

ต้นไม้เอวีแอล

(AVL Tree)

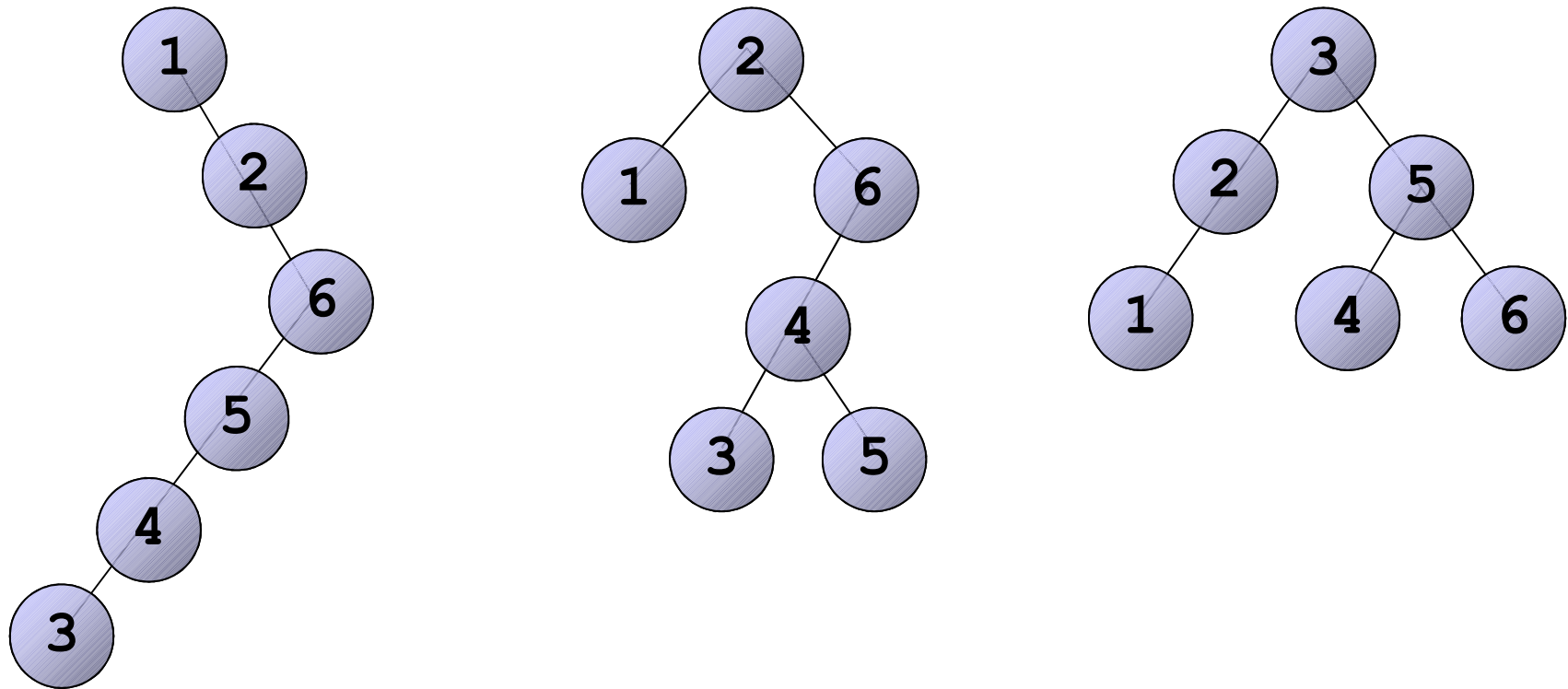
สมชาย ประสิทธิ์จตุระกุล

หัวข้อ

- นิยามต้นไม้เอวีแอล
- การวิเคราะห์ความสูงของต้นไม้เอวีแอล
- การปรับต้นไม้เอวีแอลให้สูงสมดุล
- กระบวนการหมุนปม

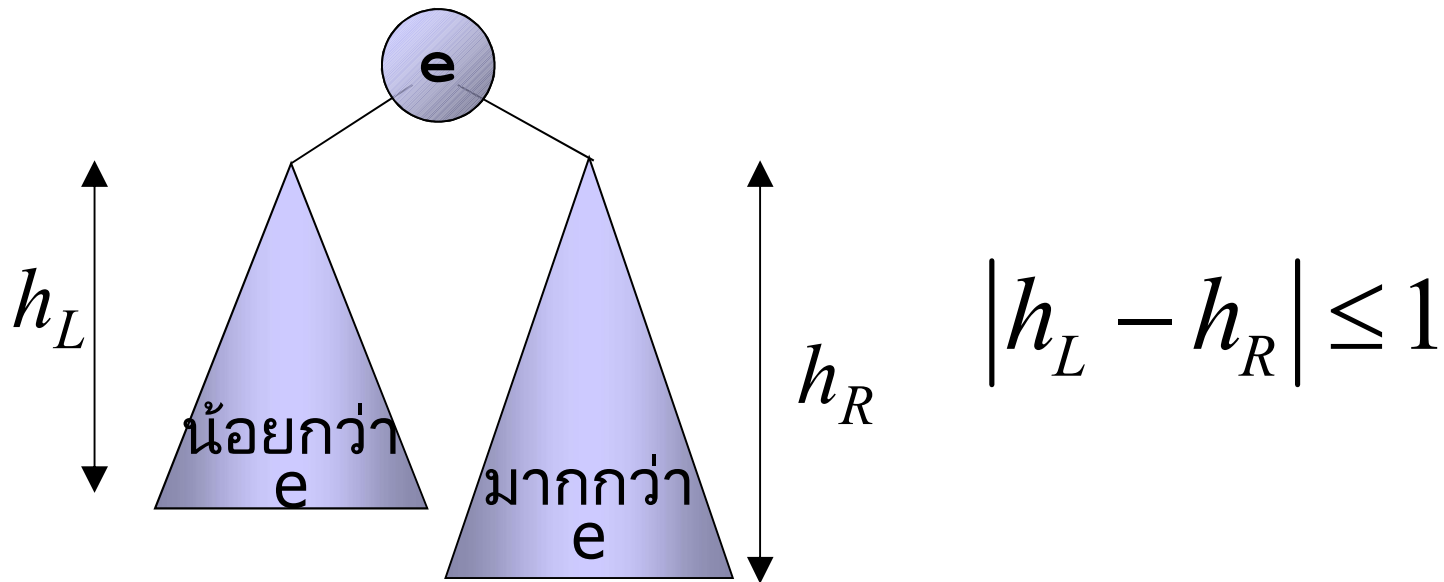
ต้นไม้ค้นหาแบบทวิภาค

- เวลาการทำงานเป็น $O(h)$
- $\lfloor \log_2 n \rfloor \leq h \leq n - 1$
- โหนดดีทำงานเร็ว $O(\log n)$, โหนดร้ายทำงานช้า $O(n)$



ต้นไม้เอวีแอล

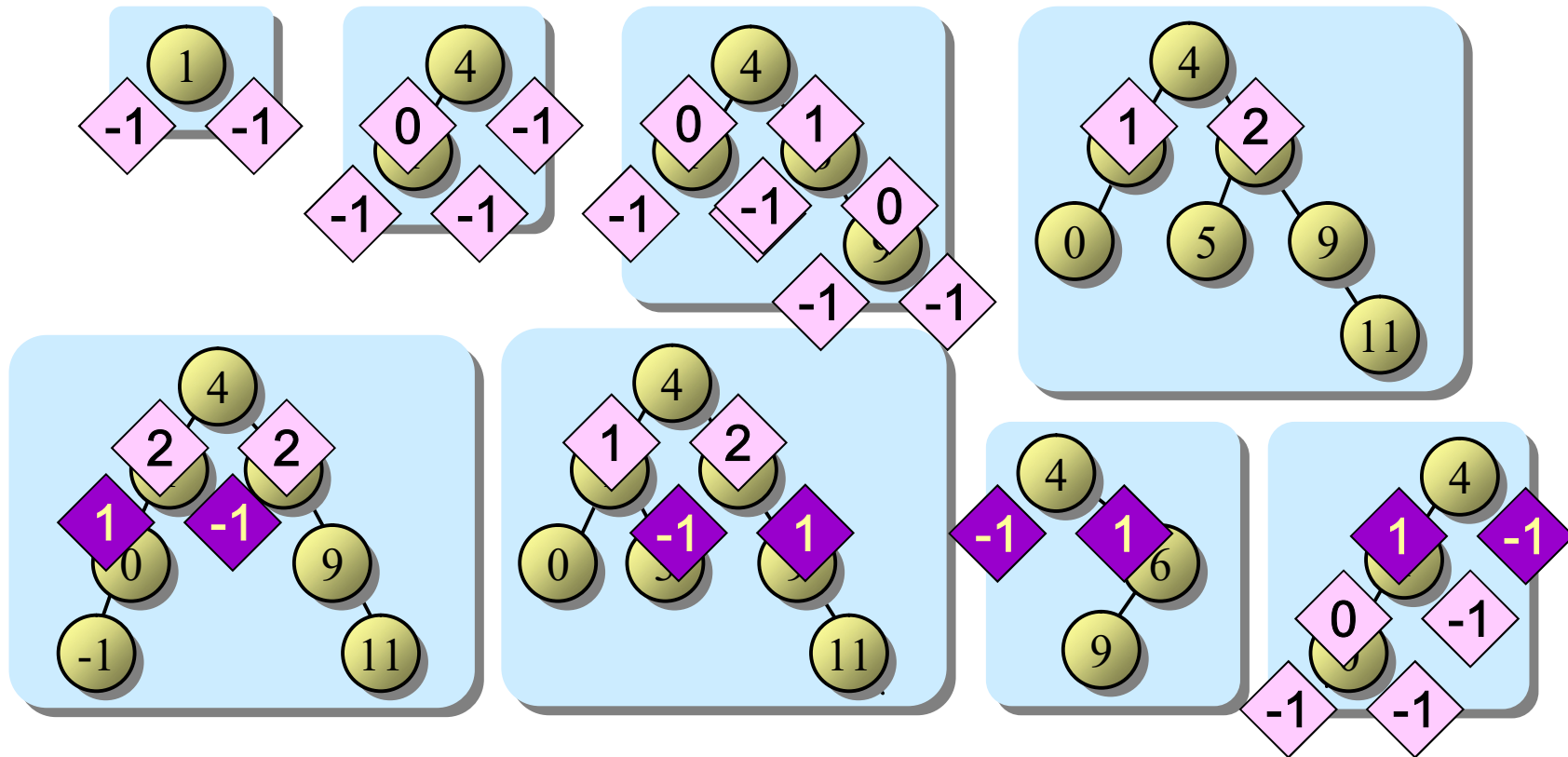
- AVL = Binary Search Tress + กฎความสูงสมดุล



ต้นไม้ย่อยทุกต้นต้องเป็นไปตามกฎ

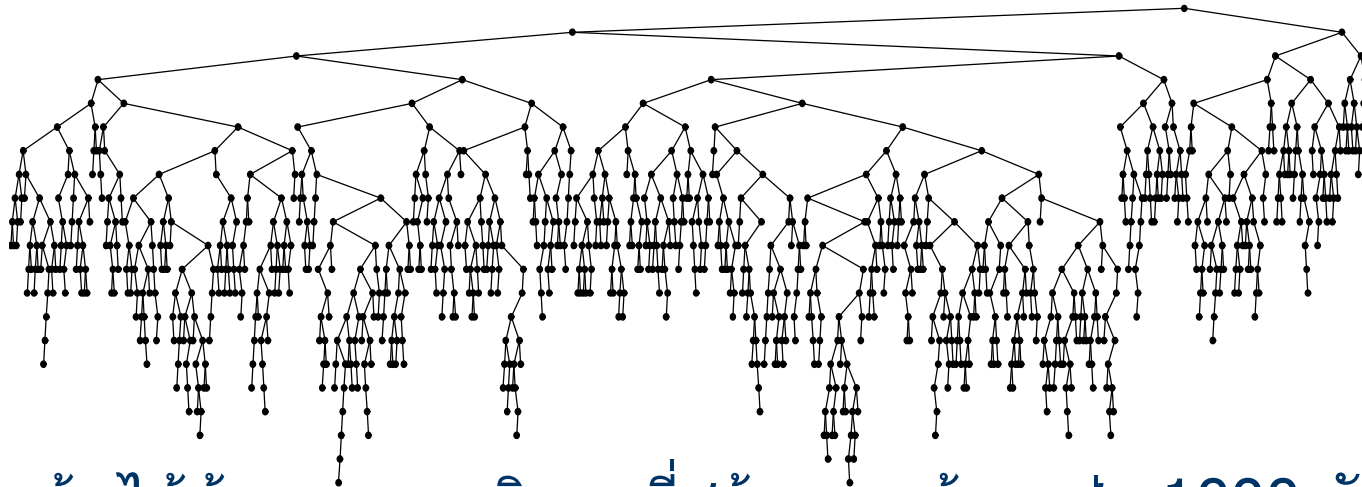
AVL : **A**delson-**V**elskii and **L**andis

ตัวอย่างต้นไม้เอวีแอล

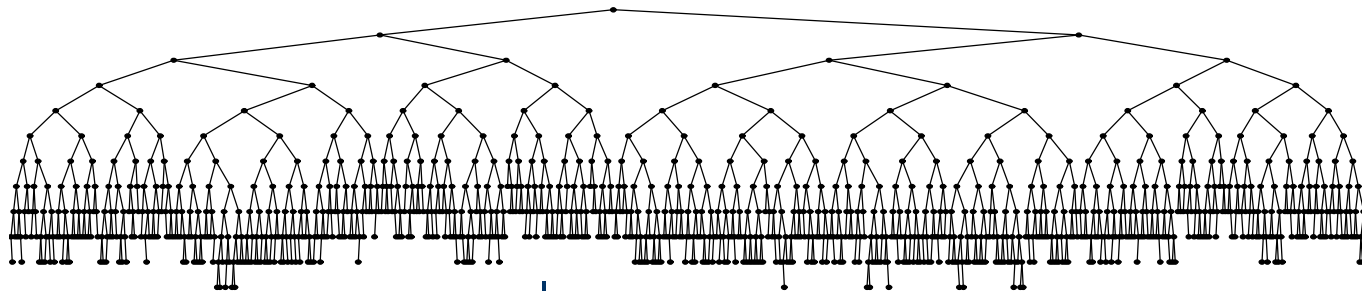


ต้นไม้ว่าง (null) สูง -1

ต้นไม้ค้นหาแบบทวิภาคกับเอวีแอล



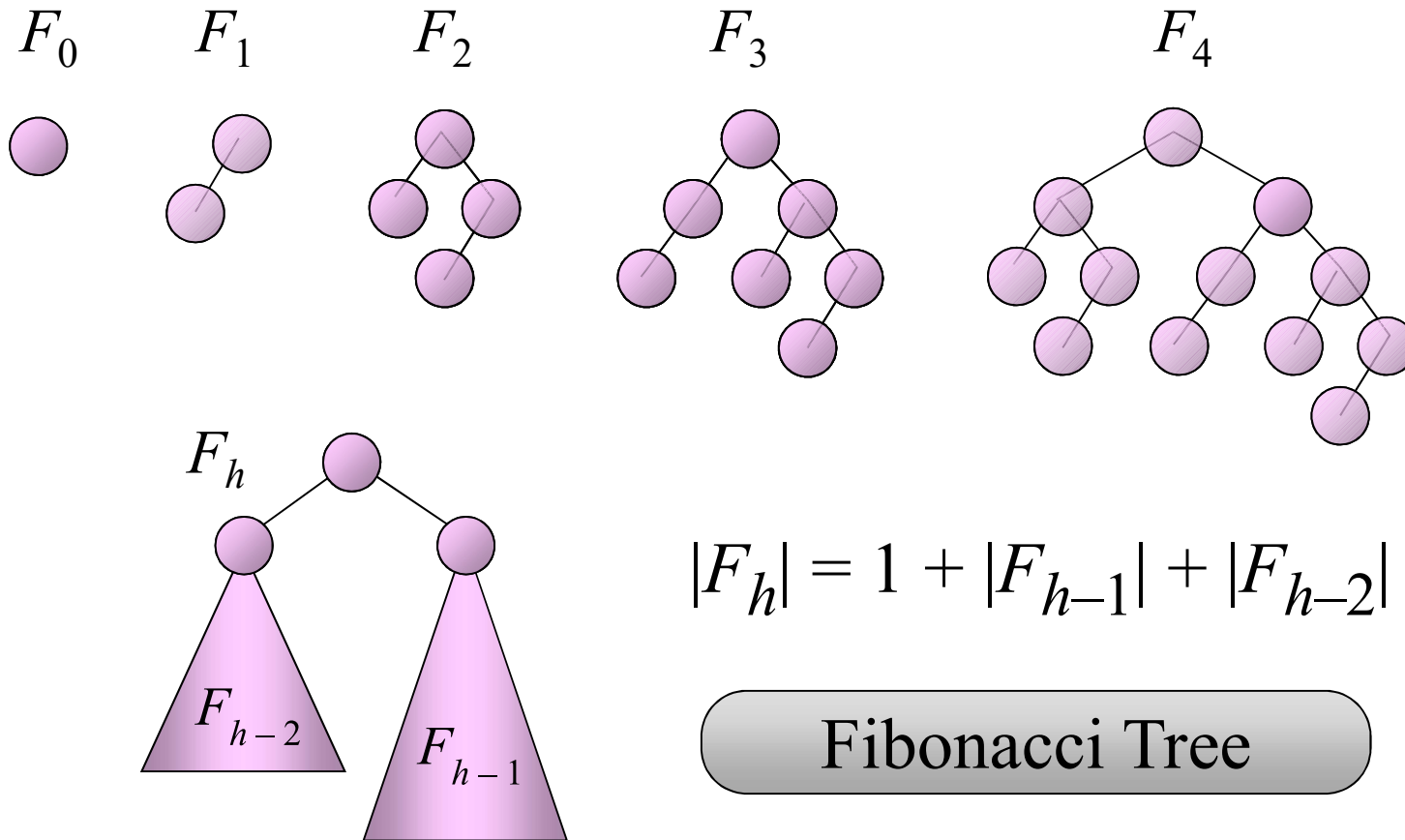
ต้นไม้ค้นหาแบบทวิภาคที่สร้างจากข้อมูลสุ่ม 1000 ตัว



ต้นไม้เอวีแอลที่สร้างจากข้อมูลสุ่ม 1000 ตัว

ต้นไม้เอวี่แอลสูงเท่าใด ?

- ให้ F_h คือต้นไม้เอวี่แอลซึ่งสูง h ที่มีจำนวนปมน้อยที่สุด



$$|F_h| = 1 + |F_{h-1}| + |F_{h-2}|$$

Fibonacci Tree

ความสูงของต้นไม้ฟีโบนัคชี

$$|F_h| = 1 + |F_{h-1}| + |F_{h-2}|$$

$$n_h = 1 + n_{h-1} + n_{h-2} \quad h \geq 2, \quad n_0 = 1, n_1 = 2$$

$$n_h = \alpha_1 \phi^h + \alpha_2 \hat{\phi}^h - 1, \quad \phi = 1.618..., \quad \hat{\phi} = -0.618$$

$$n_h \approx \alpha_1 \phi^h$$

$$h \approx \frac{1}{\log_2 \phi} (\log_2 n_h)$$

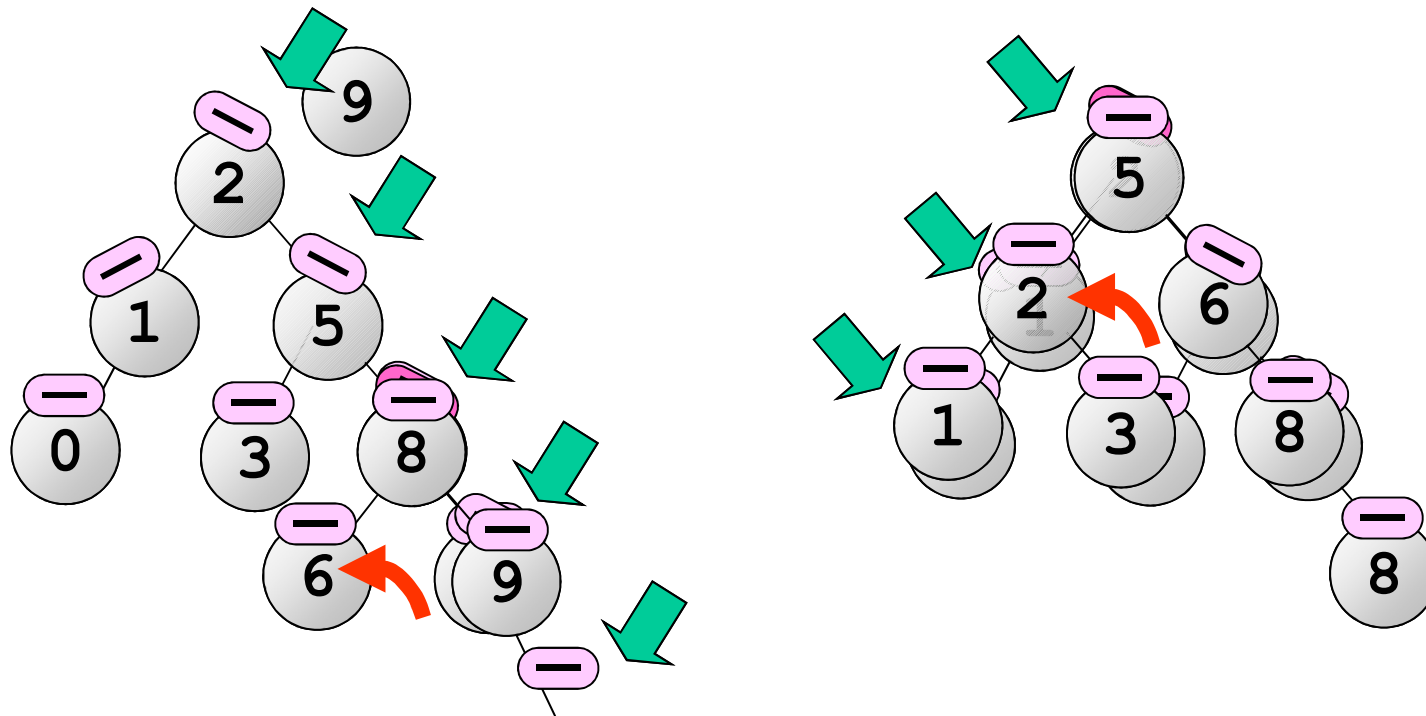
สรุป :
ต้นไม้เอวีแอลที่มี n ปม
สูงไม่เกิน $1.44 \log_2 n$

$$h \approx 1.44 (\log_2 n_h)$$

$$\lfloor \log_2 n \rfloor \leq h_{\text{AVL}} \leq 1.44 \log_2 n$$

ทำอย่างไรให้เป็นไปตามกฎของ AVL

- การเพิ่ม/ลบข้อมูลทำเหมือน BSTree
- แต่หลังการเพิ่ม/ลบ อาจทำให้ผิดกฎสูงสมดุล
- ถ้าผิดกฎ ต้องปรับต้นไม้



map_avl

```
template <typename KeyT,  
          typename MappedT,  
          typename CompareT = std::less<KeyT> >  
class map_avl {  
protected:  
  
    class node {  
        friend class map_bst;  
        ...  
    };  
  
    class tree_iterator {  
        ...  
    };  
  
public:  
    ...  
  
};
```

เหมือน map_bst

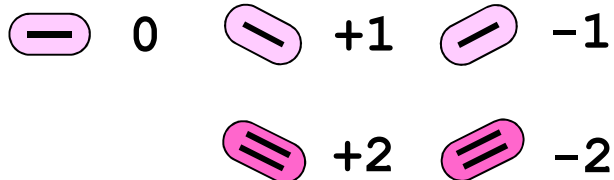
node

```
class node {  
    friend class map_avl;  
protected:  
    ValueT data;  
    node *left;  
    node *right;  
    node *parent;  
    int height;  
  
    node() :  
        data( ValueT() ), left( NULL ), right( NULL ),  
        parent( NULL ), height(0) { }  
  
    node(const ValueT& data, node* left,  
        node* right, node* parent) : data(data),  
        left(left), right(right), parent(parent) {  
        set_height();  
    }  
}
```

แต่ละปมมีความสูงกำกับ

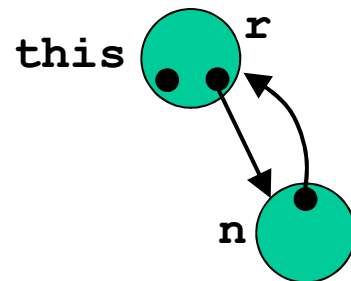
node

```
class node {  
    friend class map_avl;  
protected:  
    ...  
    int get_height(node *n) { // 100 ?  
        return (n == NULL ? -1 : n->height);  
    }  
    void set_height() {  
        int hL = get_height(this->left);  
        int hR = get_height(this->right);  
        height = 1 + (hL > hR ? hL : hR);  
    }  
    int balance_value() {  
        return get_height(this->right) -  
               get_height(this->left);  
    }  
}
```

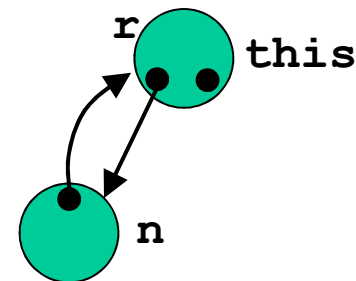


node

```
class node {  
    friend class map_avl;  
protected:  
    ...  
    void set_left(node *n) {  
        this->left = n;  
        if (n != NULL) this->left->parent = this;  
    }  
    void set_right(node *n) {  
        this->right = n;  
        if (n != NULL) this->right->parent = this;  
    }  
};
```



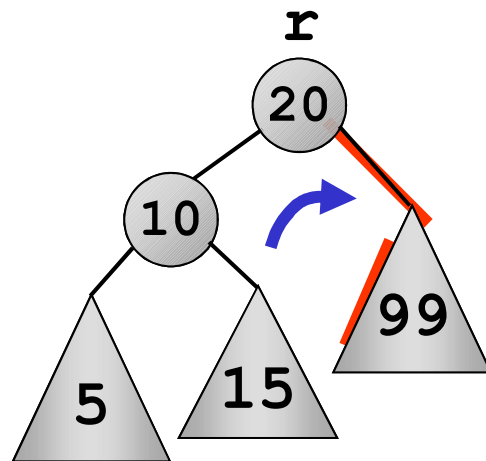
`r->set_right(n);`



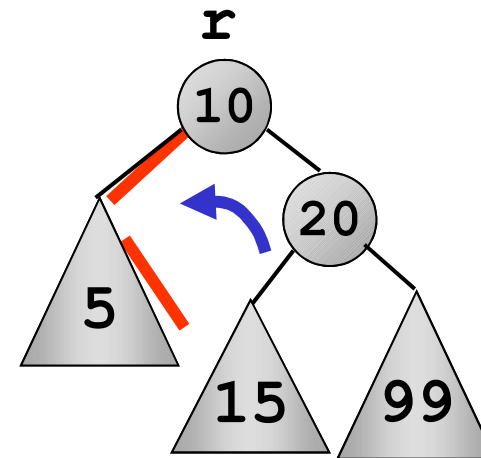
`r->set_left(n);`

การหมุนปม

- การปรับต้นไม้อาศัยการหมุน (rotation)
- การหมุนปมยังคงรักษาความเป็นต้นไม้ค้นหา



`rotate_left_child(r)`

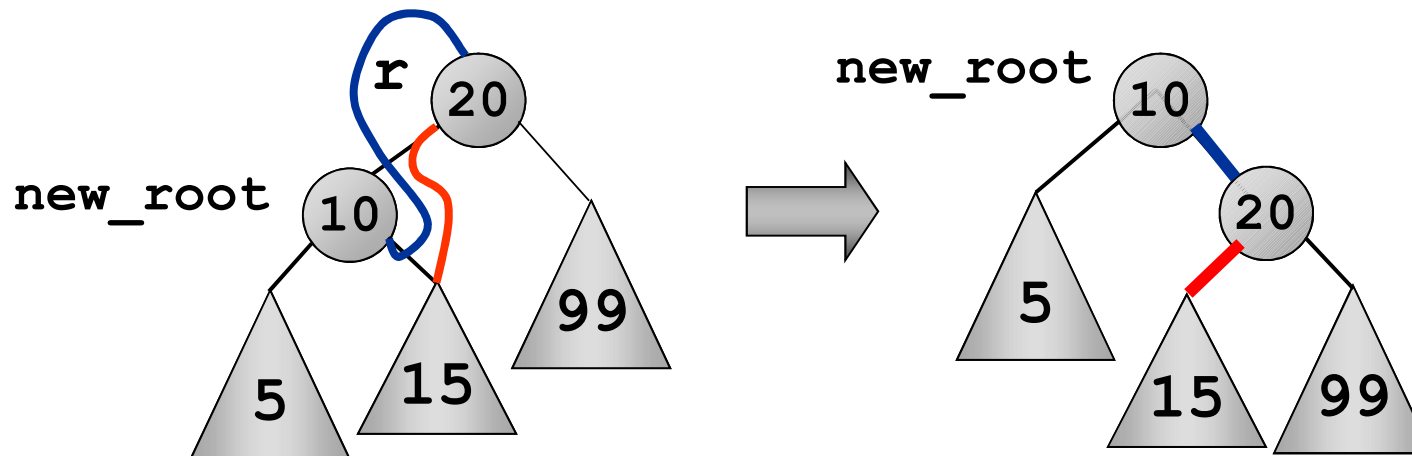


`rotate_right_child(r)`

rotate_left_child(r)

```
node *rotate_left_child(node *r) {  
    node *new_root = r->left;  
    r->set_left(new_root->right);  
    new_root->set_right(r);  
  
    new_root->right->set_height();  
    new_root->set_height();  
    return new_root;  
}
```

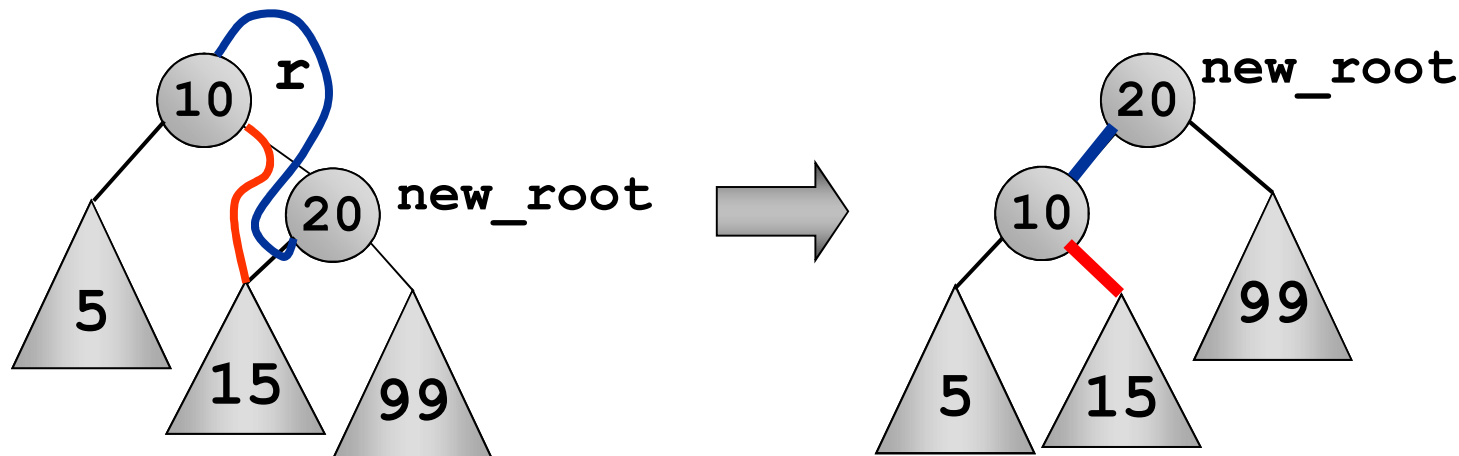
$\Theta(1)$



rotate_right_child(r)

```
node *rotate_right_child(node * r) {  
    node * new_root = r->right;  
    r->set_right(new_root->left);  
    new_root->set_left(r);  
  
    new_root->left->set_height();  
    new_root->set_height();  
    return new_root;  
}
```

$\Theta(1)$



insert และ erase ใช้ rebalance

```
node* insert(const ValueT& val, node *r, node * &ptr) {  
    ... // same as insert in map_bst
```

```
    r = rebalance(r);
```

เพิ่มตามปกติ แล้วค่อยปรับ

```
    return r;
```

```
}
```

```
node *erase(const KeyT &key, node *r) {  
    ... // same as erase in map_bst
```

```
    r = rebalance(r);
```

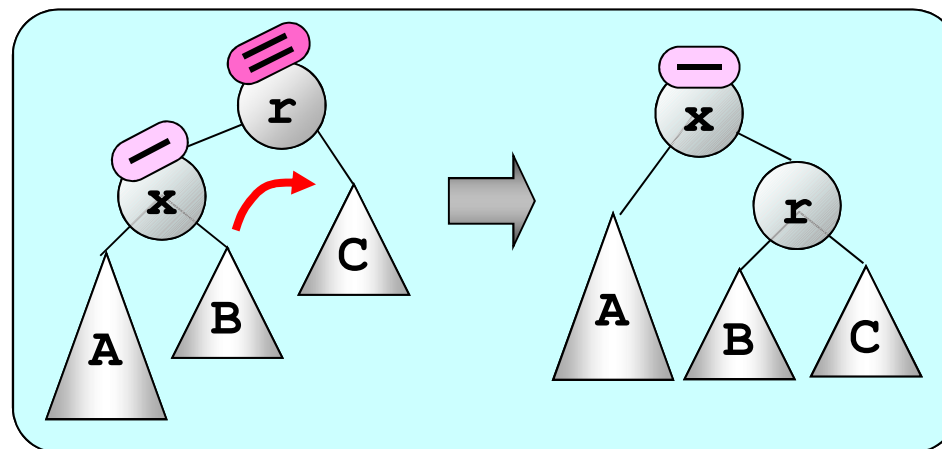
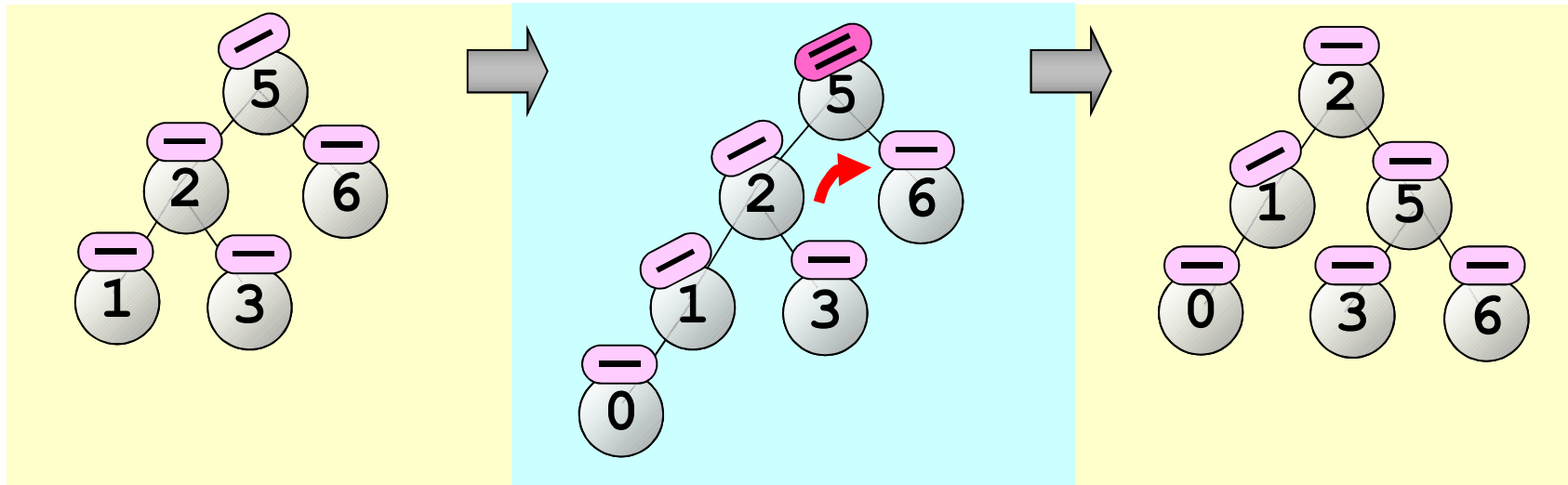
ลบตามปกติ แล้วค่อยปรับ

```
    return r;
```

```
}
```

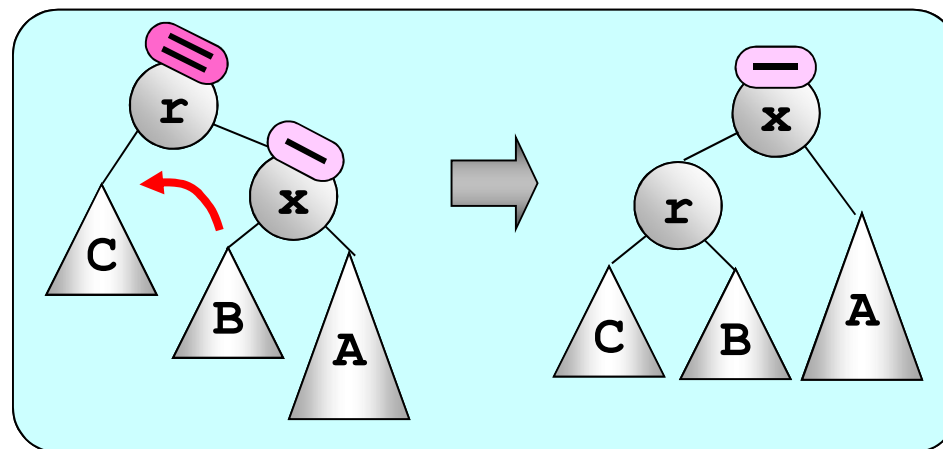
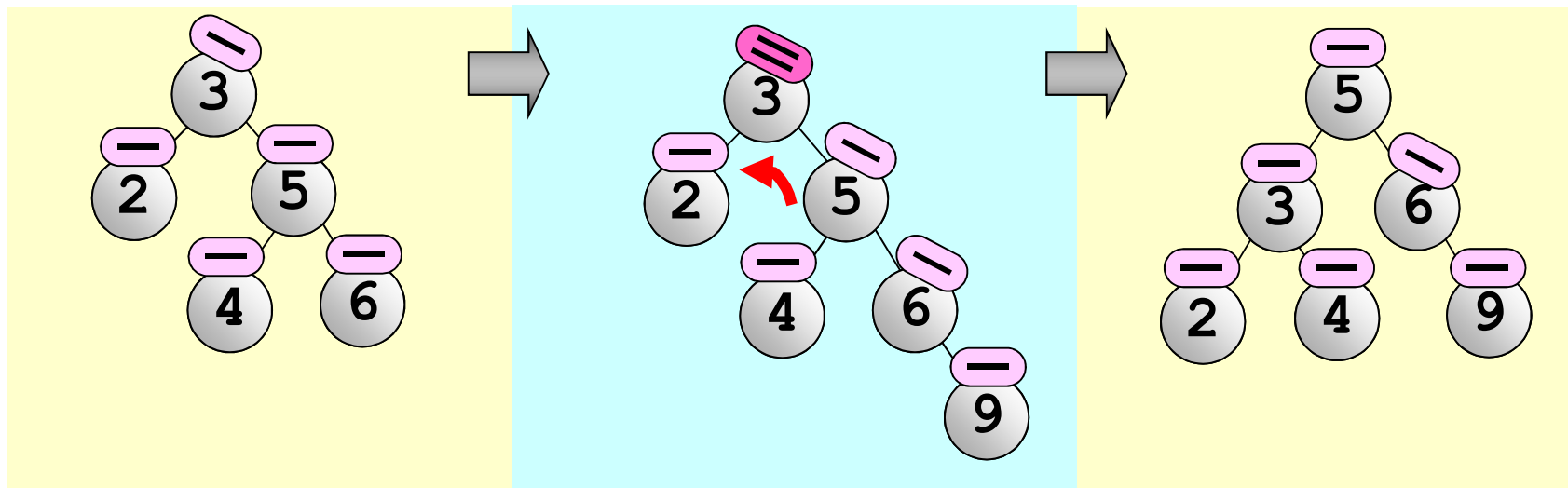
rebalance มี 4 กรณี

rebalance : กรณีที่ 1



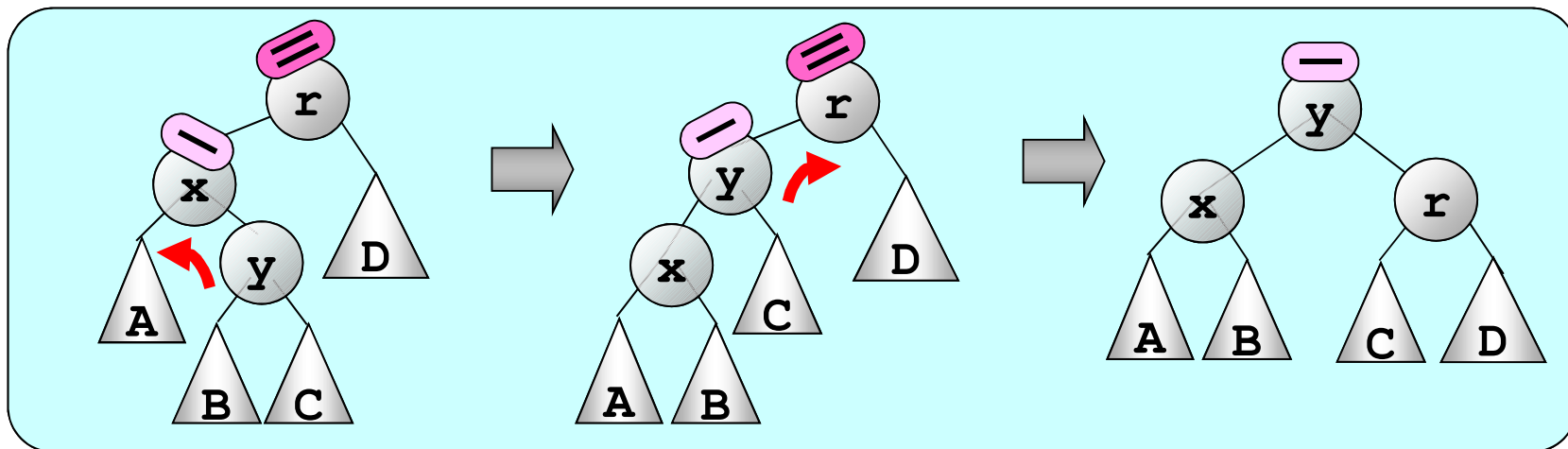
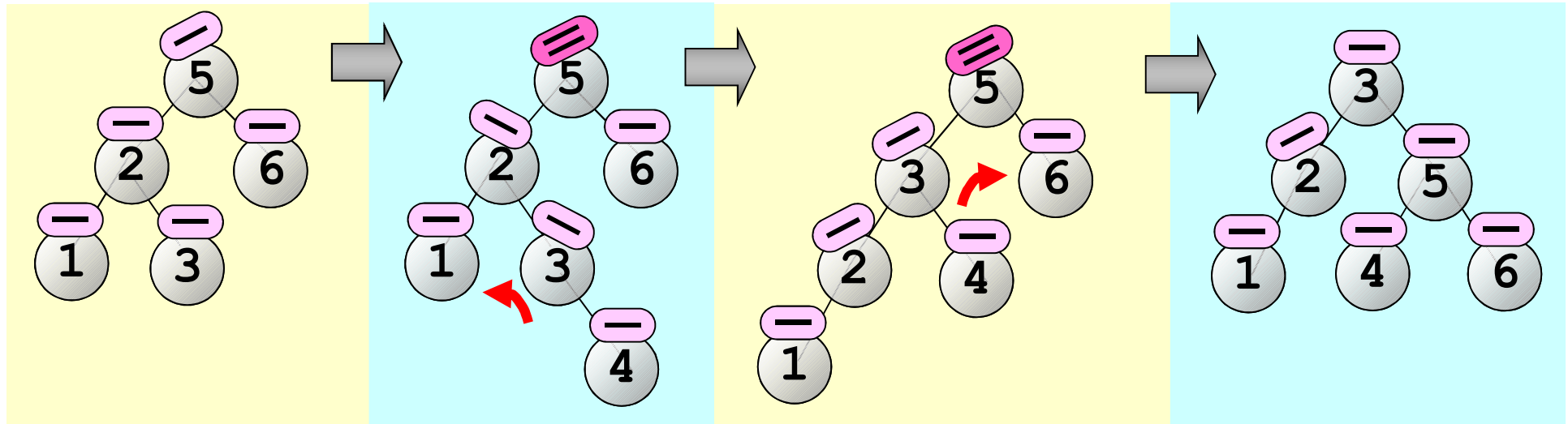
`rotate_left_child(r)`

rebalance : กรณีที่ 2



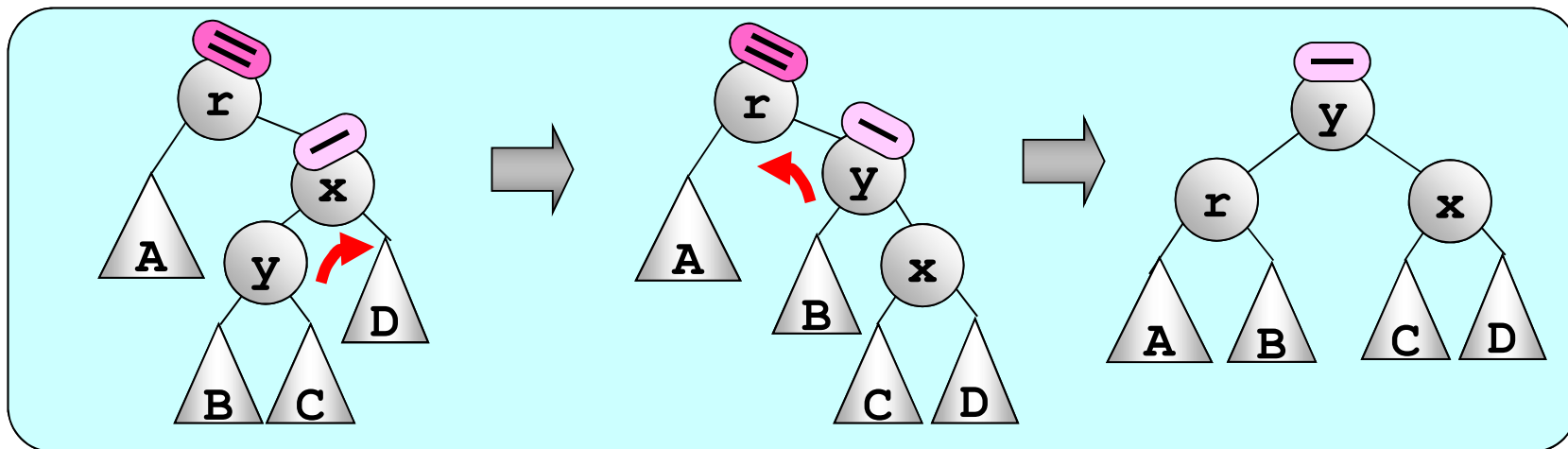
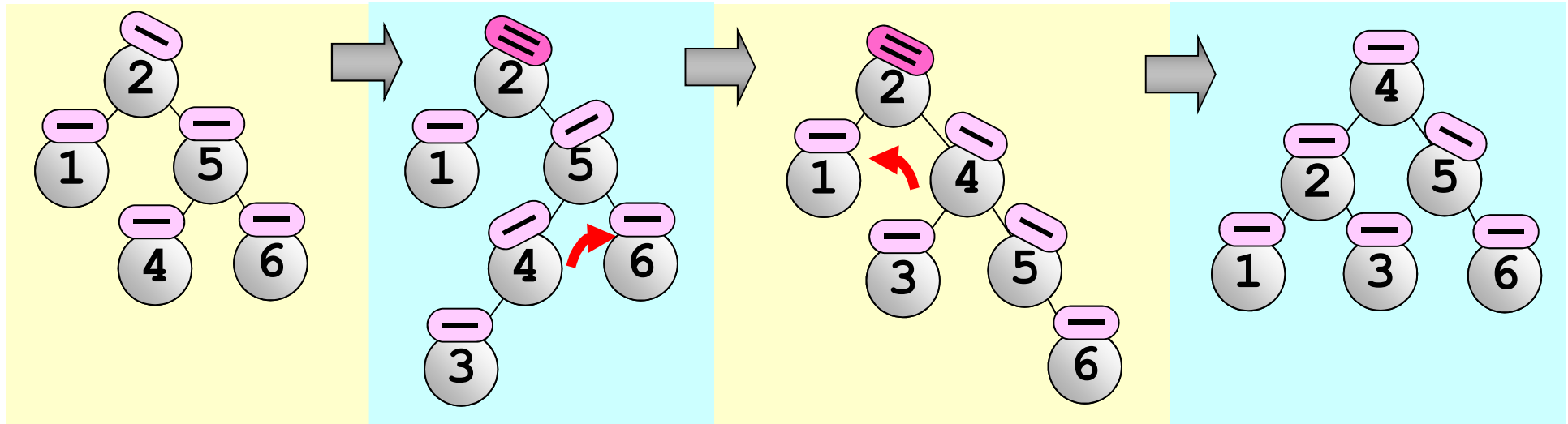
`rotate_right_child(r)`

rebalance : กรณีที่ 3



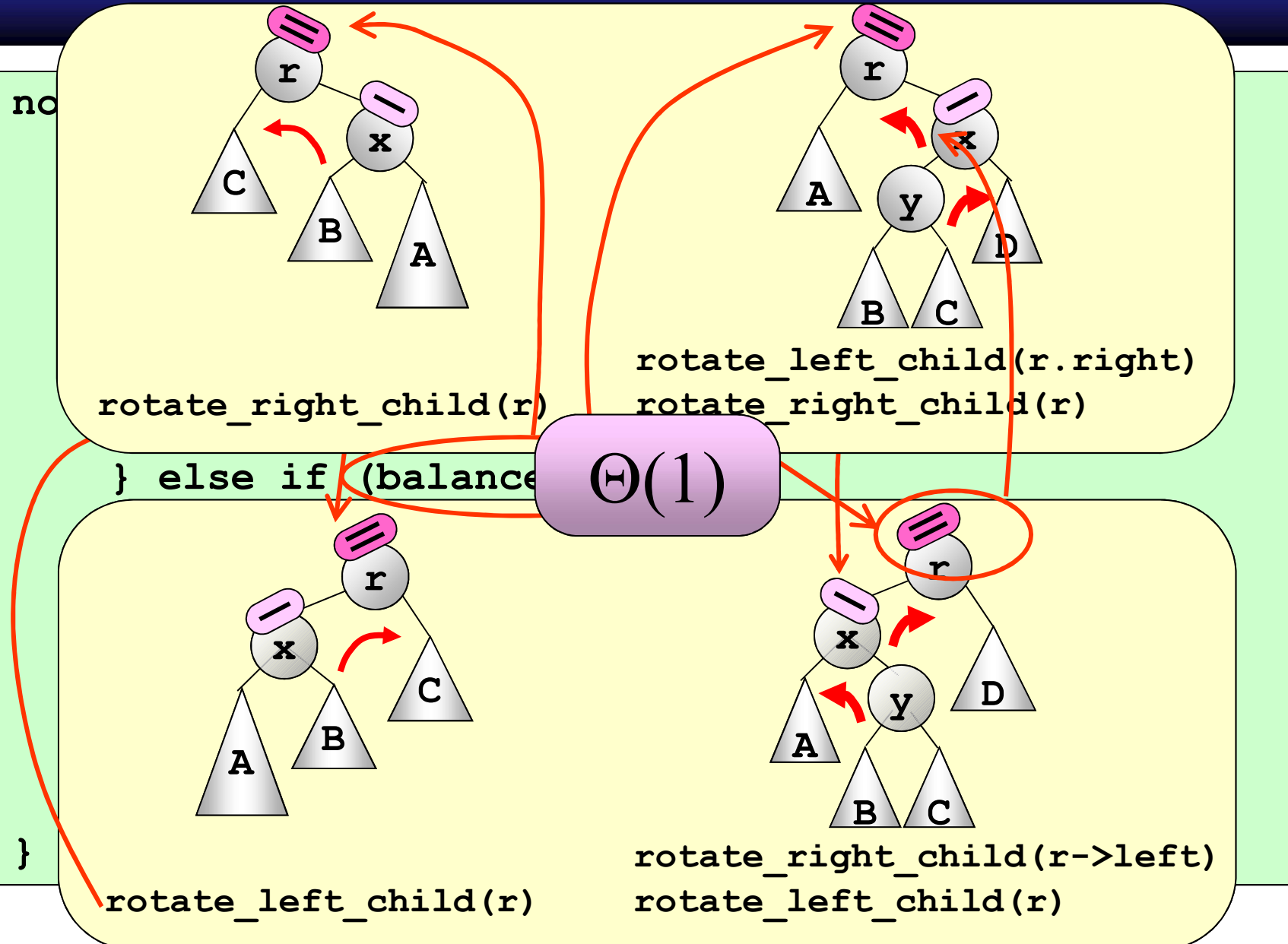
`rotate_right_child(r->left)` `rotate_left_child(r)`

rebalance : กรณีที่ 4



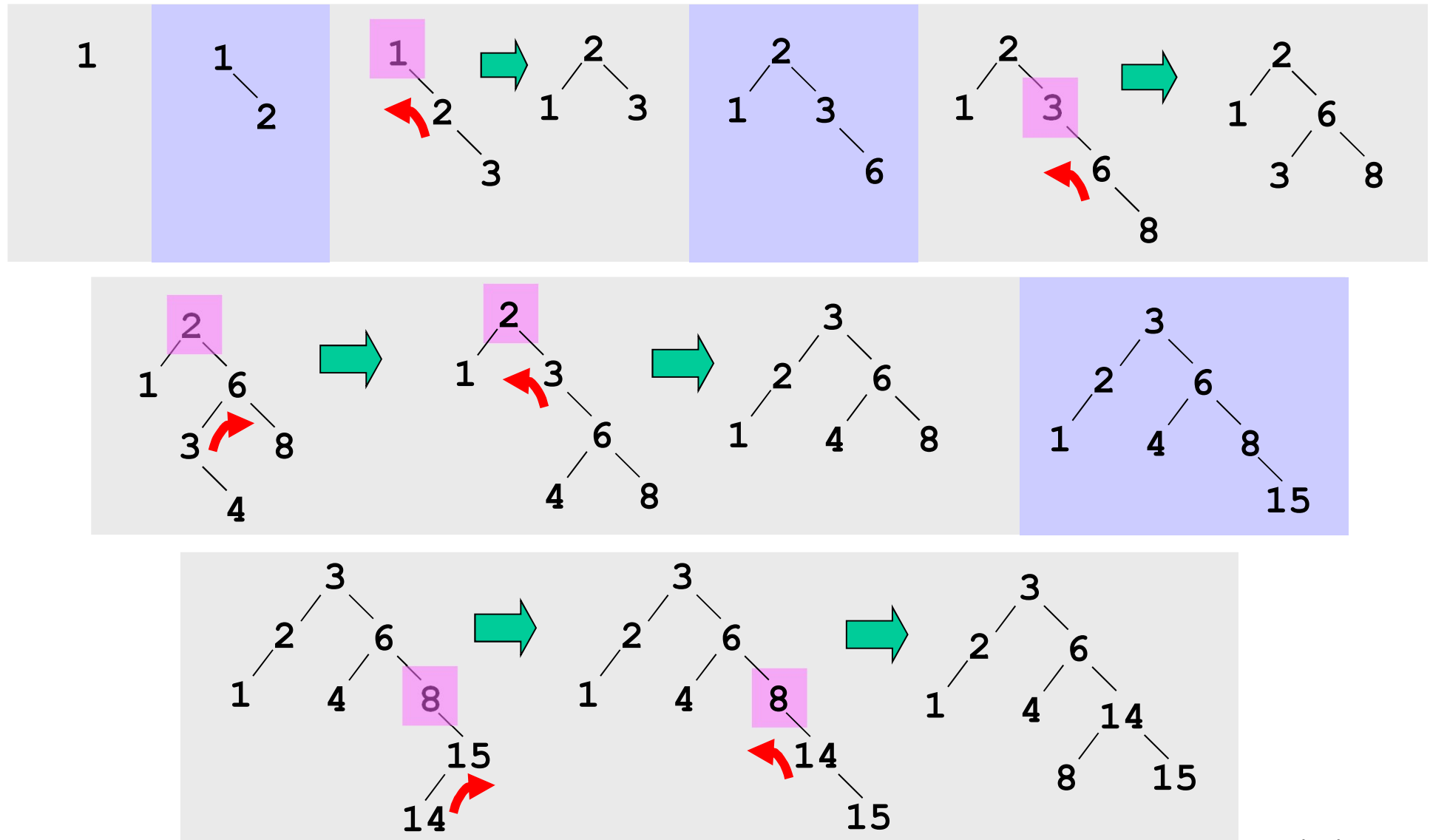
`rotate_left_child(r->right)` `rotate_right_child(r)`

rebalance



ตัวอย่าง

1, 2, 3, 6, 8, 4, 15, 14



สรุป

- ต้นไม้เอวีแอลคือต้นไม้ค้นหาที่ถูควบคุมความสูง
- ผลต่างความสูงของลูกสองข้างห้ามเกินหนึ่ง
- พิสูจน์ได้ว่า $\lfloor \log_2 n \rfloor \leq h < 1.44 \log_2 n$
- แต่ละปมเก็บความสูงไว้ตรวจสอบ
- ถ้าหลังเพิ่ม/ลบข้อมูลแล้วผิดกฎ, ให้ปรับต้นไม้
- การปรับต้นไม้อาศัยการหมุนปม
- เวลาการทำงานของ การเพิ่ม ลบ และค้นเป็น $O(\log n)$

ความสูงของต้นไม้ AVL

```
class node {  
    friend class map_avl;  
protected:  
    ValueT data;  
    node *left;  
    node *right;  
    node *parent;  
    int height;
```

```
class node {  
    friend class map_avl;  
protected:  
    ValueT data;  
    node *left;  
    node *right;  
    node *parent;  
    unsigned char height;
```