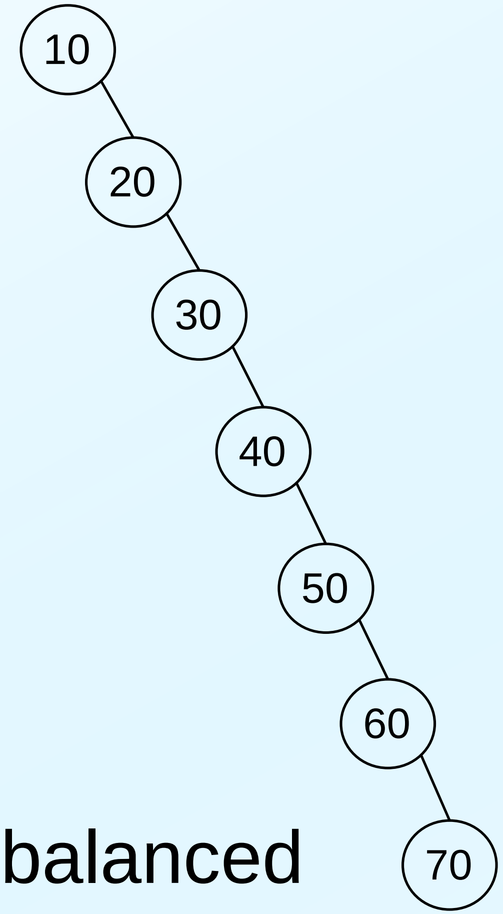
균형검색트리 Balanced Search Tree

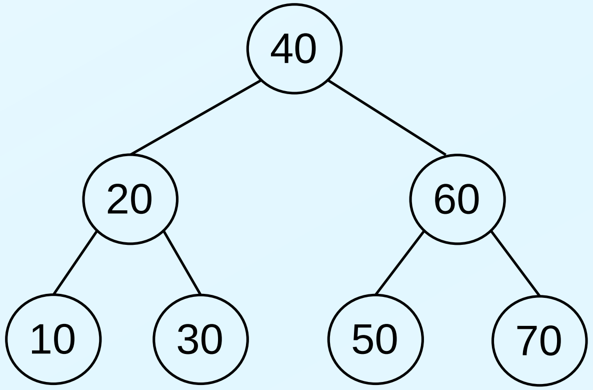
1. 균형이 왜 필요한가?
2. AVL Tree
3. Red-Black Tree
4. B-Tree

1. 균형이 왜 필요한가?

균형이 왜 필요한가?



Worst balanced



Best balanced

Balanced Search Trees

- **Operations in a search tree depend heavily on the tree height**
  - Need balanced search trees
- **Balanced binary search trees**
  - Guarantees $O(\log n)$-time search, insertion, and deletion
  - AVL tree, red-black tree
- **Balanced k-ary search trees**
  - Guarantees $O(\log n)$-time search, insertion, and deletion
  - 2-3 tree, 2-3-4 tree, B-trees, …

2. AVL Tree

AVL Tree의 정의

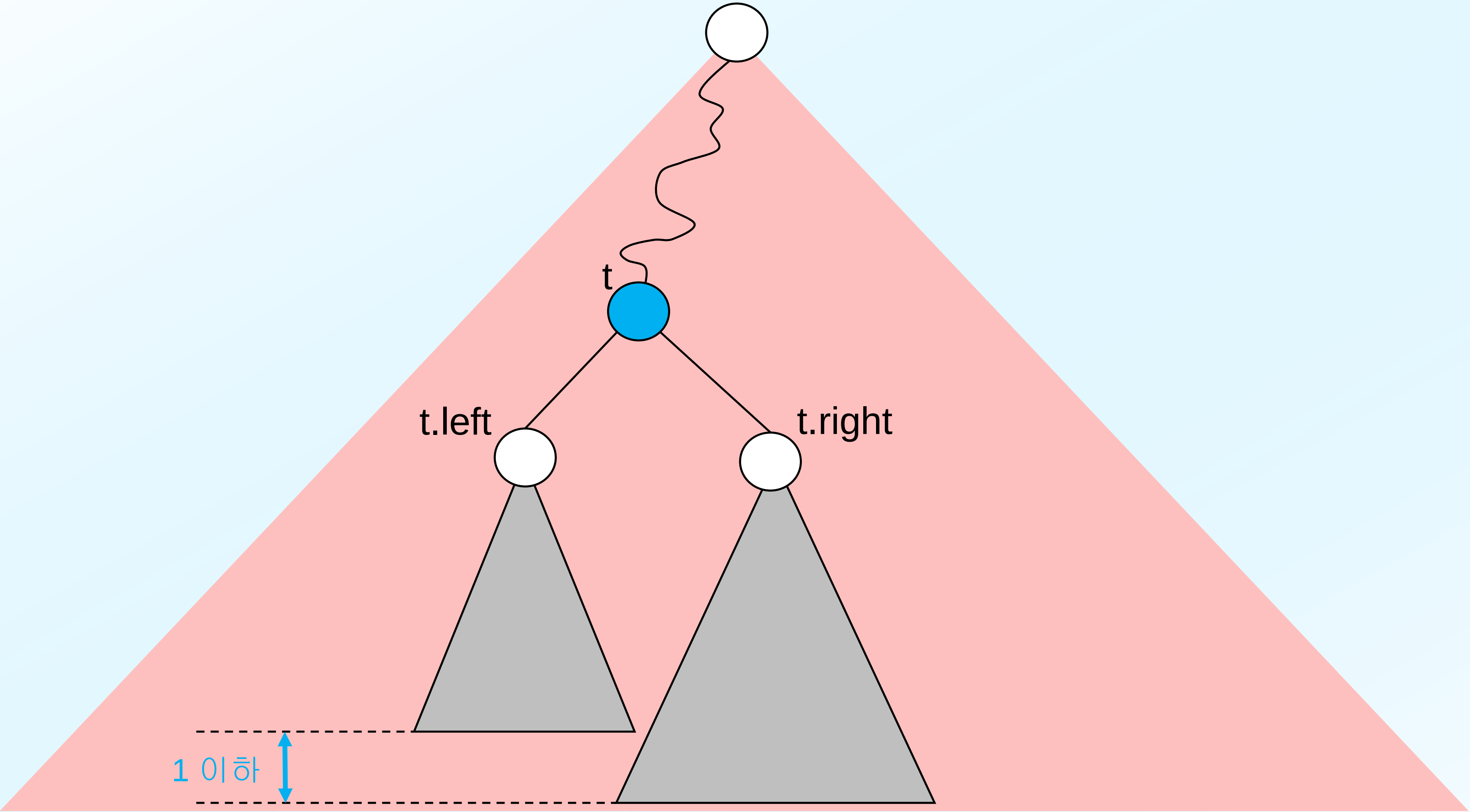
- Devised by Adelson-Velskii and Landis

- A balanced binary search tree

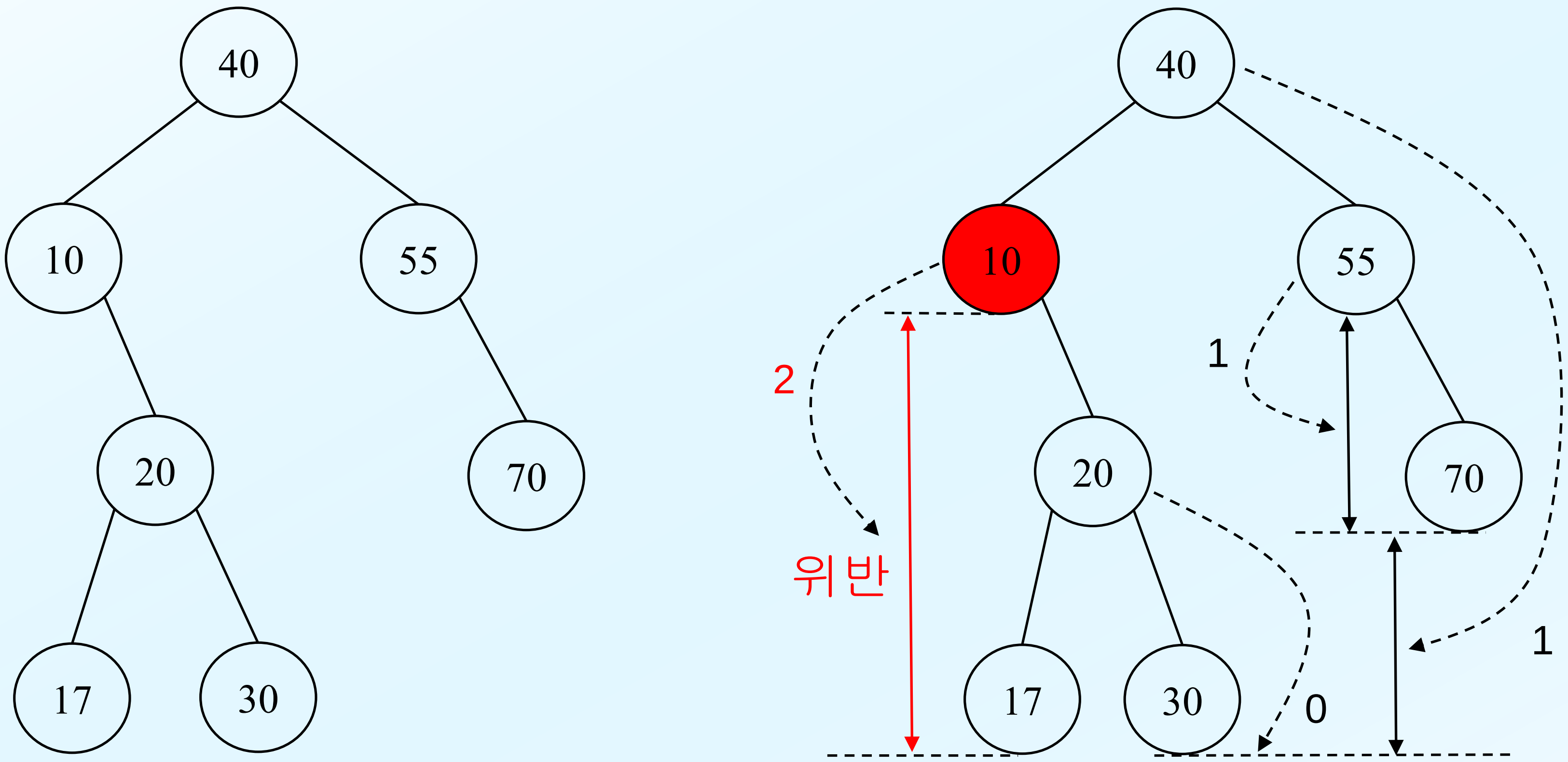
  such that

  the heights(depths) of the left and right subtrees of any node

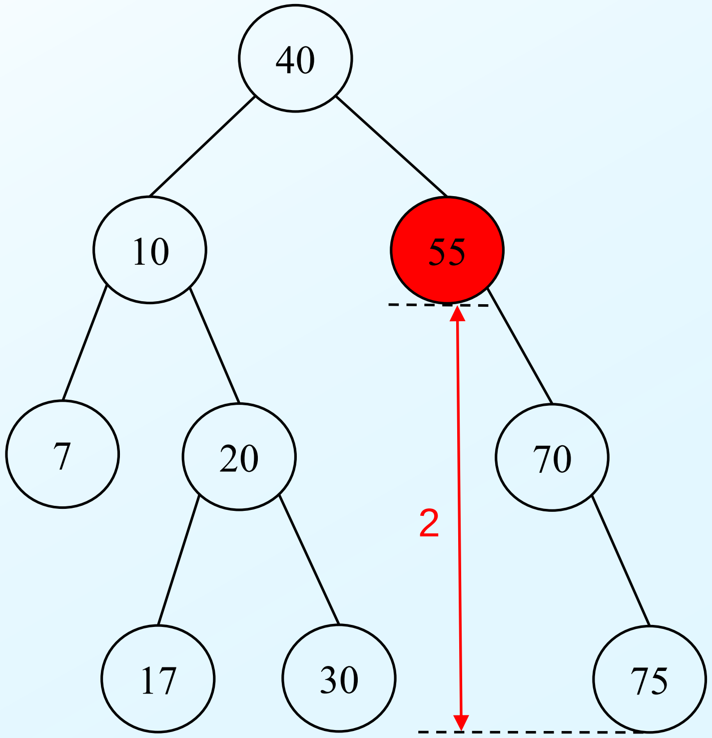
  differ by at most 1



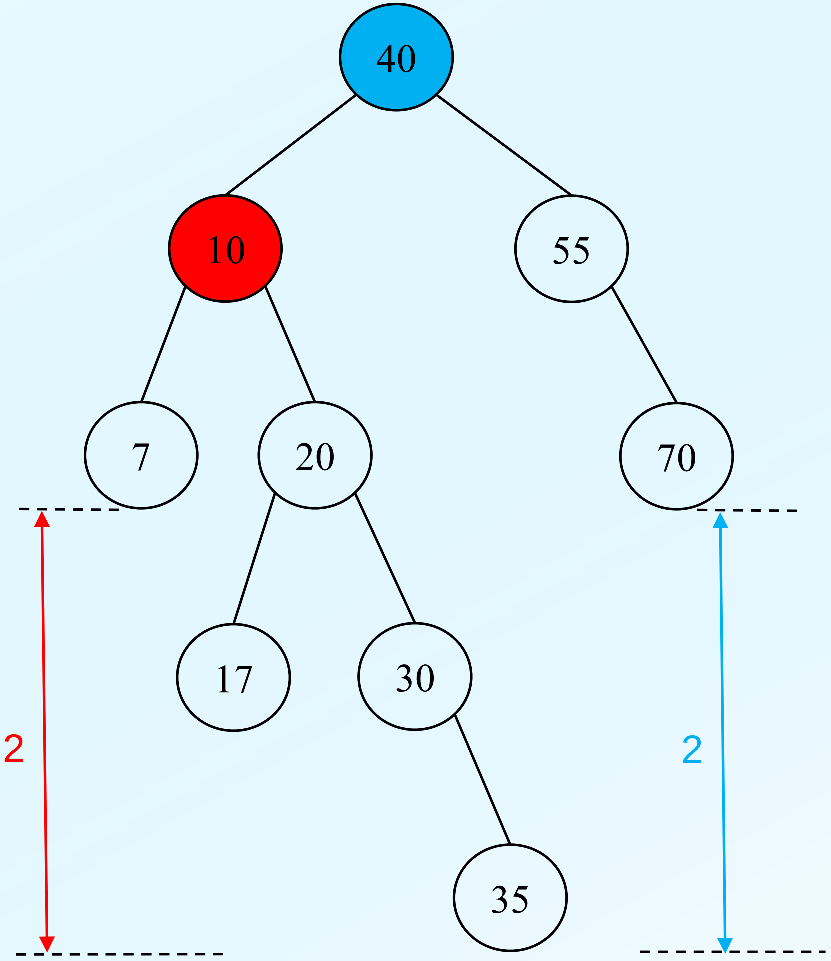
AVL Tree



AVL Tree가 아닌 예



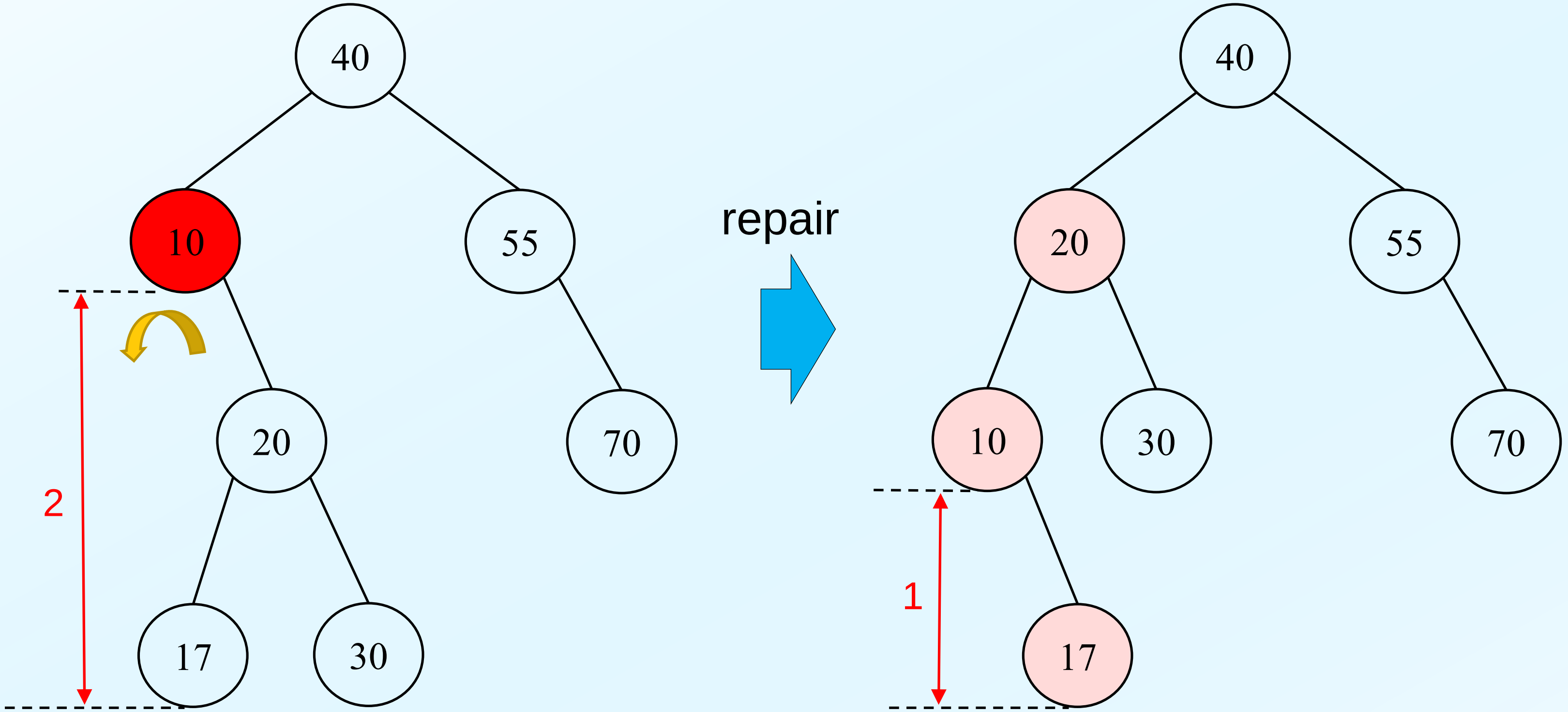
AVL Tree가 아니다



AVL Tree가 아니다

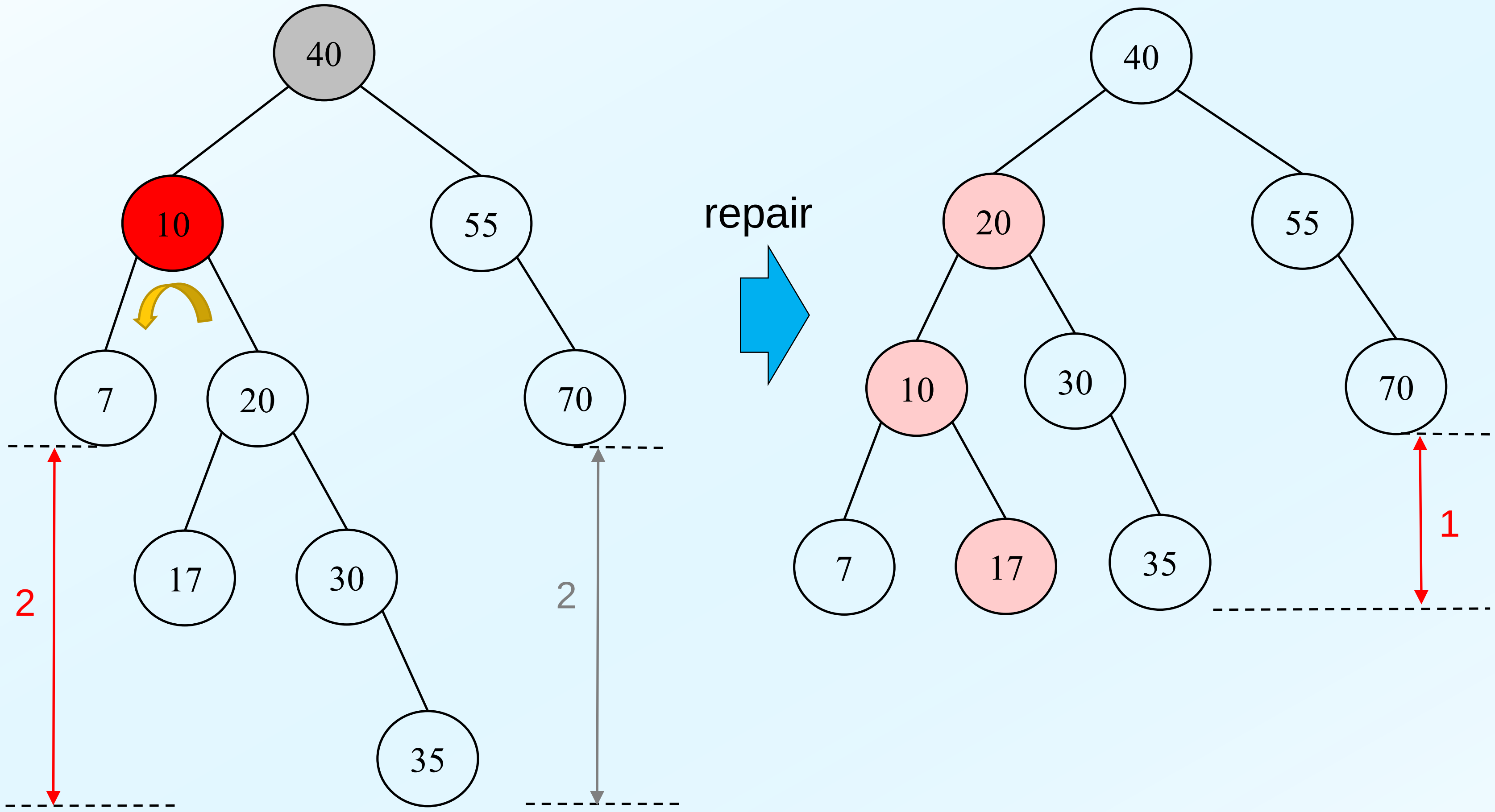
수선

수선의 예



AVL Tree가 아닌 예 좌회전 해서 AVL Tree로 수선됨

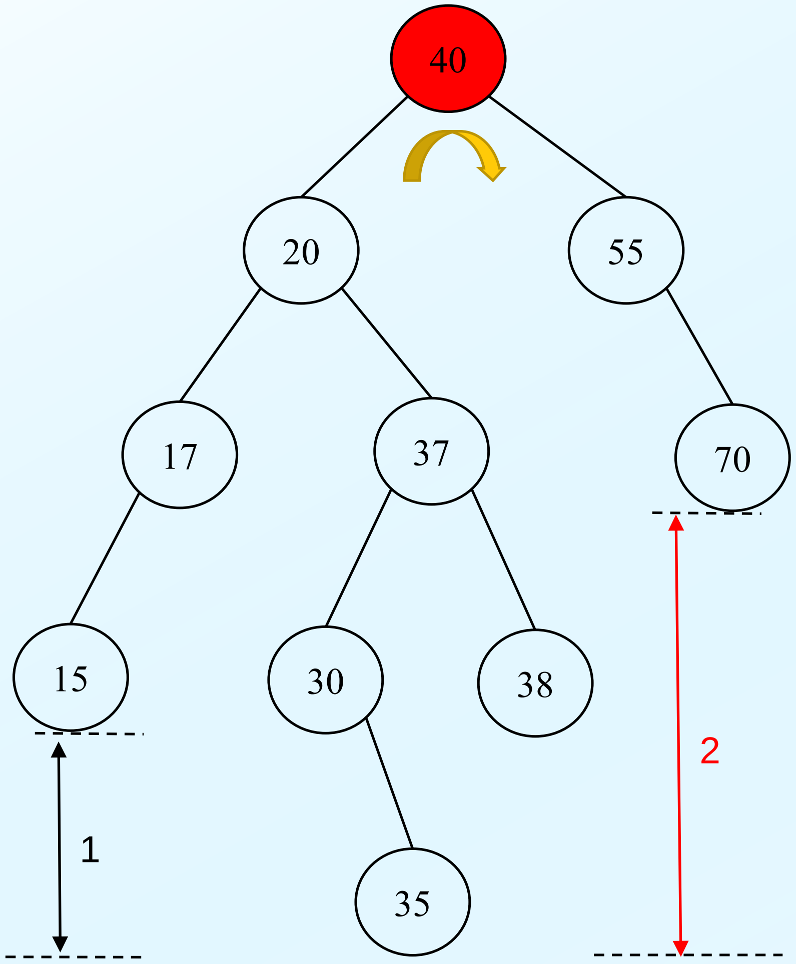
수선의 예



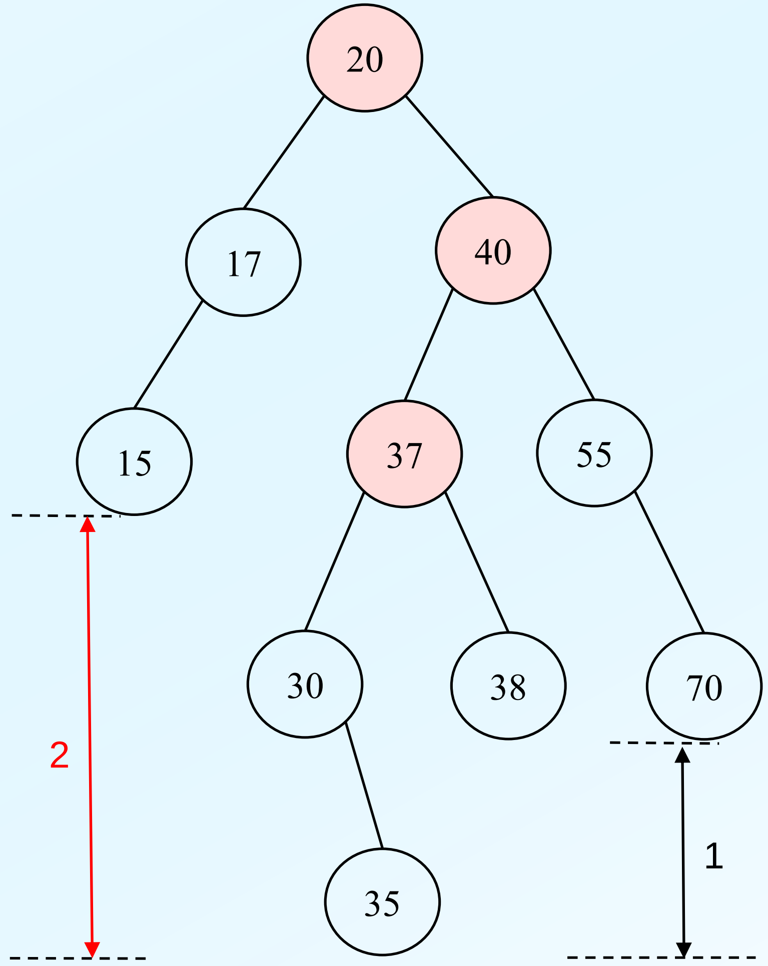
AVL Tree가 아닌 예

좌회전 해서 AVL Tree로 수선됨

수선이 안되는 예

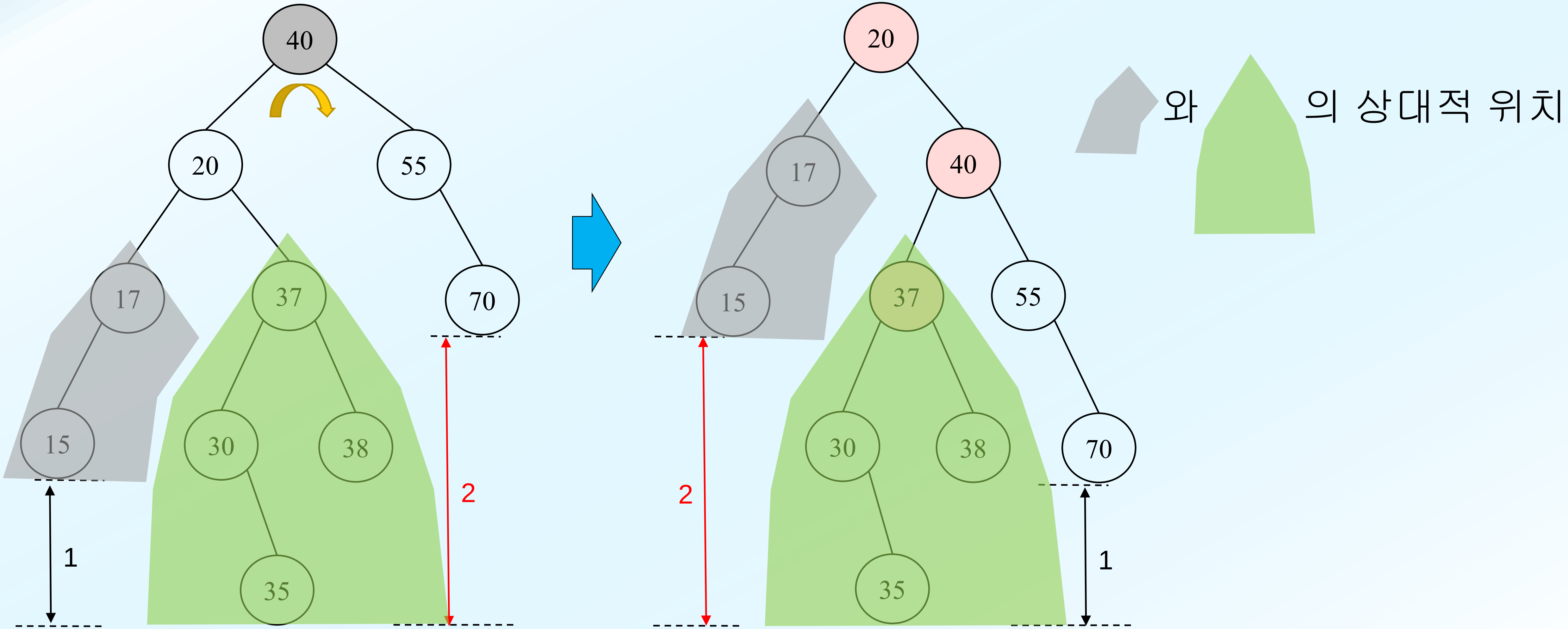


AVL Tree가 아닌 예

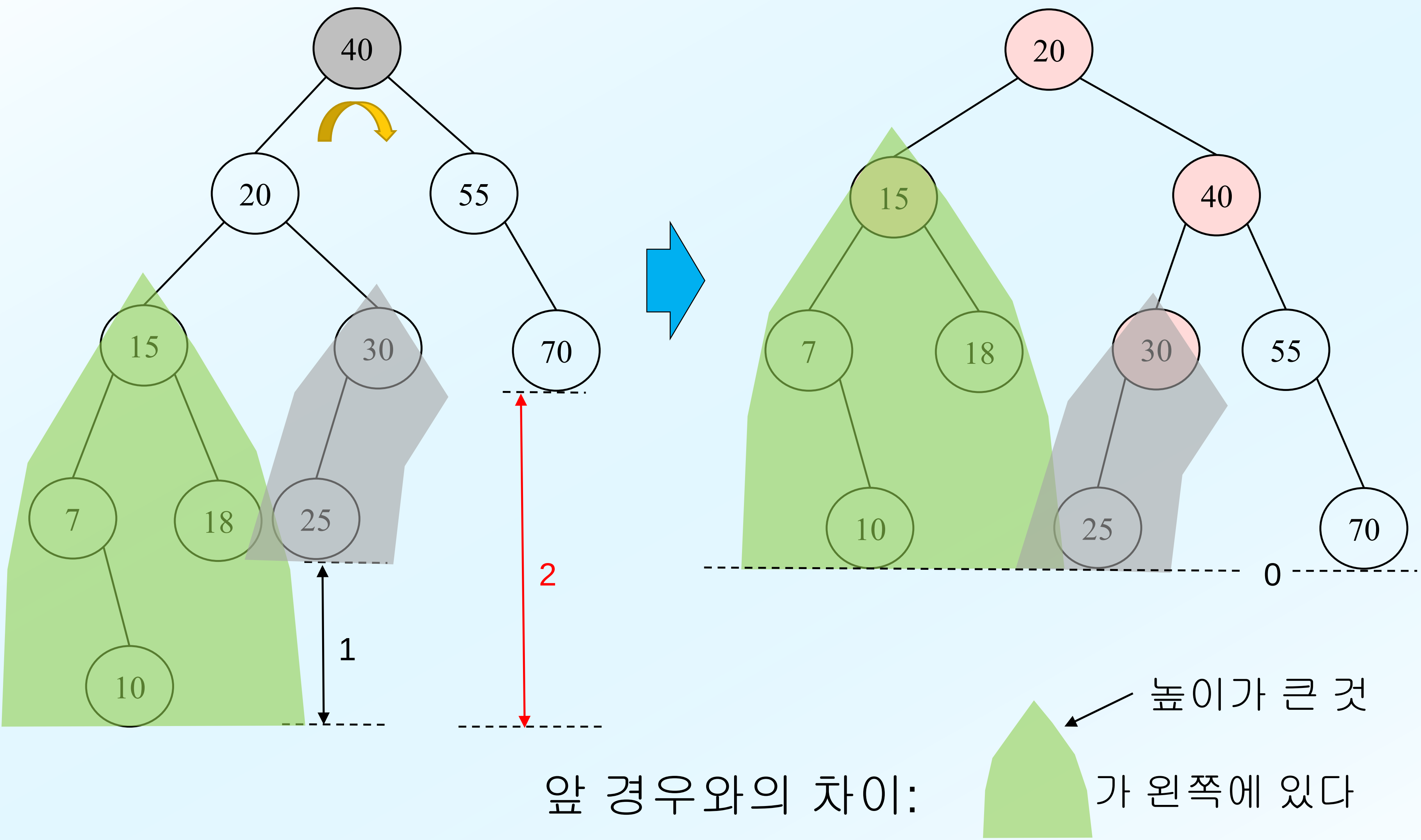


수선이 안된다

수선이 안된 이유


40
20
55
17
37
70
15
30
38
35
1
2
20
40
17
15
37
55
30
38
70
35
2
1
와
의 상대적 위치

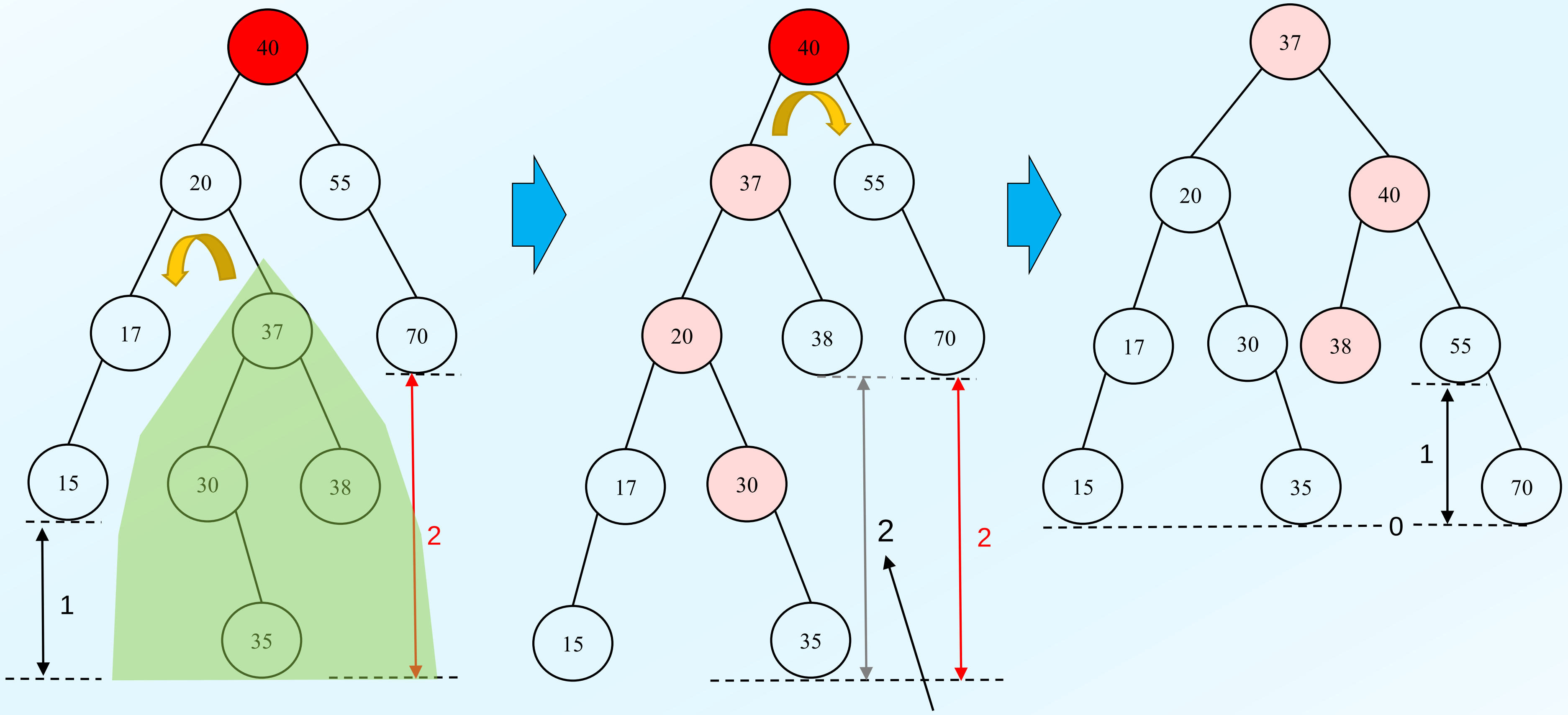
이 경우는 수선이 된다



앞 경우와의 차이:

표준화

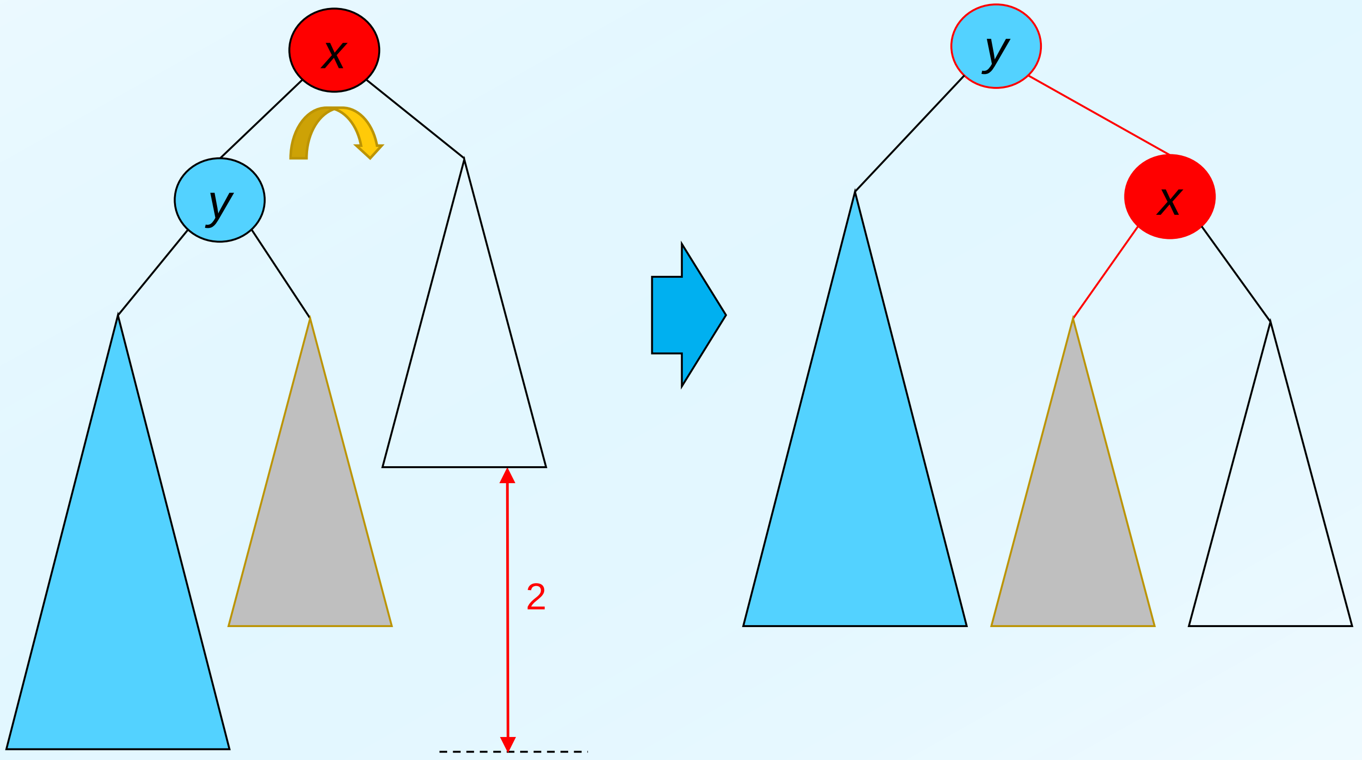
왼쪽이 더 높게 만든 다음 우회전



일시적 불균형(문제 없음)

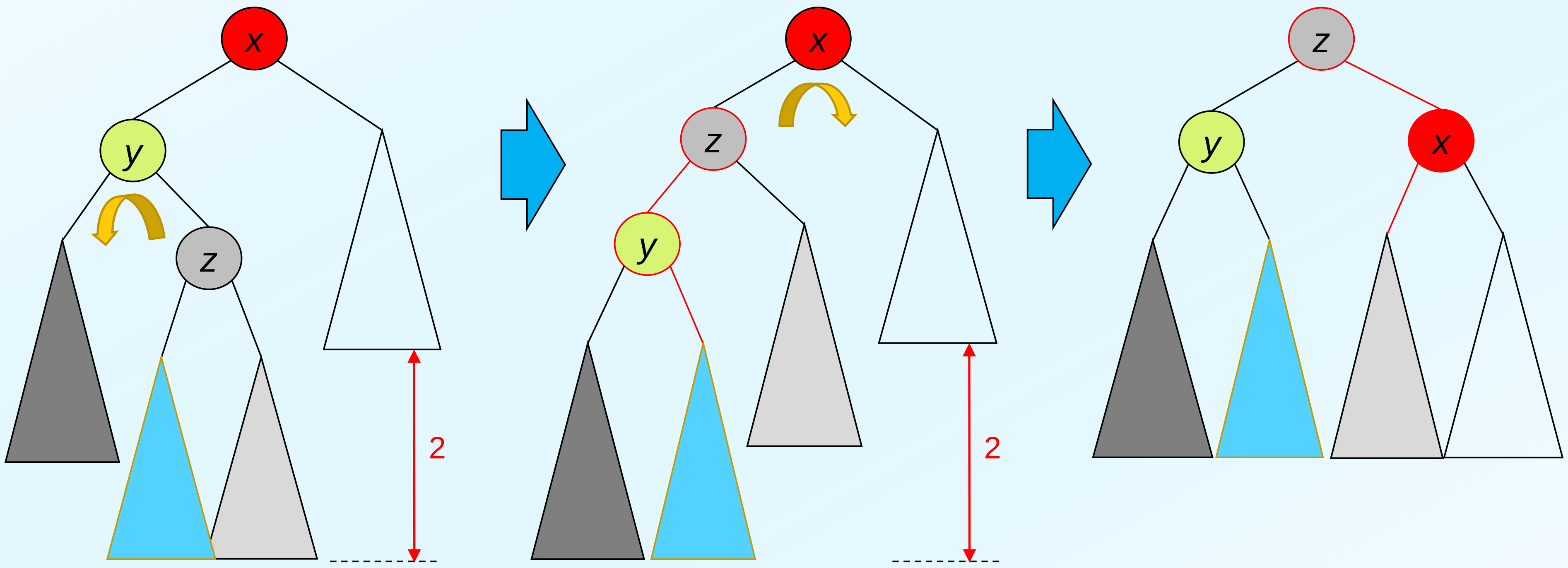
Four Types of Repairs

1. Type LL: Right rotation



Four Types of Repairs

2. Type LR: Left rotation then right rotation
(conversion to type LL)

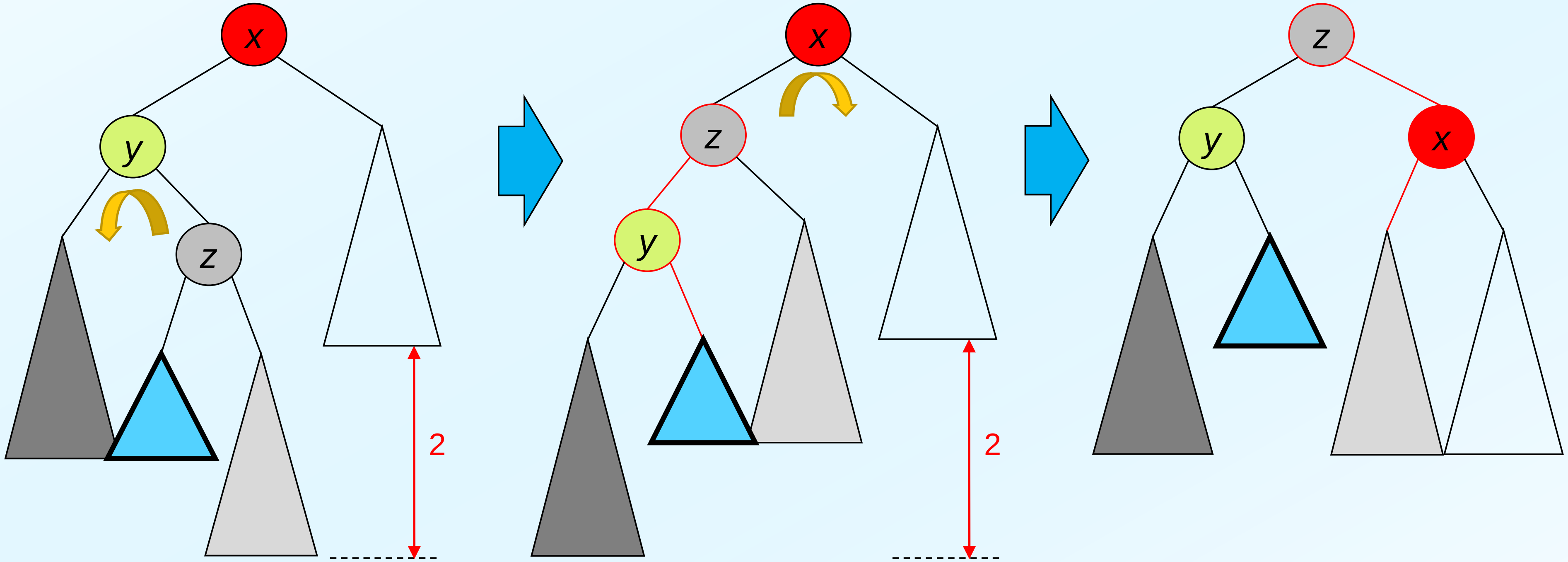


Type LR

Type LL

Four Types of Repairs

Another Instance of Type LR

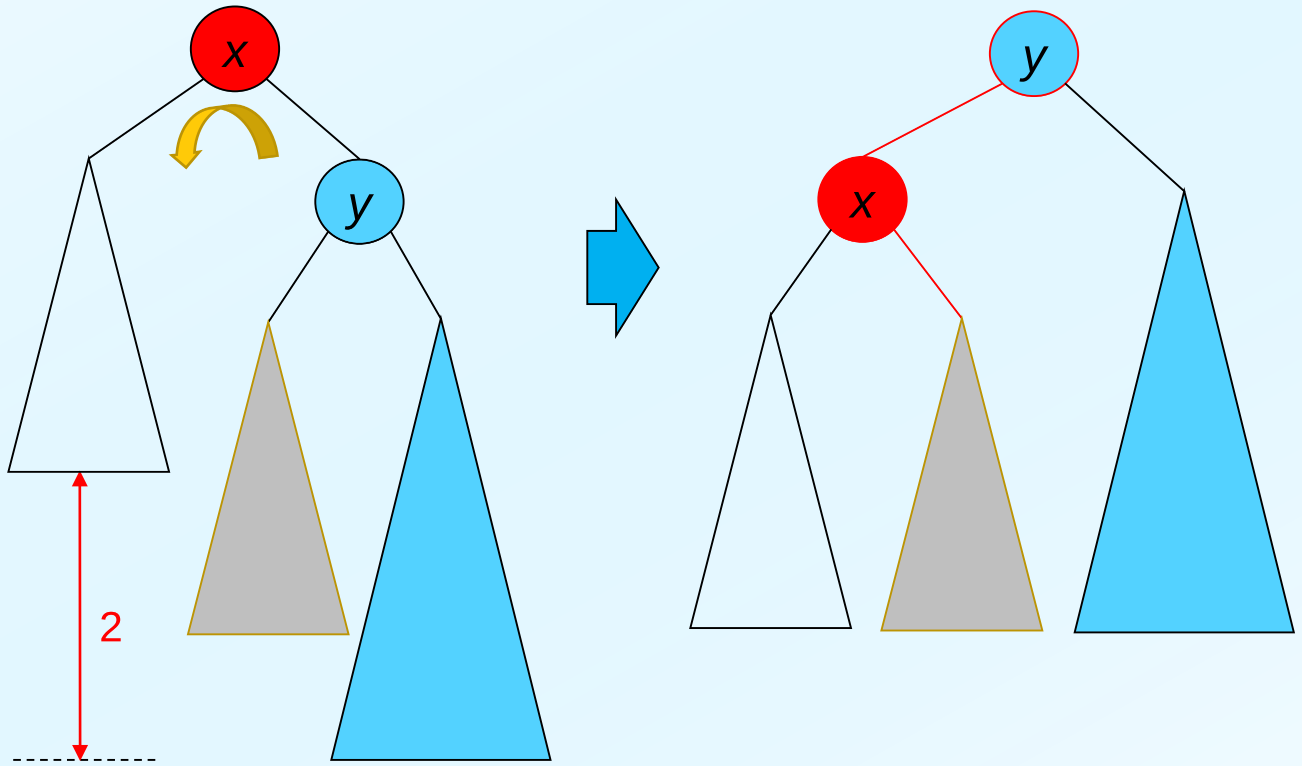


Type LR

Type LL

Four Types of Repairs

3. Type RR: Left rotation

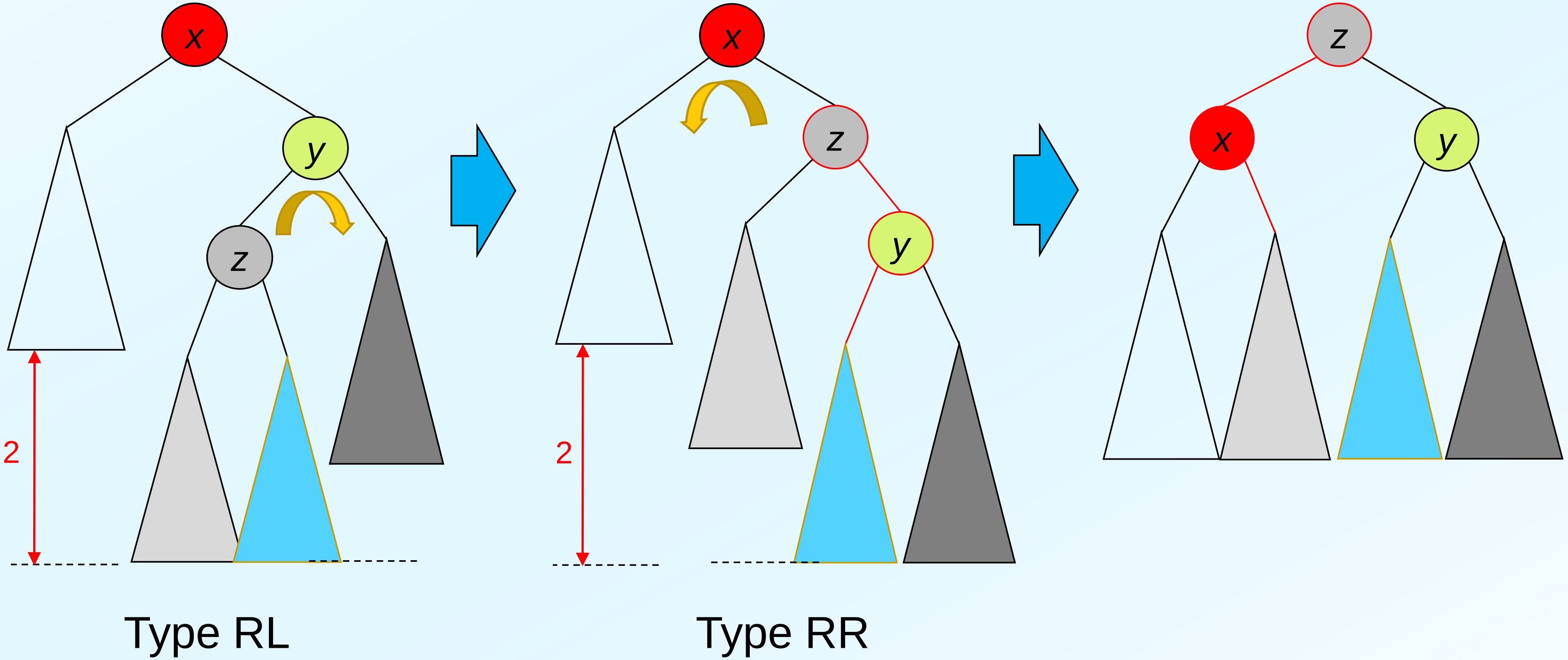


Four Types of Repairs

4. Type RL: <u>Right rotation</u> then left rotation

(conversion to type RR)

* LL과 RR, LR과 RL은 각각 symmetric



Balancing AVL Tree

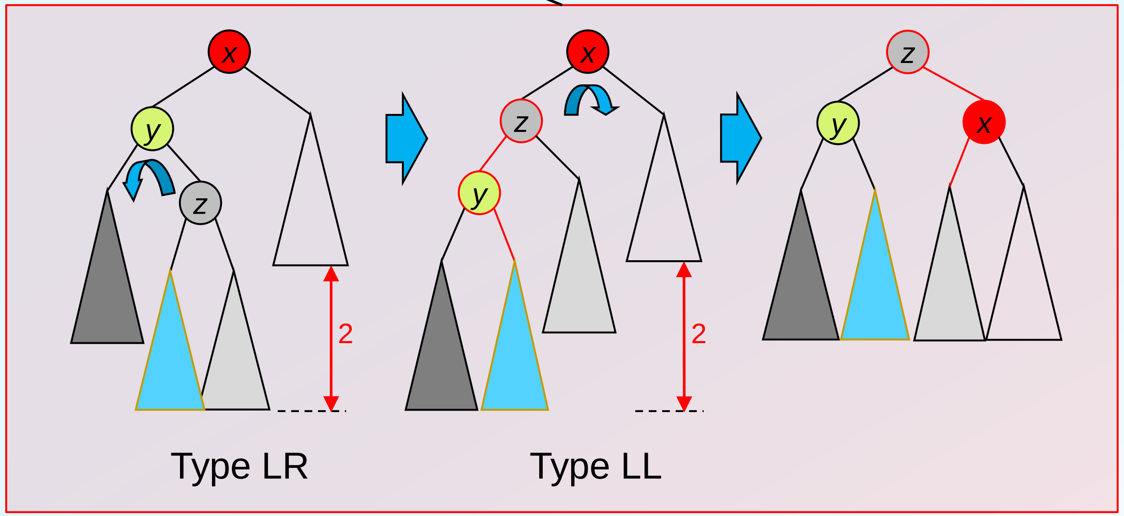
```
balanceAVL(t, type):
    switch type:
        case LL: rightRotate(t)
        case LR: leftRotate(t.left)
                 balanceAVL(t, LL)
        case RR: leftRotate(t)
        case RL: rightRotate(t.right)
                 balanceAVL(t, RR)
```

Equivalently,

```
balanceAVL(t, type):
    switch type:
        case LL: rightRotate(t)
        case LR: leftRotate(t.left)
                 rightRotate(t)
        case RR: leftRotate(t)
        case RL: rightRotate(t.right)
                 leftRotate(t)
```

Balancing AVL Tree

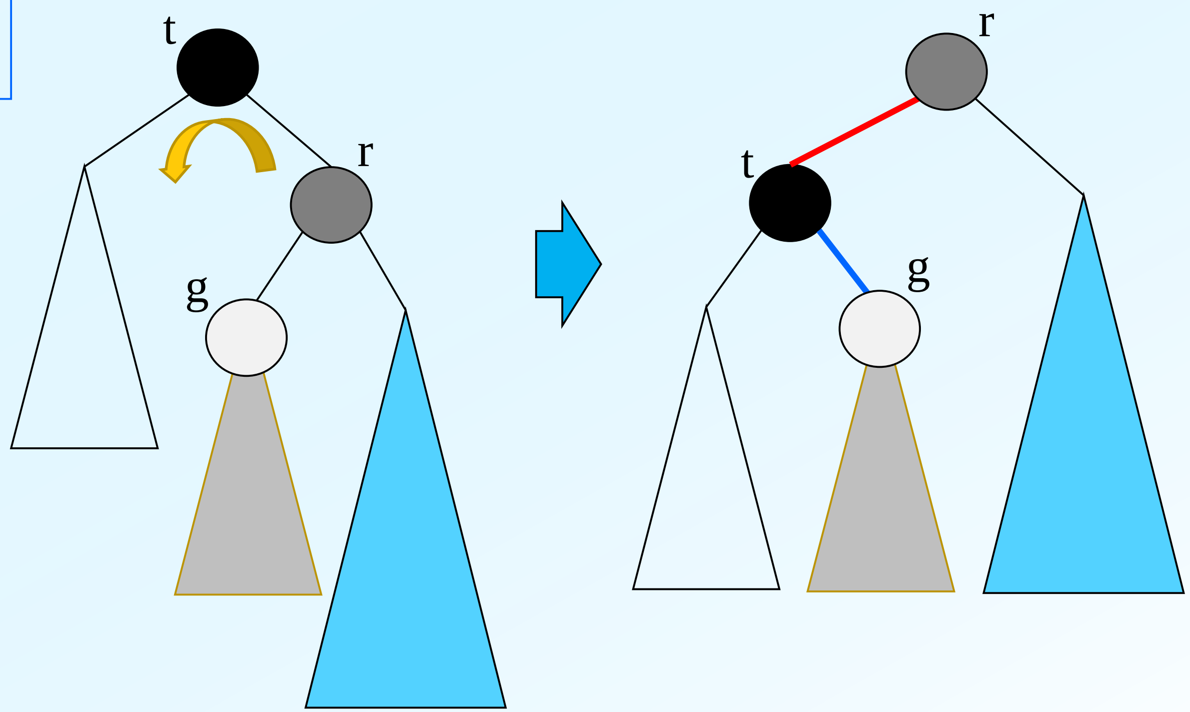
```
balanceAVL(t, type):
    switch type:
        case LL: rightRotate(t)
        case LR: leftRotate(t.left)
                 balanceAVL(t, LL)
        case RR: leftRotate(t)
        case RL: rightRotate(t.right)
                 balanceAVL(t, RR)
```



Balancing AVL Tree

Left Rotation

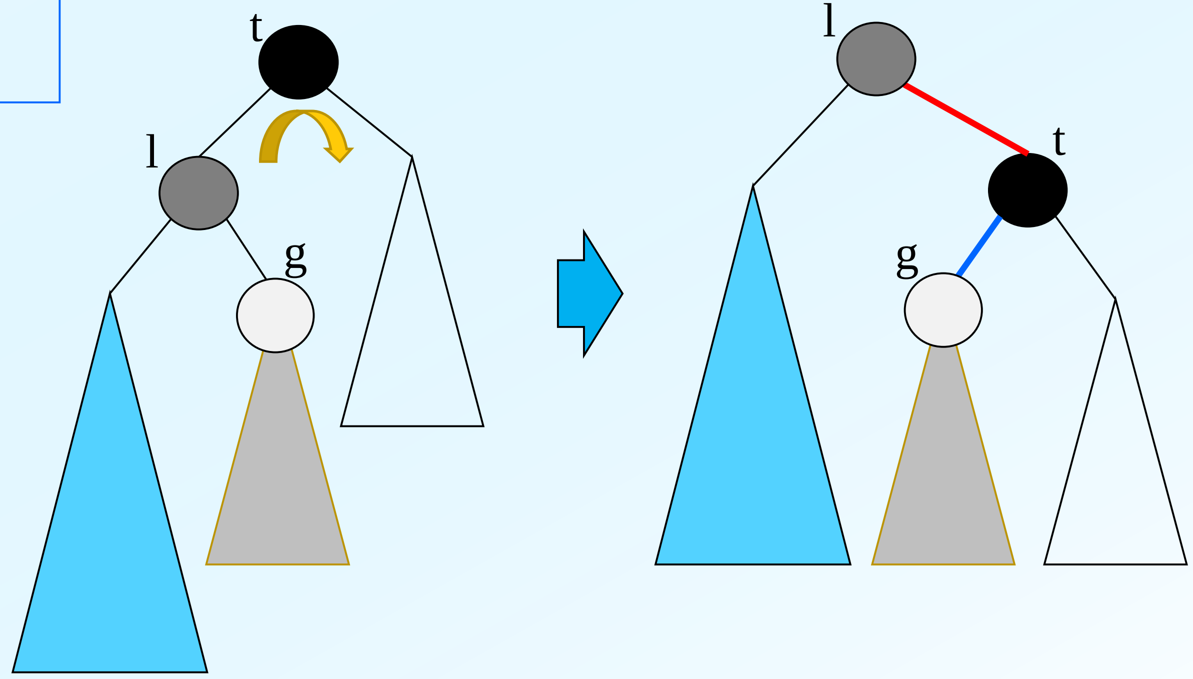
r ← t.right
g ← r.left
r.left ← t
t.right ← g
t.height ← max(t.right.height, t.left.height) + 1
r.height ← max(r.right.height, r.left.height) + 1



Balancing AVL Tree

Right Rotation

l ← t.left
g ← l.right
l.right ← t
t.left ← g
t.height ← max(t.right.height, t.left.height) + 1
l.height ← max(l.right.height, l.left.height) + 1



Balancing AVL Tree

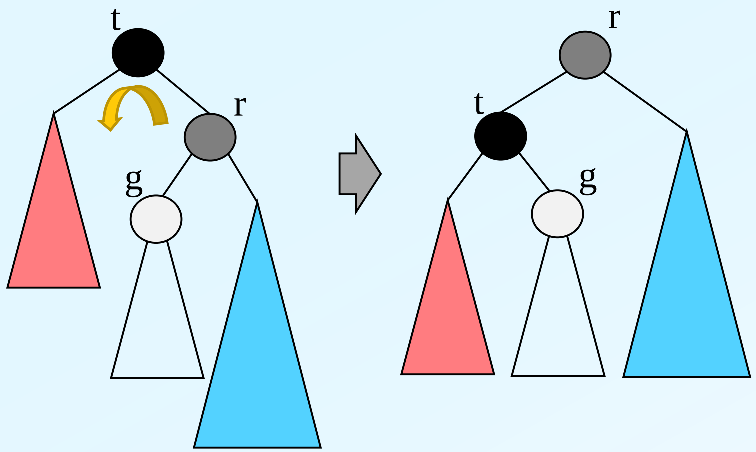
Left Rotation

r ← t.right
g ← r.left
r.left ← t
t.right ← g
t.height ← max(t.right.height, t.left.height) + 1
r.height ← max(r.right.height, r.left.height) + 1

이 코드는 문제를 일으킬 수 있다

t.right, t.left, r.right가 null이면 에러 발생

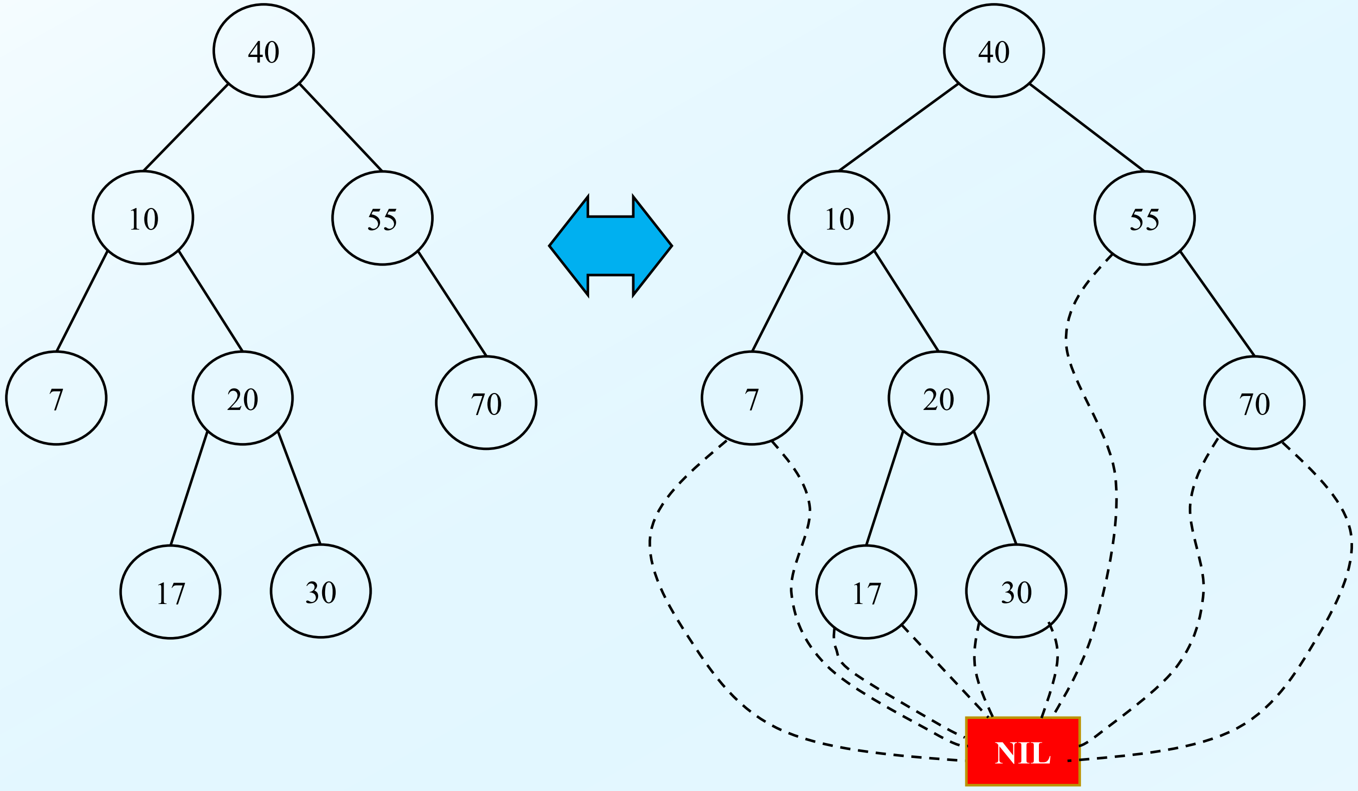
통상적인 방법으로 해결하려면
코드가 지저분해진다



An Effective Skill: Sentinel

… AVLNode **NIL** = **new** AVLNode(**null**, **null**, **null**, 0); // sentinel

height 0

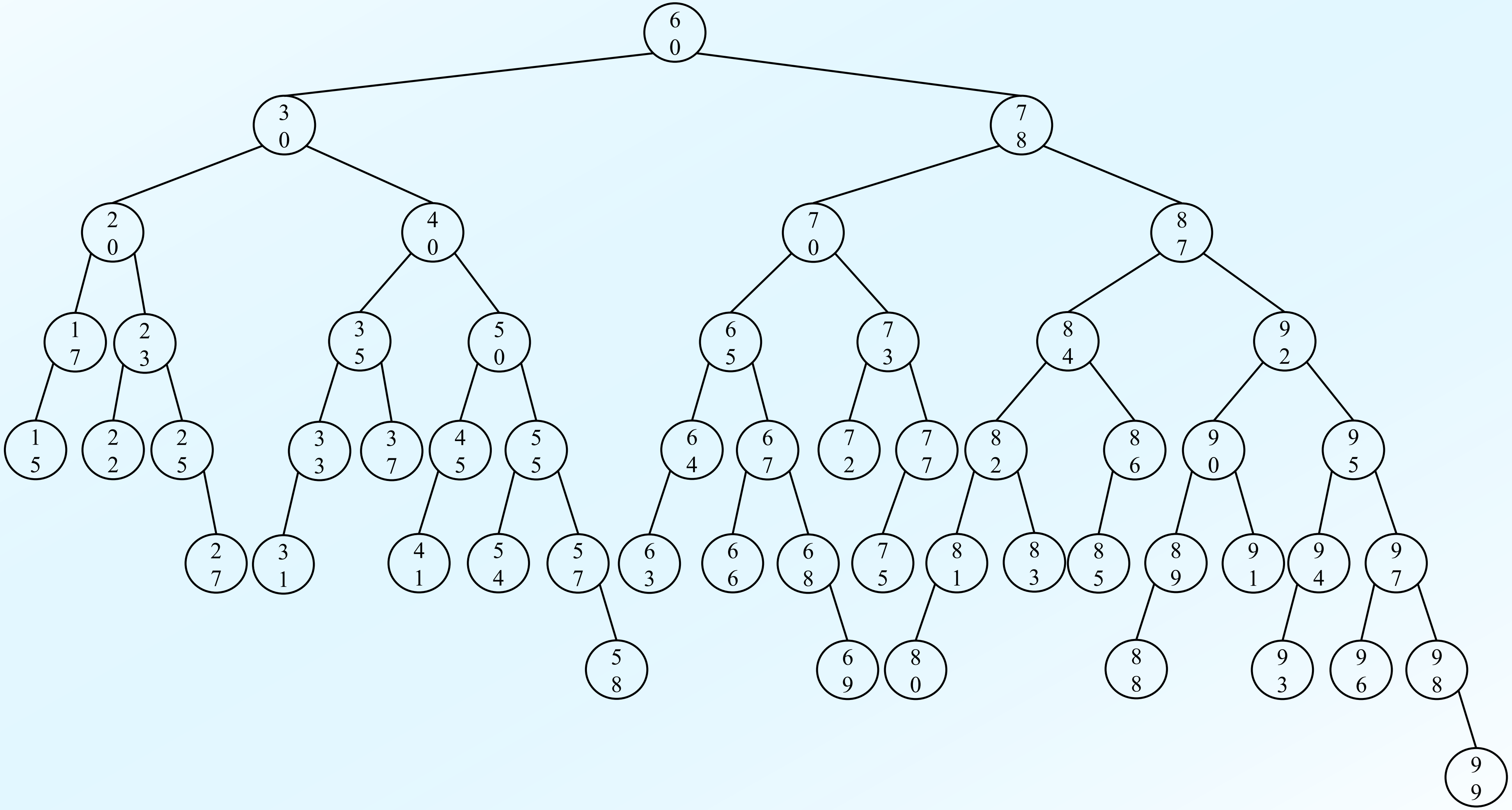


NIL이면 에러 안난다

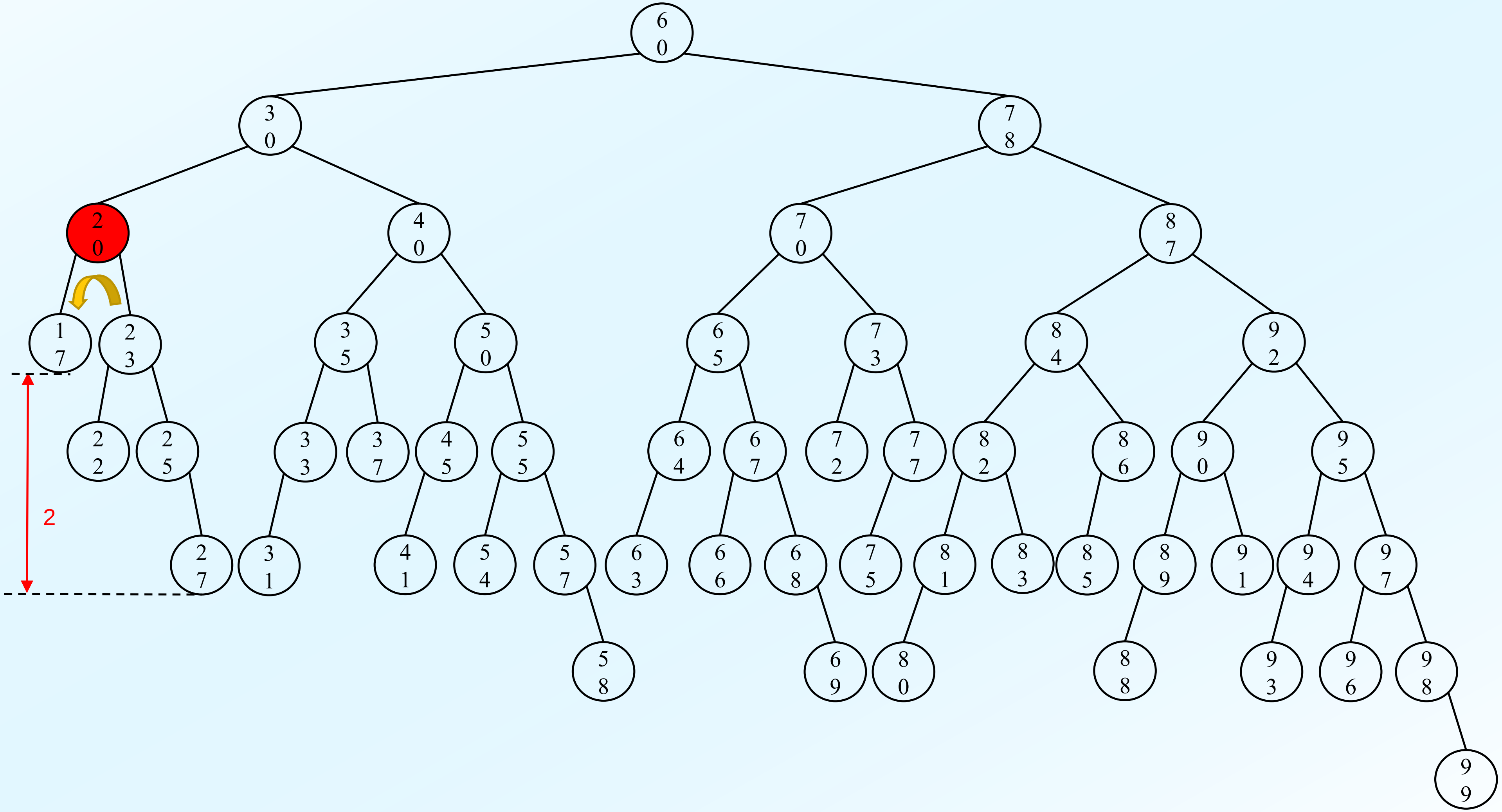
…
t.height ← max(t.right.height, t.left.height) + 1
r.height ← max(r.right.height, r.left.height) + 1

null 대신 **NIL** 노드를 사용
NIL 노드는 하나만 만들면 된다
null도 정상 노드처럼 취급할 수 있다
→ 앞 페이지의 코드를 그대로 쓸 수 있다

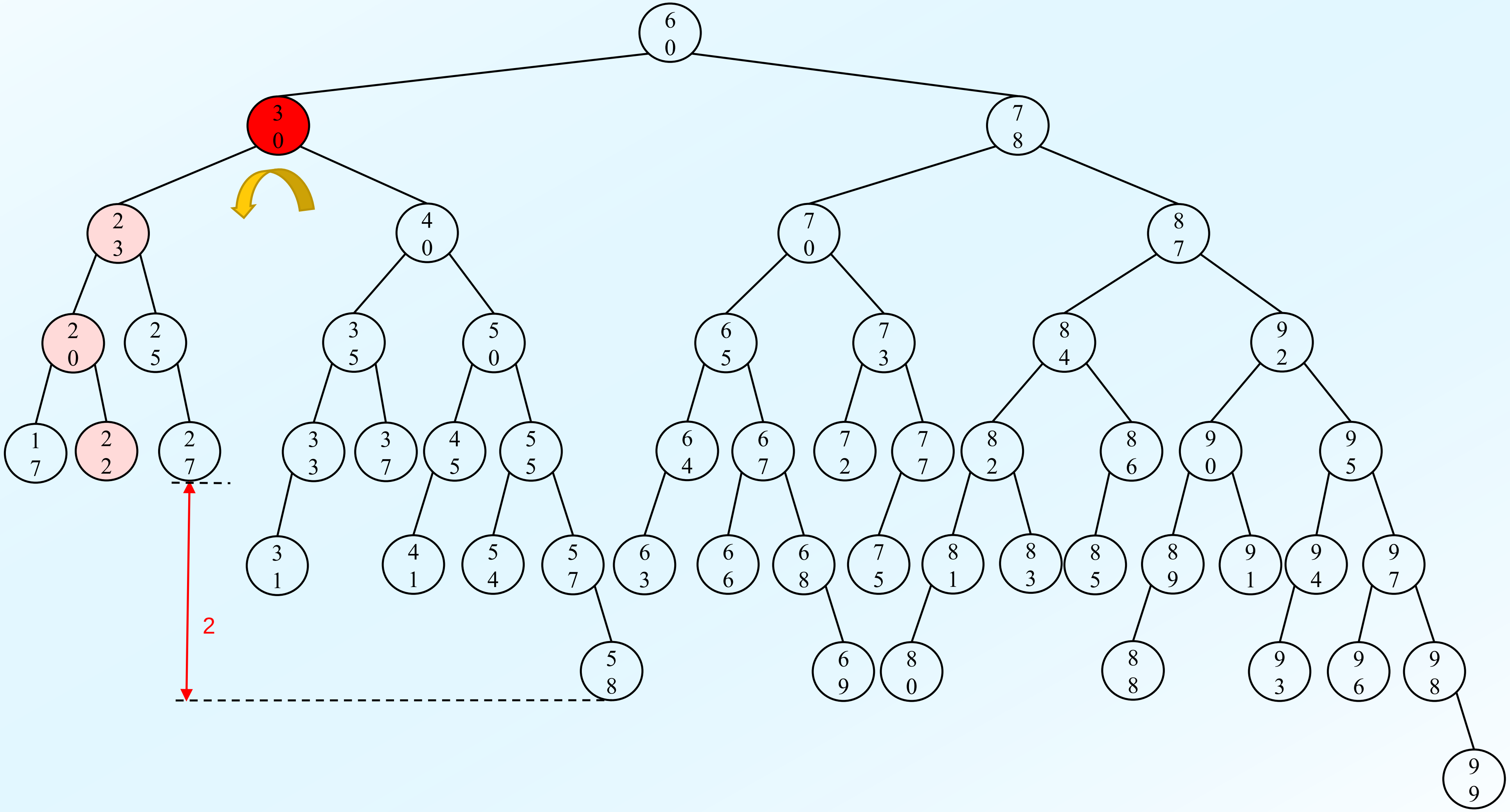
수선이 끝까지 올라갈 수도 있다



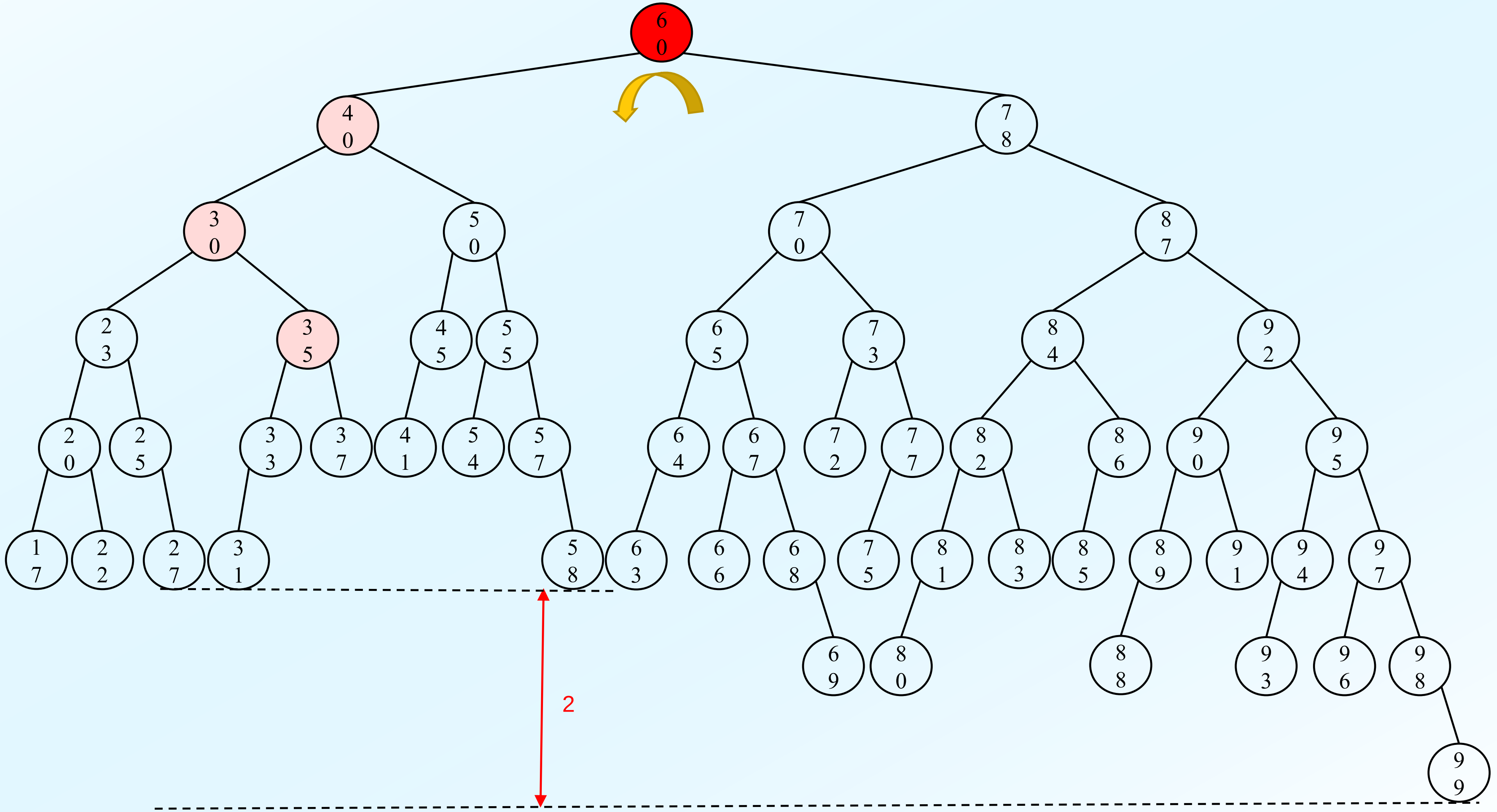
수선이 끝까지 올라갈 수도 있다



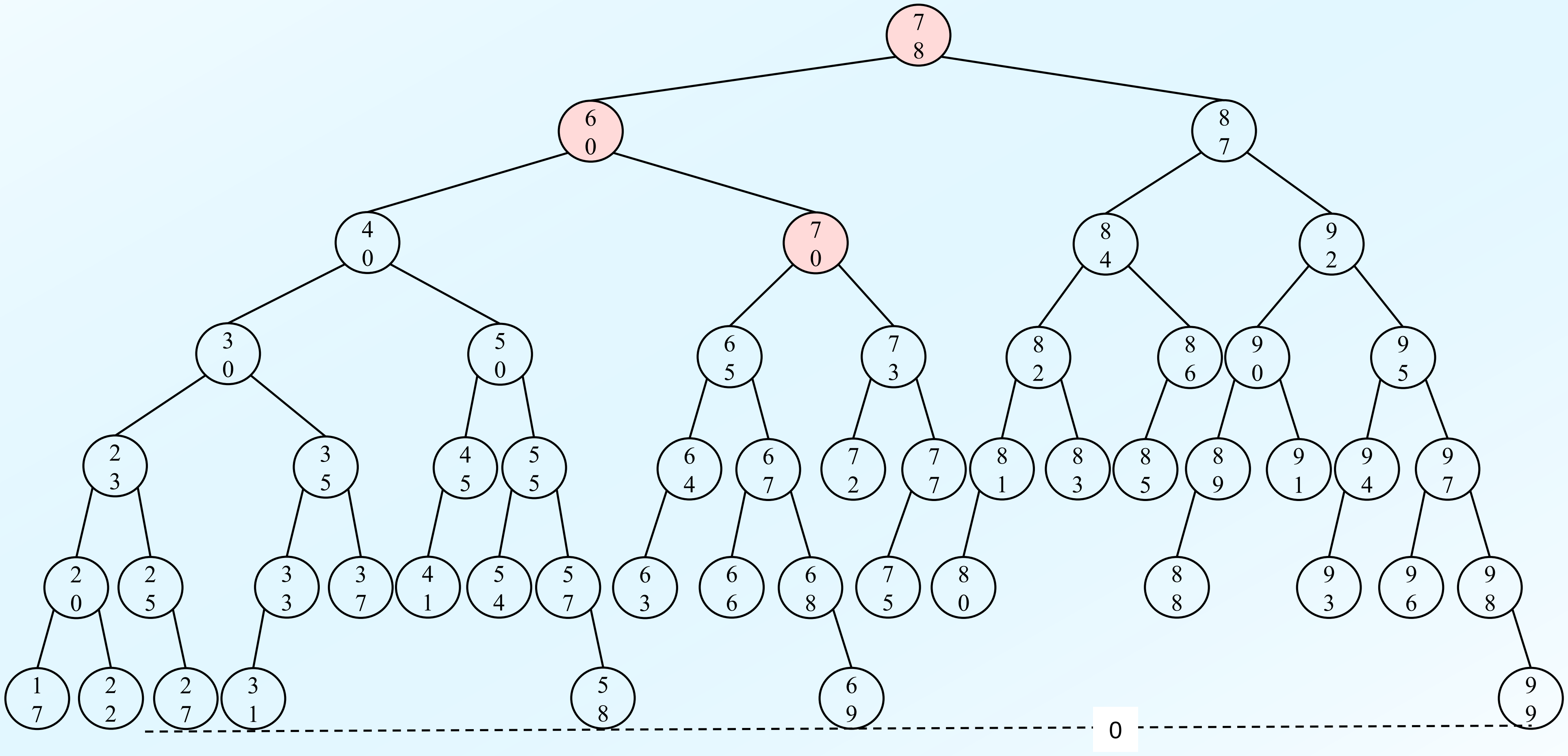
수선이 끝까지 올라갈 수도 있다



수선이 끝까지 올라갈 수도 있다

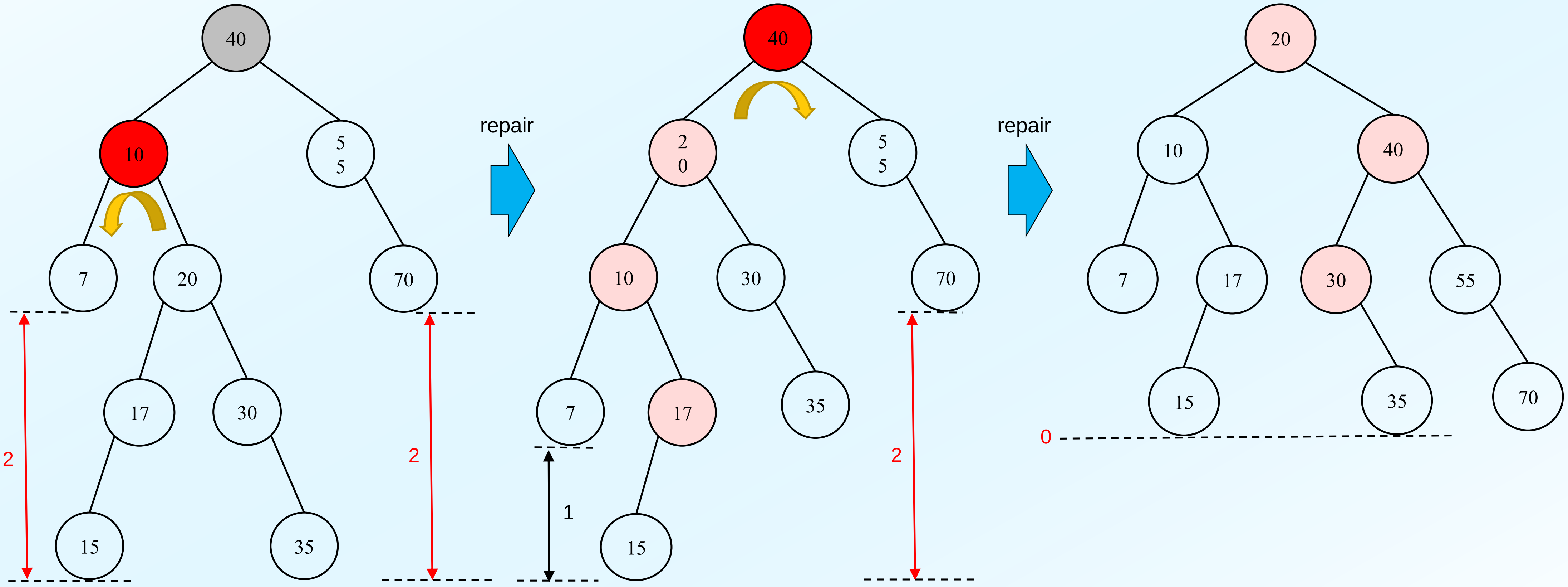


수선이 끝까지 올라갈 수도 있다



생각해보기: An Example

AVL Tree에서
이런 상황이 일어날 수 있을까?



Java 코드

```
// AVL 트리 노드
public class AVLNode {
		public Comparable x;
		public AVLNode left, right;
		public int height;
		public AVLNode(Comparable x) {
				item = x;
				left = right = AVLTree.NIL;
				height = 1;
		}
		public AVLNode(Comparable x,
						AVLNode leftChild, AVLNode rightChild, int h){
				item = x;
				left = leftChild; right = rightChild;
				height = h;
		}
}
```

Java 코드

참고: b.s.t.

```
////// 삽입 ////////////////////////////////////////////////////////////////////
public void insert(Comparable x) {
        root = insertItem(root, x);
}
private TreeNode insertItem(TreeNode t, Comparable x) {
        if (t == null) { // insert after a leaf  (or into an empty tree)
                t = new TreeNode(x);
        else if (x.compareTo(t.item) < 0)
                t.left = insertItem(t.left, x);         // branch left
        else

                t.right = insertItem(t.right, x);     // branch right
        return t;
}
…
```

```
public class AVLTree implements IndexInterface<AVLNode> {
        private AVLNode root;
        static final AVLNode NIL = new AVLNode(null, null, null, 0); // sentinel
        public AVLTree() {
                root = NIL;
        }
        ///// 검색 //////////////////////////////////////
        public AVLNode search(Comparable x) {
                return searchItem(root, x);
        }
        private AVLNode searchItem(AVLNode tNode, Comparable x) {
                if (tNode == NIL) return NIL;
                else if (x.compareTo(tNode.item) == 0) return tNode;
                else if (x.compareTo(tNode.item) < 0)
                                return searchItem(tNode.left, x);
                else
                                return searchItem(tNode.right, x);
        }
        ///// 삽입 //////////////////////////////////////
        public void insert(Comparable x) {
                root = insertItem(root, x);
        }
        private AVLNode insertItem(AVLNode tNode, Comparable x) {
                int type;
                if (tNode == NIL) { // insert after a leaf (or into an empty tree)
                                tNode = new AVLNode(x);
                } else if (x.compareTo(tNode.item) < 0) { // branch left
                                tNode.left = insertItem(tNode.left, x);
                                tNode.height = 1 + Math.max(tNode.right.height, tNode.left.height);
                                type = needBalance(tNode);
                                if (type != NO_NEED)
                                        tNode = balanceAVL(tNode, type);
                } else { // branch right
                                tNode.right = insertItem(tNode.right, x);
                                tNode.height = 1 + Math.max(tNode.right.height, tNode.left.height);
                                type = needBalance(tNode);
                                if (type != NO_NEED)
                                        tNode = balanceAVL(tNode, type);
                }
                return tNode;
        } // end insertItem()
```

Java 코드

* findAndDelete(): 교재 source의 deleteItem()

```
///// 삭제 ///////////////////////////////////////////
public void delete(Comparable x) {
        root = findAndDelete(root, x);
}
private AVLNode findAndDelete(AVLNode tNode, Comparable x) {
        if (tNode == NIL) return NIL; // item not found!
        else {
                if (x.compareTo(tNode.item) == 0) { // item found at tNode
                        tNode = deleteNode(tNode);
                } else if (x.compareTo(tNode.item) < 0) {
                        tNode.left = findAndDelete(tNode.left, x);
                        tNode.height = 1 + Math.max((tNode.right.height, tNode.left.height);
                        int type = needBalance(tNode);
                        if (type != NO_NEED)
                                tNode = balanceAVL(tNode, type);
                } else {
                        tNode.right = findAndDelete(tNode.right, x);
                        tNode.height = 1 + Math.max(tNode.right.height, tNode.left.height);
                        int type = needBalance(tNode);
                        if (type != NO_NEED)
                                tNode = balanceAVL(tNode, type);
                }
                return tNode;
        }
} // end findAndDelete()
private AVLNode deleteNode(AVLNode tNode) {
        // Three cases
        // 1. tNode is a leaf
        // 2. tNode has only one child
        // 3. tNode has two children
        if ((tNode.left == NIL) && (tNode.right == NIL)) { //case 1
                return NIL;
        } else if (tNode.left == NIL) { // case 2 (only right child)
                return tNode.right;
        } else if (tNode.right == NIL) { // case 2 (only left child)
                return tNode.left;
        } else { // case 3 - tNode has two children
                returnPair rPair = deleteMinItem(tNode.right);
                tNode.item = rPair.item; tNode.right = rPair.node;
                tNode.height = 1 + Math.max(tNode.right.height, tNode.left.height);
                int type = needBalance(tNode);
                if (type != NO_NEED)
                        tNode = balanceAVL(tNode, type);
                return tNode;
        }
} // end deleteNode()
```

Java 코드

```
private returnPair deleteMinItem(AVLNode tNode) {
	if (tNode.left == NIL) { // found min at tNode
			return new returnPair(tNode.item, tNode.right);
	} else { // branch left, then backtrack
			returnPair rPair = deleteMinItem(tNode.left);
			tNode.left = rPair.node;
			tNode.height = 1 + Math.max(tNode.right.height, tNode.left.height);
			int type = needBalance(tNode);
			if (type != NO_NEED)
					tNode = balanceAVL(tNode, type);
			rPair.node = tNode;
			return rPair;
	}
} // end deleteMinItem()
private class returnPair {
	Comparable item;
	AVLNode node;
	private returnPair(Comparable it, AVLNode nd) {
			item = it;
			node = nd;
	}
}
/////Balancing 관련 작업 /////////////////////////////
private final int LL = 1, LR =2, RR = 3, RL = 4, NO_NEED = 0, ILLEGAL = -1;
private int needBalance(AVLNode t) {
	int type = ILLEGAL;
	if ( t.left.height+2 <= t.right.height ) { // type R
			if ( (t.right.left.height) <= t.right.right.height )
					type = RR;
			else
					type = RL;
	} else if ( t.left.height >= t.right.height+2 ) { // type L
			if ( t.left.left.height >= t.left.right.height )
					type = LL;
			else
					type = LR;
	} else
			type = NO_NEED;
	return type;
}
```

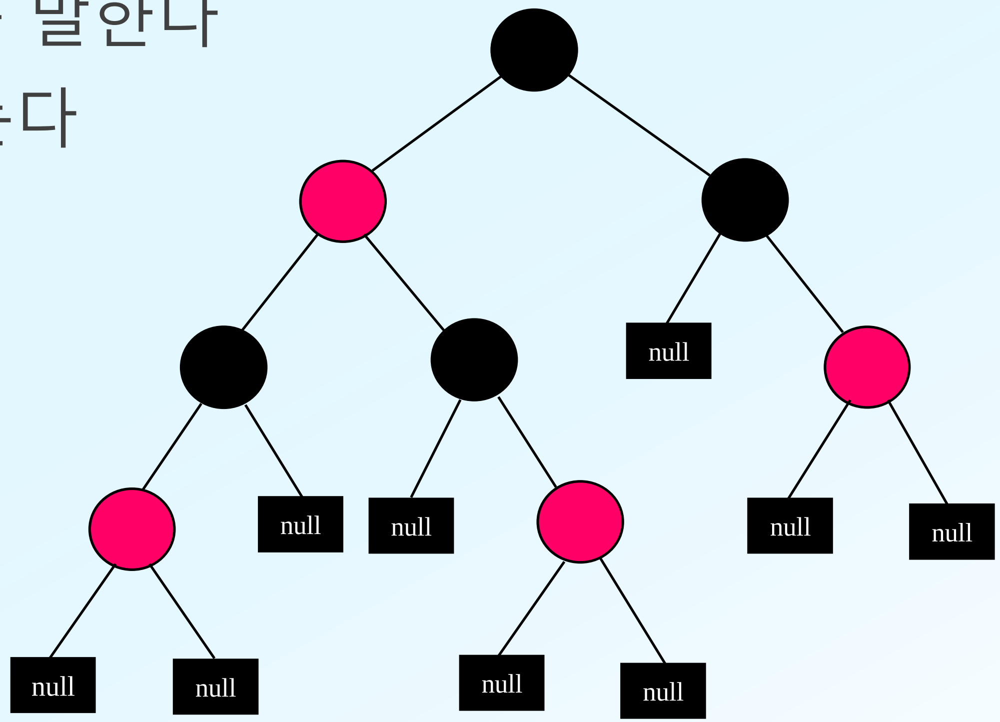
Java 코드

```
private AVLNode balanceAVL(AVLNode tNode, int type) {
        AVLNode returnNode = NIL;
        switch (type) {
            case LL:
                returnNode = rightRotate(tNode);
                break;
            case LR:
                tNode.left = leftRotate(tNode.left);
                returnNode = rightRotate(tNode);
                break;
            case RR:
                returnNode = leftRotate(tNode);
                break;
            case RL:
                tNode.right = rightRotate(tNode.right);
                returnNode = leftRotate(tNode);
                break;
            default:
                System.out.println("Impossible type! Should be one of LL, LR, RR, RL");
                break;
        }
        return returnNode;
} // end balanceAVL()

private AVLNode leftRotate(AVLNode t) {
        AVLNode RChild = t.right;
        if (RChild == NIL)
                System.out.println("t's RChild shouldn't be NIL!");
        AVLNode RLChild = RChild.left;
        RChild.left = t;
        t.right = RLChild;
        t.height = 1 + Math.max(t.left.height, t.right.height);
        RChild.height = 1 + Math.max(RChild.left.height, Rchild.right.height));
        return RChild;
}
private AVLNode rightRotate(AVLNode t) {
        AVLNode LChild = t.left;
        if (LChild == NIL)
                System.out.println("t's LChild shouldn't be NIL!");
        AVLNode LRChild = LChild.right;
        LChild.right = t;
        t.left = LRChild;
        t.height = 1 + Math.max(t.left.height, t.right.height));
        LChild.height = 1 + Math.max(LChild.left.height, LChild.right.height);
        return LChild;
}
} // end class AVLTree
```

3. Red-Black Tree

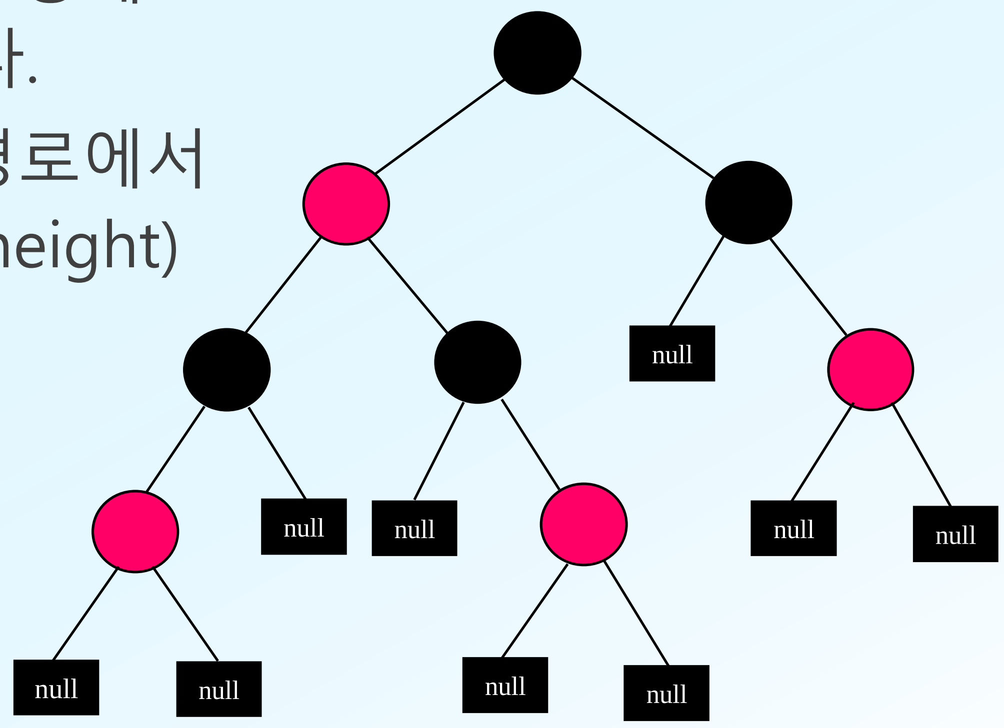
RB Tree의 정의

- 모든 null 자리에 리프 노드를 둔다
- RB-Tree에서 리프 노드는 이 null 리프를 말한다
- 모든 노드는 **레드** 또는 **블랙**의 색을 갖는다
- <u>RB 특성</u>을 만족한다

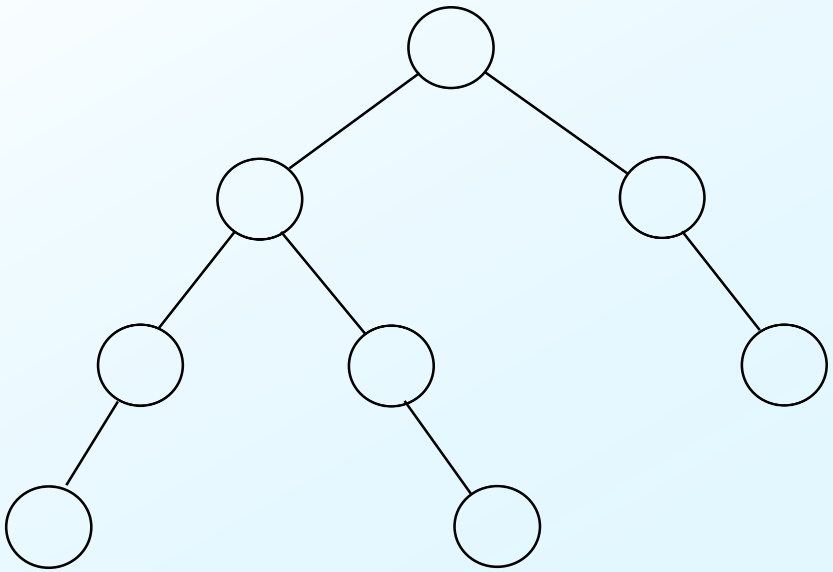


RB 특성

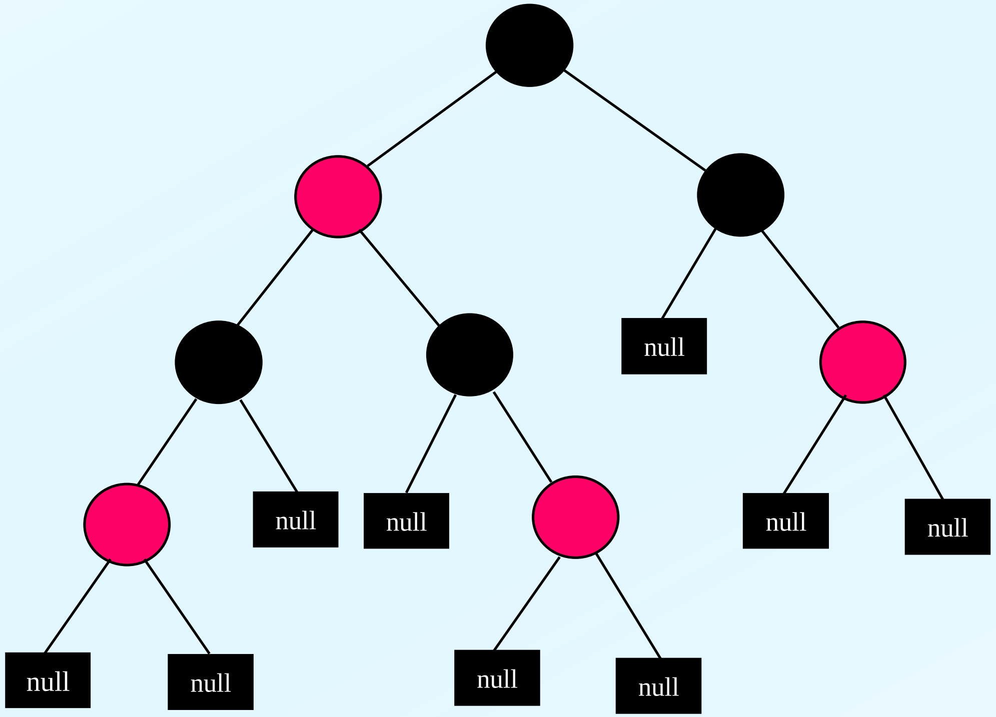
① 루트는 **블랙**이다.
② 모든 리프 노드는 **블랙**이다.
③ 루트로부터 임의의 리프 노드에 이르는 경로 상에 **레드** 노드 두 개가 연속으로 출현하지 못한다.
④ 루트 노드에서 임의의 리프 노드에 이르는 경로에서 만나는 **블랙** 노드의 수는 모두 같다. (black height)



Red-Black Tree의 예



(a) B.S.T.의 한 예



(b) (a)를 RB-트리로 만든 예

구현시에

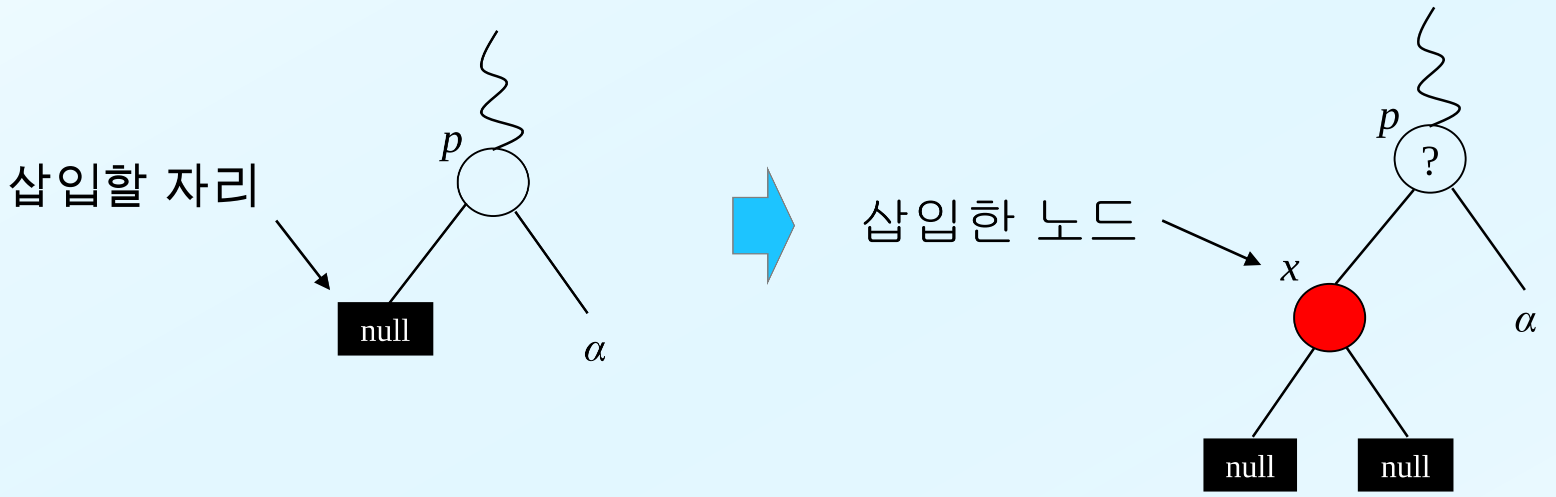
null 은 AVL-트리에서처럼 sentinel **NIL**을 레퍼런스하면 효과적이다

수선

운용 중에 RB 특성이 깨지면 만족하도록 수선한다

Insertion

삽입: 일반적인 BST의 삽입 작업 후,
삽입 노드에 레드를 칠하고,
삽입한 노드의 좌우에 **null** 리프를 달아준다

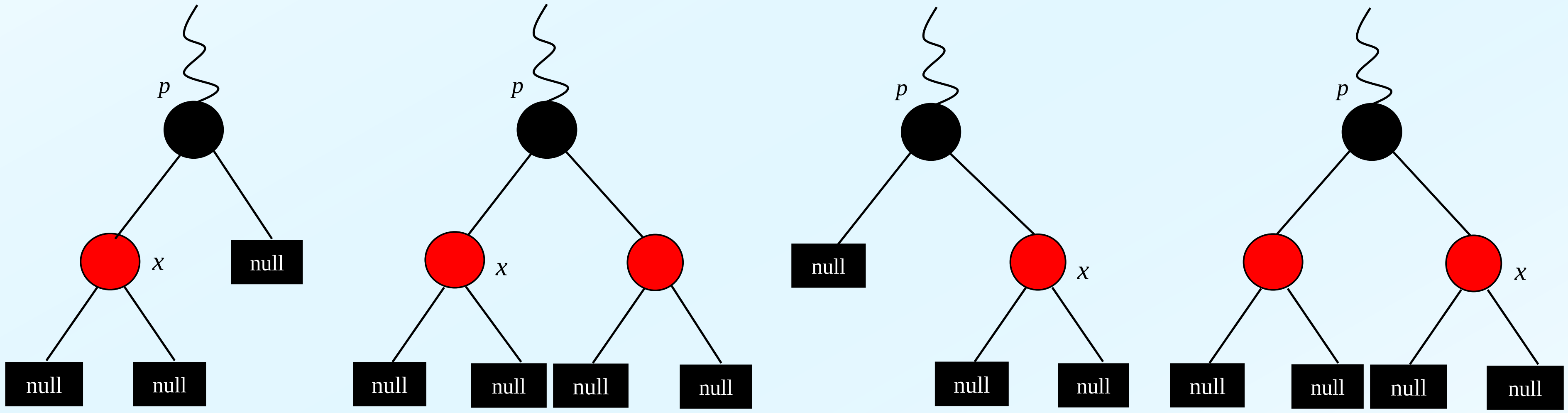


삽입 직후의 상황:
p가 **블랙** 또는 **레드**다

Insertion

삽입 직후의 상황: p가 **블랙** 또는 **레드**다

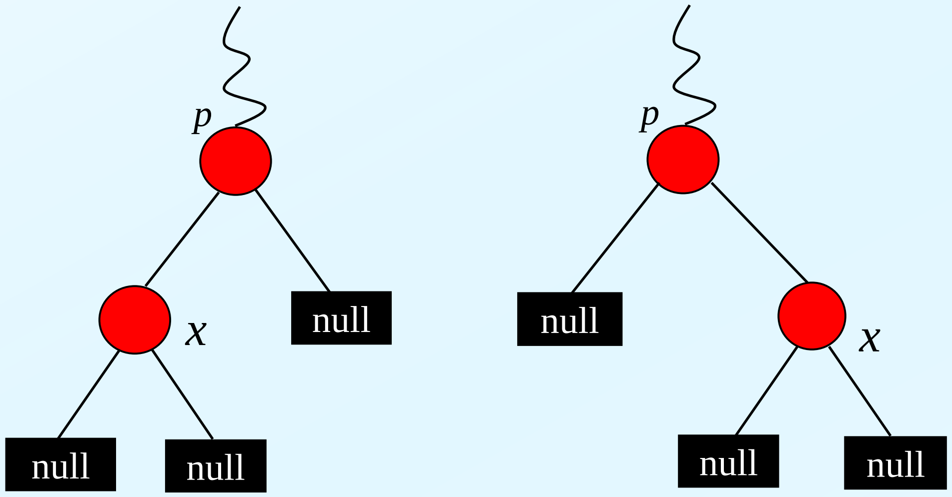
1. If p가 **블랙**: RB 특성 다 만족. 완료!



가능한 모양은 이 4가지 뿐이다

Insertion

2. If p가 **레드**: RB 특성 ③이 깨졌다 → 수선 (다음 페이지 이후)



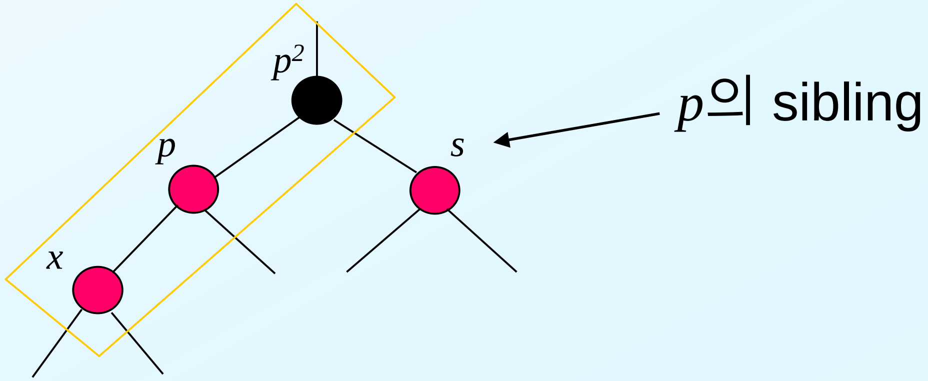
이 둘은 대칭적.
왼쪽 케이스만 소개.

가능한 모양은 이 2가지 뿐이다

수선

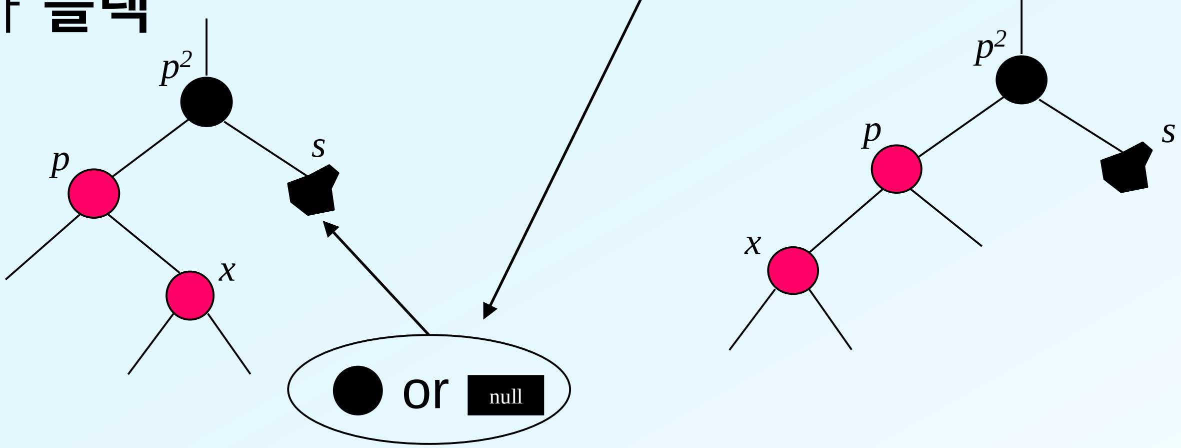
p의 sibling s에 따라 두 가지로 나눈다

Case 1: s가 **레드**



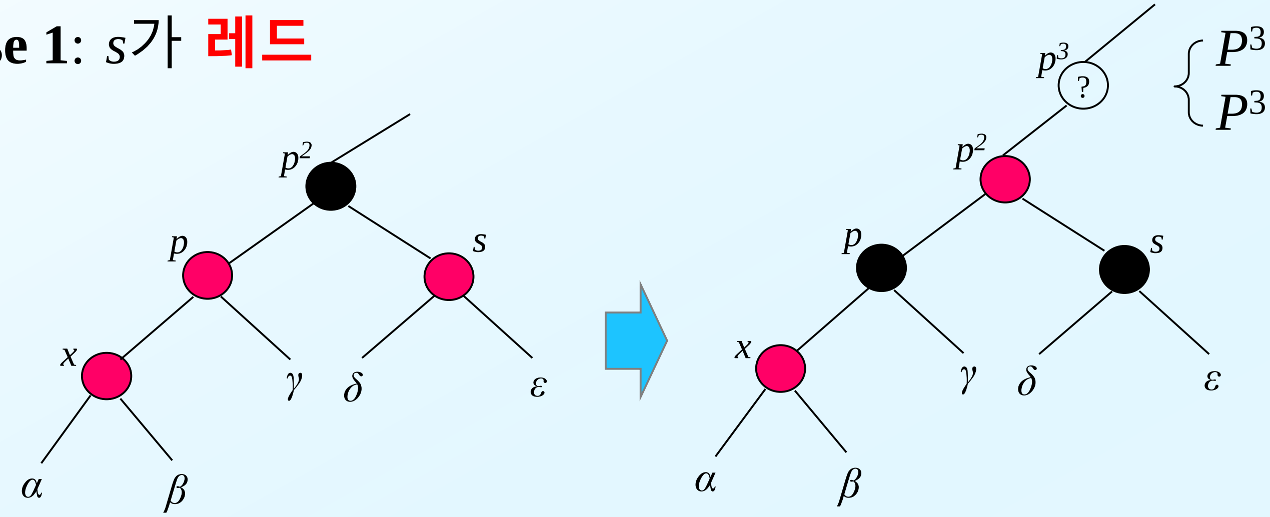
✓ 질문: 여기서 ● 가 가능한가?

Case 2: s가 **블랙**



수선

Case 1: s가 **레드**



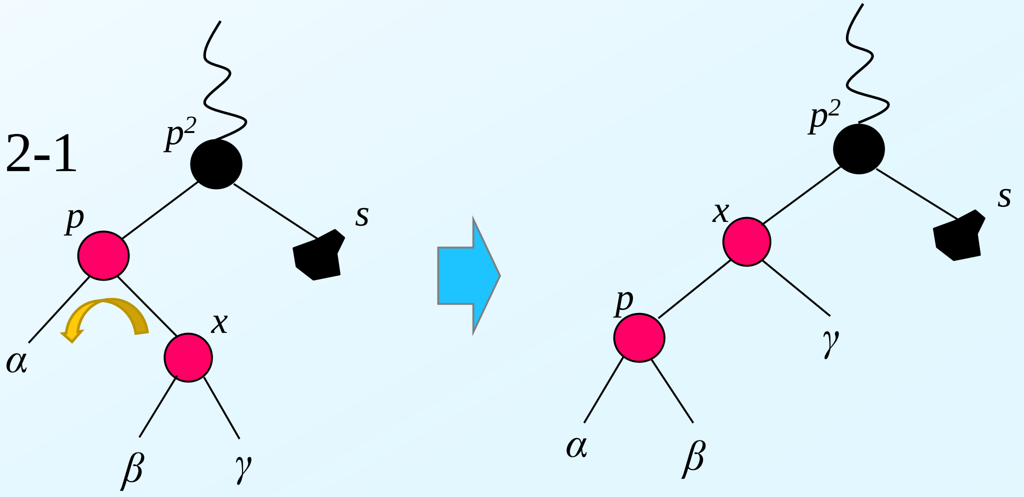
P^3가 **블랙**이면 완료
P^3가 **레드**이면 p^2가 새 x : Recursive!

p와 s를 **블랙**으로 바꾸고,
p^2을 **레드**로 바꾼다.

수선

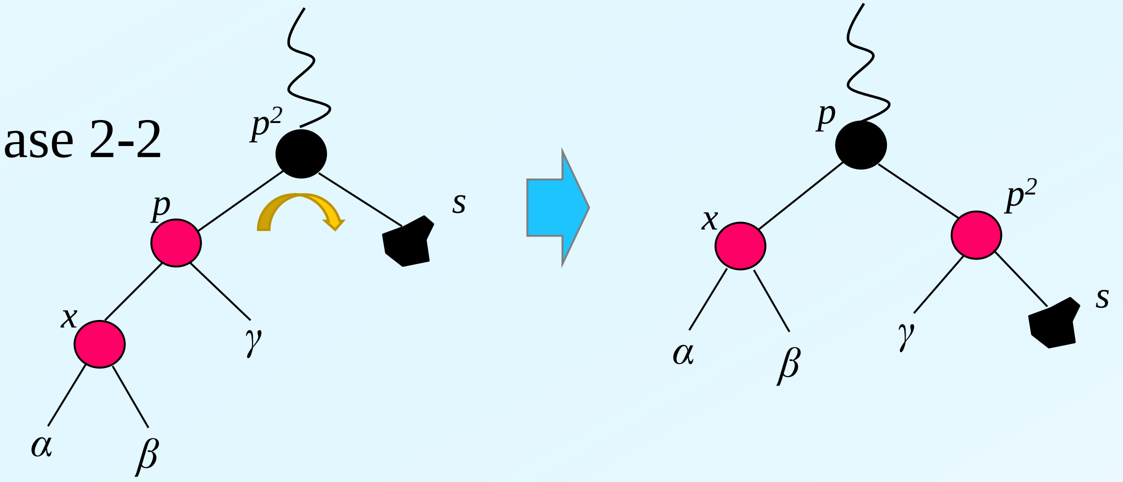
Case 2: s가 **블랙**

Case 2-1



Case 2-2로 변환

Case 2-2

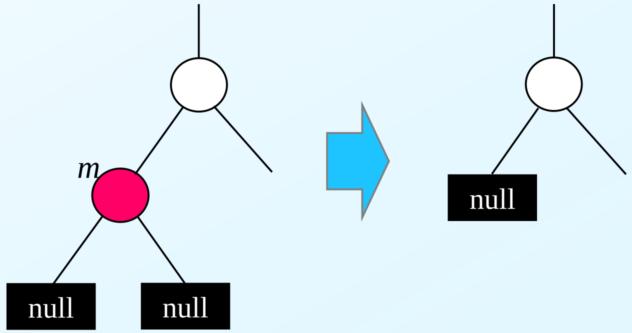


Right Rotation. 수선 끝.

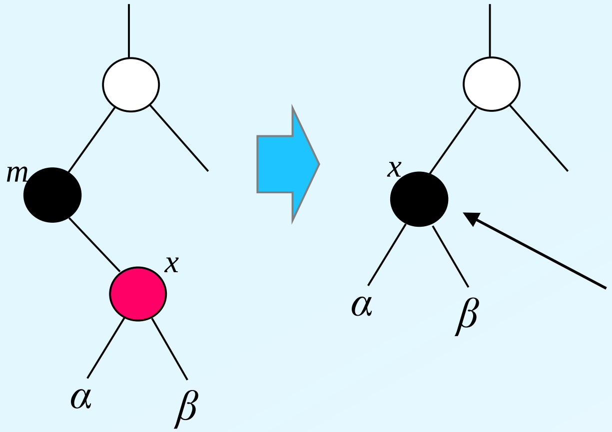
Deletion

삭제: BST의 삭제 작업 중 Case 1과 2만 고려하면 됨

✓ 질문: Case 3은 왜 고려하지 않아도 되는가?



삭제 노드(m)가 **레드**이면 문제없다

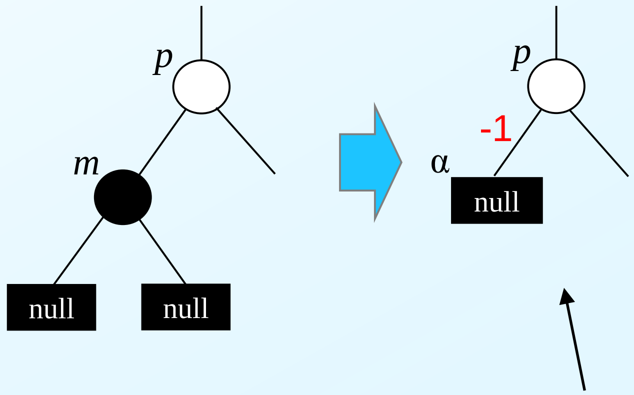


x를 **블랙**으로 색을 바꾸어준다

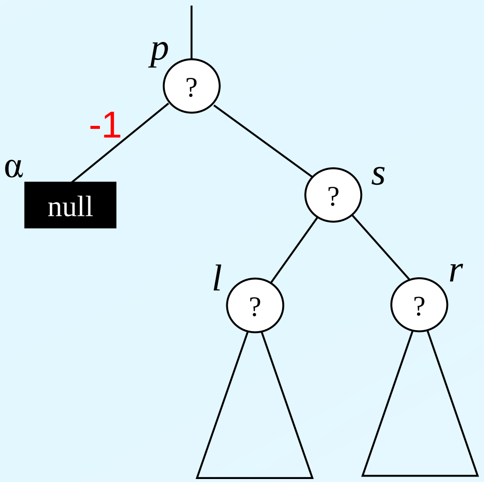
삭제 노드가 **블랙**이라도 <u>자식이 있으면</u> 문제없다
(자식(x)은 유일하고 **레드**다)

Deletion

삭제 노드가 **블랙**이고
자식이 없을 때 문제 발생



RB 특성 ④가 깨진다



✓ -1: 루트에서 α에 이르는 경로에서
블랙 노드의 수가 하나 모자람

α의 주변 상황에 따라 처리 방법이 달라진다

이후의 처리는 제법 복잡하다.
이 강좌에서는 여기까지만.

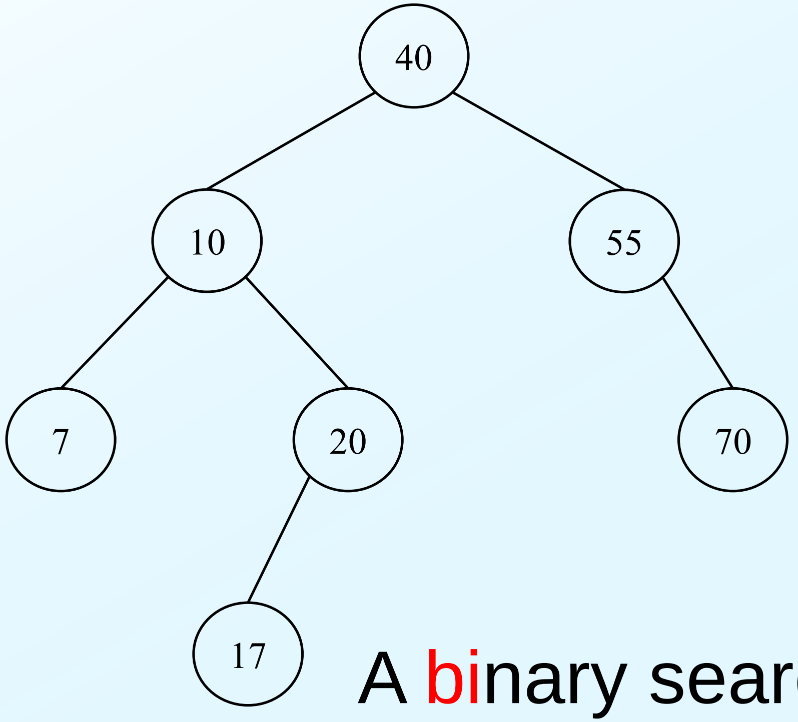
Time Complexity

AVL 트리와 RB 트리 모두
검색, 삽입, 삭제에 $O(\log n)$ 시간이 보장된다

✓생각해 보기: 위의 성질이 왜 만족되는지 생각해보자
직관적으로 생각할 것

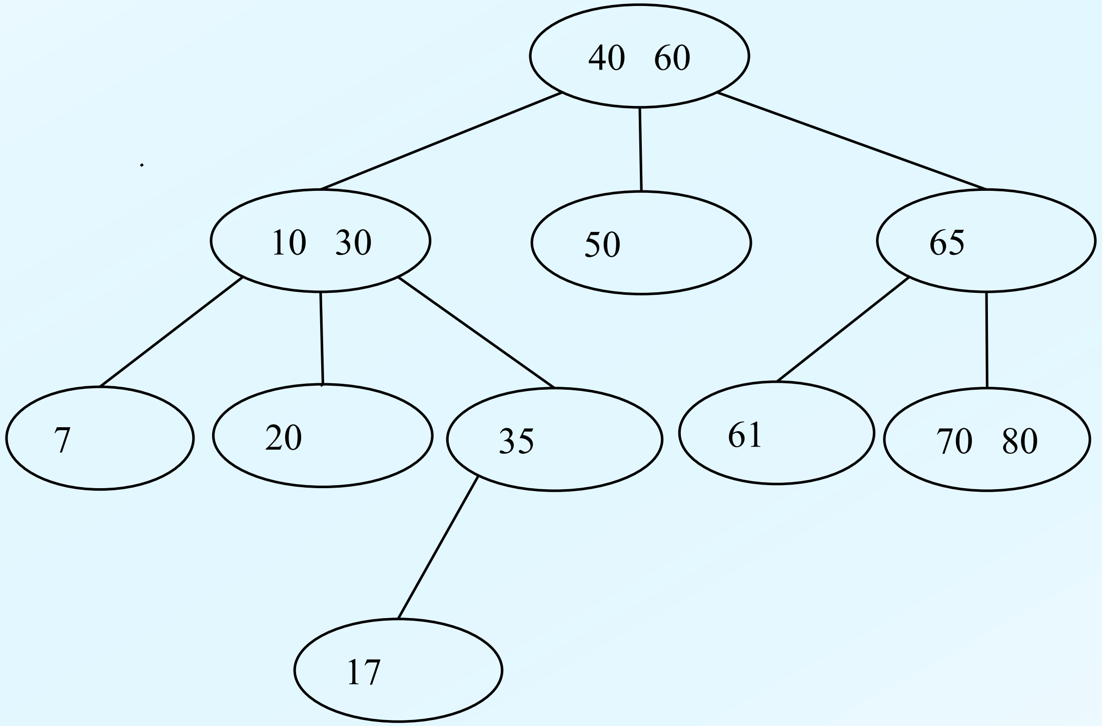
3. B-Tree

K-ary Search Tree



A binary search tree

K=2, 최대 2개 분기



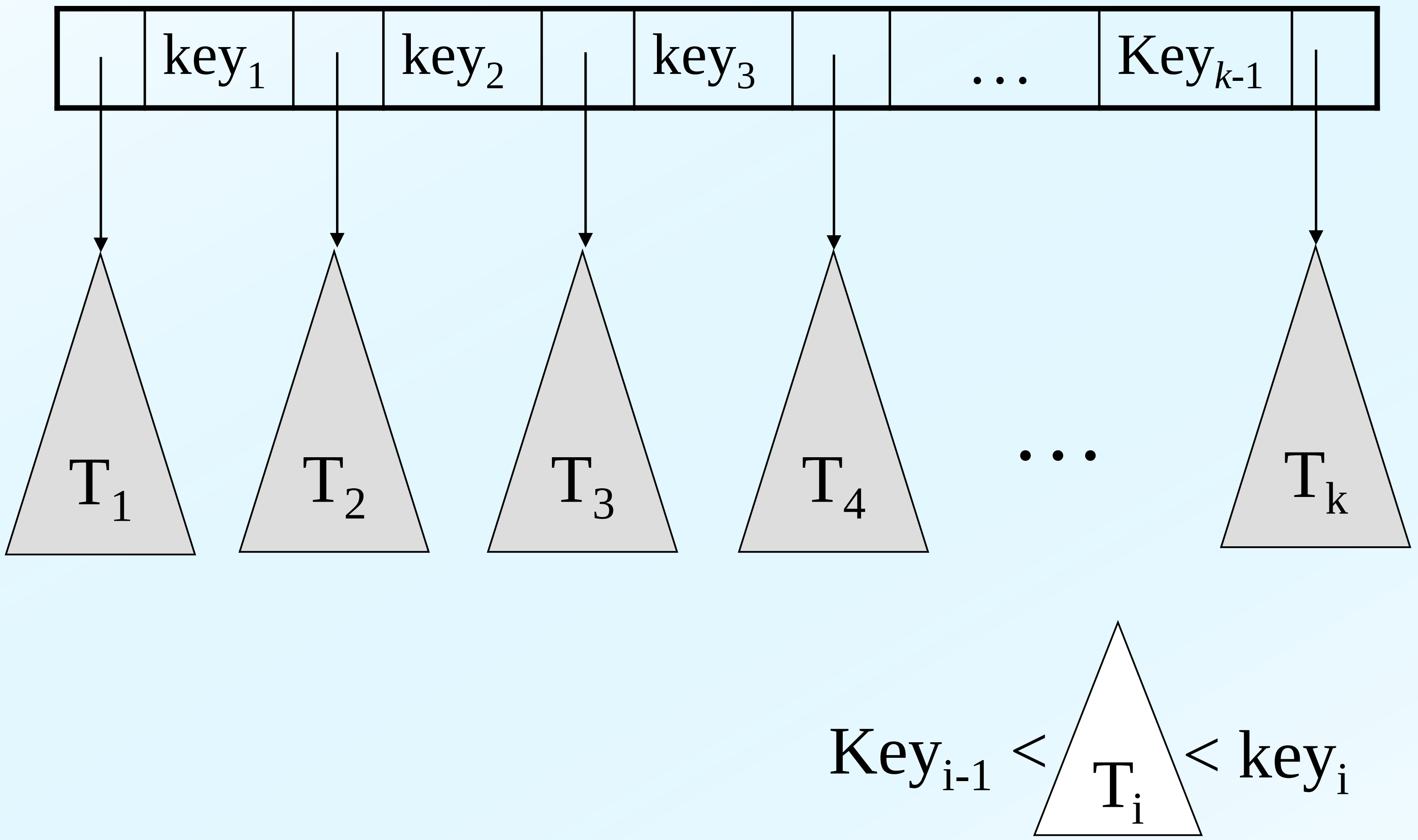
A ternary search tree

K=3, 최대 3개 분기

B-Tree의 환경

- 디스크의 접근 단위는 블록(페이지)
- 디스크에 한 번 접근하는 시간은 수십만 명령어의 처리 시간과 맞먹는다
- 검색트리가 디스크에 저장되어 있다면
  트리의 높이를 최소화하는 것이 유리하다
- B-트리는 K-ary search tree가 균형을 유지하도록 하여
  최악의 경우 디스크 접근 횟수를 줄인 것이다

K-ary Search Tree



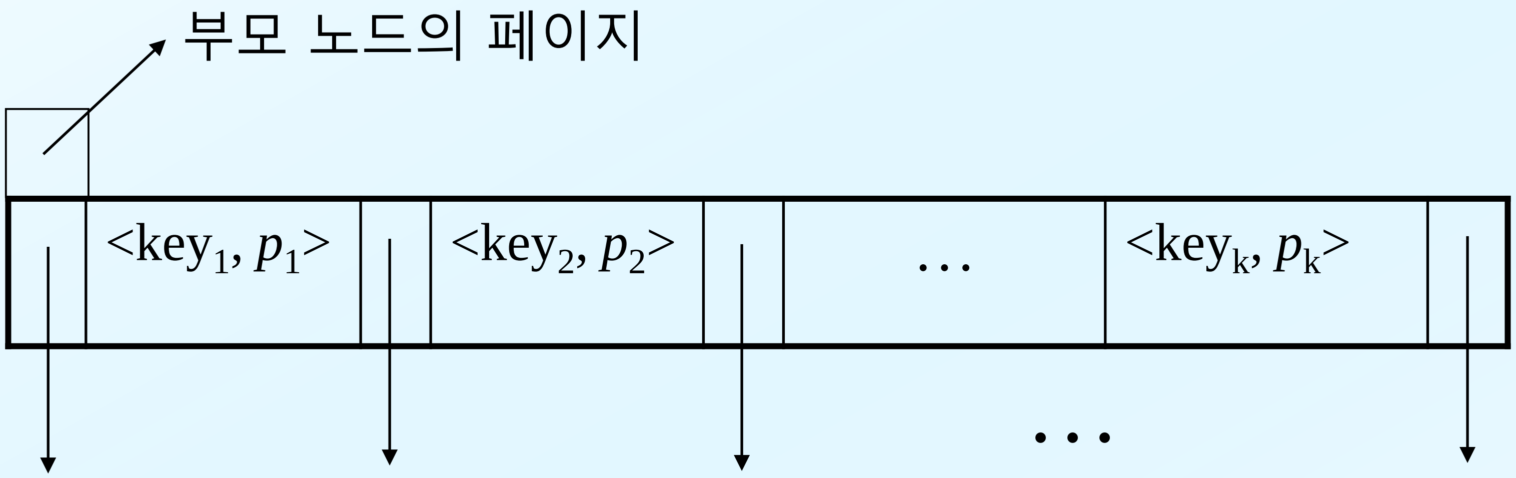
B-Tree의 성질

B-트리는 balanced (K+1)-way search로 다음의 성질을 만족한다

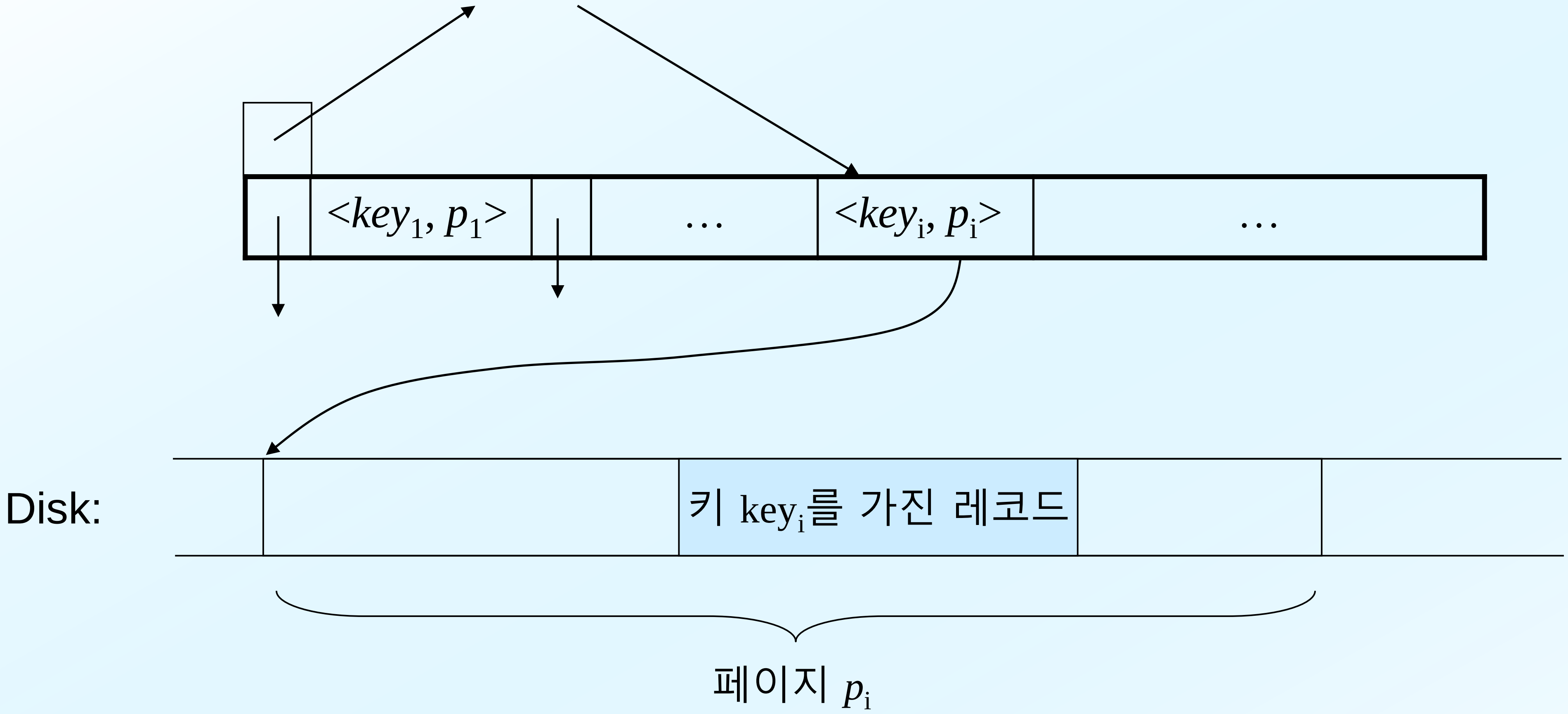
① 루트를 제외한 모든 노드는 $\lfloor K/2 \rfloor \sim K$ 개의 키를 갖는다

② 모든 리프 노드는 같은 깊이를 가진다

B-Tree의 노드 구조



B-Tree를 통해 레코드에 접근하는 과정



Insertion

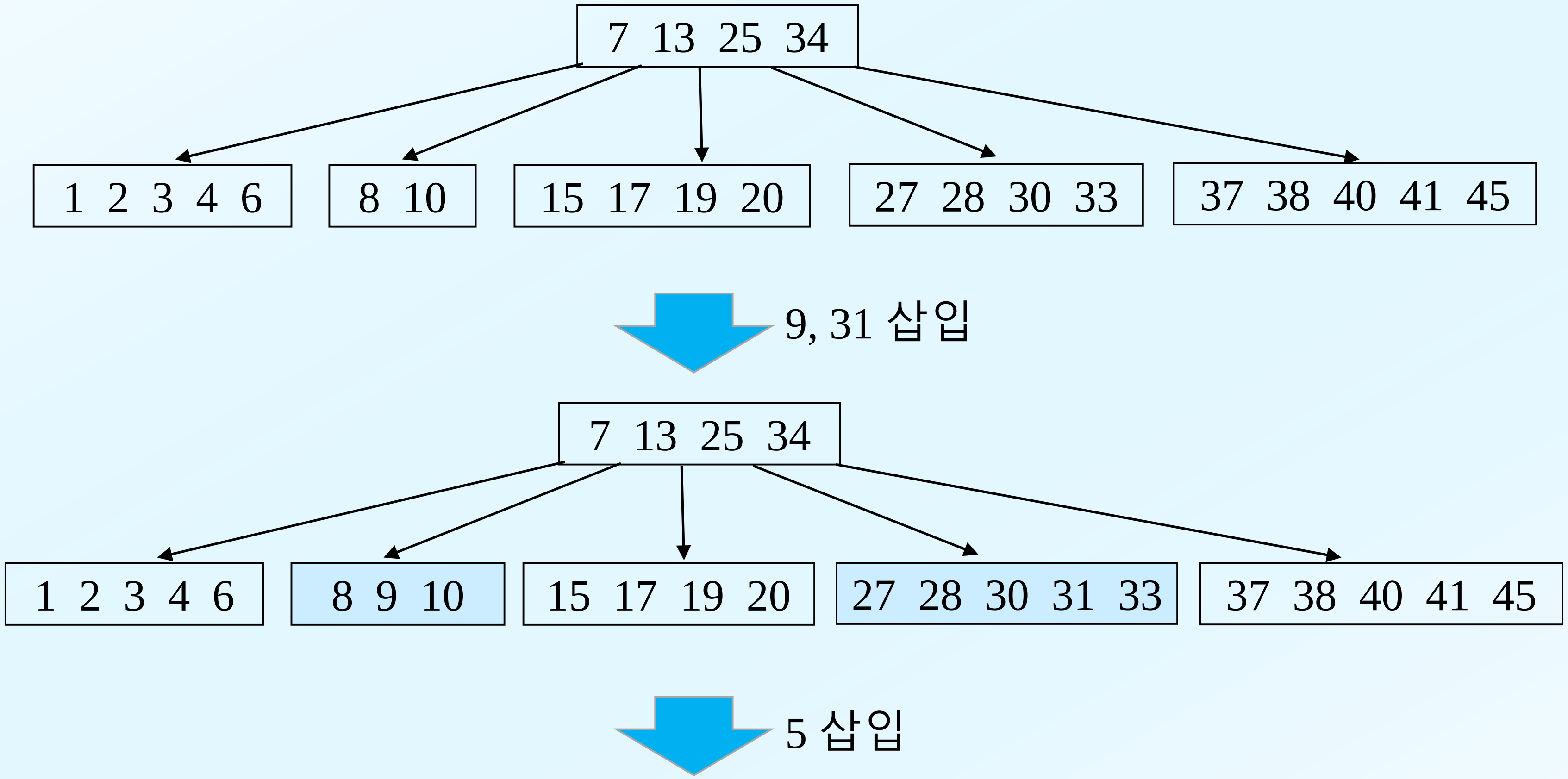
```
BTreeInsert(t, x):
        x를 삽입할 리프 노드 r을 찾는다
        x를 r에 삽입 시도한다

        if (r에  오버플로우 발생)
                clearOverflow(r)
```

▷ t : 트리의 루트 노드
▷ x : 삽입하고자 하는 키

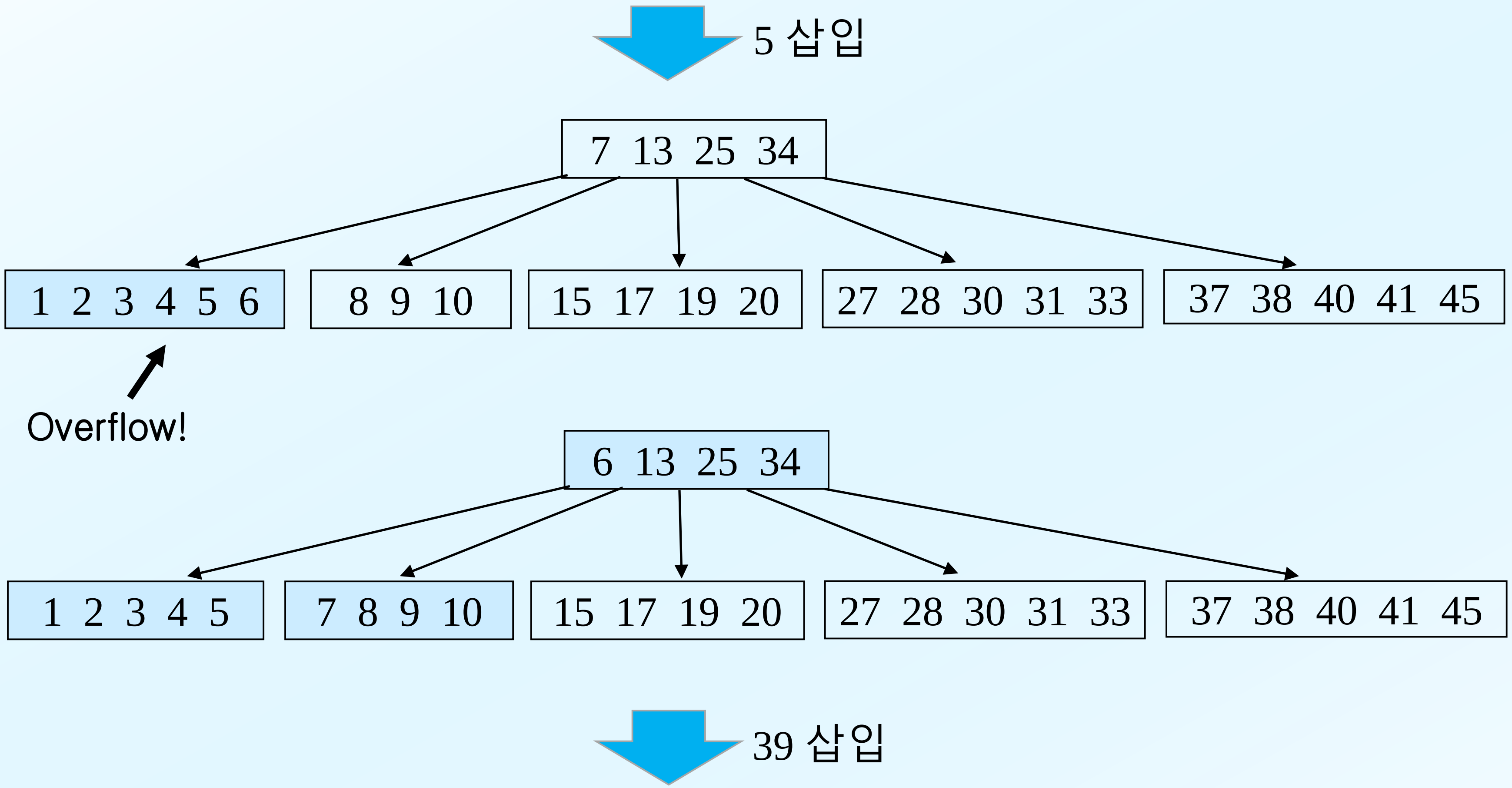
```
clearOverflow(r):
    if (r의 sibling 중 여유가 있는 노드가 있음)
            r의 남는 키를 넘긴다
    else
            r을 둘로 분할하고 가운데 키를 parent p로 넘긴다
            if (p에 오버플로우 발생)
                    clearOverflow(p)
```

Insertion의 예

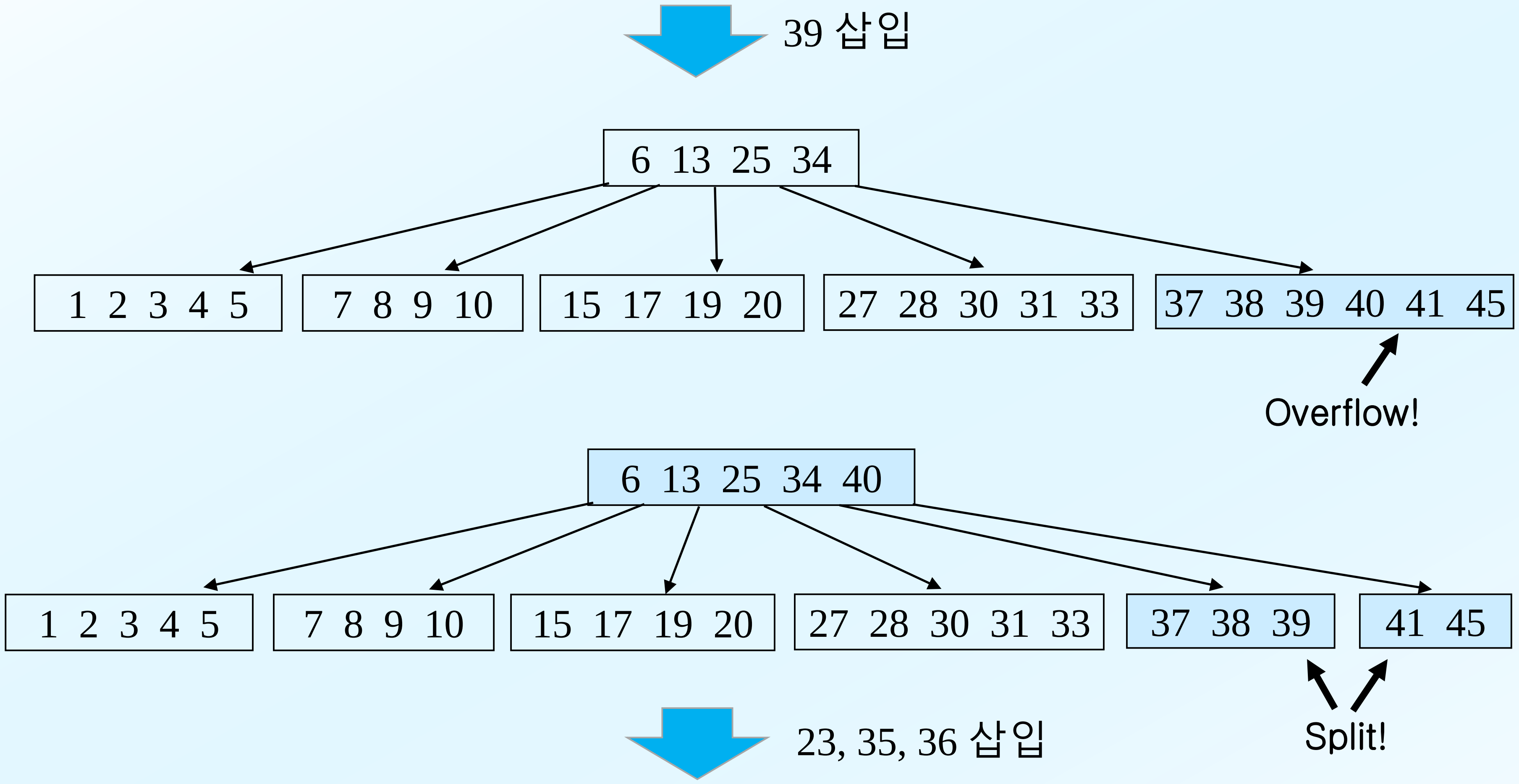
K = 5인 경우



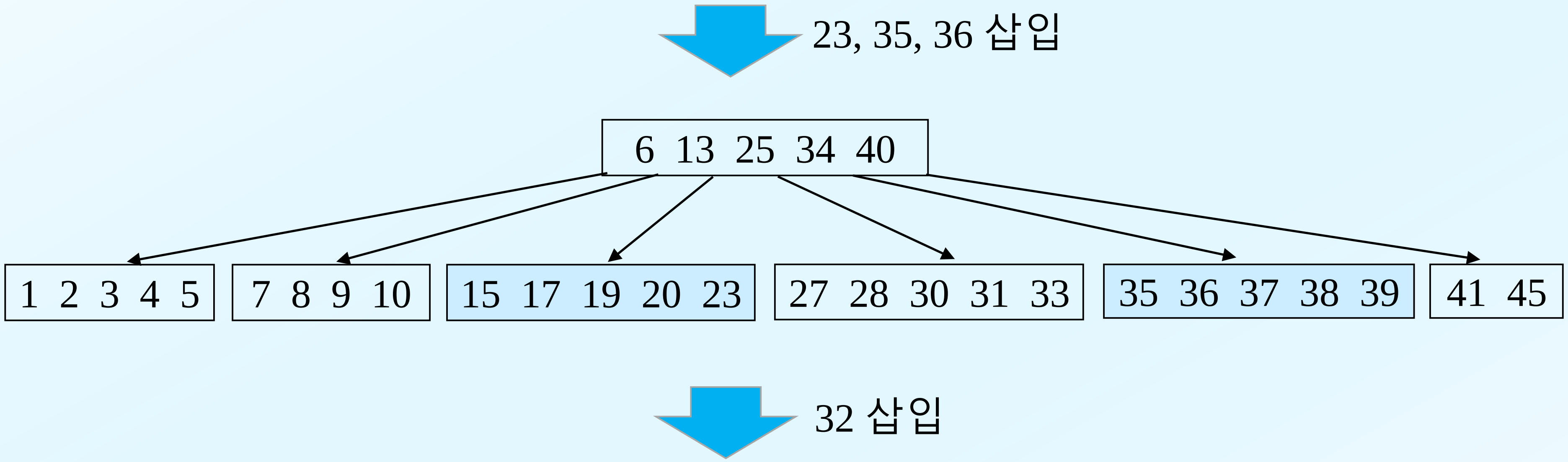
Insertion의 예



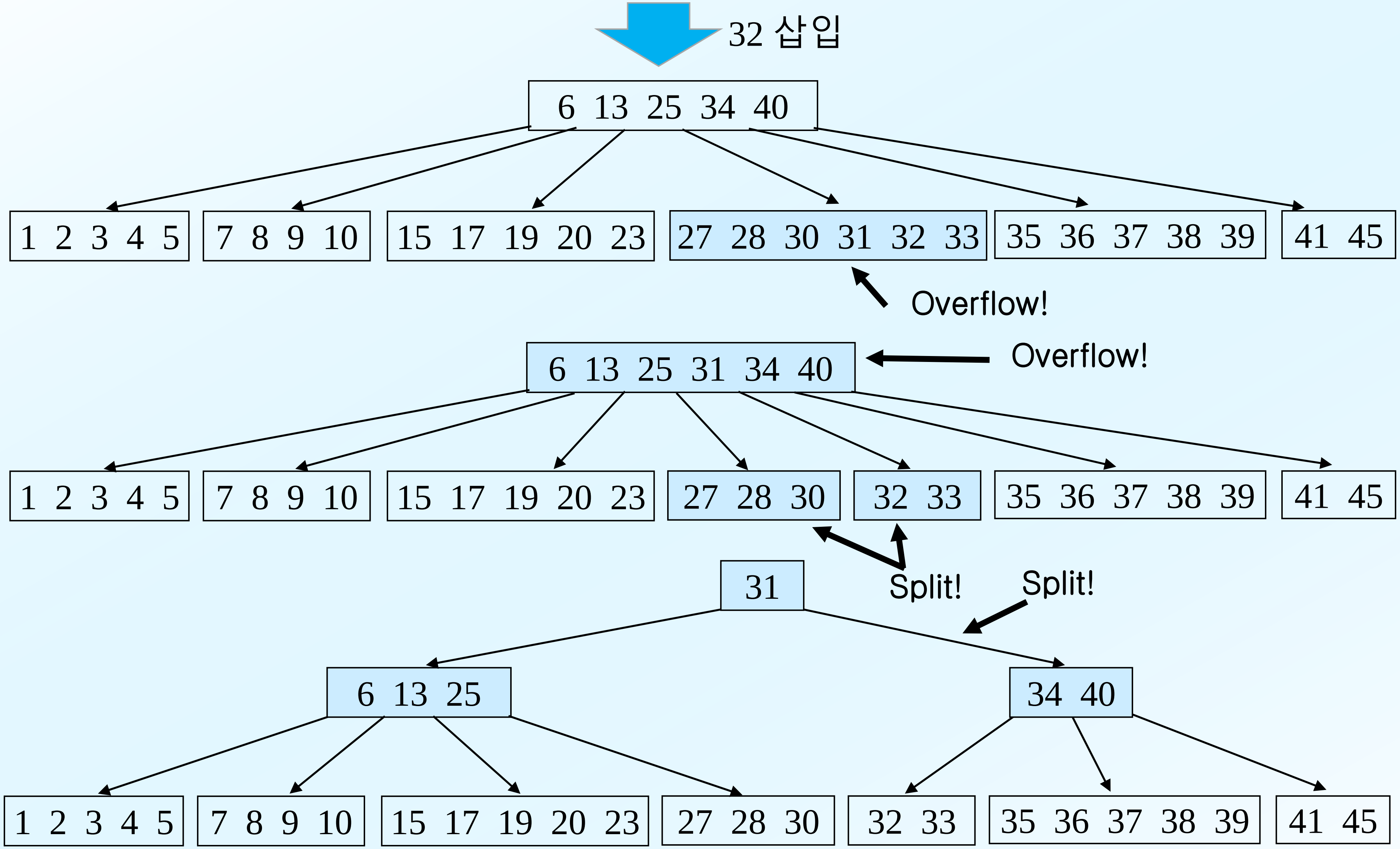
Insertion의 예



Insertion의 예



Insertion의 예



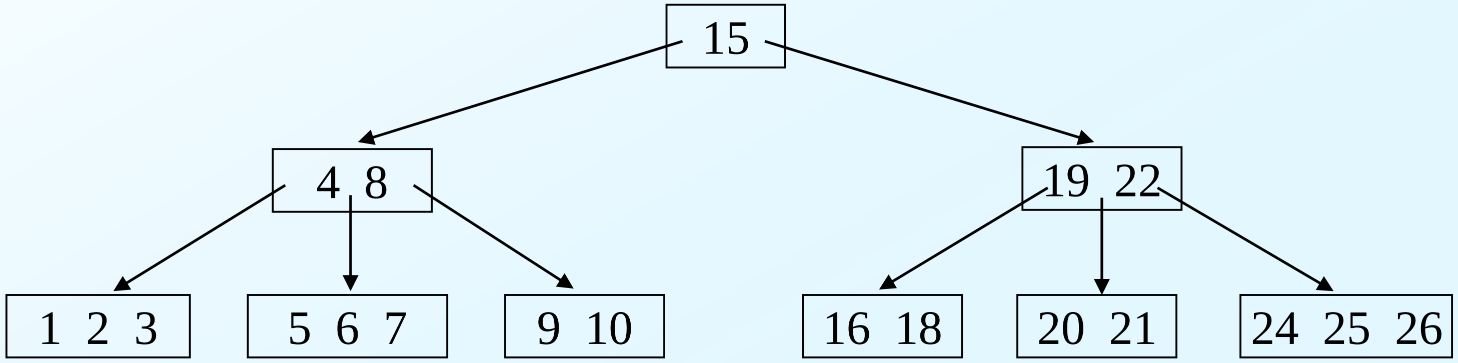
Deletion

▷ t : 트리의 루트 노드
▷ x : 삭제하고자 하는 키
▷ v : x를 갖고 있는 노드

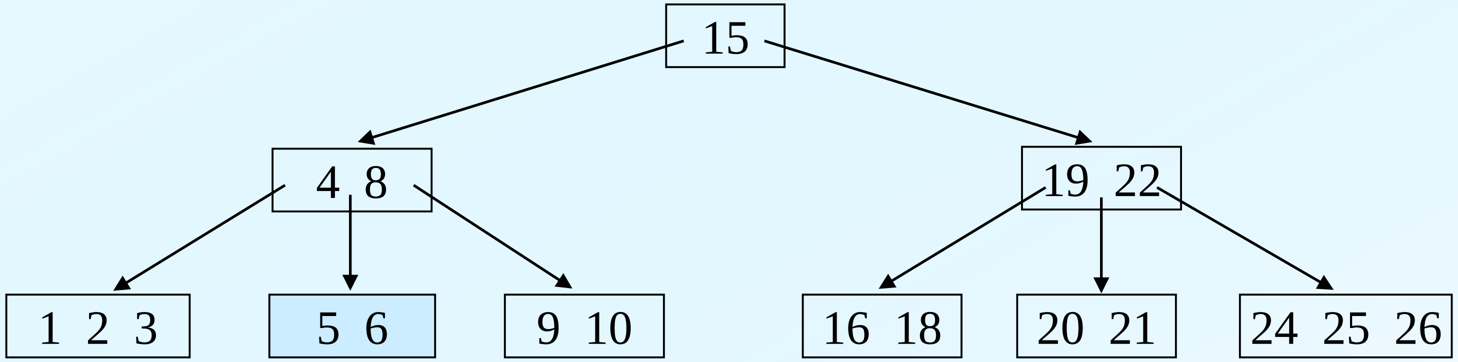
BTreeDelete(t, x, v):
    if (v가 리프 노드 아님)
        x의 직후원소 y를 가진 리프 노드를 찾는다
        x와 y를 맞바꾼다
    리프 노드에서 x를 제거하고 이 리프 노드를 r이라 한다
    if (r에서 언더플로우 발생)
        clearUnderflow(r)

clearUnderflow(r):
    if (r의 sibling 중 키를 하나 내놓을 수 있는 여분을 가진 노드가 있음)
        r이 키를 넘겨받는다
    else
        r과 옆 sibling을 합병한다
        if (부모 노드 p에 언더플로우 발생)
            clearUnderflow(p)

Deletion의 예

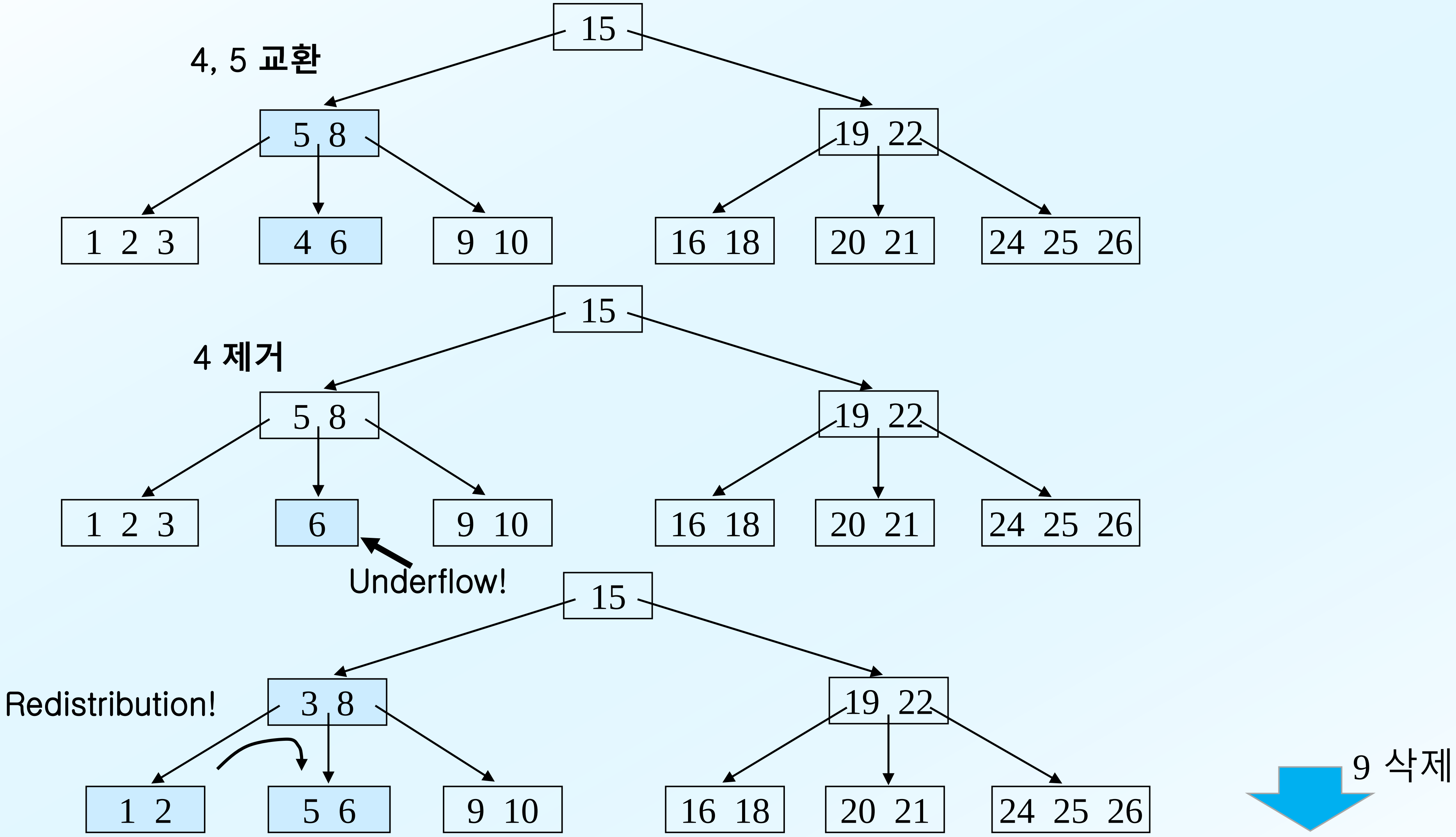


7 삭제

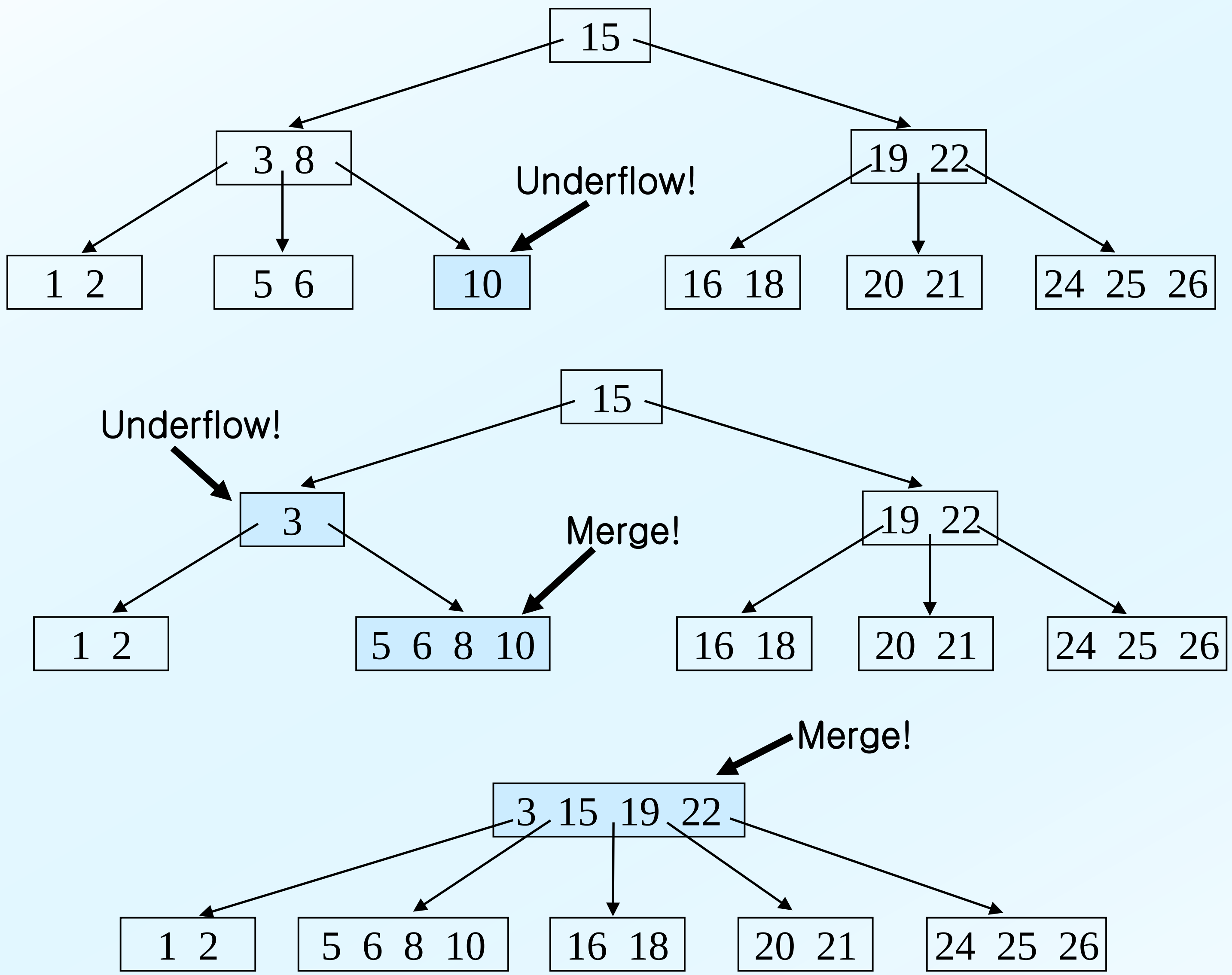


4 삭제

Deletion의 예


4, 5 교환
15
5 8
19 22
1 2 3
4 6
9 10
16 18
20 21
24 25 26
4 제거
15
5 8
19 22
1 2 3
6
9 10
16 18
20 21
24 25 26
Underflow!
15
Redistribution!
3 8
19 22
1 2
5 6
9 10
16 18
20 21
24 25 26
9 삭제

Deletion의 예



15
3 8
19 22
Underflow!
1 2
5 6
10
16 18
20 21
24 25 26
15
Underflow!
3
Merge!
19 22
1 2
5 6 8 10
16 18
20 21
24 25 26
Merge!
3 15 19 22
1 2
5 6 8 10
16 18
20 21
24 25 26

생각해 보기

1. 2000-ary B-tree에서 리프 노드에 있는 key와 internal node에 있는 key 수의 비율은 어떤 정도의 느낌인가?

2. 1조 개의 key로 색인하는 B-tree는 레코드 접근 전에 디스크에 몇 번 정도 접근해야 하는가?

   가정: 4 bytes/key

   4 bytes/page# (page numbe를 적는데 4 bytes)

   32K bytes/page