AVL Tree

Binary Tree with value condition

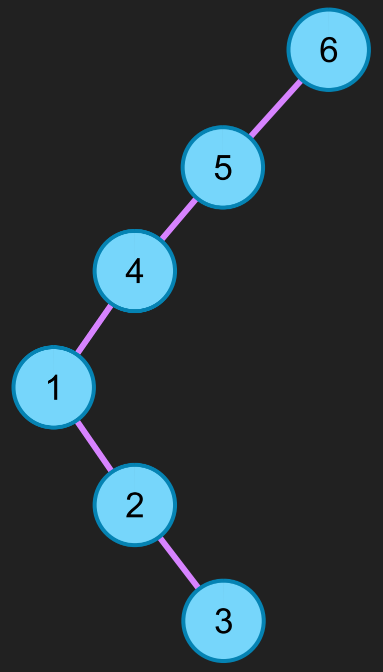
Nattee Niparnan

Overview

- Performance of a Binary Search Tree depends on its height (h)
  - A BST of n nodes has a wide range of possible height, lg(n) <= h <= n-1
  - The best case is very good (logarithmic) while the worst case is bad (linear)
- AVL Tree introduces additional constraint on the structure of the tree, called balance constraint
  - For any node, the difference between the height of the the node's subtree and right subtree cannot be larger than 1
  - To enforce this constraint, we use several kind of rotation operations on nodes.
  - These operations are applied when the tree is modified and applied on every nodes on the path from the modifying node to the root. Hence, this operation is additional O(h) to insert / erase
- We can show that this makes the height of the tree as O(log n)
- AVL Tree is named after Adelson-Velsky and Landis

How likely a bad tree may happen?

- A bad tree is, on uniform data assumption, is unlikely to happen
  - Even though there are several possible bad trees, but the number is very small
  - Consider a tree of n node (1 to n). The worst possible height is n-1. There is (n-1)! possible trees but only $2^{(n-1)}$ trees are of height (n-1)
- With the assumption of uniform insertion, it is proven that the expected height of a binary search tree is approximately 4.31107 log n
  - This means that, on average, the height of a BST is O(lg n)
- However, in practice, the bad sequence is more likely to happen because a bad sequence is when the data is sort (or partially sorted). This kind of data happens more frequently in practice



A Note on the Height of Binary Search Trees, Luc Devroye, 1986

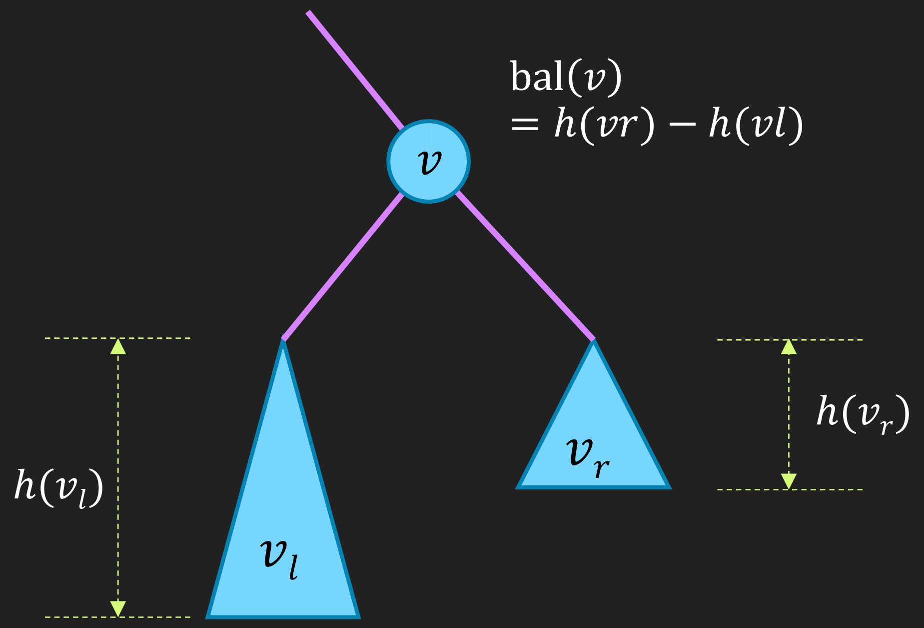
The Balance Constraint

- For each node v
  - We define v_l and v_r as the left and right subtree of v
  - We define $h(x)$ as the height of the tree x
    - Recall from that an empty tree has height as -1
  - We define a balance value of the node v as $\mathrm{bal}(v)$
    - $\mathrm{bal}(v) = h(v_r) - h(v_l)$
- the balance constraint is that $|\mathrm{bal}(v)| \leq 1$
- Every node in the AVL tree must satisfy the balance constraint

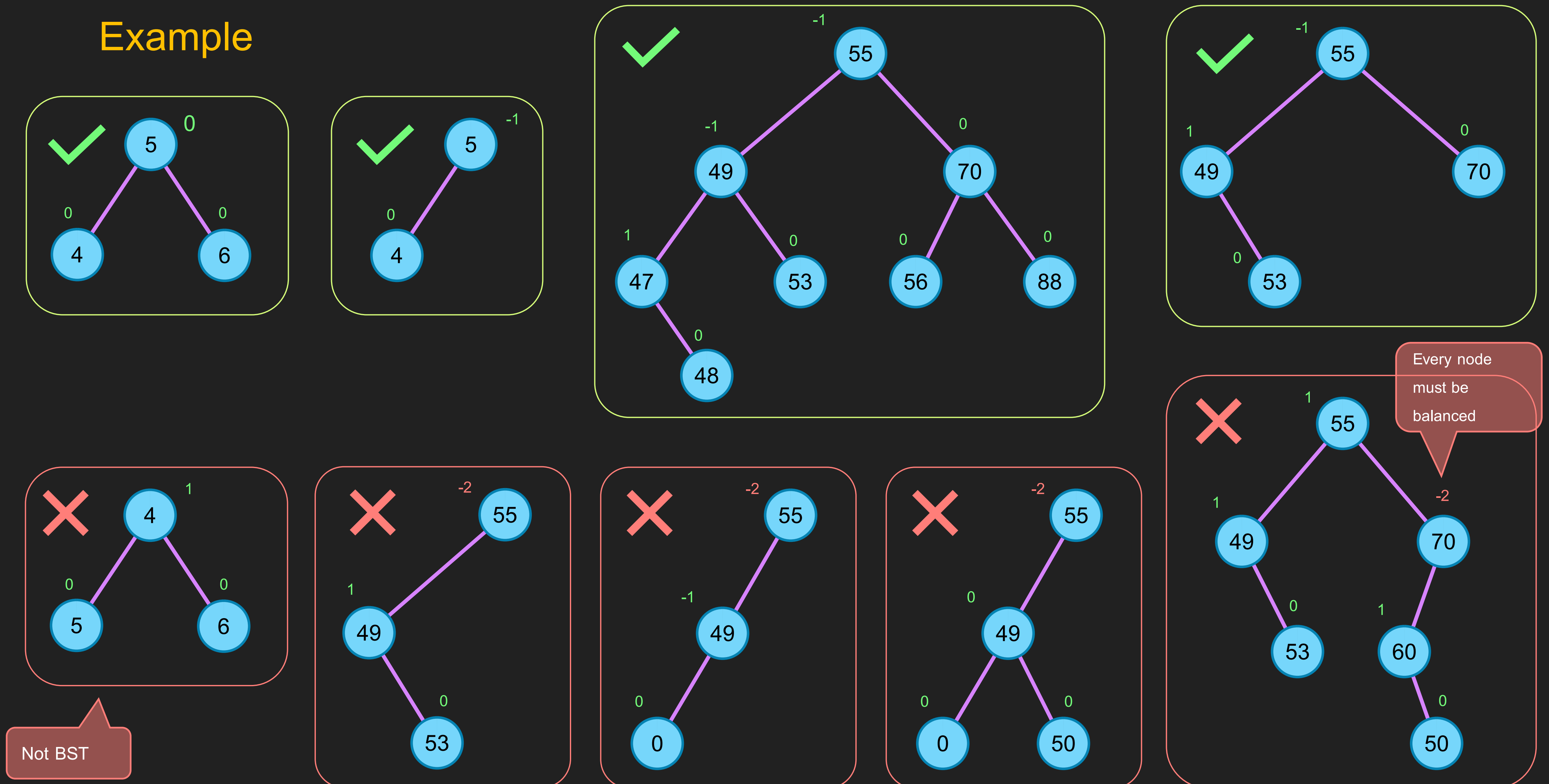
```
class node {
  public:
    int data;
    node *left, *right;
};

int get_height(node *n) {
  if (n == NULL) return -1;
  return 1 + std::max(get_height(n->left),
                      get_height(n->right));
}

int balance_value(node *n) {
  if (n == NULL) return 0;
  return get_height(n->right) - get_height(n->left);
}
```



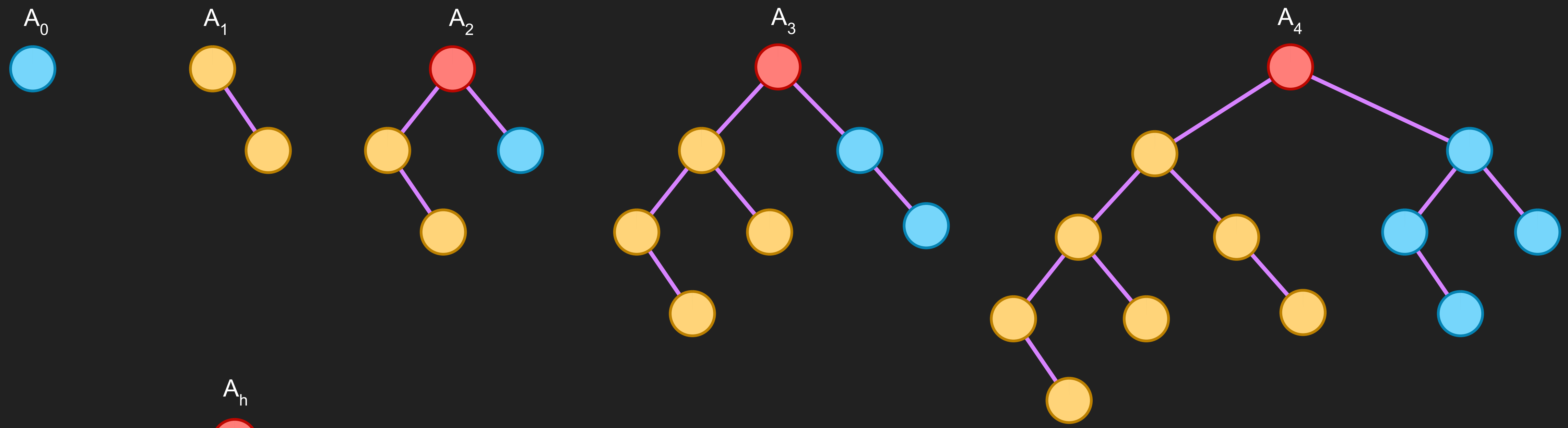
Example

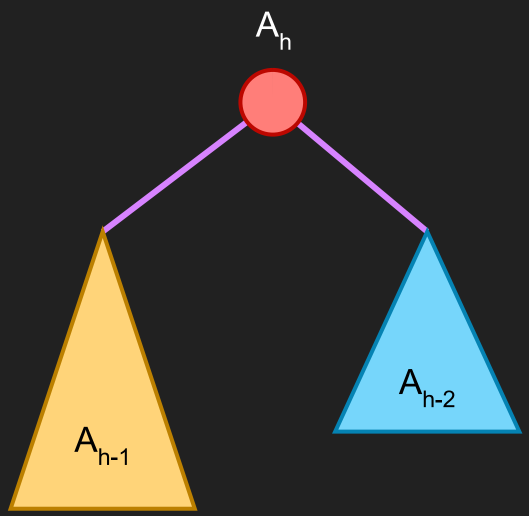


What is the maximum height of an AVL Tree

- Why maintaining the balance constraint makes the height of the tree as O(lg n)?
- For a non-zero integer h, define A_h as an AVL tree of height h that has minimum number of nodes
  - A_1, A_2, …, A_h are called Fibonacci tree
- Let $|A_h|$ be the number of nodes in the three A_h
  - To get to height h, we need at least $|A_h|$ nodes,
  - We will show the relation between h and $|A_h|$
    - It is that $|A_h|$ is $\Omega(2^h)$

Fibonacci Tree





- We will solve for $|A_h|$

$$|A_h| = \begin{cases} 1 & h = 0 \\ 2 & h = 1 \\ 1 + |A_{n-1}| + |A_{n-2}| & h \geq 2 \end{cases}$$

Solving for $|A_h|$

$$|A_h| = \begin{cases} 1 & h = 0 \\ 2 & h = 1 \\ 1 + |A_{h-1}| + |A_{h-2}| & h \geq 2 \end{cases}$$

$$\begin{aligned} |A_h| &= 1 + |A_{h-1}| + |A_{h-2}| \\ &= 1 + (1 + |A_{h-2}| + |A_{h-3}|) + |A_{h-2}| \\ &= 2 + 2|A_{h-2}| + |A_{h-3}| \\ &> 2|A_{h-2}| \end{aligned}$$

We get that $|A_h| > 2|A_{h-2}|$

$$\begin{aligned} |A_h| &> 2|A_{h-2}| \\ &> 4|A_{h-4}| \\ &> 8|A_{h-6}| \\ &> 16|A_{h-8}| \\ &> \cdots \\ &> 2^{h/2} \end{aligned}$$

This is what we want, $|A_h| = \Omega(2^h)$

Another Method with better bound

$$|A_h| = 1 + |A_{h-1}| + |A_{h-2}|$$

$$\begin{pmatrix} |A_h| \\ |A_{h-1}| \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} |A_{h-1}| \\ |A_{h-2}| \\ 1 \end{pmatrix}$$

Re-write $|A_h|$ as a matrix equation

$$\begin{pmatrix} |A_h| \\ |A_{h-1}| \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}^{h-1} \begin{pmatrix} |A_1| \\ |A_0| \\ 1 \end{pmatrix}$$

$$|A_h| = a\phi^h + b\hat{\phi}^h + 1$$

Solve using eigen decomposition
$\phi \approx 1.618$
$\hat{\phi} \approx -0.618$
The term $b\hat{\phi}^h$ vanish

$$|A_h| \approx a\phi^h$$

$$\log(|A_h|) \approx \log(a) + \log(\phi^h)$$

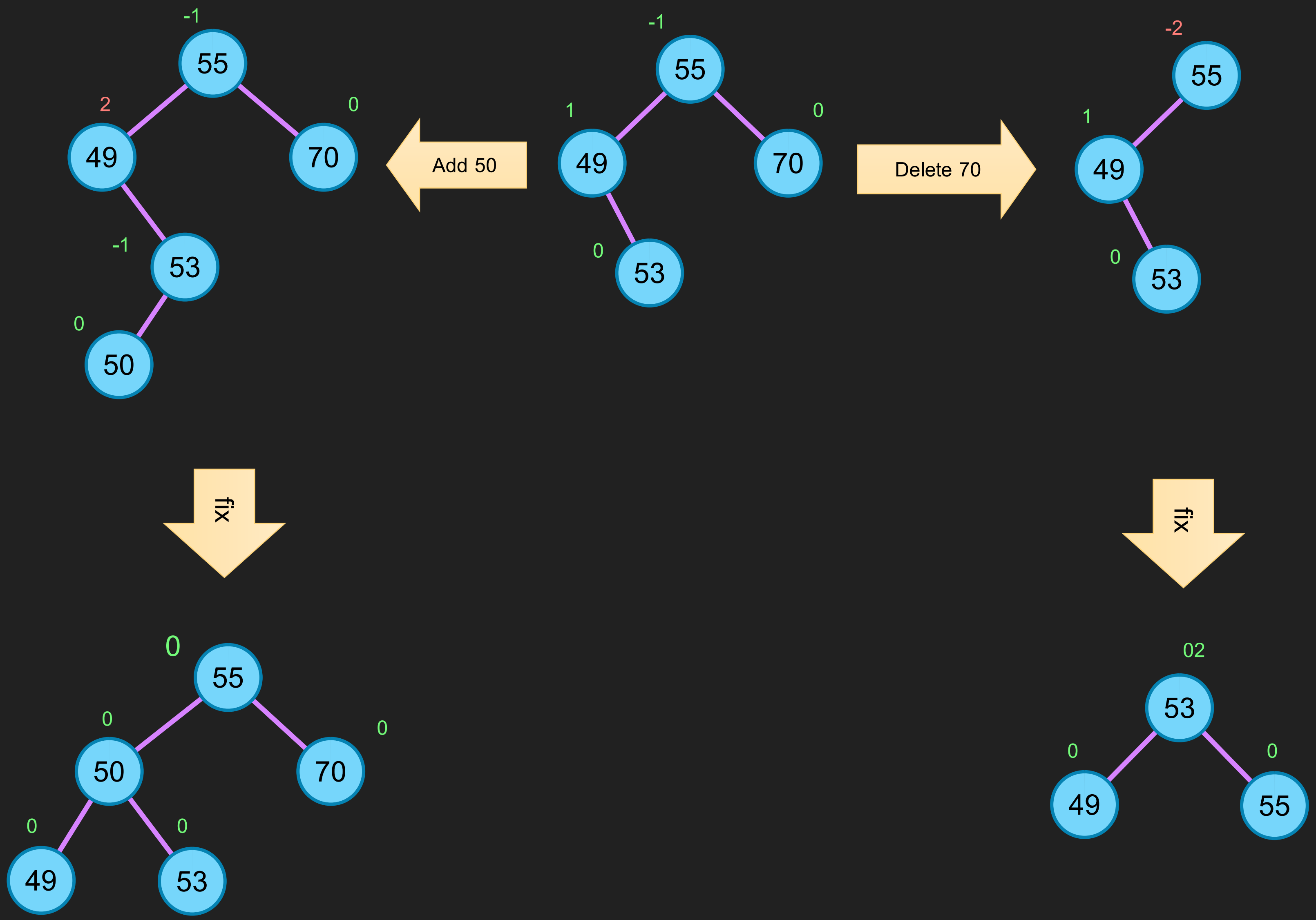
$$\approx \log(a) + h \cdot \log(\phi)$$

$\log_\phi x$ is approximately $1.44 * \log_2 x$

$$\log_\phi(|A_h|) - \log_\phi(a) \approx h$$

This is what we need, the height is approximately 1.44 $\log_2$ n

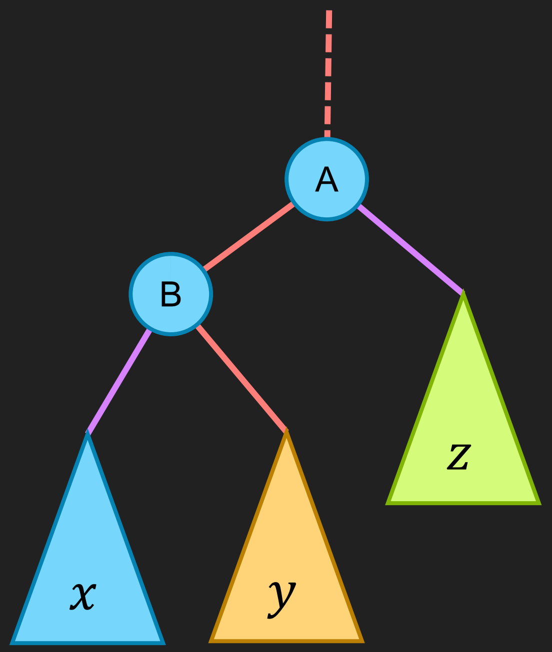
How we maintain the balance value



- See that height only changes when we add or remove nodes
  - Balance values only change along the path from the modified nodes to the root
  - After modification, we calculate new balance value along the path and fix any error along the way using rotation operation

Rotation Operation

- There are two basic rotation: Rotate Right and Rotate Left
- It operates on a root of a subtree
- Does not BST violate value constraint

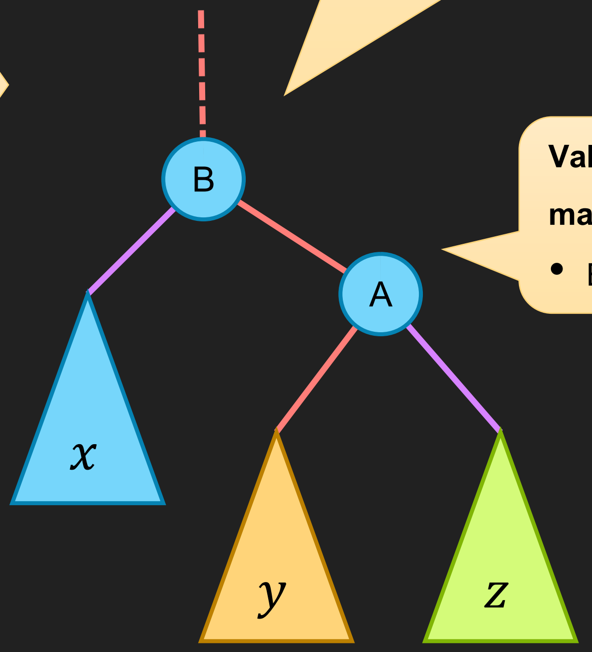


Rotate Right at A

- Move B up as the new root of the subtree
- Move A to be the right child of B
- Move y to be the left child of A
- Subtree x and z stay the same

Before rotation:

- A is the root of a subtree
- B, the left child of A, must exist
- x, y and z are some (maybe empty) subtrees



After rotation:

- B is a new root
- Height of y stays the same
- Height of x decreases
- Height of z increases

Value Relation is maintained:

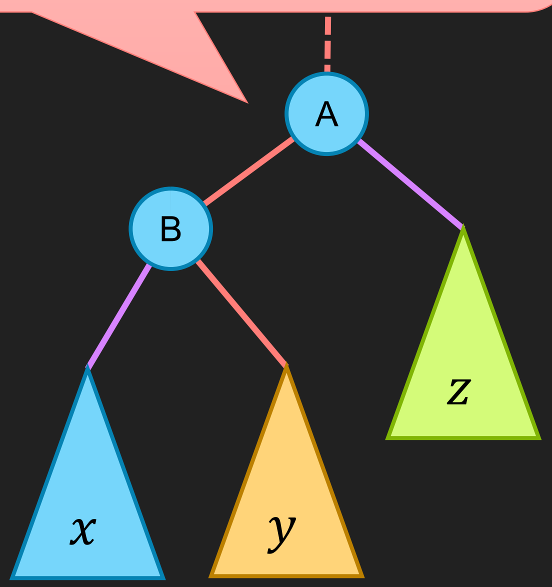
- B < y < A

Mirror between left and right

- Rotate Left is just the reverse of the rotate right

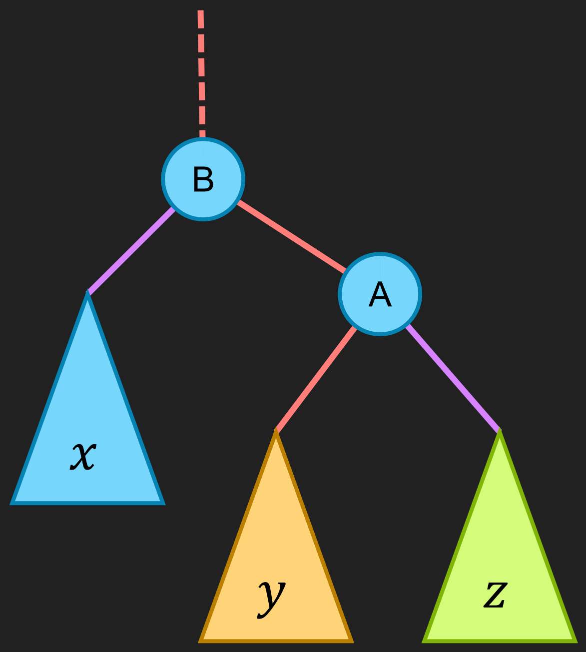
After rotation:

- A is the new root
- Height of y stays the same
- Height of x increases
- Height of z decreases



Rotate Left at B

- Move A up as the new root of the subtree
- Move B to be the left child of A
- Move y to be the right child of B
- Subtree x and z stay the same



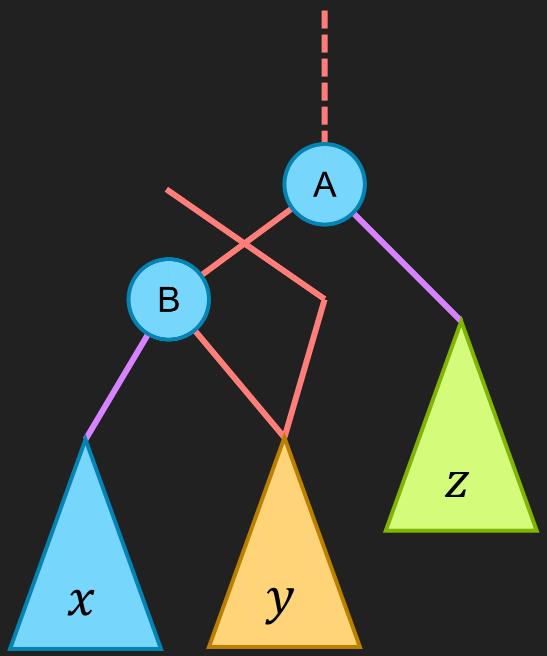
Before rotation:

- B is the root of a subtree
- A, the right child of B, must exist
- x, y and z are some (maybe empty) subtrees

Rotation Code

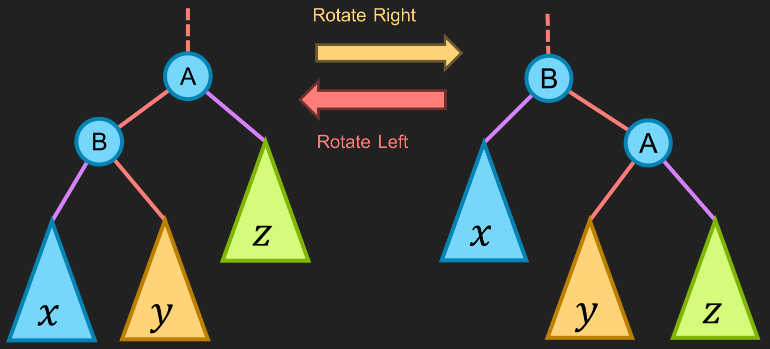
```
node* rotate_left_child(node * r) {
  node *new_root = r->left;
  r->set_left(new_root->right);
  new_root->set_right(r);
  new_root->right->set_height();
  new_root->set_height();
  return new_root;
}
```

- Take a root of a subtree
- return the new root of the subtree

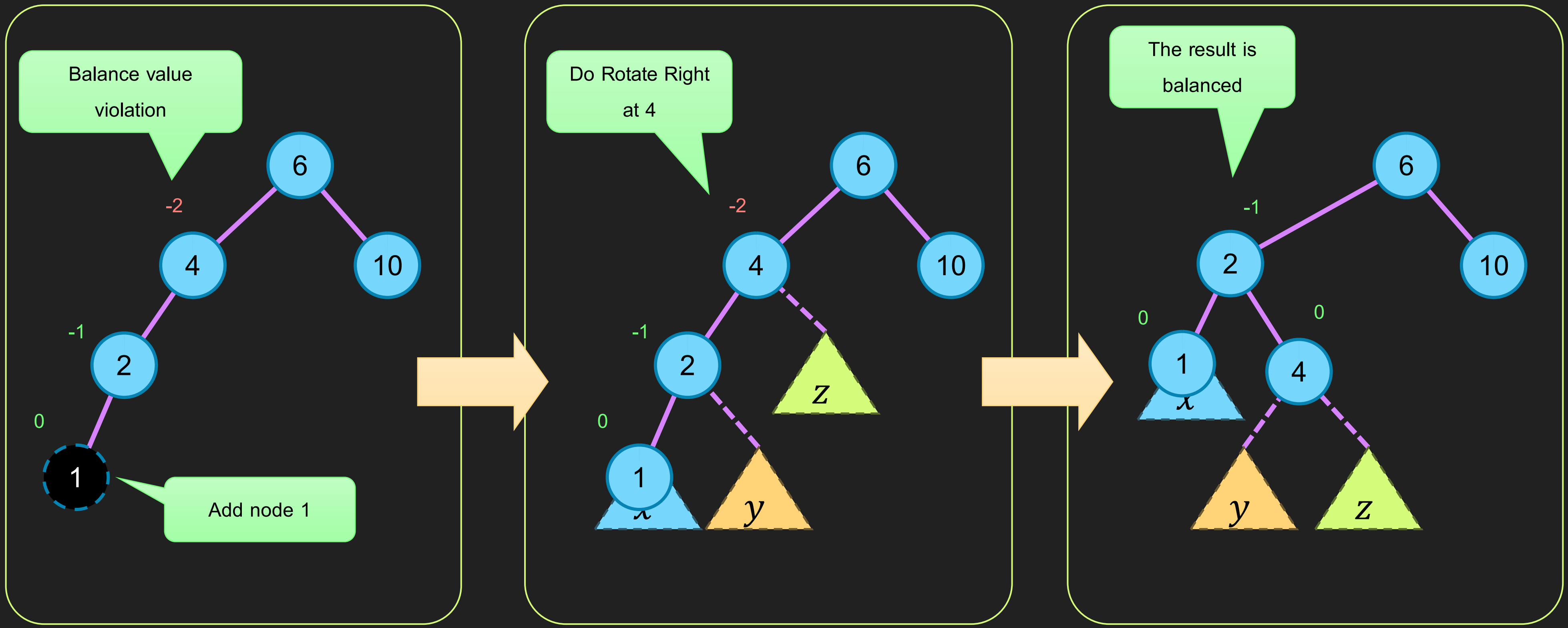



```
node* rotate_right_child(node * r) {
  node *new_root = r->right;
  r->set_right(new_root->left);
  new_root->set_left(r);
  new_root->left->set_height();
  new_root->set_height();
  return new_root;
}
```

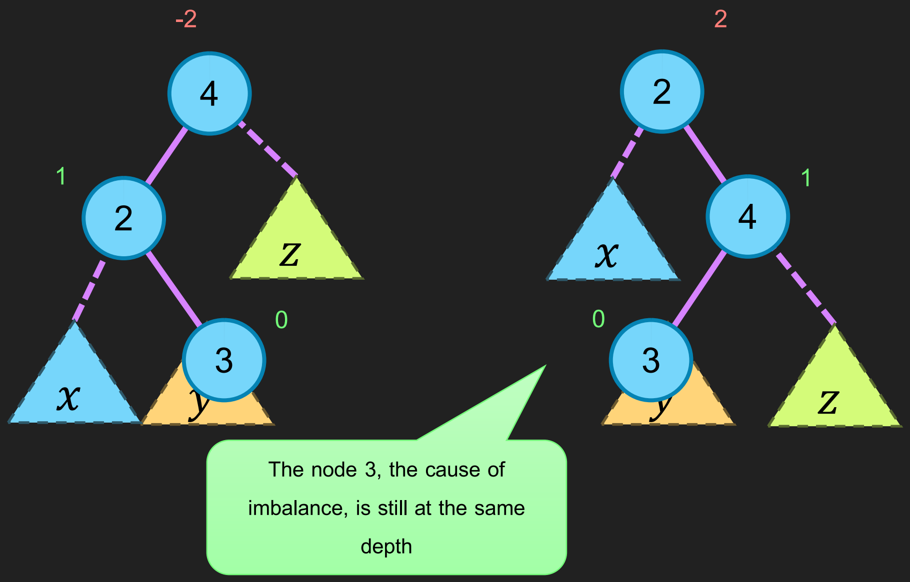
When to use Rotation



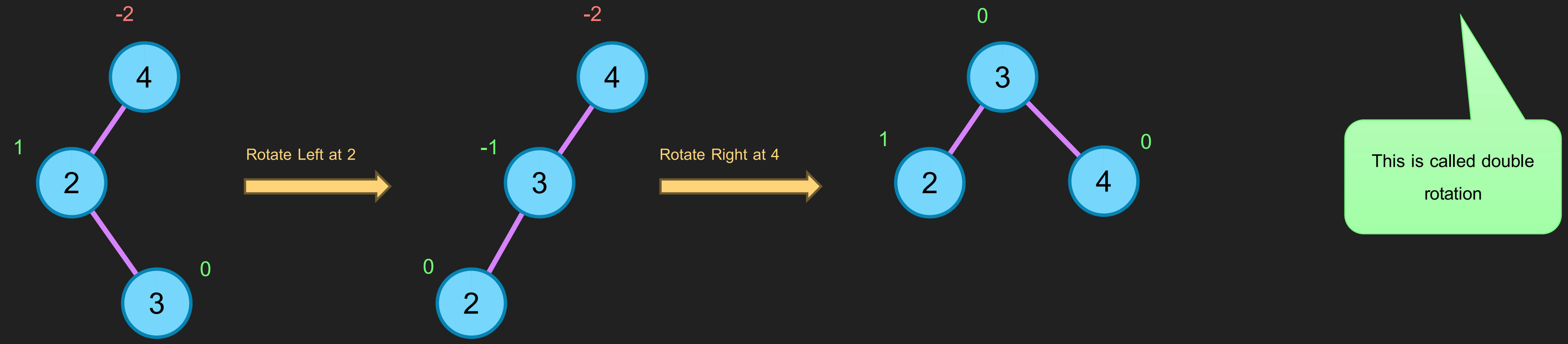
- Rotate Right reduces the height of left subtree of the left child
    - We use it when the node is left heavy, i.e., balance value is -2, and this left heavy is caused by having too deep left subtree of the left child while having too shallow right subtree of the root



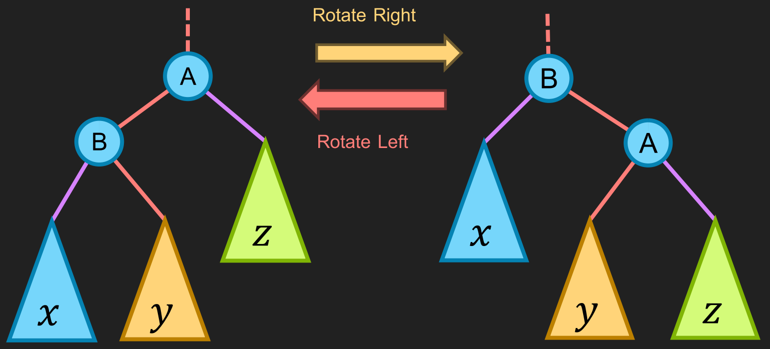
When one Rotation Right does not solve the problem



- Rotate Right move the left subtree of the left child up
- This does not help when the cause of left heavy is NOT the left subtree of the left child but rather the right subtree
  - When the balance value of the left child is not of the same sign as the balance value of the root
- Fix by first make the left subtree heavy, using rotate left at the left subtree and then do the rotate right at the root

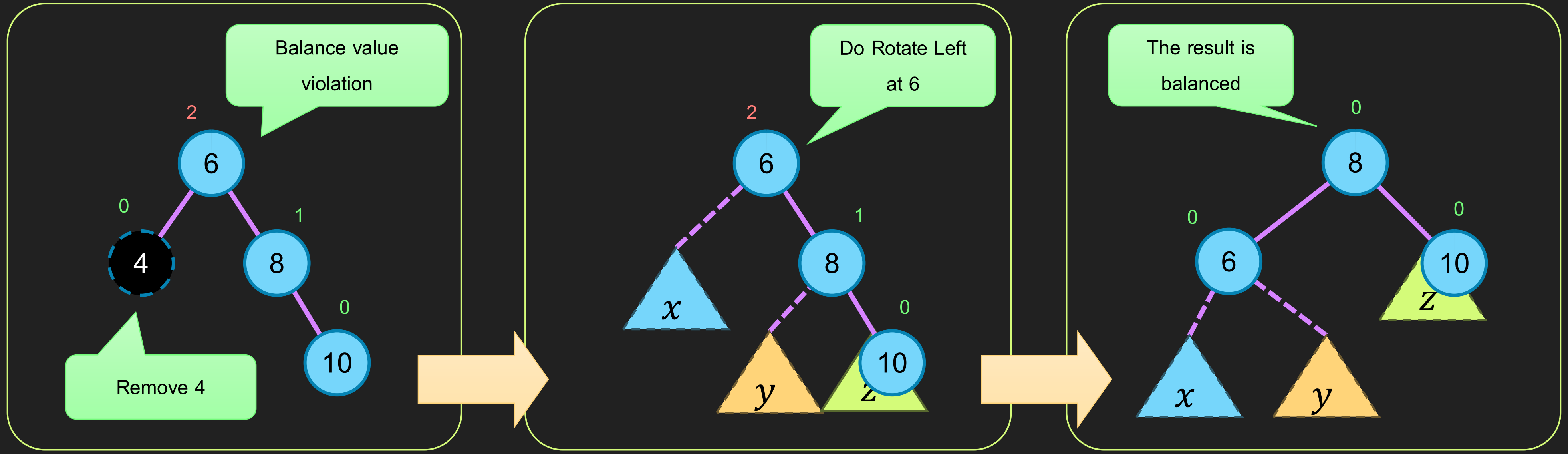


What About Rotate Left?



- Rotate Left is The mirror case of the Rotate Right

- Use when it is right heavy

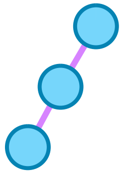
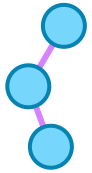
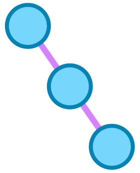
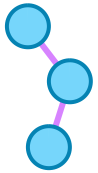


- Double Rotation when right heavy (balance =2) with the right child has a balance value of -1

Rotation Summary

- Rebalance according to the balance value of the node

```
node * rebalance(node * r) {
  if (r == NULL) return r;
  int balance = r->balance_value();
  if (balance == -2) {
    if (r->left->balance_value() == 1)
      r->set_left(rotate_right_child(r->left));
    r = rotate_left_child(r);
  } else if (balance == 2) {
    if (r->right->balance_value() == -1)
      r->set_right(rotate_left_child(r->right));
    r = rotate_right_child(r);
  }
  r->set_height();
  return r;
}
```

Balance Value	Case	Example	Fix
-2 (Left Heavy)	Balance Value of Left Child is -1		Rotate Right at the node
	Balance Value of Left Child is +1		Rotate Left at the left child then Rotate Right at the node
+2 (Right Heavy)	Balance Value of Right Child is +1		Rotate Left at the node
	Balance Value of Right Child is -1		Rotate Right at the right child then Rotate Left at the node

Integrate with Insert and Erase

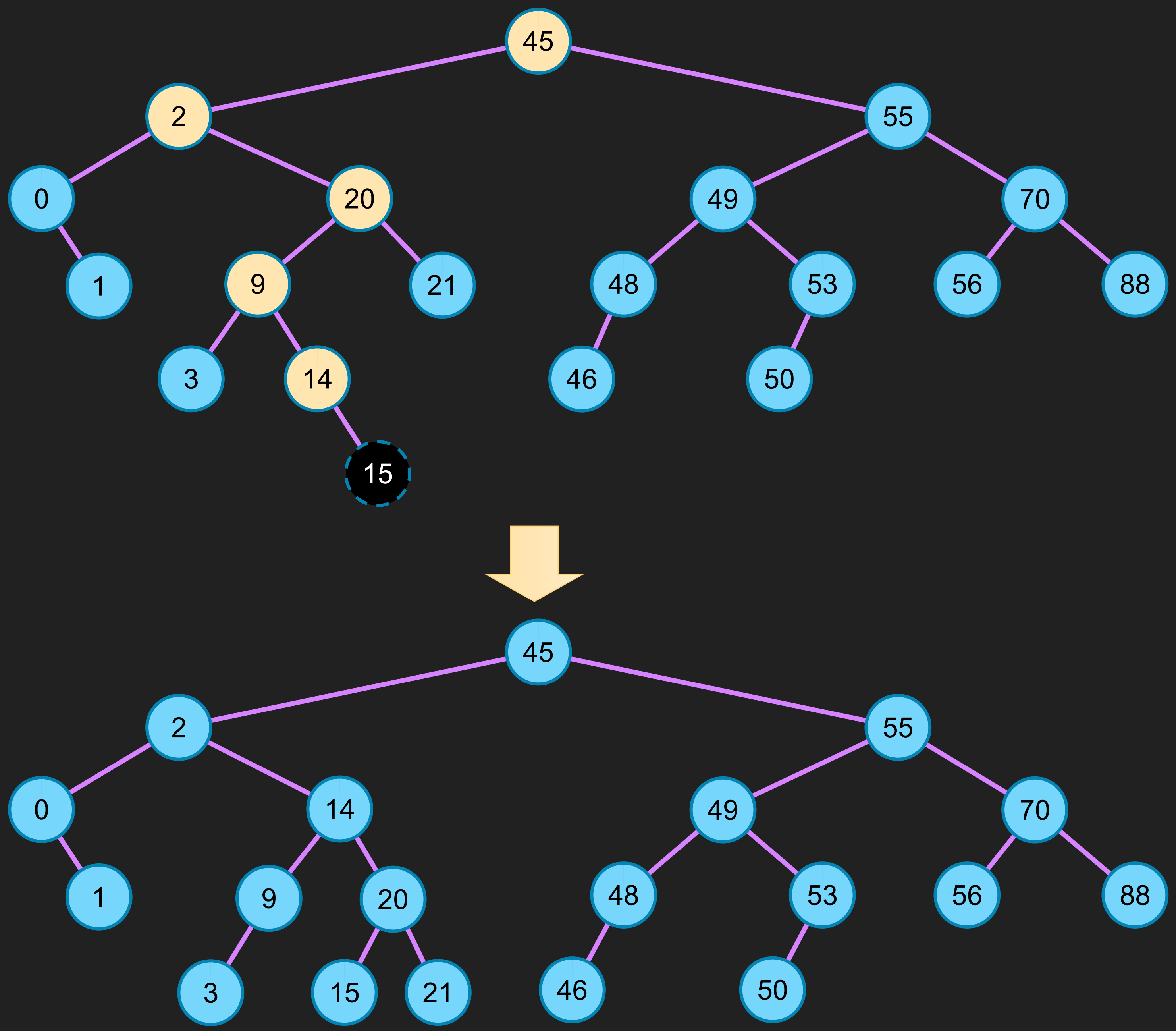
- Insert by recursion
- After modification, call rebalance
  - This re-calculate balance value and do the rotation if necessary

```
node* insert(const ValueT& val, node *r, node * &ptr) {
  if (r == NULL) {
    mSize++;
    ptr = r = new node(val,NULL,NULL,NULL);
  } else {
    int cmp = compare(val.first, r->data.first);
    if (cmp == 0) ptr = r;
    else if (cmp <  0) {
       r->set_left(insert(val, r->left, ptr));
    } else {
       r->set_right(insert(val, r->right, ptr));
    }
  }
  r = rebalance(r);
  return r;
}
```

```
node * rebalance(node * r) {
  if (r == NULL) return r;
  int balance = r->balance_value();
  if (balance == -2) {
    if (r->left->balance_value() == 1)
      r->set_left(rotate_right_child(r->left));
    r = rotate_left_child(r);
  } else if (balance == 2) {
    if (r->right->balance_value() == -1)
      r->set_right(rotate_left_child(r->right));
    r = rotate_right_child(r);
  }
  r->set_height();
  return r;
}
```

Example

1. add 15
2. Check balance at 15: 0 (OK)
3. Check balance at 14: 1 (OK)
4. Check balance at 9: 1 (OK)
5. Check balance at 20: -2 (Not OK)
    1. Rotate left at 9
    2. Rotate right at 20
6. Check balance at 2: (OK)
7. Check Balance at 45: (OK)



Erase

```
node *erase(const KeyT &key, node *r) {
  if (r == NULL) return NULL;
  int cmp = compare(key, r->data.first);
  if (cmp < 0) {
    r->set_left(erase(key, r->left));
  } else if (cmp > 0) {
    r->set_right(erase(key, r->right));
  } else {
    if (r->left == NULL || r->right == NULL) {
       node *n = r;
       r = (r->left == NULL ? r->right : r->left);
       delete n;
       mSize--;
    } else {
       node * m = r->right;
       while (m->left != NULL) m = m->left;
       std::swap(r->data.first, m->data.first);
       std::swap(r->data.second, m->data.second);
       r->set_right(erase(m->data.first, r->right));
    }
  }
  r = rebalance(r);
  return r;
}
```

- Also use recursion and call rebalance after a node is erased

The Node Class

- Additional Height Value stored in each node
  - Update after insert / erase only along the path
  - set_left and set_right link child node
  - set_height recalculates height from its children's height, it is used in rebalance

```
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();
    }
    //...
};
```

```
class node {

  //...
  int get_height(node *n) {
    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);
  }
  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;
  }
};
```

Example Sequence of Operation