ต้นไม้เอวีแอล

(AVL Tree)

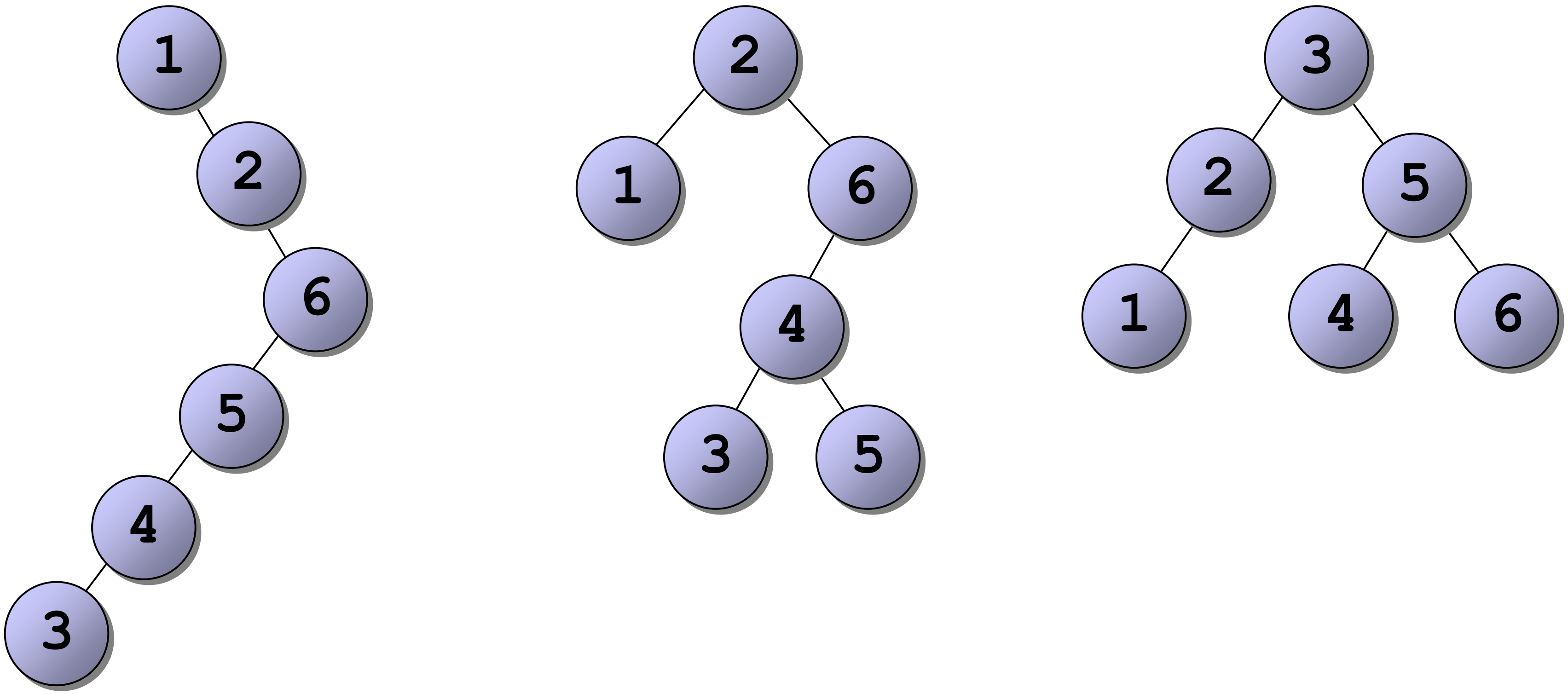
สมชาย ประสิทธิ์จูตระกูล
Translated to English by
Nuttapong Chentanez

Topic

- AVL tree definition
- Height of AVL tree
- Balancing AVL tree
- Tree rotation

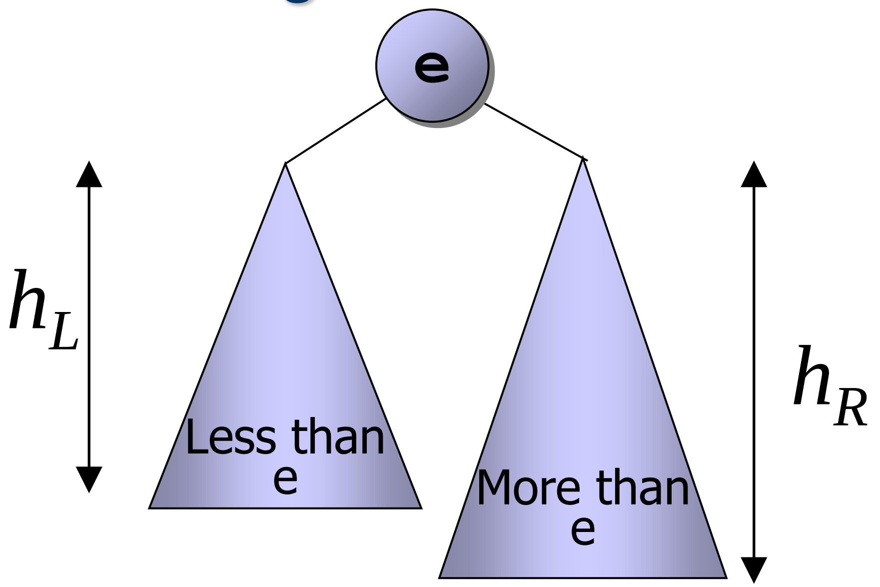
Binary search tree

- Running time is O(h)
- $\lfloor \log_2 n \rfloor \le h \le n-1$
- Best case O(log n), worst case O(n)



AVL Tree

- AVL = Binary Search Tress + height balancing rules

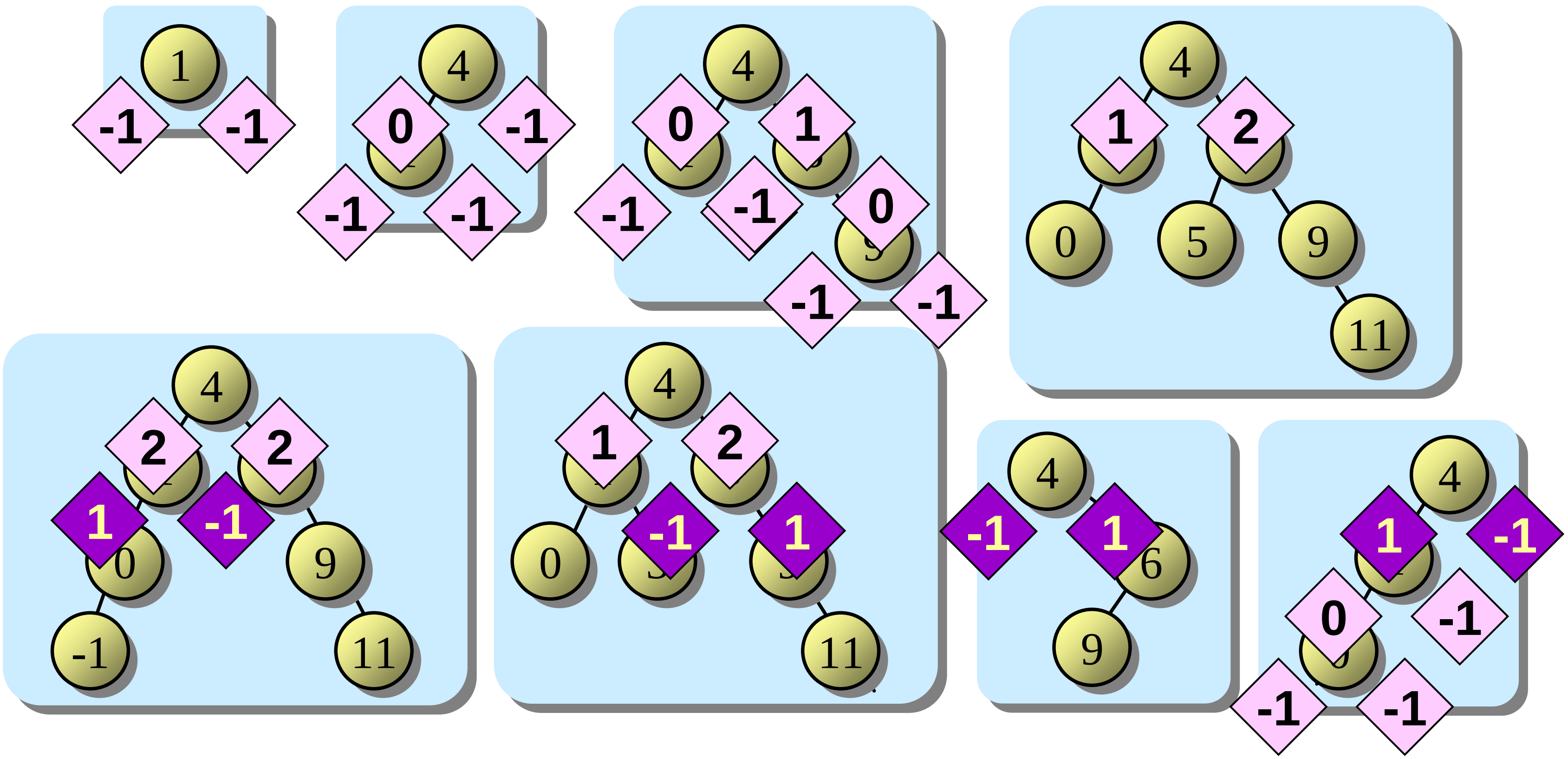


$$|h_L - h_R| \le 1$$

All subtrees must obey this rule

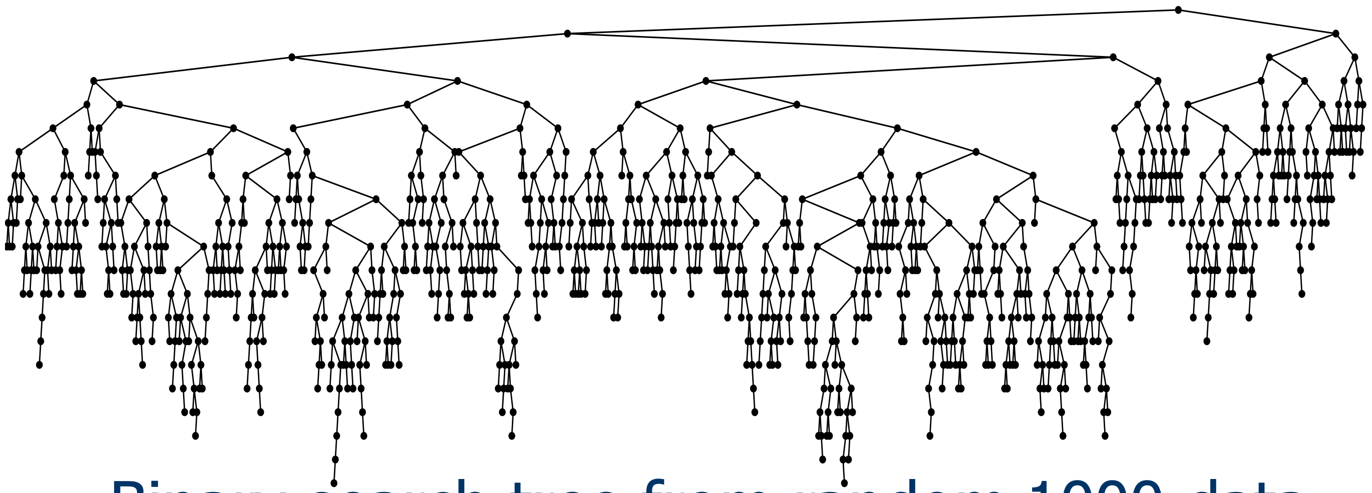
AVL : Adelson-Velskii and Landis

AVL Tree example

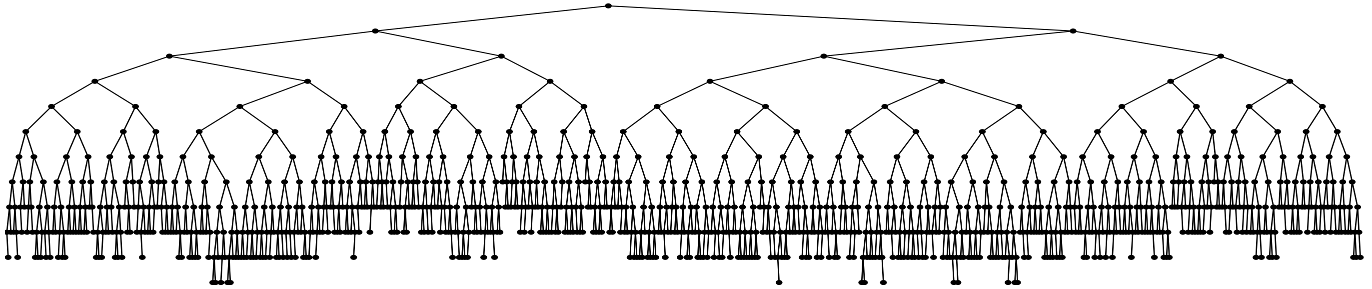


Empty tree (null) has height -1

Binary search tree vs AVL tree



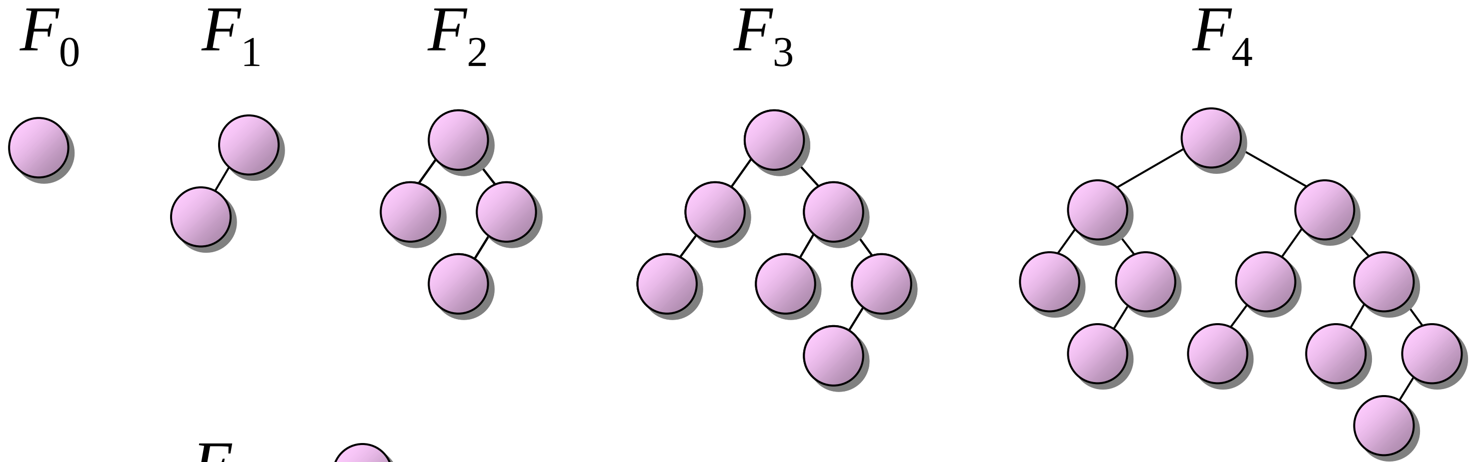
Binary search tree from random 1000 data

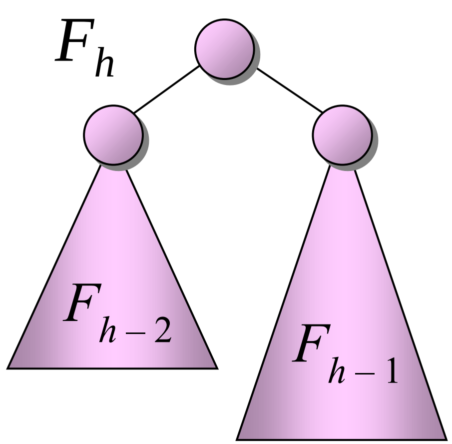


AVL tree from random 1000 data

How tall is AVL tree?

- Let F_h be AVL tree with height h that has lowest number of nodes





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

Fibonacci Tree

Height of Fibonacci Tree

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

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

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

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

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

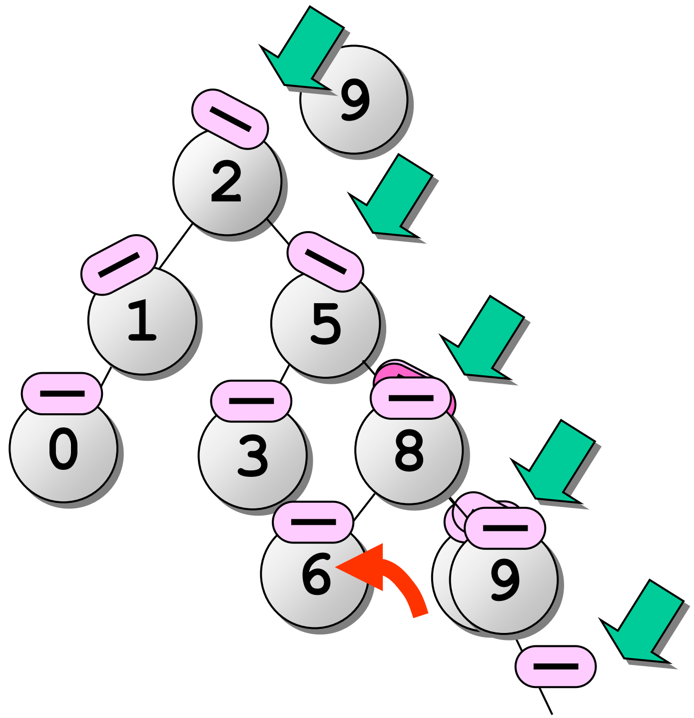
Summary :
AVL tree with n node
is no taller than 1.44 $\log_2 n$

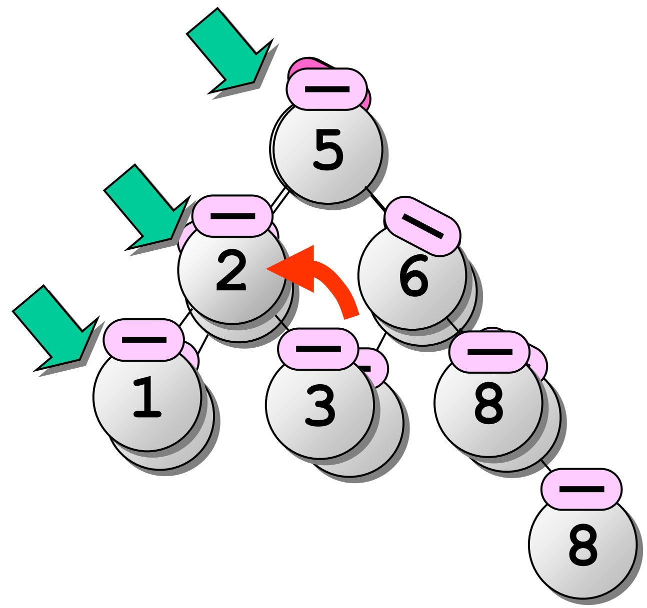
$$h \approx 1.44 \left(\log_2 n_h \right)$$

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

How to satisfy AVL rule?

- Insert/erase like BST
- But insert/erase may violate the rule
- If so, need to adjust the trees





map_avl

```
template <typename KeyT,
          typename MappedT,
          typename CompareT = std::less<KeyT> >
class map_avl {
protected:

  class node {
    friend class map_avl;
    ...
  };

  class tree_iterator {
    ...
  };

public:
  ...

};
```

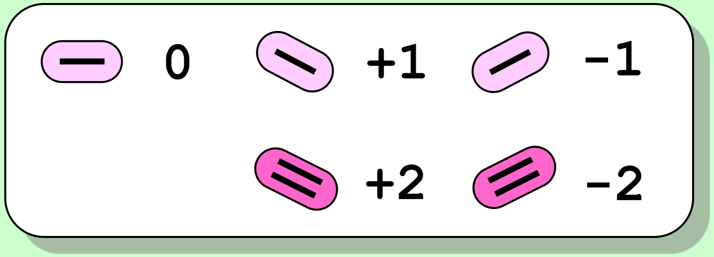
Same as map_bst

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();
    }
```

Each node stores height

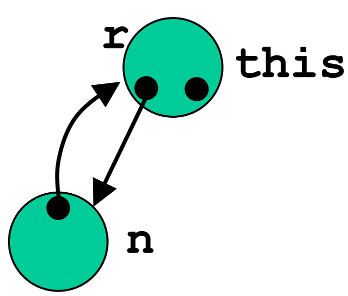
node



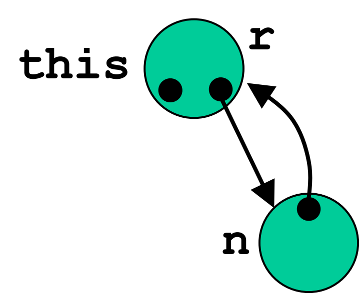

```
class node {
  friend class map_avl;
  protected:
    ...
    int    height;
    ...
    int get_height(node *n) { // ไม่ OO ?
      return (n == NULL ? -1 : n->height);
    }
    void set_height() {
      int hL = get_height(this->left);
      int hR = get_height(this->right);
      height = 1 +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



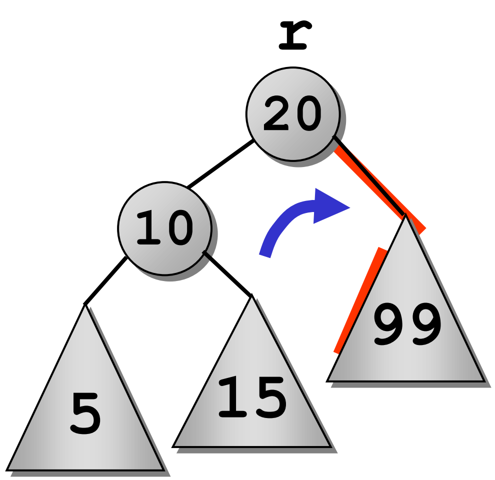
`r->set_left(n);`



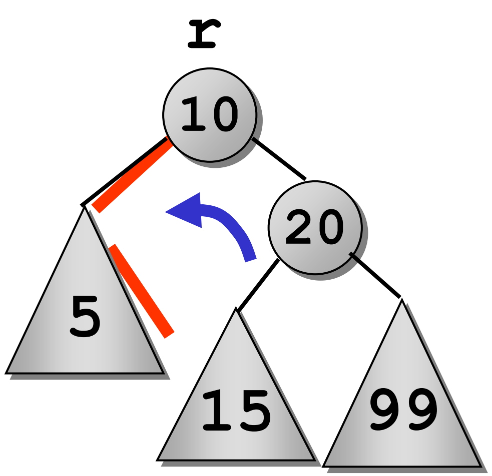
`r->set_right(n);`

Node rotation

- Use rotation to maintain height
- Rotated tree remain a binary search tree



`rotate_left_child(r)`

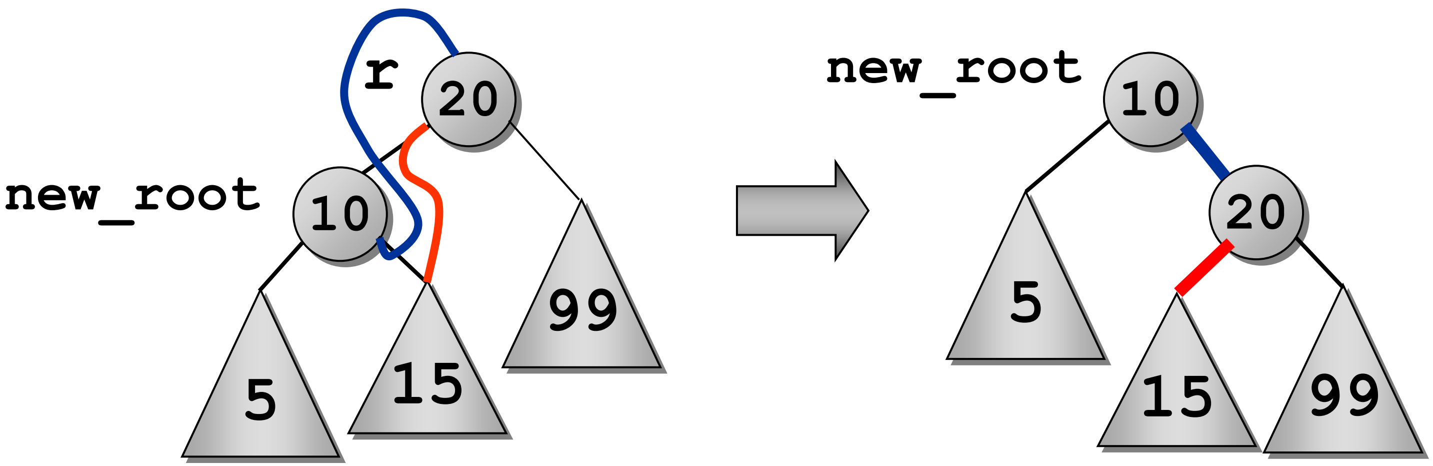


`rotate_right_child(r)`

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;
}
```

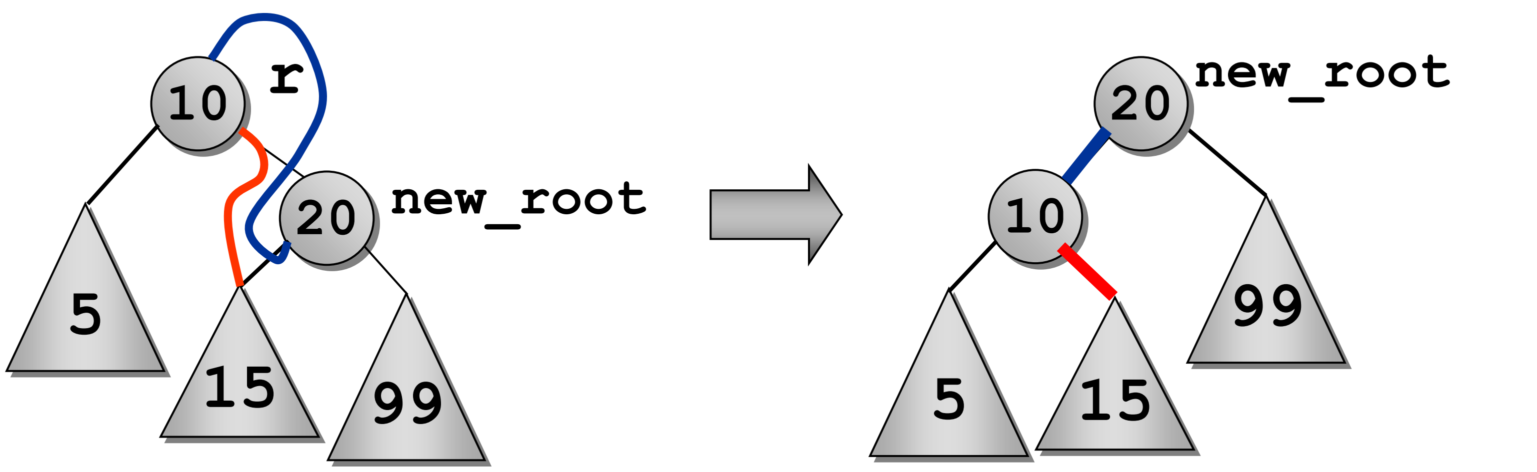
$\Theta(1)$



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 and erase using rebalance

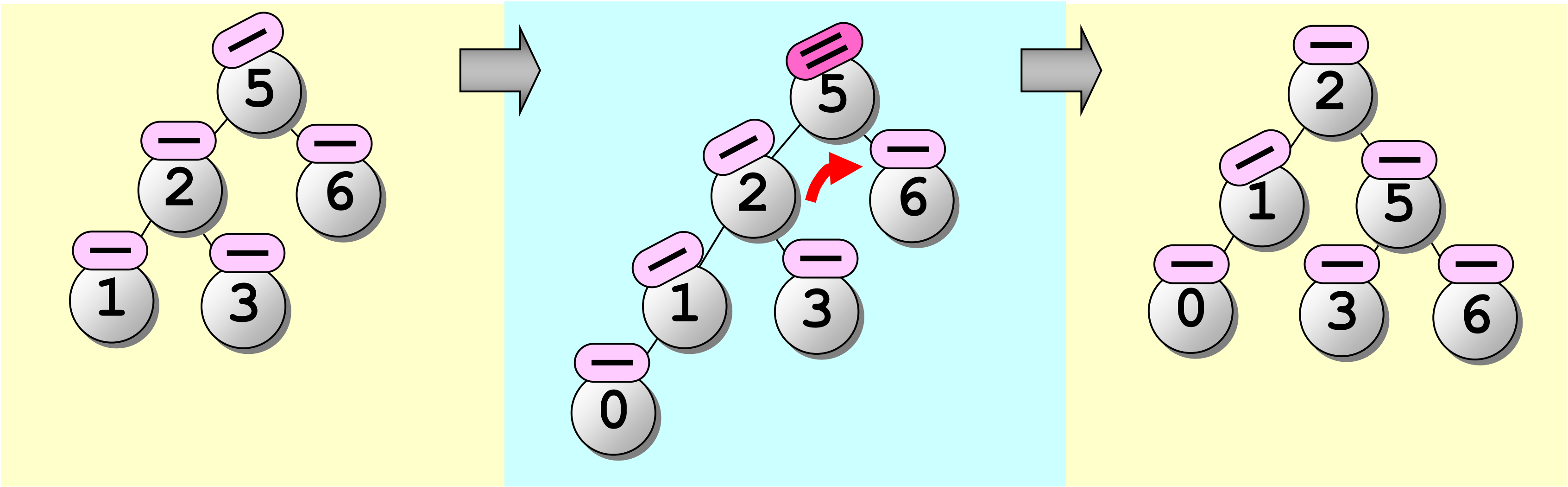
```
node* insert(const ValueT& val, node *r, node * &ptr) {
    ... // same as insert in map_bst

    r = rebalance(r);
    return r;
}

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

    r = rebalance(r);
    return r;
}
```

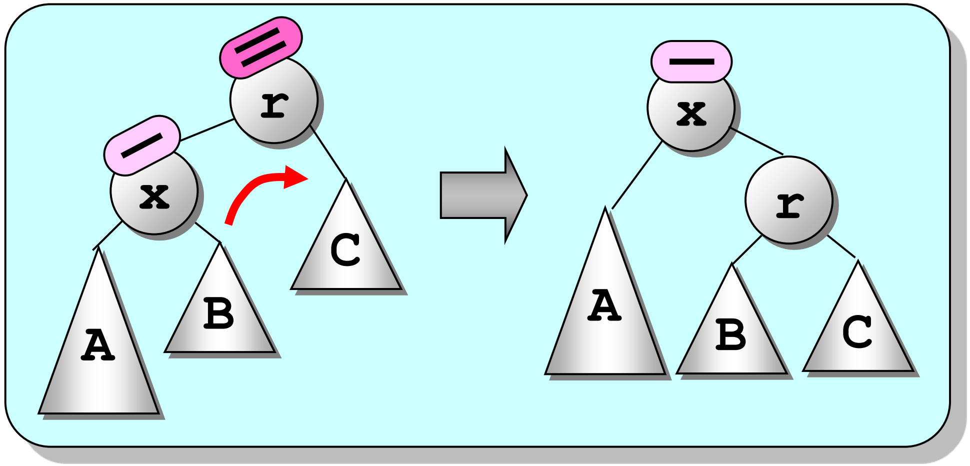
Insert normally and adjust

Erase normally and adjust

rebalance has 4 cases

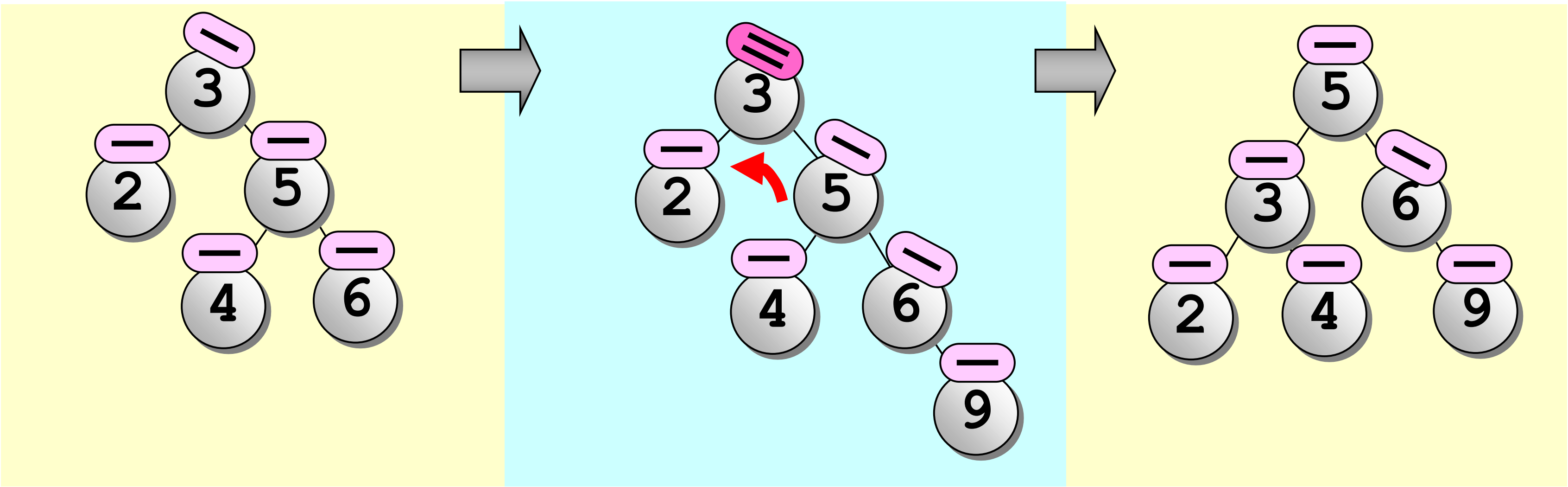
rebalance : case 1

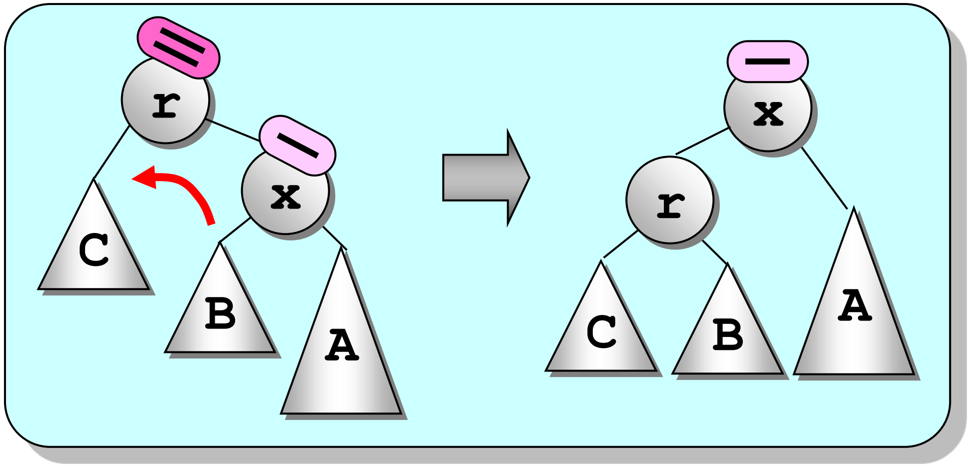




rotate_left_child(r)

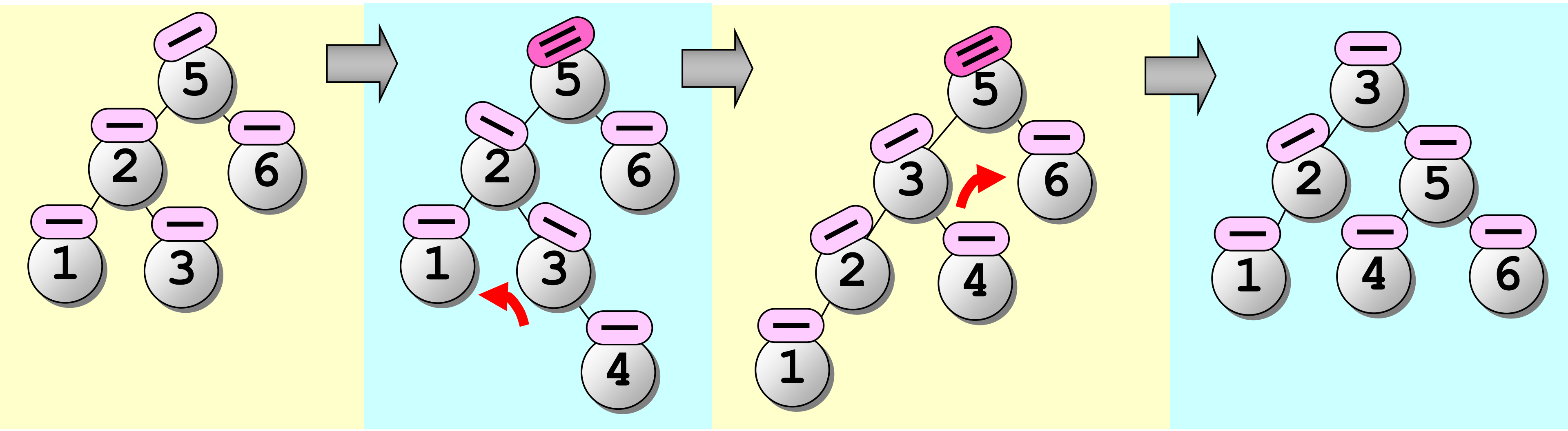
rebalance : case 2

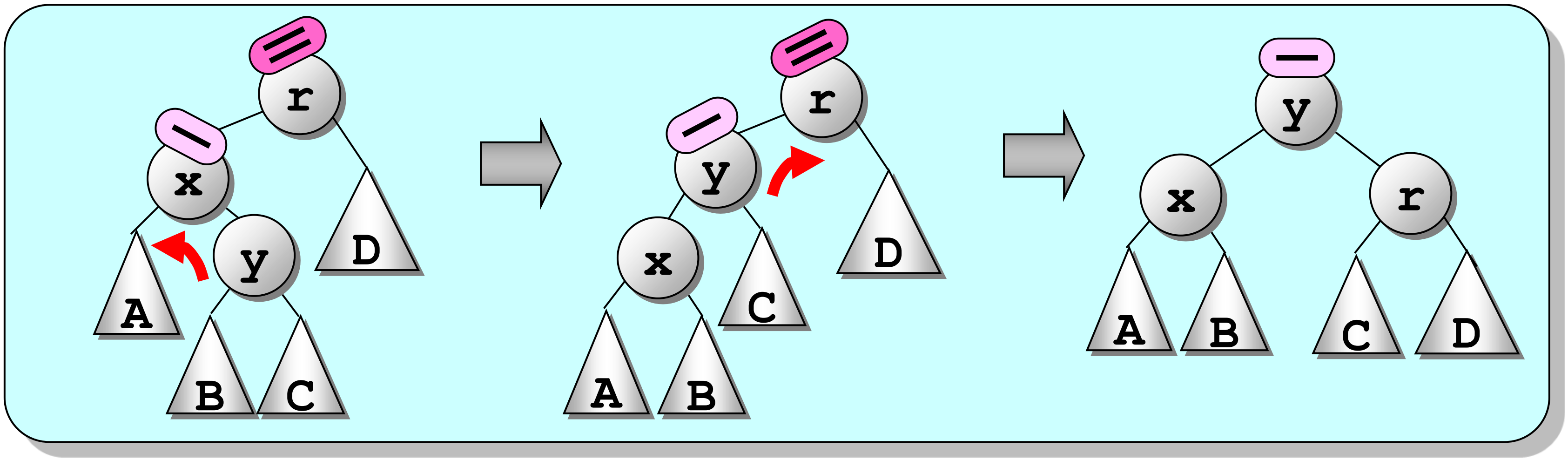




`rotate_right_child(r)`

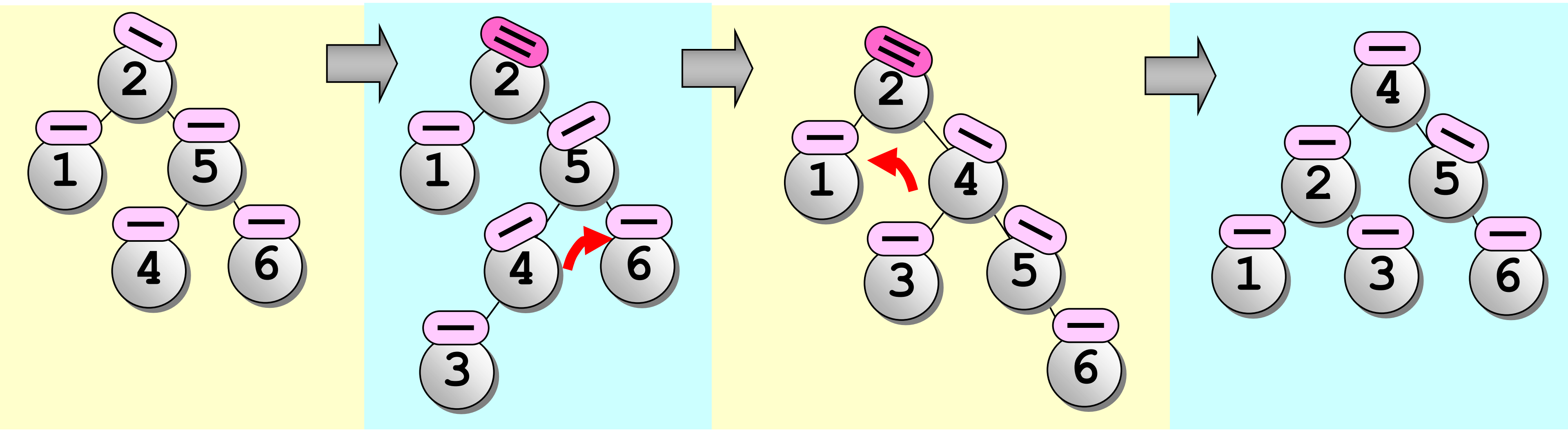
rebalance : case 3

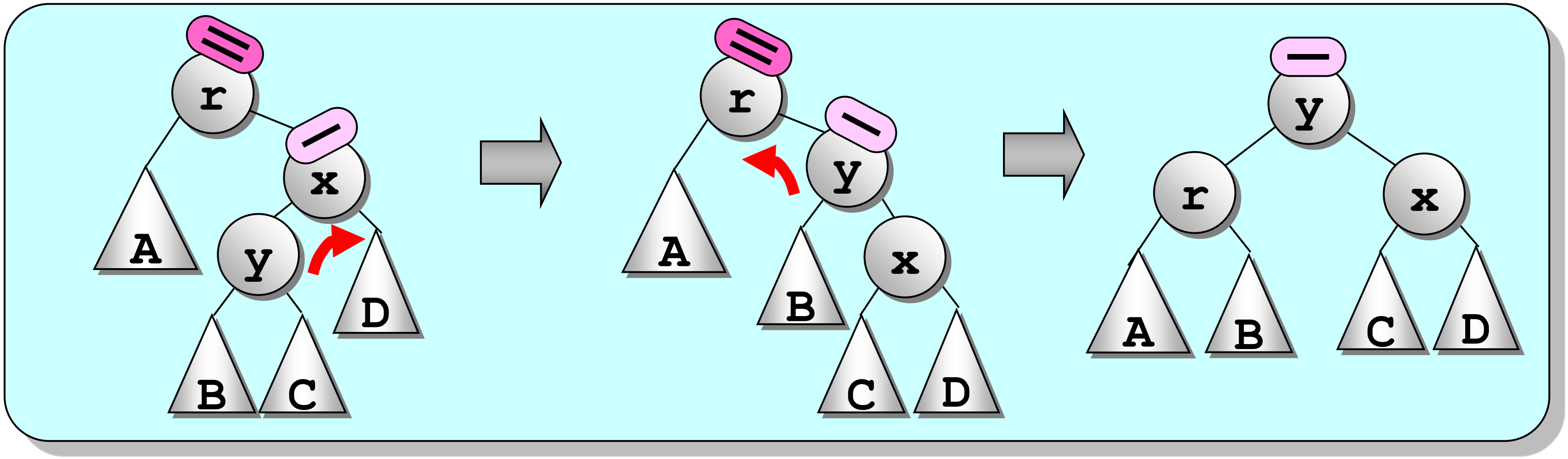




`rotate_right_child(r->left)` **`rotate_left_child(r)`**

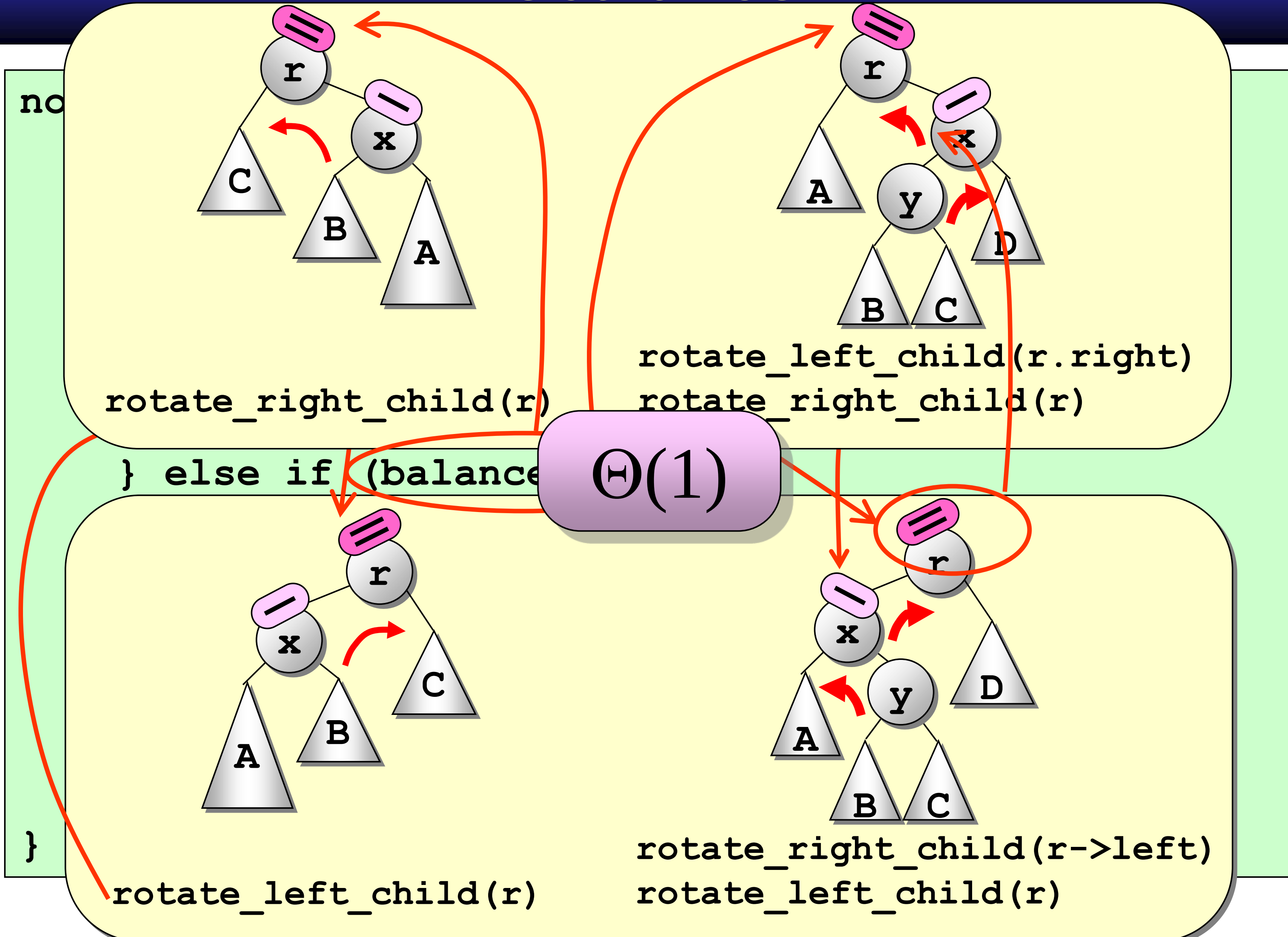
rebalance : case 4





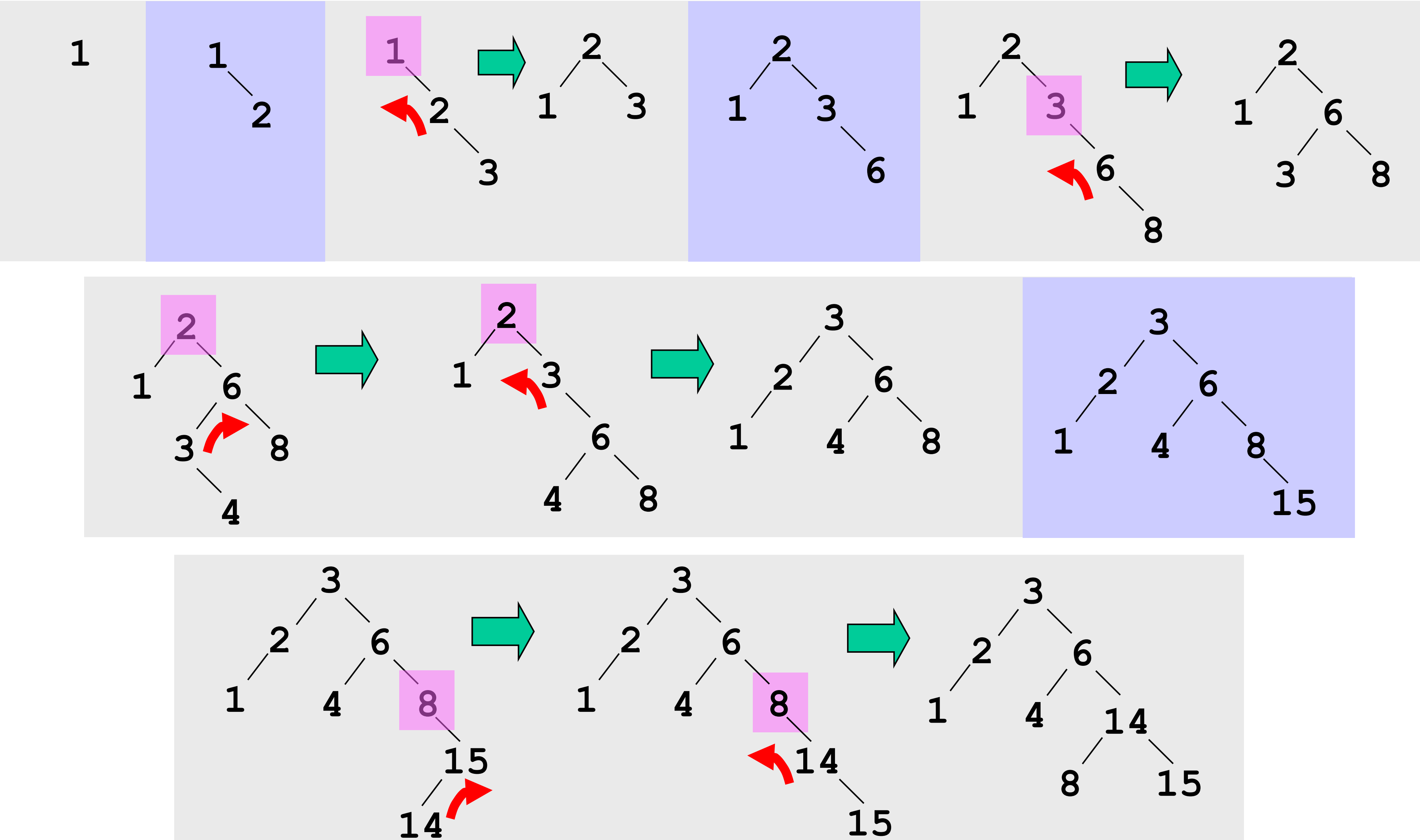
`rotate_left_child(r->right)   rotate_right_child(r)`

rebalance


no
} else if (balance
}
r
x
C
B
A
rotate_right_child(r)
r
x
y
A
B
C
D
rotate_left_child(r.right)
rotate_right_child(r)
Θ(1)
r
x
A
B
C
rotate_left_child(r)
r
x
y
A
B
C
D
rotate_right_child(r->left)
rotate_left_child(r)

Example

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



Summary

- AVL Tree is a binary search tree with height control
- Difference in height of the two children <= 1
- Can prove that $\lfloor \log_2 n \rfloor \leq h < 1.44 \log_2 n$
- Each node store height
- After insertion/deletion, if the height rule is violated, need to balance the tree
- Balancing is done by rotation
- Insert, delelete, search cost 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;
```