SWIFT
Data Structure - AVL

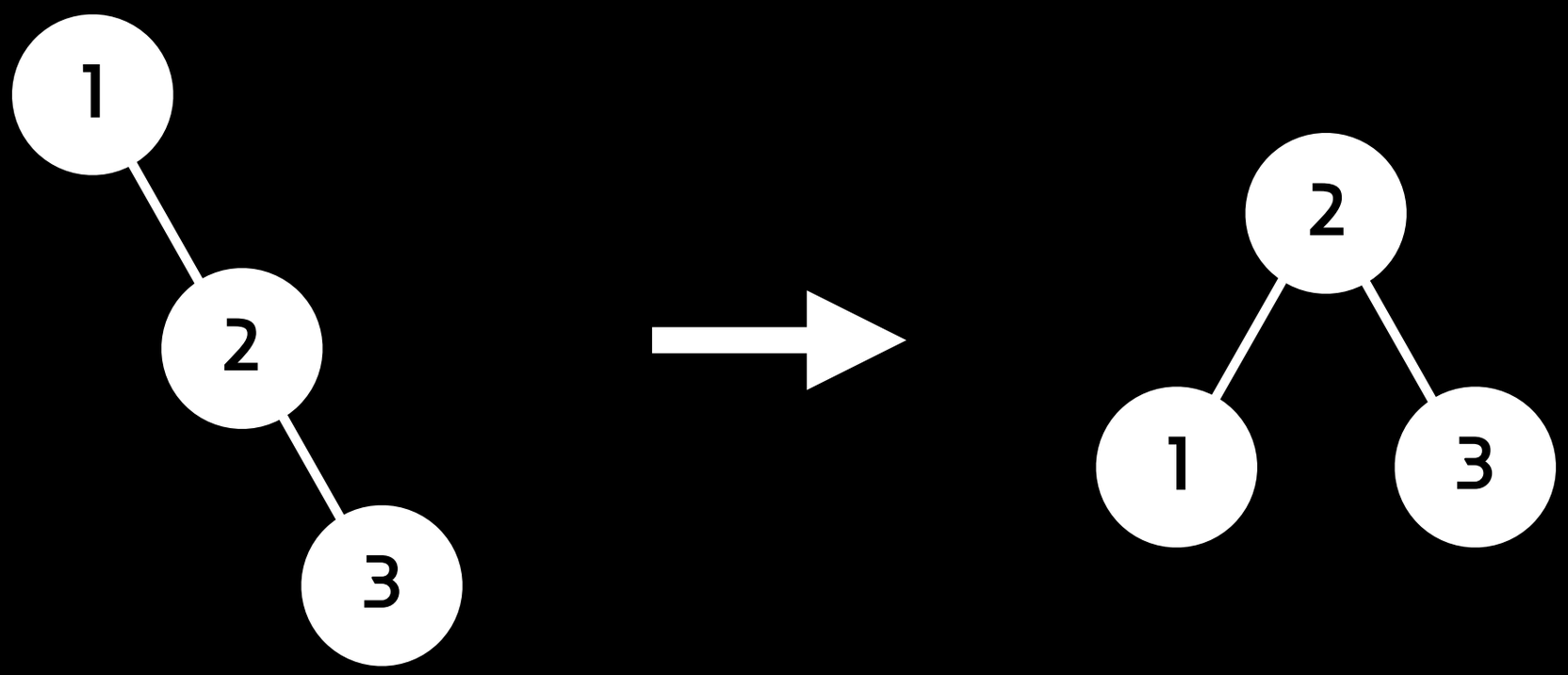
Bill Kim(김정훈) | ibillkim@gmail.com

목차

AVL

AVL는 자가 균형 트리로서 트리가 한쪽으로 치우쳐 트리가 가지는 높이의 균형이 깨지지 않도록 스스로 균형을 맞춰주는 이진 탐색 트리입니다.

트리의 균형을 항상 맞추면서 결국 탐색, 삽입, 삭제 연산의 수행 시간을 O(logn)을 보장할 수 있게 됩니다.



AVL

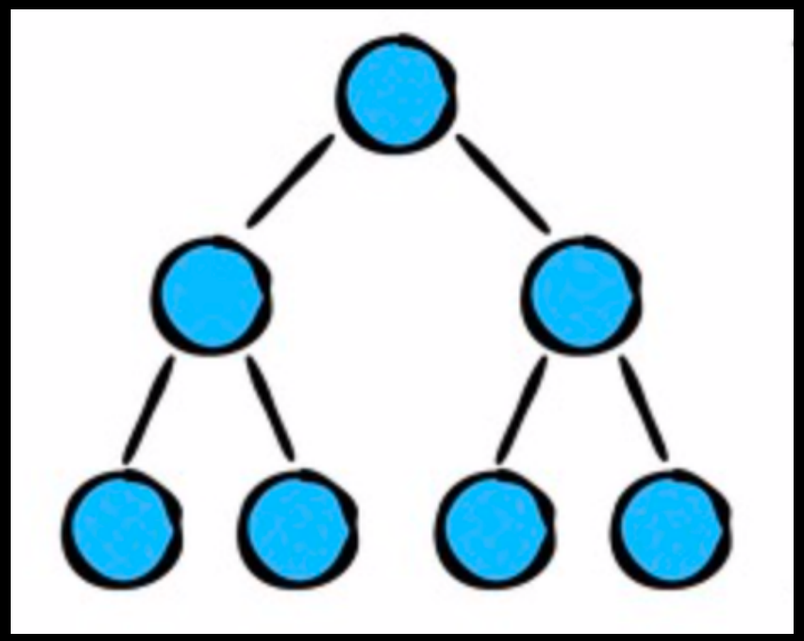
AVL 트리는 기본 연산은 이진 탐색 트리와 동일하지만 아래와 같이 최악의 경우 시간복잡도가 다릅니다.

이진탐색트리	평균	최악
공간	O(N)	O(N)
삽입	O(logN)	O(N)
삭제	O(logN)	O(N)
탐색	O(logN)	O(N)

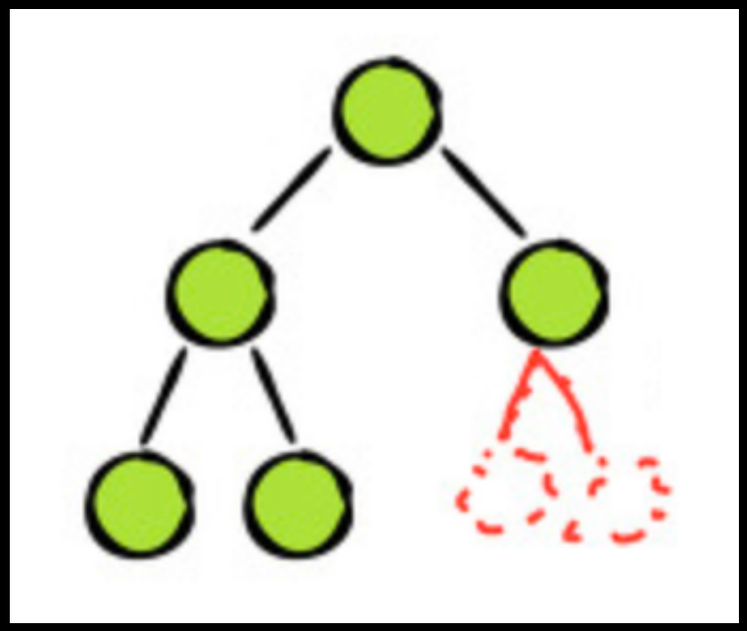
AVL트리	평균	최악
공간	O(N)	O(N)
삽입	O(logN)	**O(logN)**
삭제	O(logN)	**O(logN)**
탐색	O(logN)	**O(logN)**

Balance

그렇다면 이진 트리에서의 균형(Balance)이란 무엇일까요?
이진 트리에서 균형 상태는 크게 아래와 같은 형태를 말할 수 있습니다.



완전 균형 상태
(Perfect Balance)



충분한 균형 상태
(Good-enough Balance)

Balance

앞서 본 트리의 예시에서 알 수 있듯이 균형 상태를 다시 풀이해보면 아래와 같습니다.

균형 상태 : 각 노드의 두 하위 트리(왼쪽, 오른쪽)의 높이의 차이가 최대 1을 넘지 않은 상태

AVL 트리에서는 바로 트리의 높이를 가지고 최대 높이의 차이가 1 이하가 되도록 항상 균형을 맞추는 트리입니다.

해당 규칙을 가진 AVL트리는 바로 Adelson-Velskii와 Landis 에 의해 1962년에 제안된 트리입니다.

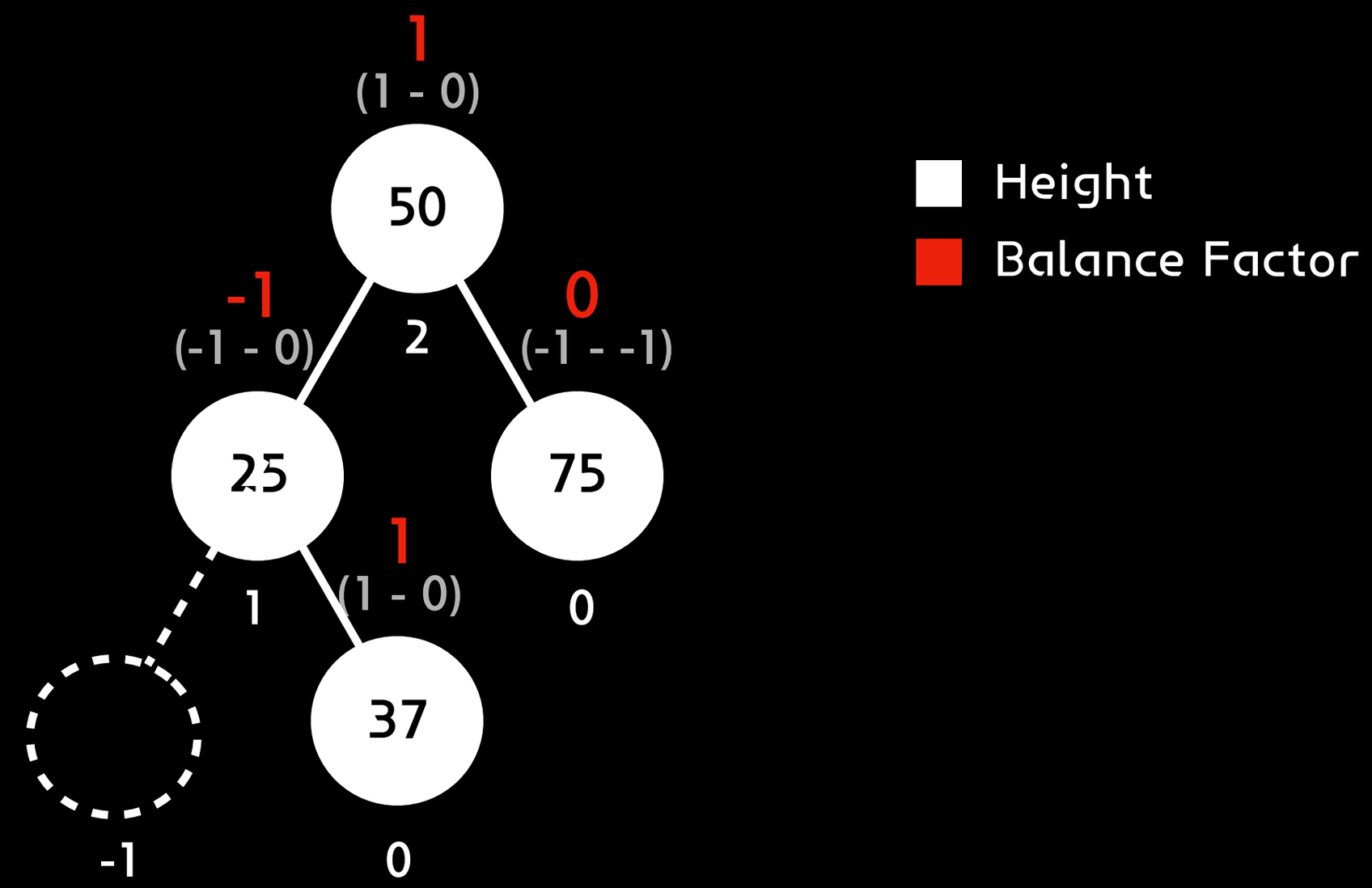
Balance

그렇다면 AVL 트리의 특징을 한번 살펴보겠습니다.

1. 균형의 상태를 나타내기 위한 균형도(Balance Factor)라는 개념이 있다.
2. 리프(Leaf) 노드의 균형도는 0 이다.
3. 균형도는 왼쪽 자식의 높이(Height) - 오른쪽 자식의 높이(Height)로 계산된다.
4. 트리의 높이(Height)란 특정 노드로부터 Leaf 노드까지 가장 긴 경로의 깊이를 말한다.
5. 빈 노드의 높이는 -1로 간주한다.
5. 왼쪽 자식 노드와 오른쪽 자식 노드의 균형도는 -1, 0, 1의 세 가지 값만 갖는다.

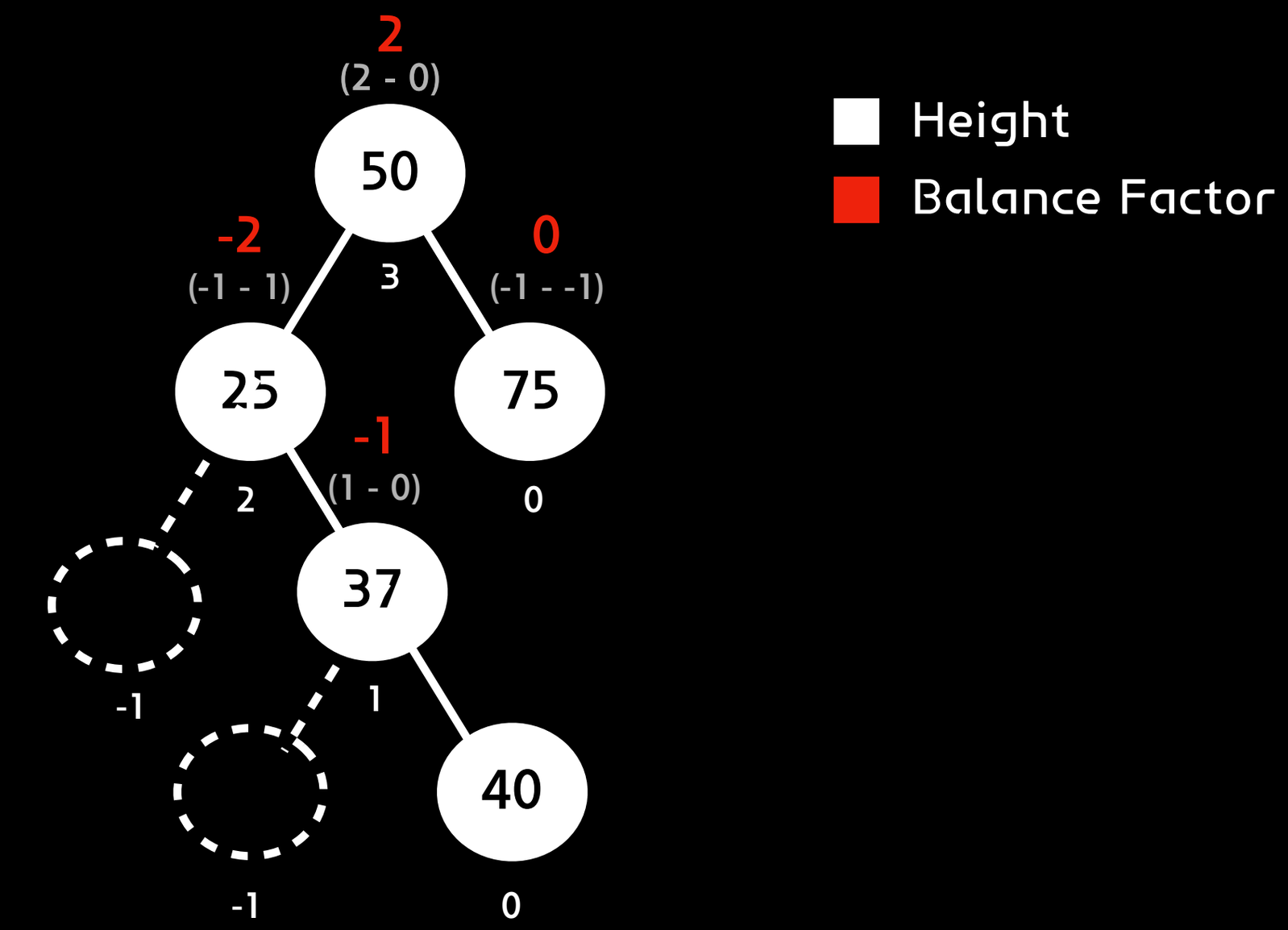
Balance

아래의 예시는 각 노드의 높이(Height)와 균형도(Balance Factor)를 나타내는 예시입니다.



Balance

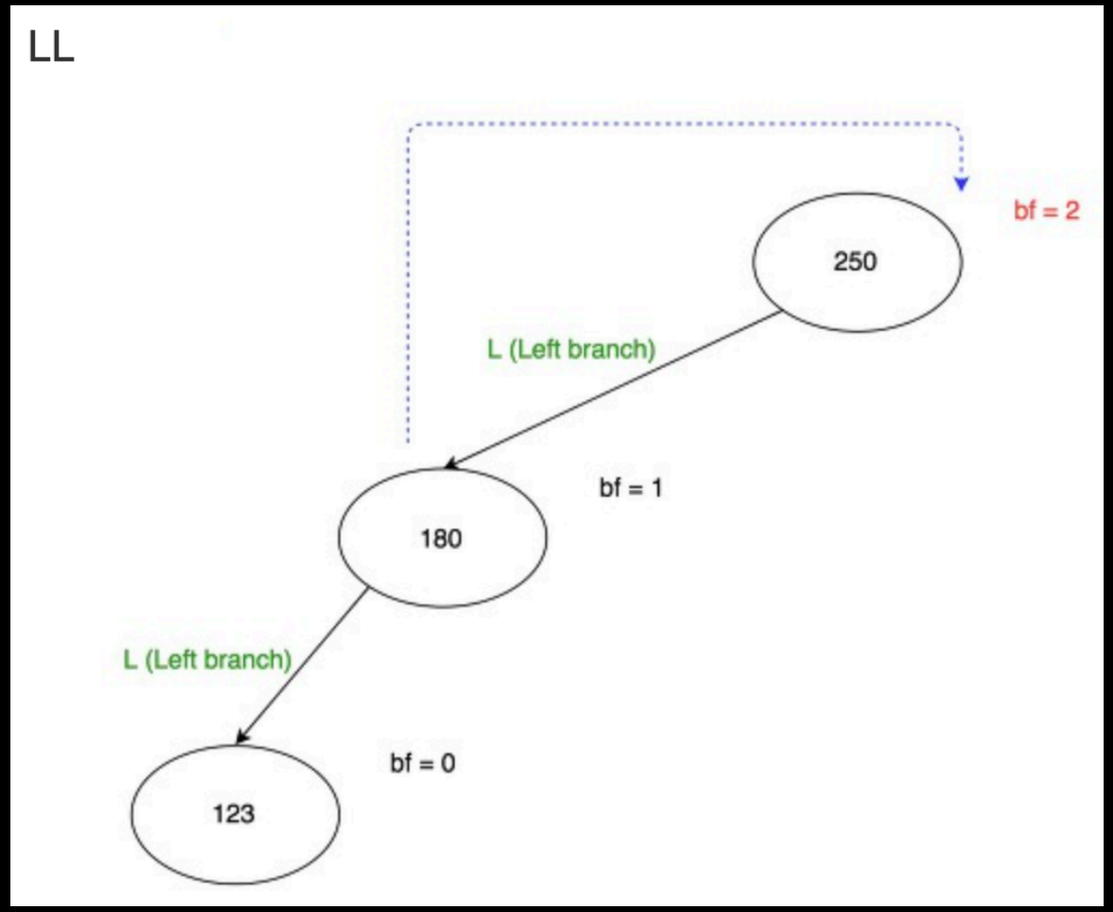
위의 예시에서 40이라는 노드를 삽입하면 다음과 같이 균형 상태가 깨지게 됩니다. 결국 균형도(Balance Factor) 값이 -2 또는 2가 되었을 경우 Unbalance 상태로 됨을 알 수 있습니다.



Unbalance

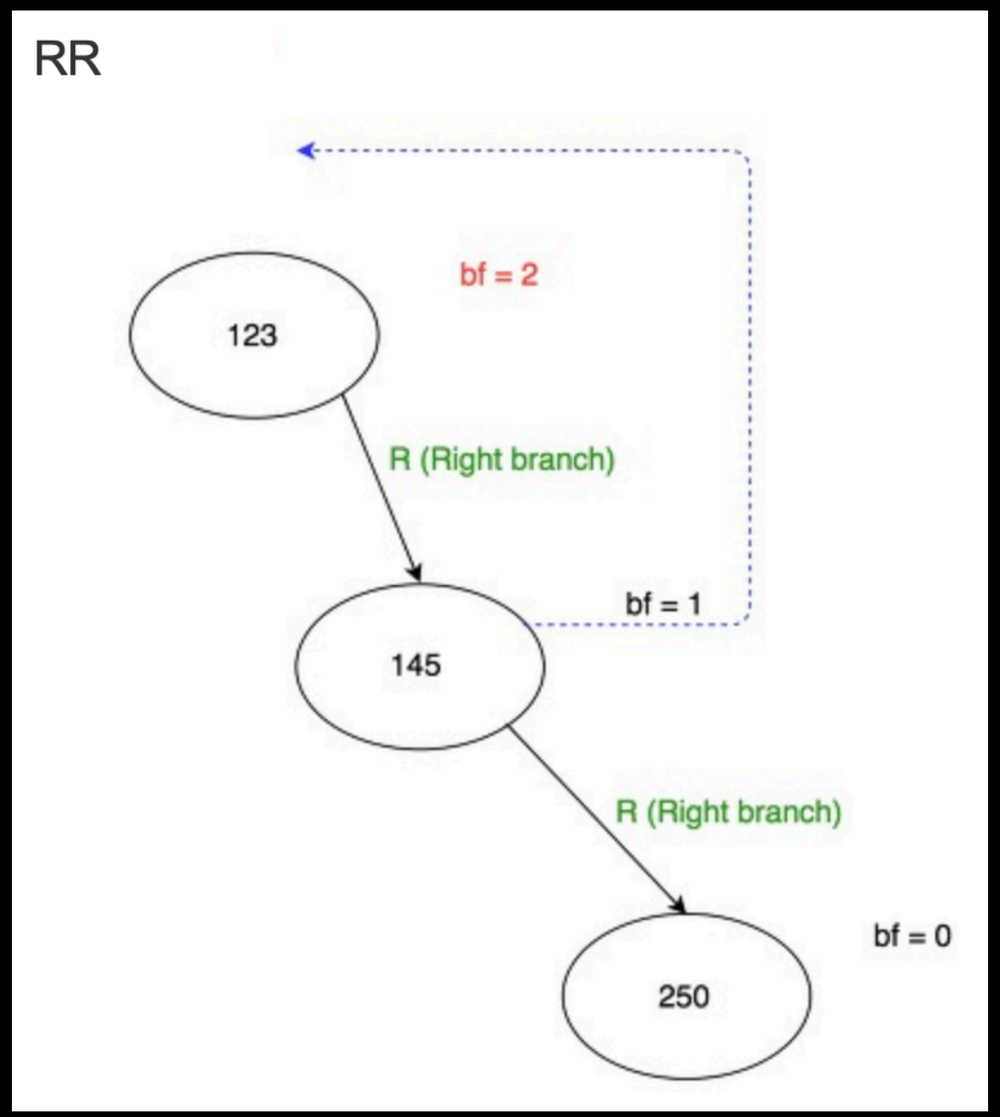
AVL 트리에서 균형이 깨지는 경우는 크게 4가지 경우가 있습니다.

(1) LL타입 : N이 A의 왼쪽 서브트리의 왼쪽에 삽입되는 경우(A노드의 왼쪽노드의 왼쪽에 삽입)



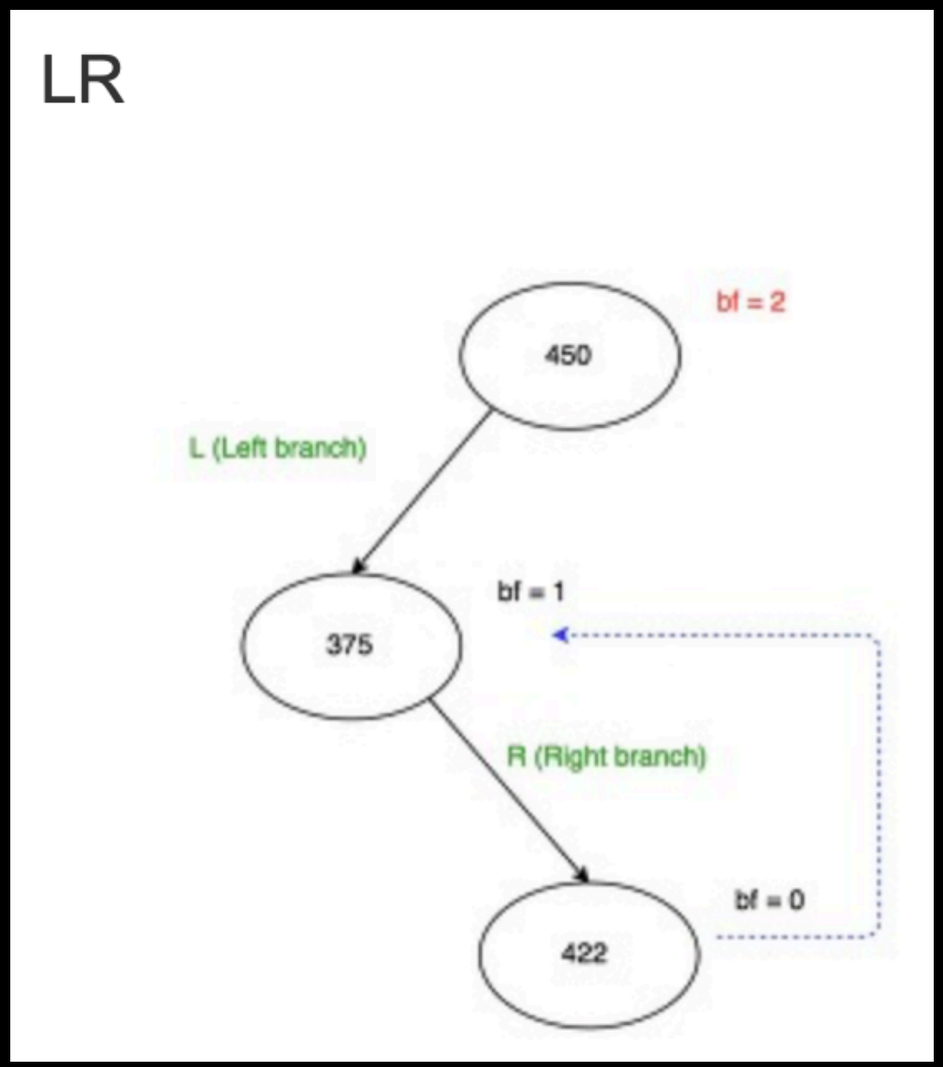
Unbalance

(2) RR타입 : n이 A의 오른쪽 서브트리의 오른쪽에 삽입되는 경우
(A노드의 오른쪽노드의 오른쪽에 삽입)



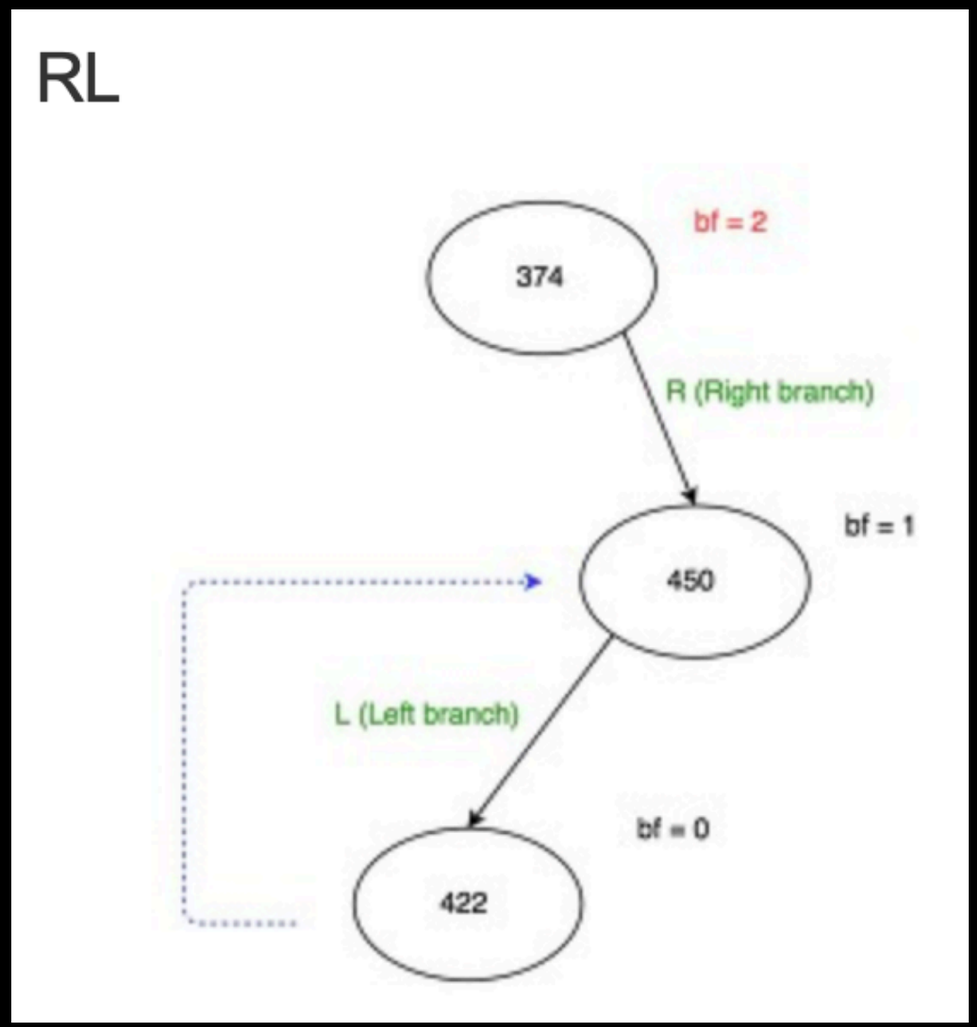
Unbalance

(3) LR타입 : ㄱ이 ㅅ의 왼쪽 서브트리의 오른쪽에 삽입되는 경우 (ㅅ 노드의 왼쪽노드의 오른쪽에 삽입)



Unbalance

(4) RL타입 : n이 A의 오른쪽 서브트리의 왼쪽에 삽입되는 경우(A 노드의 오른쪽노드의 왼쪽에 삽입)



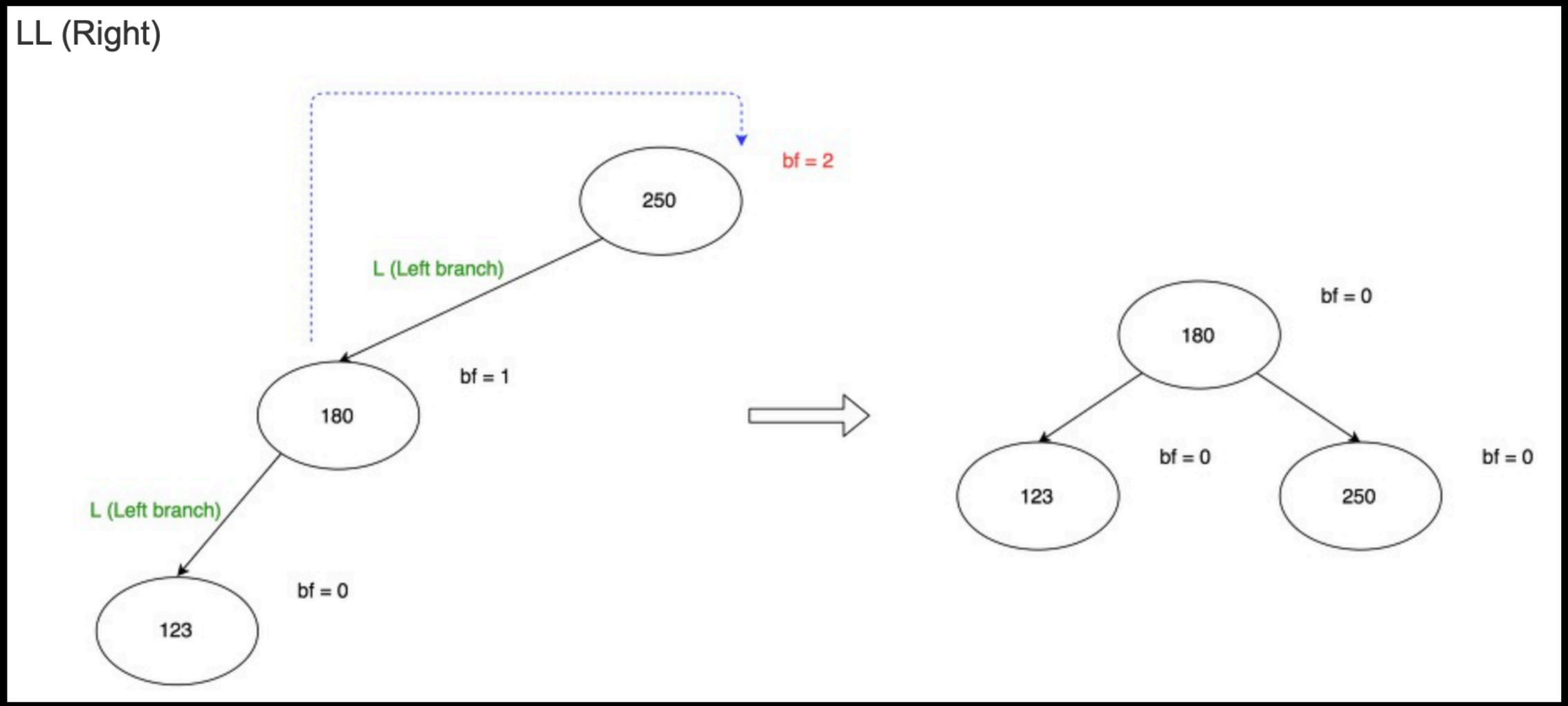
Rotations

위에서 Unbalance 한 형태를 살펴보았습니다. 각 유형의 불균형 한 트리를 AVL은 회전(Rotations)을 통하여 균형(Balance) 상태로 바꾸어 줍니다.

그렇다면 각 불균형 상태에서 균형 상태로 바꾸는 방법을 살펴보겠습니다.

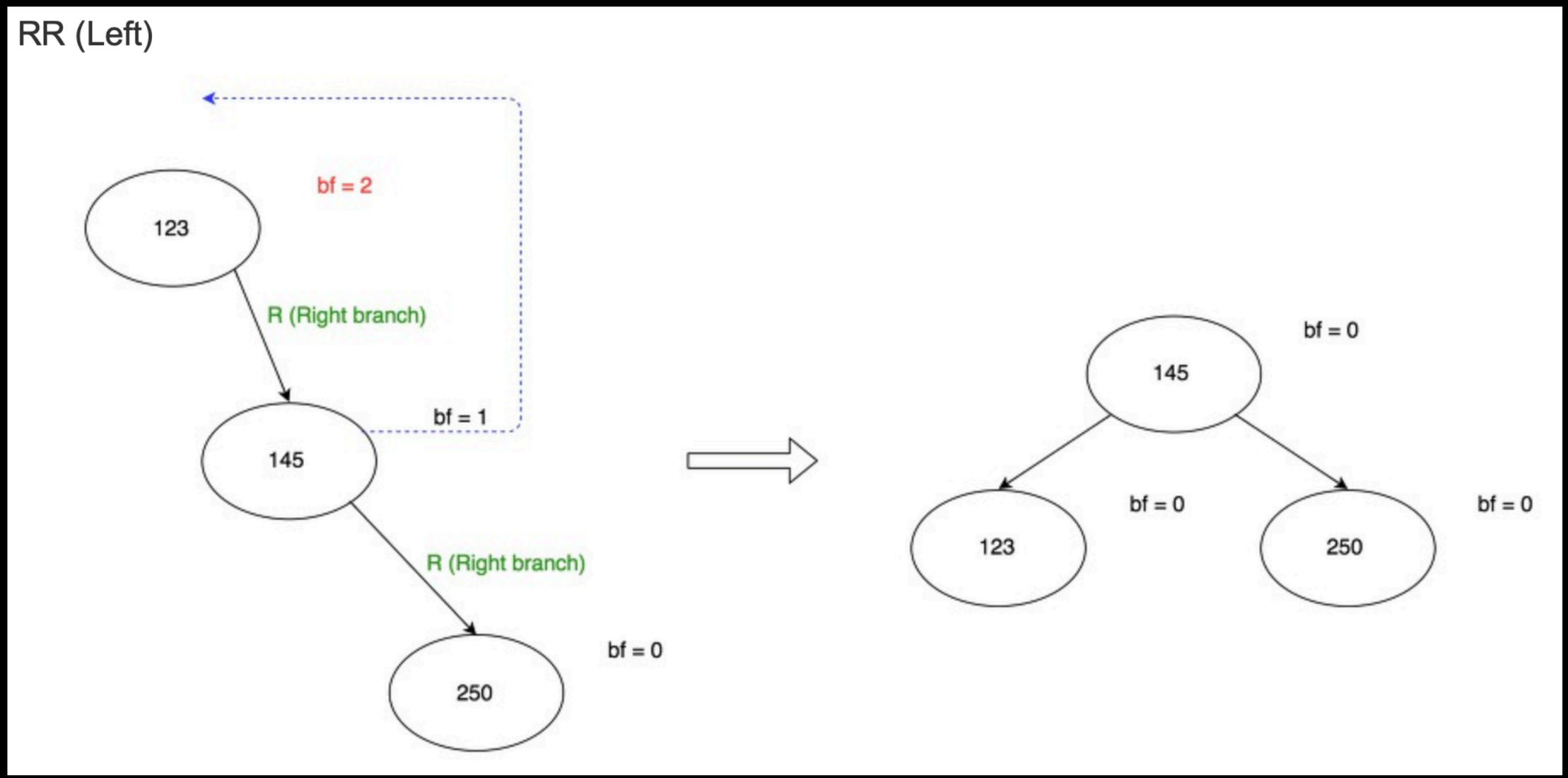
Rotations

(1) LL타입 : 트리를 오른쪽(LL 회전)으로 회전 시켜서 균형을 맞춘다.



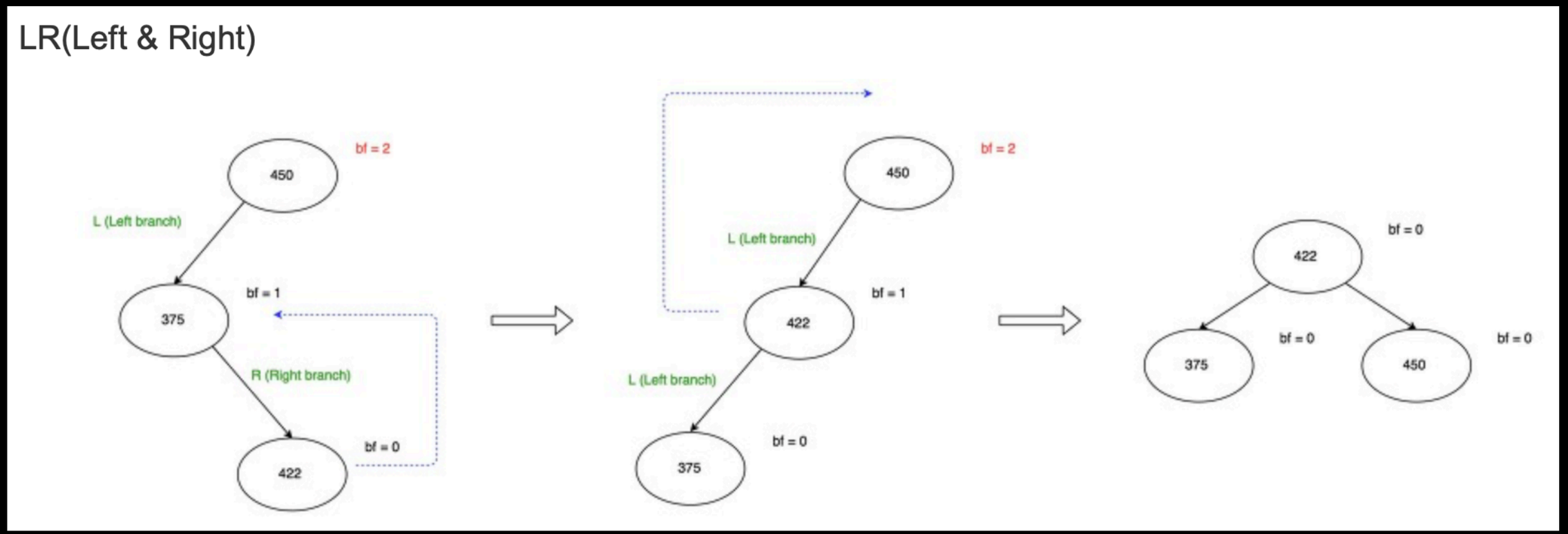
Rotations

(2) RR타입 : 트리를 왼쪽(RR 회전)으로 회전 시켜서 균형을 맞춘다.



Rotations

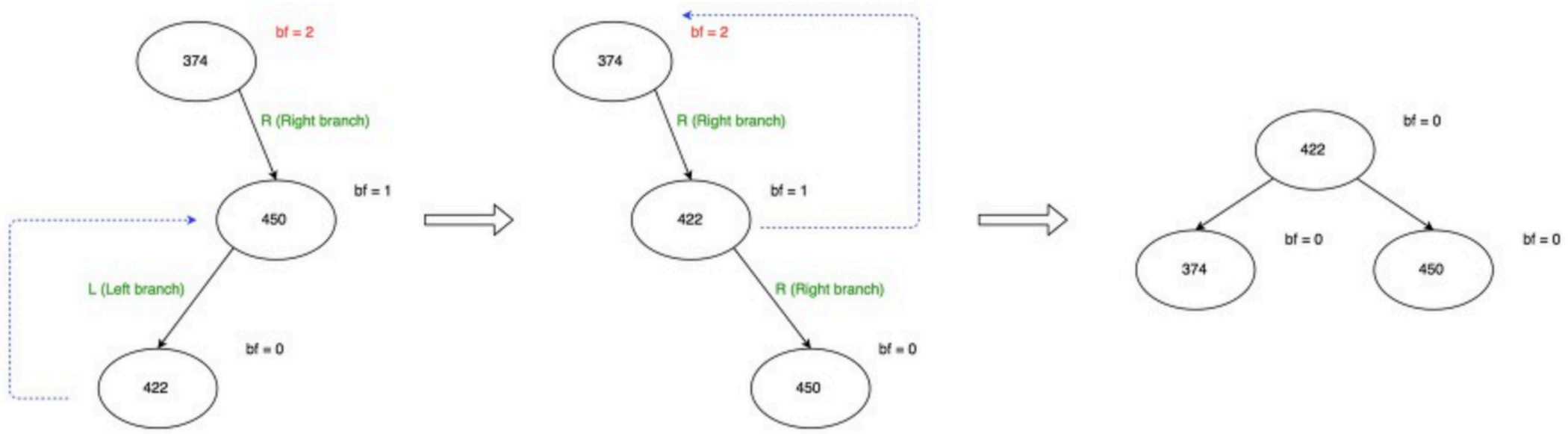
(3) LR타입 : 트리를 왼쪽과 오른쪽(LR 회전)으로 회전 시켜서 균형을 맞춘다.



Rotations

(4) RL타입 : 트리를 오른쪽과 왼쪽(RL 회전)으로 회전 시켜서 균형을 맞춘다.

RL (Right & Left)



Implementation

Swift를 활용하여 가장 기본적인 AVL를 구현해보겠습니다.
우선 기본적으로 필요한 객체 및 메소드는 아래와 같습니다.

필요한 객체

- AVLNode 객체 : Value, 자식 노드(왼쪽, 오른쪽), Height, BalanceFactor 값 등을 소유
- AVLTree 객체 : AVL 트리를 관리하고 수행하는 클래스

기본 메소드

- insert : 노드를 추가하는 함수
- remove : 특정 노드를 삭제하는 함수
- balanced : 현재 트리의 균형 상태를 체크하고 균형 상태를 맞추는 함수
- leftRotate : 왼쪽으로 트리를 돌리는 함수
- rightRotate : 오른쪽으로 트리를 돌리는 함수
- leftRightRotate : 왼쪽과 오른쪽으로 트리를 돌리는 함수
- rightLeftRotate : 오른쪽과 왼쪽으로 트리를 돌리는 함수

Implementation

```
public class AVLNode<Element> {
    public var value: Element
    public var leftChild: AVLNode?
    public var rightChild: AVLNode?

    public var height = 0

    public var balanceFactor: Int {
        return leftHeight - rightHeight
    }

    public var leftHeight: Int { return leftChild?.height ?? -1 }
    public var rightHeight: Int { return rightChild?.height ?? -1 }

    public init(value: Element) {
        self.value = value
    }
}
```

Implementation

```
extension AVLNode: CustomStringConvertible {
    var min: AVLNode {
        return leftChild?.min ?? self
    }

    public var description: String {
        return diagram(for: self)
    }

    private func diagram(for node: AVLNode?,
                         _ top: String = "",
                         _ root: String = "",
                         _ bottom: String = "") -> String {
        guard let node = node else {
            return root + "nil\n"
        }
        if node.leftChild == nil && node.rightChild == nil {
            return root + "\(node.value)\n"
        }
        return diagram(for: node.rightChild, top + " ", top + "┌──", top + "│ ")
            + root + "\(node.value)\n"
            + diagram(for: node.leftChild, bottom + "│ ", bottom + "└──", bottom
+ " ")
    }
}
```

Implementation

```
public struct AVLTree<Element: Comparable> {
    public private(set) var root: AVLNode<Element>?

    public init() {}

    public mutating func insert(_ value: Element) {
        root = insert(from: root, value: value)
    }

    private func insert(from node: AVLNode<Element>?, value: Element) ->
AVLNode<Element> {
        guard let node = node else {
            return AVLNode(value: value)
        }

        if value < node.value {
            node.leftChild = insert(from: node.leftChild, value: value)
        } else {
            node.rightChild = insert(from: node.rightChild, value: value)
        }

        let balancedNode = balanced(node)
        balancedNode.height = max(balancedNode.leftHeight, balancedNode.rightHeight) + 1

        return balancedNode
    }
}
```

Implementation

```
extension AVLTree {
    public mutating func remove(_ value: Element) {
        root = remove(node: root, value: value)
    }

    private func remove(node: AVLNode<Element>?, value: Element) -> AVLNode<Element>? {
        guard let node = node else {
            return nil
        }

        if value == node.value {
            if node.leftChild == nil && node.rightChild == nil {
                return nil
            }
            if node.leftChild == nil {
                return node.rightChild
            }
            if node.rightChild == nil {
                return node.leftChild
            }
            node.value = node.rightChild!.min.value
            node.rightChild = remove(node: node.rightChild, value: node.value)
        } else if value < node.value {
            node.leftChild = remove(node: node.leftChild, value: value)
        } else {
            node.rightChild = remove(node: node.rightChild, value: value)
        }

        let balanceNode = balanced(node)
        balanceNode.height = max(balanceNode.leftHeight, balanceNode.rightHeight) + 1
        return balanceNode
    }
}
```

Implementation

```
extension AVLTree {
    private func leftRotate(_ node: AVLNode<Element>) -> AVLNode<Element> {
        let pivot = node.rightChild!

        node.rightChild = pivot.leftChild
        pivot.leftChild = node

        node.height = max(node.leftHeight, node.rightHeight) + 1
        pivot.height = max(pivot.leftHeight, pivot.rightHeight) + 1

        return pivot
    }

    private func rightRotate(_ node: AVLNode<Element>) -> AVLNode<Element> {
        let pivot = node.leftChild!

        node.leftChild = pivot.rightChild
        pivot.rightChild = node

        node.height = max(node.leftHeight, node.rightHeight) + 1
        pivot.height = max(pivot.leftHeight, pivot.rightHeight) + 1

        return pivot
    }
}
```

Implementation

```
extension AVLTree {
    private func rightLeftRotate(_ node: AVLNode<Element>) -> AVLNode<Element> {
        guard let rightChild = node.rightChild else { return node }

        node.rightChild = rightRotate(rightChild)

        return leftRotate(node)
    }
    private func leftRightRotate(_ node: AVLNode<Element>) -> AVLNode<Element> {
        guard let leftChild = node.leftChild else { return node}

        node.leftChild = leftRotate(leftChild)

        return rightRotate(node)
    }
}
```

Implementation

```
extension AVLTree {
    private func balanced(_ node: AVLNode<Element>) -> AVLNode<Element> {
        switch node.balanceFactor {

        case 2:
            if let leftChild = node.leftChild, leftChild.balanceFactor == -1 {
                return leftRightRotate(node)
            } else {
                return rightRotate(node)
            }

        case -2:
            if let rightChild = node.rightChild, rightChild.balanceFactor == 1 {
                return rightLeftRotate(node)
            } else {
                return leftRotate(node)
            }

        default:
            return node
        }
    }
}
```

Implementation

```
extension AVLTree: CustomStringConvertible {
    public var description: String {
        guard let root = root else { return "empty tree" }
        return String(describing: root)
    }
}
```

Implementation

```
var tree = AVLTree<Int>()

tree.insert(1)
tree.insert(2)
tree.insert(3)
tree.insert(4)
tree.insert(5)

print(tree)
//   ┌──5
//  ┌──4
//  │ └──3
// 2
// └──1

var treeRemove = tree
treeRemove.remove(4)

print(treeRemove)
//   ┌──nil
//  ┌──5
//  │ └──3
// 2
// └──1
```

References

[1] [스위프트 : 자료구조] AVL Tree: 자가 균형 트리 : https://the-brain-of-sic2.tistory.com/28

[2] 자료구조 :: 탐색(3) "균형 이진 탐색 트리 - AVL 트리" : http://egloos.zum.com/printf/v/913998

[3] 자료구조(AVL 트리(tree)) : https://www.zerocho.com/category/Algorithm/post/583cacb648a7340018ac73f1

[4] 자료구조 AVL 트리(AVL tree) : https://blog.naver.com/PostView.nhn?blogId=beaqon&logNo=221265761300&categoryNo=121&parentCategoryNo=0

[5] AVL tree : https://ratsgo.github.io/data%20structure&algorithm/2017/10/27/avltree/

References

[6] AVL 트리 : https://ko.wikipedia.org/wiki/AVL_트리

[7] 숨 쉬는 것만큼 쉬운 '트리' 만들기 - AVL tree편 : https://m.blog.naver.com/PostView.nhn?blogId=dhdh6190&logNo=221062784111&proxyReferer=https%2F%2Fwww.google.com%2F

[8] AVL Tree : https://doublesprogramming.tistory.com/237

[9] 자료구조 AVL 트리와 구현 : https://gurumee92.tistory.com/146

[10] AVL 트리 (AVL Tree) : https://casterian.net/archives/217

Thank you!