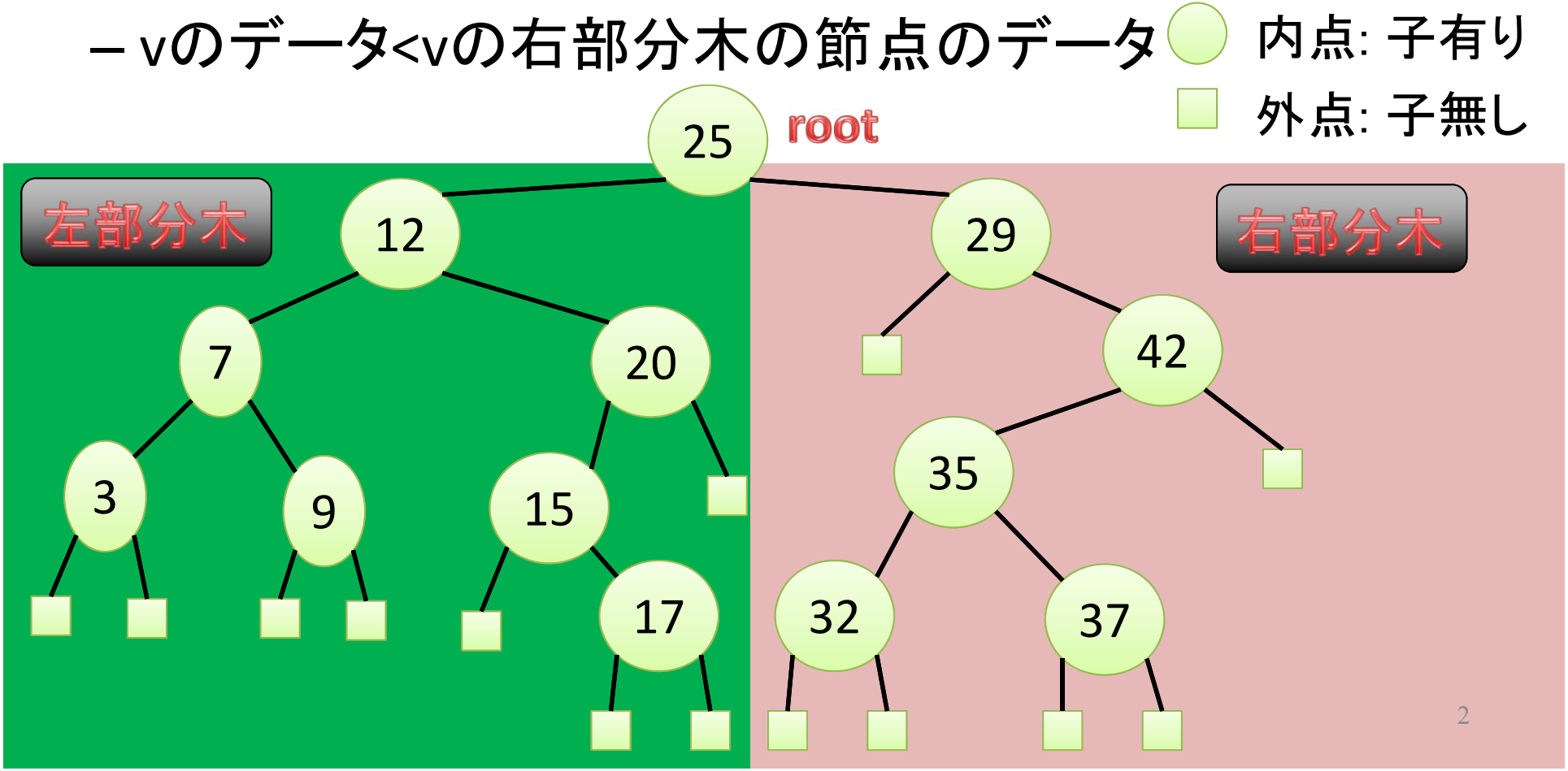
アルゴリズムとデータ構造
第8回: 動的探索問題とデータ構造

担当: 上原隆平 (uehara)

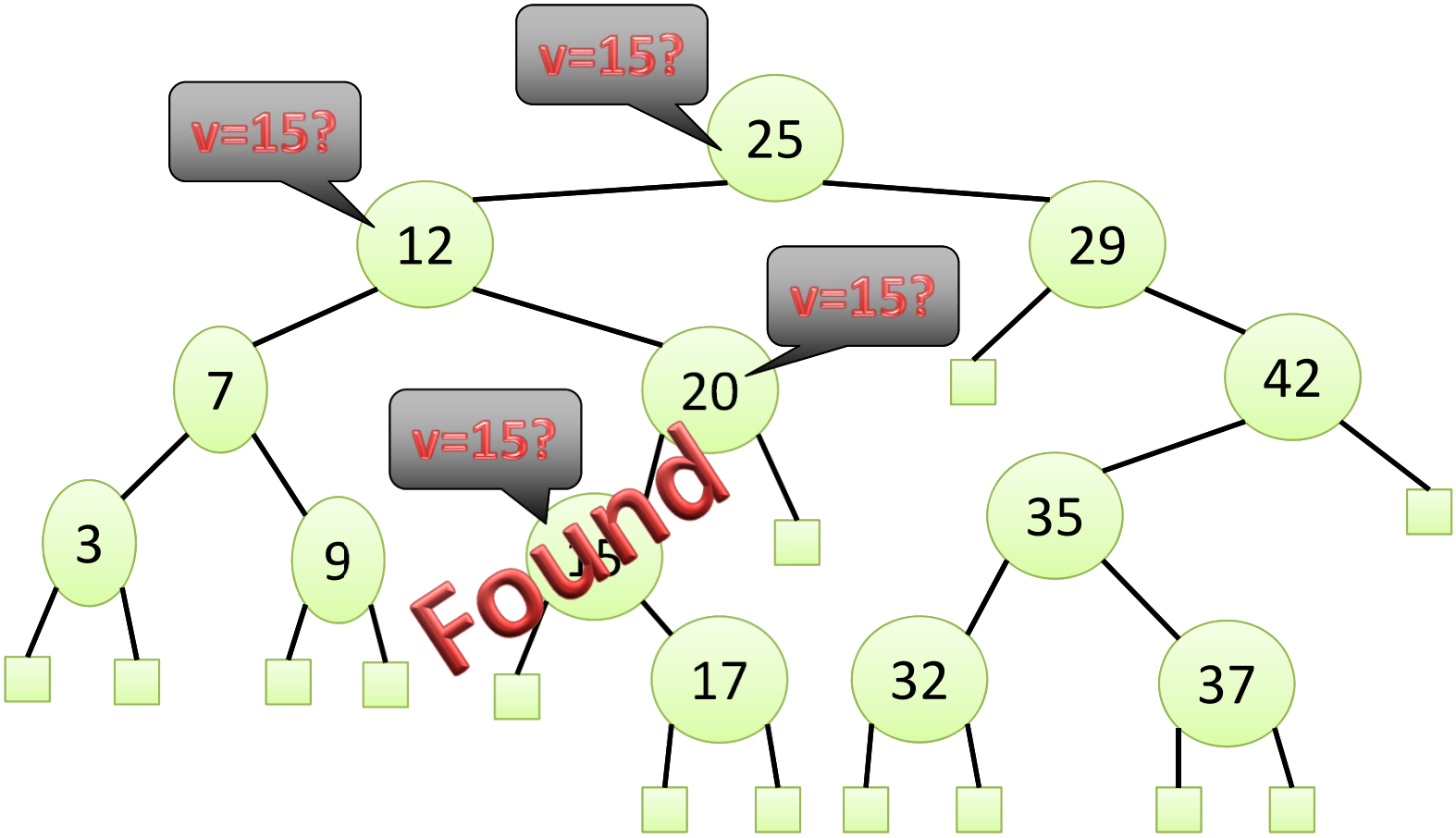
2014/05/08

2分探索木

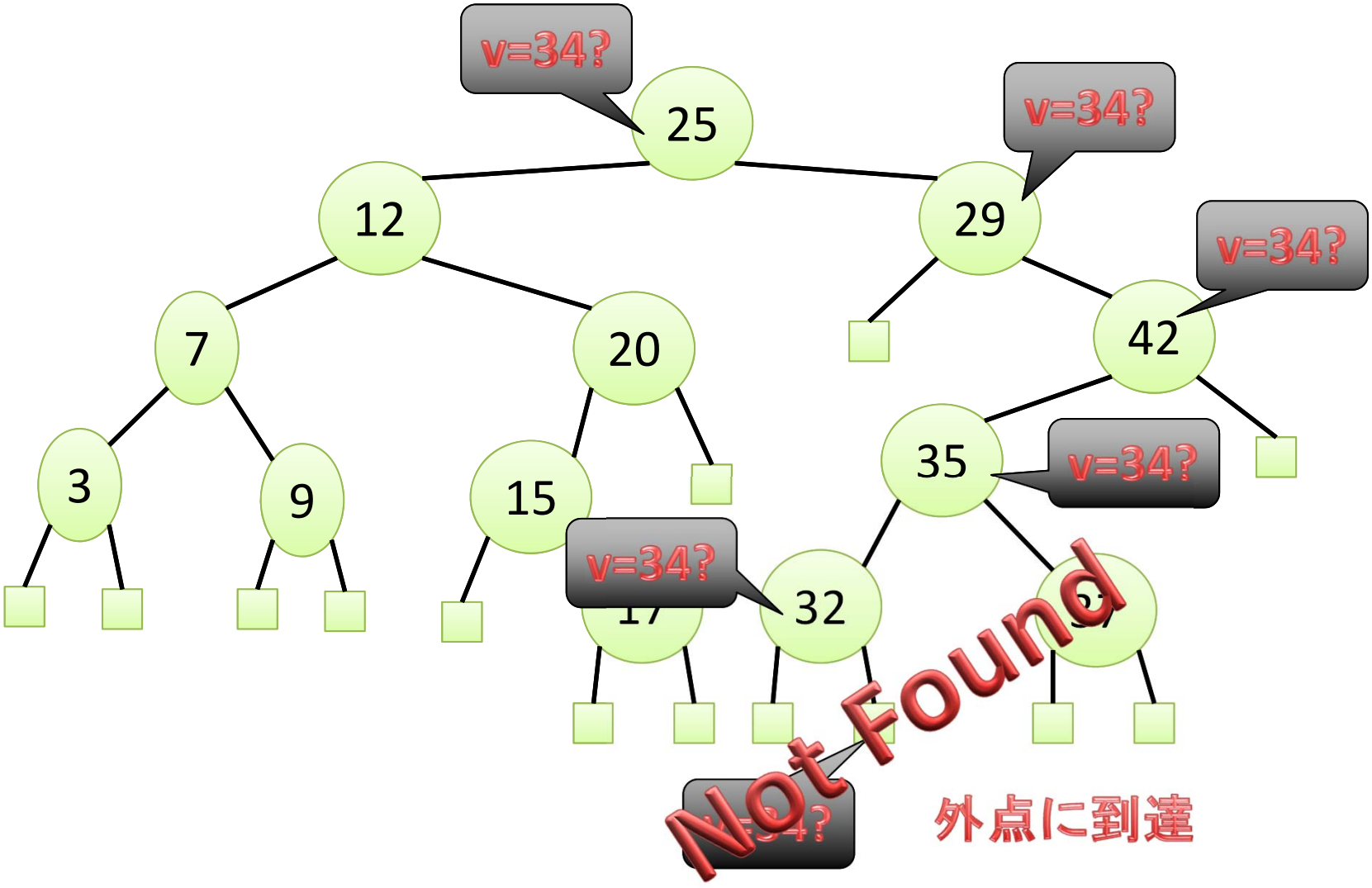
- どの節点vにおいても以下が成立
  - vのデータ>vの左部分木の節点のデータ
  - vのデータ<vの右部分木の節点のデータ



2分探索木における検索: v=15の検索

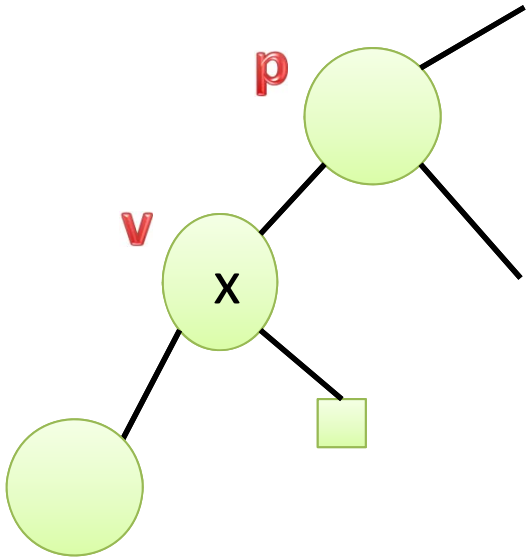


2分探索木における検索: v=34の検索

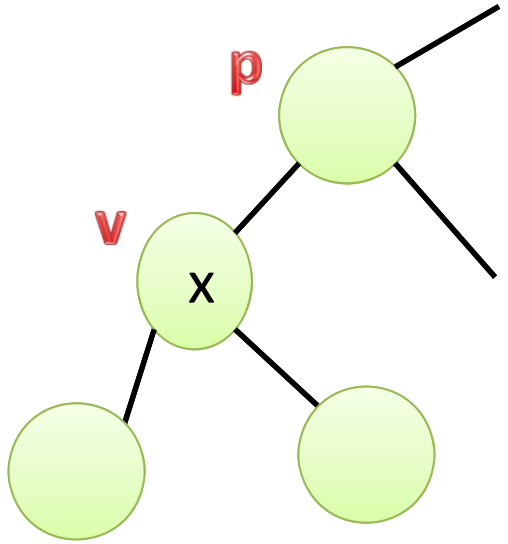


2分木からのデータの削除: キー値がxの節点について場合分け

- Case 1. vの一方の子が外点の場合



- Case 2. vの左右の子が共に内点の場合

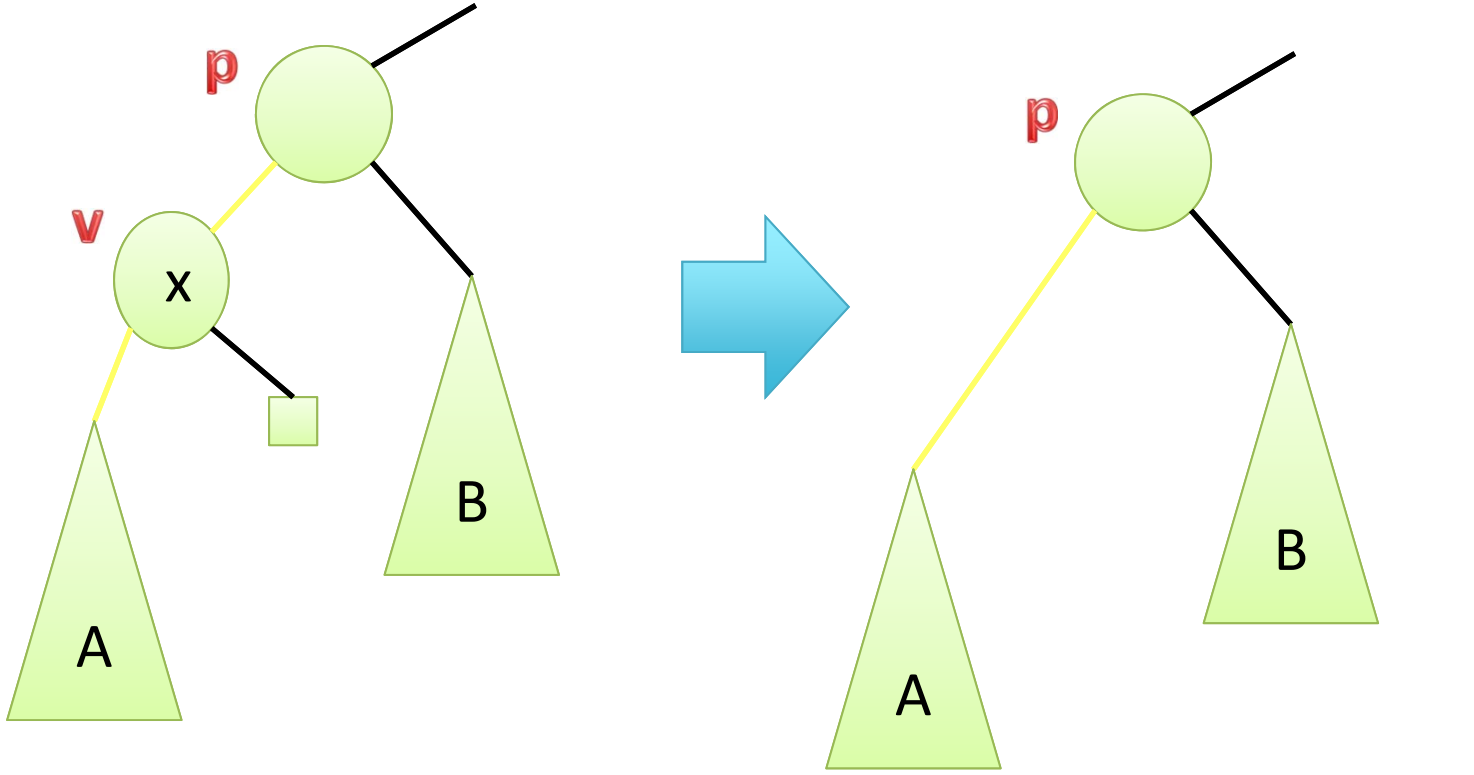


内点: 子有り

外点: 子無し

2分木からのデータの削除: Case 1. vの一方の子が外点

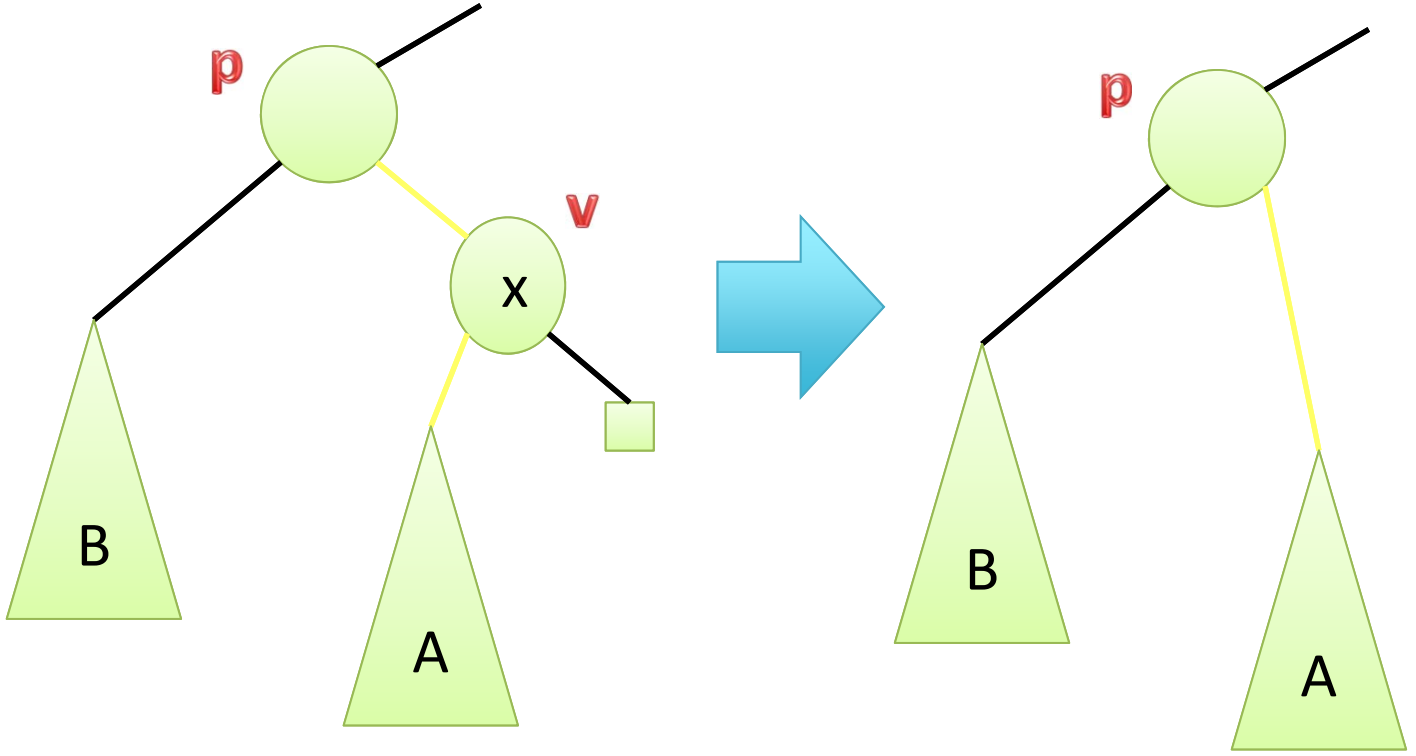
- vが親pの左の子: pの左の子=vの空でない子



Q. 2分木の性質は保たれる?

2分木からのデータの削除: Case 1. vの一方の子が外点

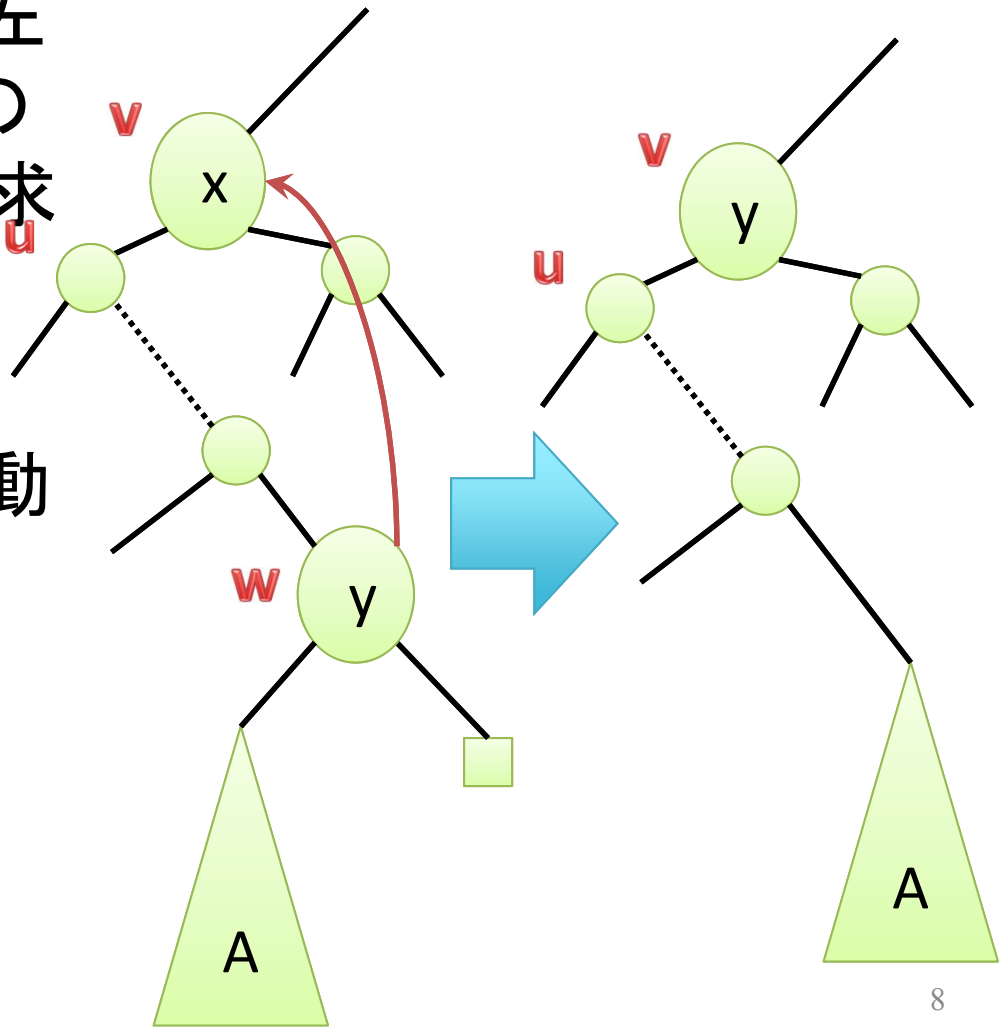
- vが親pの右の子: pの右の子=vの空でない子



2分木からのデータの削除: Case 2. vの左右の子が共に内点

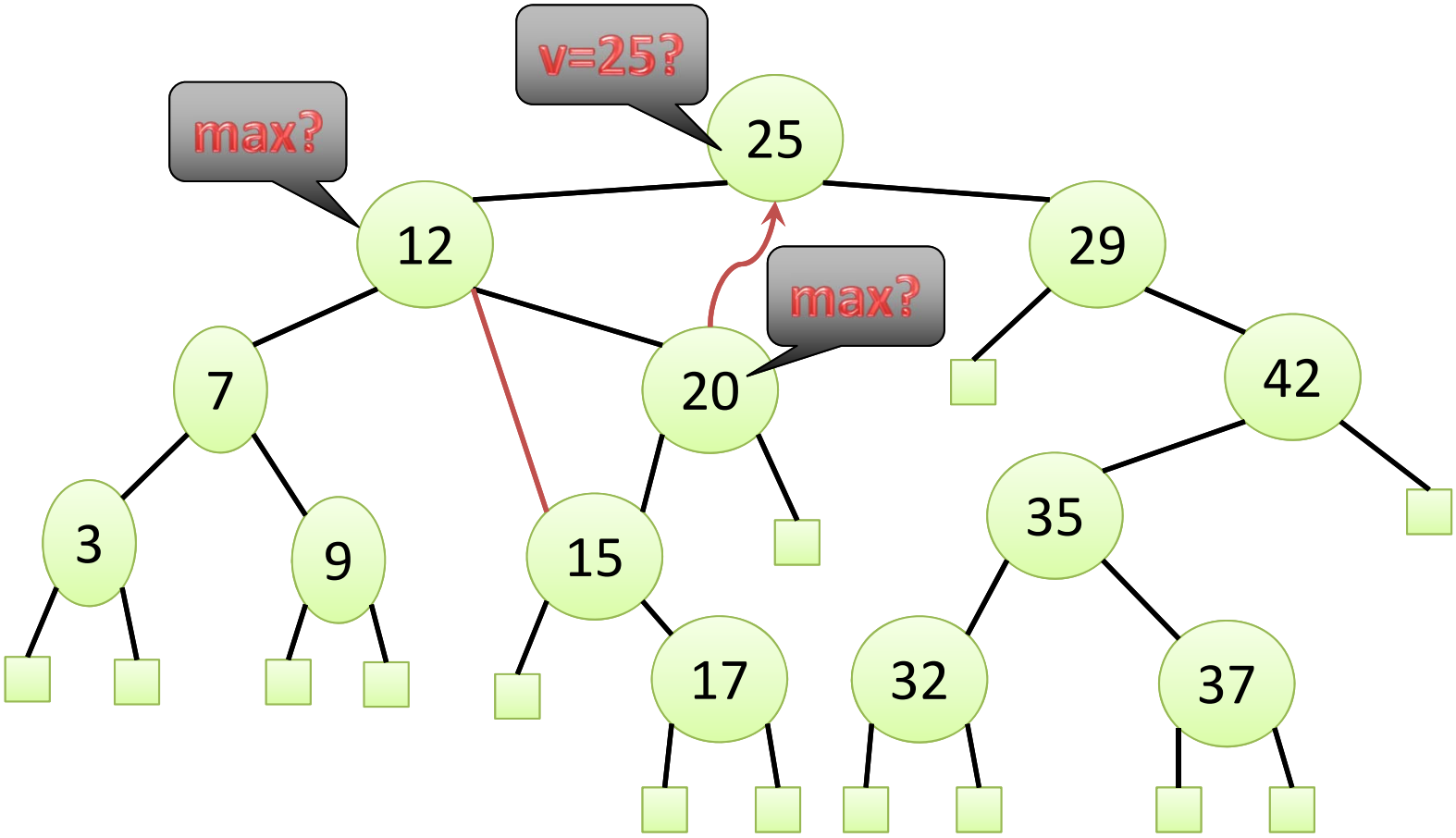
- キーxを含む節点vの左の子uの子孫で最大のキーyを含む節点wを求める
  - 節点wは右の子無し
- yの値をxの場所に移動して節点wを削除
  - Case 1に相当

Why?

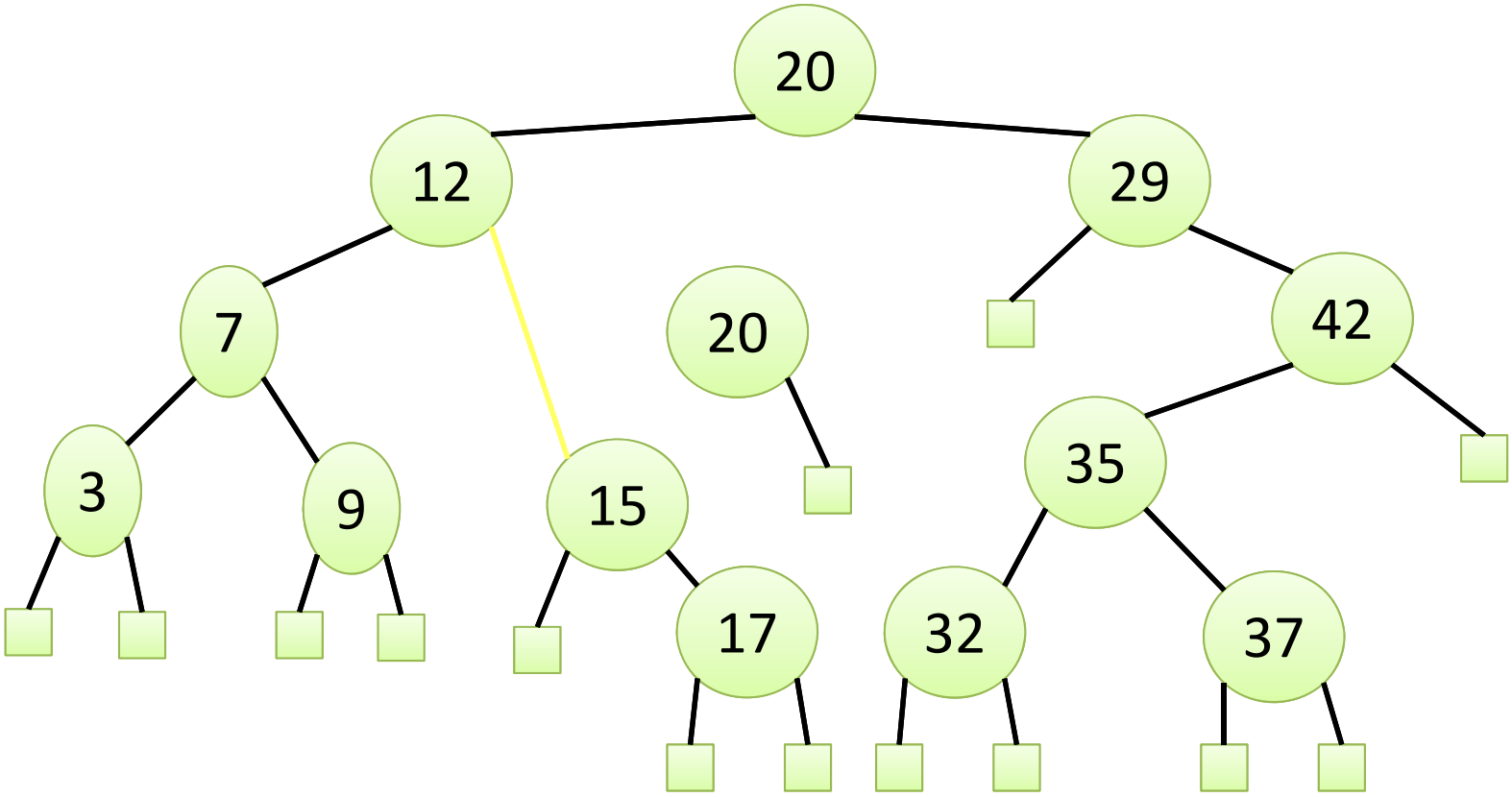


Q. 2分木の性質は?

2分木からのデータの削除:
x=25を削除



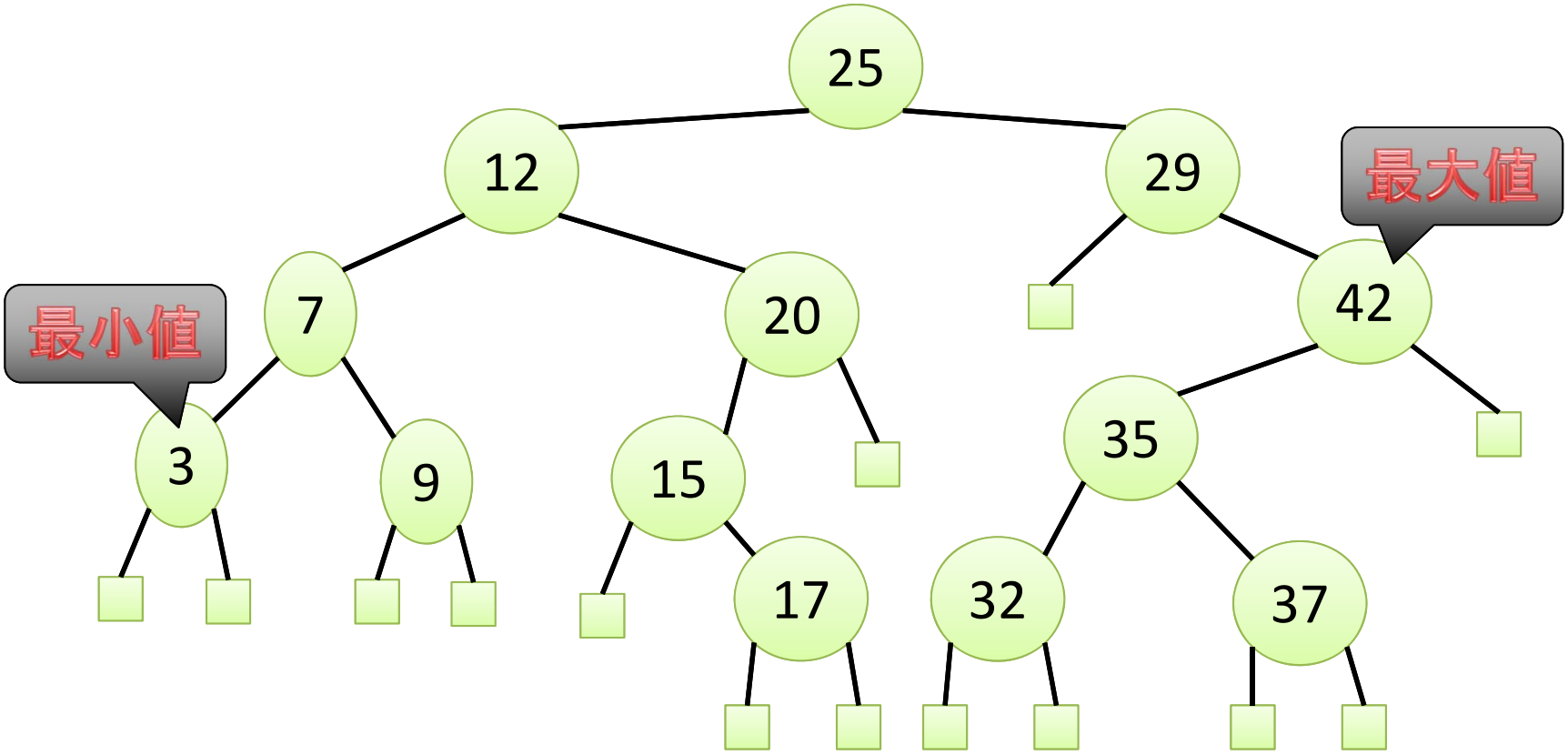
2分木からのデータの削除:
x=25を削除



2分探索木における最大・最小値

- 2分探索木の性質
  - 左の子: キーの値は小さくなる
  - 右の子: キーの値は大きくなる
- 性質の応用
  - 最小値: 根から出発して左の子だけをずっと辿る
  - 最大値: 根から出発して右の子だけをずっと辿る

2分探索木における最大・最小値



2分探索木における最大・最小値

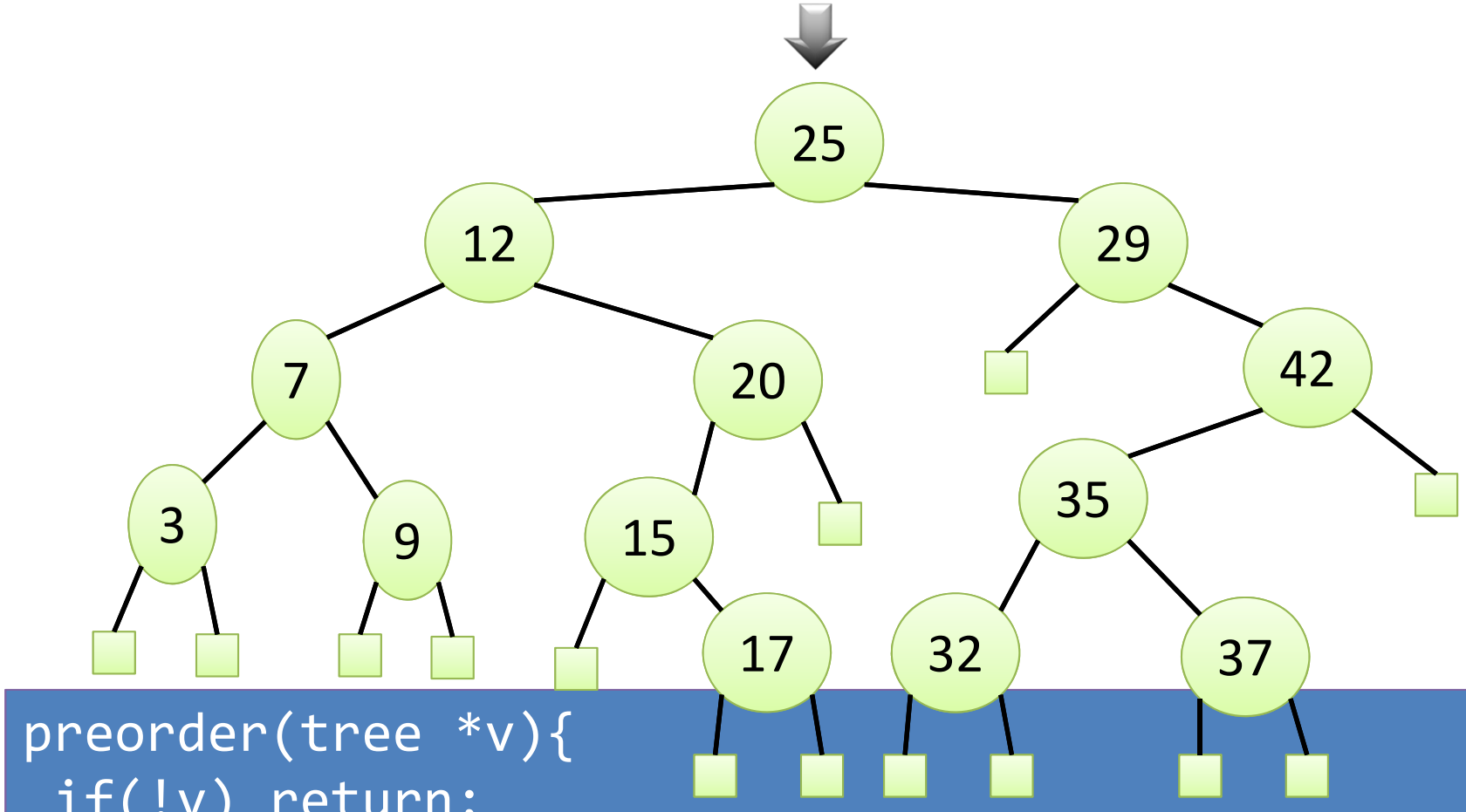
- 2分探索木の性質
  - 左の子: キーの値は小さくなる
  - 右の子: キーの値は大きくなる
- 性質の応用
  - 最小値: 根から出発して左の子だけをずっと辿る
  - 最大値: 根から出発して右の子だけをずっと辿る

- Note: 最小値・最大値を含むノードの削除は簡単 Why?
  - 共に少なくともどちらか一方の子が外点

2分探索木に蓄えられた値の表示の仕方 (あるいは2分探索木の辿り方)

- 先行順 (preorder):
  ノードの値 → 左部分木 → 右部分木
- 中間順 (inorder):
  左部分木 → ノードの値 → 右部分木
- 後行順 (postorder):
  左部分木 → 右部分木 → ノードの値

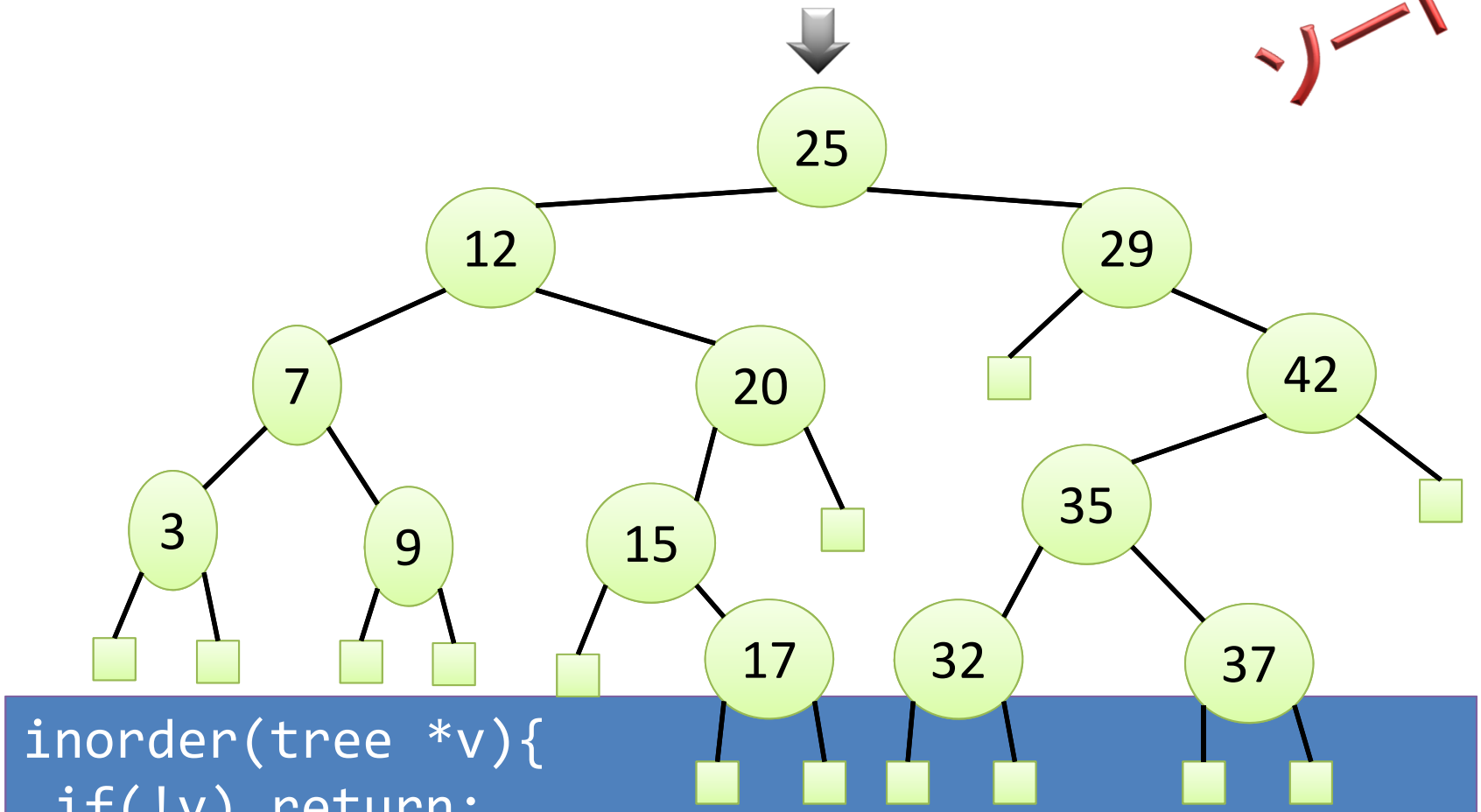
2分探索木の辿り方: preorder
ノードの値 → 左部分木 → 右部分木




```
preorder(tree *v){
 if(!v) return;
 visit(v); preorder(v->lson); preorder(v->rson);
}
```

25
12
7
3
9
20
15
17
29
42
35
32
37

2分探索木の辿り方: inorder
左部分木 →ノードの値 → 右部分木

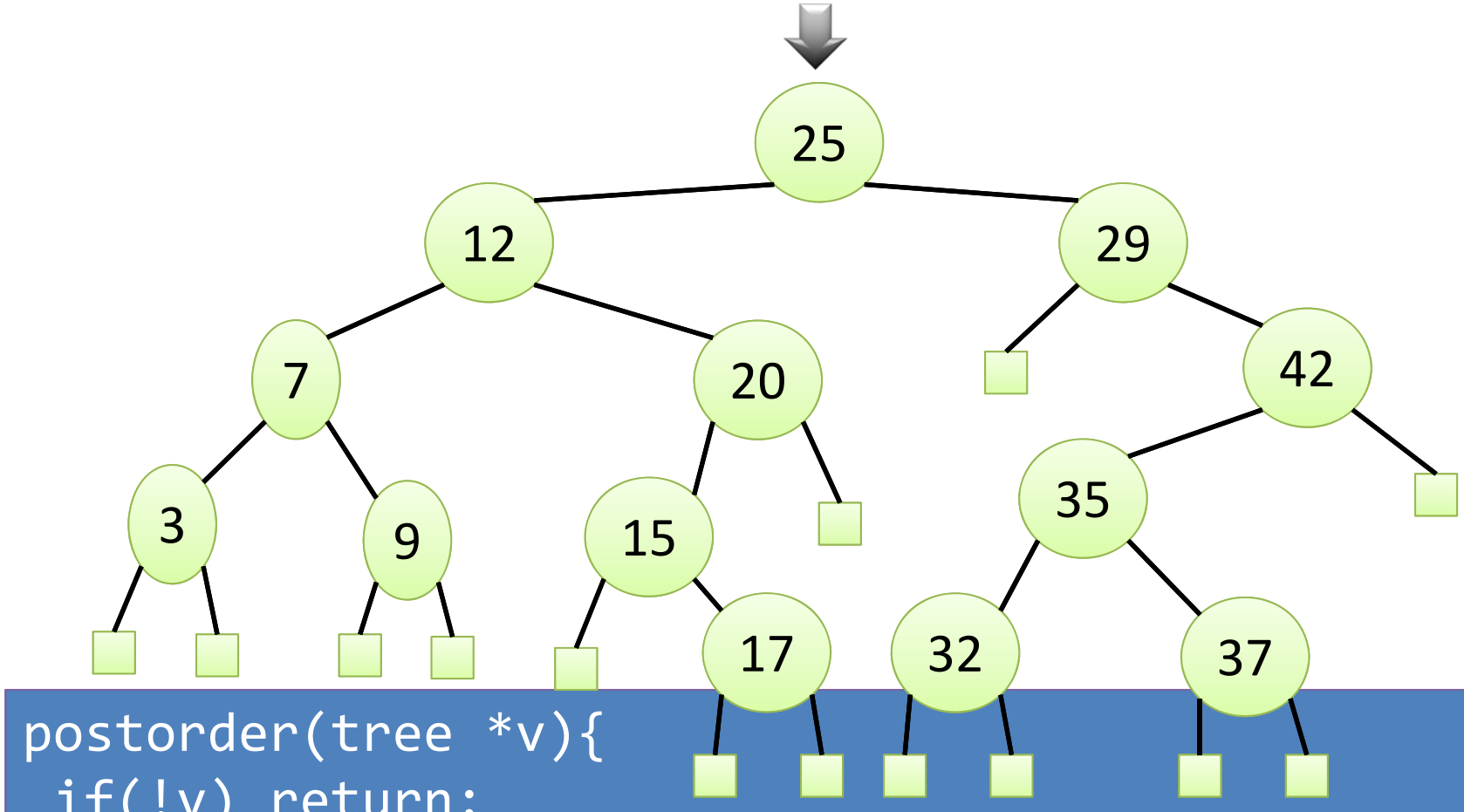


ソート順！

3
7
9
12
15
17
20
25
29
32
35
37
42

```
inorder(tree *v){
 if(!v) return;
 inorder(v->lson); visit(v); inorder(v->rson);
}
```

2分探索木の辿り方: postorder
左部分木 → 右部分木 →ノードの値

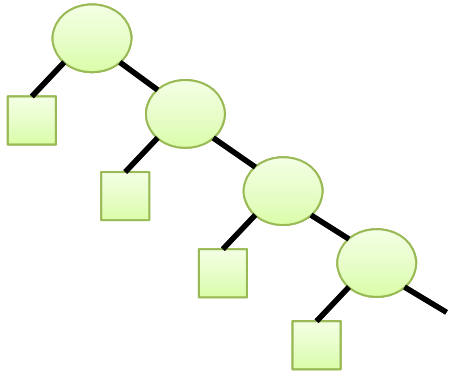



```
postorder(tree *v){
 if(!v) return;
 postorder(v->lson);postorder(v->rson);visit(v);
}
```

3
9
7
17
15
20
12
32
37
35
42
29
25

2分探索木の効率

- 最悪の場合: O(n)
  - 小さい値から順に挿入→
    レベルの数=節点数



- ランダムに挿入: O(log n)
  - 木の深さ: O(log n)

平衡2分探索木
(Self-)Balanced Binary Search Tree

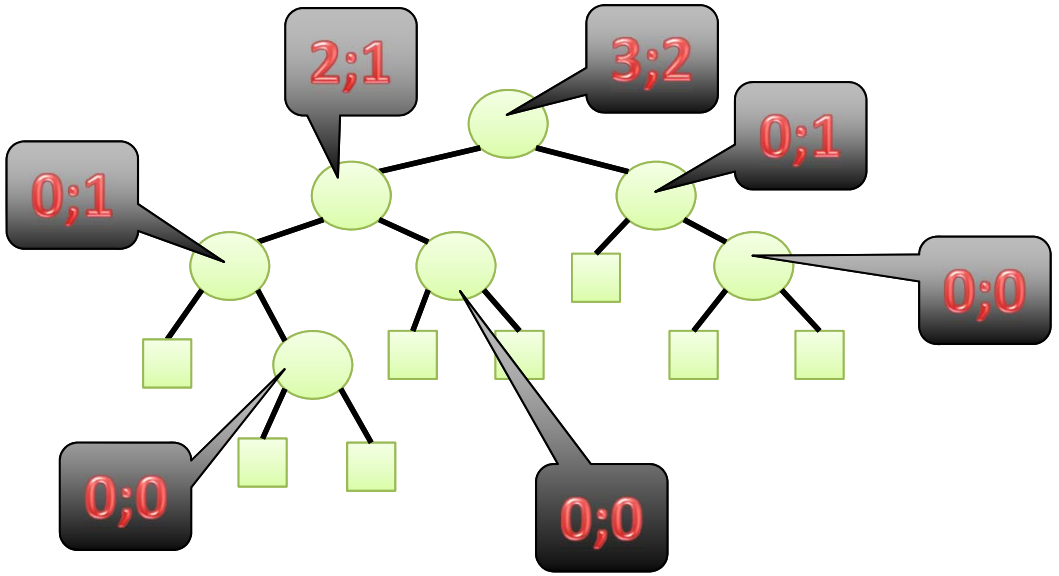
- 常に最大の深さが節点数に対して$O(\log n)$になるようにバランスを取る
  - e.g., AVL木, 2-3木, 2色木(赤黒木)

Georgy M. Adelson-Velsky
(1922−2014)

Evgenii M. Landis
(1921−1997)

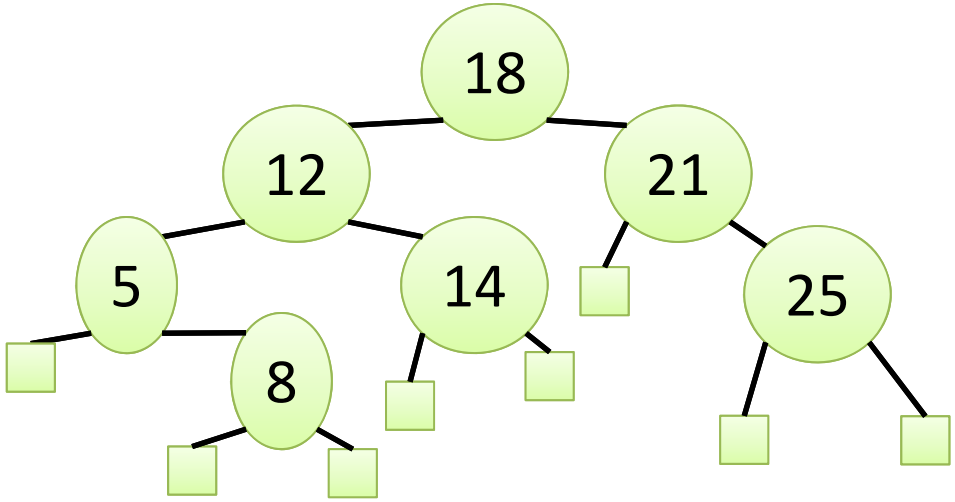
AVL木 [G.M. Adelson-Velskii and E.M. Landis '62]

- 性質: どの節点でも左部分木の深さと右部分木の深さの差(バランス度)が1以内
- 例:



AVL木: データの挿入

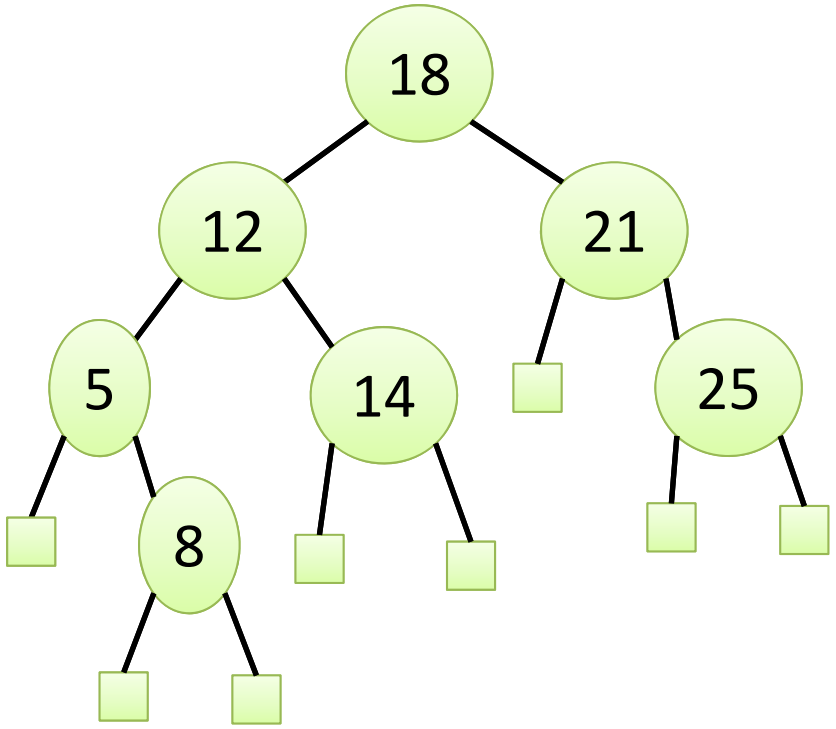
- 新データxを蓄えるべき節点(=外点)vを求める
- vを内点にして，与えられたデータxを蓄える
- xの挿入によるバランス度の変化を計算
- 根に戻りながら各節点のバランス度を調べ、崩れている場合は復元操作(木の回転)を行う



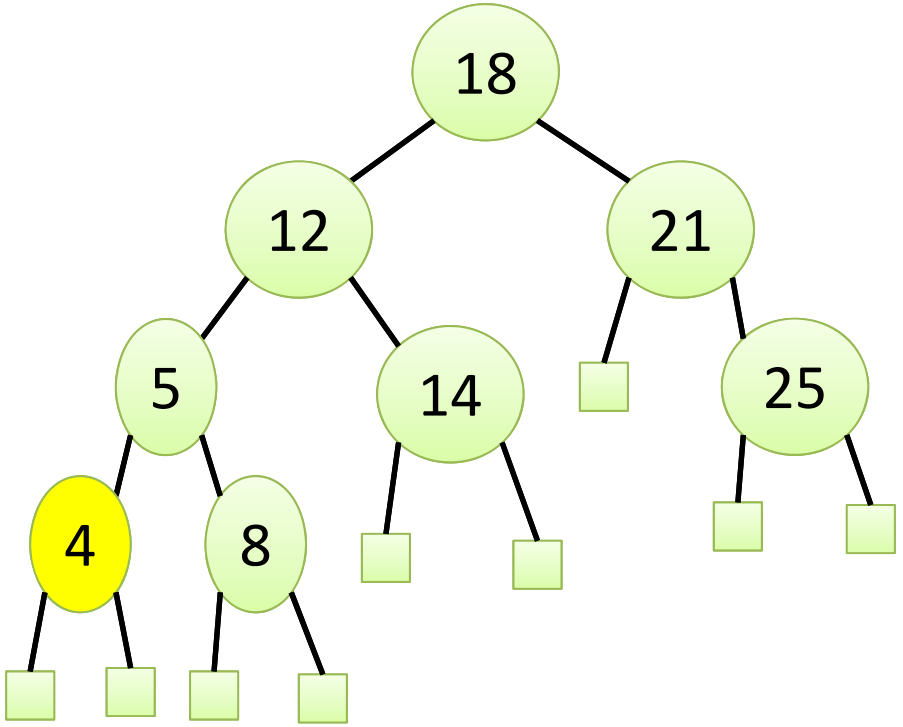
x=4を挿入したとき、バランス度はどう変化する? x=10, x=20, x=23のときは?

AVL木: データの挿入
x=4を挿入

挿入前



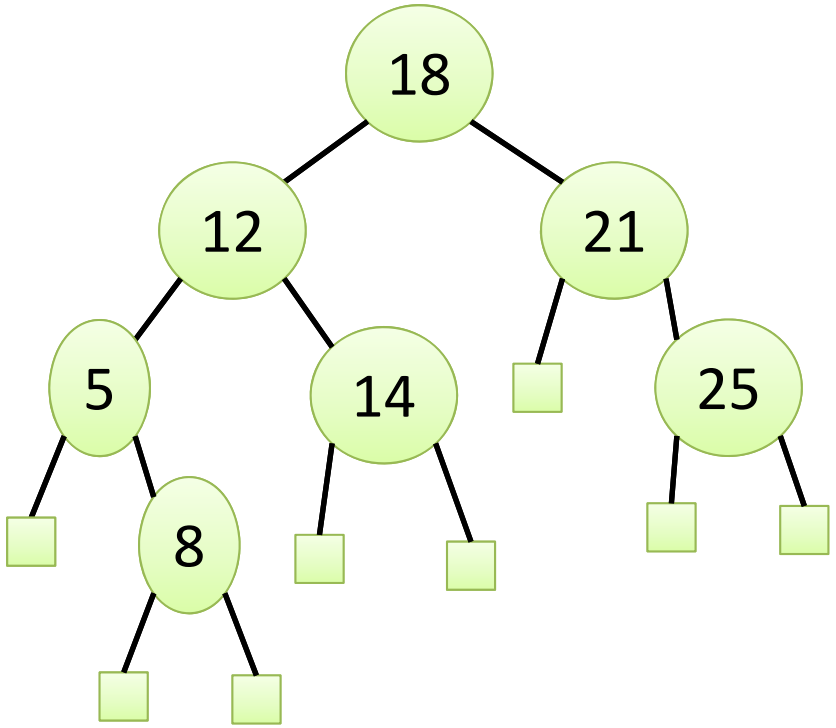
挿入後



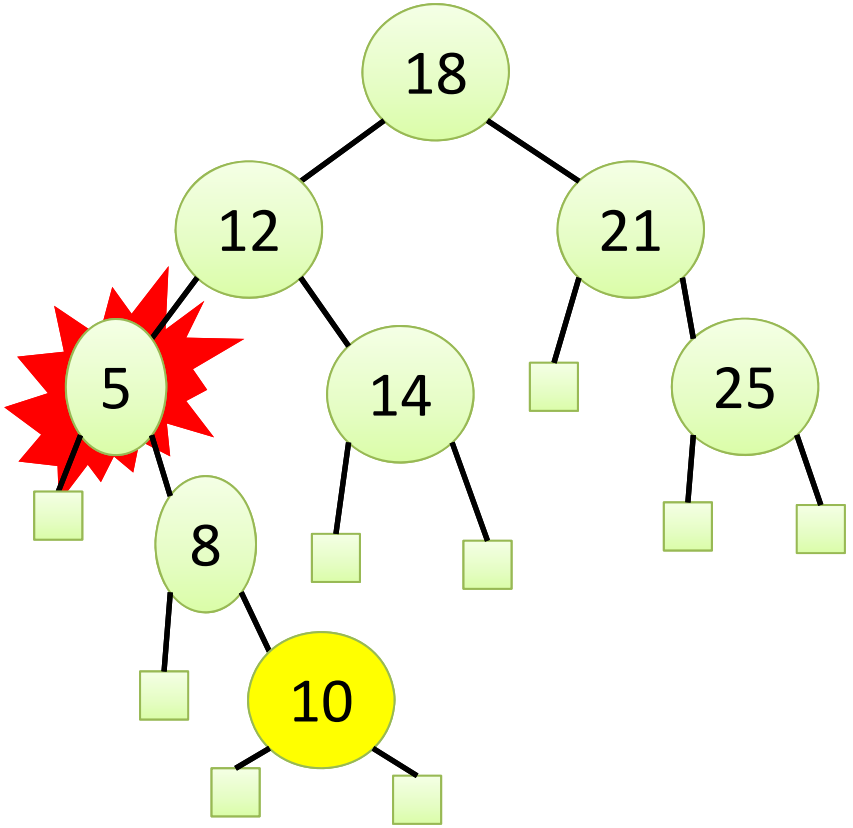
バランス度OK

AVL木: データの挿入
x=10を挿入

挿入前



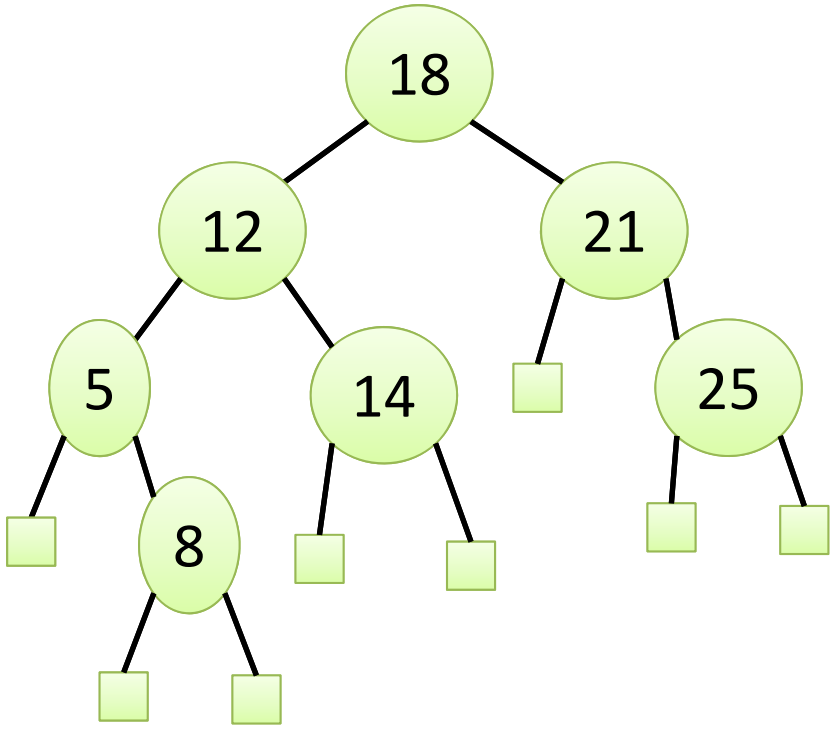
挿入後



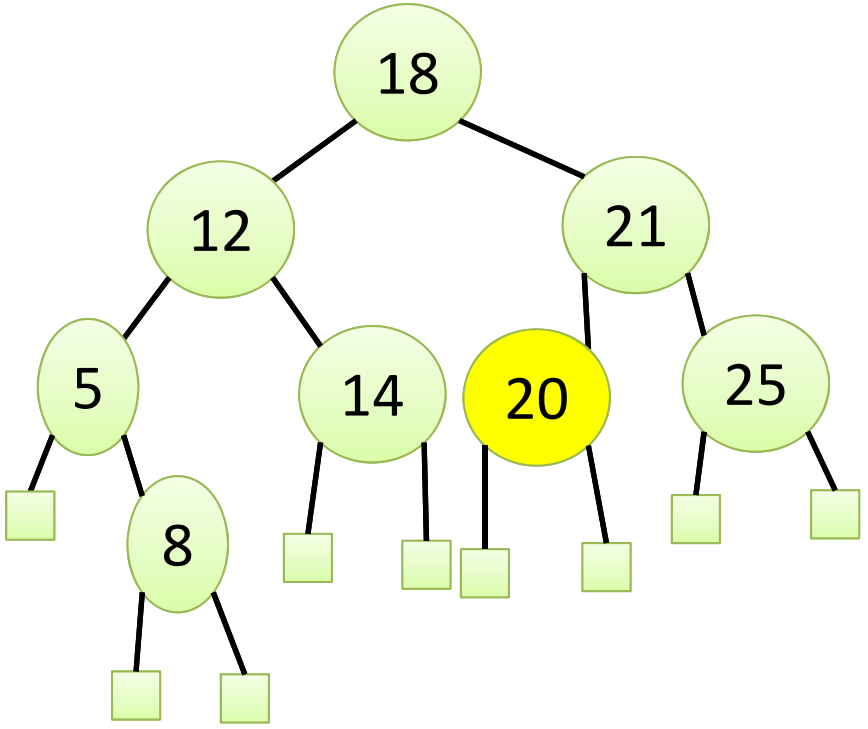
0;2@node 5

AVL木: データの挿入
x=20を挿入

挿入前



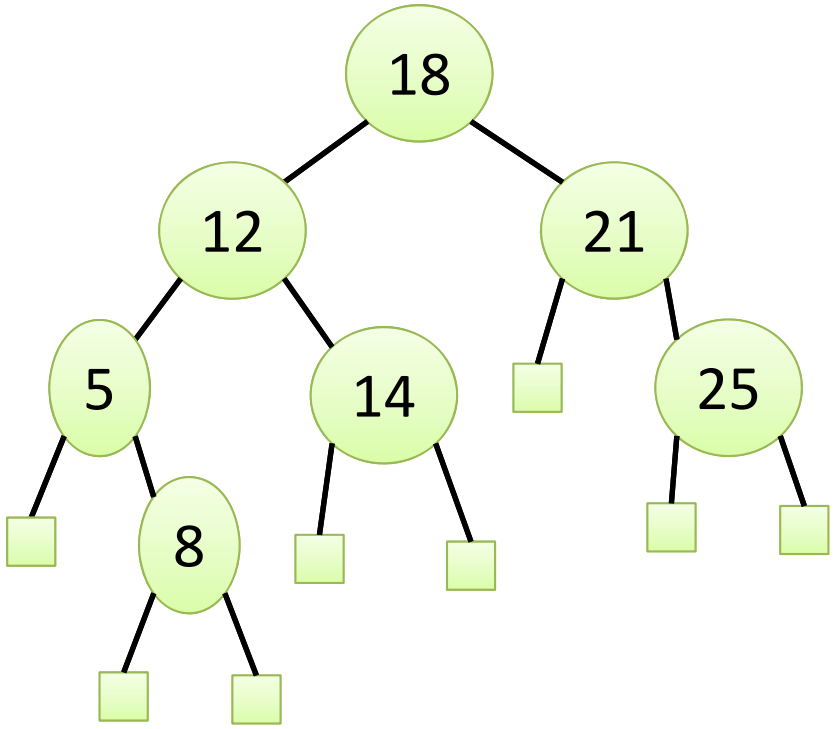
挿入後



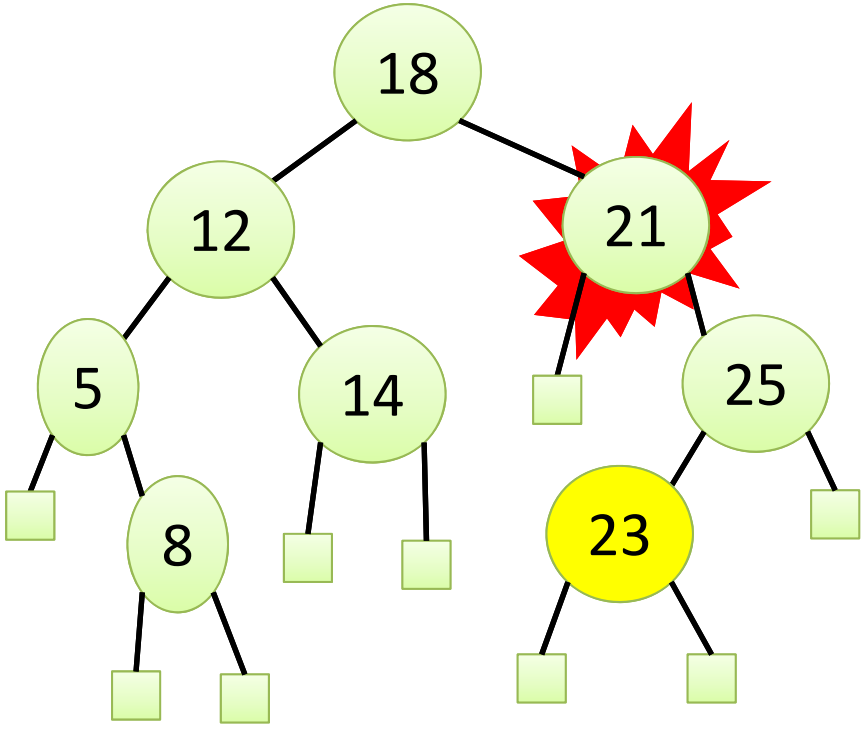
バランス度OK

AVL木: データの挿入
x=23を挿入

挿入前



挿入後



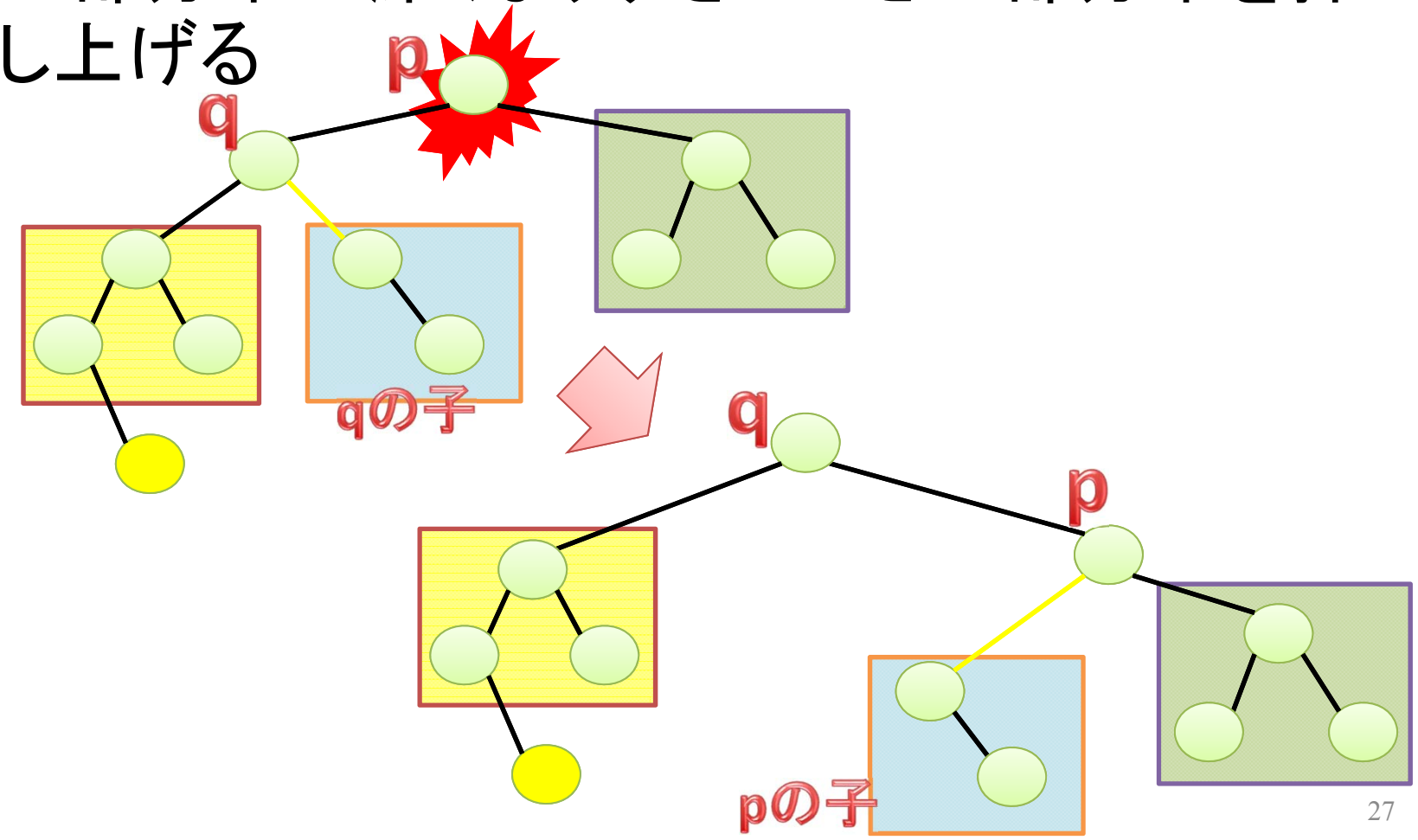
0;2@node 21

AVL木: バランスの復元/回転操作

- 深さの差が1以内になるように木を回転させる
  - 単一LL回転
  - 単一RR回転
  - 二重LR回転
  - 二重RL回転

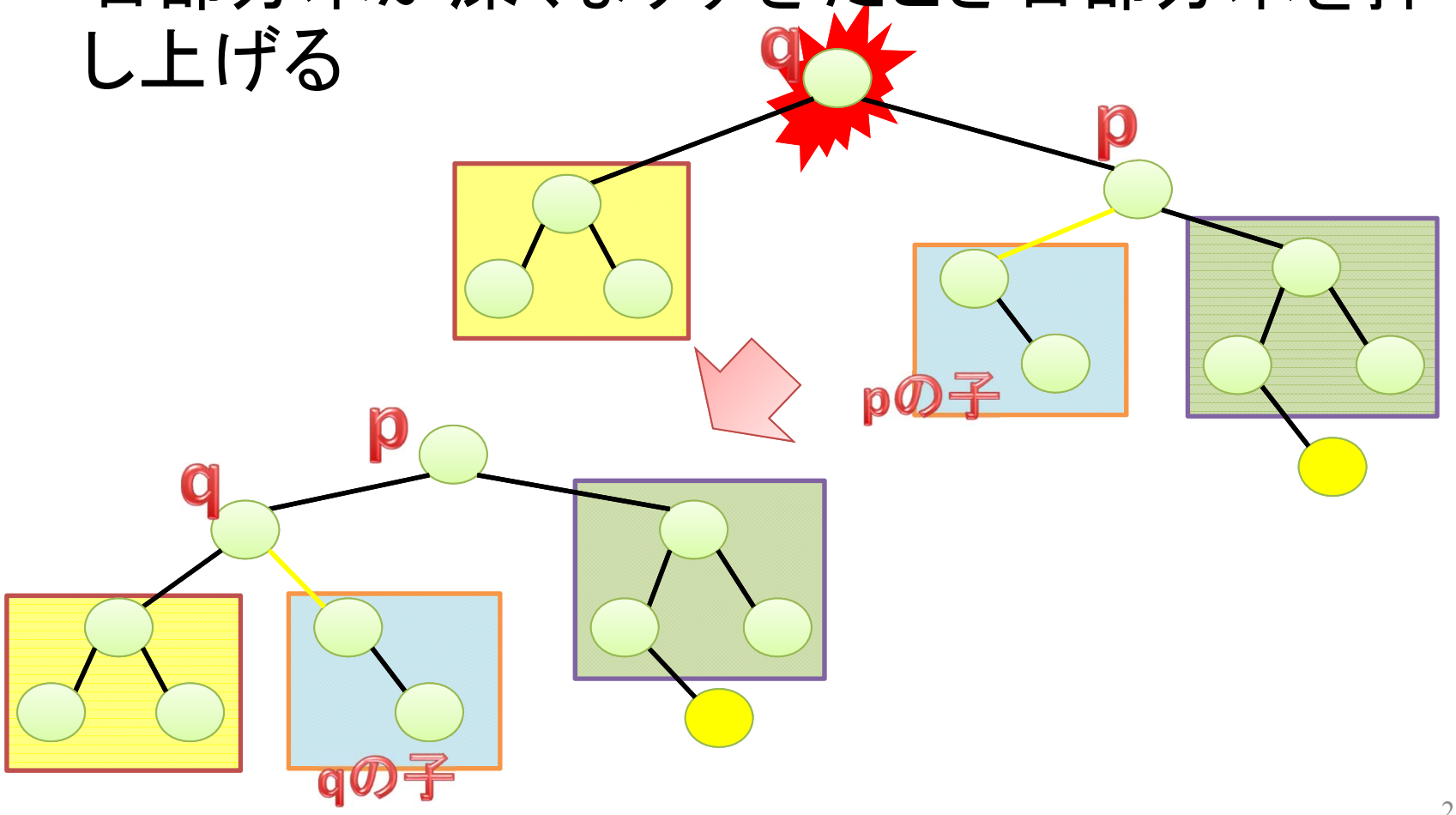
AVL木: バランスの復元/回転操作
単一LL回転

- 左部分木が深くなりすぎたとき左部分木を押し上げる



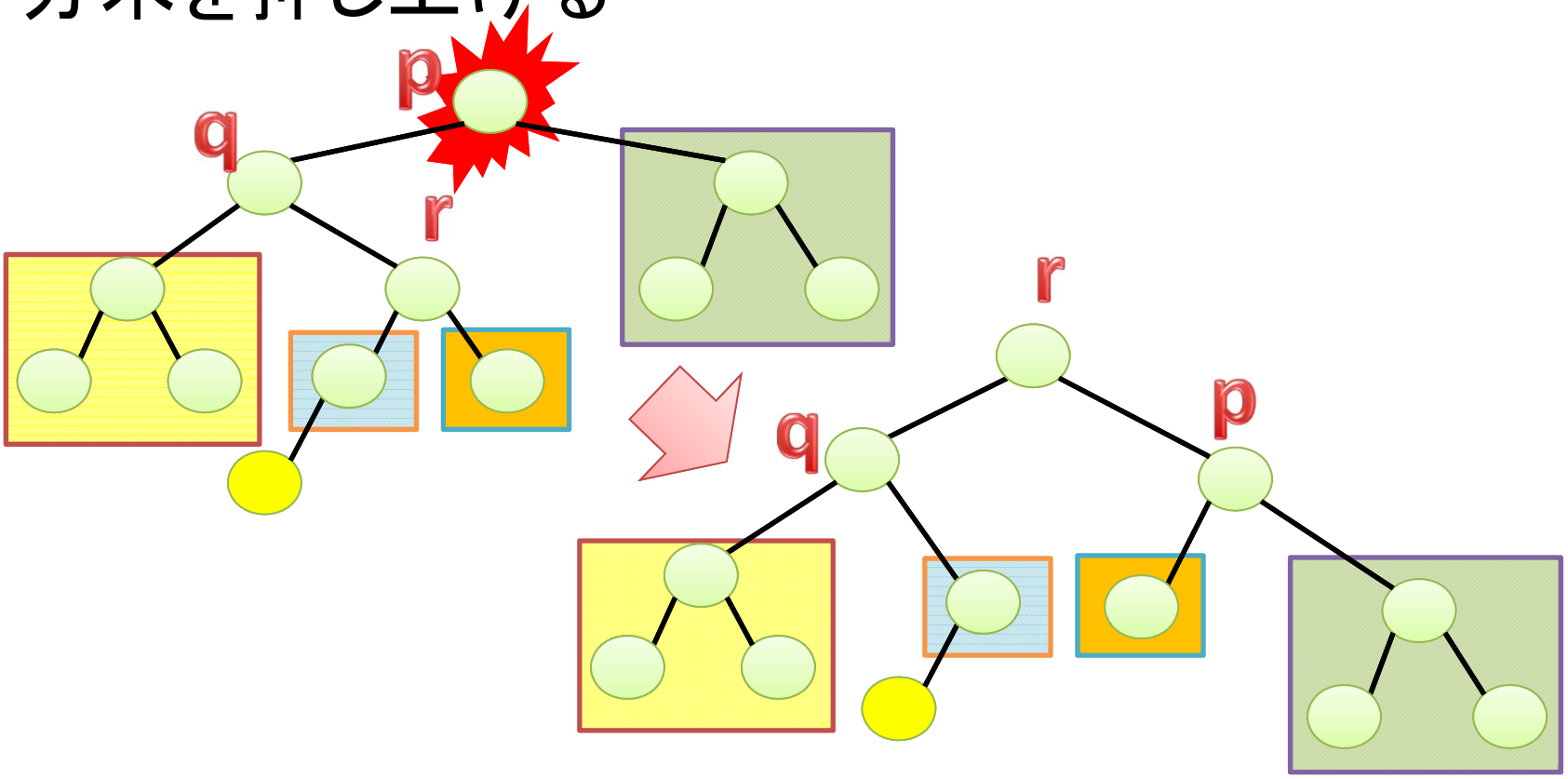
AVL木: バランスの復元/回転操作
単一RR回転

- 右部分木が深くなりすぎたとき右部分木を押し上げる



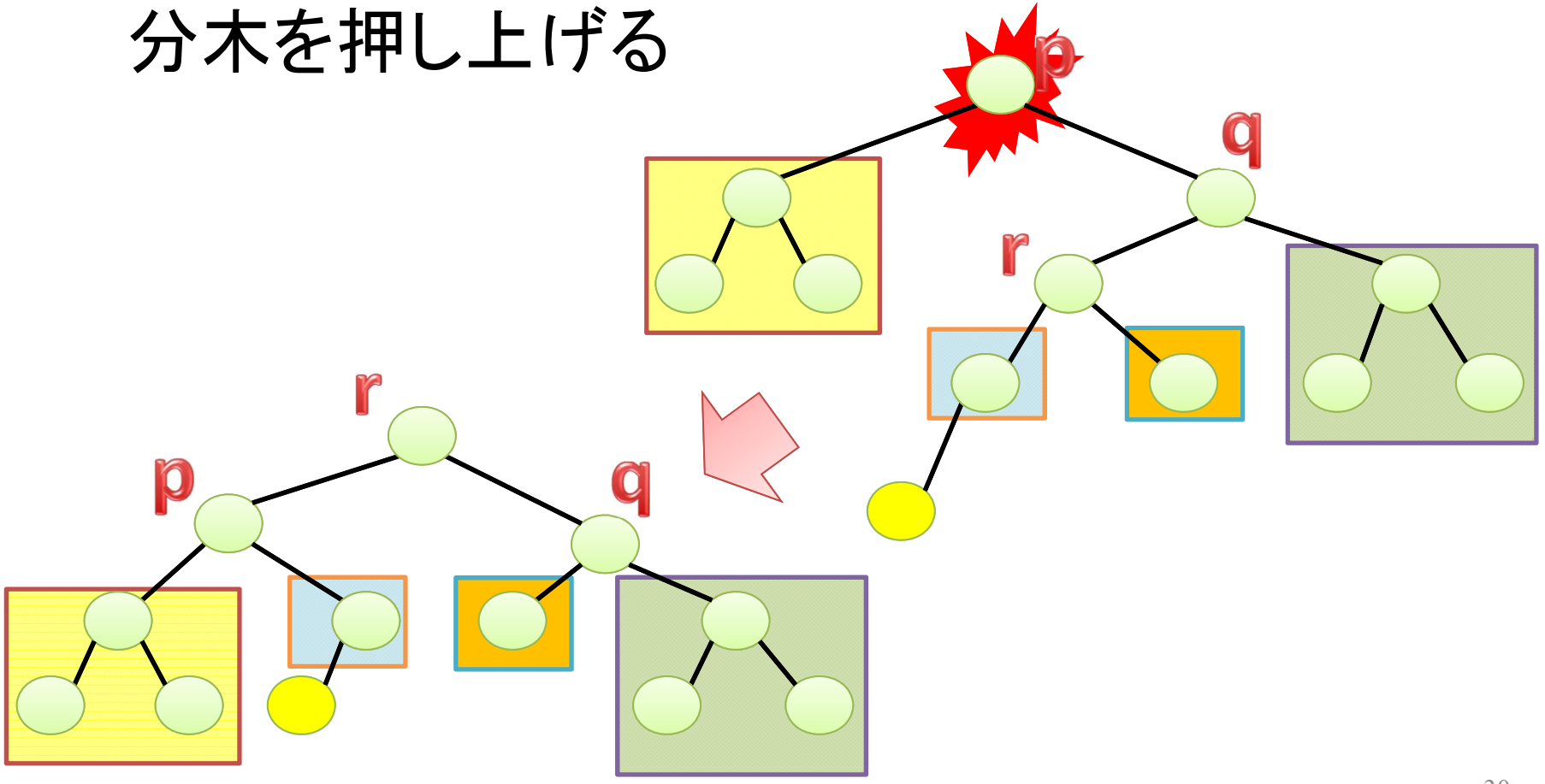
AVL木: バランスの復元/回転操作
二重LR回転

- 左→右部分木が深くなりすぎたとき左→右部分木を押し上げる



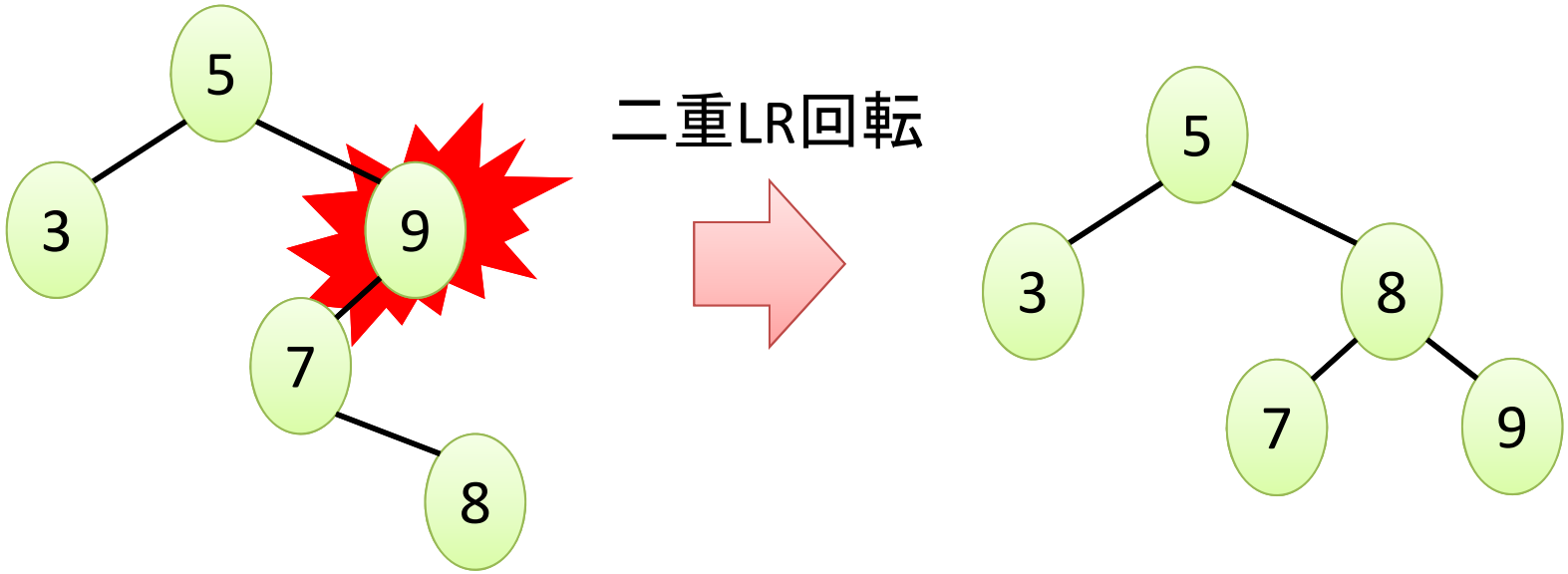
AVL木: バランスの復元/回転操作
二重RL回転

- 右→左部分木が深くなりすぎたとき右→左部分木を押し上げる



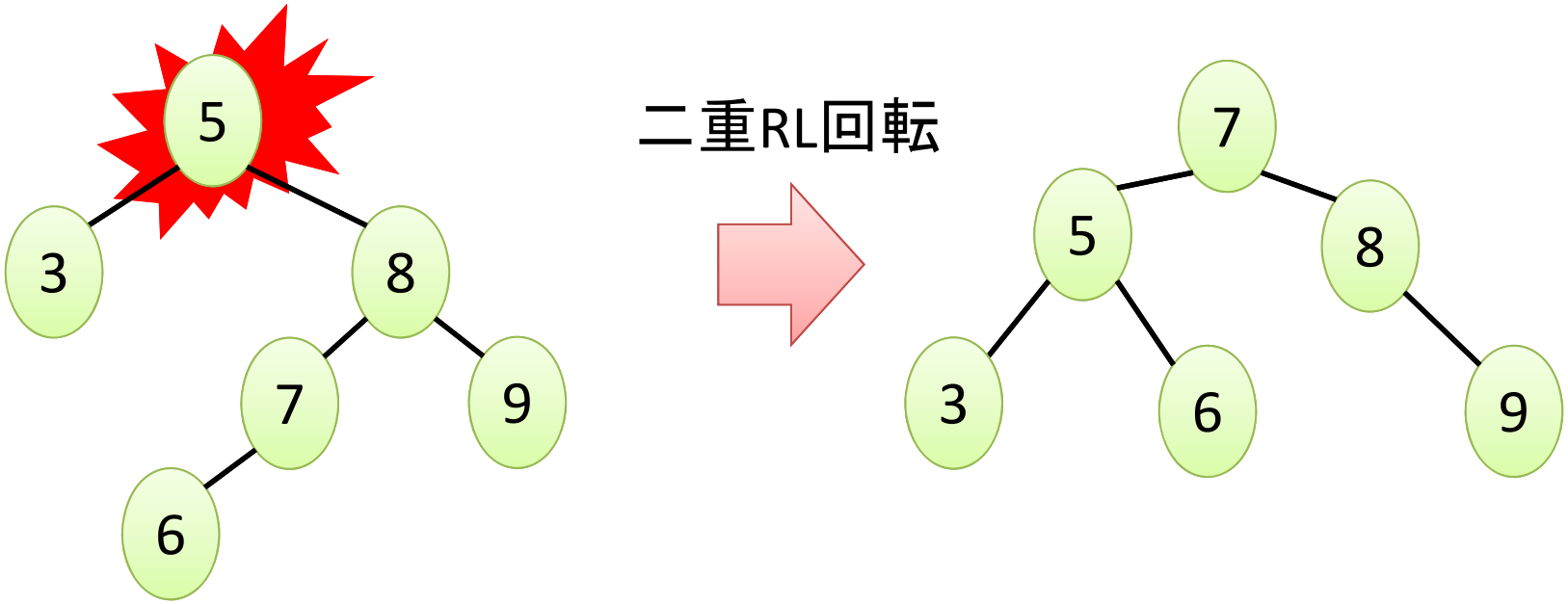
AVL木: 動作例

- 8を挿入



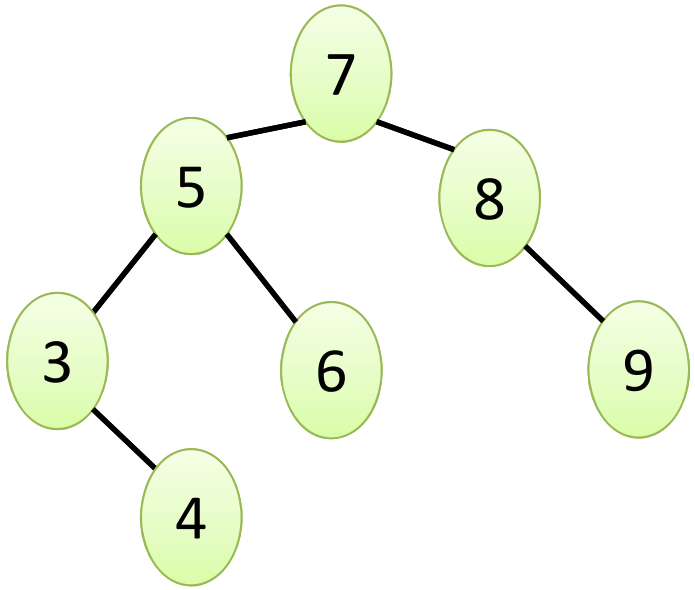
AVL木: 動作例

- 6を挿入



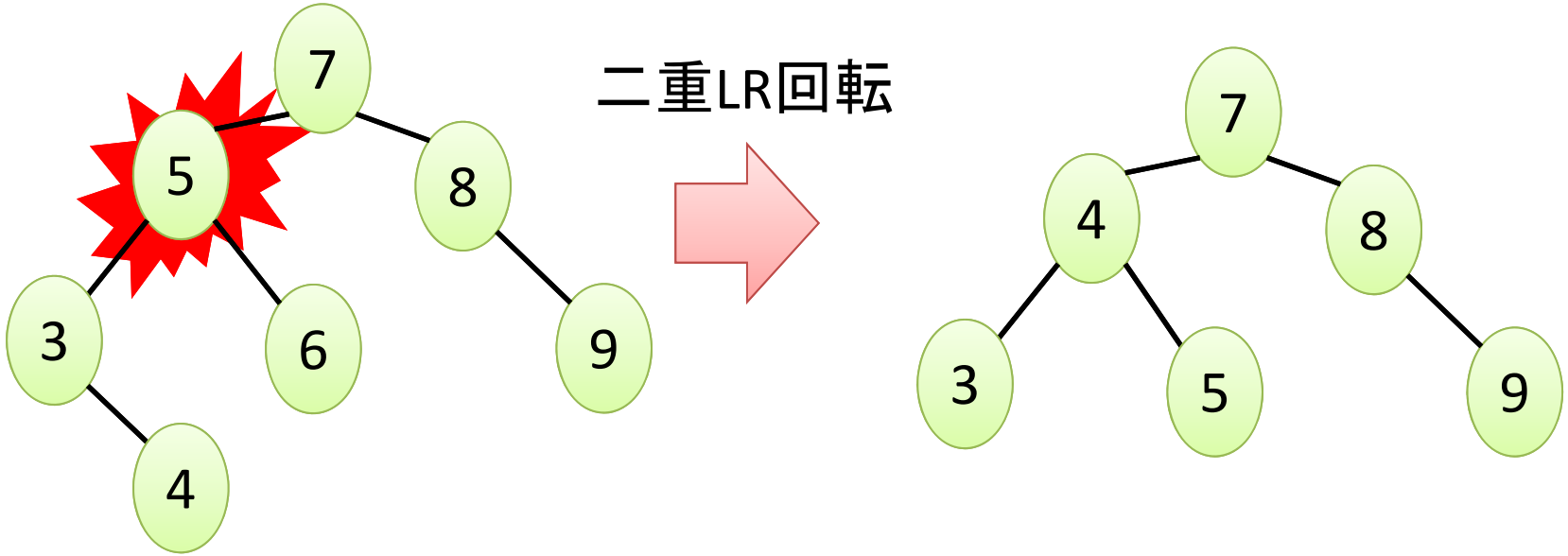
AVL木: 動作例

- 4を挿入



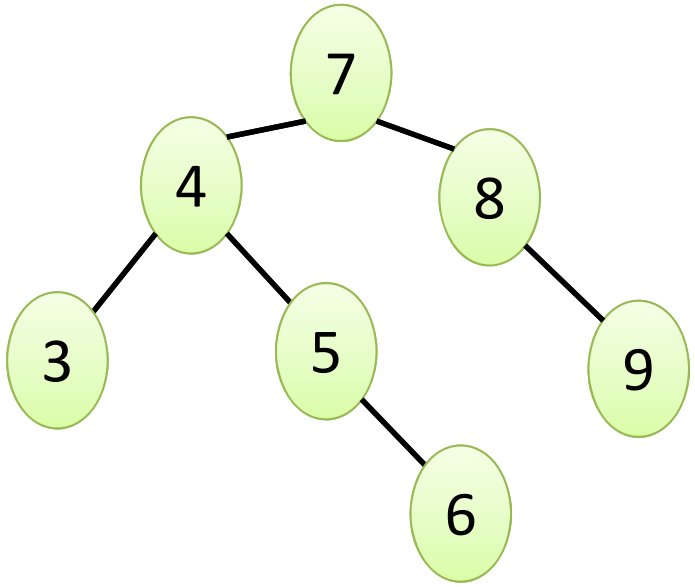
AVL木: 動作例

- 6を削除



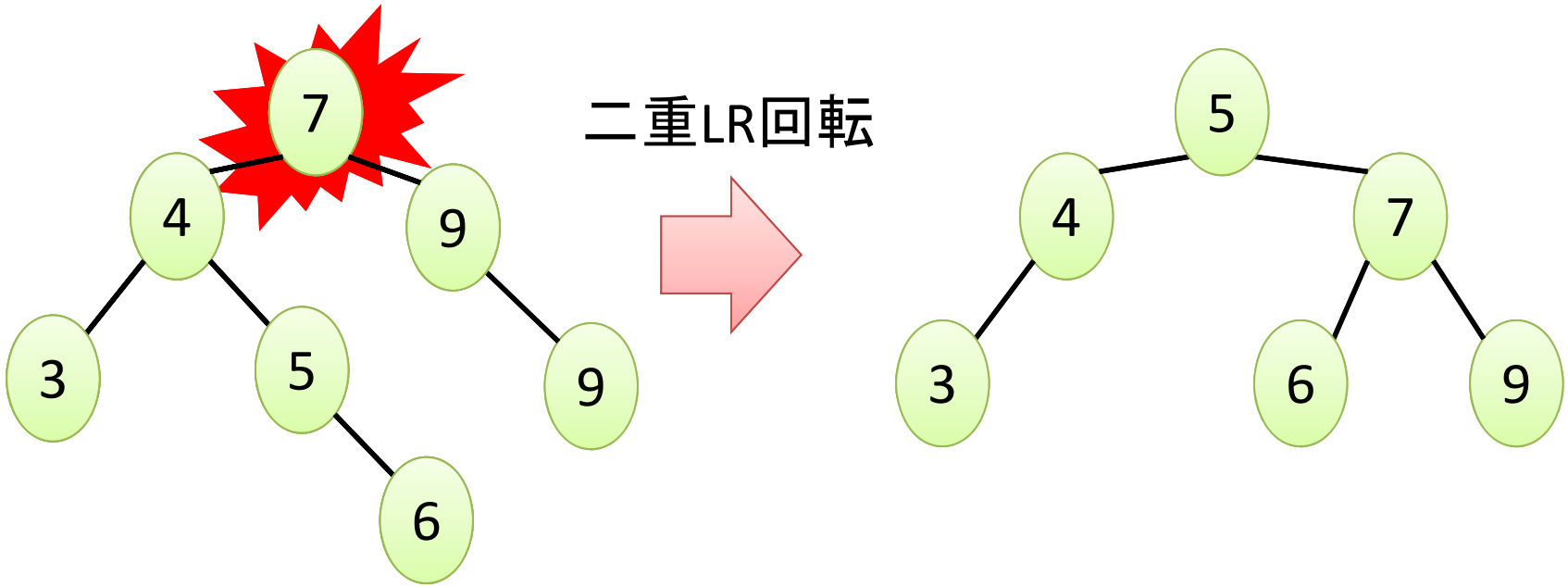
AVL木: 動作例

- 6を挿入



AVL木: 動作例

- 8を削除



平衡2分探索木の効率

- 探索: $O(\log n)$
- 挿入と削除: $O(\log n)$回の回転操作
  - 1回の回転操作にかかる時間は定数時間

- まとめると平衡2分探索木では挿入・削除・探索が全て$O(\log n)$で実行
  - nは木に蓄えられたデータの個数

ミニ演習

- 値1を持つノードのみを持つAVL木に，
  2, 3, 4, 5, 6, … と順番にデータを追加せよ

- 普通の2分探索木で同じことをして，
  できる木の深さを比べよ