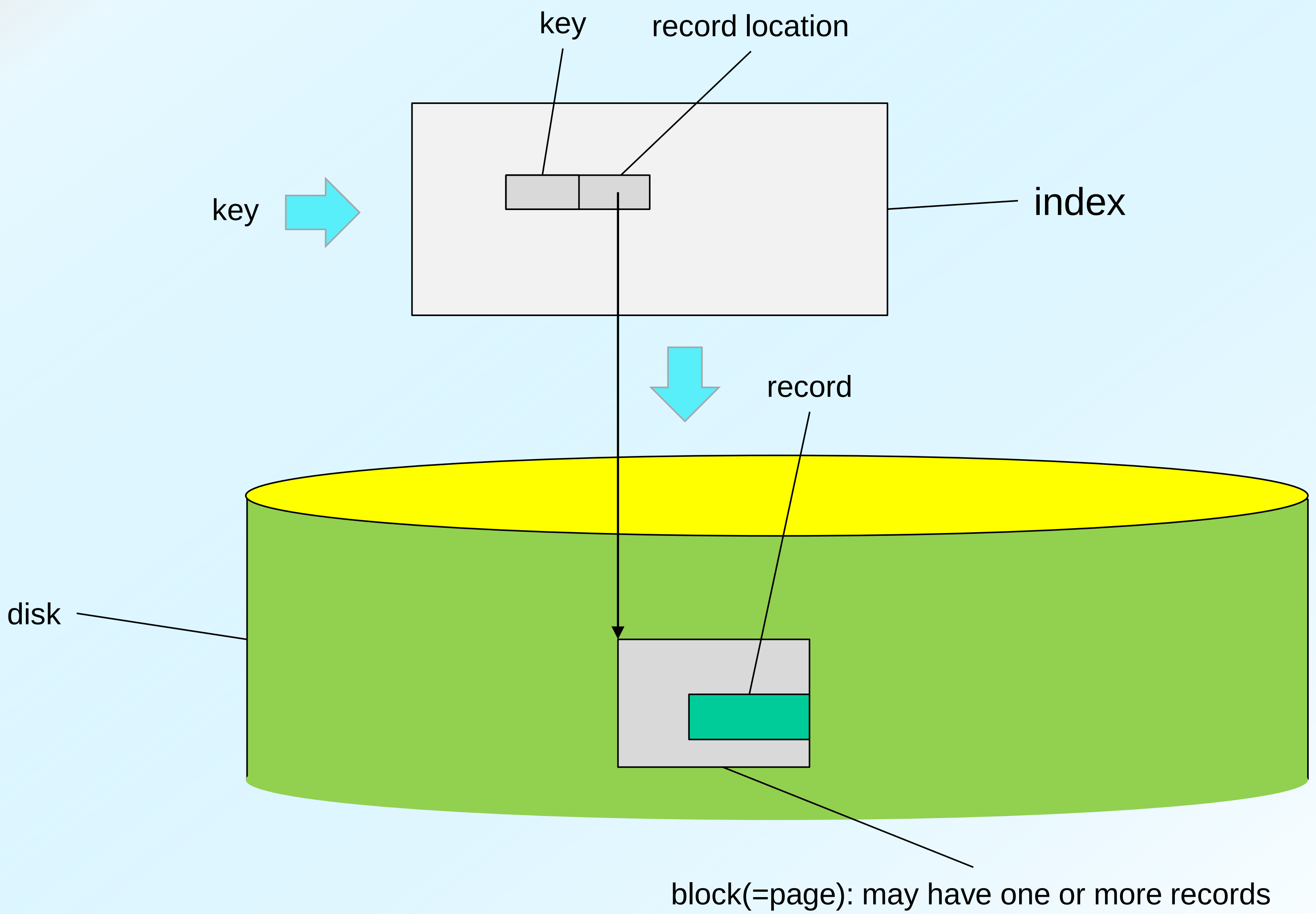
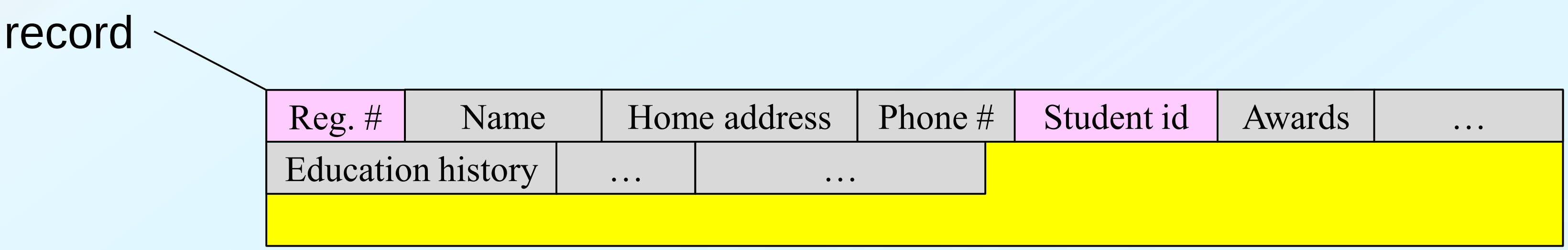
Binary Search Trees (BST)

Reminder:
Basics of Binary Search Trees

Index


key
record location
key
index
record
disk
block(=page): may have one or more records

Record

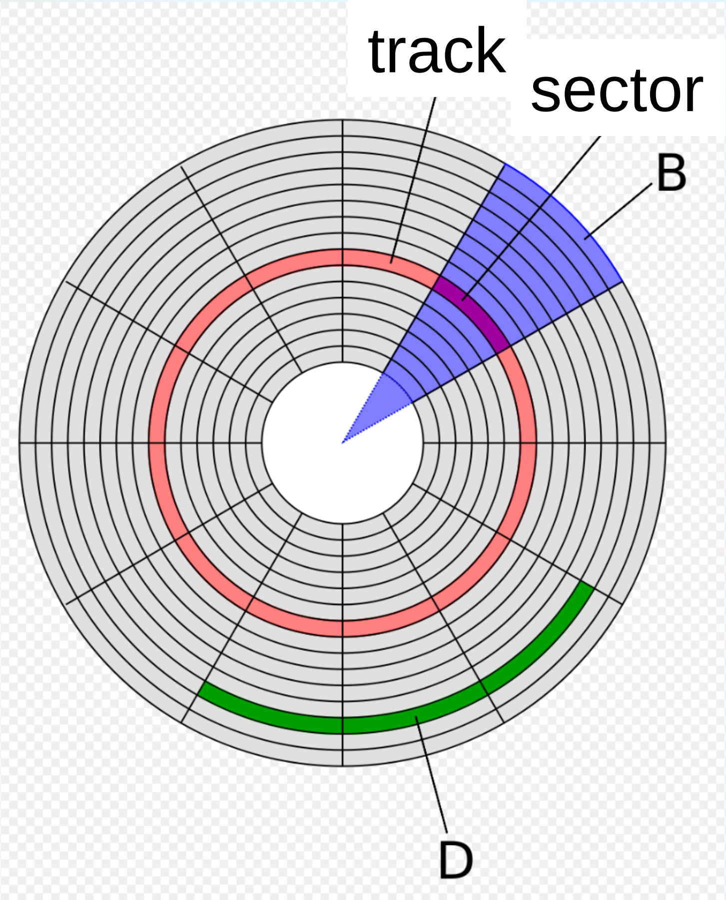


하나의 record는

한 block의 일부,
한 block 전체, 또는
여러 block

을 점유

Blocks



<그림. from Wikipedia>

Sector: 512, 4K, … bytes

Disk block: multiple of sectors

File block: multiple of disk blocks

* OS에서 file system block size 지정 가능
* 한 컴퓨터가 여러 file system을 가질 수 있다

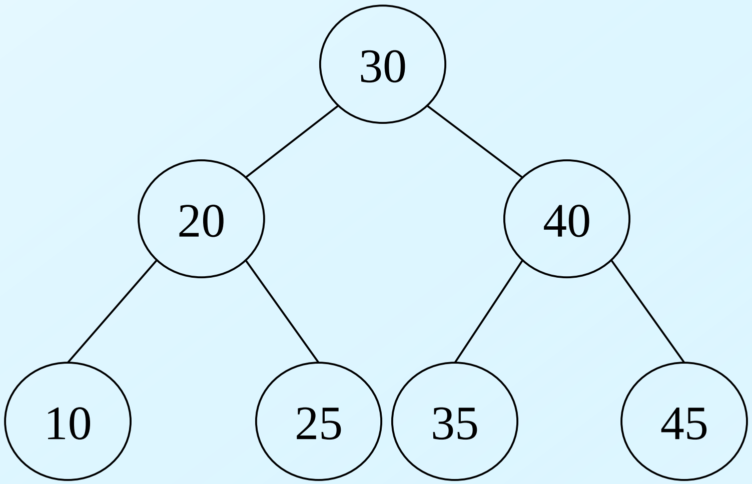
Record, Key, Search Tree

- A record
  - Contains all information for an object
  - e.g., human record
    - <resident registration #주민번호, name, home address, phone #s, education history, income, family members, … > ← a collection of such fields
- Field
  - each unit information in a record
  - e.g., in the above record, resident registration #, name, home address, …
- Search key or Key
  - A field which can uniquely identifies each record
  - A key may consist of a field or more
- Search tree
  - A key in each node
  - An index for keys and corresponding record locations

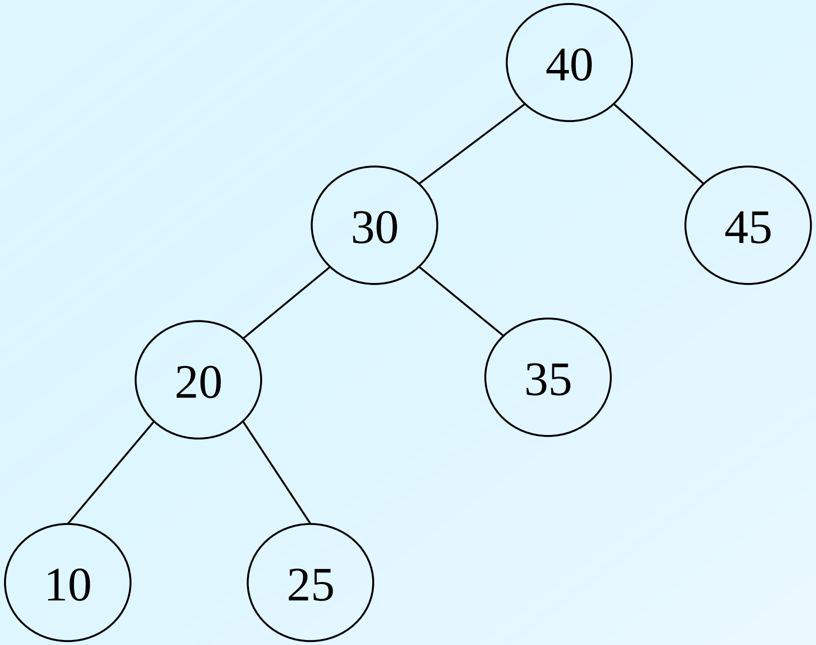
Binary Search Trees

- A key in each node. All keys are distinct.
- Root node in the top level
- Each node has at most two children.
- The key of a node is greater than all the keys of its left subtree, and smaller than all the keys of its right subtree.

Examples of Binary Search Trees

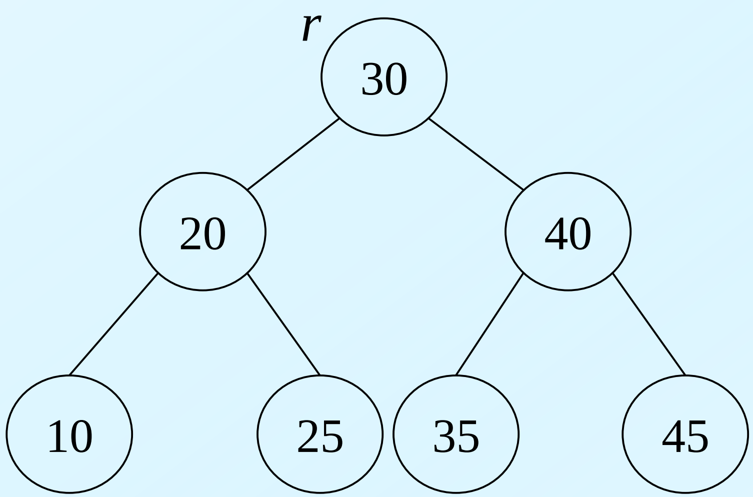


(a)

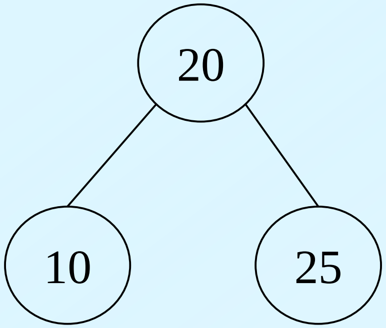


(b)

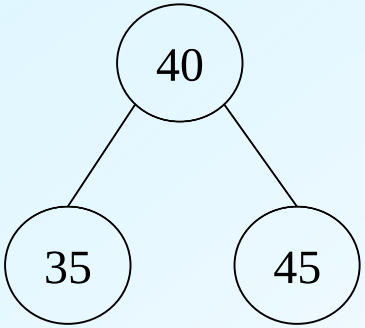
Examples of Subtrees



(a)



(b) Node *r*'s left subtree



(c) Node *r*'s right subtree

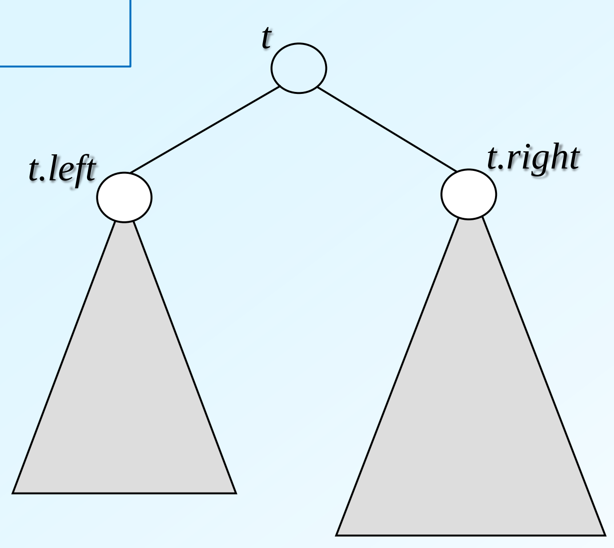
(Static) Optimal Binary Search Tree

Possible to construct optimal binary search tree.

See <Dynamic Programming>.

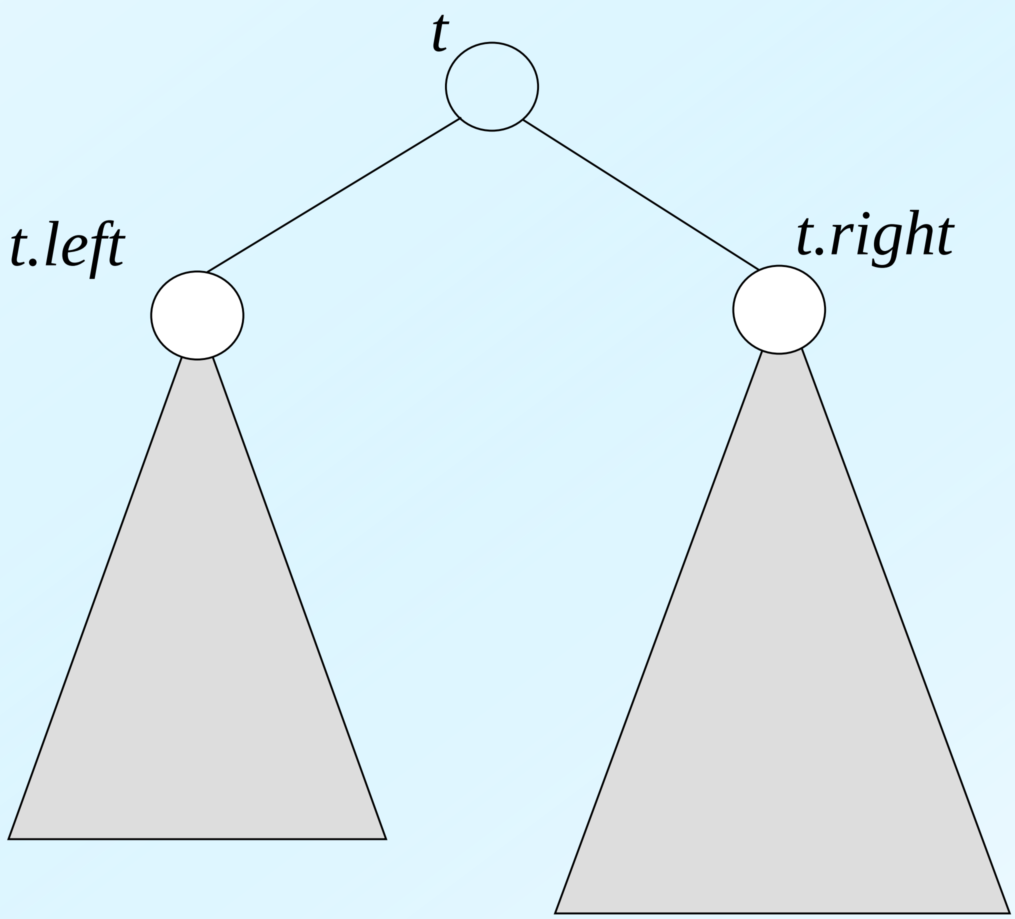
Reminder: Search in Binary Search Trees

```
search(t, x):
◀ t : root node, x : key
        if (t = NIL or t.key = x) then return t
        if (x < t.key)
                then return search(t.left, x)
                else return search(t.right, x)
```



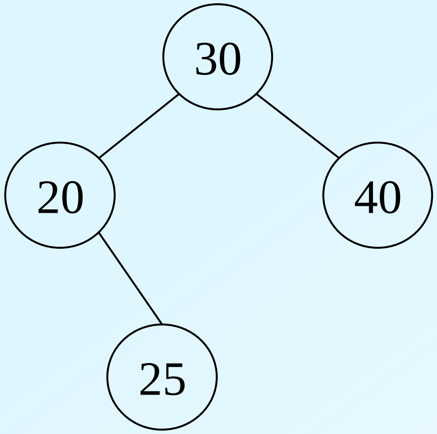
In real fields, a node contains the key field and the corresponding record location

Recursive View of Search

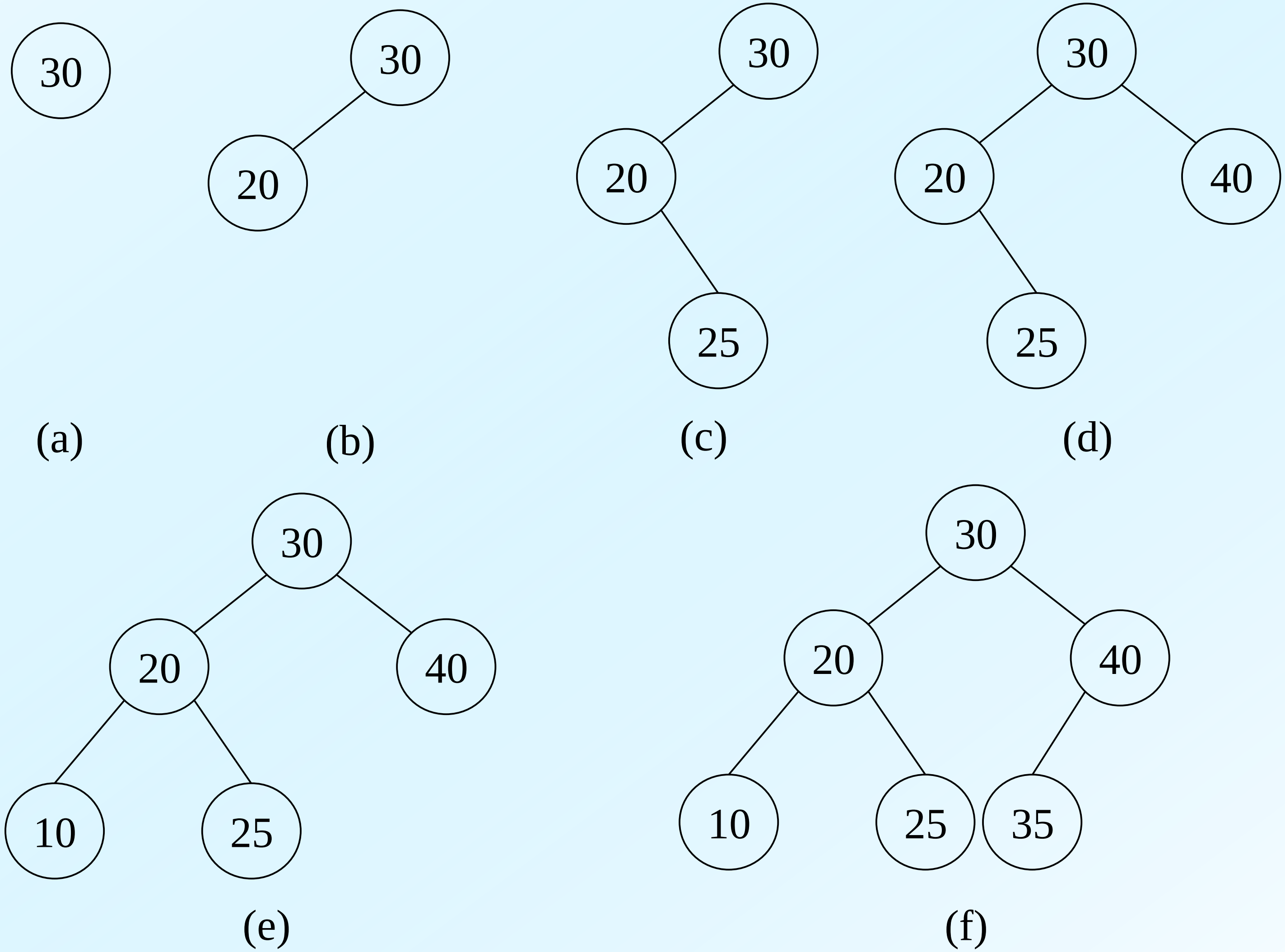


Reminder: Insertion in Binary Search Trees

```
insert(t, x):
◀ t : root node, x : key to insert
    if (t=NIL)
        r.key ← x            ◀ r: new node
        return r
    if (x < t.key)
        t.left ← insert(t.left, x)
        return t
    else t.right ← insert(t.right, x)
        return t
```



Examples of Insertion



Reminder: Deletion in Binary Search Trees

t: root node
r: node to deleted

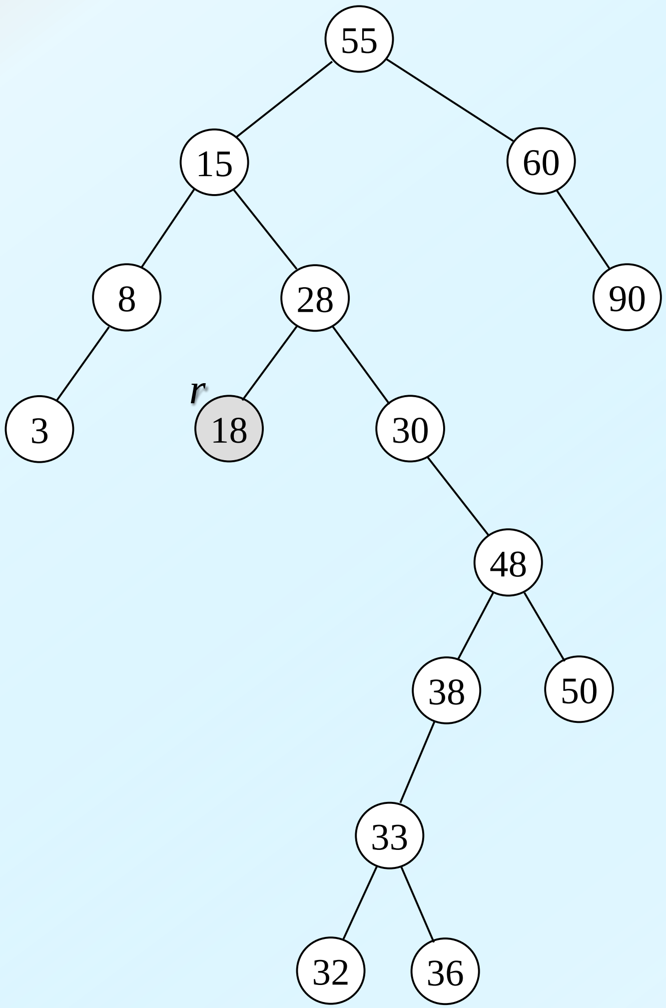
There are three cases

- Case 1 : *r* is a leaf node
- Case 2 : *r* has only one child
- Case 3 : *r* has two children

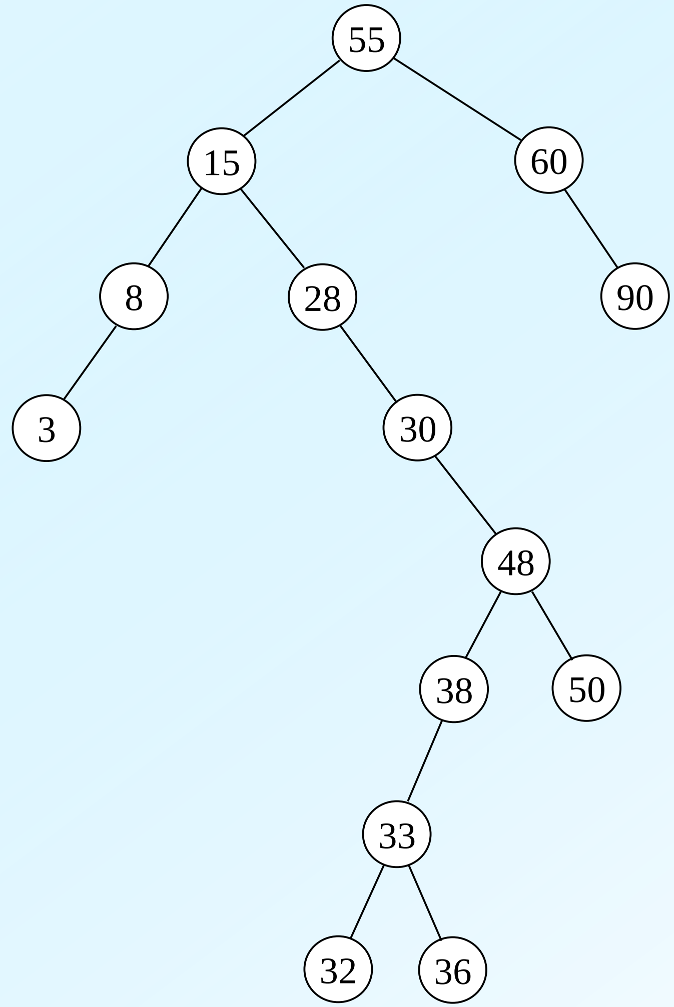
Reminder: Deletion in Binary Search Trees

sketch_delete(t, r):
◀ t : root node, r : node to delete
    if (r is a leaf node) ◀ Case 1
        Just throw away r
    else if (r has only one child) ◀ Case 2
        Let r's parent links to the (only) child of r
    else ◀ Case 3
        Remove the minimum node s of r's right subtree,
        and copy the key of s to node r

Example: Case 1

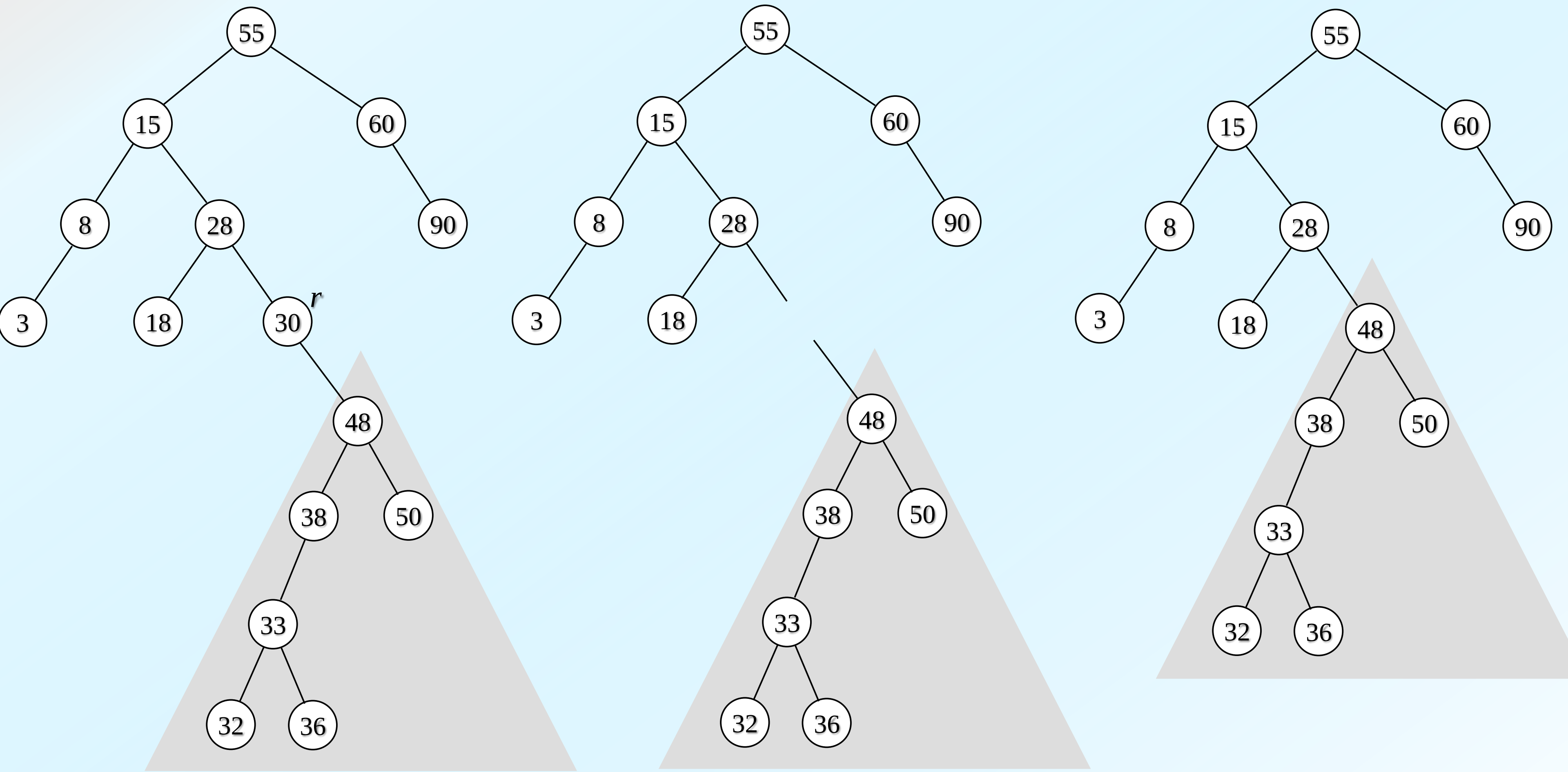


(a) *r* has no child



(b) Simply throw away *r*

Example: Case 2

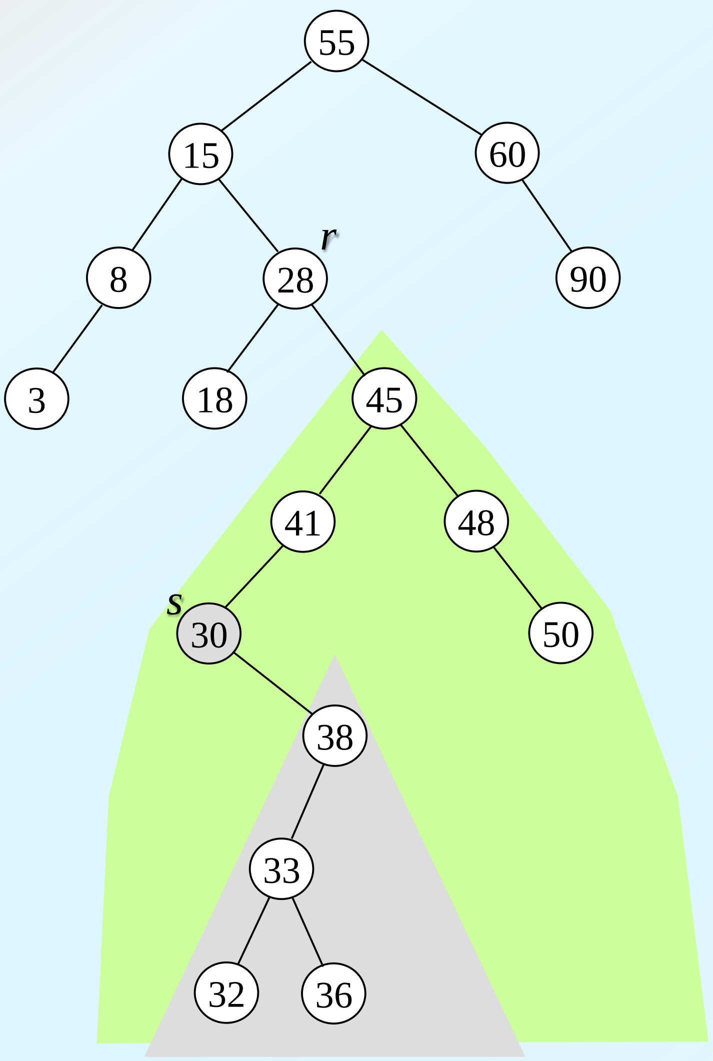


(a) *r* has only one child

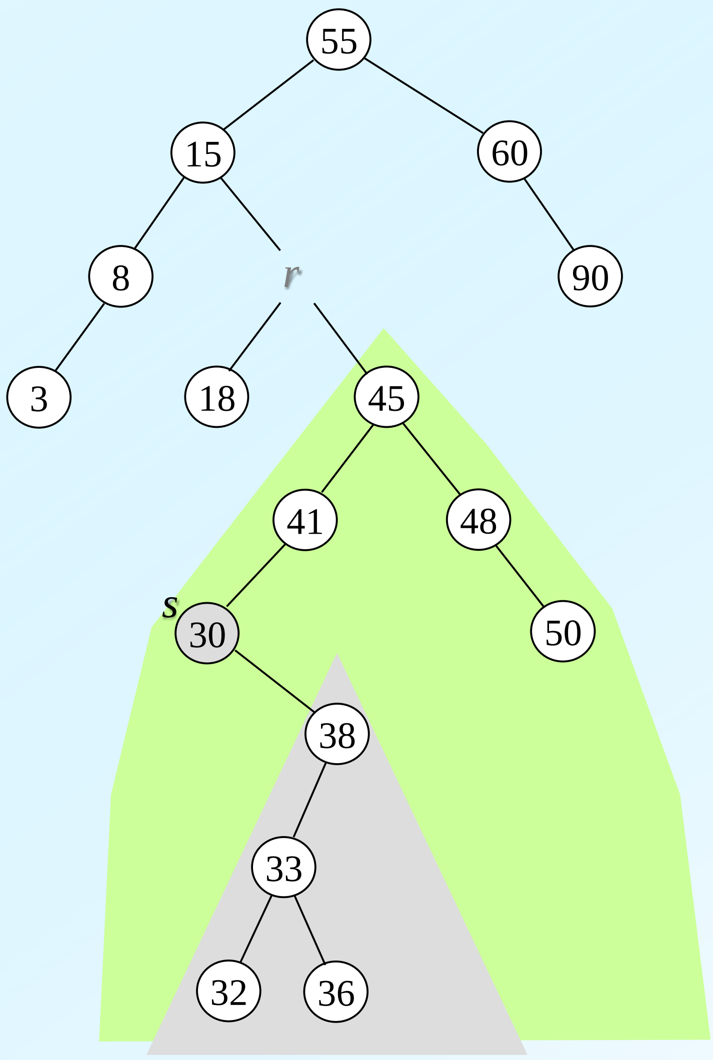
(b) Remove *r*

(c) Put *r*'s (only) child at *r*'s location

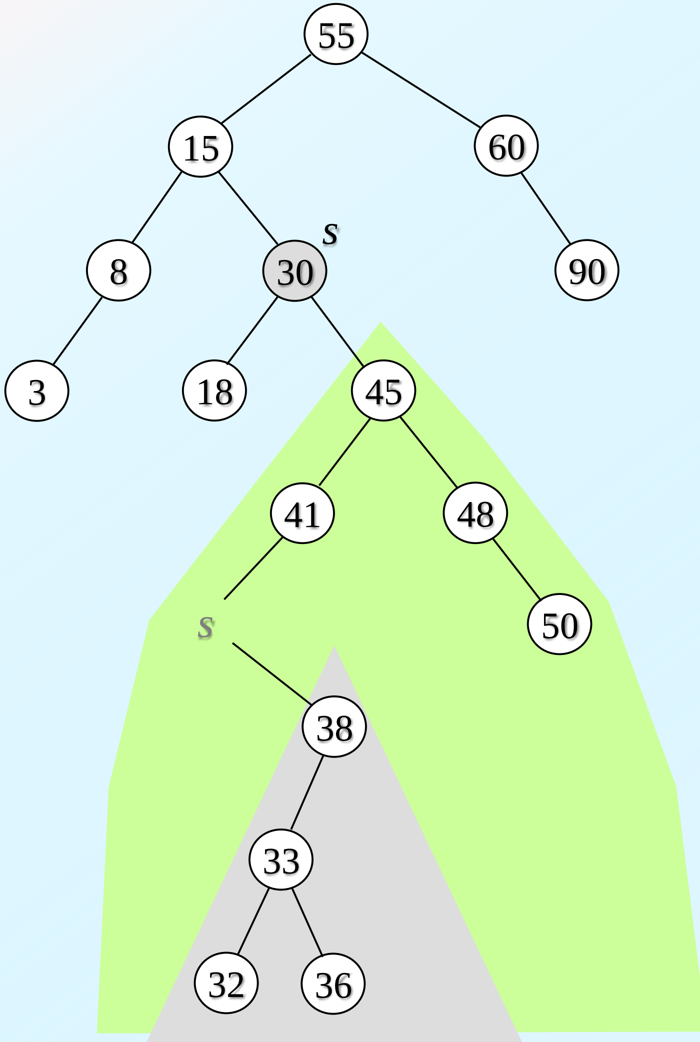
Example: Case 3



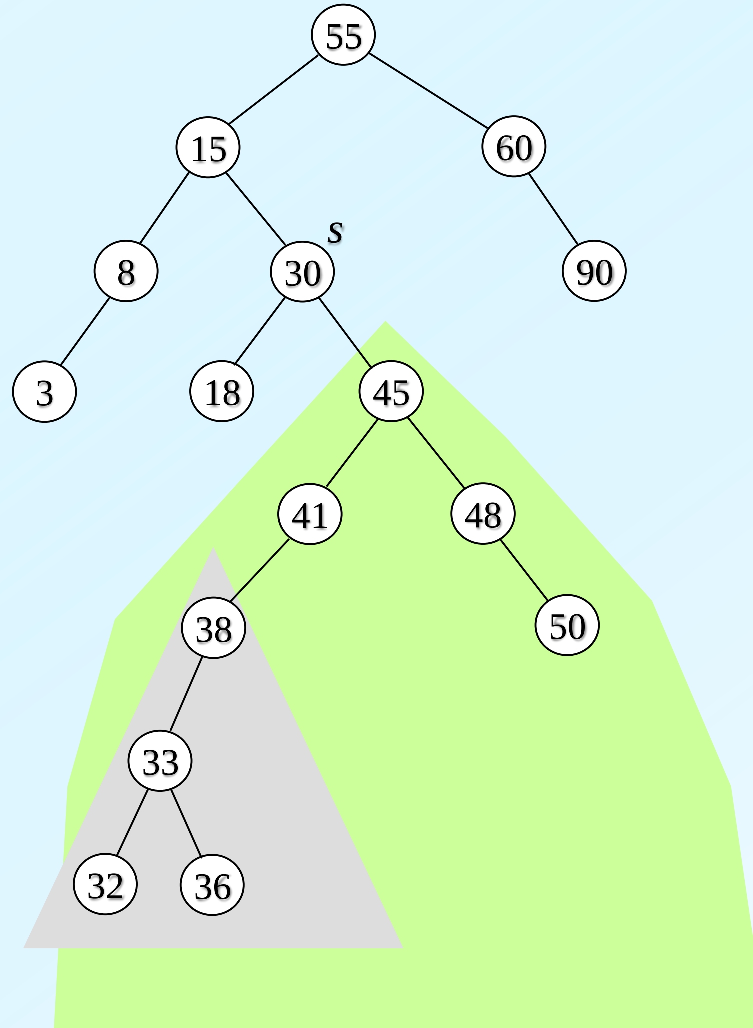
(a) Find *r*'s inorder successor *s*



(b) Remove *r* (imaginary removal)



(c) Move s to r's location
(copy s to r)

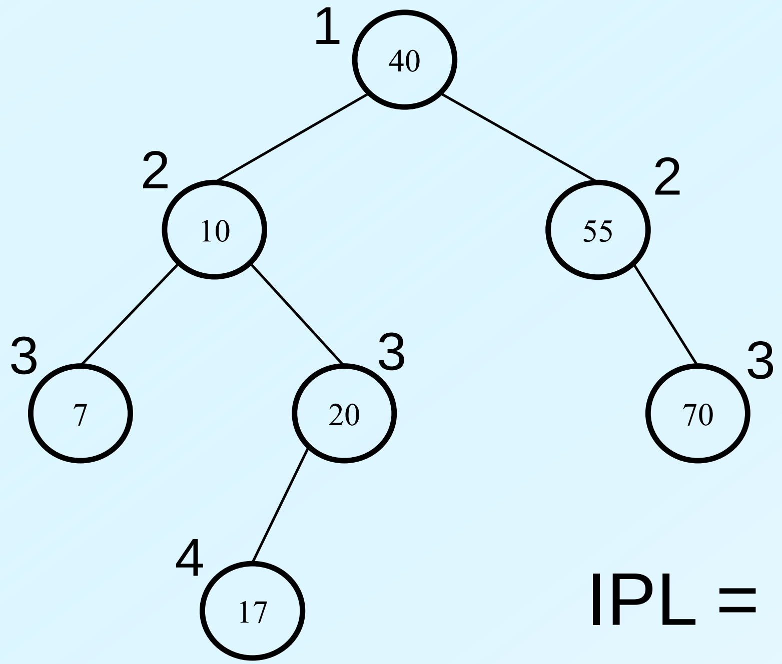


(d) Move s's (only) child to s's location

Efficiency of Binary Search Trees

IPL: Internal Path Length

Sum of depths of all the nodes



IPL = 18

Theorem

The IPL of a binary search tree made at random is $O(n \log n)$ on average (Assume every permutation of the input sequence is equally likely)

Equivalently, a sequence of n inserts into an empty binary search tree takes $O(n \log n)$ on average (Assume every permutation of the input sequence is equally likely)

<Proof>

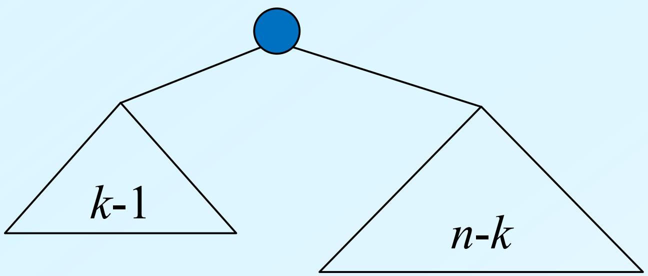
Next page

✓ Meaning: Average search time for a key is $O(\log n)$.

<Proof>

$D(n)$: the average IPL(Internal Path Length) of a binary tree with n nodes.

Clearly $D(0) = 0$, $D(1)=1$.



$$D(n) = \frac{1}{n}\sum_{k=1}^{n}[D(k-1) + (k-1) + D(n-k) + (n-k) + 1]$$

$$= \frac{2}{n}\sum_{k=0}^{n-1} D(k) + n$$

Assume that $\exists c > 0$ s.t. $D(k) \leq ck\log k \; \forall k < n$.

Then, we verify that $D(n) \leq cn\log n$ (i.e., $D(n) = O(n\log n)$)

$$
\begin{aligned}
D(n) &= \frac{2}{n}\sum_{k=0}^{n-1} D(k) + n \\
&= \frac{2}{n}\sum_{k=2}^{n-1} D(k) + \Theta(n) \qquad D(0),\, D(1) \text{ absorbed} \\
&\le \frac{2}{n}\sum_{k=2}^{n-1} ck\log k + \Theta(n) \\
&\le \frac{2}{n}\int_{1}^{n} cx\log x \, dx + \Theta(n) \\
&= \frac{2c}{n}\left(\left[\frac{1}{2}x^2\log x\right]_1^n - \left[\frac{1}{4}x^2\right]_1^n\right) + \Theta(n)
\end{aligned}
$$

$$= \frac{2c}{n}\left(\frac{1}{2}n^2\log n - \frac{1}{4}n^2 + \frac{1}{4}\right) + \Theta(n)$$

$$= cn\log n - \frac{cn}{2} + \frac{c}{2n} + \Theta(n)$$

absorbed

$$= cn\log n - \frac{cn}{2} + \Theta(n)$$

$$\le cn\log n$$

We can choose $c > 0$ s.t. $\frac{cn}{2}$ dominates $\Theta(n)$

$$\therefore D(n) = O(n\log n)$$

Balanced Binary Search Trees

Reminder: AVL Tree

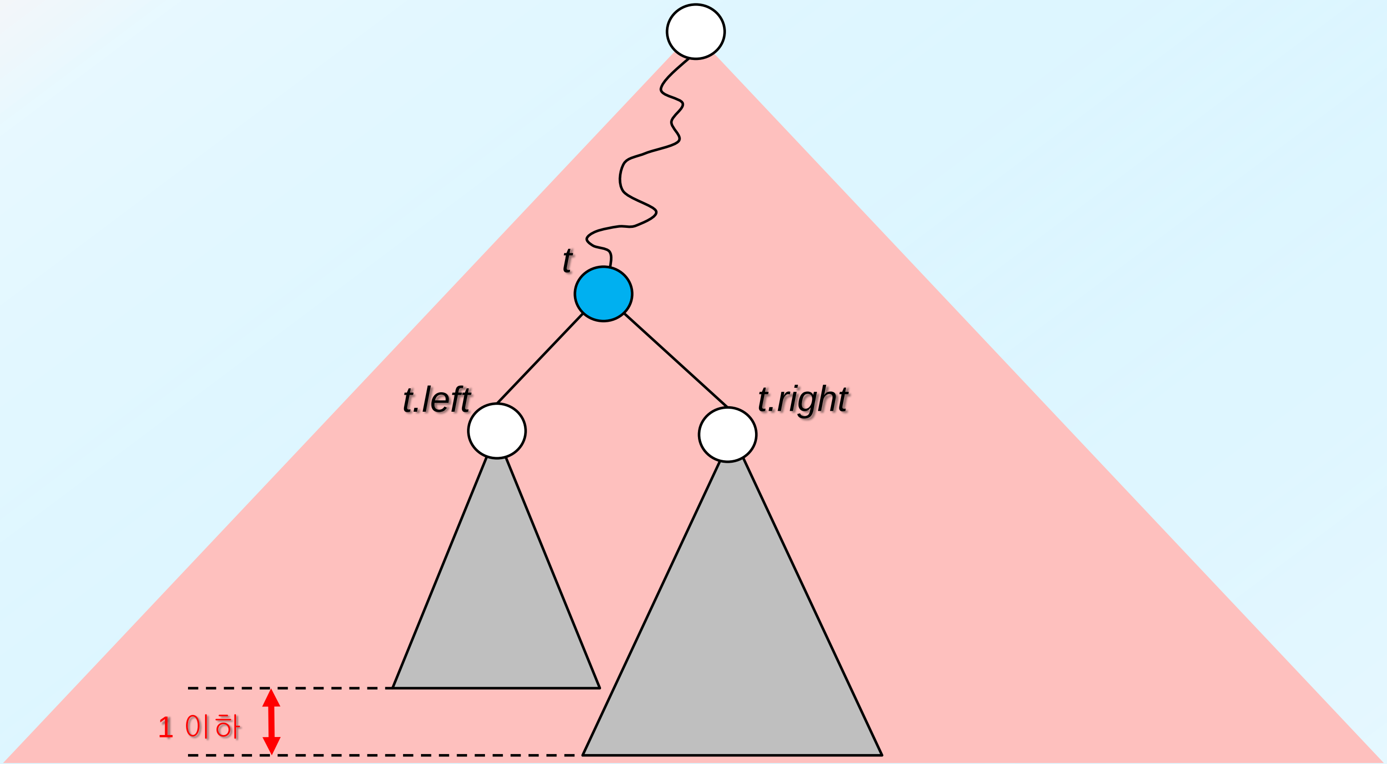
Devised by Adelson-Velskii and Landis

A balanced search tree

such that

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

differ by at most 1

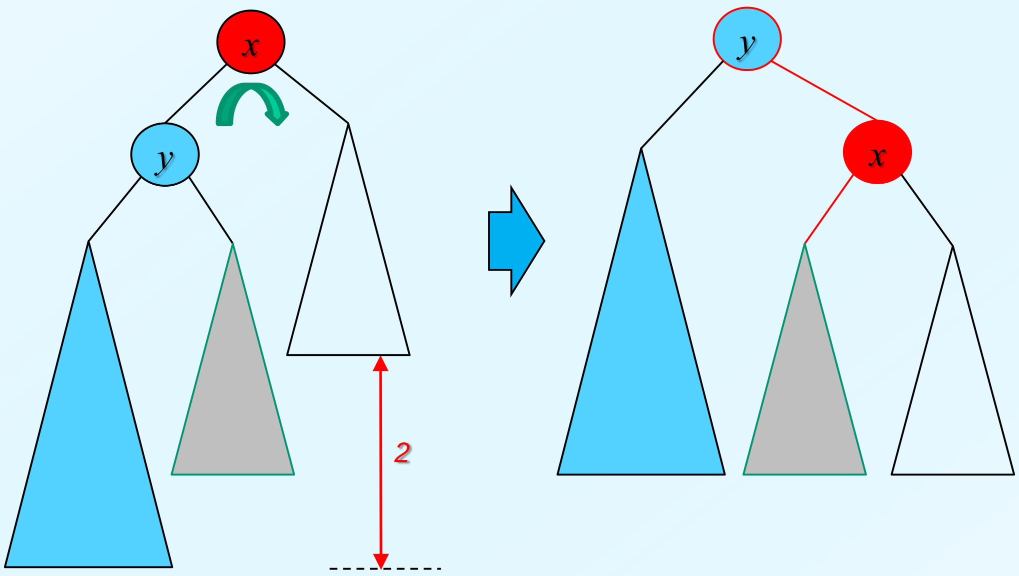


AVL Tree

Covered in <Data Structures>

Four Types of Repairs

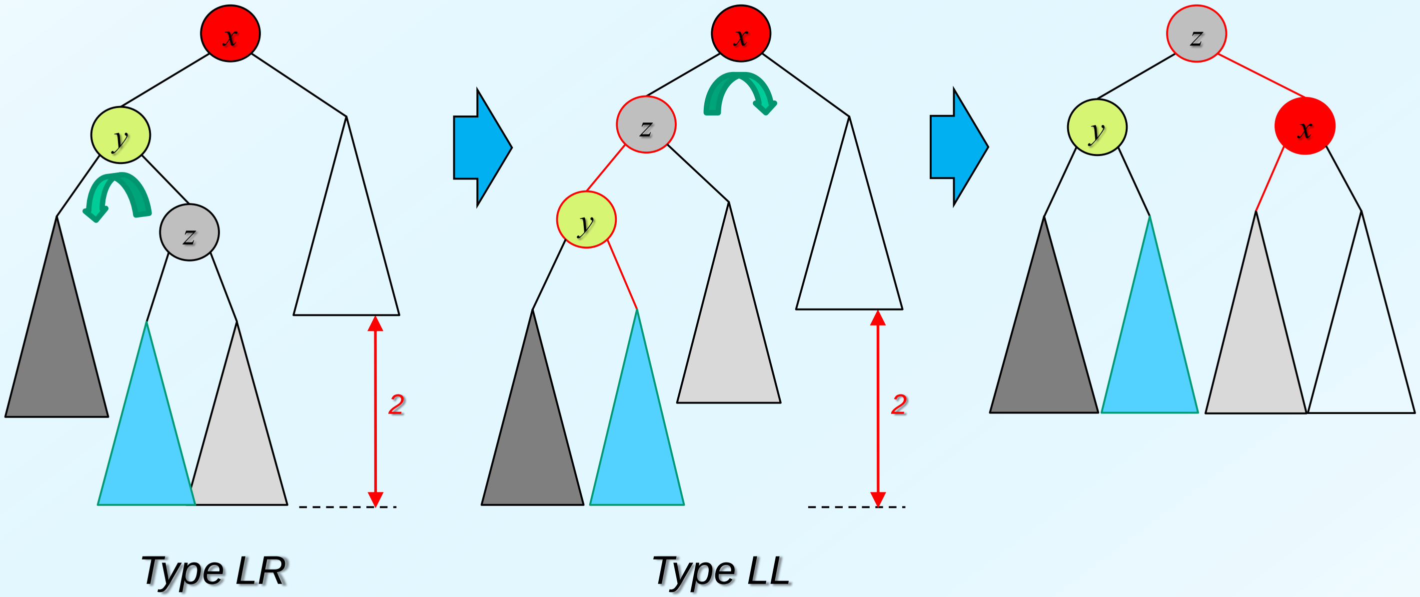
1. **Type LL: Right rotation**



Four Types of Repairs

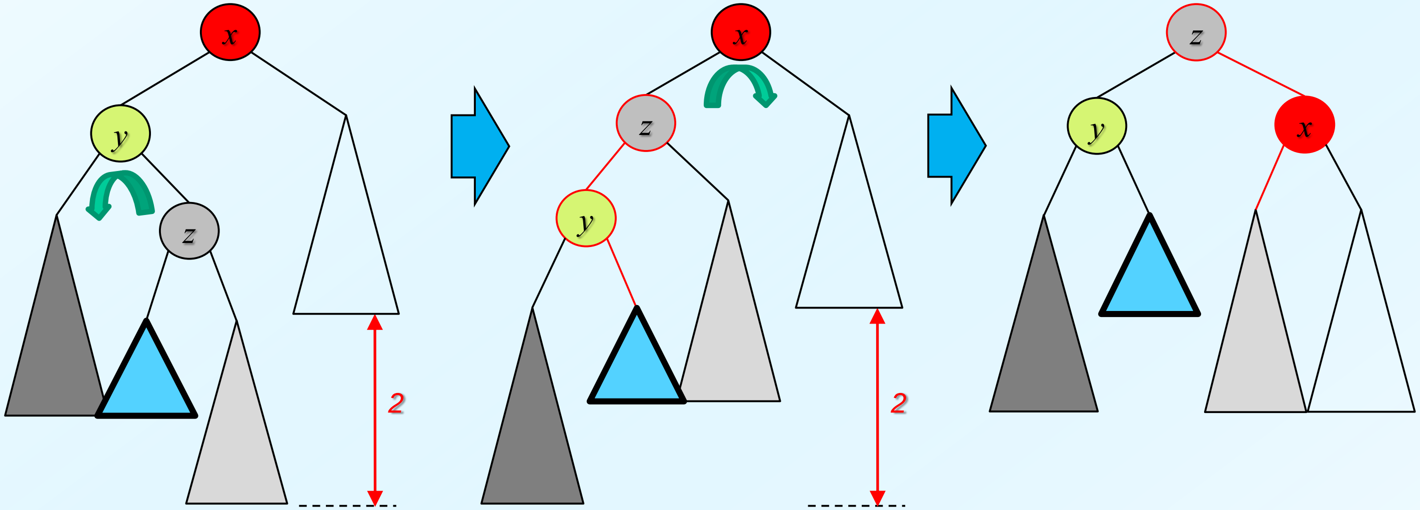
2. Type LR: <u>Left rotation</u> then right rotation

(conversion to type LL)



Four Types of Repairs

Another Instance of Type LR

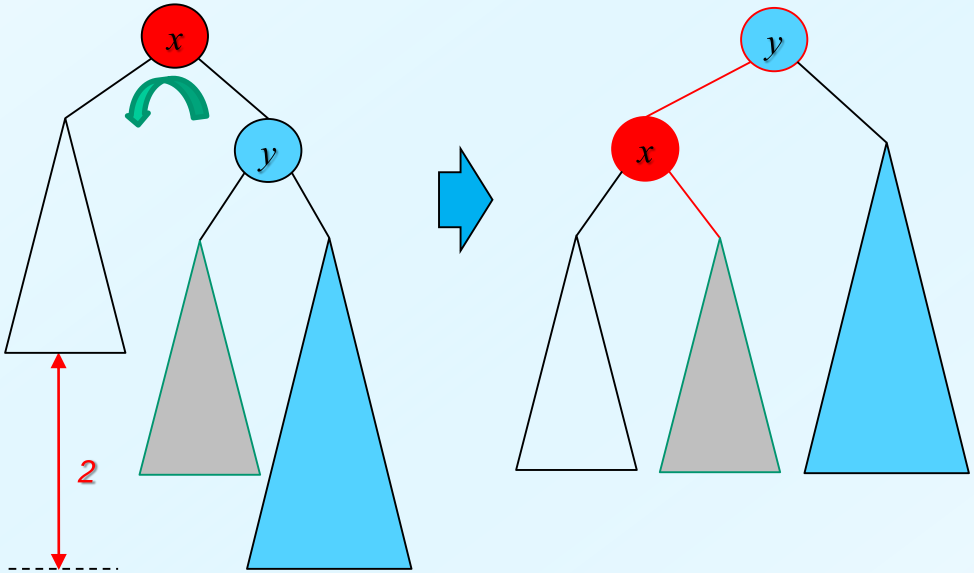


Type LR

Type LL

Four Types of Repairs

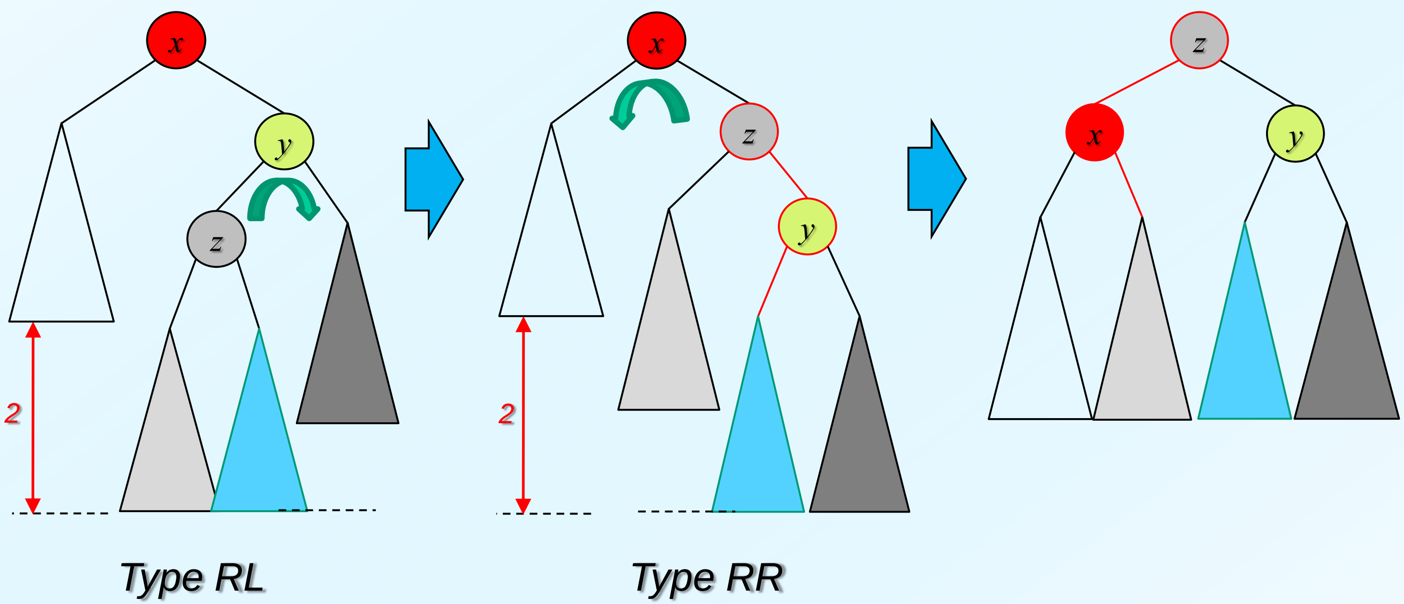
3. Type RR: Left rotation



Four Types of Repairs

4. Type RL: <u>Right rotation</u> then left rotation
(conversion to type RR)

* LL과 RR, LR과 RL은
  각각 symmetric



Red-Black Trees

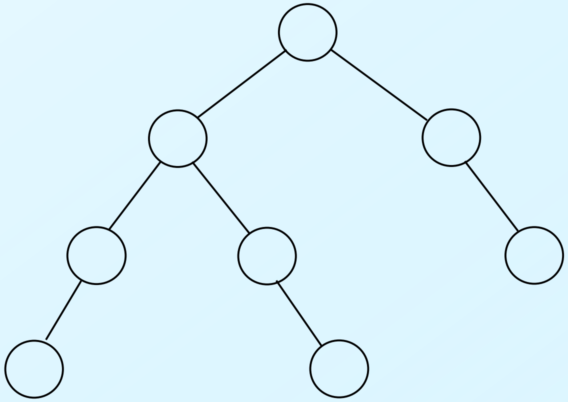
- Every node in the search tree has a color: red or black.
- It has to satisfy the following properties
  (**red-black** properties = **RB** properties):

① Every leaf is **black**
② If a node is **red**, its children should be **black** (no two consecutive **red**s)
③ In any path from the root to a leaf, the # of **black** nodes on the path is the same (**black** height)

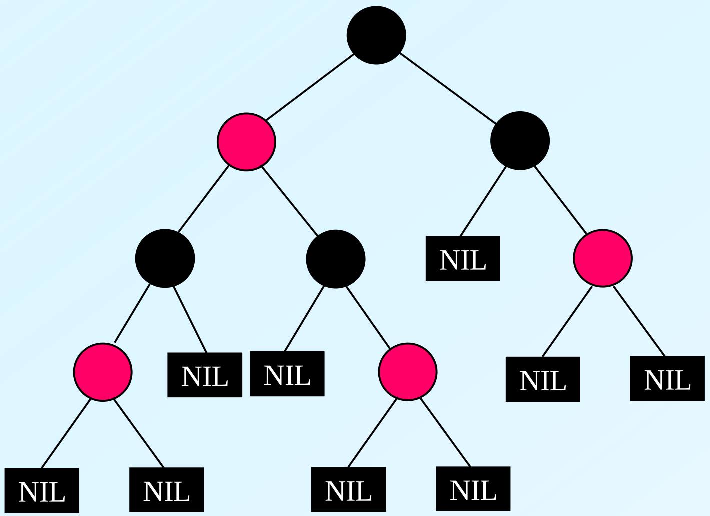
✓ Here, a leaf is not a general leaf node.
Every **null** reference links to the **NIL** leaf node(sentinel).

✓ 보통은 root가 **black**이라는 성질이 포함되는데 제외해도 별 문제 없음

BST to RB Tree

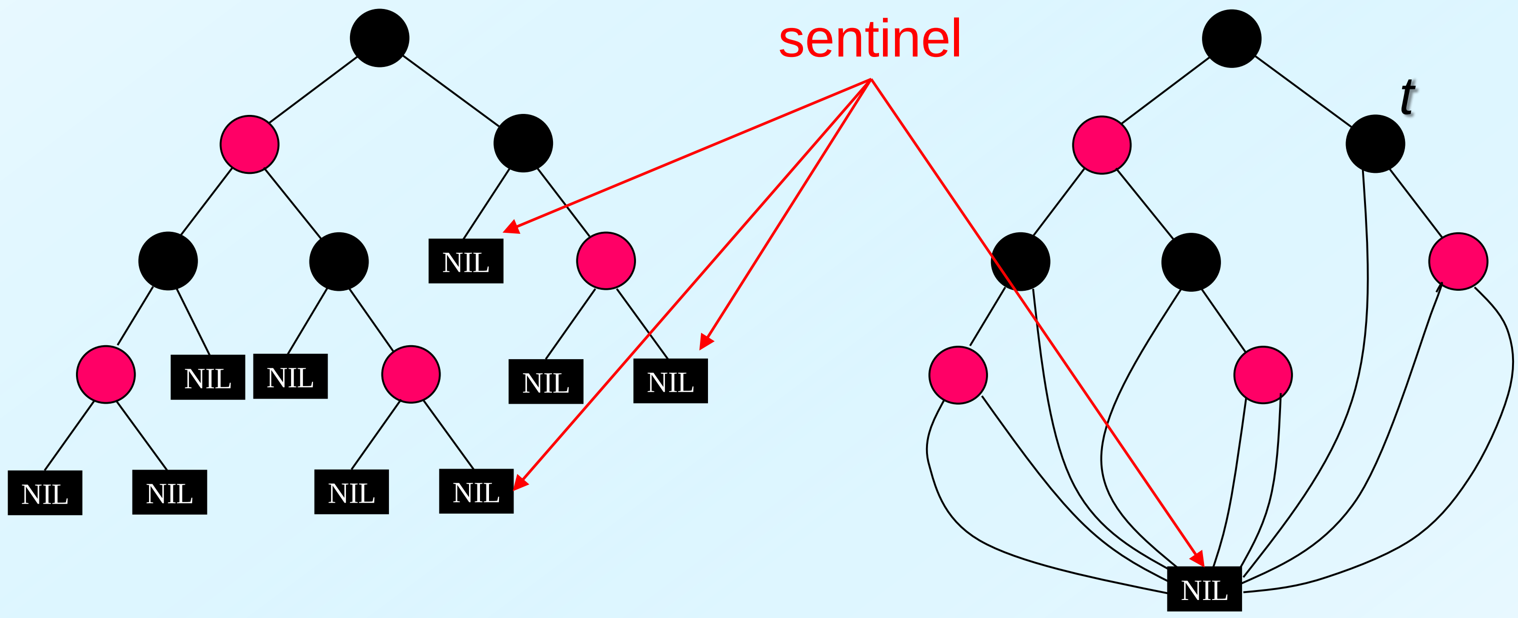


(a) An example binary search tree



(b) A red-black counterpart of (a)

Sentinel: An Imaginary Leaf Node



(b) A red-black counterpart of (a)

(c) Implementation of (b)

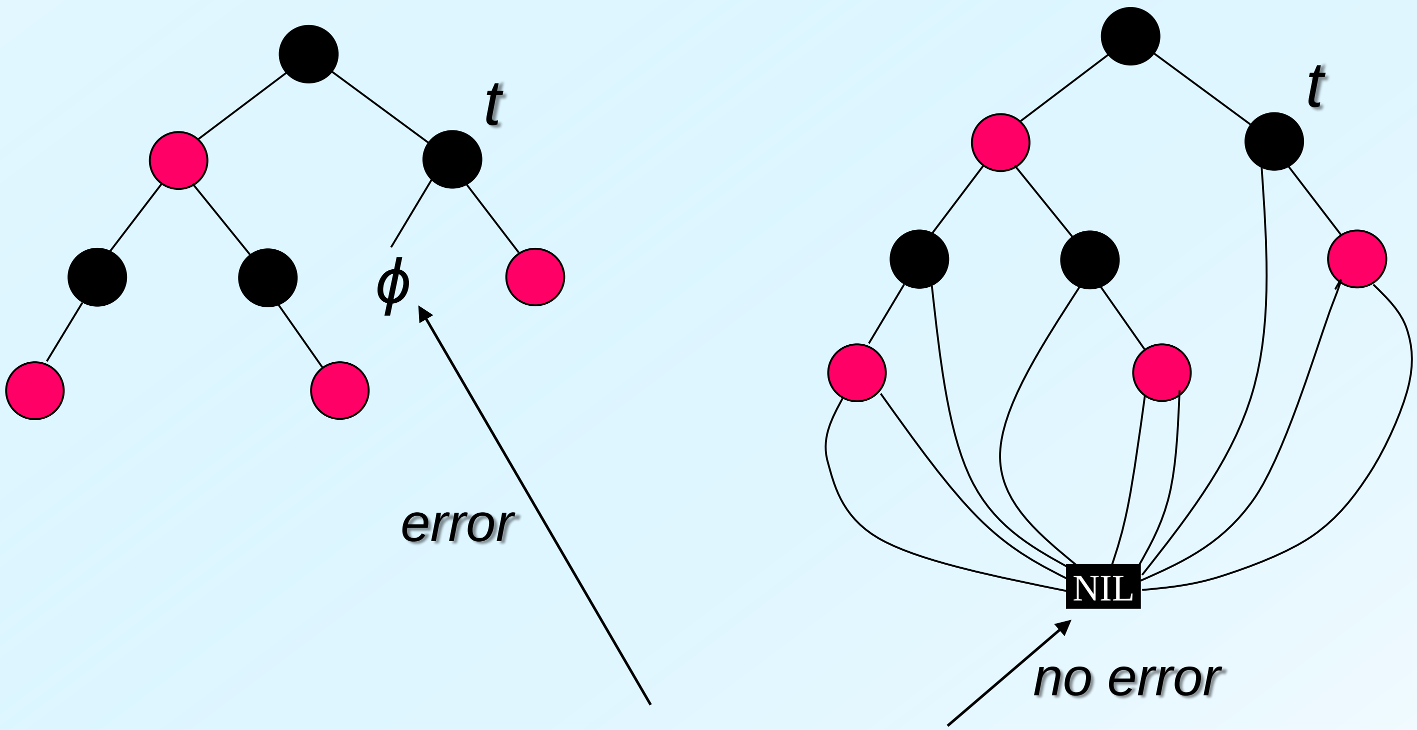
✓ NIL leaf는 구현상 매우 유용하다

Sentinel is Useful

```
class TreeNode {
        … key;
        TreeNode left;
        TreeNode right;
        int color;
        …
}
```

Programming example:

NIL is a **black**-colored TreeNode object



if (t.left.color == black) …

Theorem

* black height: 루트에서 리프 노드에 이르는 경로상에서
만나는 블랙 노드의 개수(루트는 제외)

Theorem

키가 총 *n*개인 RB 트리의 가능한 최대 깊이는 O(log n)이다

<Proof>

키의 총 수가 $n \rightarrow$ internal node의 수가 n.

$\rightarrow$ 가장 이상적으로 균형잡힌 이진검색트리의 깊이는 $\lfloor \log_2 n \rfloor + 1$.

$\rightarrow$ RB 트리가 이상적으로 만들어져도 black height는 $\lfloor \log_2 n \rfloor + 1$를 넘을 수 없다.

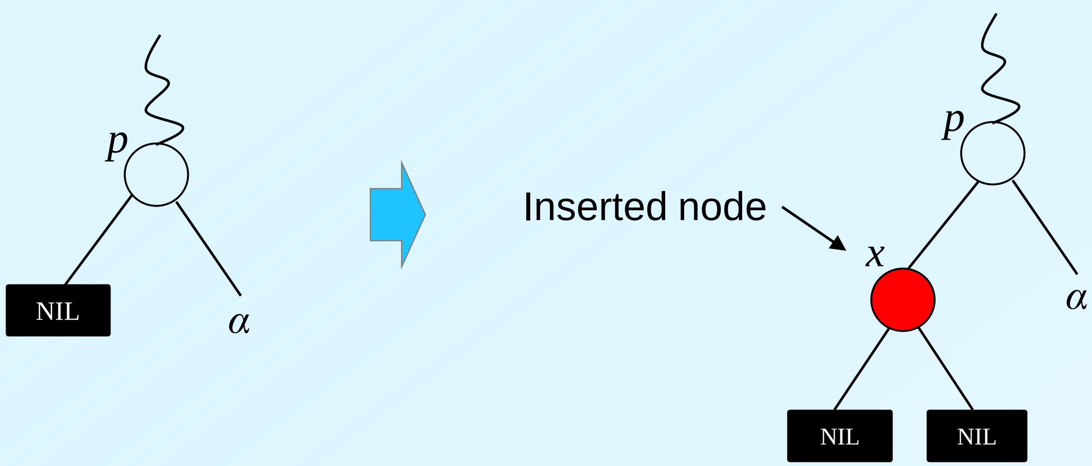
RB property ②에 의해,
루트에서 리프에 이르는 경로 상에서 레드 노드가 블랙 노드보다 많을 수 없다

RB 트리의 internal node의 경로 길이는 $2(\lfloor \log_2 n \rfloor + 1)$를 넘을 수 없다

이것은 $O(\log n)$이다.

Insertion

Do general BST insertion,
color **red** to the inserted node x,
and link two NIL leaves from x

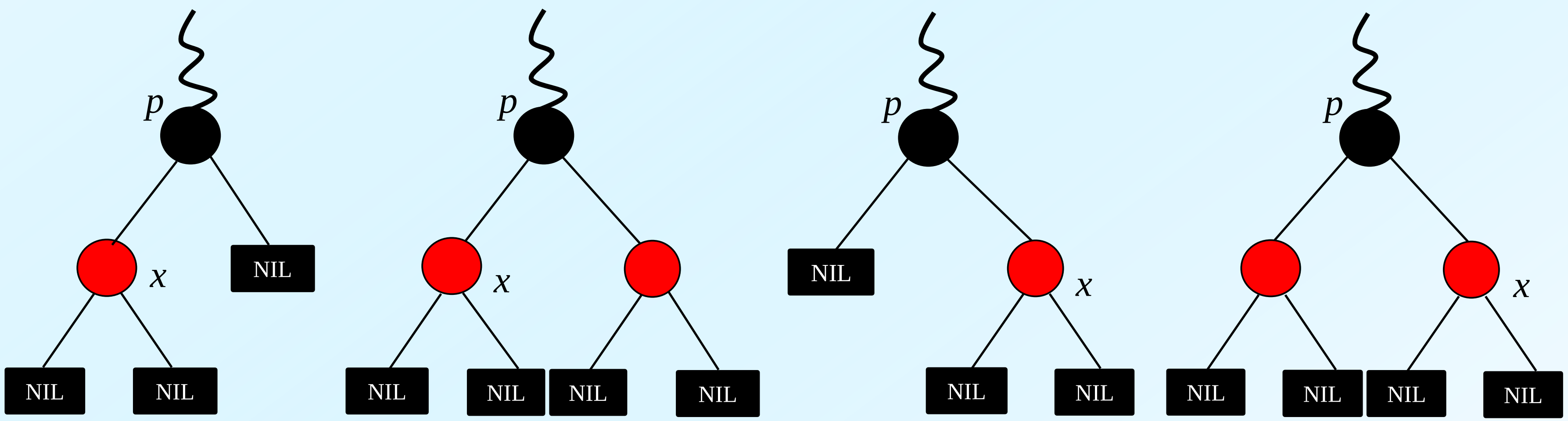


situation after insertion:
p is **black** or **red**

Insertion

Situation after insertion: p is **black** or **red**

1. If p is **black**: satisfies all the RB properties. Completed!



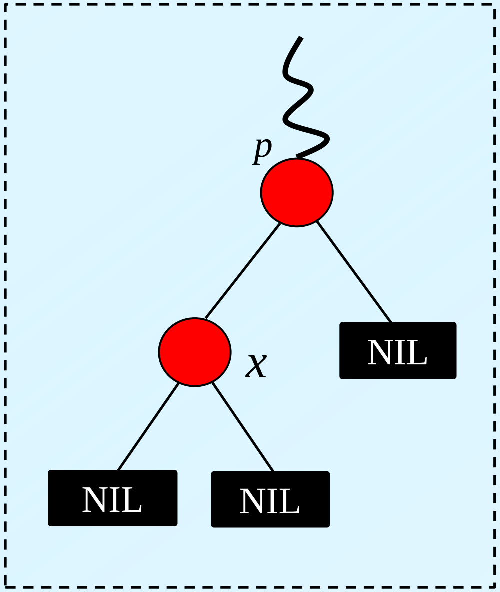
There are only these four cases

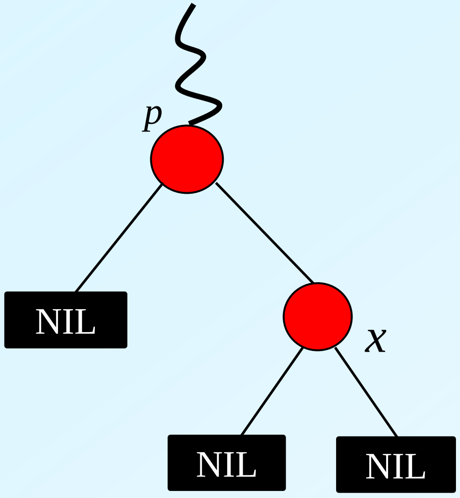
Insertion

2. If p is **red**: RB property ② is violated

- If p is the root, change p to **black**. Completed!
- otherwise, repair (next page..)

root가 **black**이라는
제약을 없앤 뒷수습



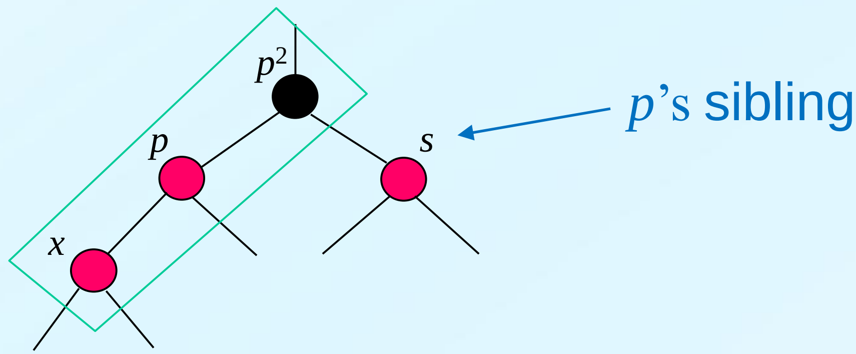


There are only these two cases.
They are symmetric. Here I show just the left case.

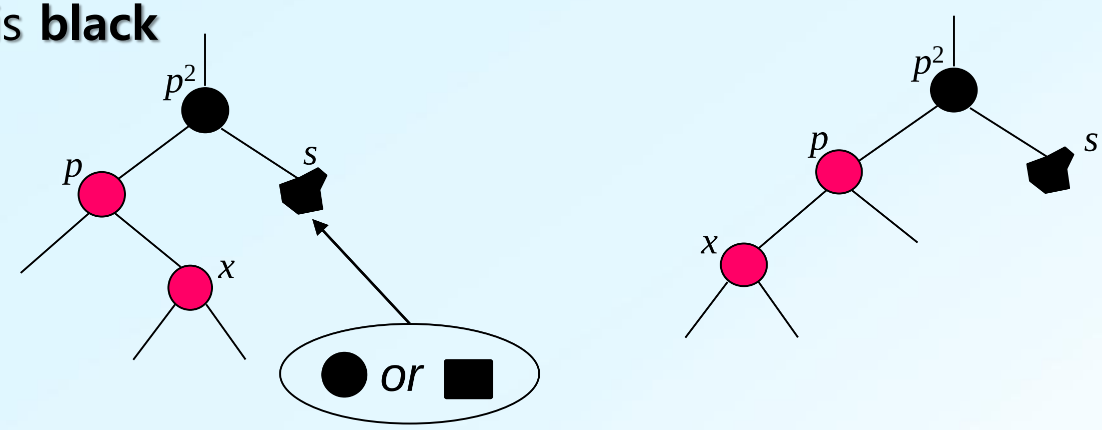
Insertion

Two cases depending on p's sibling s

Case 1: s is **red**

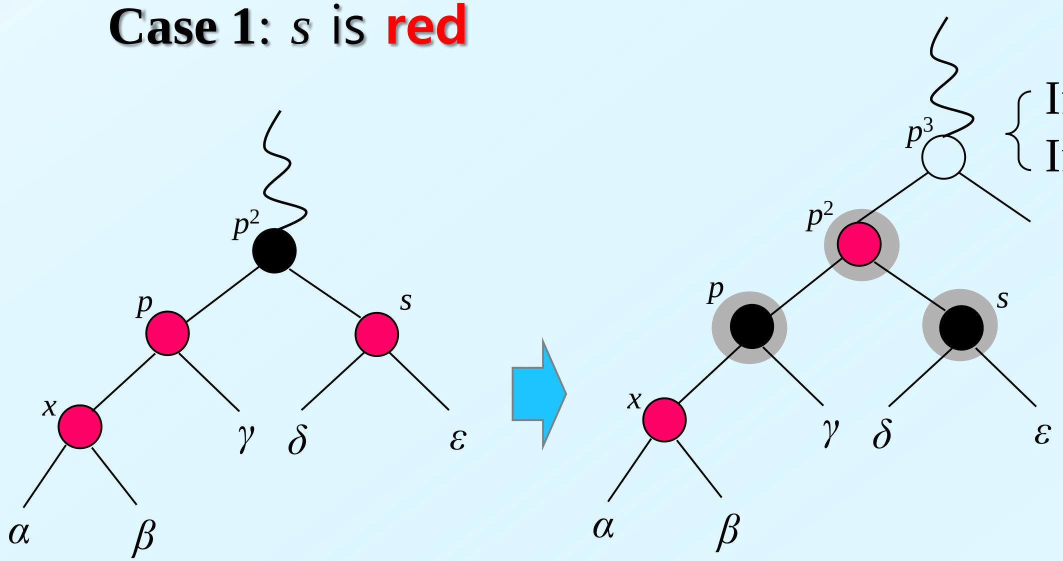


Case 2: s is **black**



Insertion

Case 1: s is **red**



If p^3 is **black**, completed
If p^3 is **red**, p^2 becomes new x: Recursive!

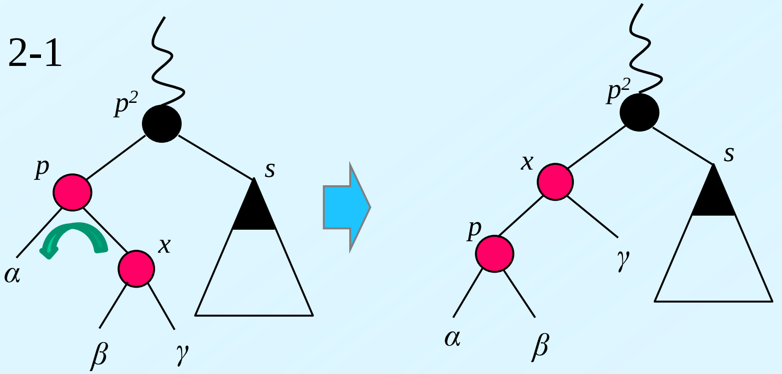
: 색이 바뀐 노드

Change p and s to **black**, p^2 to **red**.

Insertion

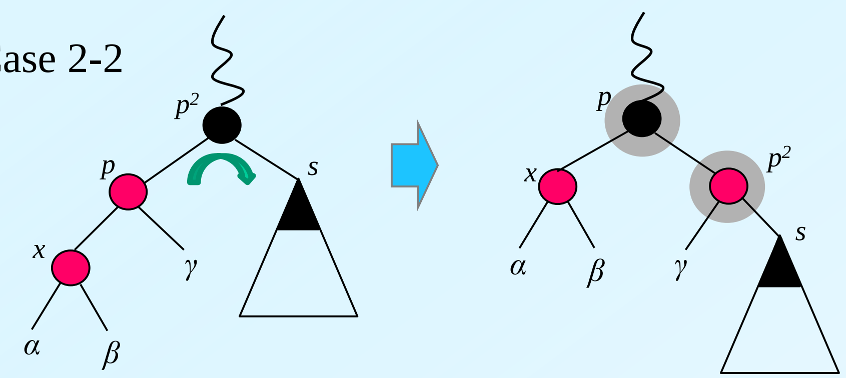
Case 2: s is **black**

Case 2-1



✓ Case 2-2로

Case 2-2



✓ completed

Running Time

Case 1: $O(\log n)$

Case 2: $\Theta(1)$

$\longrightarrow$ $O(\log n)$ in total (considered only repairing)

Deletion

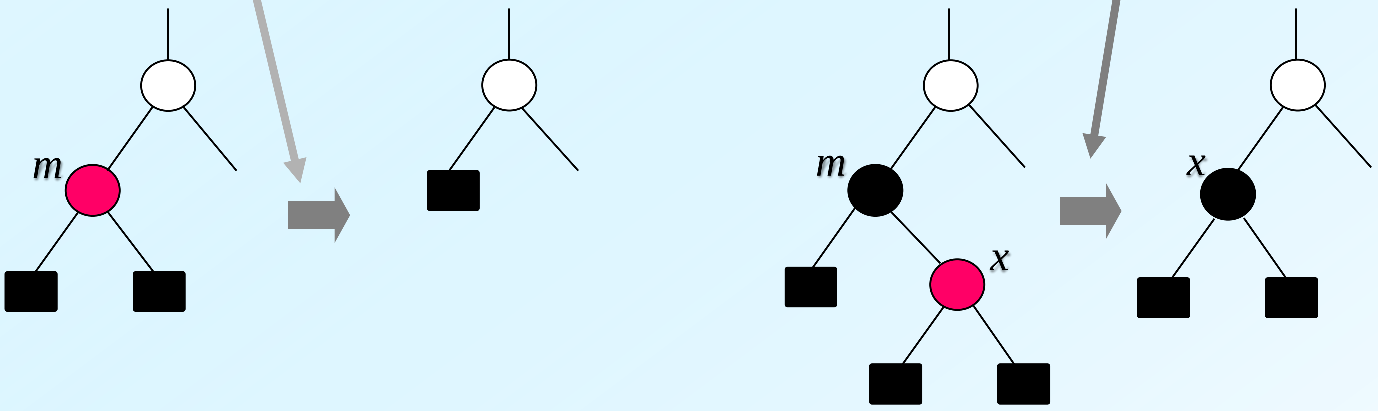
- We can **restrict to the cases** that the deleted node has

  no child or only one child

  – reason: <쉽게 배우는 알고리즘>(p.174) or

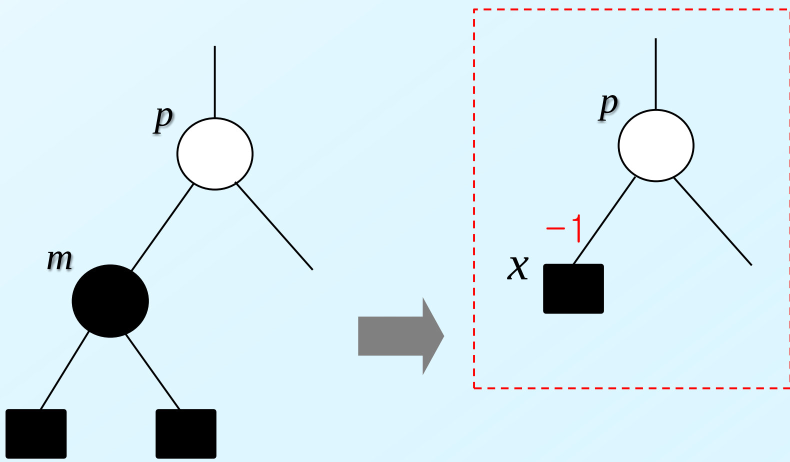
  <Introduction to Algorithms>(p.289)

  – *m*: 삭제될 노드
- If *m* is red: no problem! (*m* has no child)
- Even when *m* is black, no problem if *m* has a child(반드시 red다)!



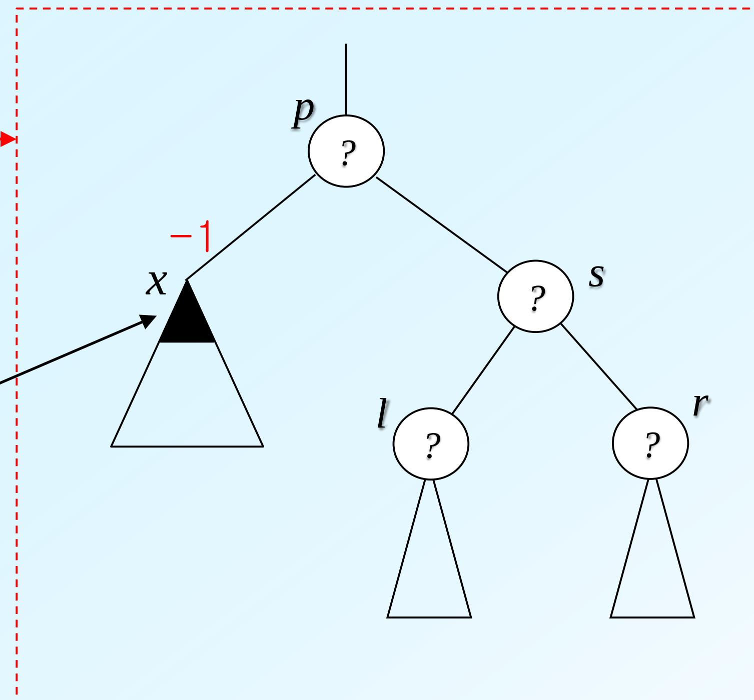
Problematic Case

Problem occurs only when the black *m* has no child.



RB property ③ violation after deleting *m*

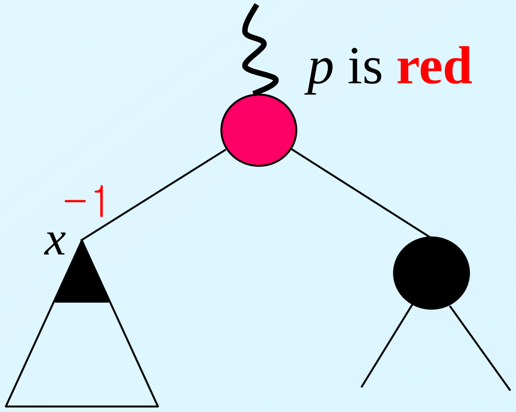
✓ -1: lack of one **black** node in the path from the root to *x*



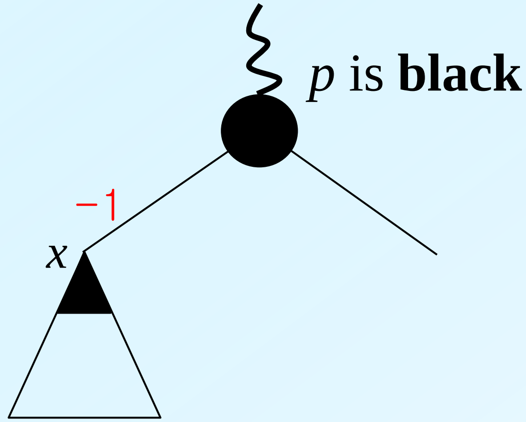
최초에는 왼쪽 박스와 같은 NIL leaf 밖에 가능하지 않으나
수선 과정에서 recursive하게 위로 올라가면서
이런 tree가 될 수도 있다

Depends on colors around *x*

Classification of Cases

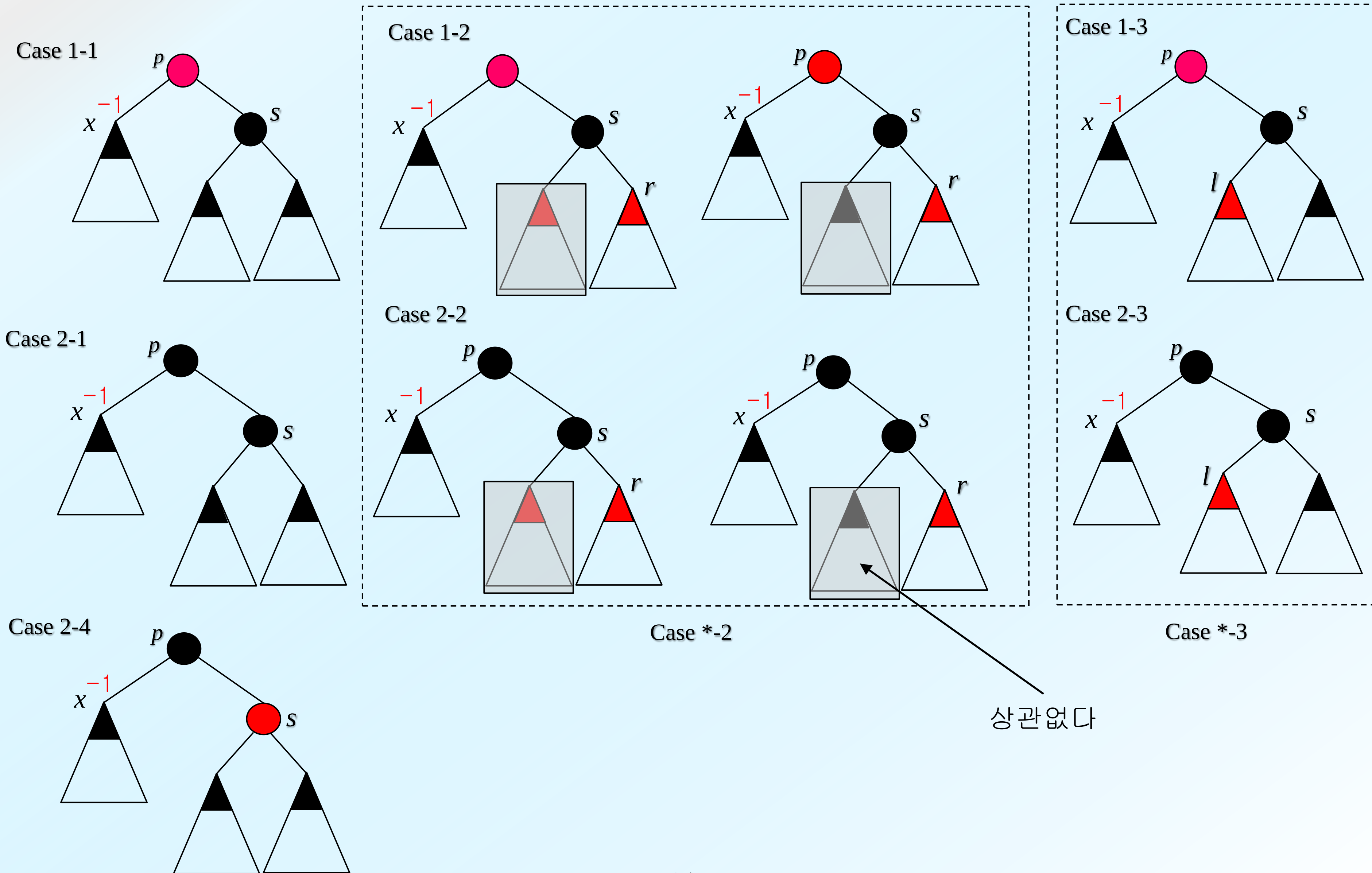


Case 1

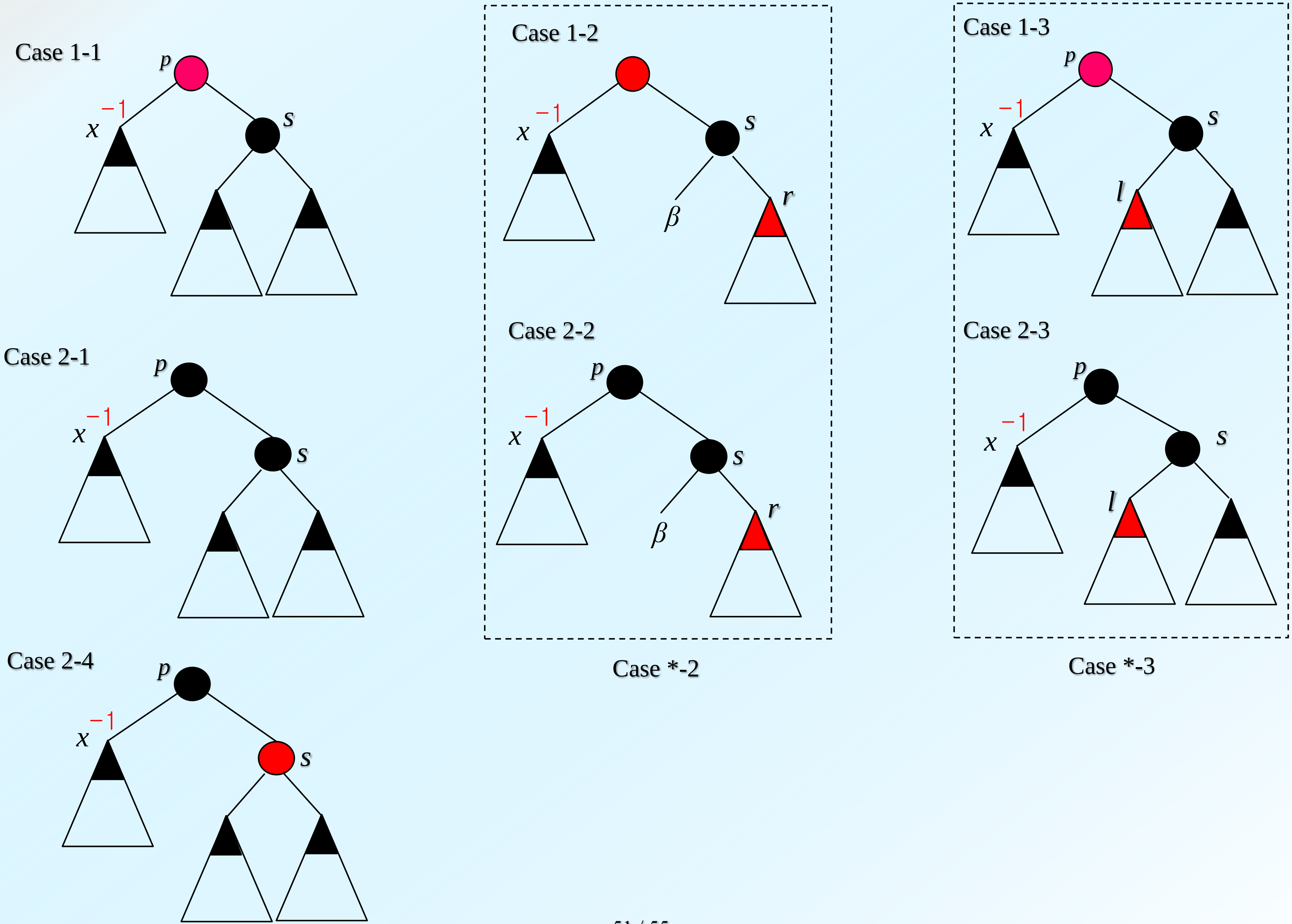


Case 2

All Possible Cases



Five Groups

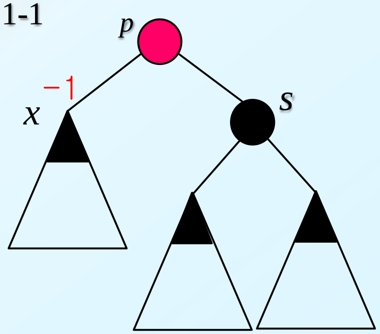


Case 1-1
Case 1-2
Case 1-3
Case 2-1
Case 2-2
Case 2-3
Case 2-4
Case *-2
Case *-3
p
x
s
r
l
β
−1

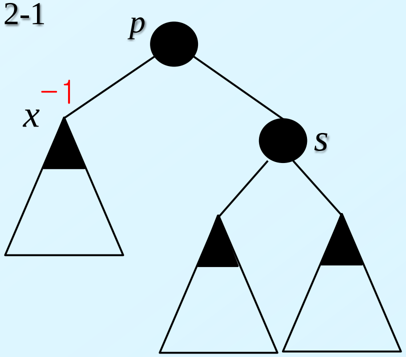
Five Groups

 : either **black** or red

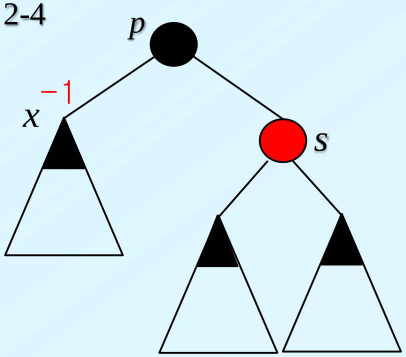
Case 1-1

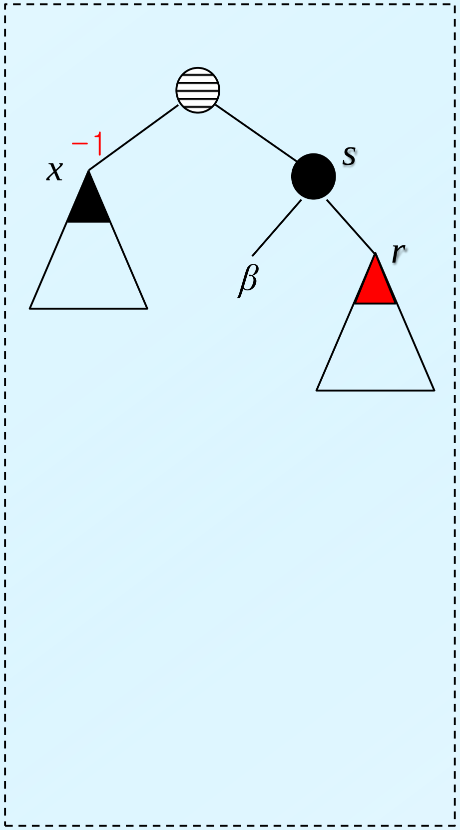


Case 2-1

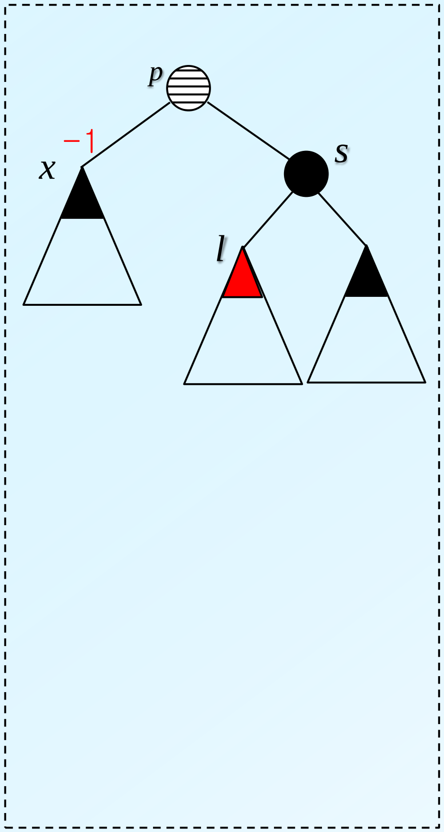


Case 2-4





Case *-2

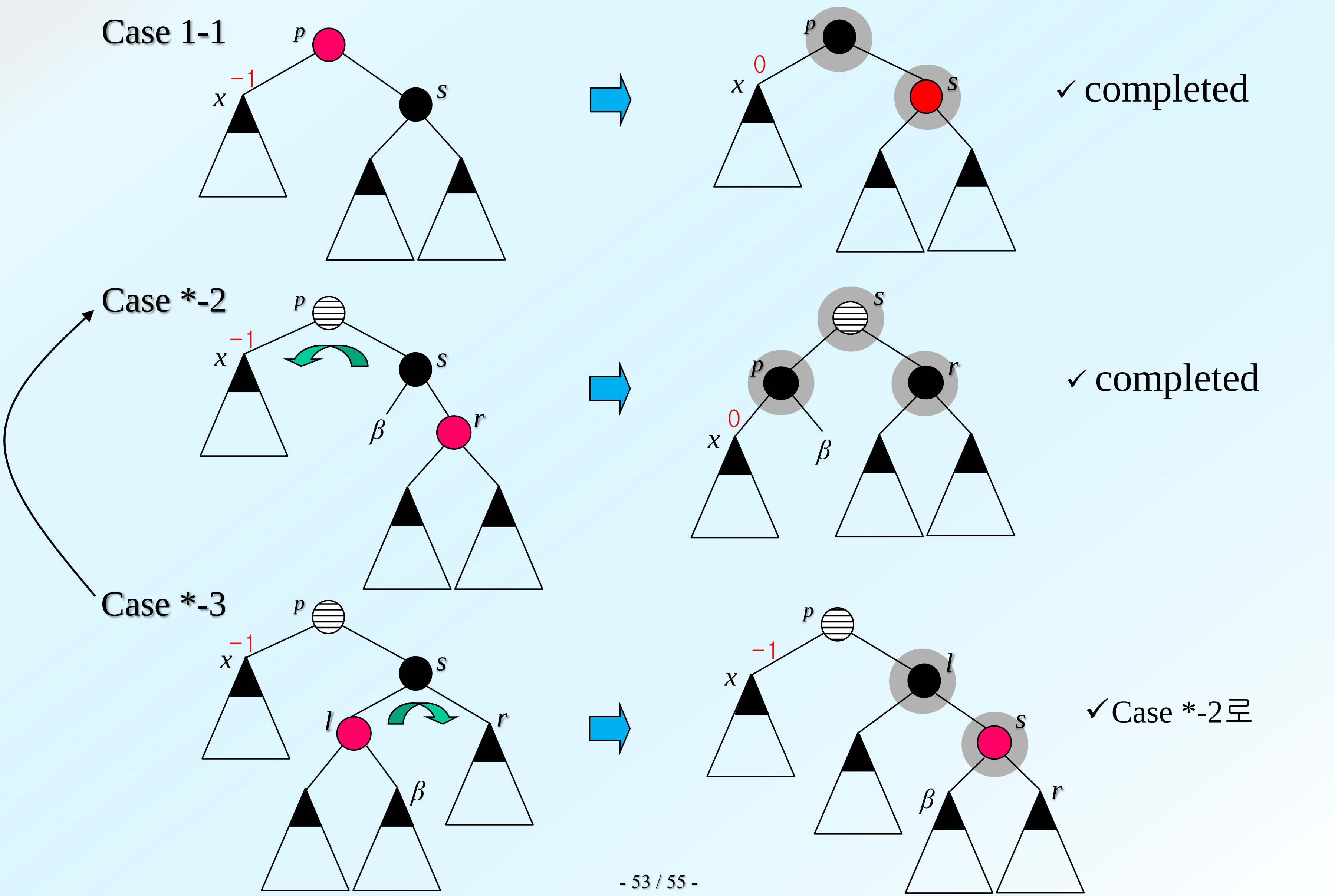


Case *-3

Repair Operations

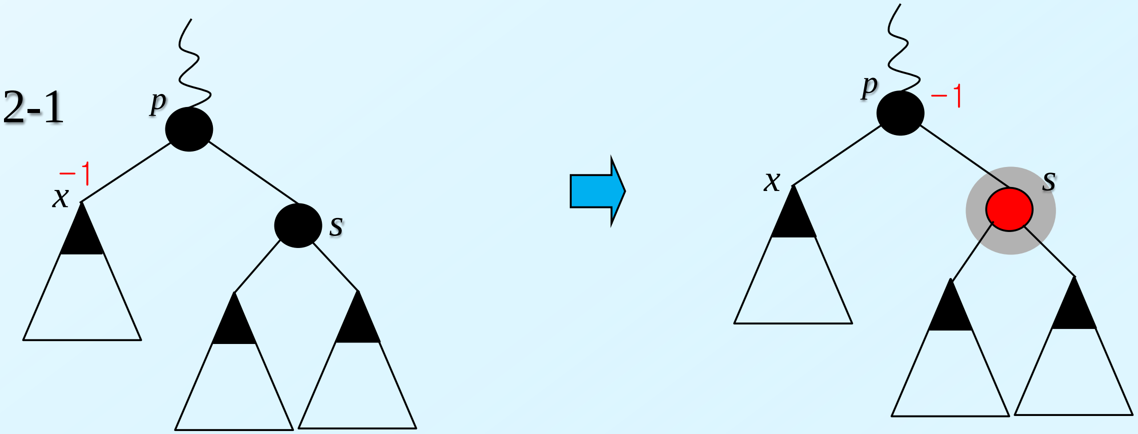
: either **black** or red

: node whose color changes or may change



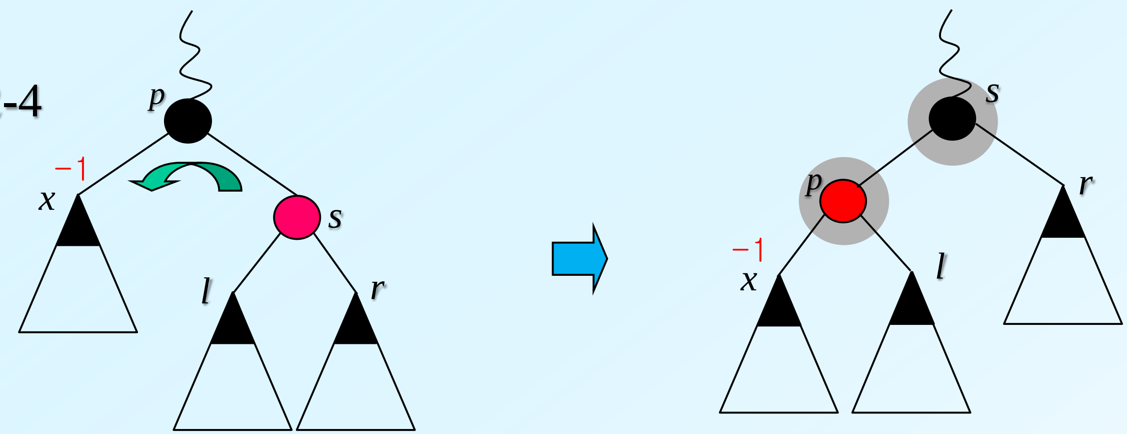
Repair Operations

Case 2-1



✓ p becomes new x: Recursive!

Case 2-4



✓ Case 1-1, 1-2, 1-3 중의 하나로

Running Time

Case 2-1: $O(\log n)$

All the other cases except Case 2-1: $\Theta(1)$

→ $O(\log n)$ in total (considered only repairing)

✓ In addition, intuitively think about why repairs (insertion and deletion) takes $O(\log n)$