Lecture 4*

AVL Trees

Data Structures

Shiri Chechik, Or Zamir

Winter semester 2025-6

* based on the TAU course slides, edited by AR and TAUOnline

Definition

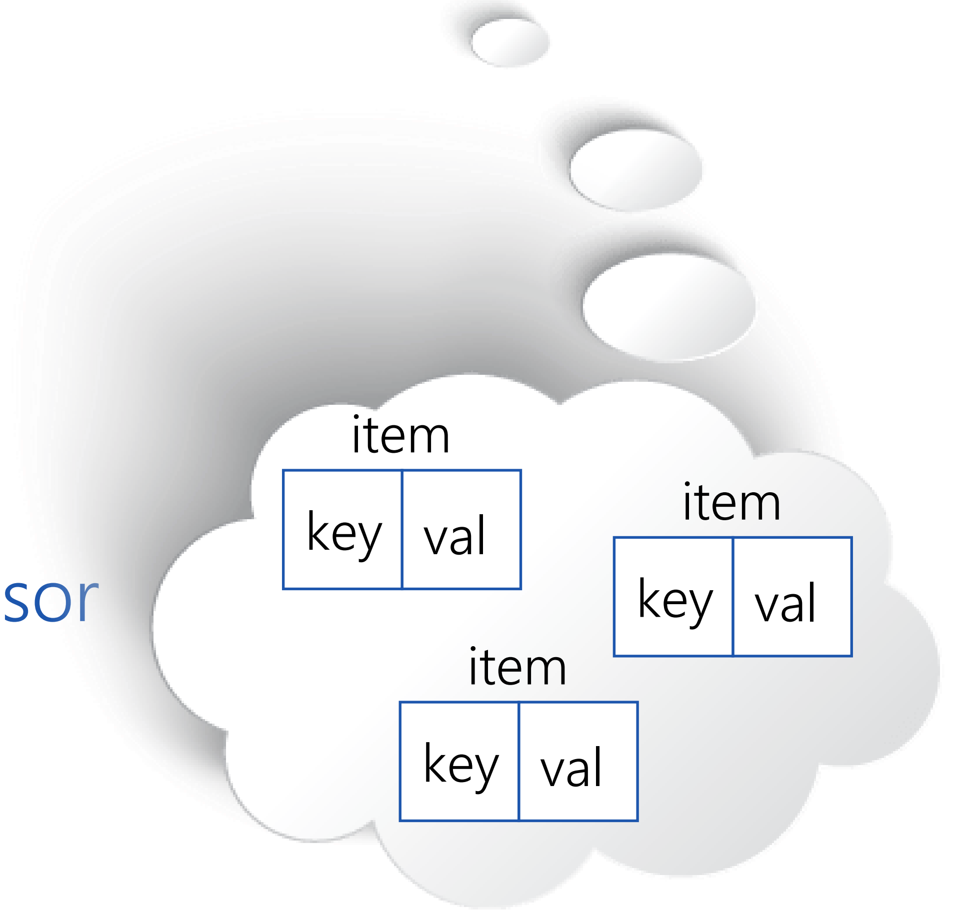
What Is an AVL tree?

ADT Dictionary - Reminder

- Maintain a set of items, with keys and associated values
  - Assume keys are distinct

- Support insert, delete, search

- If keys are totally ordered
  support min/max and successor/predecessor



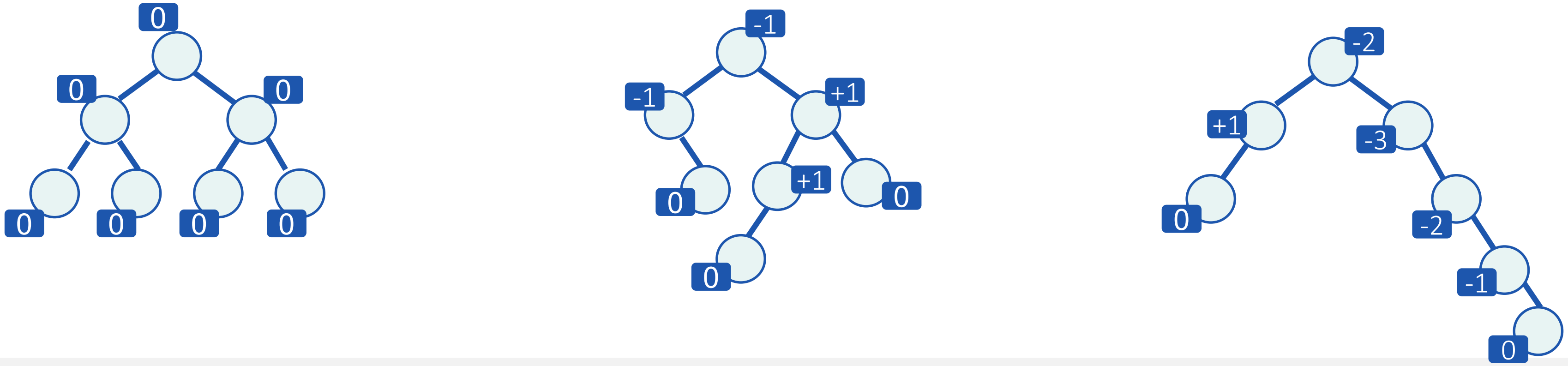
Motivation

- We saw a dictionary implementation using binary search trees. The complexity of the above operations is $O(h+1)$ worst case, when h is the tree height.
  - h can be between $O(\log n)$ and $O(n)$
  - When $h = O(\log n)$ the tree is called **balanced**.

- Important examples:
  - **AVL trees** (*this lesson*) and their improved version, **WAVL trees**
  - **B trees**, **B$^+$ trees**
  - **Red-Black trees**

AVL Trees

AVL trees were invented by Adelson-Velsky and Landis in 1962.



<u>Balance factor</u>

$$\mathrm{BF}(v) = h(v.left) - h(v.right)$$

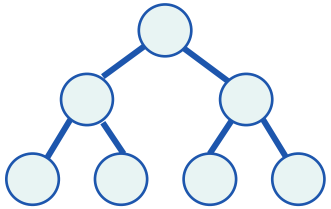
Reminder: The height of an empty tree is set to be -1.

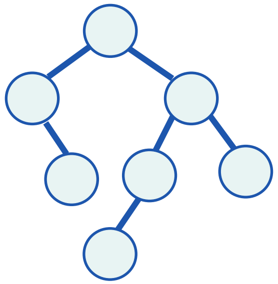
AVL Trees

Definition:
An **AVL** tree is a binary search tree where each node v maintains the following attribute:

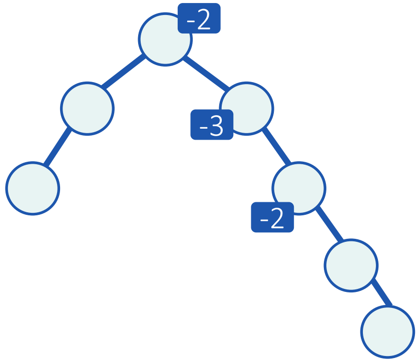
$$|BF(v)| \leq 1$$

Examples:





Counter example:



Upper Bound of an AVL Tree Height

Claim: An AVL tree with n nodes satisfies $h = O(\log n)$.

Proof: next

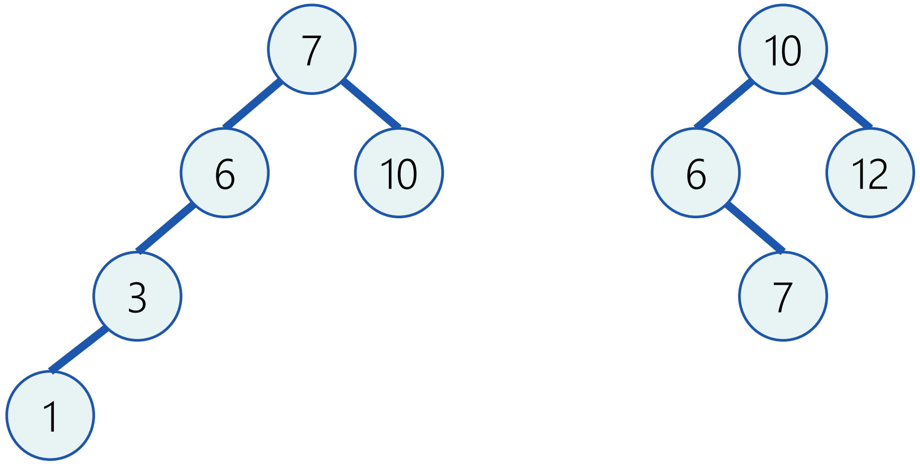
Conclusion

All queries (search, minimum/maximum, predecessor/successor) are executed in $O(\log n)$ time **worst case** in AVL trees.

AVL Tree Is a Balanced BST

And what about inserting and deleting an item?

Also in logarithmic time, but these operations may violate the tree balance and create nodes that are “AVL criminals”. For example:



Later on we will understand how to deal with this issue.

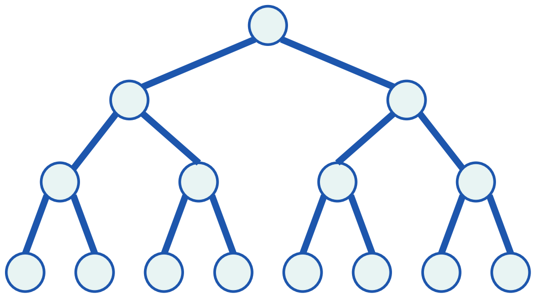
Upper Bound of an AVL Tree Height

Intuition

In a balanced tree $h = O(\log n)$. Meaning $n = \Omega(\alpha^h)$, for some constant $\alpha > 1$.

Literally: the number of nodes in a balanced tree is exponential in the height of the tree.

For example, in a perfectly balanced binary search tree (complete binary tree),
$h = O(\log_2 n)$ and $n = \Omega(2^h)$.



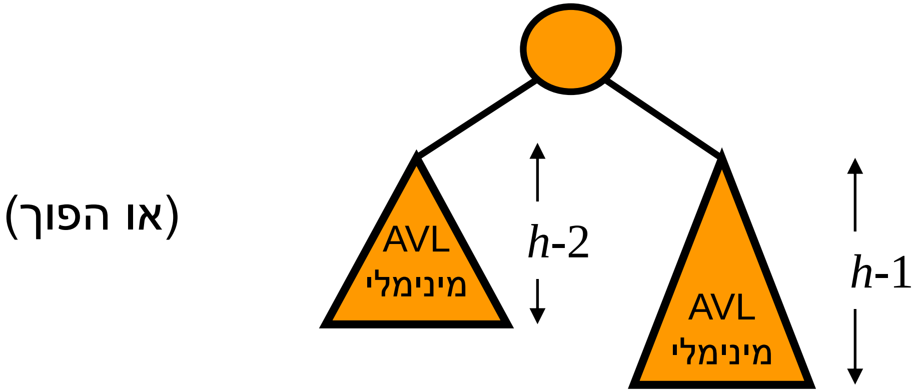
We will prove that the above holds for an **AVL tree** with the constant $\Phi = \frac{1+\sqrt{5}}{2} \approx 1.618$ (**golden ratio**), hence $h = O(\log_\Phi n)$.

Proof That AVL is Balanced

$h = O(logn)$ for an AVL Tree

חסם לגובה עץ AVL - הוכחה

כיצד נראה עץ AVL בגובה h בעל מספר הצמתים מינימלי ?



עץ כזה נקרא **עץ פיבונאצ'י**.

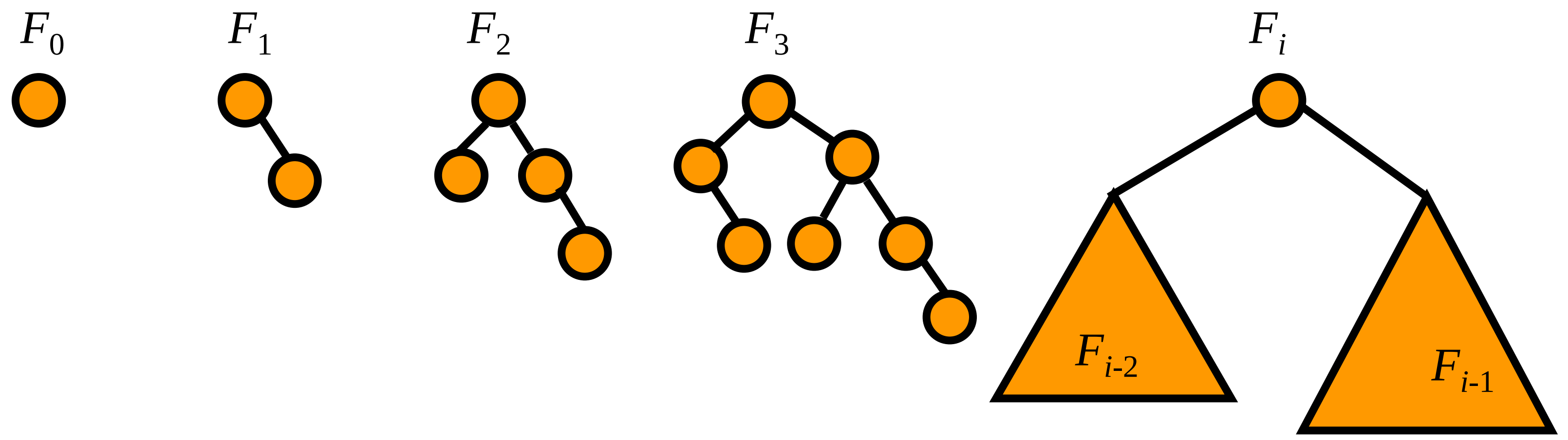
סדרת פיבונאצ'י: $f_n = f_{n-1} + f_{n-2} \qquad f_1 = f_2 = 1$

איברים ראשונים בסדרה: $1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$

ידוע (הוכחה בבדידה 2) כי: $f_n = \dfrac{\Phi^n - \overline{\Phi}^n}{\sqrt{5}}$ כאשר $\Phi = \dfrac{1+\sqrt{5}}{2} \approx 1.618$ $\qquad \overline{\Phi} = 1 - \Phi = \dfrac{1-\sqrt{5}}{2} \approx -0.618$

חסם לגובה עץ AVL – הוכחה (2)

הגדרת עצי פיבונאצ'י ברקורסיה:



<u>תכונות (תרגיל: הוכיחו כל אחת מהתכונות)</u>

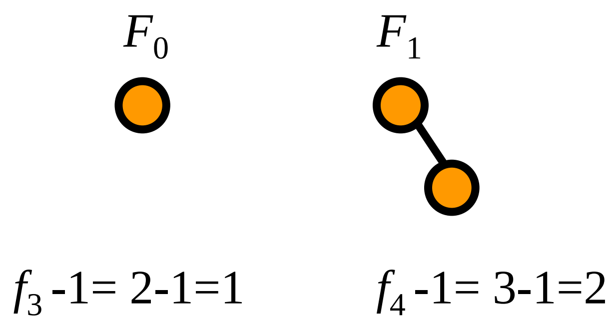
1. גובהו של F_h הוא h.
2. $|F_h| = |F_{h\text{-}1}| + |F_{h\text{-}2}| + 1$ (כאשר $|F_i|$ הוא מספר הצמתים ב- F_i).
3. F_h הוא עץ AVL בעל מספר צמתים מינימלי מבין כל עצי ה- AVL בגובה h.

חסם לגובה עץ AVL – הוכחה (3)

<u>טענה</u>: מספר הצמתים ב- F_h הוא $|F_h| = f_{h+3} - 1$ צמתים כאשר f_i הוא מספר פיבונאצ'י ה- i.

<u>הוכחה</u>: באינדוקציה על h.

<u>בסיס</u>: עבור $h=0$ ו- $h=1$ הטענה מתקיימת:



<u>צעד</u>: נניח שהטענה נכונה לכל $h' < h$.

$$|F_h| = |F_{h-1}| + |F_{h-2}| + 1 = (f_{h+2} - 1) + (f_{h+1} - 1) + 1 = f_{h+3} - 1$$

חסם לגובה עץ AVL – הוכחה (4)

אם כן, עבור עץ AVL בעל n צמתים וגובה h מתקיים :

$$n \geq |F_h|$$

$$n \geq |F_h| = f_{h+3} - 1 = \frac{\Phi^{h+3} - \overline{\Phi}^{h+3}}{\sqrt{5}} - 1 \geq \frac{\Phi^{h+3}}{\sqrt{5}} - 2$$

$$\sqrt{5}(n+2) \geq \Phi^{h+3}$$

$$h + 3 \leq \log_{\Phi}\left(\sqrt{5}(n+2)\right)$$

$$h \leq \log_{\Phi}(n+2) + \log_{\Phi}\left(\sqrt{5}\right) - 3 = O(\log n)$$

ניתן אף להראות (עם עוד טיפה מאמץ):

$$h \leq \log_{\Phi} n \sim 1.44 \log_2 n$$

Insertion Into AVL

How to Keep It Balanced?

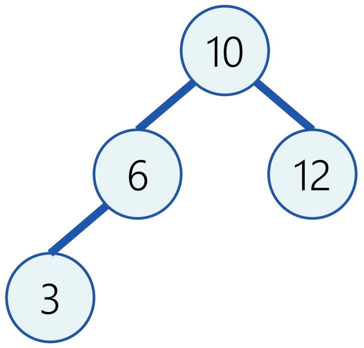
Fixing After Insertion

Rotations

What can we do if an insertion created "AVL criminals"?

Rotation: change of a few pointers in order to balance the height difference.

Here are some simple examples:



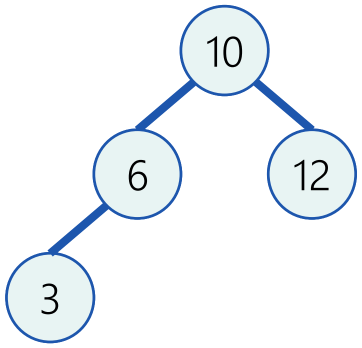
Fixing After Insertion

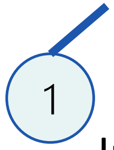
Rotations

What can we do if an insertion created "AVL criminals"?

Rotation: change of a few pointers in order to balance the height difference.

Here are some simple examples:





Insert (1)

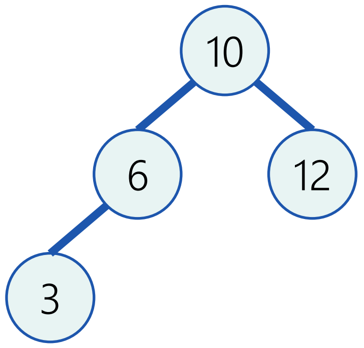
Fixing After Insertion

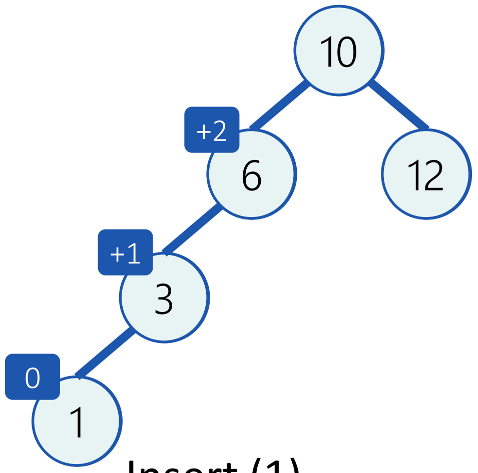
Rotations

What can we do if an insertion created "AVL criminals"?

Rotation: change of a few pointers in order to balance the height difference.

Here are some simple examples:





Insert (1)

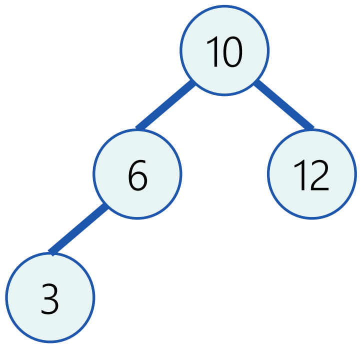
Fixing After Insertion

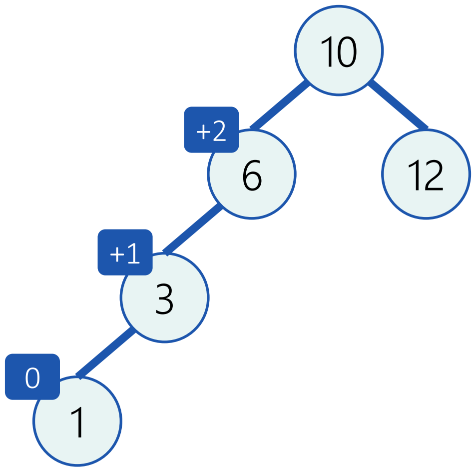
Rotations

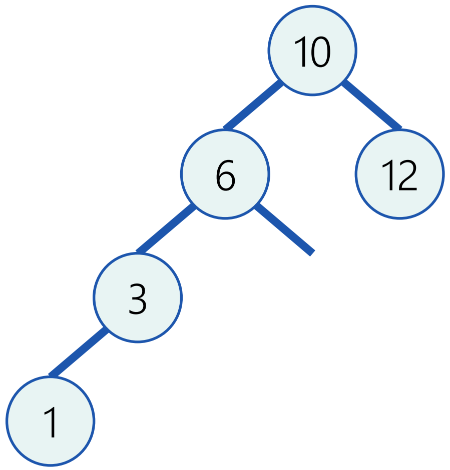
What can we do if an insertion created “AVL criminals”?

Rotation: change of a few pointers in order to balance the height difference.

Here are some simple examples:







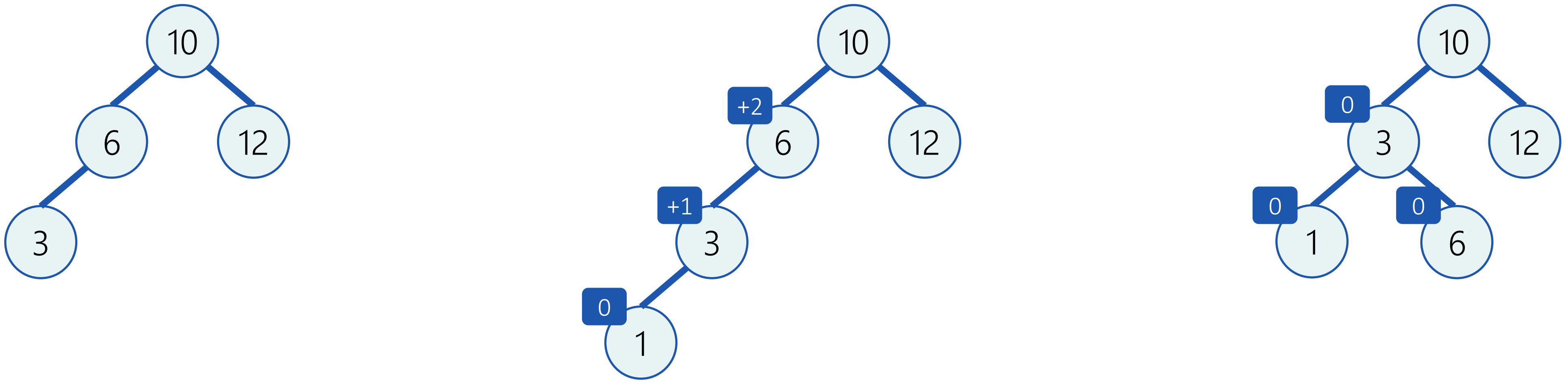
Fixing After Insertion

Rotations

What can we do if an insertion created "AVL criminals"?

Rotation: change of a few pointers in order to balance the height difference.

Here are some simple examples:



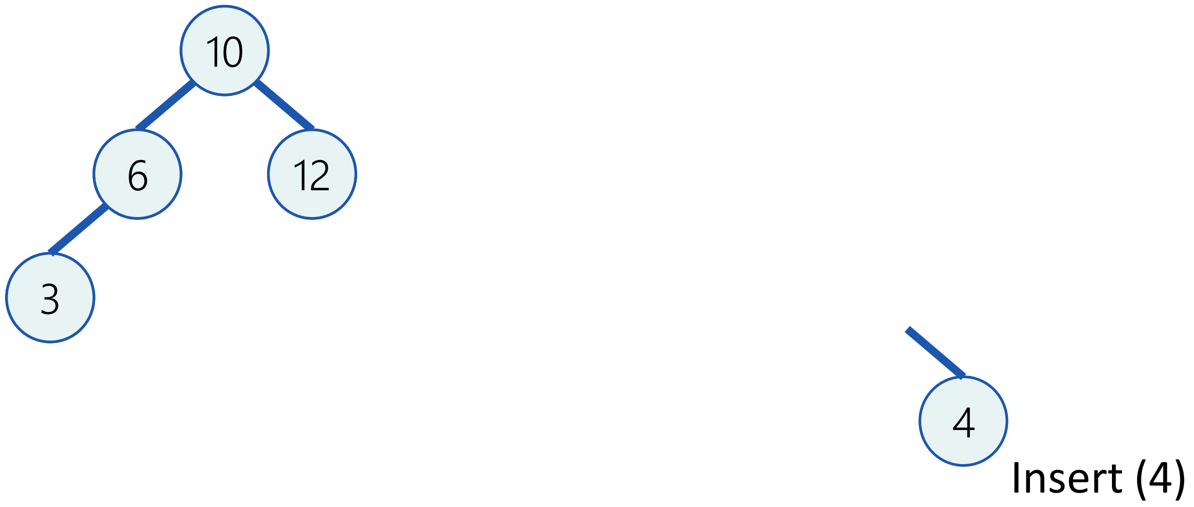
Fixing After Insertion

Rotations

What can we do if an insertion created "AVL criminals"?

Rotation: change of a few pointers in order to balance the height difference.

Here are some simple examples:



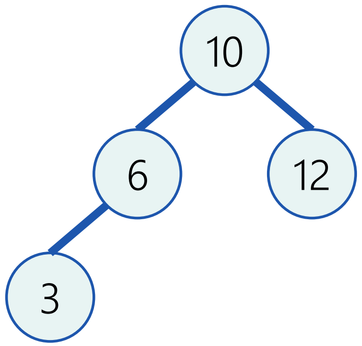
Fixing After Insertion

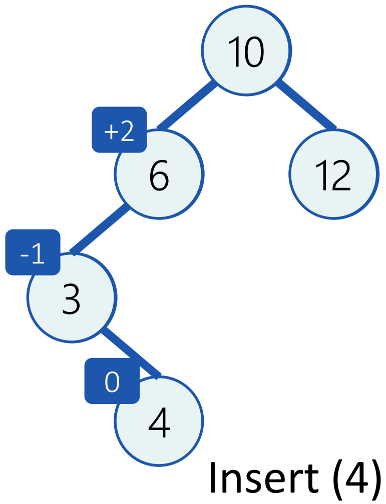
Rotations

What can we do if an insertion created "AVL criminals"?

Rotation: change of a few pointers in order to balance the height difference.

Here are some simple examples:





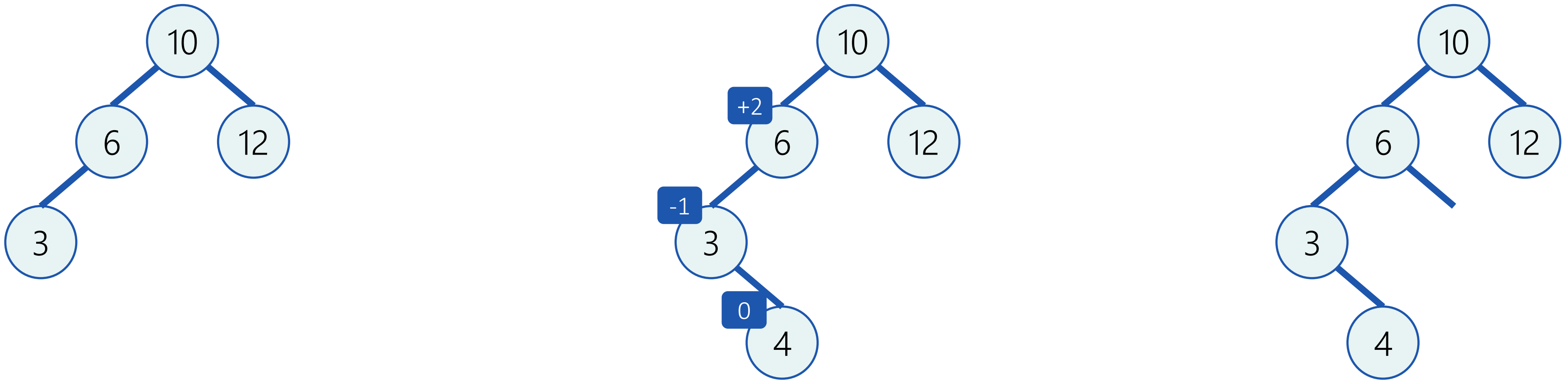
Fixing After Insertion

Rotations

What can we do if an insertion created "AVL criminals"?

Rotation: change of a few pointers in order to balance the height difference.

Here are some simple examples:



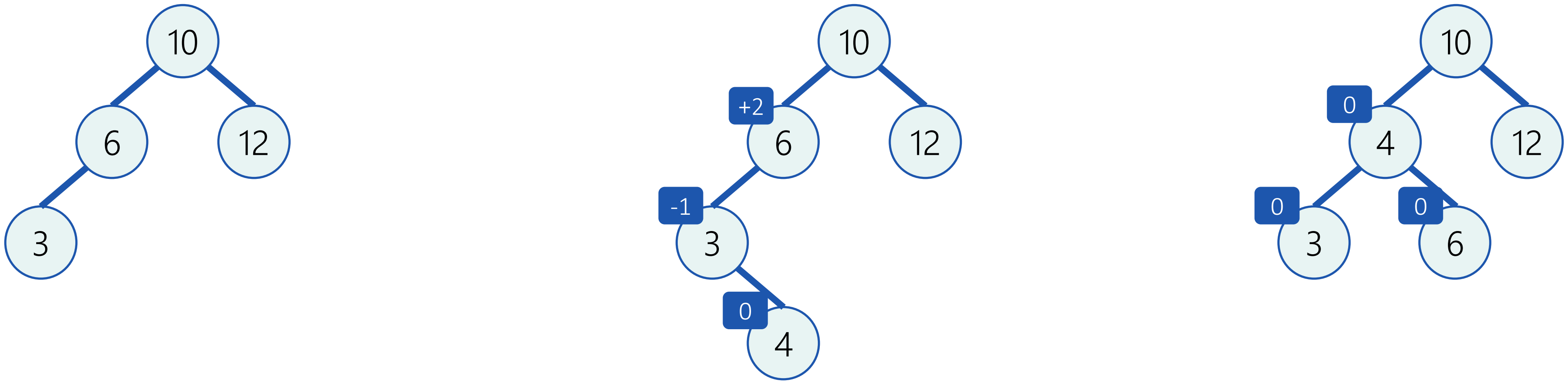
Fixing After Insertion

Rotations

What can we do if an insertion created “AVL criminals”?

Rotation: change of a few pointers in order to balance the height difference.

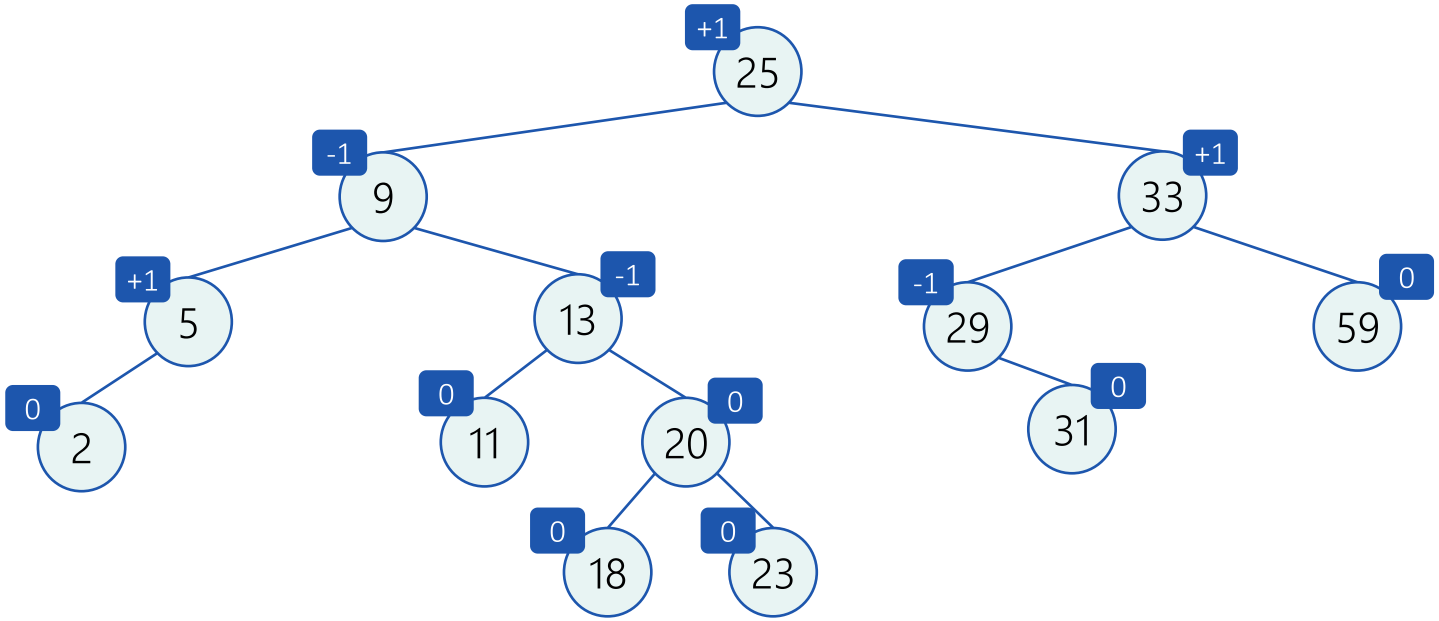
Here are some simple examples:



Fixing After Insertion

Reminder: $BF(v) = height(v.left) - height(v.right)$

In a valid AVL tree, each node v satisfies: $|BF(v)| \leq 1$

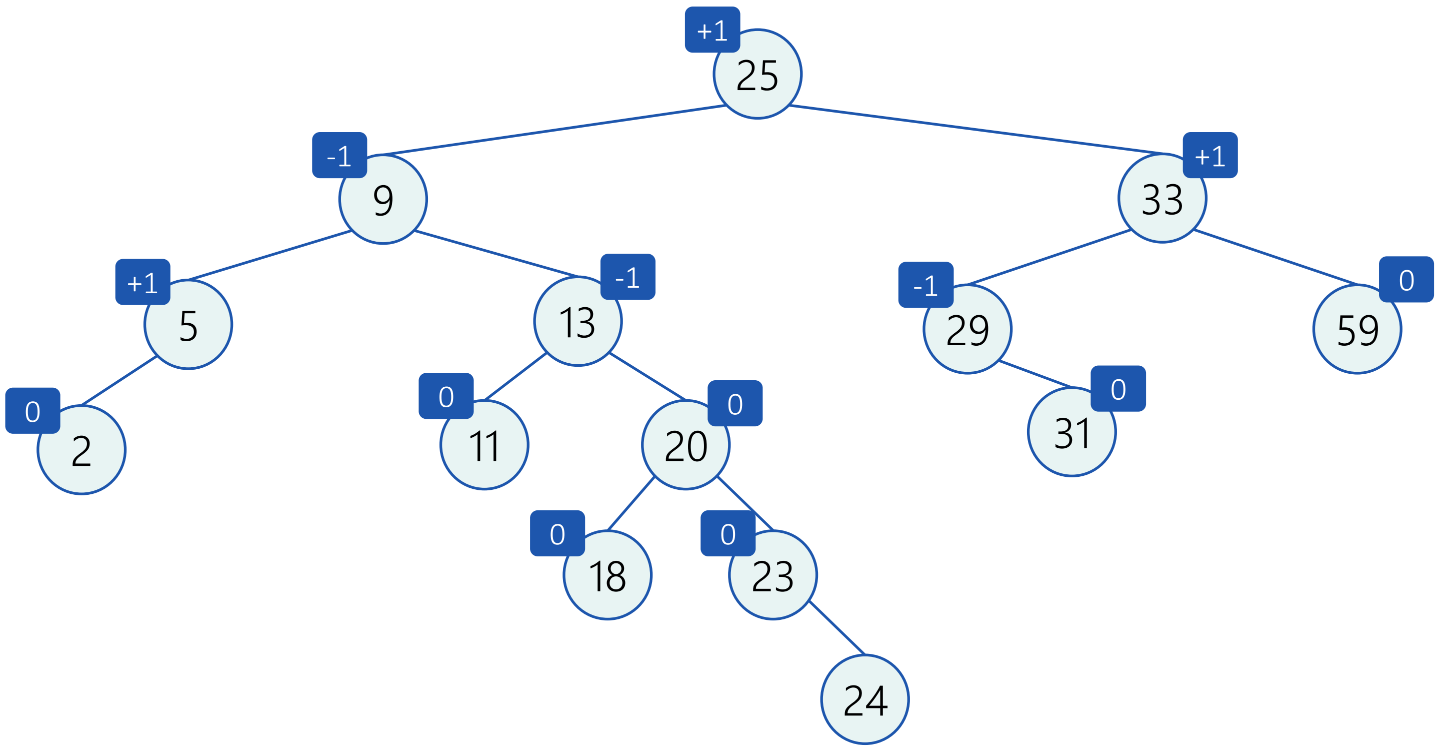


Fixing After Insertion

Observation 1

Assume we inserted 24 into the tree.

What are the possible BF values?

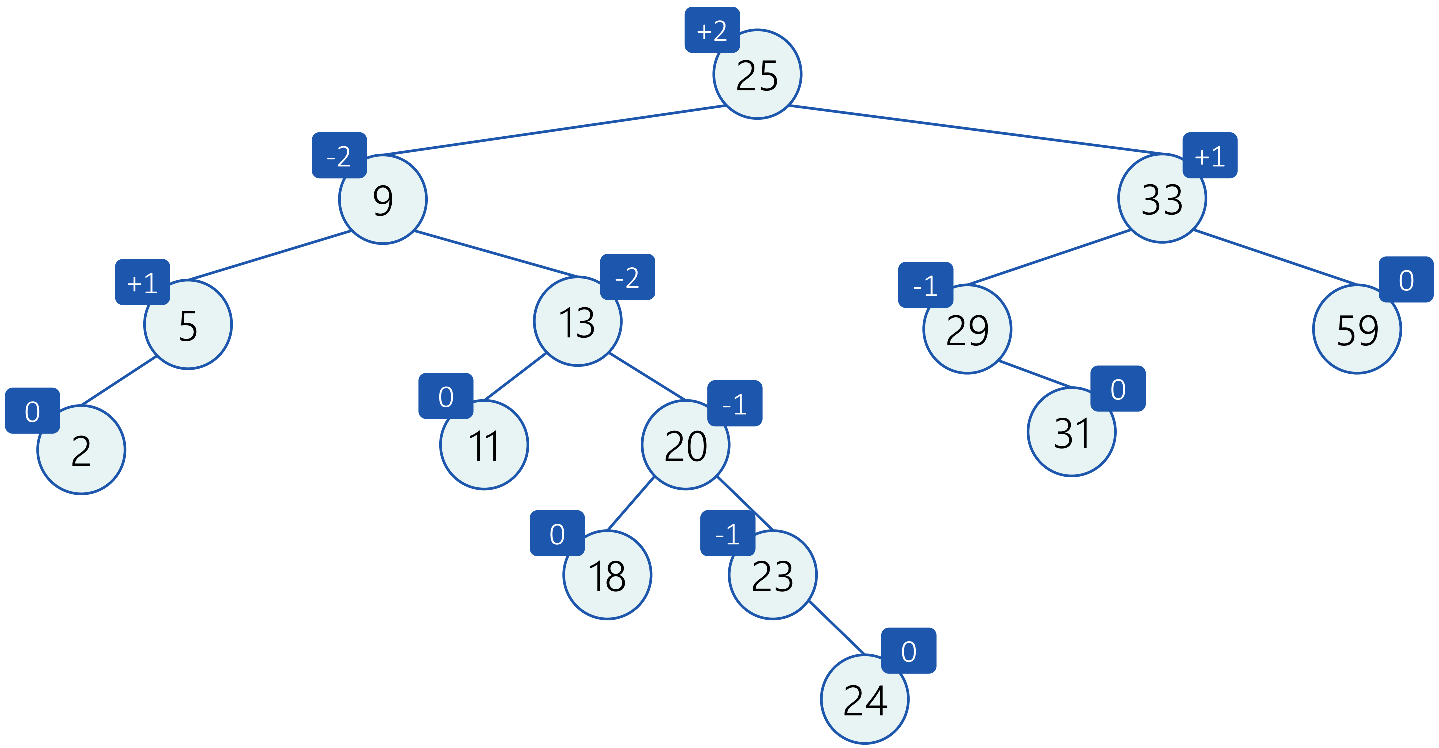


Fixing After Insertion

Observation 1

Assume we inserted 24 into the tree.

What are the possible BF values?



Fixing After Insertion

Observation 1

Assume we inserted 24 into the tree.

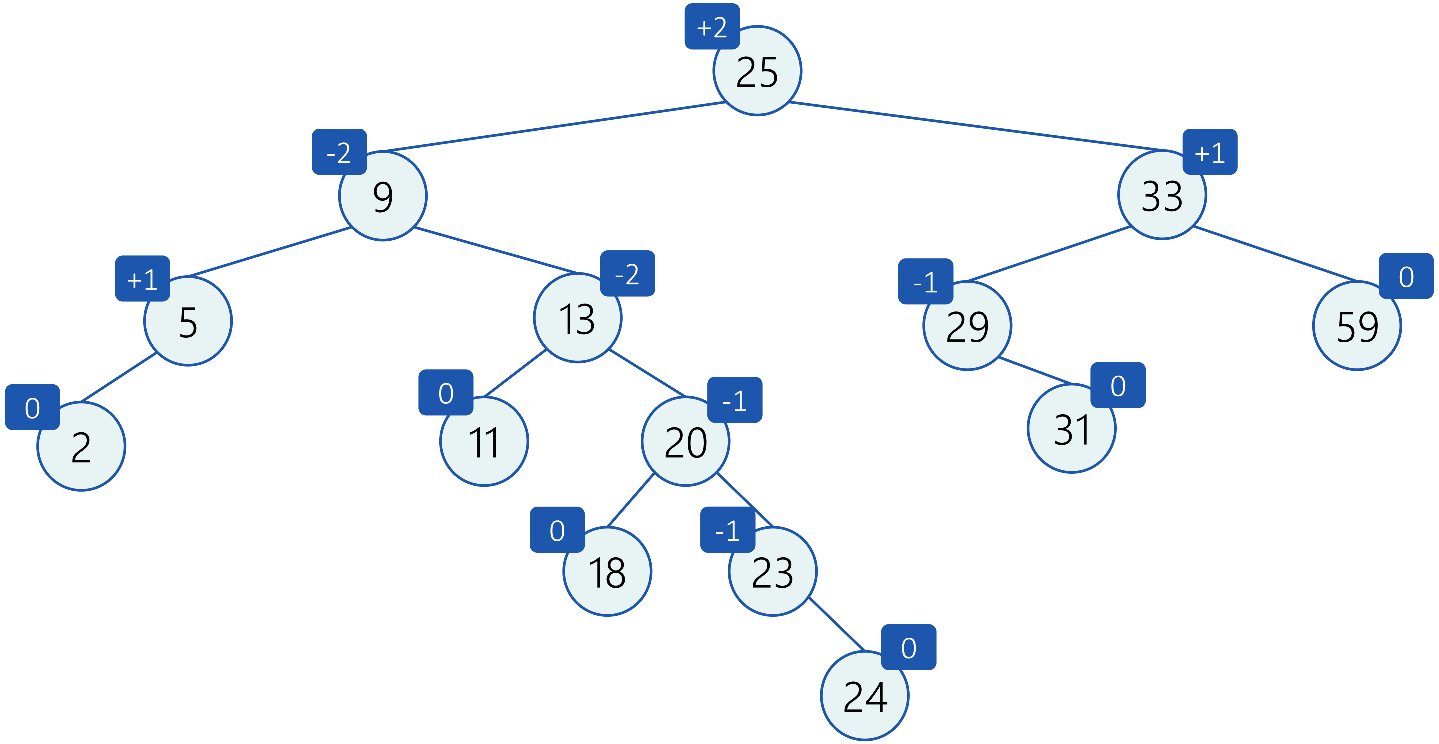
What are the possible BF values?

Observation 1:
After an insertion, the new balance factors are between -2 and +2, because they can change at most by ± 1.

Fixing After Insertion

Observation 2

Which nodes might change their BF?



Fixing After Insertion

Observation 2

Which nodes might change their BF?

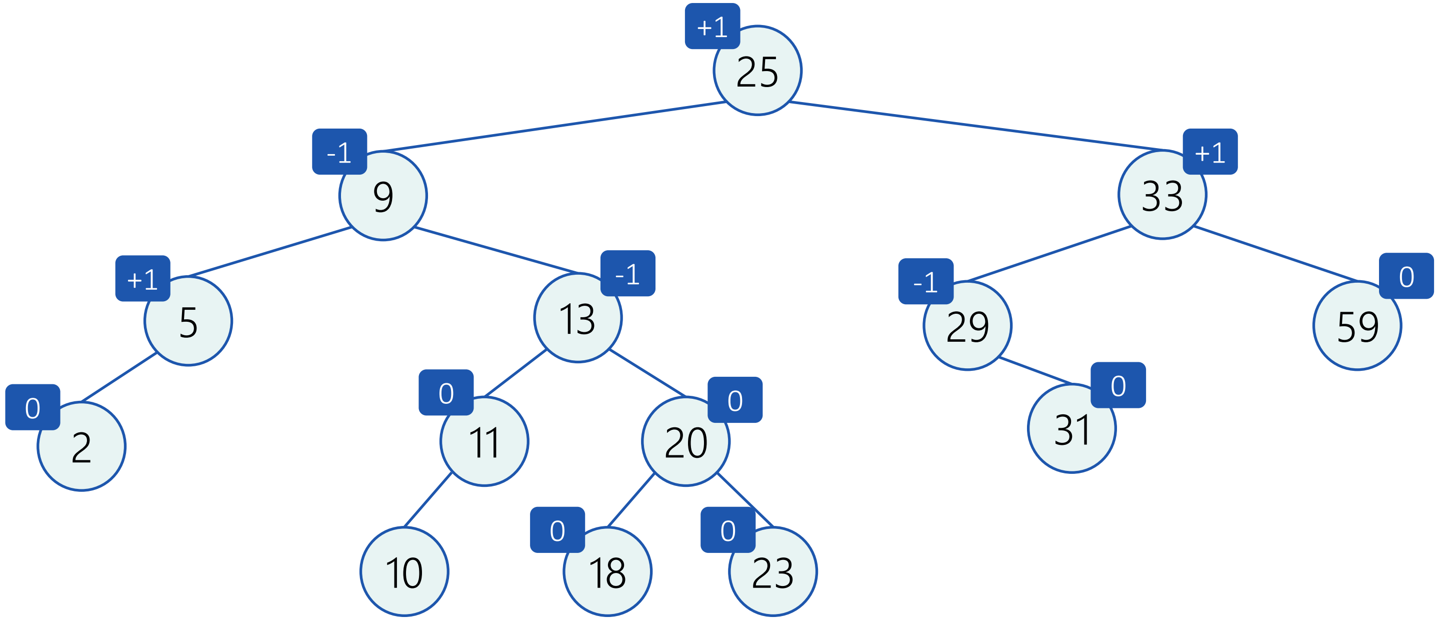
Observation 2:
The only nodes whose balance may have been changed are the nodes on the path from the root to the new node.

Fixing After Insertion

Observation 3

Do all nodes on the path from the root to the new node necessarily change their BF?

For example, inserting 10:

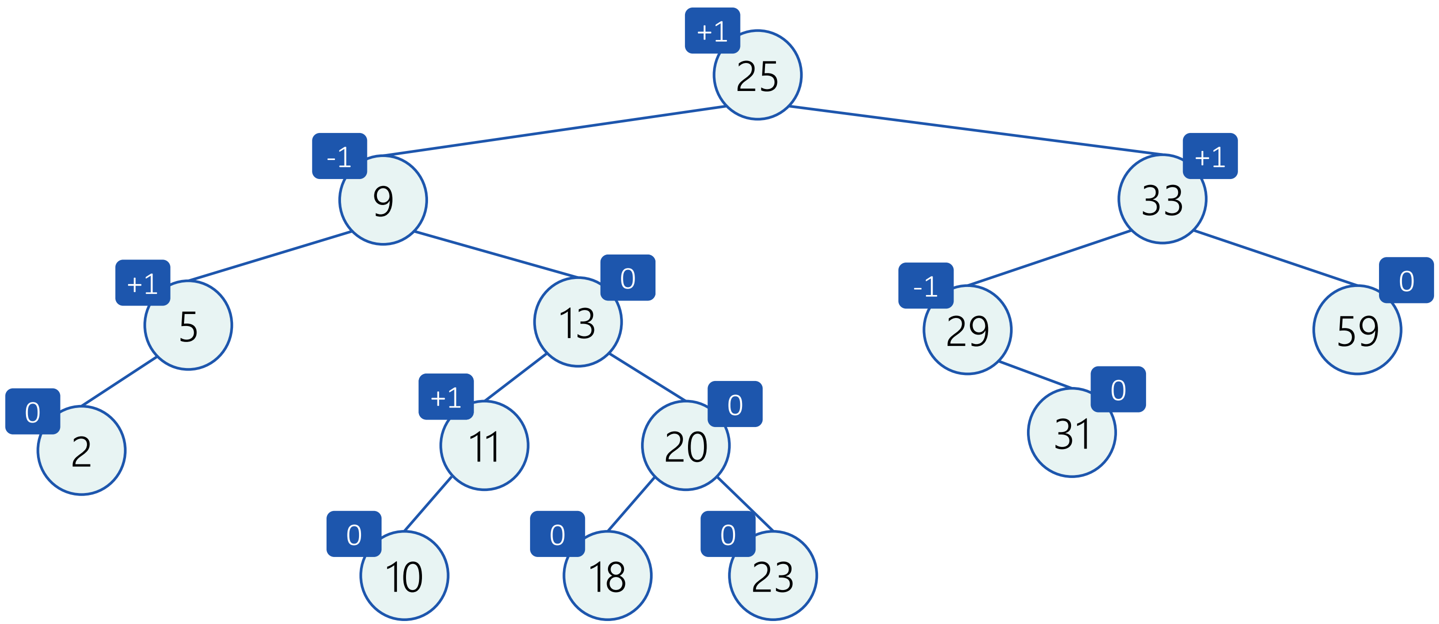


Fixing After Insertion

Observation 3

Do all nodes on the path from the root to the new node necessarily change their BF?

For example, inserting 10:



Fixing After Insertion

Observation 3

Do all nodes on the path from the root to the new node necessarily change their BF?

Observation 3:
If there’s a node on the above-mentioned path, whose **height hasn’t changed** after the insertion, then the **BFs** of the nodes **above** it haven’t changed.

The Insertion Algorithm

The Algorithm

AVL-Insert(T, z)

1. insert z as usual (as in a BST)
2. let y be the parent of the inserted node.
3. while $y \neq Null$ do:

   3.1. compute $BF(y)$*

   3.2. if $|BF(y)| < 2$ and y's height hasn't changed: terminate Obs. 3

   3.3. else if $|BF(y)| < 2$ and y's height changed: go back to stage 3 with y's parent

   3.4. else ($|BF(y)| = 2$): perform a rotation and go back to stage 3 with y's parent Obs. 2

Obs. 1

*Requires maintaining additional information at each node (its height). We will refer to this topic later.

The Insertion Algorithm

Time Complexity

AVL-Insert(T, z)

1.	Insertion into a BST	$O(h+1)$
2.	Go up to the root to spot "AVL criminals"	$O(h+1)$
3.	Rotations	$O(1) \times O(h+1)$
		$O(h+1) \ = \ O(logn)$

Rotations

Four Rotations to Rule Them All

Part A

Fixing After Insertion

Rotations

What is the "criminal" BF?

Fixing After Insertion

Rotations

What is the “criminal” BF?

-2	+2

Fixing After Insertion

Rotations

What is the “criminal” BF?

-2	+2
What is the BF of the **right** son?	What is the BF of the **left** son?

Fixing After Insertion

Rotations

What is the “criminal” BF?

-2		+2	
What is the BF of the **right** son?		What is the BF of the **left** son?	
-1	+1	-1	+1

Fixing After Insertion

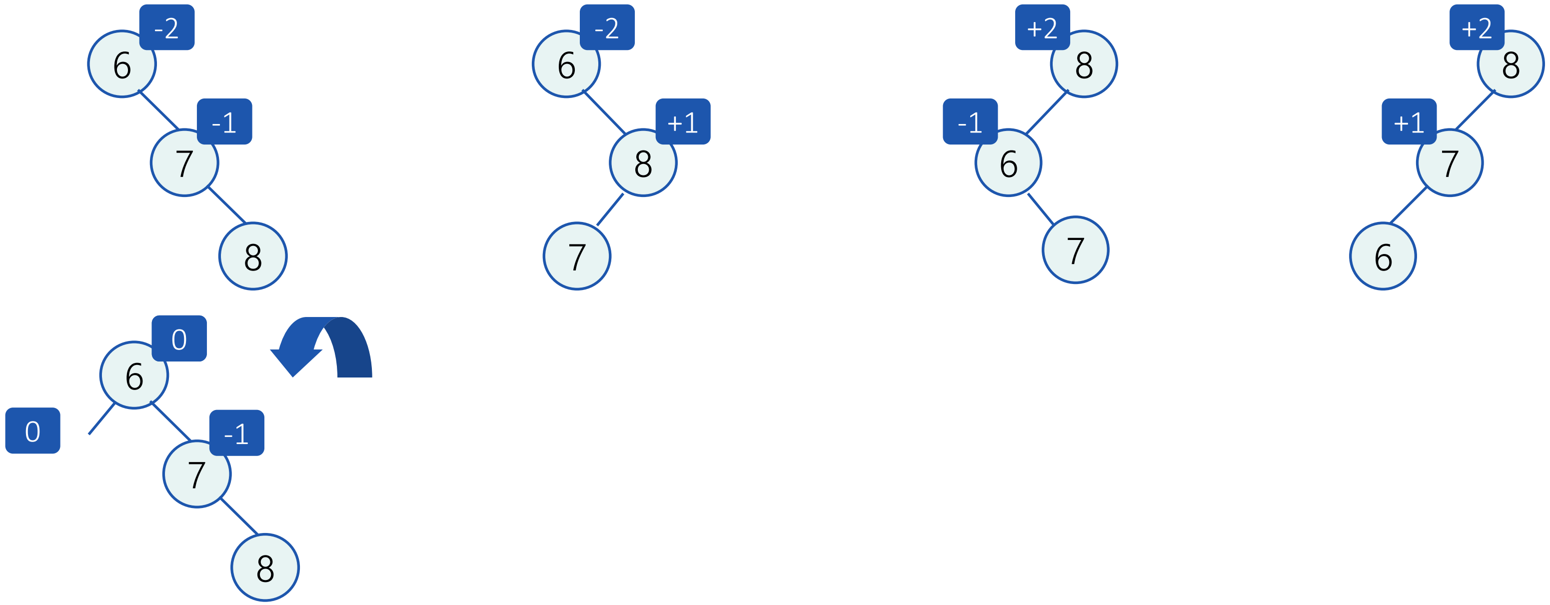
Rotations

What is the “criminal” BF?

-2		+2	
What is the BF of the right son?		**What is the BF of the left son?**	
-1 Left rotation	+1	-1	+1

Before rotation

After rotation



Fixing After Insertion

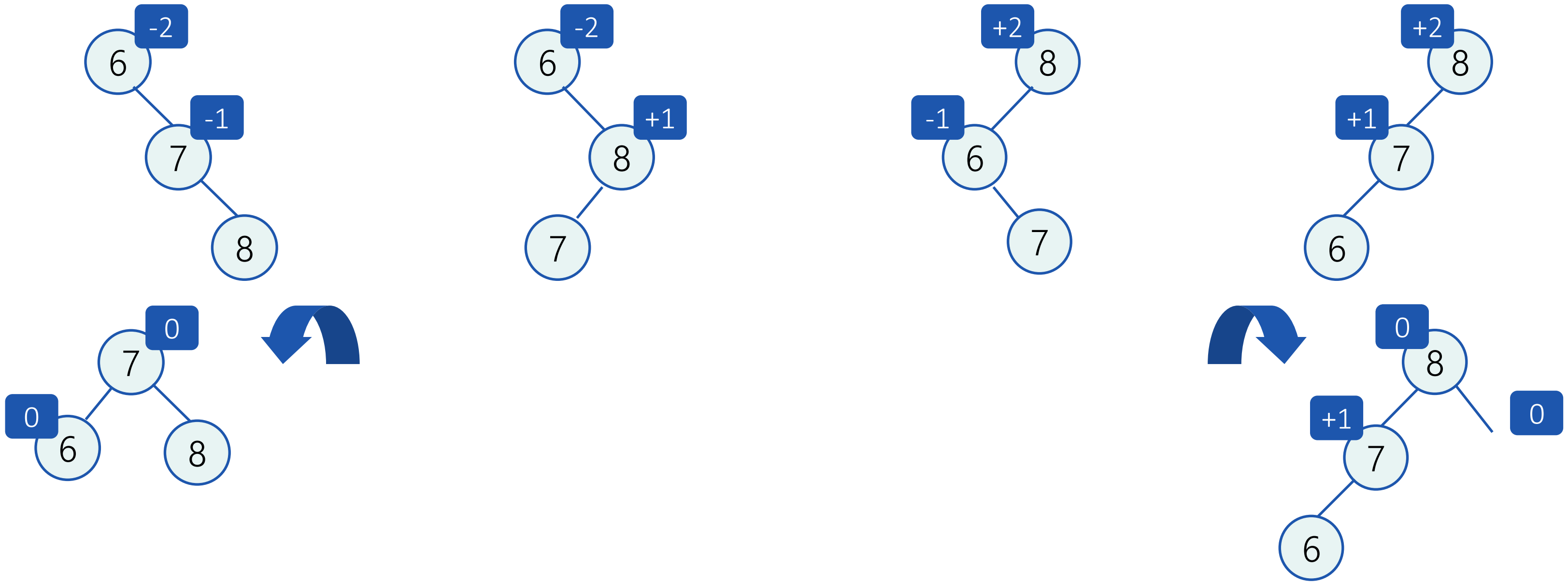
Rotations

What is the “criminal” BF?

-2		+2	
What is the BF of the **right** son?		What is the BF of the **left** son?	
-1 Left rotation	+1	-1	+1 Right rotation

Before rotation

After rotation



Fixing After Insertion

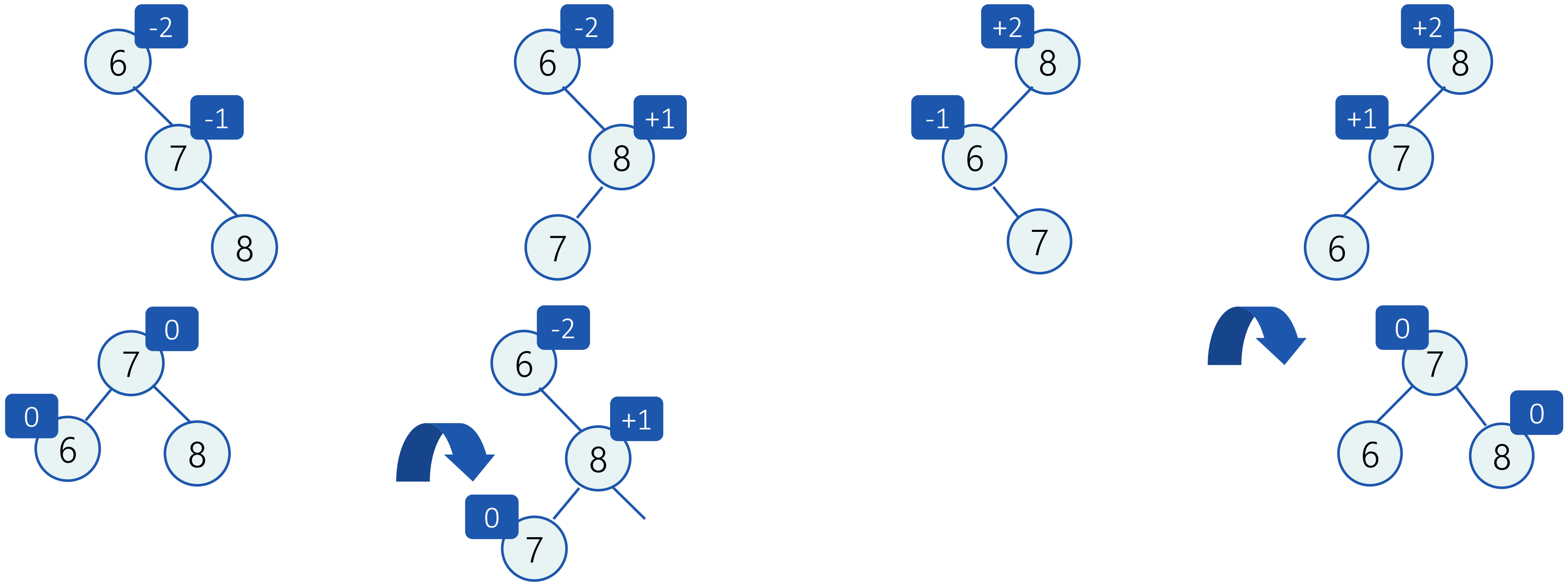
Rotations

What is the "criminal" BF?

-2		+2	
What is the BF of the **right** son?		What is the BF of the **left** son?	
-1 Left rotation	+1 Right then left rotation	-1	+1 Right rotation

Before rotation

After rotation



Fixing After Insertion

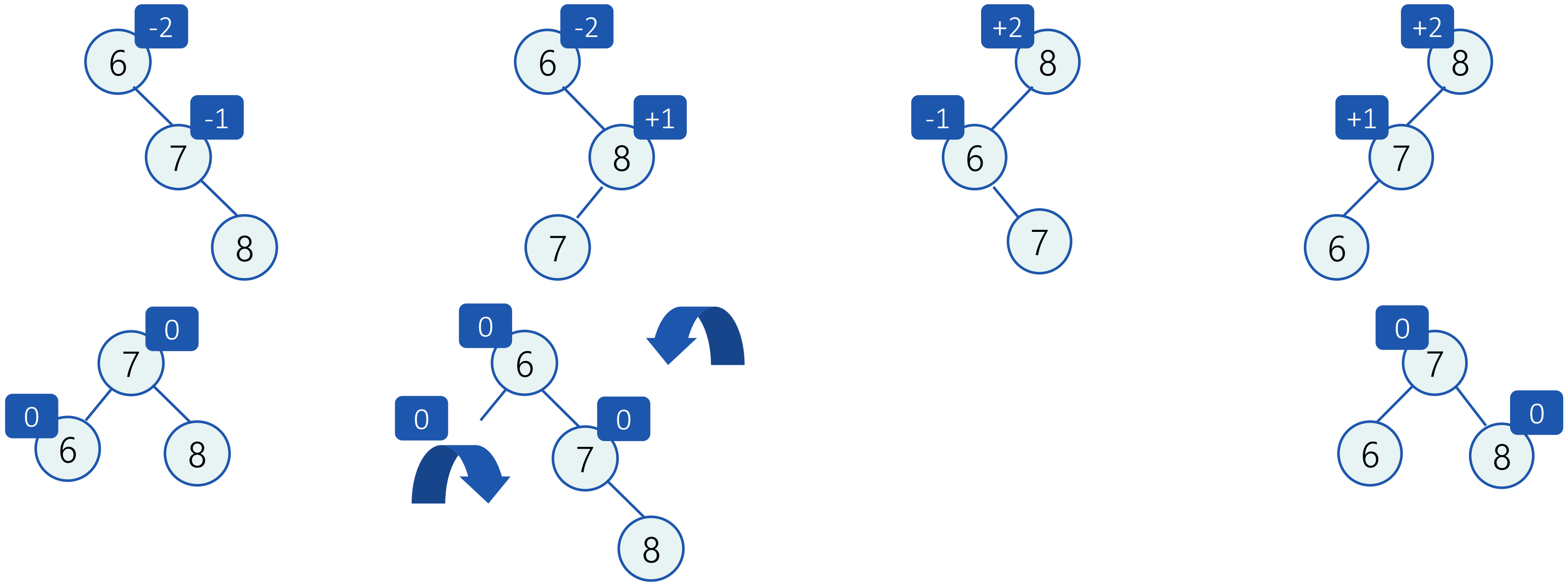
Rotations

What is the “criminal” BF?

-2		+2	
What is the BF of the **right** son?		What is the BF of the **left** son?	
-1 Left rotation	+1 Right then left rotation	-1	+1 Right rotation

Before rotation

After rotation



Fixing After Insertion

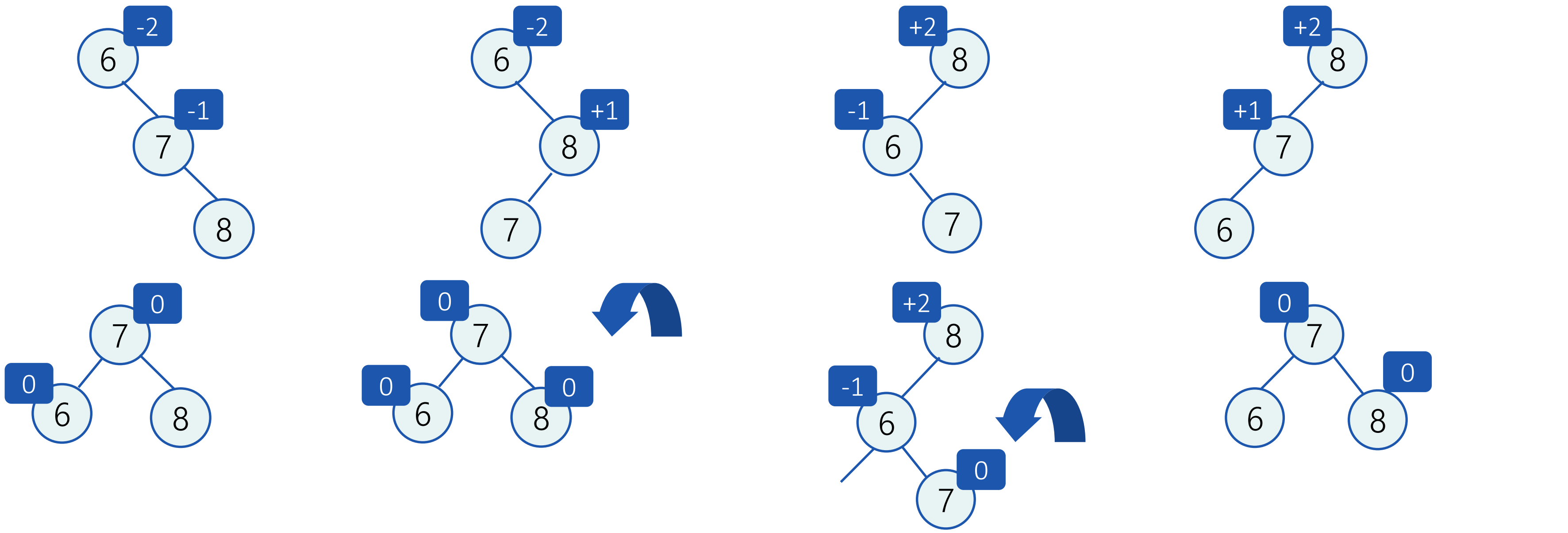
Rotations

What is the “criminal” BF?

-2		+2	
What is the BF of the **right** son?		What is the BF of the **left** son?	
-1 Left rotation	+1 Right then left rotation	-1 Left then right rotation	+1 Right rotation

Before rotation

After rotation



Fixing After Insertion

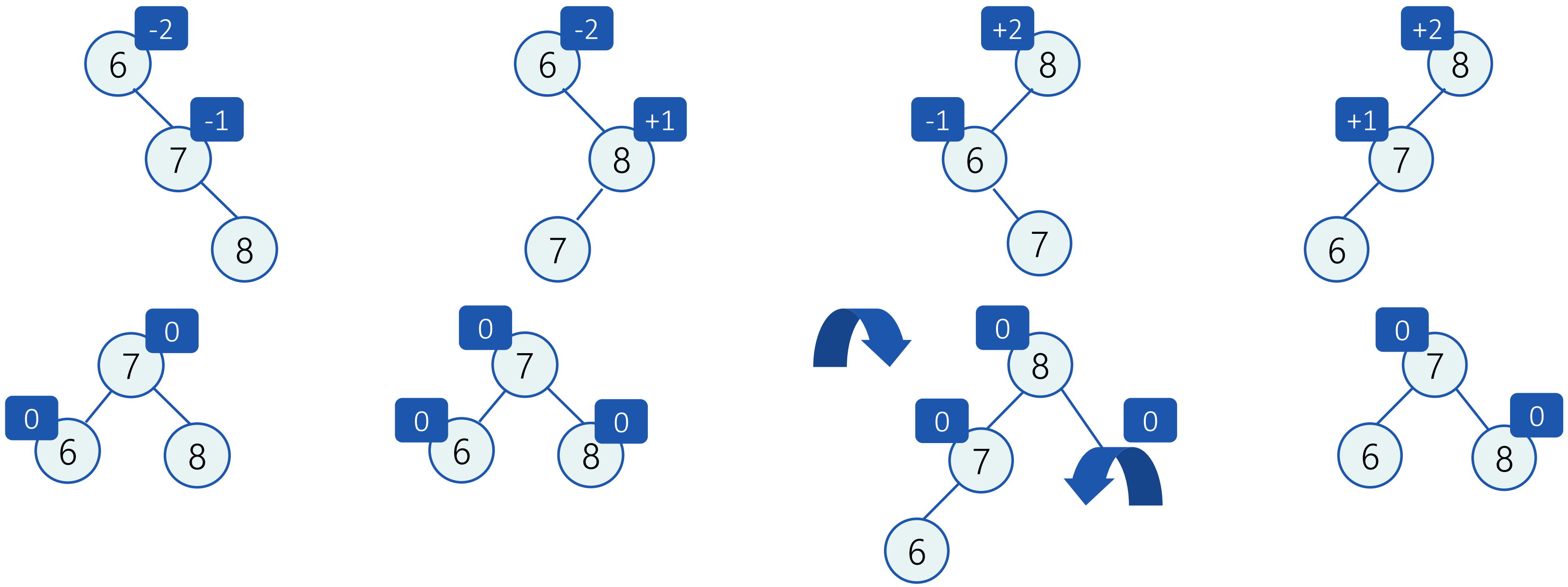
Rotations

What is the “criminal” BF?

-2		+2	
What is the BF of the **right** son?		What is the BF of the **left** son?	
-1 Left rotation	+1 Right then left rotation	-1 Left then right rotation	+1 Right rotation

Before rotation

After rotation



Rotations

Four Rotations to Rule Them All

Part B

Fixing After Insertion

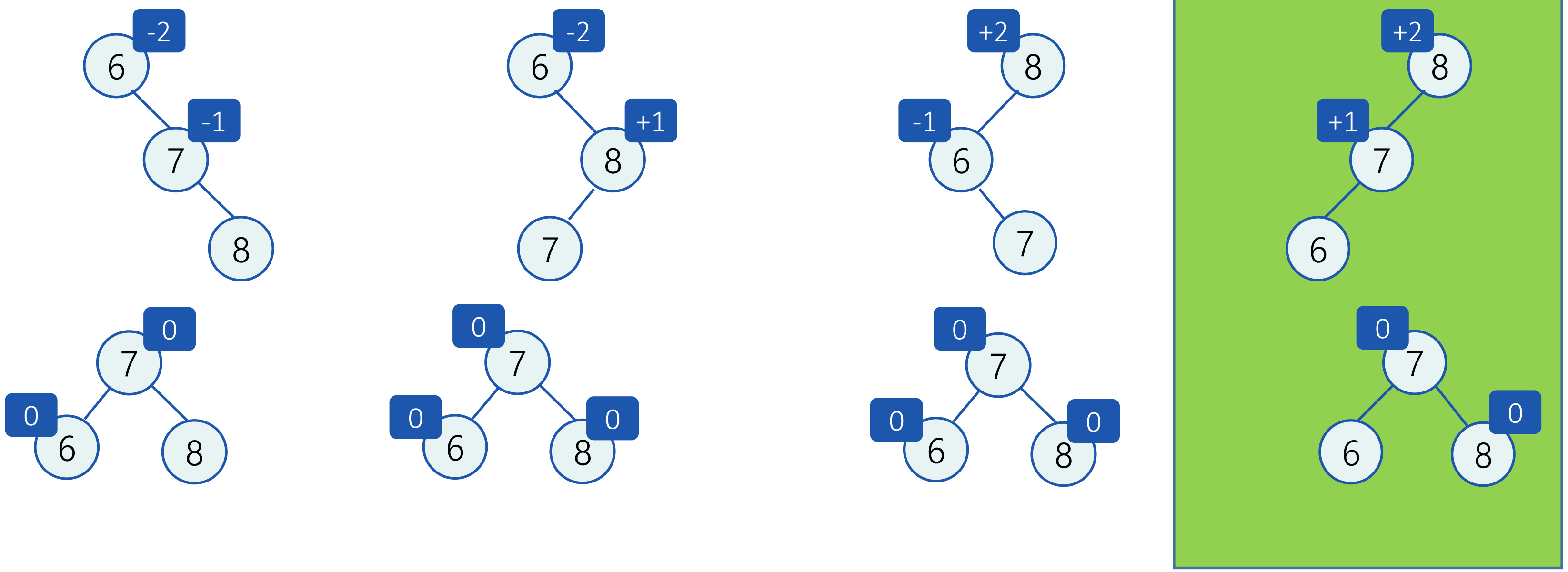
Rotations

What is the “criminal” BF?

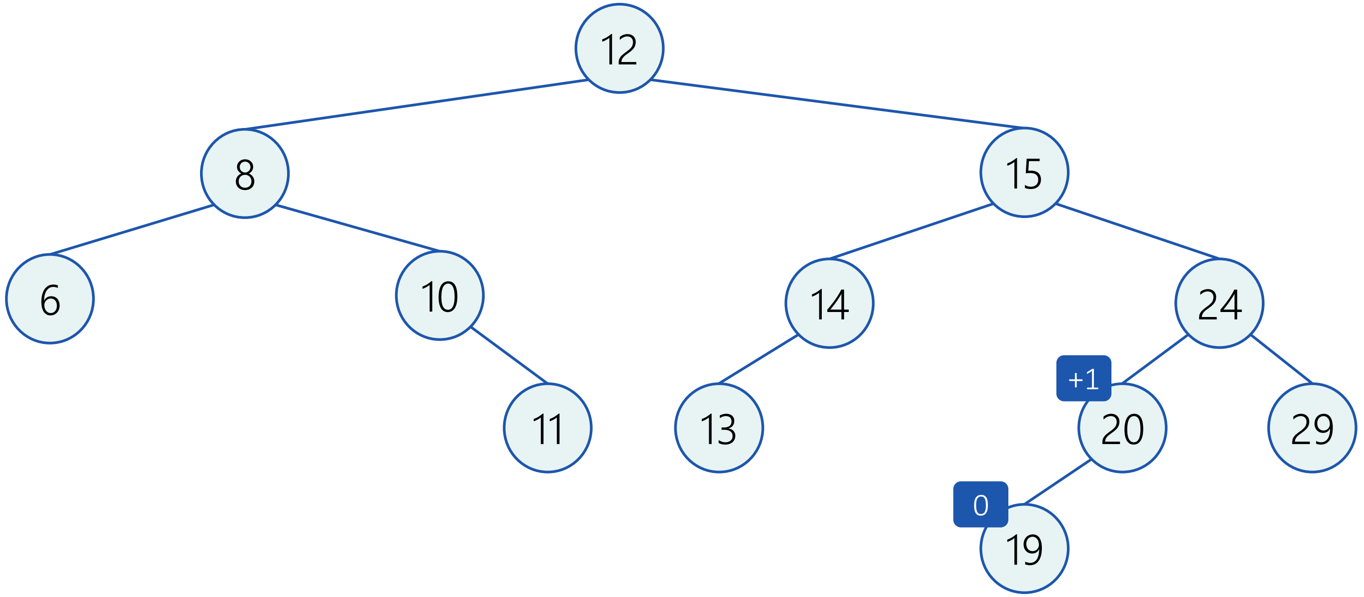
-2		+2	
What is the BF of the **right** son?		What is the BF of the **left** son?	
-1 Left rotation	+1 Right then left rotation	-1 Left then right rotation	+1 Right rotation

Before rotation

After rotation



Right Rotation
Example



Before insertion

Insert 18

Temporarily after insertion

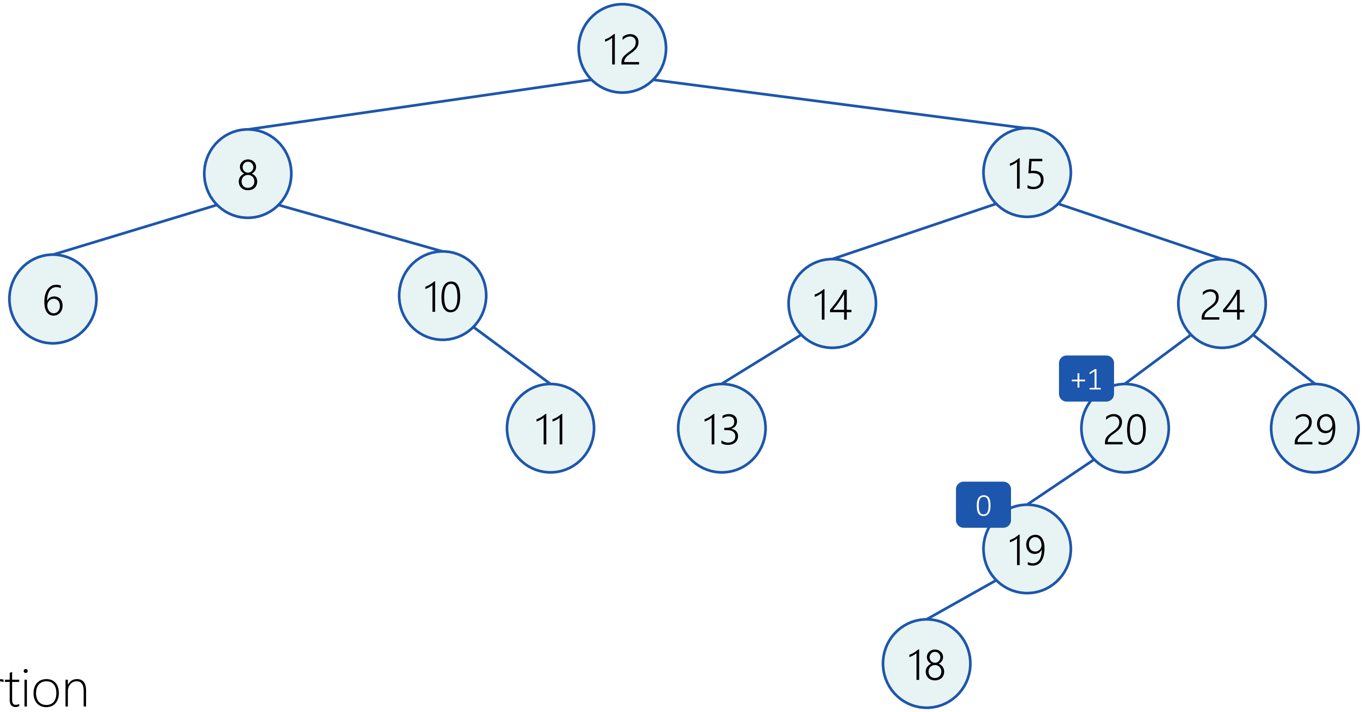
Rotate Right

After Rotation

Interactive (https://visualgo.net)

Right Rotation

Example



Before insertion

Insert 18

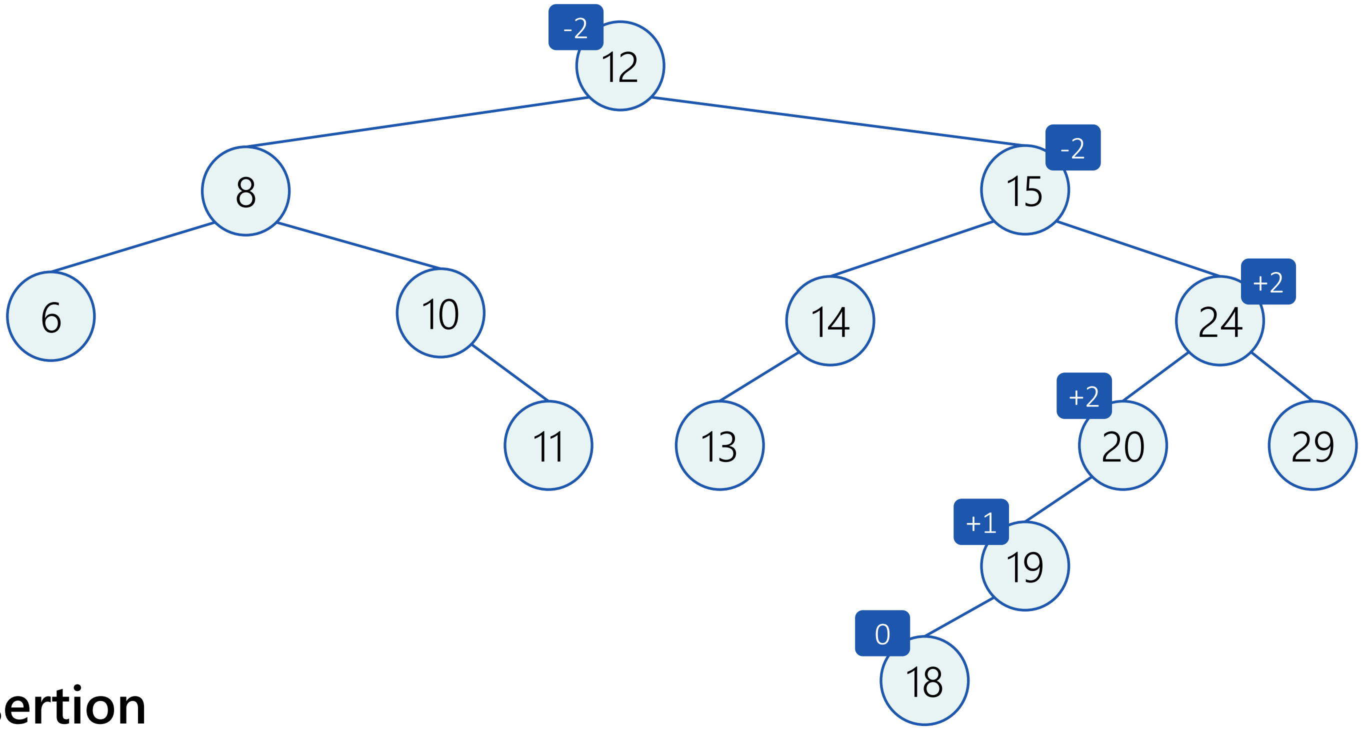
Temporarily after insertion

Rotate Right

After Rotation

Right Rotation

Example



Before insertion

Insert 18

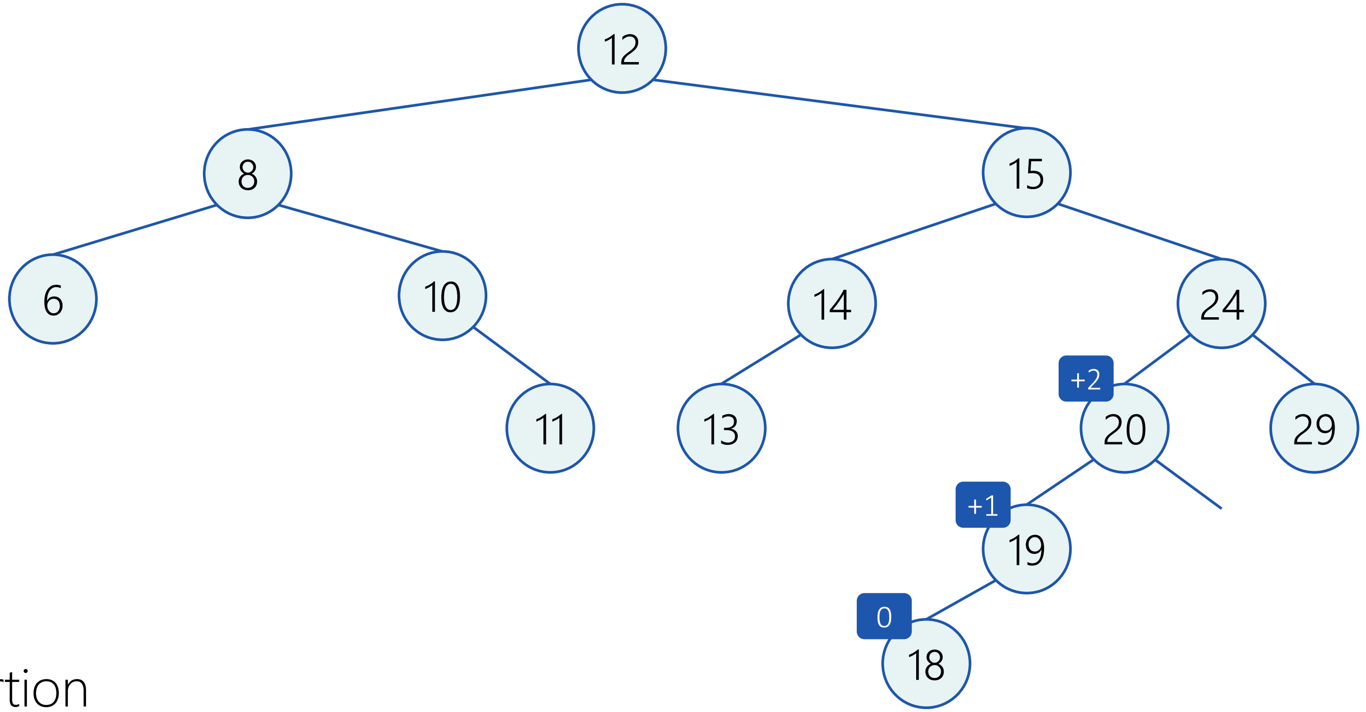
Temporarily after insertion

Rotate Right

After Rotation

Right Rotation

Example



Before insertion

Insert 18

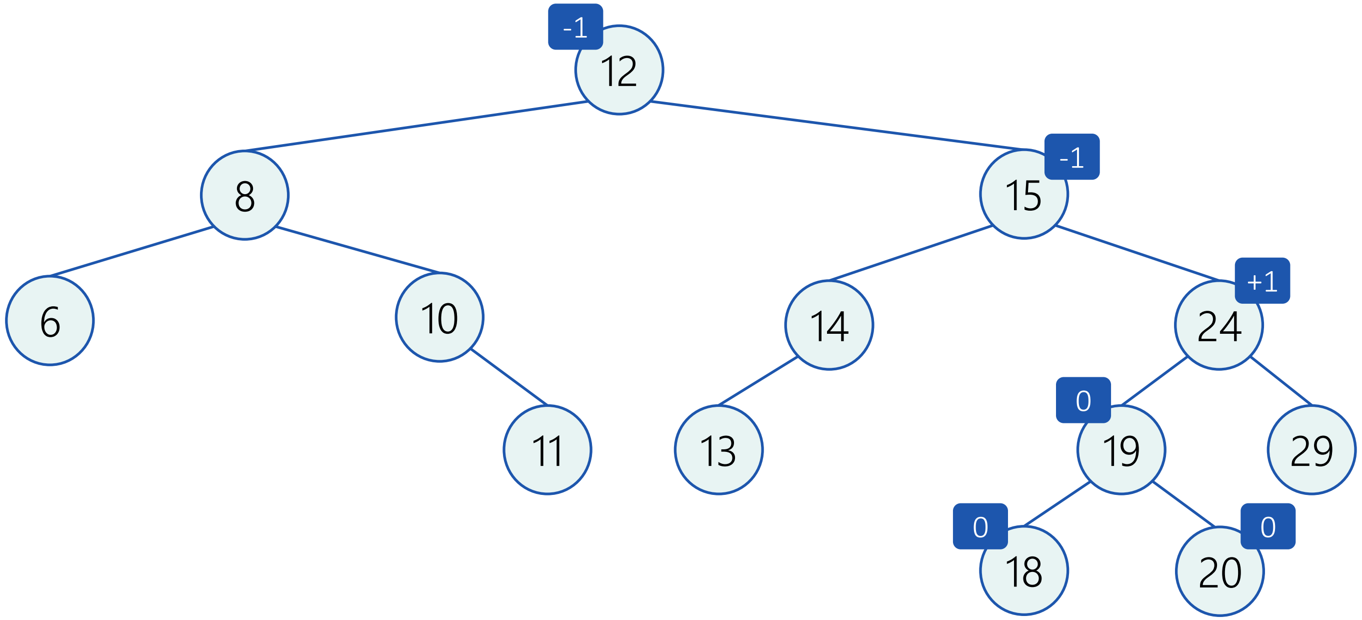
Temporarily after insertion

Rotate Right

After Rotation

Right Rotation

Example



Before insertion

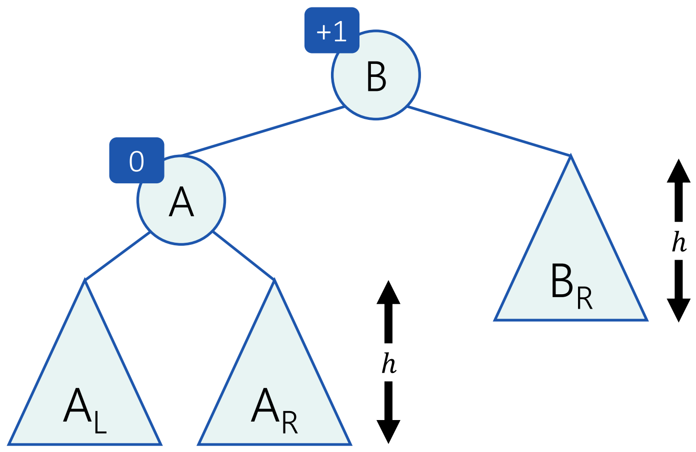
Insert 18

Temporarily after insertion

Rotate Right

After Rotation

Right Rotation



Before insertion

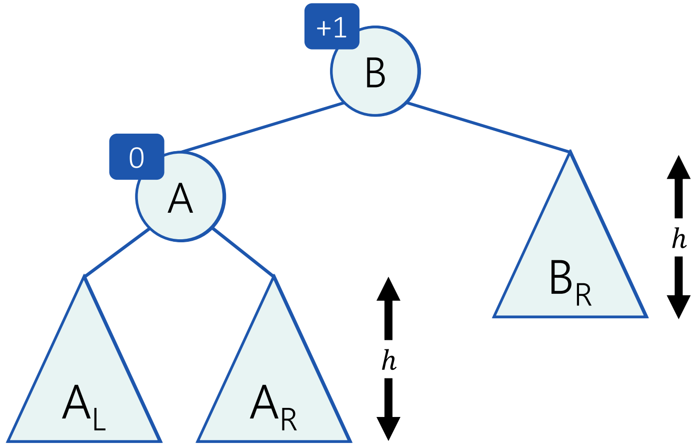
Insert v

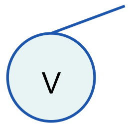
Temporarily after insertion

Rotate right

After rotation

Right Rotation





Before insertion

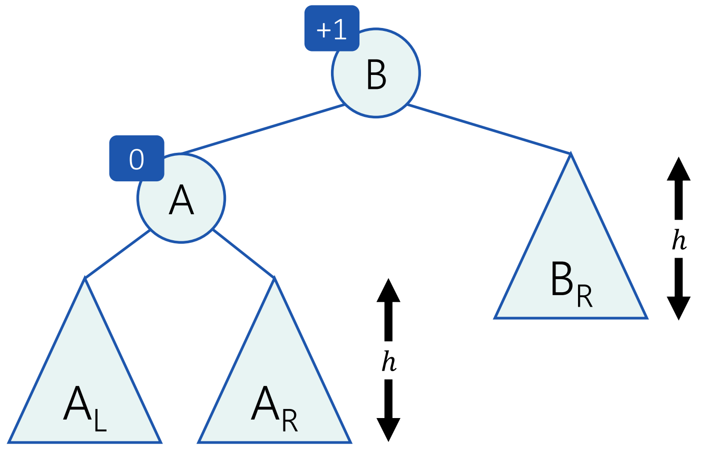
Insert v

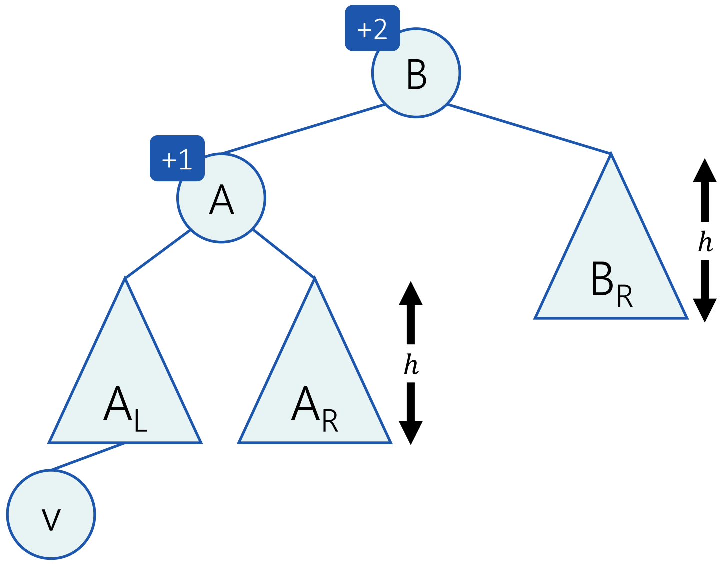
Temporarily after insertion

Rotate right

After rotation

Right Rotation





Before insertion

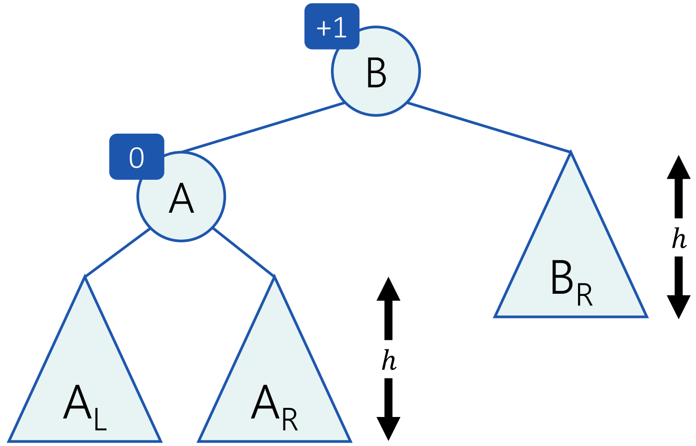
Insert v

Temporarily after insertion

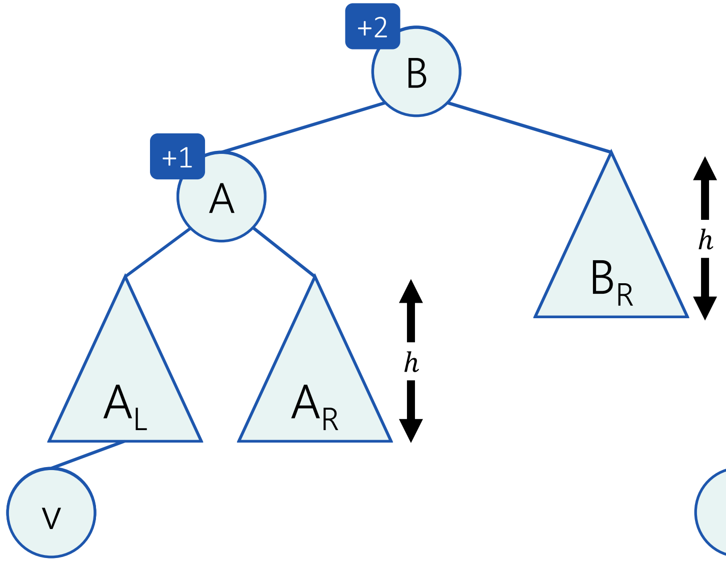
Rotate right

After rotation

Right Rotation

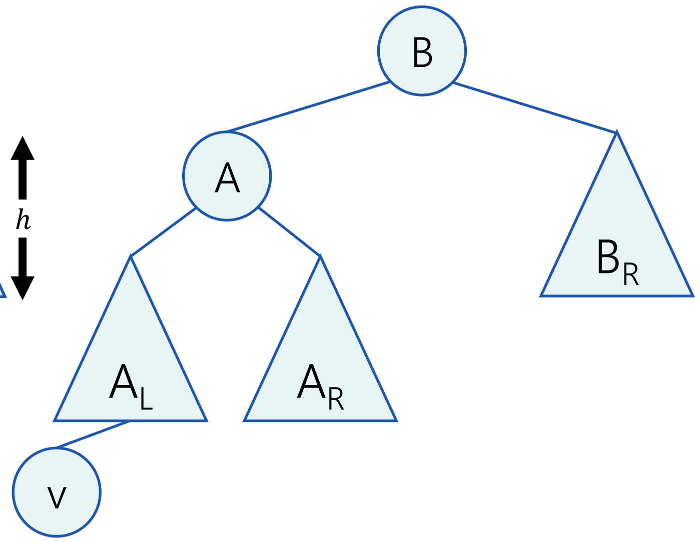


Before insertion



Insert v

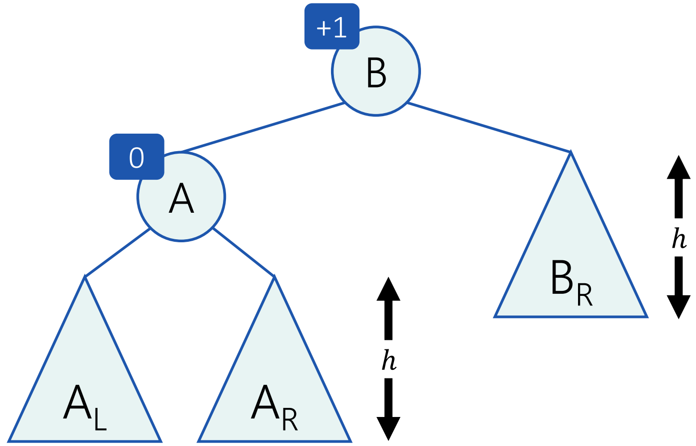
Temporarily after insertion



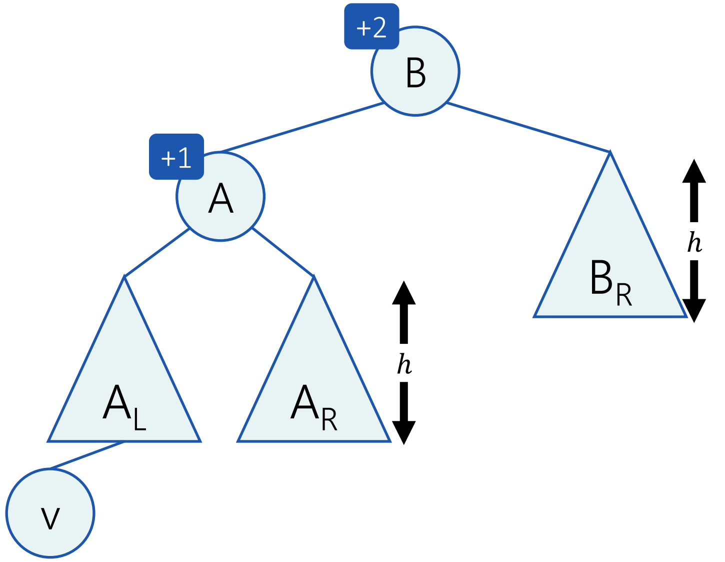
Rotate right (A-B)

After rotation

Right Rotation

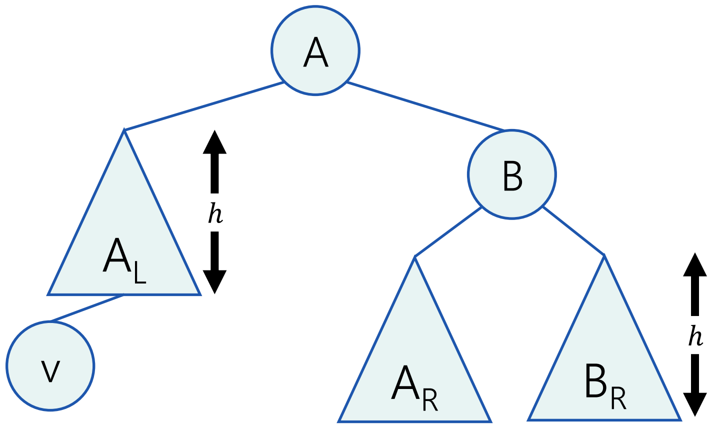


Before insertion



Insert v

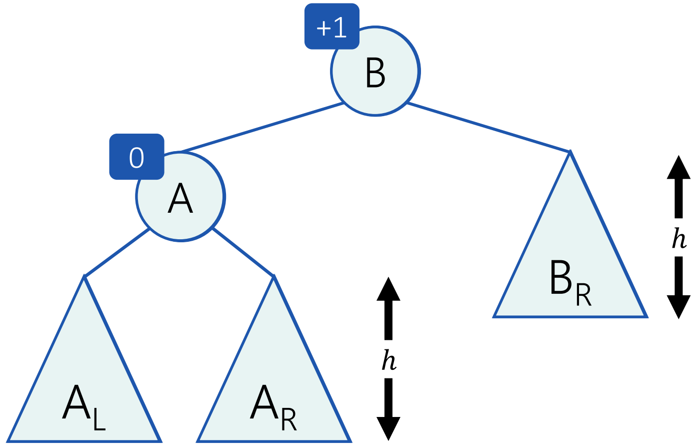
Temporarily after insertion



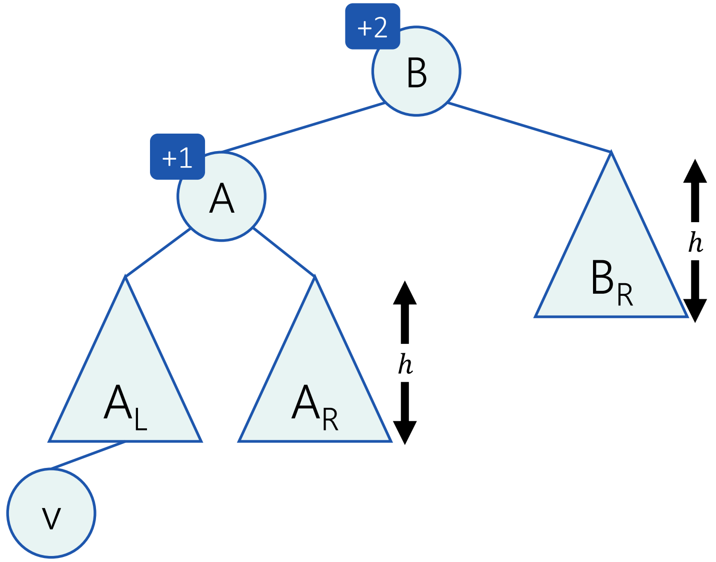
Rotate right (A-B)

After rotation

Right Rotation

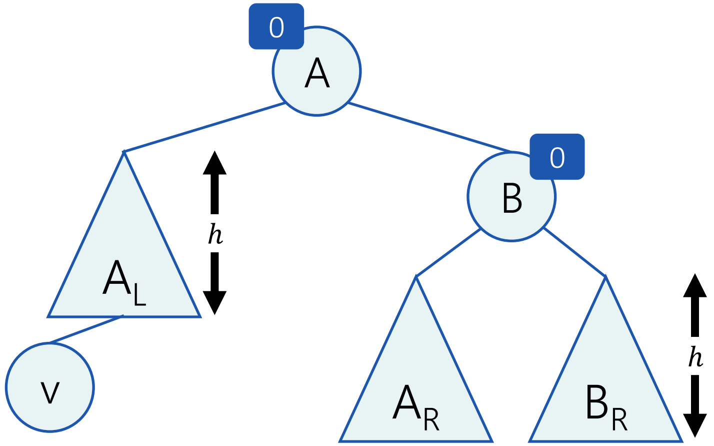


Before insertion



Insert v

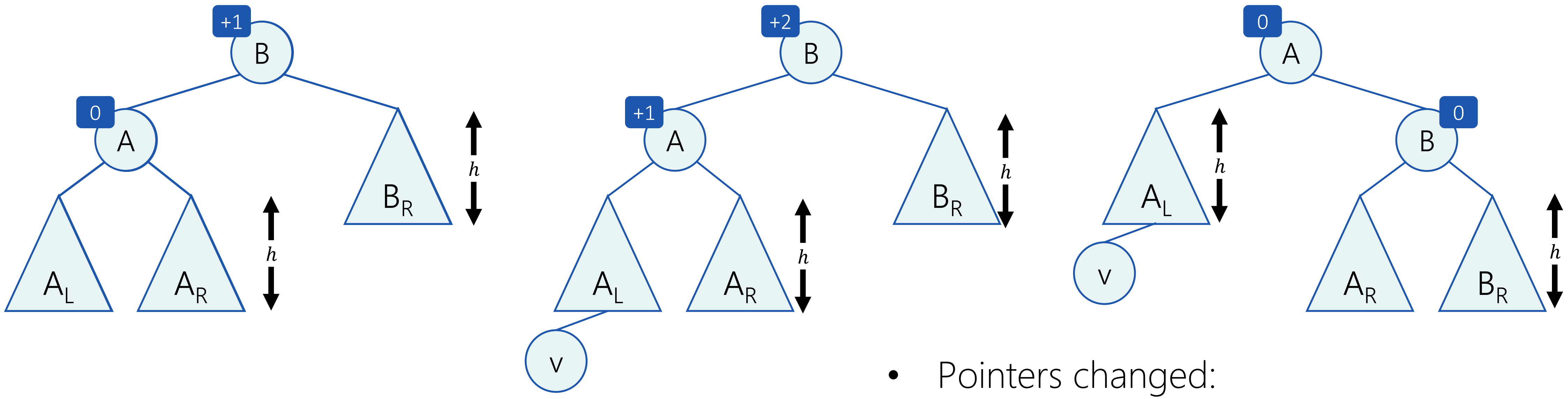
Temporarily after insertion



Rotate right (A-B)

After rotation

Right Rotation



- The search tree property is preserved
- The violation of balance was fixed in this subtree
- Left rotation is symmetric

- Pointers changed:
  - $B.left \leftarrow A.right$
  - $B.left.parent \leftarrow B$
  - $A.right \leftarrow B$
  - $A.parent \leftarrow B.parent$
  - $A.parent.left/right \leftarrow A$
  - $B.parent \leftarrow A$

Fixing After Insertion

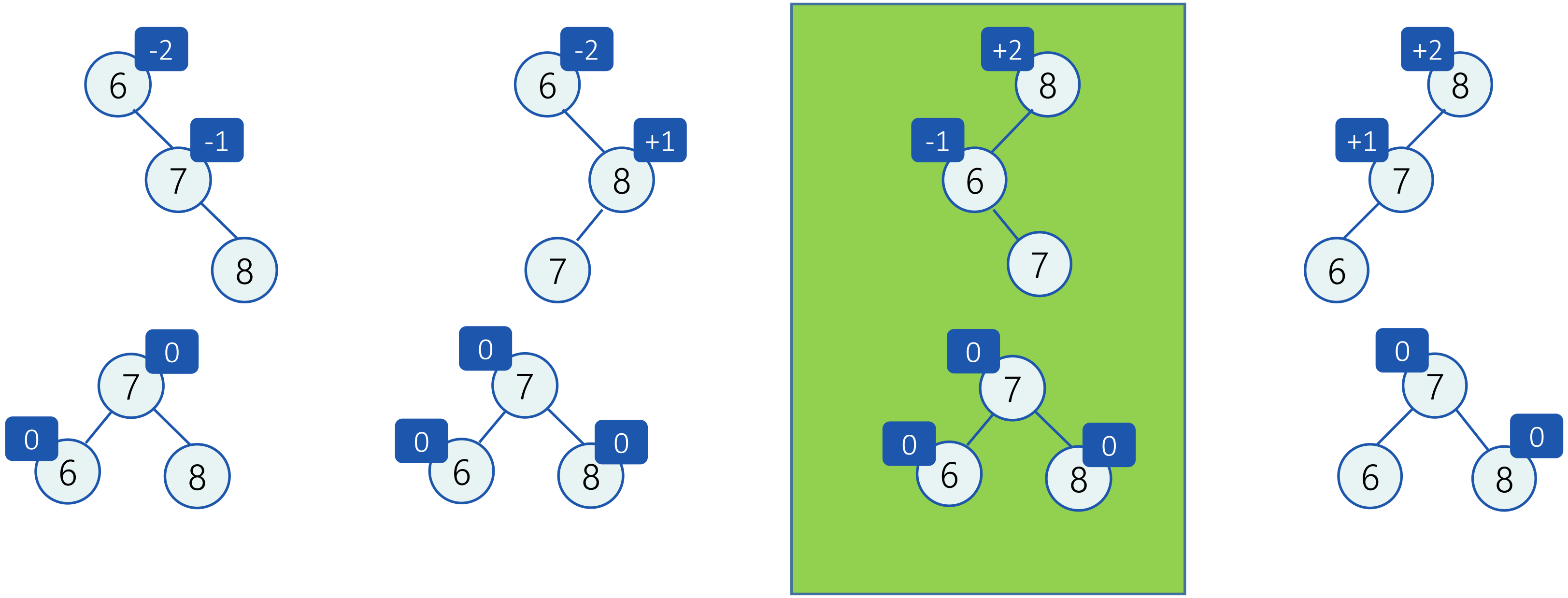
Rotations

What is the “criminal” BF?

-2		+2	
What is the BF of the **right** son?		What is the BF of the **left** son?	
-1 Left rotation	+1 Right then left rotation	-1 Left then right rotation	+1 Right rotation

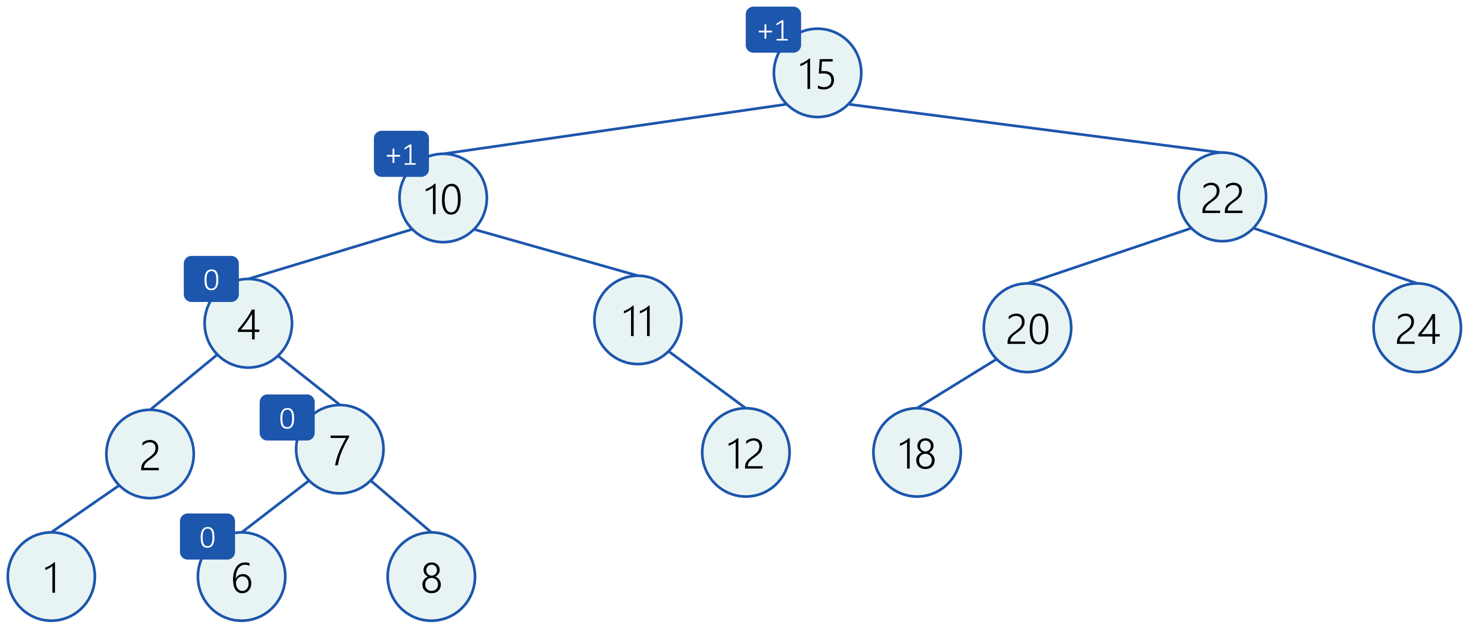
Before rotation

After rotation



Left Then Right Rotation

Example



Before insertion

Insert 5

Temporarily after insertion

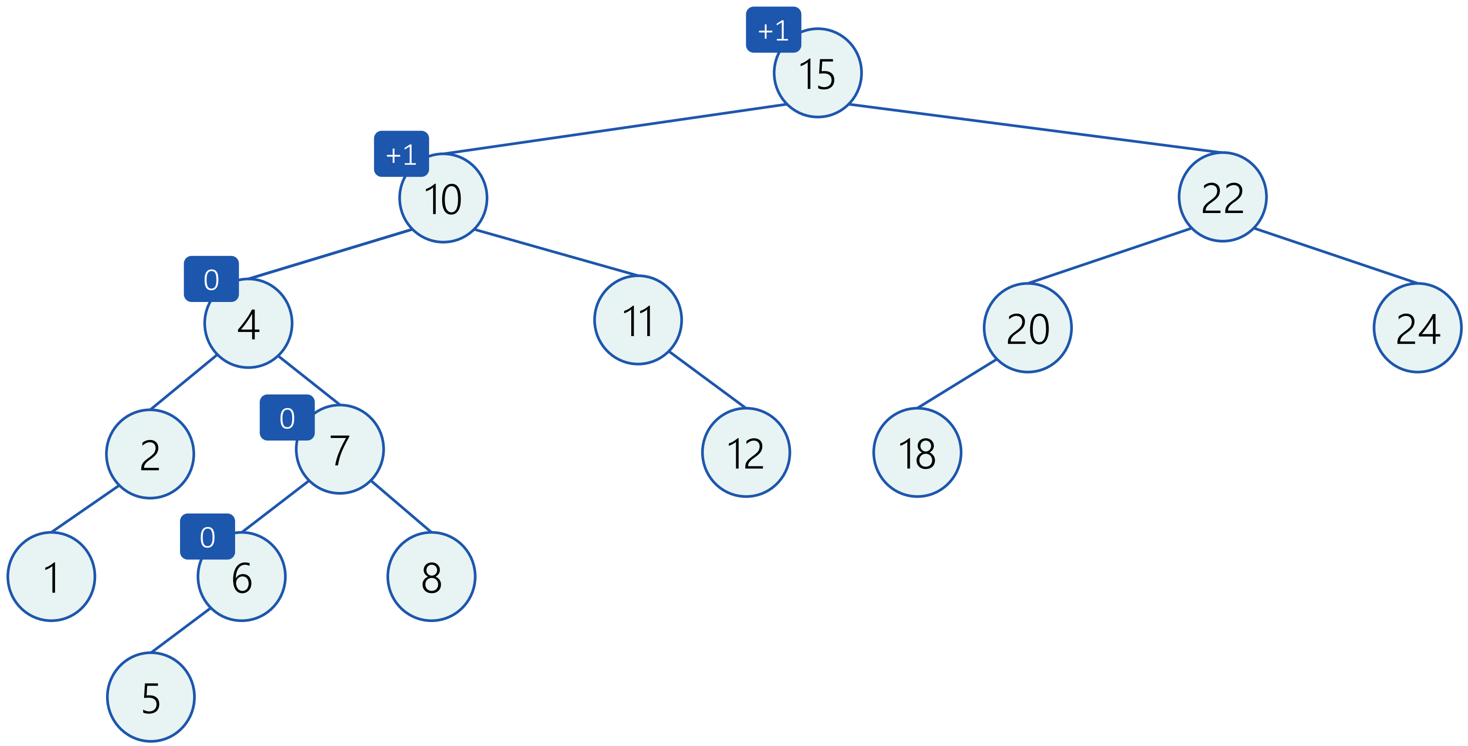
Rotate left then right

After rotation

Interactive (https://visualgo.net)

Left Then Right Rotation

Example



Before insertion

Insert 5

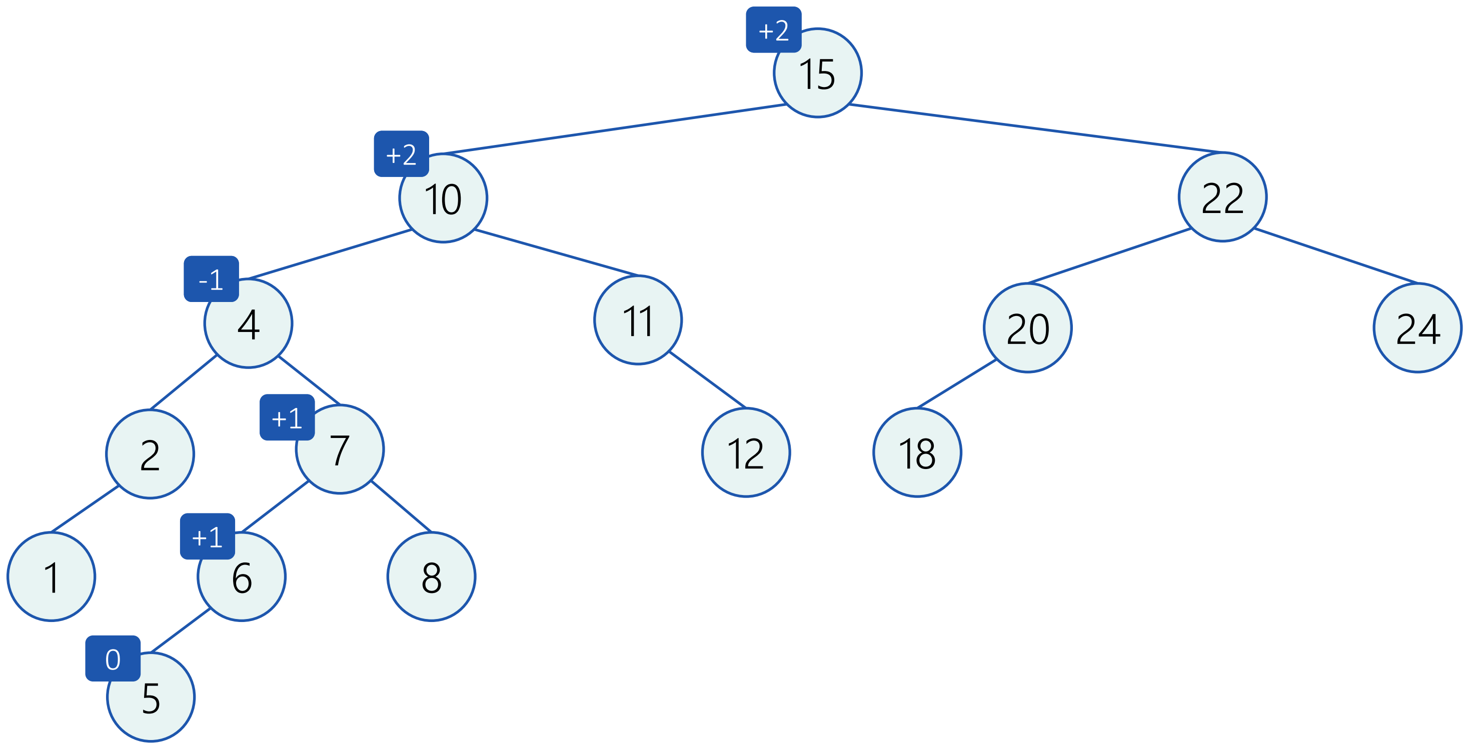
Temporarily after insertion

Rotate left then right

After rotation

Left Then Right Rotation

Example



Before insertion

Insert 5

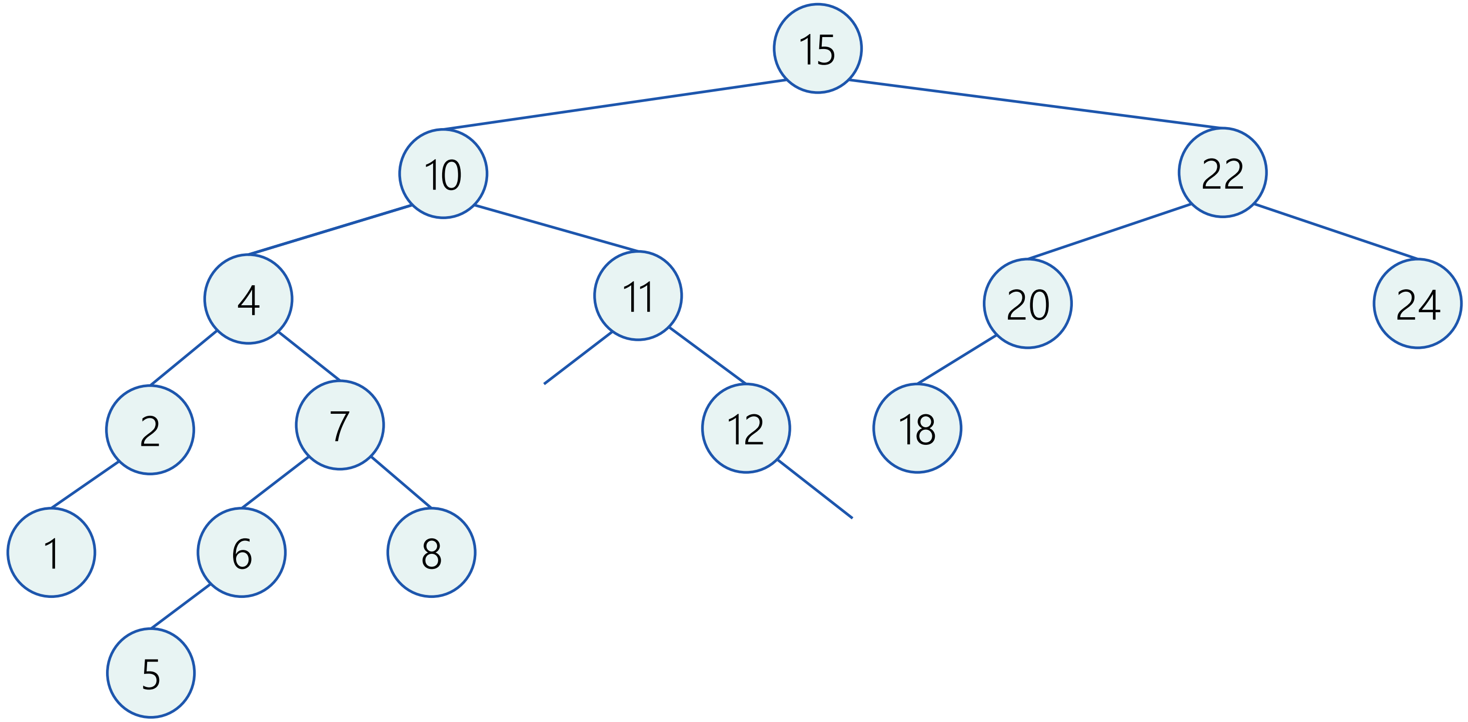
Temporarily after insertion

Rotate left then right

After rotation

Left Then Right Rotation

Example



Before insertion

Insert 5

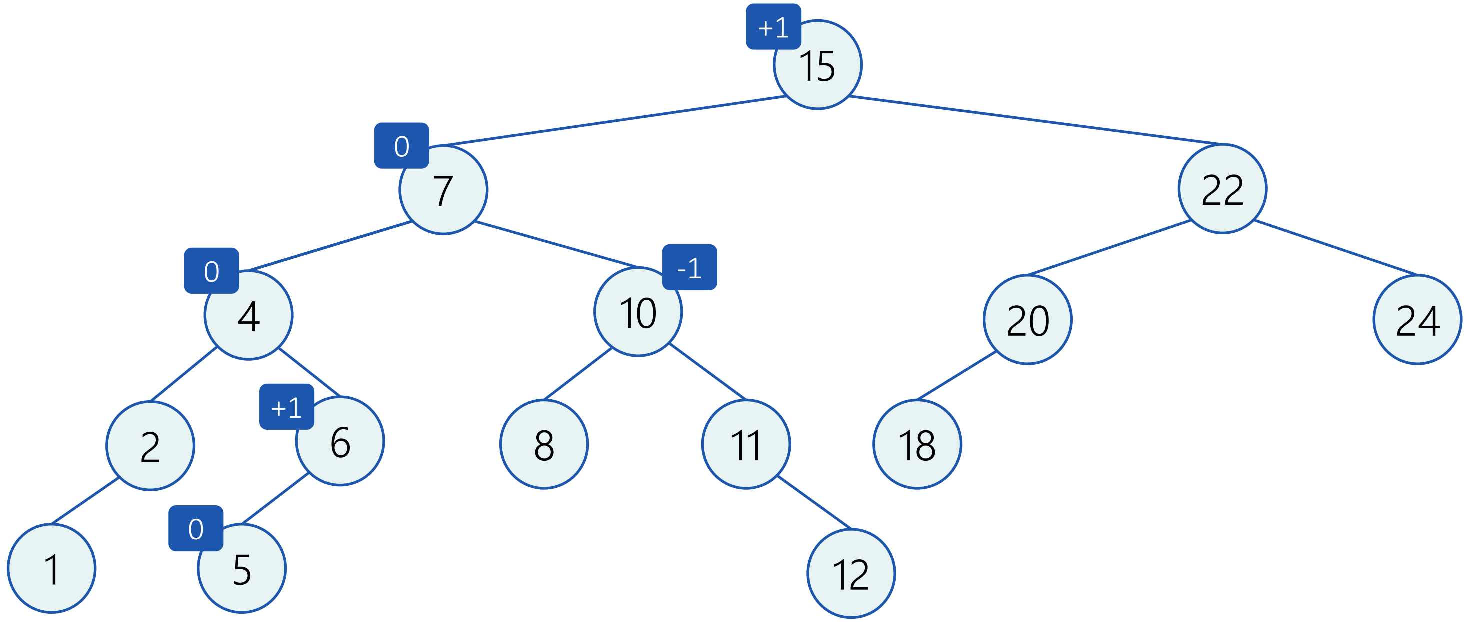
Temporarily after insertion

Rotate left then right

After rotation

Left Then Right Rotation

Example



Before insertions

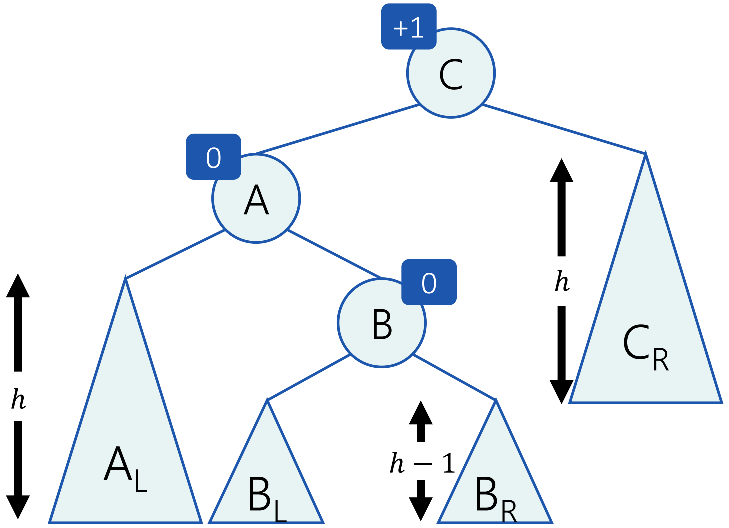
Insert 5

Temporarily after insertion

Rotate left then right

After rotation

Left Then Right Rotation



Before insertion

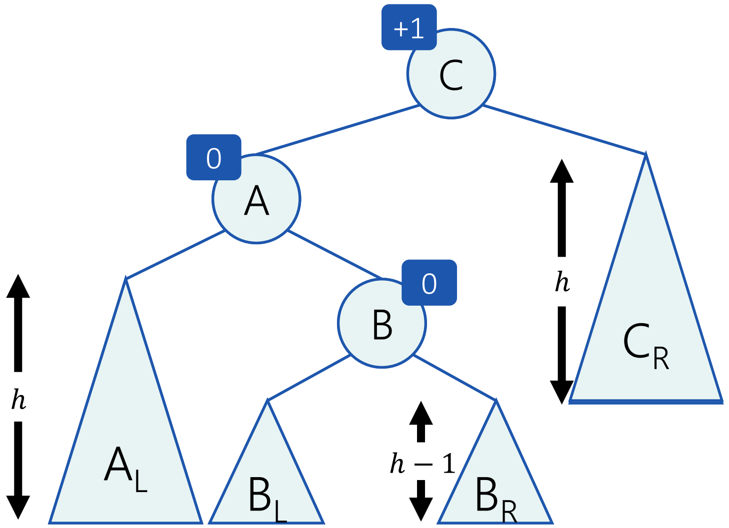
Insert v

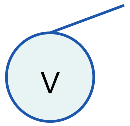
Temporarily after insertion

Rotate left then right

After rotation

Left Then Right Rotation





Before insertion

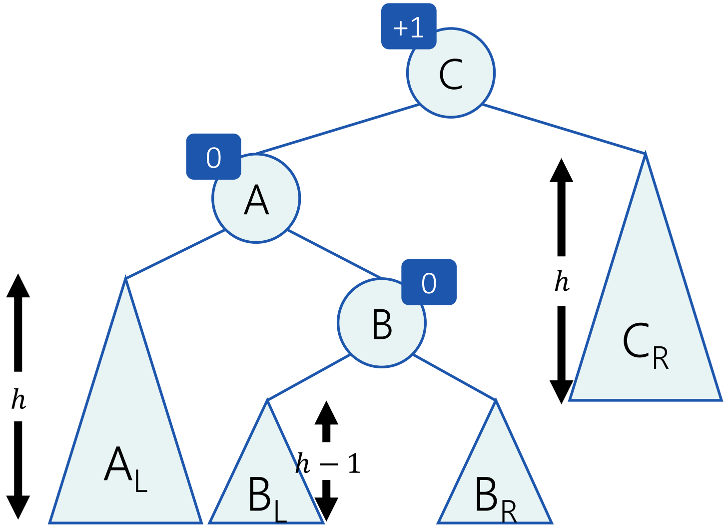
Insert v

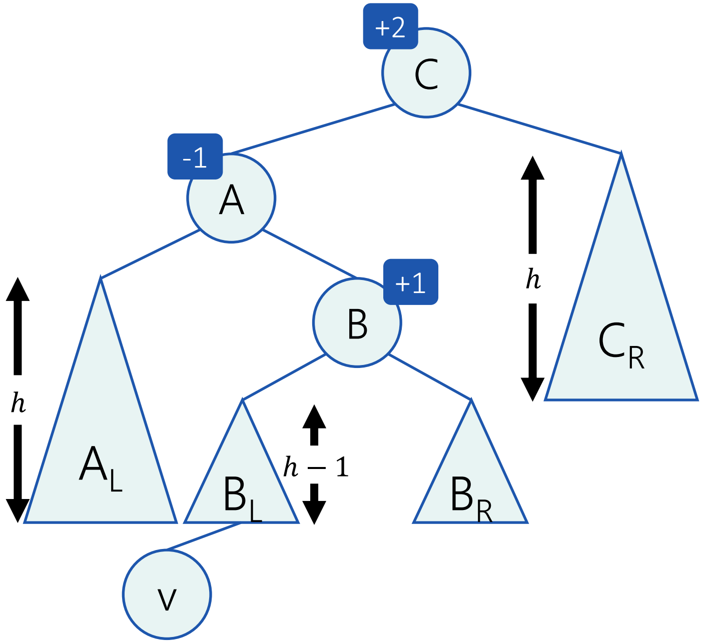
Temporarily after insertion

Rotate left then right

After rotation

Left Then Right Rotation





Before insertion

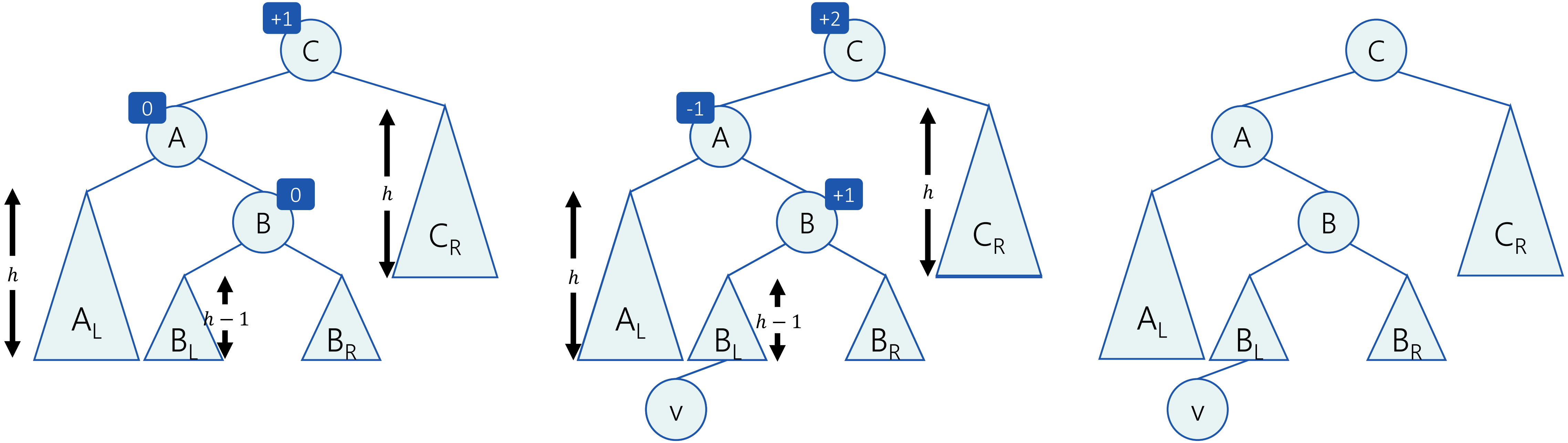
Insert v

Temporarily after insertion

Rotate left then right

After rotation

Left Then Right Rotation

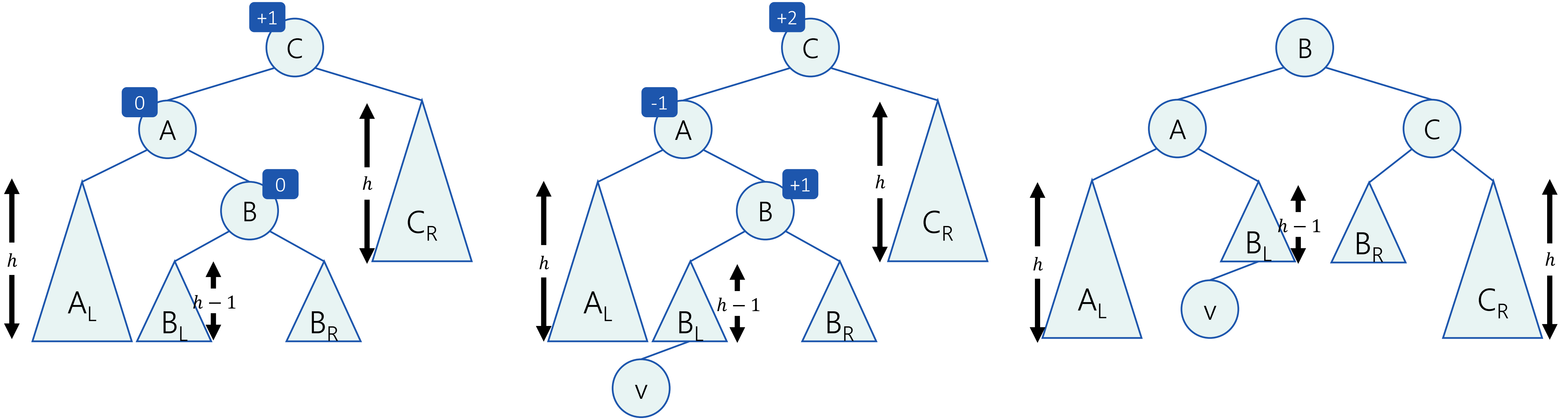


Before insertion

Insert v
Temporarily after insertion

Rotate left (A-B) then right (B-C)
After rotation

Left Then Right Rotation

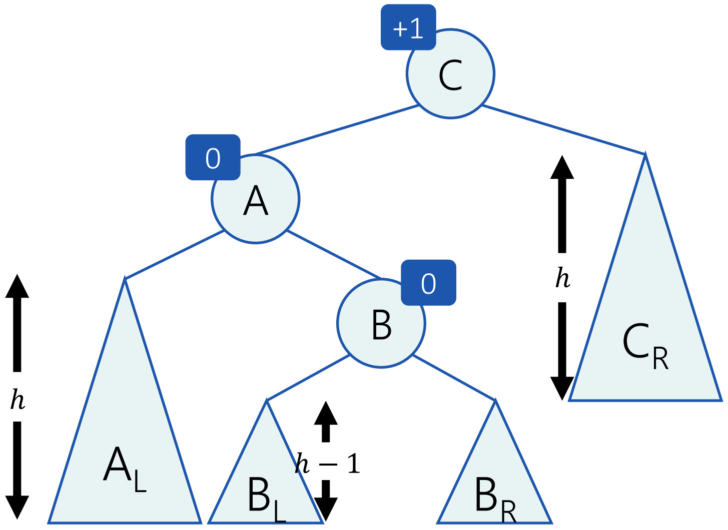


Before insertion

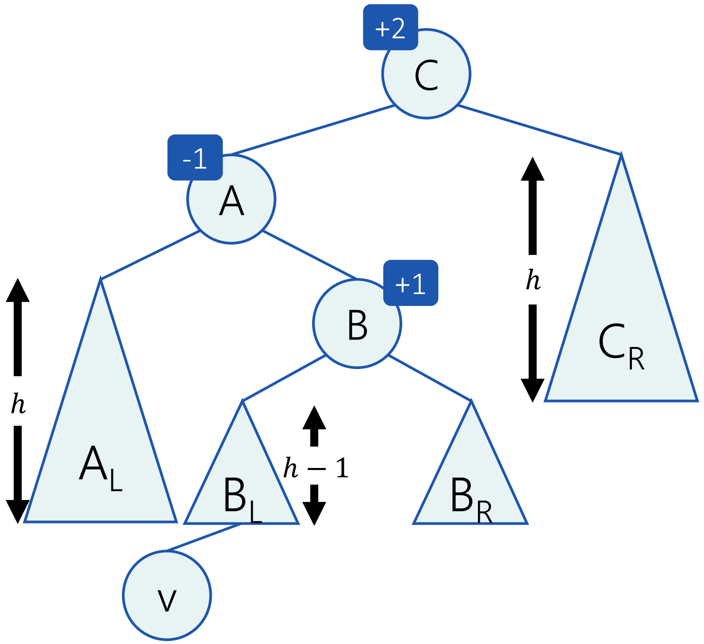
Insert v
Temporarily after insertion

Rotate left (A-B) then right (B-C)
After rotation

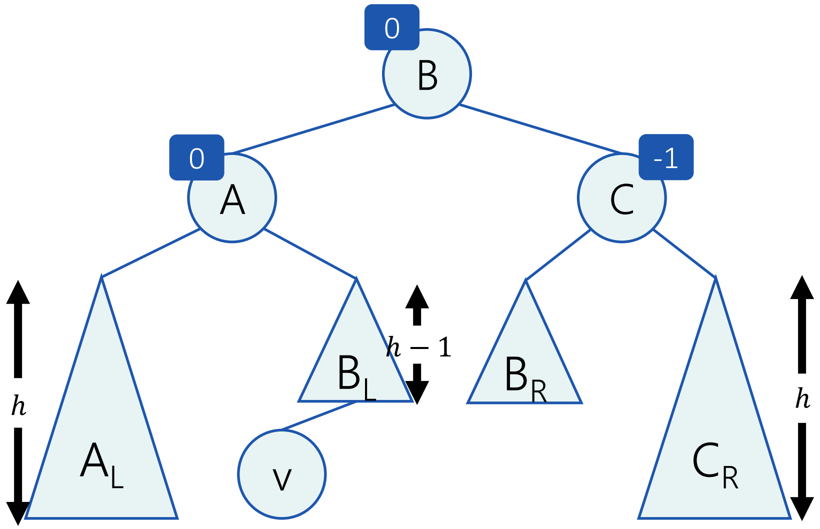
Left Then Right Rotation



Before insertion

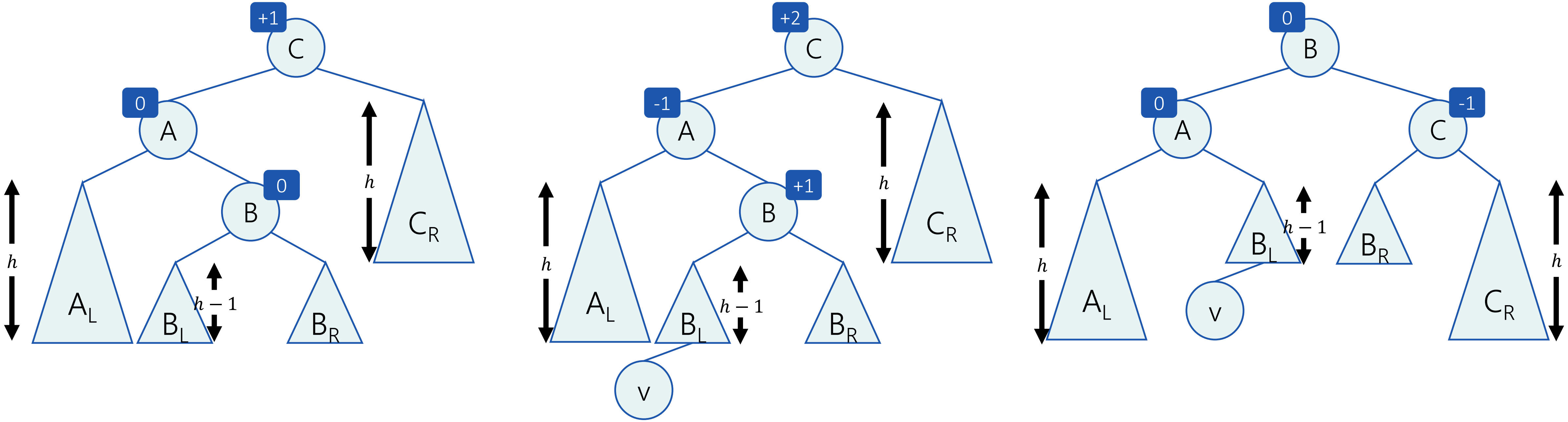


Insert v
Temporarily after insertion



Rotate left then right
After rotation

Left Then Right Rotation

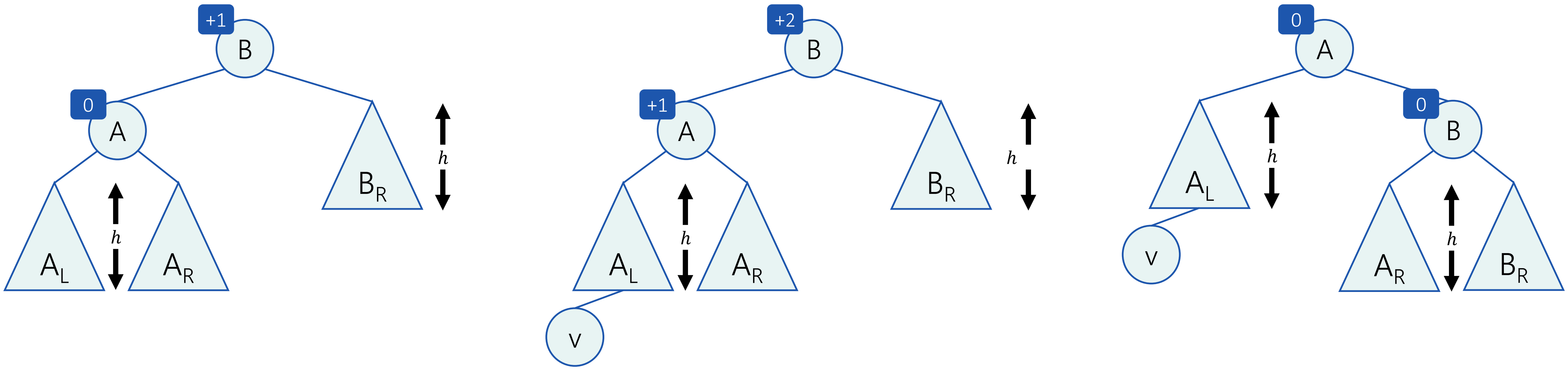


- The search tree property is preserved
- The violation of balance was fixed in this subtree
- Right then left rotation is symmetric

AVL Insertion:
One Rotation At Most

Number of Rotations After Insertion

Right Rotation (Left Rotation is symmetric):



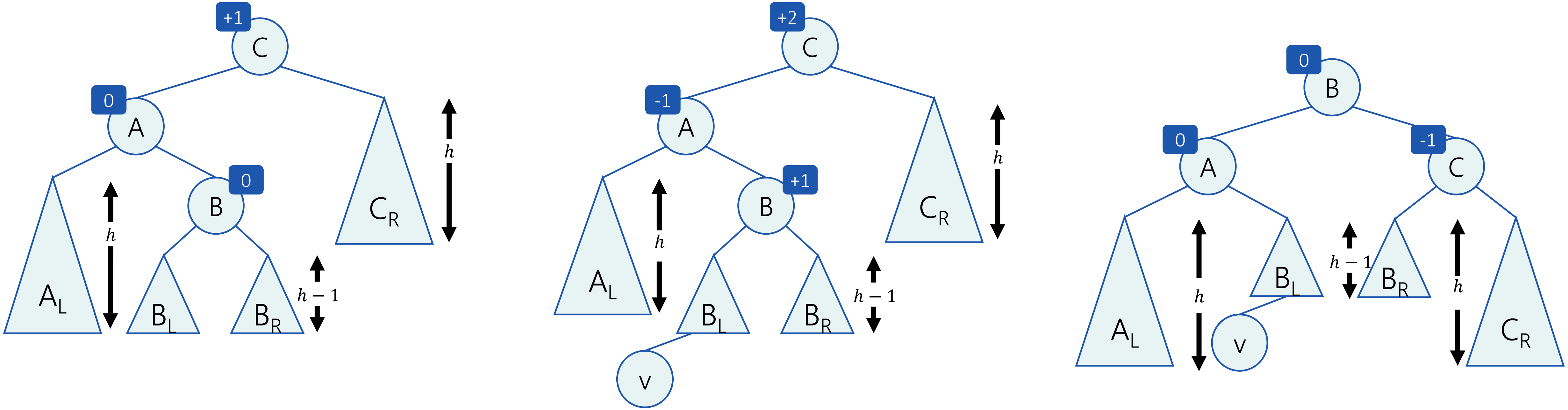
Before insertion
$height(B) = h + 2$

Right after insertion

After rotation
$height(A) = h + 2$

Number of Rotations After Insertion

Left Then Right Rotation (Right Then Left is symmetric):



Before insertion
$height(C) = h + 2$

Right after insertion

After rotation
$height(B) = h + 2$

Number of Rotations After Insertion

Conclusion:

After insertion, it takes one rotation at most in order to fix the tree.

Proof:

Rotation after insertion always restores the height of the sub-tree prior to the insertion.

The Insertion Algorithm

The Algorithm

AVL-Insert(T, z)

1. insert z as usual (as in a BST)
2. let y be the parent of the inserted node.
3. while $y \neq Null$ do:

   3.1. compute $BF(y)$*

   3.2. if $|BF(y)| < 2$ and y's height hasn't changed: terminate

   3.3. else if $|BF(y)| < 2$ and y's height changed: go back to stage 3 with y's parent

   3.4. else ($|BF(y)| = 2$): perform a rotation and go back to stage 3 with y's parent

*Requires maintaining additional information at each node. We will refer to this topic later.

The Insertion Algorithm

The Algorithm

AVL-Insert(T, z)

1. insert z as usual (as in a BST)
2. let y be the parent of the inserted node.
3. while $y \neq Null$ do:

   3.1. compute $BF(y)$*

   3.2. if $|BF(y)| < 2$ and y's height hasn't changed: terminate

   3.3. else if $|BF(y)| < 2$ and y's height changed: go back to stage 3 with y's parent

   3.4. else ($|BF(y)| = 2$): perform a rotation and terminate

*Requires maintaining additional information at each node. We will refer to this topic later.

Deletion From AVL

How to Keep It Balanced?

Fixing After Deletion

Rotations

- Deletion may also cause balance factor violation
- “Criminals” may appear on the path from the deleted node to the root
  - Deleted node = physically deleted (not necessarily the element removed)

- The 4 types of rotations are also used here to fix the violation.

Fixing After Insertion

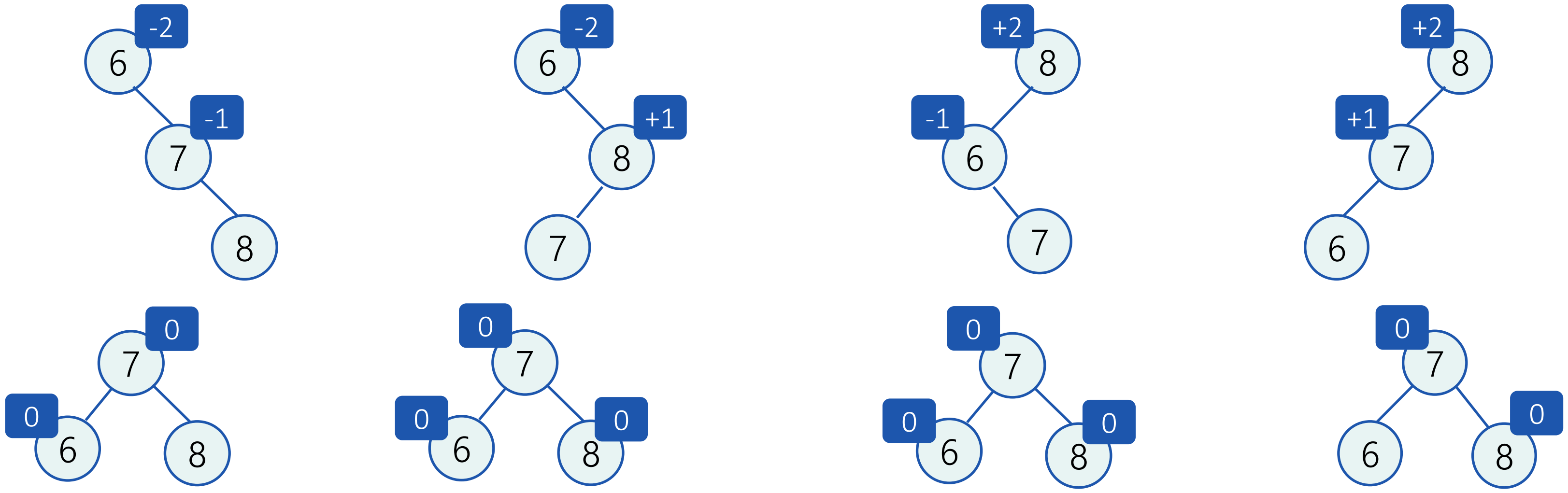
Rotations

What is the “criminal” BF?

-2		+2	
What is the BF of the **right** son?		What is the BF of the **left** son?	
-1 Left rotation	+1 Right then left rotation	-1 Left then right rotation	+1 Right rotation

Before rotation

After rotation



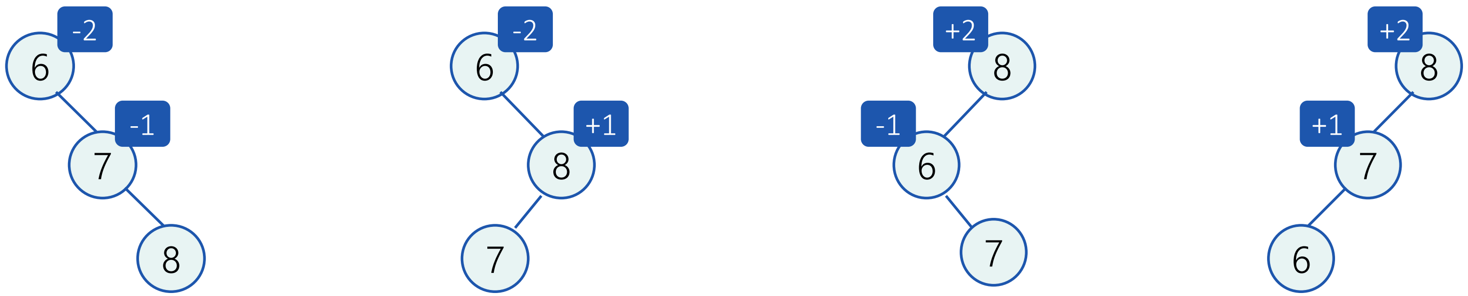
Fixing After Deletion

Rotations

What is the “criminal” BF?

-2		+2	
What is the BF of the **right** son?		What is the BF of the **left** son?	
-1 or 0 Left rotation	+1 Right then left rotation	-1 Left then right rotation	+1 or 0 Right rotation

Before rotation

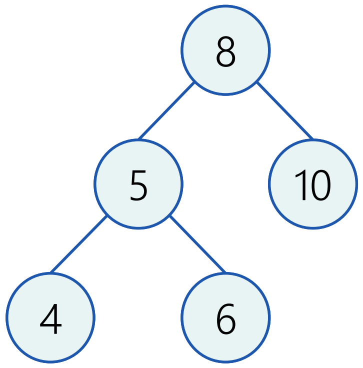


After rotation

Fixing After Deletion

Differences Between Insertion and Deletion

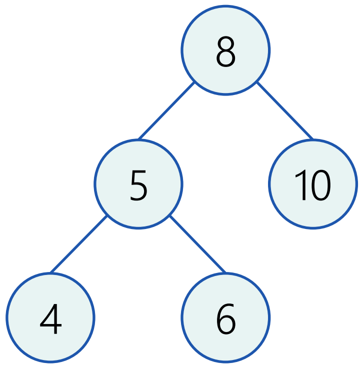
In deletion, there may be cases that cannot occur in insertion.
For example: criminal with BF=+2 whose left son has BF=0:

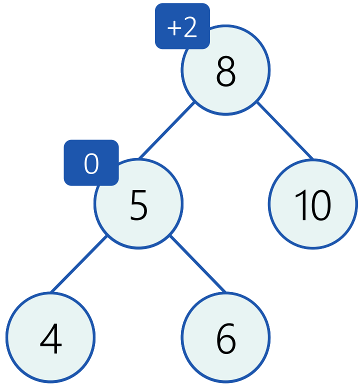


Fixing After Deletion

Differences Between Insertion and Deletion

In deletion, there may be cases that cannot occur in insertion.
For example: criminal with BF=+2 whose left son has BF=0:



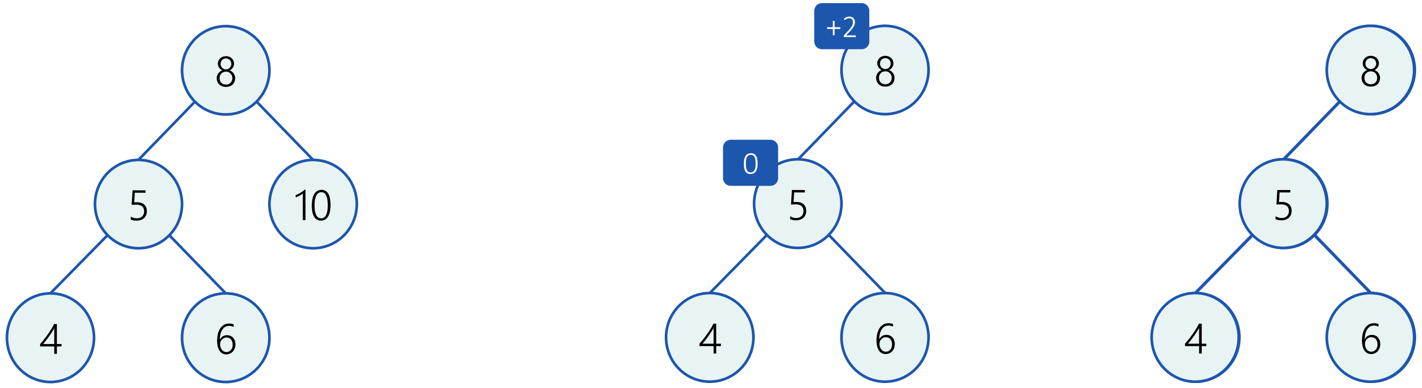


Delete 10
Rotate right

Fixing After Deletion

Differences Between Insertion and Deletion

In deletion, there may be cases that cannot occur in insertion.
For example: criminal with BF=+2 whose left son has BF=0:

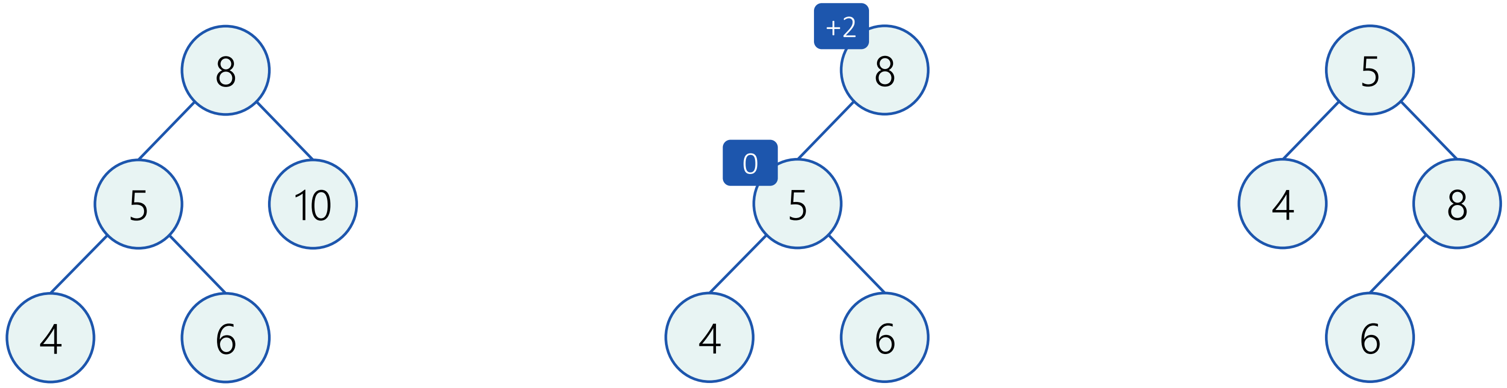


Delete 10
Rotate right

Fixing After Deletion

Differences Between Insertion and Deletion

In deletion, there may be cases that cannot occur in insertion.
For example: criminal with BF=+2 whose left son has BF=0:

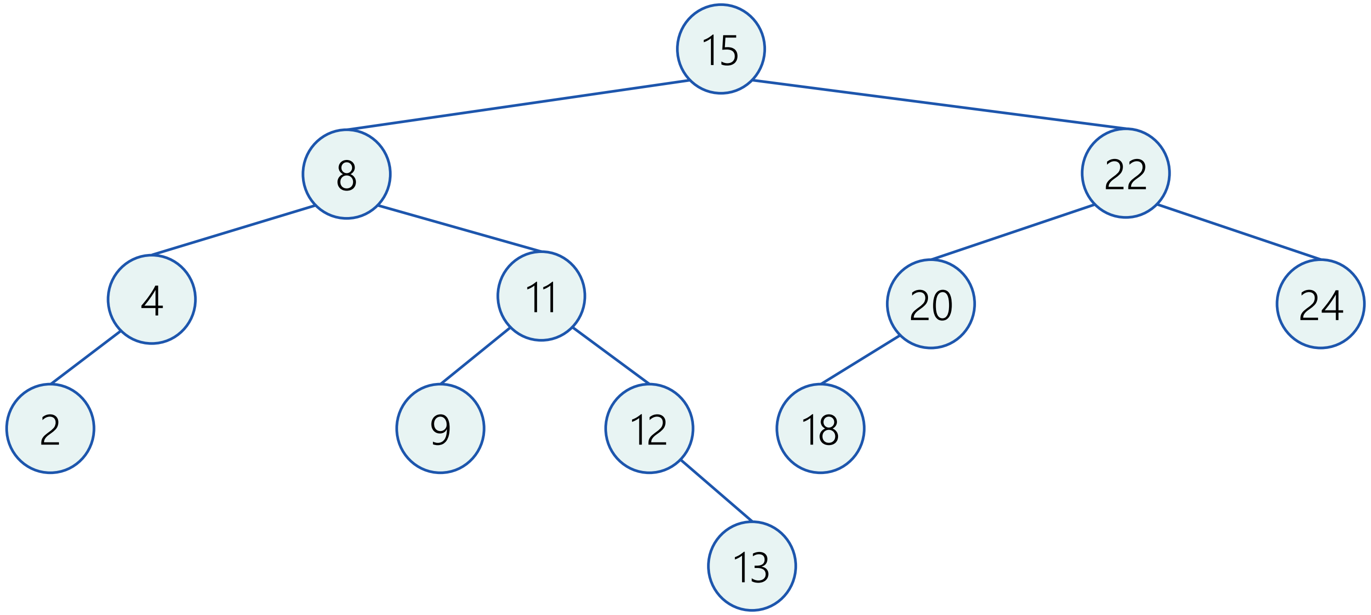


Delete 10
Rotate right

Fixing After Deletion

Differences Between Insertion and Deletion

In deletion, more than one rotation is possible
(one rotation may be performed at each level of the tree)



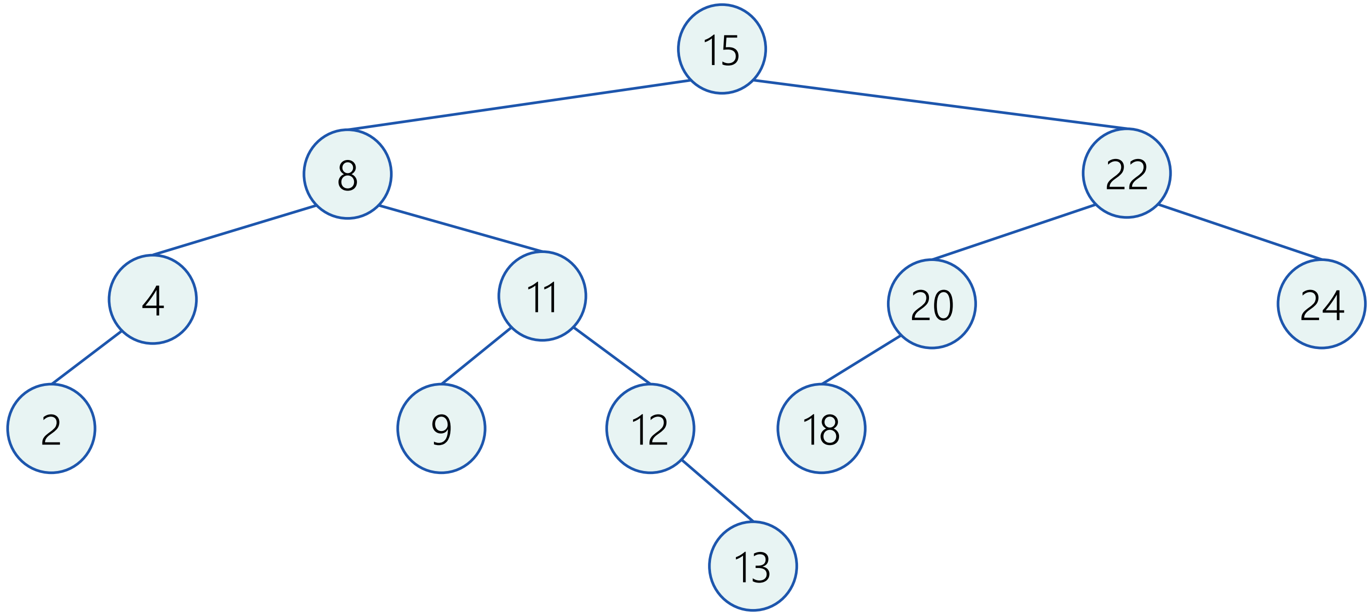
Delete 24
Rotate right
Rotate left then right

[Interactive](https://visualgo.net) (https://visualgo.net)

Fixing After Deletion

Differences Between Insertion and Deletion

In deletion, more than one rotation is possible
(one rotation may be performed at each level of the tree)

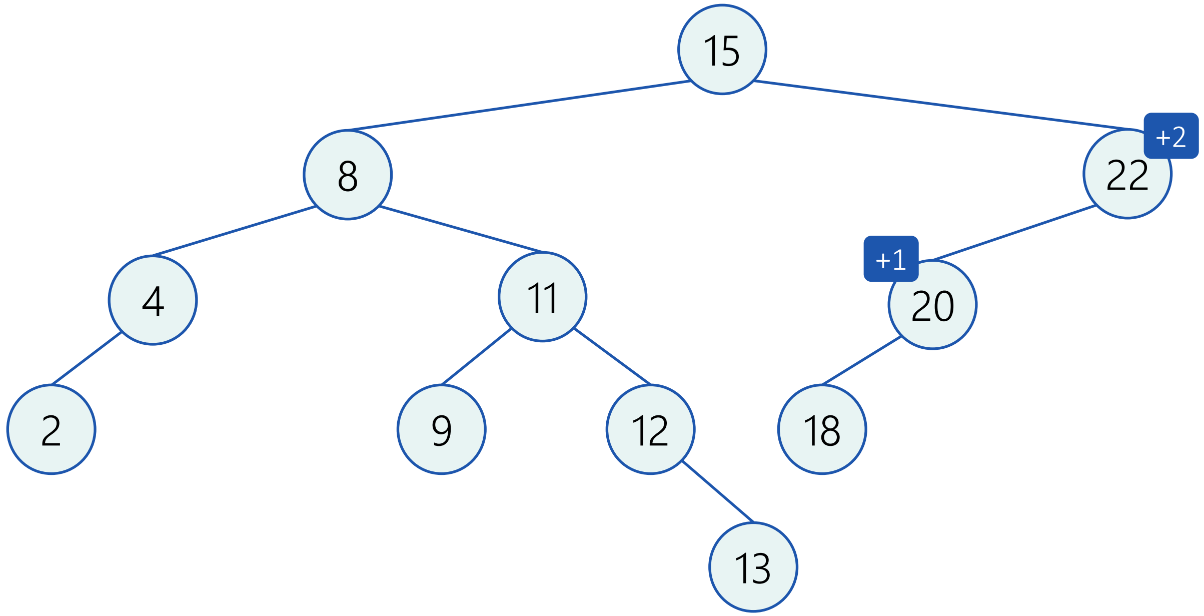


Delete 24
Rotate right
Rotate left then right

Fixing After Deletion

Differences Between Insertion and Deletion

In deletion, more than one rotation is possible
(one rotation may be performed at each level of the tree)

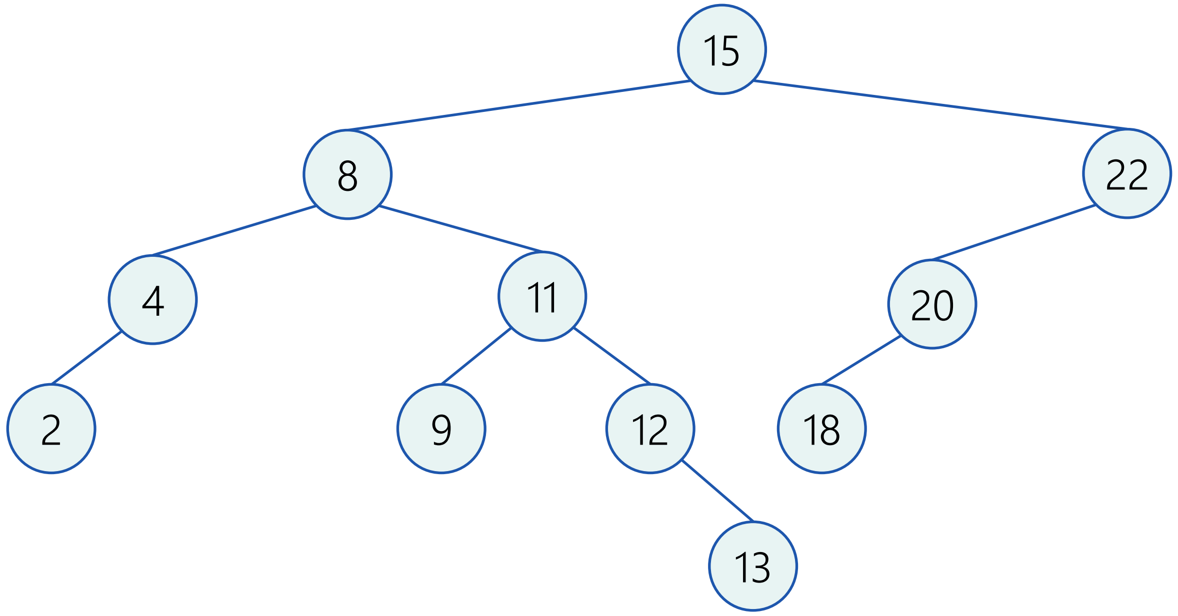


Delete 24
Rotate right
Rotate left then right

Fixing After Deletion

Differences Between Insertion and Deletion

In deletion, more than one rotation is possible
(one rotation may be performed at each level of the tree)

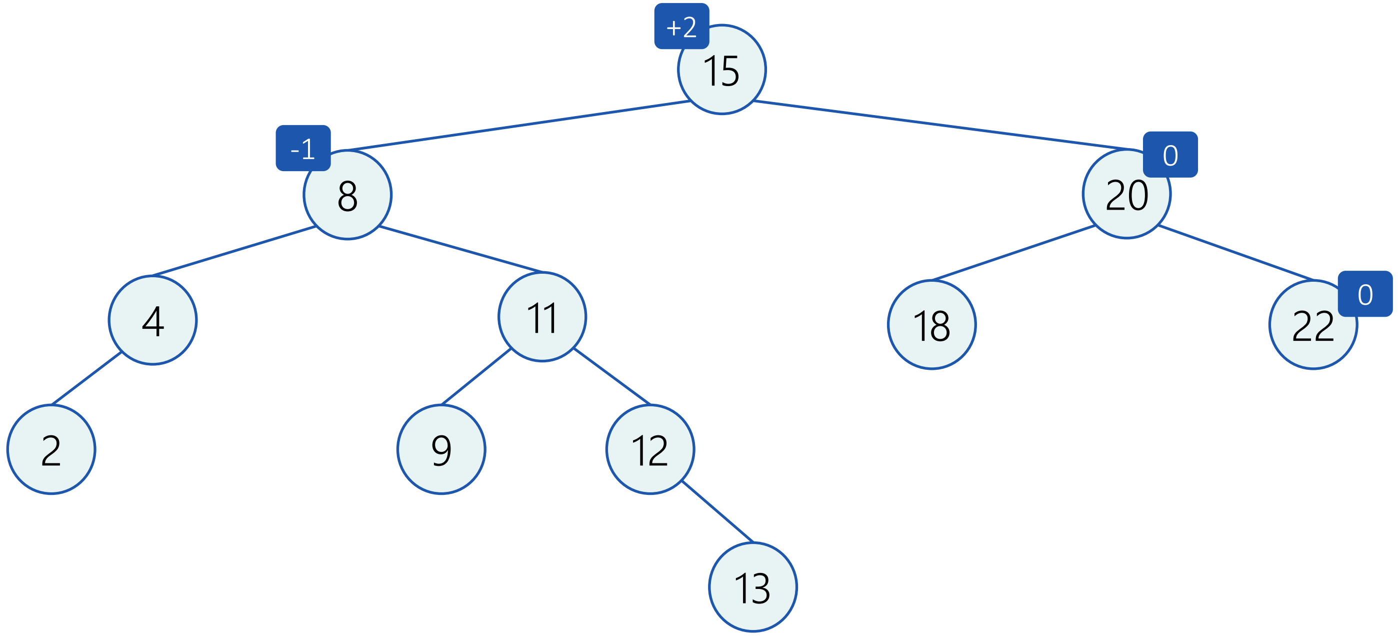


Delete 24
Rotate right
Rotate left then right

Fixing After Deletion

Differences Between Insertion and Deletion

In deletion, more than one rotation is possible
(one rotation may be performed at each level of the tree)

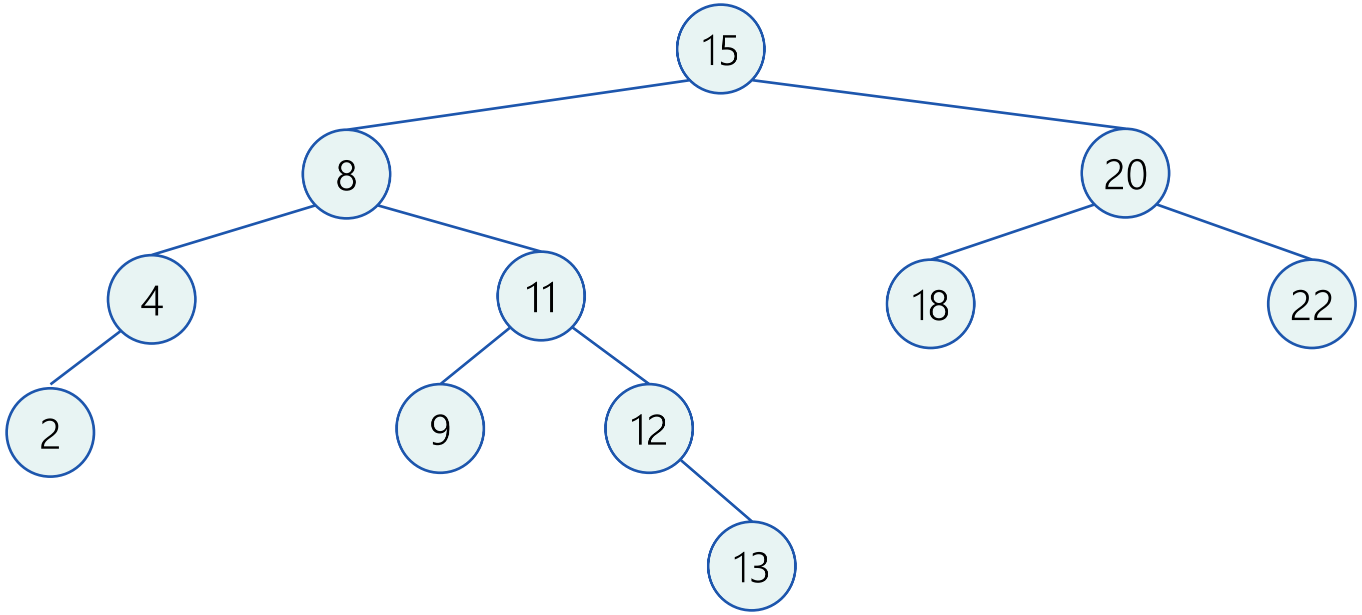


Delete 24
Rotate right
Rotate left then right

Fixing After Deletion

Differences Between Insertion and Deletion

In deletion, more than one rotation is possible
(one rotation may be performed at each level of the tree)

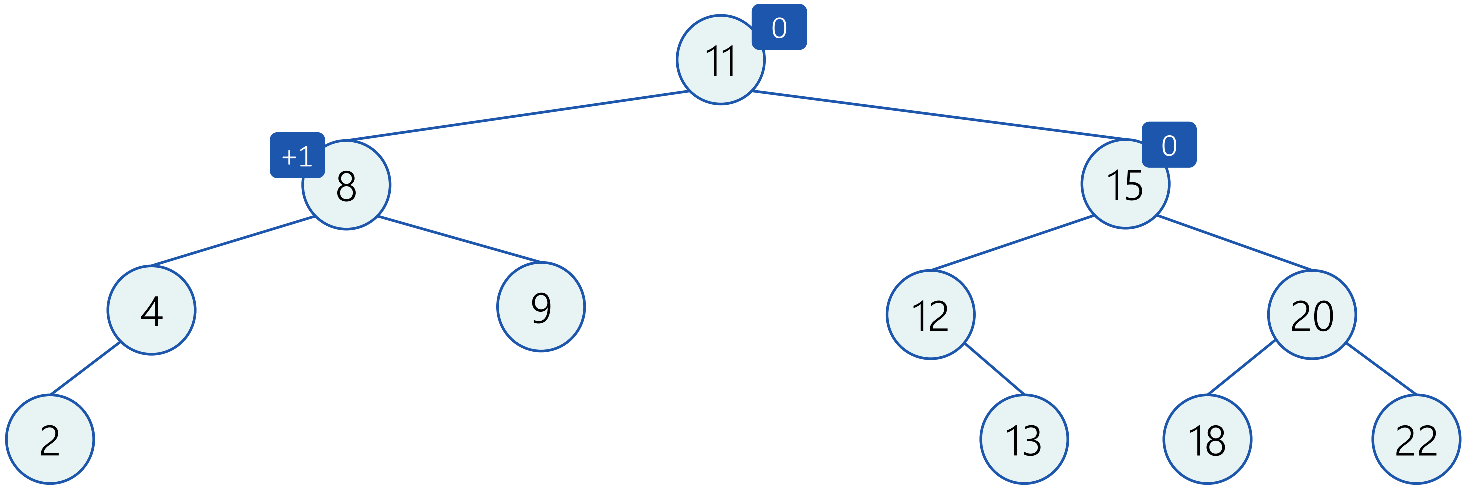


Delete 24
Rotate right
Rotate left then right

Fixing After Deletion

Differences Between Insertion and Deletion

In deletion, more than one rotation is possible
(one rotation may be performed at each level of the tree)



Delete 24
Rotate right
Rotate left then right

Fixing After Deletion

The Algorithm

AVL-Delete(T, z)

1. delete z as usual (as in a BST).
2. Let y be the parent of the (physically) deleted node.
3. while $y \neq Null$ do:

   3.1. compute $BF(y)$*

   3.2. if $|BF(y)| < 2$ and y's height hasn't changed: terminate

   3.3. else if $|BF(y)| < 2$ and y's height changed: go back to stage 3 with y's parent

   3.4. else ($|BF(y)| = 2$): perform a rotation and go back to stage 3 with y's parent

*Requires maintaining additional information at each node. We will refer to this topic later.

The Deletion Algorithm

Time Complexity

AVL-Delete(T, z)

1. Deleting from a BST $\quad O(h+1)$
2. At most one rotation at each level $\quad O(h+1)$

$$O(h+1) = O(\log n)$$

Recommended Animations on the Web

- https://people.ksp.sk/~kuko/gnarley-trees/Balance.html

  Explains rotations

- https://people.ksp.sk/~kuko/gnarley-trees/AVL.html

  Explains AVL, accompanied by a nice summary

- https://visualgo.net/en/bst?mode=AVL

  Explains AVL

Amortization in AVL

Example 4 - AVL (reminders)

- Inserting a node into AVL takes $O(logn)$ time
  - $O(logn)$ *W.C.* spent on finding the insertion place

  - $O(logn)$ *W.C.* spent on upward ascend to rebalance the tree (+maybe a rotation).
    - we'll prove this ascend to be $O(1)$ amortized
    - intuition: most insertions require little upward ascend

Amortized Cost – AVL: insertions only

Theorem 1 (AVL):
In any sequence of insertions (only), the *amortized* cost of rebalancing per insertion is $O(1)$

SIAM J. COMPUT.
Vol. 15, No. 1, February 1986

AN AMORTIZED ANALYSIS OF INSERTIONS INTO AVL-TREES*

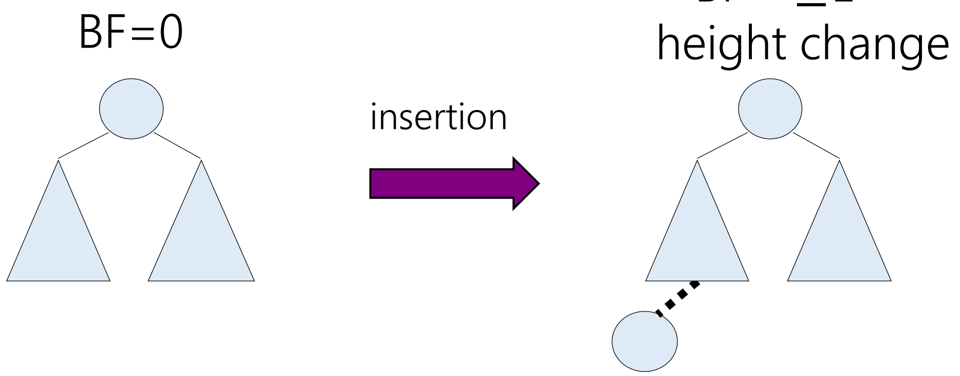
KURT MEHLHORN† AND ATHANASIOS TSAKALIDIS†

Abstract. We analyse the amortized behavior of AVL-trees under sequences of insertions. We show that the total rebalancing cost (=balance changes) for a sequence of n arbitrary insertions is at most $2.618n$. For random insertions the bound is improved to $2.26n$. We also show that the probability that t or more balance changes are required decreases exponentially with t.

Amortized Cost – AVL: insertions only

Reminder

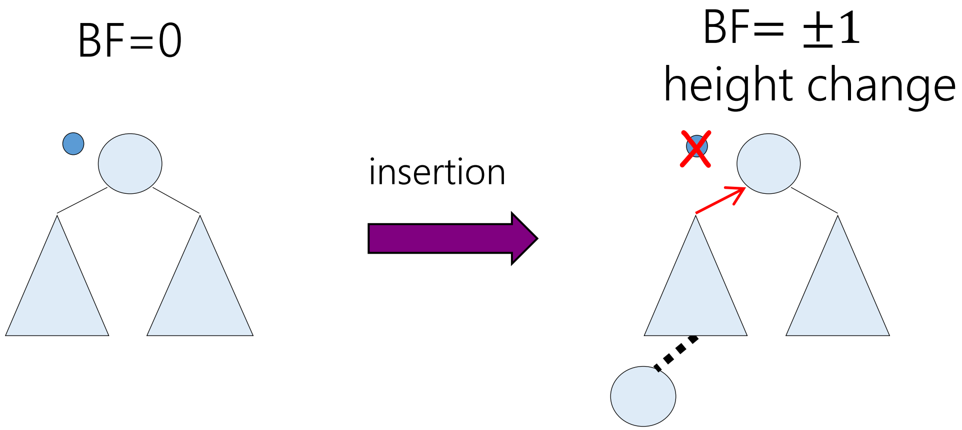
- After insertion, you only need to go up as long as the heights of nodes changed
- Such height changes are characterized by change from $\mathrm{BF} = 0$ (call these balanced nodes) to $\mathrm{BF} = \pm 1$
  - $\mathrm{BF} = \pm 1 \;\; \Rightarrow \;\; \mathrm{BF} = 0$ means height wasn't changed and we can terminate



- So cost of upward ascend ≤ number of previously balanced nodes along path (which as a result of insertion lost their balance)

Amortized Cost – AVL: insertions only

<u>Proof</u>

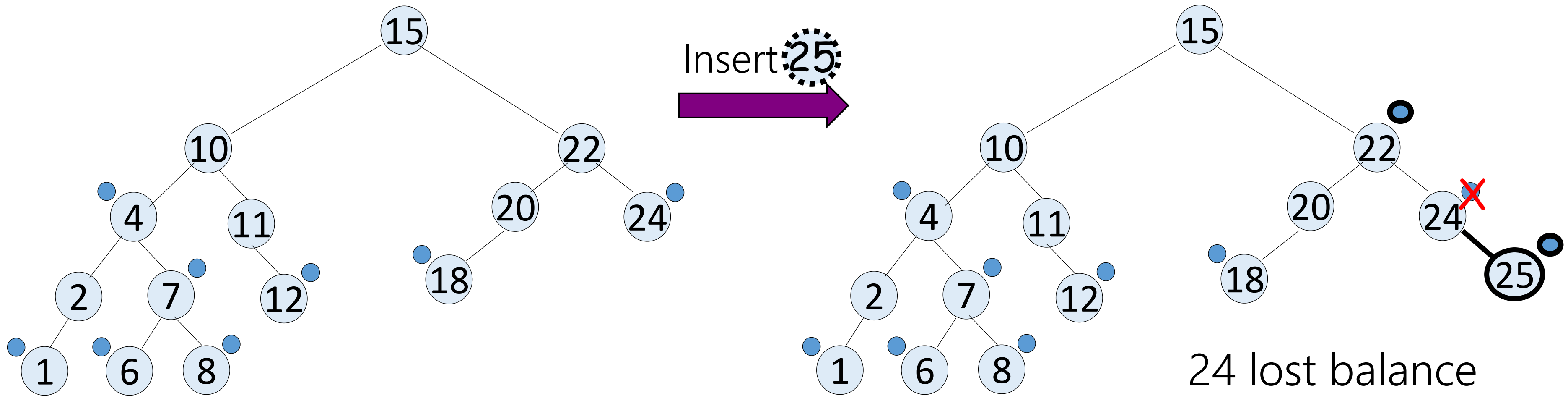


- Put $1 on each balanced node (alternatively, define Φ = No. of balanced nodes)
  - such nodes will "pay for" ascending towards them
  - Amortized cost of fixup is number of balanced nodes formed during ascend

- Prove that $O(1)$ balanced nodes are formed during insertion

Amortized Cost – AVL: insertions only

Example:

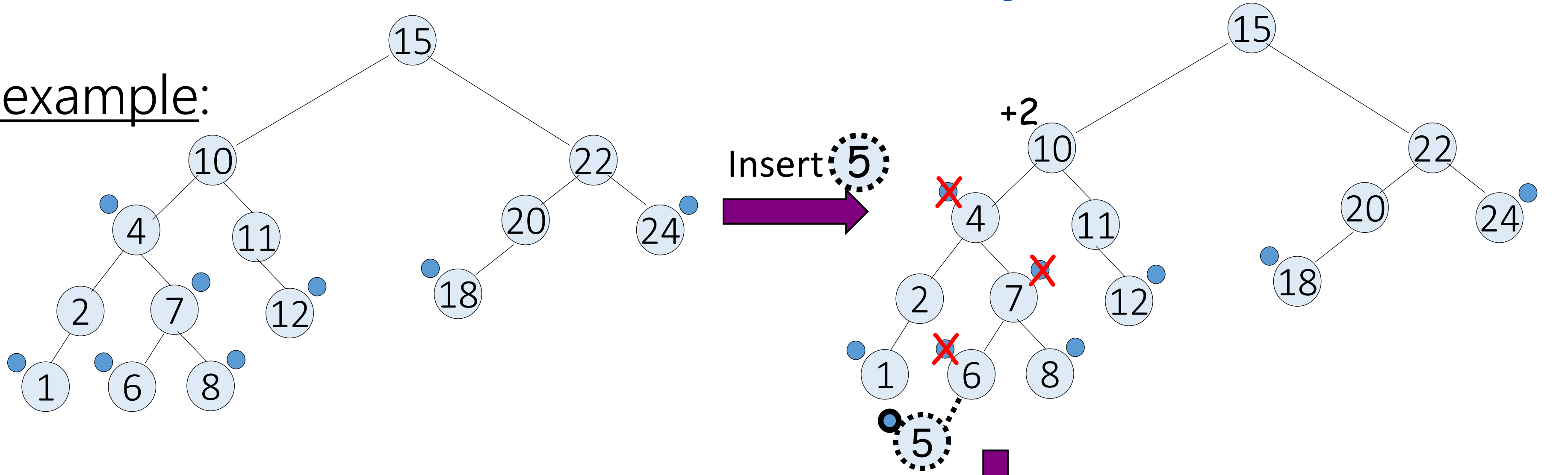


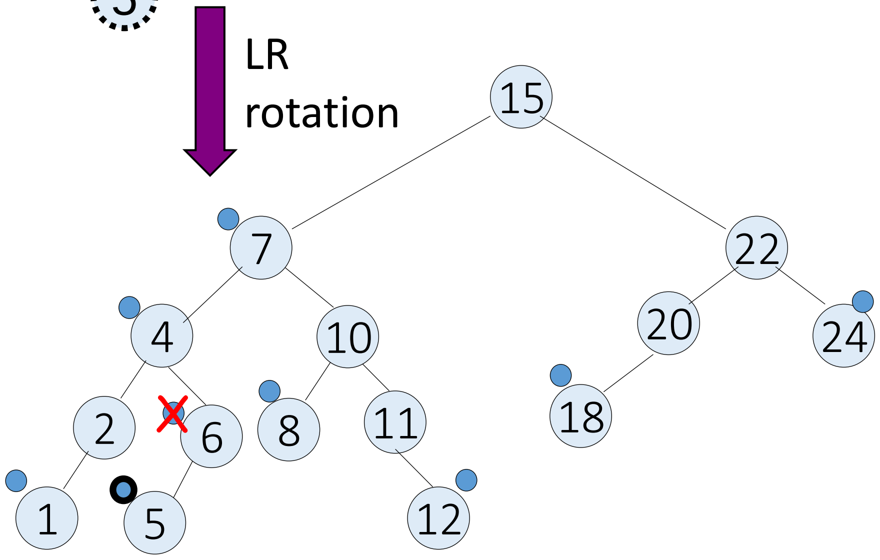
24 lost balance

25 and 22 are new balanced nodes

Amortized Cost – AVL: insertions only

<u>Another example</u>:



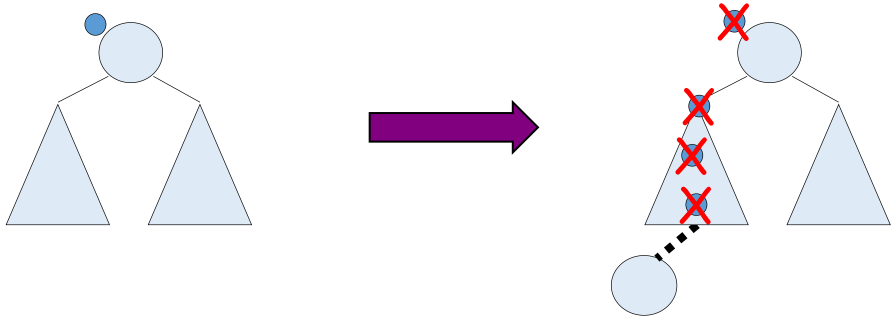


- 5 is a new balanced node
- 6,7,4 stop (temporarily) being balanced
- 4 and 7 retain balance

Amortized Cost – AVL: insertions only

Proof (cont.): during insertion,

- The new inserted node is balanced
- Then we go upward, as long as nodes changed their height
  - such nodes became unbalanced



- This process ends either when:
  - reaching an AVL criminal → rotation
    forming at most two balanced nodes (look at rotations)
  - or reaching a node whose height did not change
    it must be a new balanced node
- So each insertion buys ≤ 3● → amortized cost $O(1)$

Δ balance

\+ ●

\- ●
per level

\+ ● ●

or

\+ ●

Amortized Cost – AVL: insertions only

Application 1:

- What is the cost for inserting $1,2,\dots,n$ (in sorted order) into an initially empty AVL, such that each insertion starts from the last node inserted?

- Trivial: $O(nlogn)$

- Conclusion from amortized bound shown: $O(n)$

Amortized Cost – AVL: insertions only

Application 2:

- Improving insertion sort using finger trees
- In the recitations / HW

- Bottom line: if array of size n has $I > n$ inversions*
  we can sort in $O(n \log(I/n))$

*Inversion: $i < j$ and $a_i > a_j$

Amortized Cost – AVL: insertions and then deletions

Theorem 2 (AVL):
In any sequence of insertions (only) and then deletions (only) the *amortized cost of rebalancing* is $O(1)$.

(proof omitted)

REBALANCING OPERATIONS FOR DELETIONS IN AVL-TREES (*)

by Athanasios K. Tsakalidis (1)

Communicated by W. Brauer

Abstract. — *Considering only deletions performed in an AVL-tree with n leaves, we show that the total rebalancing time for n arbitrary deletions is linear. More precisely, the number of balance and structural changes is bounded by* 1.618 *n.*

Amortized Cost – AVL: insertions/deletions intermixed

- However, if insertions and deletions are intermixed, the *amortized* cost of *rebalancing* is $\Omega(logn)$

- Prove! (hint – show a tree in which repeated insertion followed by deletion takes $\Theta(logn)$ upward ascend per operation

- If you like to know more take a look here:

“Amortized Rotation Cost in AVL Trees”
Mahdi Amani, Kevin A. Lai, Robert E. Tarjan

Heaupler, Sen, and Tarjan [2] conjectured that alternating insertions and deletions in an n-node AVL tree can cause each deletion to do $\Omega(\log n)$ rotations, but they provided no construction to justify their claim. We provide such a construction: we show that,

WAVL trees

- However, if insertions and deletions are intermixed, the *amortized* cost of *rebalancing* is $\Omega(\log n)$

- Can we improve that?
  - that is, have $O(1)$ amortized rebalancing time for any sequence of insertions and deletions?

- Yes! Weak-AVL (WAVL) trees (2015)