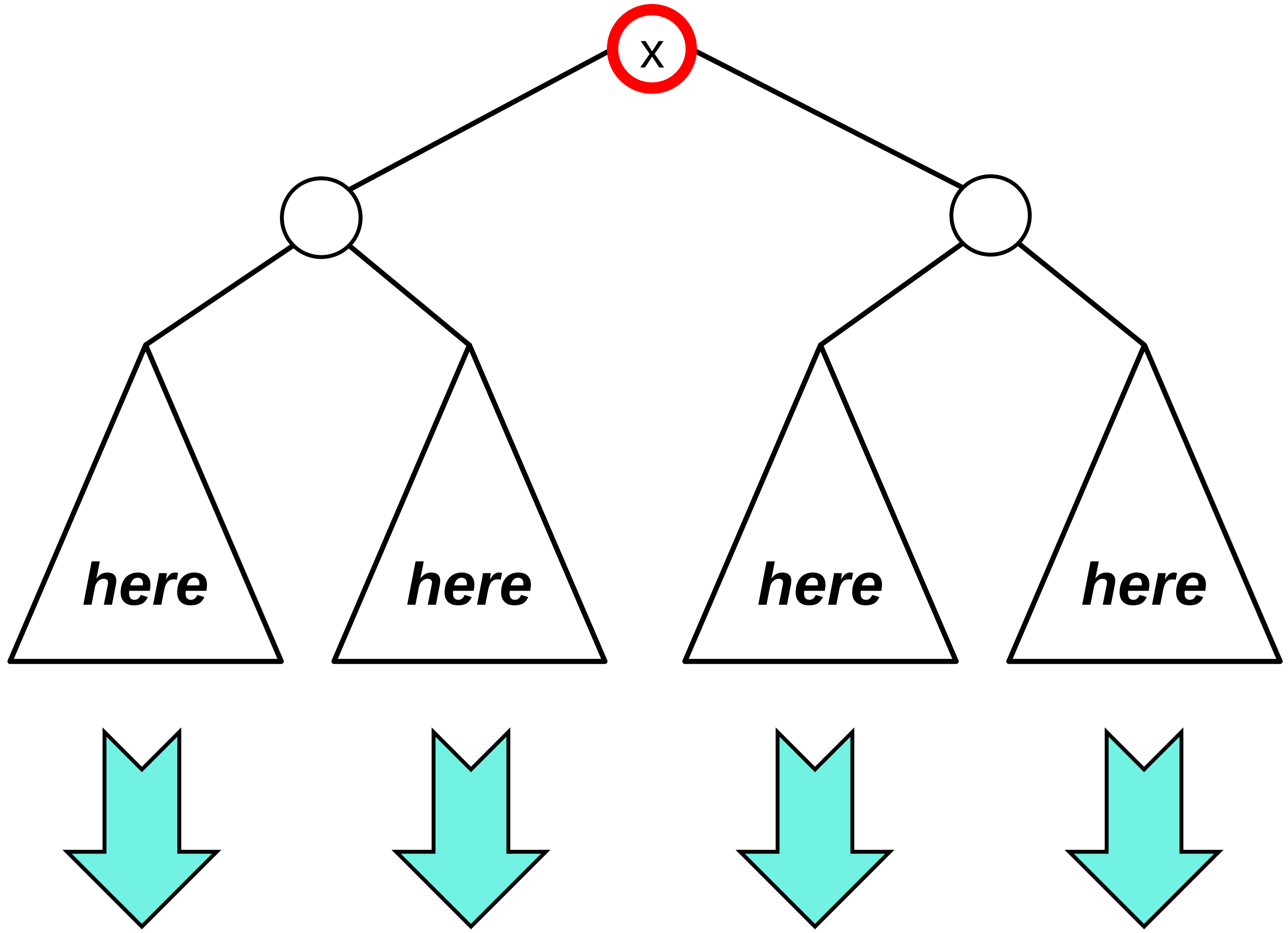
Why is it Called a ***Double*** Rotation?



- We can view a double rotation as a sequence of two single rotations
  - this is convenient when implementing AVL trees

AVL Rotation When-to

If the insertion that caused the lowest violation (x) happened …



… **then** do a …

right single rotation at x | **left/right double rotation at x** | **right/left double rotation at x** | **left single rotation at x**

Self-balancing Requirements

- Does the height constraint satisfy our requirements?

1. It guarantees that $h \in O(\log n)$ ✓ Left as exercise

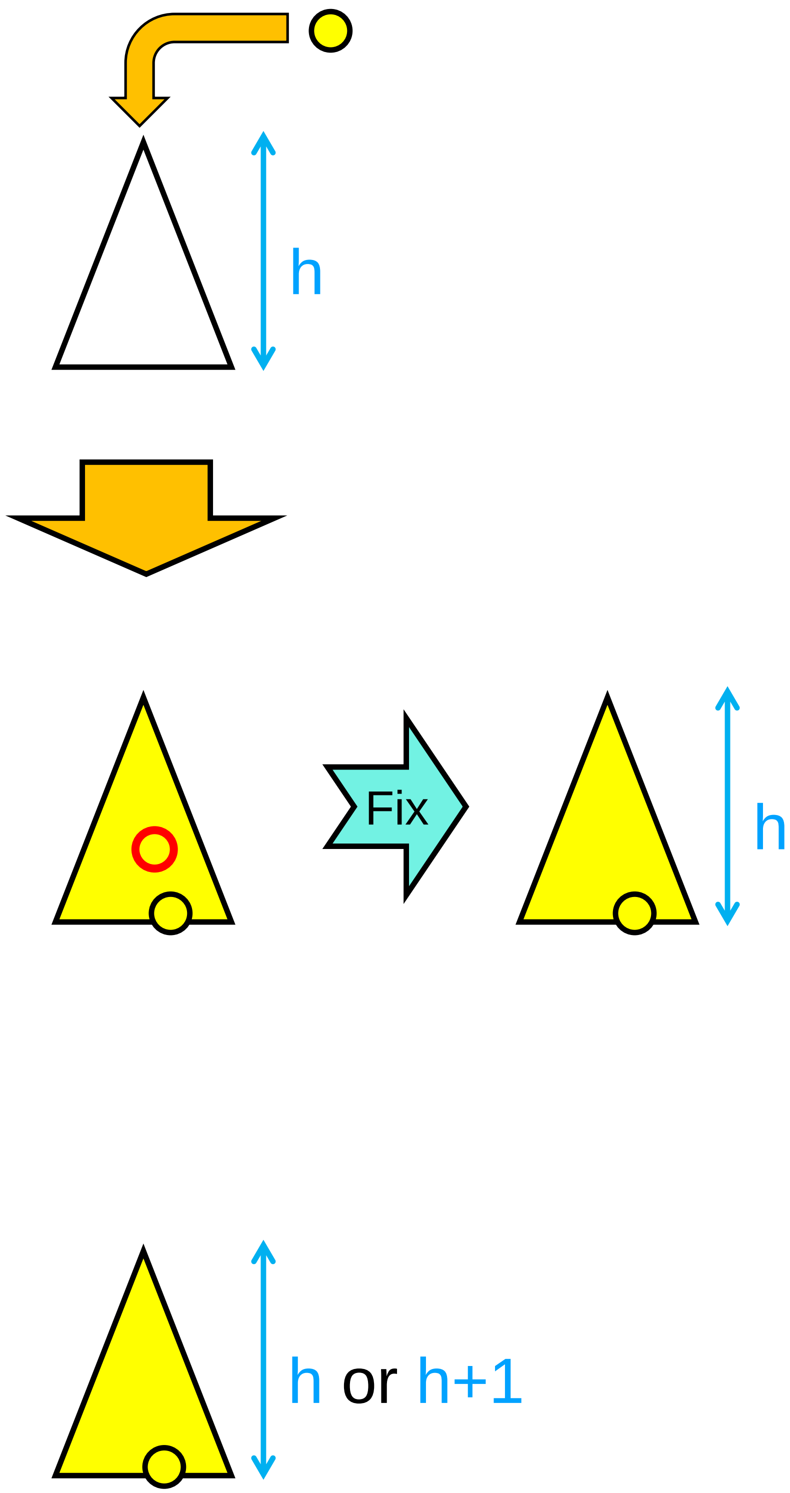
2. It is cheap to maintain — at most $O(\log n)$
   - each type of rotation costs O(1)
   - at most one rotation is needed for each insertion (We will see why next)

   So, maintaining the height invariant costs O(1) ✓

Height Analysis

Insertion into an AVL Tree

- Assume we are inserting a node into an AVL tree of height h



One of two things can happen:

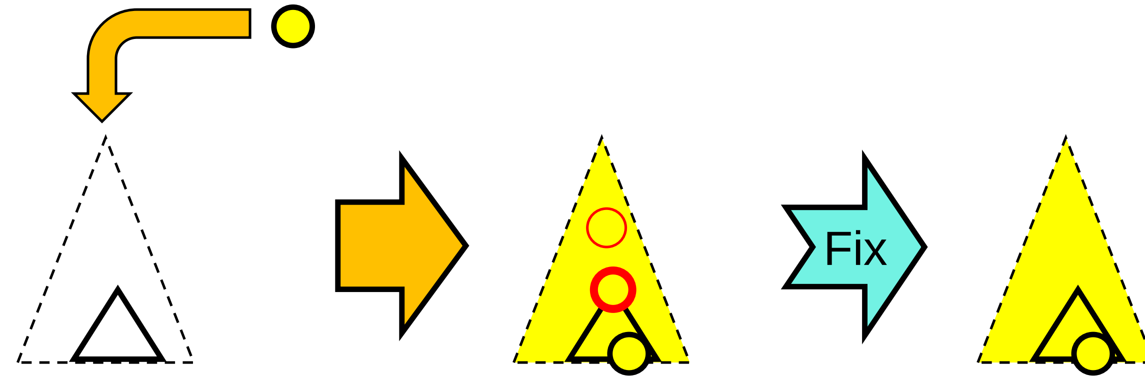
1. This causes a height violation
   - we fix it with a rotation
     - the resulting tree is a valid AVL tree
   - the fixed tree still has height h
     - the tree does not grow

Let's see why

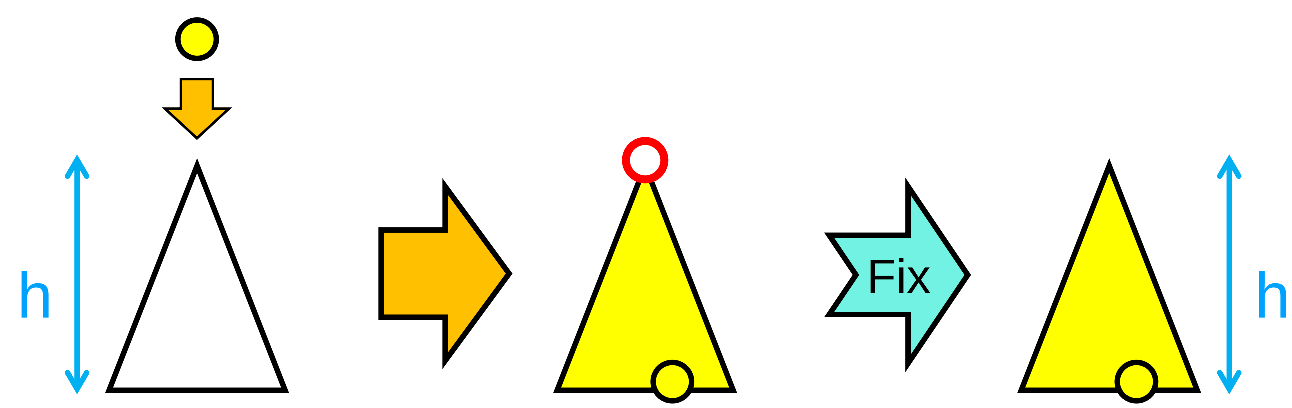
2. This does not cause a violation
   - the resulting tree has height h or h+1
     - the tree may grow only when there is no violation

Fixing the Lowest Violation

- Assume an insertion causes a violation
  - ➢ possibly more than one

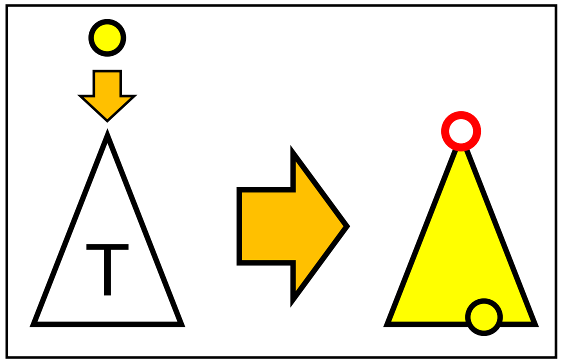


- We will focus on the subtree under the **lowest** violation
  - ○ We will find that fixing it yields a subtree with the **same height** h as the original subtree
  - ○ This necessarily resolves all violations above it
    - ➢ because the height of this subtree has not changed
    - ➢ if it satisfied the height invariant for the nodes above it before, it still satisfies it after



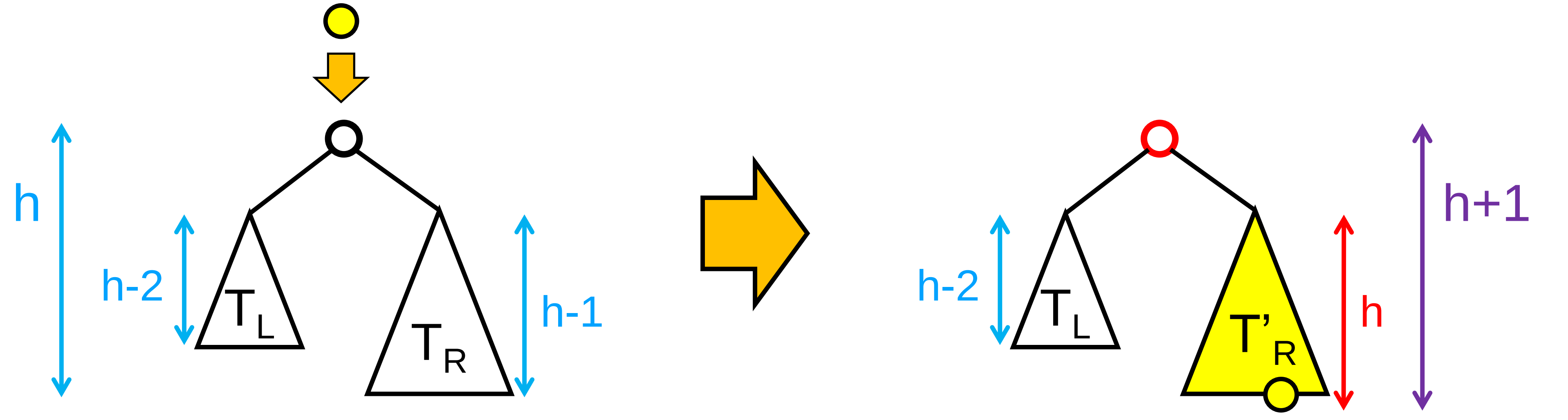
Fixing the lowest violation fixes the whole tree

The Lowest Violation



- Let's expand the tree
  - T cannot be empty — No violation possible
  - the new node can have been inserted in its left or right subtree

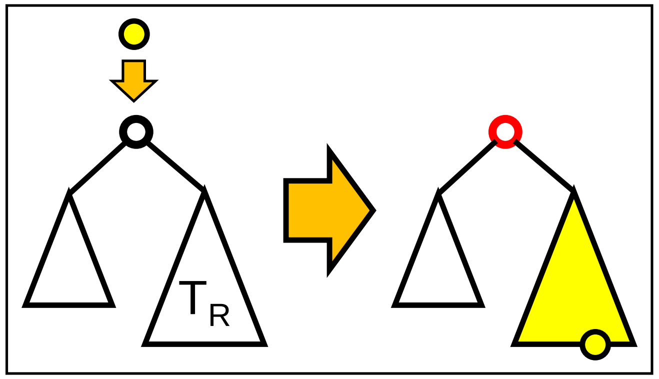
- Let's consider insertion in T_R — Insertion in T_L is symmetric



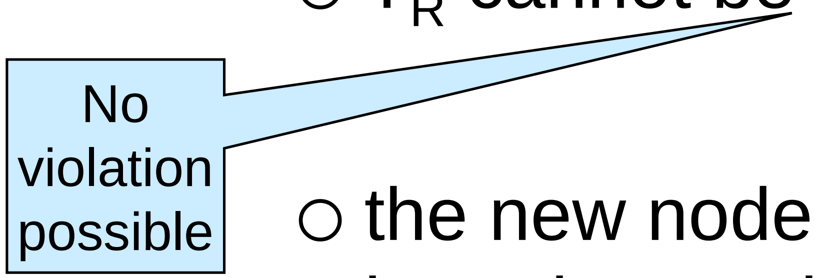
The right subtree has become too tall

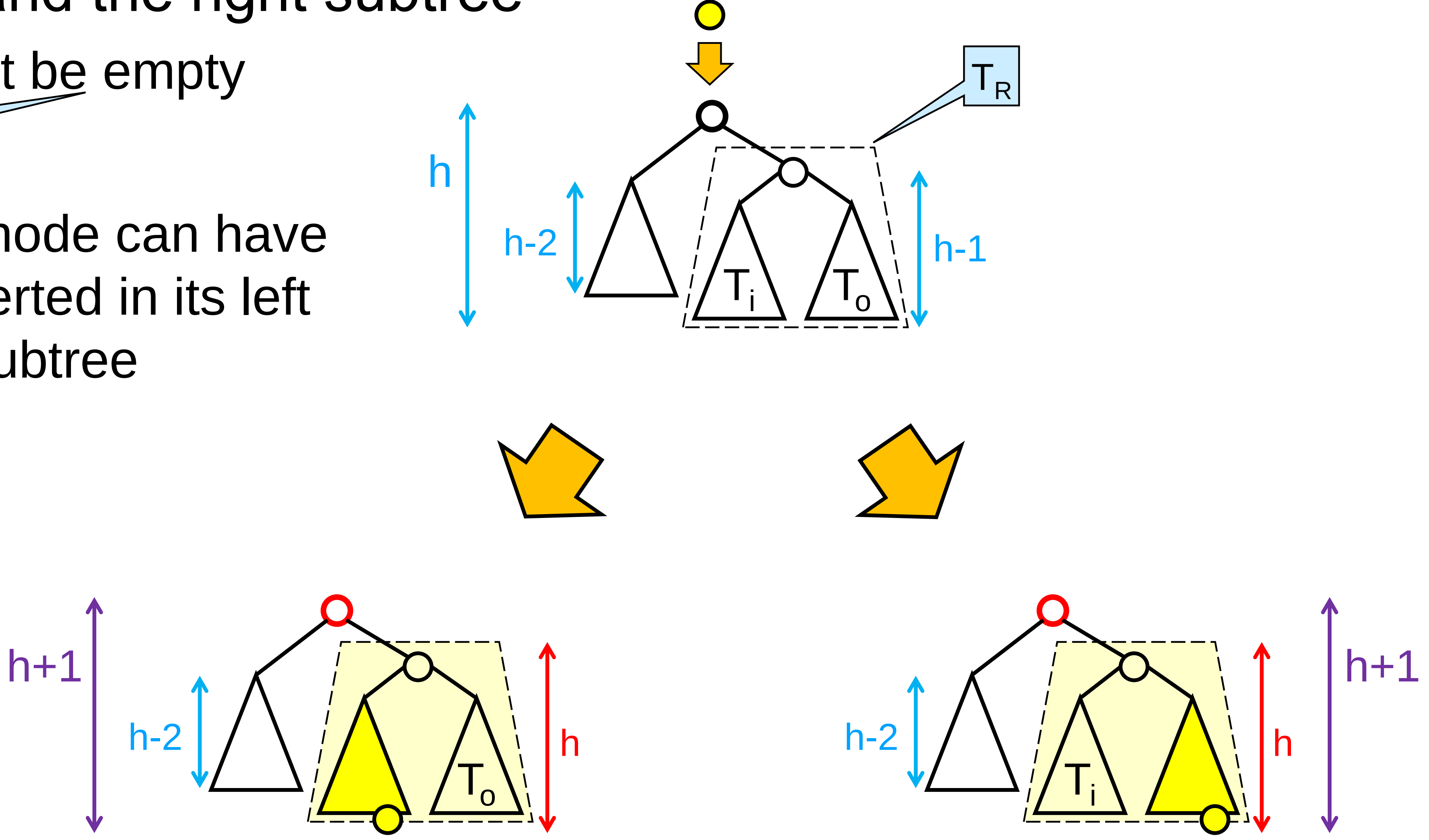
  - To have a violation
    - T_R must be taller than T_L
      - h-1 vs. h-2
    - T_R must have grown after the insertion
      - from h-1 to h

The Lowest Violation



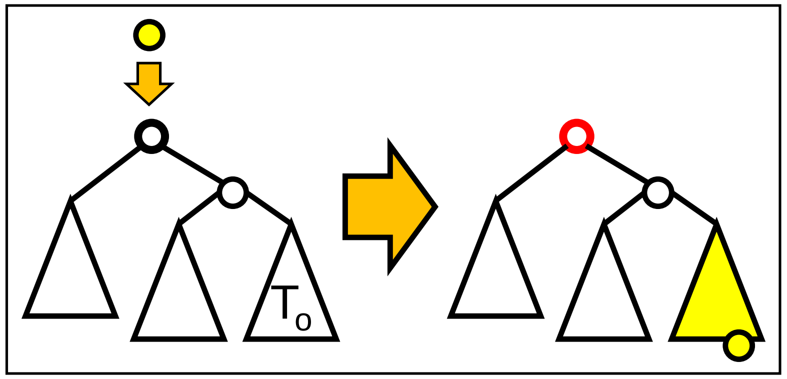
- Let's expand the right subtree
  - T_R cannot be empty
  - the new node can have been inserted in its left or right subtree



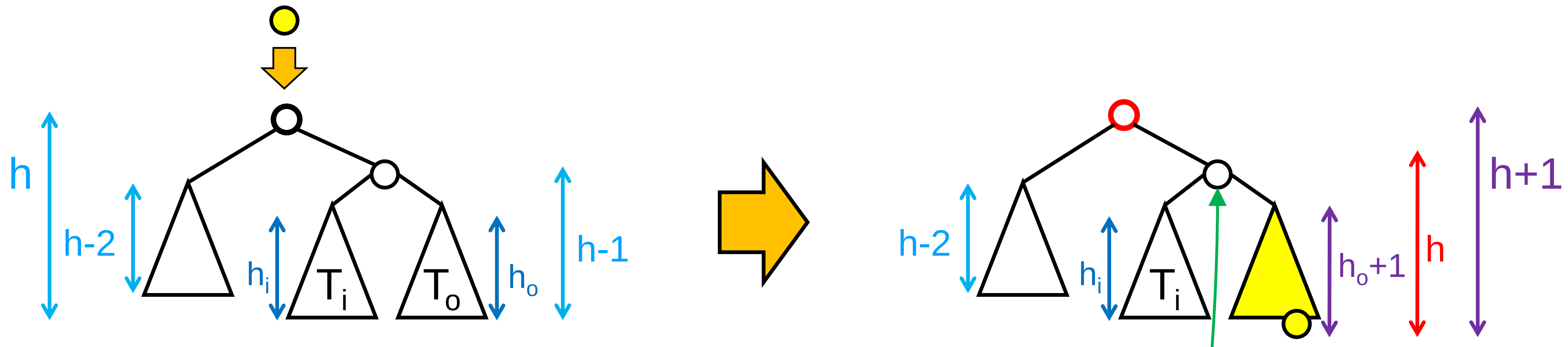


  - *Let's examine each case in turn*

Insertion in the Outer Subtree



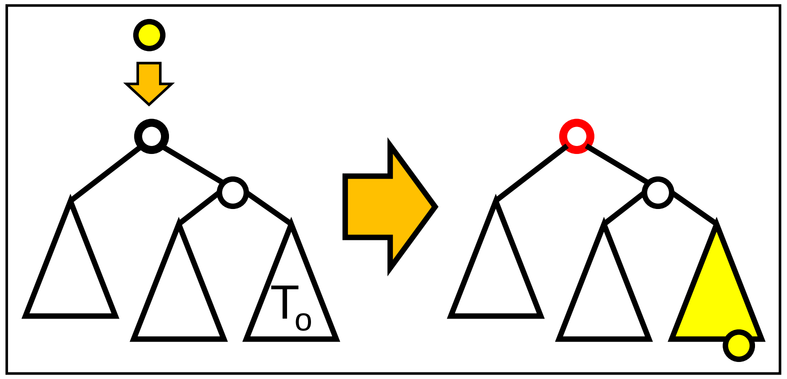
- How tall are T_i and T_o?



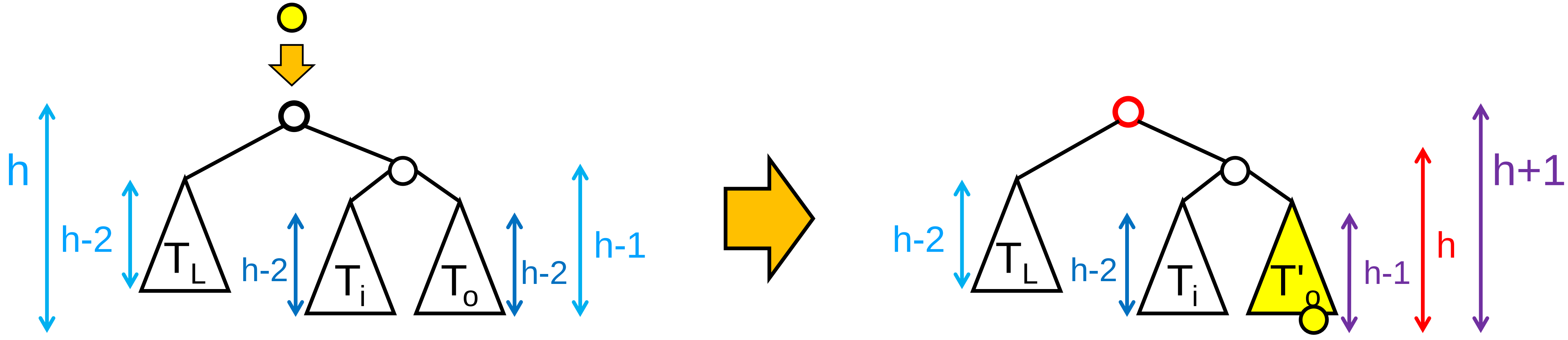
- ○ h_o = h-2
  - ➢ T_o needs to be as tall as possible to causes the violation
- ○ h_i = h_o = h-2
  - ➢ h_i may be either h-2 or h-3
  - ➢ but if h_i were h-3, the lowest violation would be here

T_i and T_o have the same height

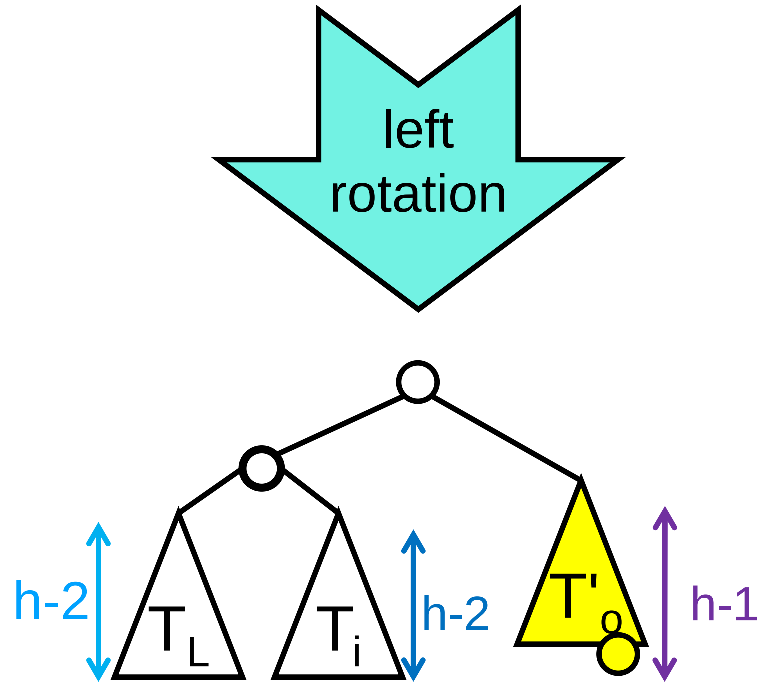
Insertion in the Outer Subtree



- T_i and T_o have height h-2

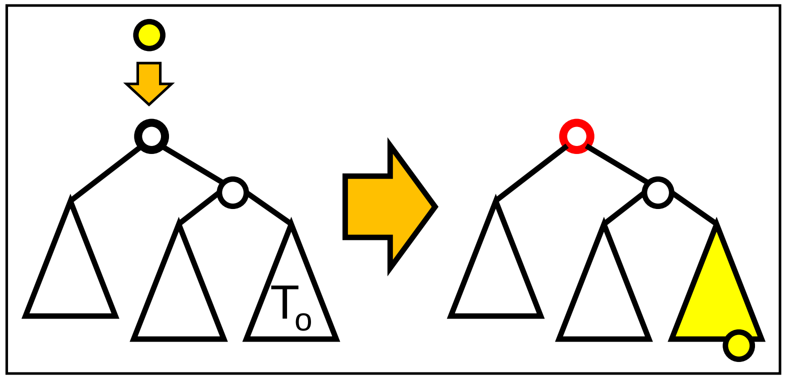


- This is the situation where we do a **single left rotation**

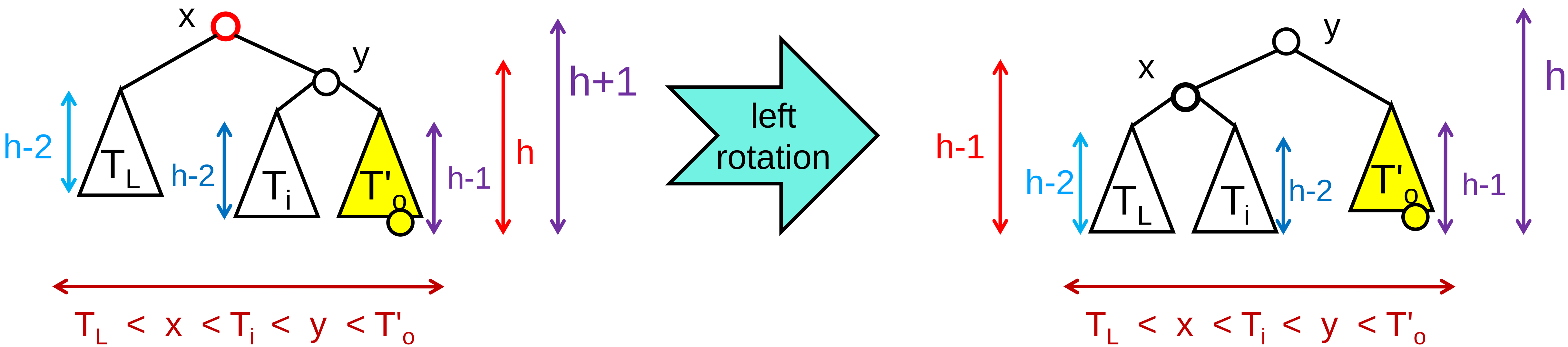


  - *Is this an AVL tree?*

Insertion in the Outer Subtree



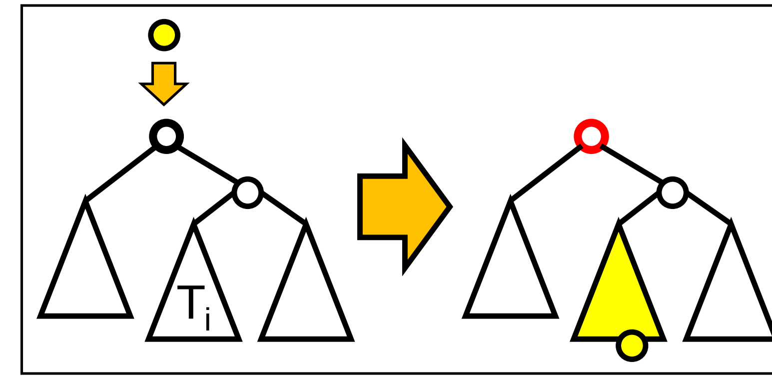
- Is this an AVL tree?



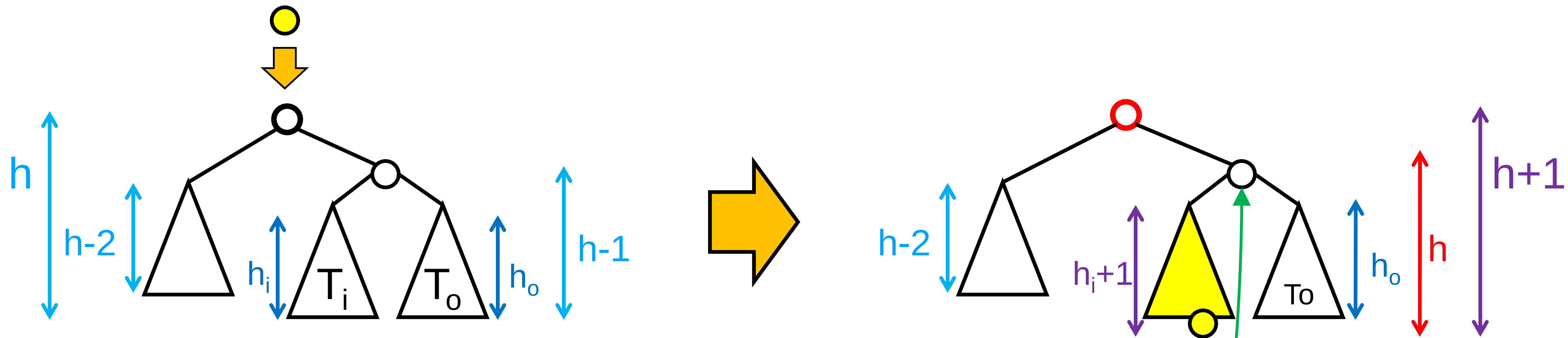
- BST insertion and the rotations maintains the ordering invariant
- T_L, T_i and T'_o are AVL trees
  - because x was the lowest violation
- T_L–x–T_i is an AVL tree of height h-1
  - because both T_L and T_i have height h-2
- (T_L–x–T_i)–y–T'_o is an AVL tree of height h
  - because T'_o also has height h-1

The height invariant is restored

Insertion in the Inner Subtree



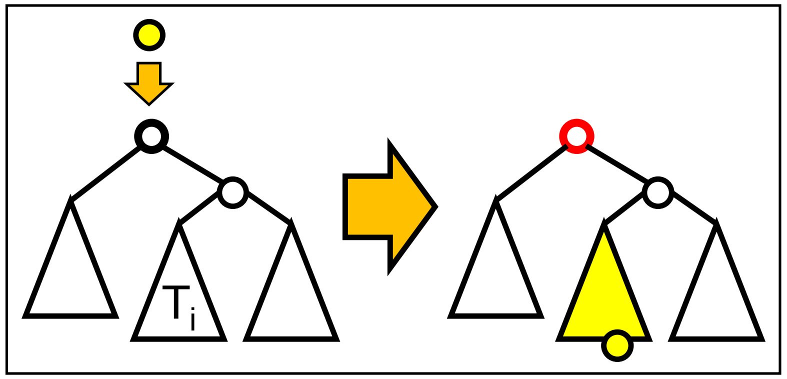
- How tall are T_i and T_o?



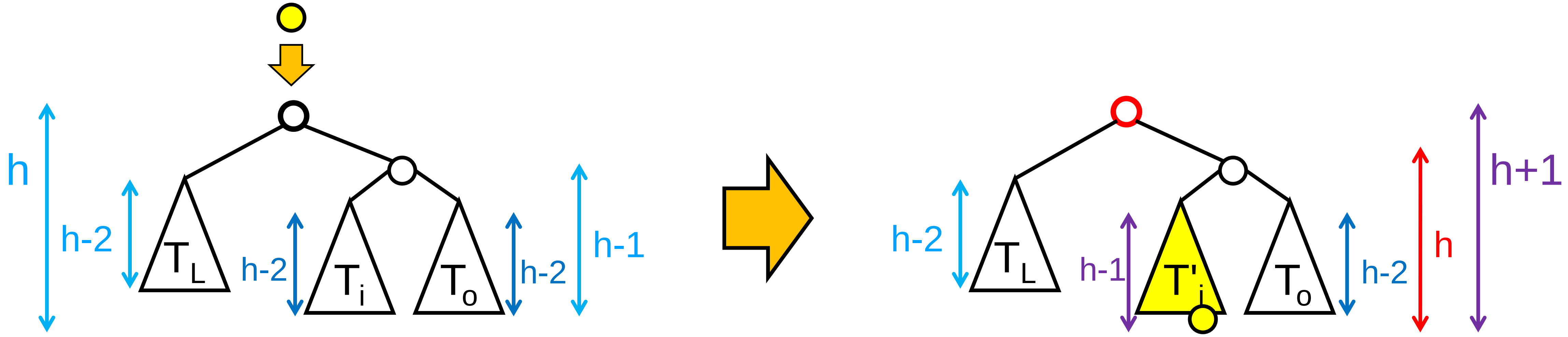
  - h_i = h-2
    - T_i needs to be as tall as possible to causes the violation
  - h_o = h_i = h-2
    - h_o may be either h-2 or h-3
    - but if h_o were h-3, the lowest violation would be here

T_i and T_o have the same height

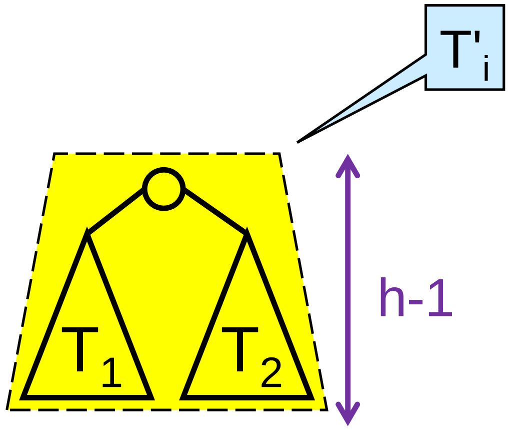
Insertion in the Inner Subtree

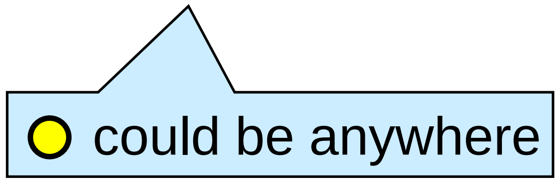


- T_i and T_o have height h-2

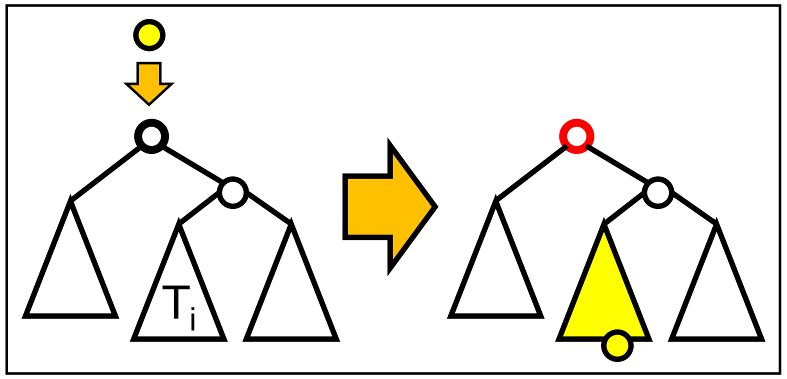


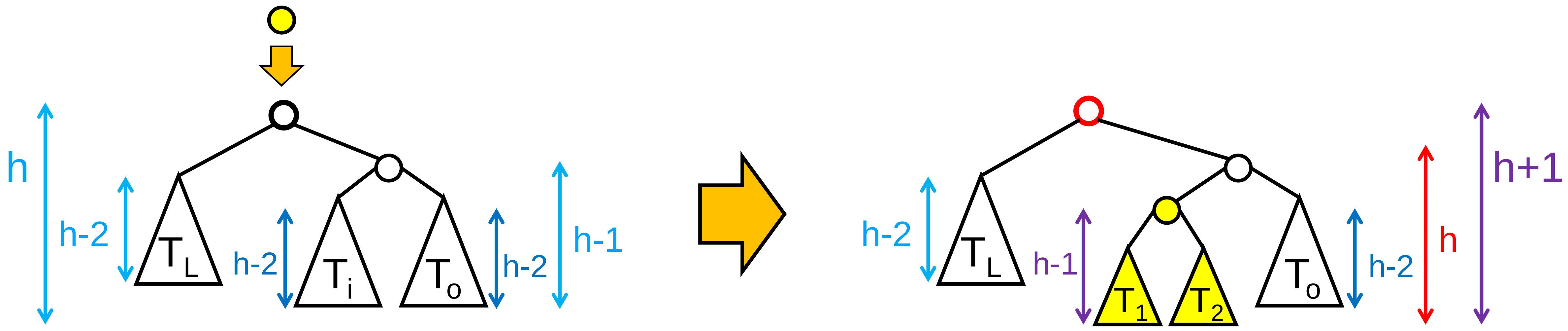
- T'_i contains at least the inserted node
  - let's expand it
  - T_1 and T_2 have height h-2 or h-3
    - one of them has height h-2
  - the inserted node could be
    - the root – if T_1 and T_2 are empty
    - in T_1
    - in T_2



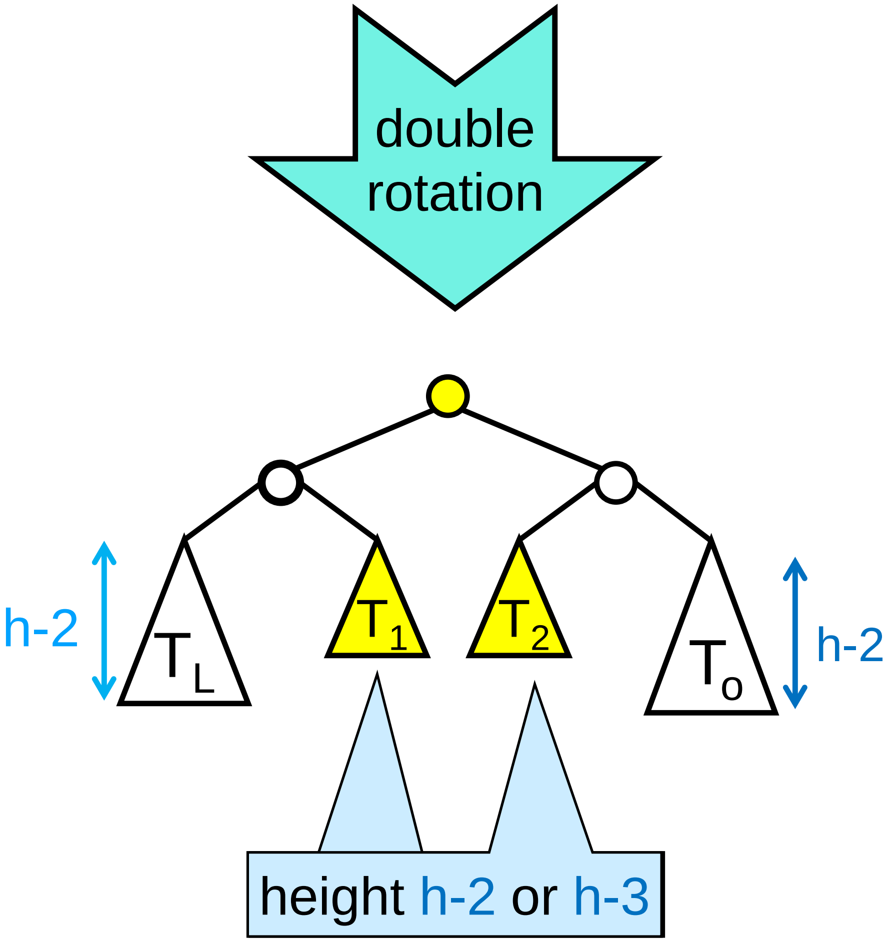


Insertion in the Inner Subtree



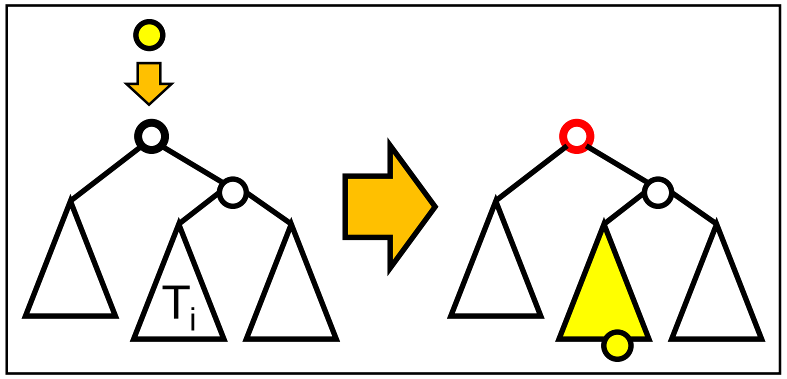


- This is the situation where we do a **double right/left rotation**

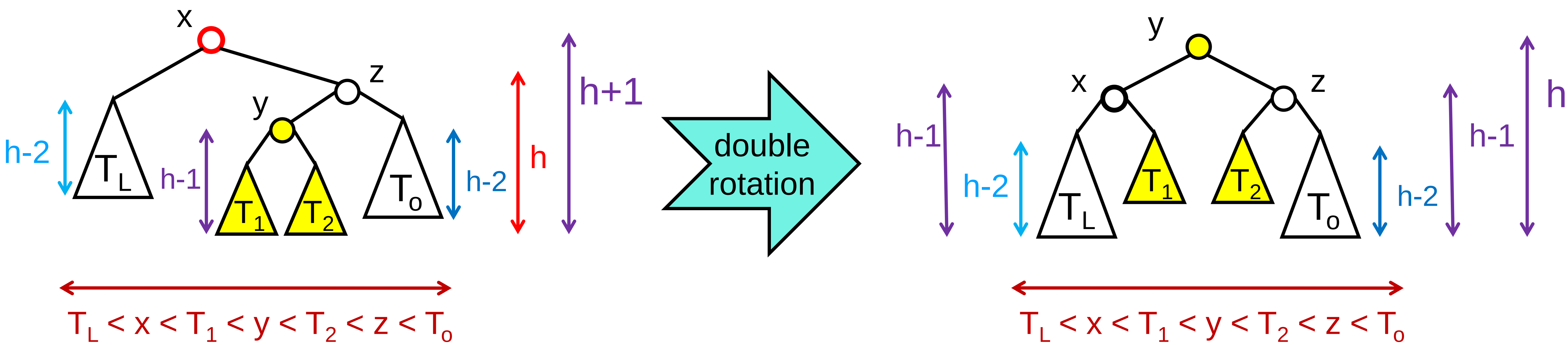


  - *Is this an AVL tree?*

Insertion in the Inner Subtree

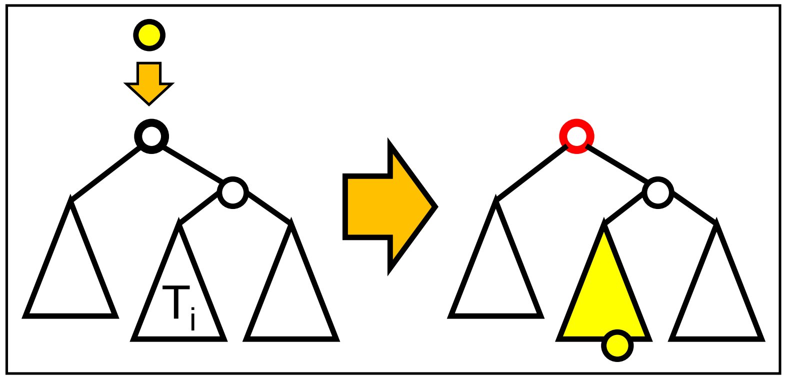


- Is this an AVL tree?

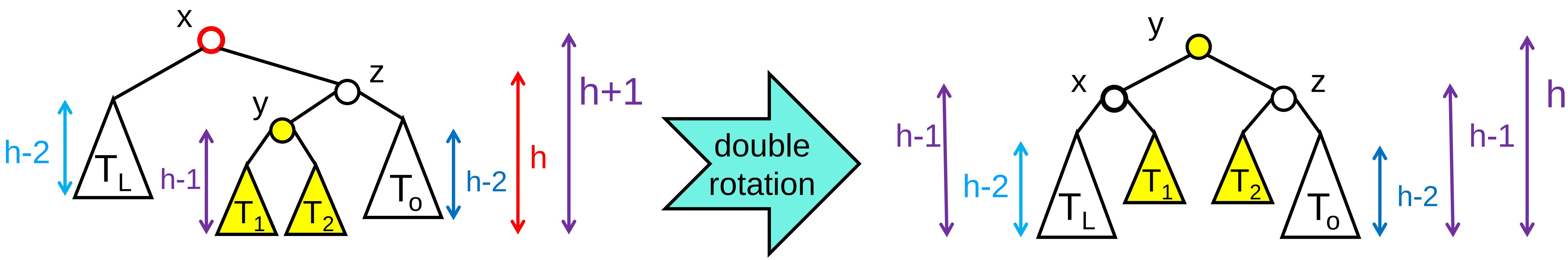


  - BST insertion and the rotations maintains the ordering invariant

Insertion in the Inner Subtree



- Is this an AVL tree?



- ○ T_L, T_1, T_2 and T_o are AVL trees
  - ➢ because x was the lowest violation
- ○ T_L–x–T_1 is an AVL tree of height h-1
  - ➢ because T_L has height h-2 and
  - ➢ T_1 has height either h-2 or h-3
- ○ T_2–z–T_o is an AVL tree of height h-1
  - ➢ because T_2 has height either h-2 or h-3
  - ➢ T_o has height h-2 and
- ○ (T_L–x–T_i)–y–(T_2–z–T_o) is an AVL tree of height h

The height invariant is restored

Summary

- When inserting into an AVL tree of height *h*
  - If there is no violation, the tree height remains *h* or grows to *h+1*
  - If there is a violation, the tree height remains *h*

- To fix a violation
  - perform a rotation on the **lowest violation**
    - a **single rotation** if the node was inserted in its **outer subtree**
    - a **double rotation** if the node was inserted in its **inner subtree**

- **One** rotation fixes the whole tree
  - The resulting tree is again an AVL tree
  - lookup, insert and find_min cost *O(log n)* in it
    - where *n* is the number of nodes

Implementation

The AVL Dictionary Interface

- This is exactly the same interface we had for BST dictionaries
  - the client can't tell the difference
    - except that it's much faster

Library Interface

```
// typedef ______* dict_t;

dict_t  dict_new()
  /*@ensures \result != NULL;                              @*/ ;

entry dict_lookup(dict_t D, key k)
  /*@requires D != NULL;                                   @*/
  /*@ensures \result == NULL
         || key_compare(entry_key(\result), k) == 0;  @*/ ;

void dict_insert(dict_t D, entry e)
  /*@requires D != NULL && e != NULL;                      @*/
  /*@ensures dict_lookup(D, entry_key(e)) == e;            @*/ ;

entry dict_min(dict_t D,)
  /*@requires D != NULL;                                   @*/ ;
```

Client Interface

```
// typedef ______* entry;
// typedef ______ key;

key entry_key(entry  e)
  /*@requires e != NULL;                   @*/ ;

int key_compare(key k1, key k2) ;
```

- We modify the BST *implementation* to use AVL trees

The AVL Dictionary Implementation

- We make surgical changes to the BST dictionary implementation
  - because AVL trees are BSTs and the BST implementation *mostly* works
- Specifically,
  - we extend the representation invariant to account the height invariant of AVL trees (3)
  - insert now needs to perform rotations to rebalance the tree when needed (2)
  - lookup and find_min remains unchanged (1)
    - because an AVL tree is a special case of a BST

Order in which we will examine them

avl_lookup

1

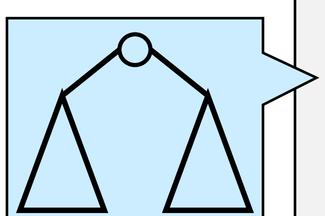
- The implementation remains unchanged
  - but we rename all the …bst… functions …avl…

```
entry avl_lookup(tree* T, key k)
//@requires is_avl(T);
//@ensures \result == NULL
              || key_compare(entry_key(\result), k) == 0;
{
  // Code for empty tree
  if (T == NULL) return NULL;

  // Code for non-empty tree
  int cmp = key_compare(k, entry_key(T->data));
  if (cmp == 0) return T->data;
  if (cmp < 0) return avl_lookup(T->left, k);
  //@assert cmp > 0;
  return avl_lookup(T->right, k);
}
```

We will implement it later

EMPTY



- If T is an AVL tree with *n* nodes, then
  - it has height *O(log n)*
  - so avl_lookup costs *O(log n)*

- find_min stays the same too
  - it now costs *O(log n)*

Inserting into an AVL Tree

```
tree* avl_insert(tree* T, entry e)
//@requires is_avl(T) && e != NULL;
//@ensures is_avl(\result) && \result != NULL;
//@ensures avl_lookup(\result, entry_key(e)) == e;
{
  // Code for empty tree
  if (T == NULL) return leaf(e);

  // Code for non-empty tree
  int cmp = key_compare(entry_key(e), entry_key(T->data));
  if (cmp == 0) T->data = e;
  else if (cmp < 0) {
    T->left = avl_insert(T->left, e);
    T = rebalance_left(T);
  else { //@assert cmp > 0;
    T->right = avl_insert(T->right, e);
    T = rebalance_right(T);
  }
  return T;
}
```

The tree layout does not change

added

added

- **After** each recursive call, we rebalance the tree
  - ➢ rebalance_left after an insertion in the **left** subtree
  - ➢ rebalance_right after an insertion in the **right** subtree
  - ○ This guarantees we fix the **lowest violation**
- For insert to cost *O(log n)*
  - ○ rebalance_left/right must cost *O(1)*

Let's look at one of them

rebalance_right

- We call it right after an insertion in the right subtree

```
tree* rebalance_right(tree* T)
//@requires T != NULL && T->right != NULL;
{
  if (height(T->right) - height(T->left) == 2) {   // violation!
    if (height(T->right->right) > height(T->right->left)) {
      // Single rotation
      T = rotate_left(T);
    } else
    { //@assert height(T->right->left) > height(T->right->right);
      // Double rotation
      T->right = rotate_right(T->right);
      T = rotate_left(T);
    }
  }
  return T;
}
```

The height invariant doesn't hold

Just return T if it holds

The insertion was in T->right

The insertion was in the **outer subtree**

we perform a **single rotation**

The insertion was in the **inner subtree**

we perform a **double rotation**

  - rebalance_right must have cost O(1)

rebalance_right

- We use the height of various subtrees to determine
  - if there is a violation
  - if the insertion happened in the inner or outer subtree

The height invariant doesn't hold

```
tree* rebalance_right(tree* T)
//@requires T != NULL && T->right != NULL;
{
  if (height(T->right) - height(T->left) == 2) {   // violation!
    if (height(T->right->right) > height(T->right->left)) {
      // Single rotation
      T = rotate_left(T);
    } else
    { //@assert height(T->right->left) > height(T->right->right);
      // Double rotation
      T->right = rotate_right(T->right);
      T = rotate_left(T);
    }
  }
  return T;
}
```

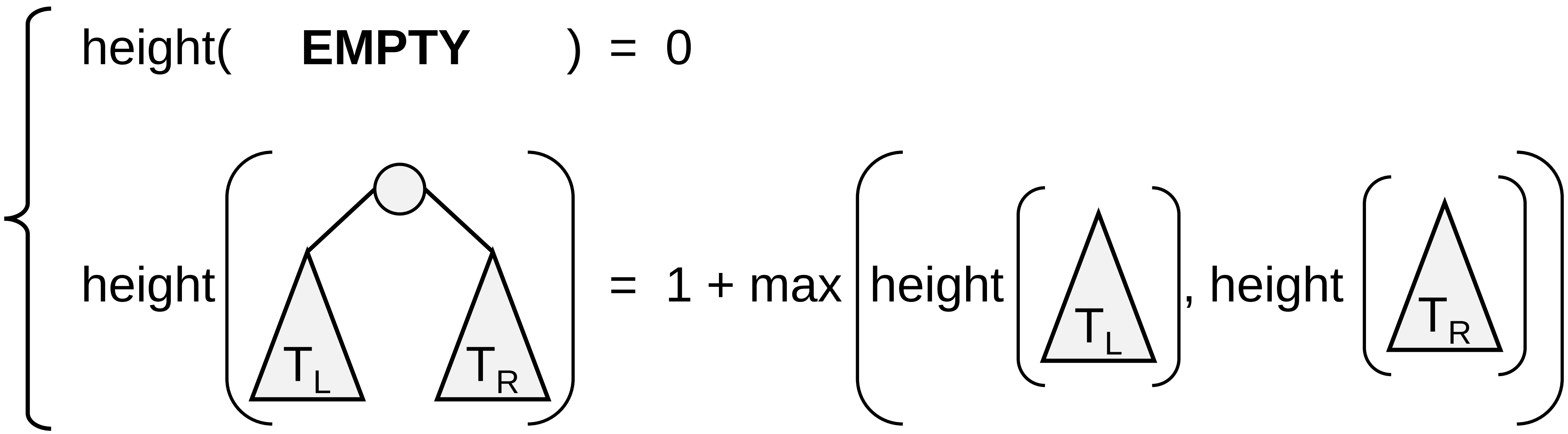
The insertion was in the **outer subtree**

The insertion was in the **inner subtree**

  - rebalance_right must have cost O(1)
    - so height, rotate_left and rotate_right must cost O(1)

height

- We can transcribe the mathematical definition



and get

```
int height(tree* T)
//@requires is_tree(T);
//@ensures \result >= 0;
{
  if (T == NULL) return 0;
  return 1 + max(height(T->left), height(T->right));
}
```

height

- By transcribing the mathematical definition, we get

```
int height(tree* T)
//@requires is_tree(T);
//@ensures \result >= 0;
{
  if (T == NULL) return 0;
  return 1 + max(height(T->left), height(T->right));
}
```

  - If T has *n* nodes, height(T) costs *O(n)*
    - it recursively goes over every node in T

- But we need height to cost *O(1)*
  - otherwise insert will cost more than *O(log n)*

- *What can we do?*

height

- Rather than computing the height of a tree by traversing it, we can **store** it
  - we add a **height field** in each node

```
typedef struct tree_node tree;
struct tree_node {
  tree* left;
  int data;
  tree* right;
  int height;  // >= 0
};
```

- Then, the function height simply returns the contents of this field
  - or 0 if T is NULL
  - Its cost is now *O(1)*

```
int height(tree* T)
//@requires is_tree(T);
//@ensures \result >= 0;
{
  return T == NULL ? 0 : T->height;
}
```

Return 0 if T is NULL and T->height otherwise

- This is a **space-time tradeoff**
  - we are using a bit of extra space to save a lot of time

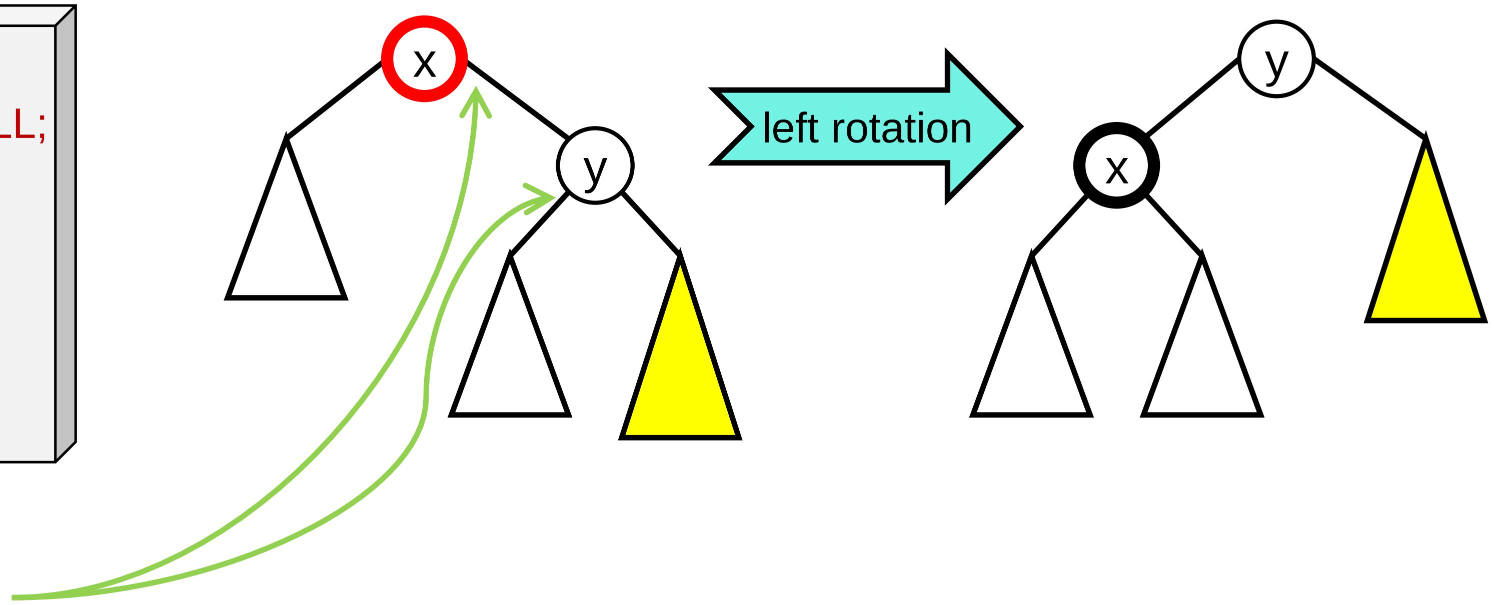
The new height field in the nodes

Computing the height of the tree over and over

Rotations

- We implement single rotations by transcribing the figure

```
tree* rotate_left(tree* T)
//@requires T != NULL && T->right != NULL;
{
  tree* temp = T->right;
  T->right = T->right->left;
  temp->left = T;
  return temp;
}
```



by updating two pointers

  - The cost is O(1)

- We implement double rotations as two single rotations
  - The cost is O(1)

from rebalance_right

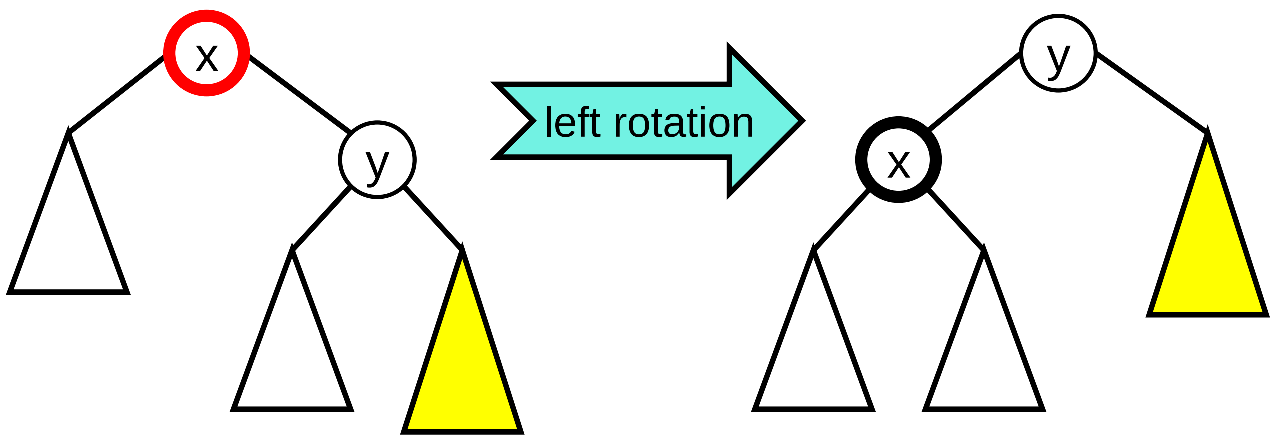
```
// Double rotation
T->right = rotate_right(T->right);
T = rotate_left(T);
```

- *Can it be this simple?*

Rotations

- *Can it be this simple?*

```
tree* rotate_left(tree* T)
//@requires T != NULL && T->right != NULL;
{
  tree* temp = T->right;
  T->right = T->right->left;
  temp->left = T;
  return temp;
}
```



- The height fields of nodes x and y are now wrong! ✘
  - We need to update them
  - We can do so based on the height of their subtrees
- Let's write a general function:
  - ➢ fix_height costs O(1)
    - ❑ because height costs O(1)

```
void fix_height(tree* T)
//@requires is_tree(T) && T != NULL;
{
  int hl = height(T->left);
  int hr = height(T->right);
  T->height = 1 + max(hl, hr);
}
```

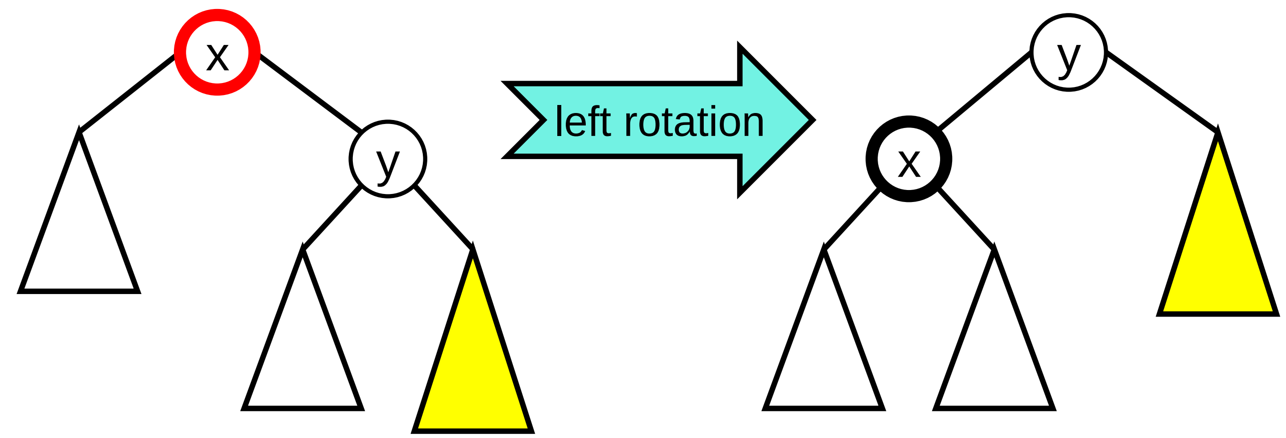
Rotations Revisited

- We implement single rotations by transcribing the figure

```
tree* rotate_left(tree* T)
//@requires T != NULL && T->right != NULL;
{
  tree* temp = T->right;
  T->right = T->right->left;
  temp->left = T;
  fix_height(T);
  fix_height(temp);
  return temp;
}
```

node x

node y



by updating two pointers

and then fixing the height of the affected nodes ✓

- rotate_left costs O(1)

rebalance_right Revisited

- We also need to fix the height when there is no violation

```
tree* rebalance_right(tree* T)
// T must be immediate result of a right-insertion
//@requires T != NULL && T->right != NULL;
{
  if (height(T->right) - height(T->left) == 2) {   // violation!
    if (height(T->right->right) > height(T->right->left)) {
      // Single rotation
      T = rotate_left(T);
    } else {
      //@assert height(T->right->left) > height(T->right->right);
      // Double rotation
      T->right = rotate_right(T->right);
      T = rotate_left(T);
    }
  } else { // No rotation needed, but tree may have grown
    fix_height(T);
  }
  return T;
}
```

When we handle a violation, the rotations fix the heights

Fixes the heights when no rotation was performed

✓

```
typedef struct tree_node tree;
struct tree_node {
  tree* left;
  int data;
  tree* right;
  int height;  // >= 0
};
```

New Leaves

- When insertion creates a new leaf, we need to set its height to 1

```
tree* leaf(entry e)
//@requires e != NULL;
//@ensures is_avl(\result) && \result != NULL;
{
  tree* T = alloc(tree);
  T->data = e;
  T->left = NULL;     // not necessary
  T->right = NULL;   // not necessary
  T->height = 1;
  return T;
}
```

Representation Invariants

The AVL Representation Invariant

- An AVL tree is a BST that satisfies the height invariant
  - additionally, the height fields must all contain the true height

```
bool is_specified_height(tree* T)
//@requires is_tree(T);
{
  if (T == NULL) return true;
  return is_specified_height(T->left)      // height(T->left)  is correct
     && is_specified_height(T->right)      // height(T->right) is correct
     && T->height == max(height(T->left),
                         height(T->right)) + 1;   // height(T) is correct
}

bool is_balanced(tree* T)
//@requires is_tree(T);
{
  if (T == NULL) return true;
  return abs(height(T->left) - height(T->right)) <= 1
     && is_balanced(T->left)
     && is_balanced(T->right);
}

bool is_avl(tree* T) {
  return is_tree(T) && is_ordered(T, NULL, NULL)
     && is_specified_height(T)
     && is_balanced(T);
}
```

Checks that the height field in each node contains the true height of its subtree (is_specified_height)

Checks the height invariant (is_balanced)

The AVL representation invariant (is_avl)

is_tree(T) && is_ordered(T, NULL, NULL): our old is_bst

is_specified_height(T): checks the height

is_balanced(T): checks the height invariant

- We can use them to give precise contracts to all other functions

avl_insert Revisited

- We can track the representation invariants at each step of insertion

```
tree* avl_insert(tree* T, entry e)
//@requires is_avl(T) && e != NULL;
//@ensures is_avl(\result) && \result != NULL;
//@ensures avl_lookup(\result, entry_key(e)) == e;
{
  // Code for empty tree
  if (T == NULL) return leaf(e);

  // Code for non-empty tree
  //@assert is_avl(T->left) && is_avl(T->right);      <- added
  int cmp = key_compare(entry_key(e), entry_key(T->data));
  if (cmp == 0) T->data = e;
  else if (cmp < 0) {
    T->left = avl_insert(T->left, e);
    //@assert is_avl(T->left) && is_avl(T->right);    <- added
    T = rebalance_left(T);
     //@assert is_avl(T);                             <- added
  else { //@assert cmp > 0;
    T->right = avl_insert(T->right, e);
    //@assert is_avl(T->left) && is_avl(T->right);    <- added
    T = rebalance_right(T);
     //@assert is_avl(T);                             <- added
  }
  return T;
}
```

If T is an AVL tree, its subtrees are too

T->left is an AVL tree by the postcondition of avl_insert

T->right did not change

rebalance_left restores T into a valid AVL tree

Similar

rebalance_right Revisited

- rebalance_right
  - takes a tree whose two subtrees are AVL trees
    - but itself may not be a valid AVL tree
  - return an AVL tree

This is what we learned from avl_insert

```
tree* rebalance_right(tree* T)
// T must be immediate result of a right-insertion
//@requires T != NULL && T->right != NULL;
//@requires is_avl(T->left) && is_avl(T->right);
//@ensures is_avl(\result);
{
  if (height(T->right) - height(T->left) == 2) {   // violation!
    if (height(T->right->right) > height(T->right->left)) {
      // Single rotation
      T = rotate_left(T);
    } else {
      //@assert height(T->right->left) > height(T->right->right);
      // Double rotation
      T->right = rotate_right(T->right);
      T = rotate_left(T);
    }
  } else { // No rotation needed, but tree may have grown
    fix_height(T);
  }
  return T;
}
```

but T itself may not be an AVL tree

T may not be an AVL tree

T is again an AVL tree

T may not be an AVL tree

T is again an AVL tree

T may not be an AVL tree

T is again an AVL tree

✓

Rotations revisited

- We expect rotate_left to
  - takes a tree whose two subtrees are AVL trees
    - but itself may not be a valid AVL tree
  - return an AVL tree

```
tree* rotate_left(tree* T)
//@requires T != NULL && T->right != NULL;
//@requires is_avl(T->left) &&  is_avl(T->right);
//@ensures is_avl(\result);
{
  tree* temp = T->right;
  T->right = T->right->left;
  temp->left = T;
  fix_height(T);
  fix_height(temp);
  return temp;
}
```

but T itself may not be an AVL tree

- This would be true if used to implement single rotations **only**
- But we are also using it to implement double rotations
  - these contracts **do not hold** in this case

```
// Double rotation
T->right = rotate_right(T->right);
T = rotate_left(T);
```

Rotations revisited

- Because we implement double rotations using single rotations, we must deploy weaker contracts

```
tree* rotate_left(tree* T)
//@requires T != NULL && T->right != NULL;
//@requires is_specified_height(T->left);
//@requires is_specified_height(T->right);
//@ensures is_specified_height(\result);
{
  tree* temp = T->right;
  T->right = T->right->left;
  temp->left = T;
  fix_height(T);
  fix_height(temp);
  return temp;
}
```

This only says that the heights are right

Maintaining the Height

```
typedef struct tree_node tree;
struct tree_node {
  tree* left;
  int data;
  tree* right;
  int height;  // >= 0
};
```

- We can use the same contracts in fix_height

```
void fix_height(tree* T)
//@requires is_tree(T) && T != NULL;
//@requires is_specified_height(T->left);
//@requires is_specified_height(T->right);
//@ensures is_specified_height(T);
{
  int hl = height(T->left);
  int hr = height(T->right);
  T->height = (hl > hr ? hl+1 : hr+1);
}
```

Assuming the subtrees have valid height fields, it will make the height field in the whole tree valid