# CSE332: Data Abstractions

## Lecture 6: Dictionary, BST, AVL Tree

James Fogarty

Winter 2012

# *Reminders and Questions*

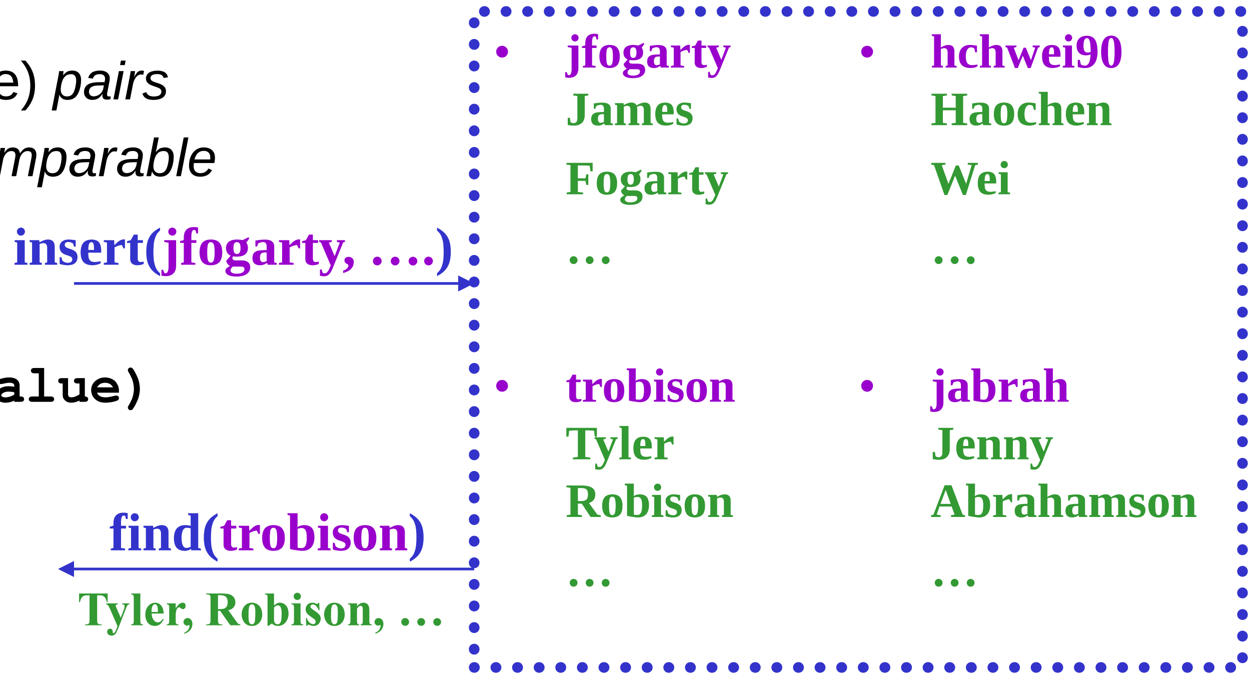
- Homework 2 Due Now

- Homework 3 Posted
  - Due Friday

- Project 2 Posted
  - Group Emails Due Wednesday
  - Milestone Due Next Wednesday

# *The Dictionary (a.k.a. Map) ADT*

- Data:
  - *Set* of (key, value) *pairs*
  - keys must be *comparable*
- Operations:
  - **`insert(key,value)`**
  - **`find(key)`**
  - **`delete(key)`**
  - …




*Probably the single most common ADT in everyday programs*

*We will tend to emphasize the keys, don't forget about the stored values*

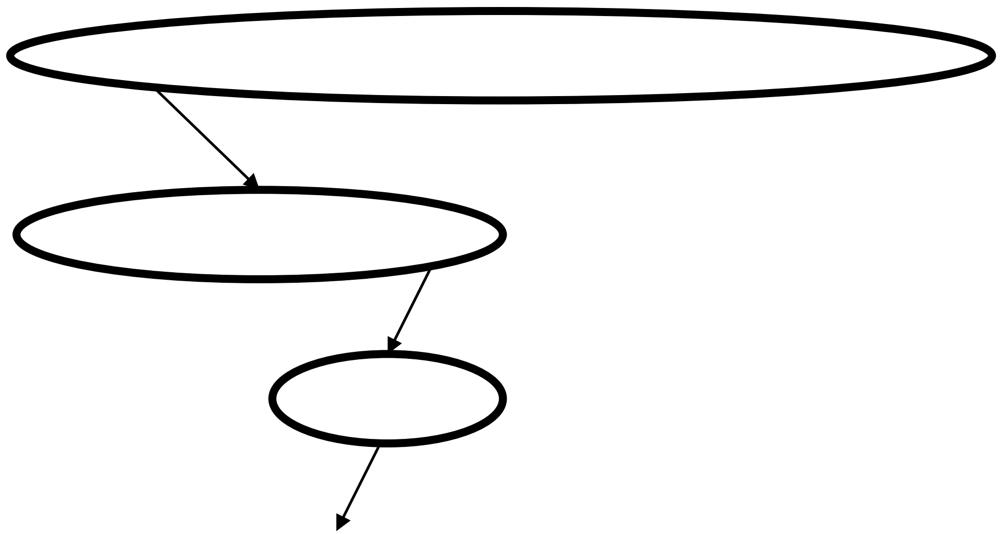
# *Simple Implementations*

For dictionary with *n* key/value pairs

| | `insert` | `find` | `delete` |
|---|---|---|---|
| • Unsorted linked-list | $O(1)$ | $O(n)$ | $O(n)$ |
| • Unsorted array | $O(1)$ | $O(n)$ | $O(n)$ |
| • Sorted linked list | $O(n)$ | $O(n)$ | $O(n)$ |
| • Sorted array | $O(n)$ | $O(\log n)$ | $O(n)$ |
| | $\log n + n$ | | $\log n + n$ |

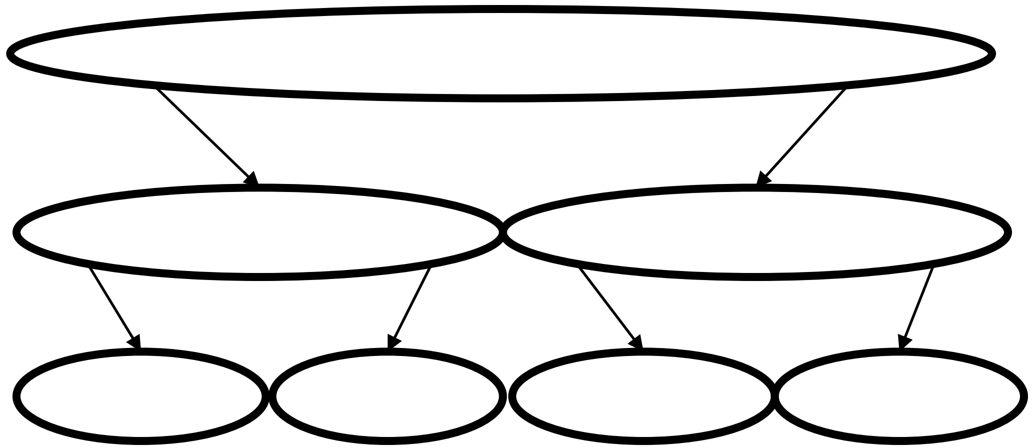
# *Binary Search*

**Target 4**



| | | | | | | | |
|---|---|---|---|---|---|---|---|
| **1** | **3** | **4** | **5** | **7** | **8** | **9** | **10** |

# *Binary Search Tree*

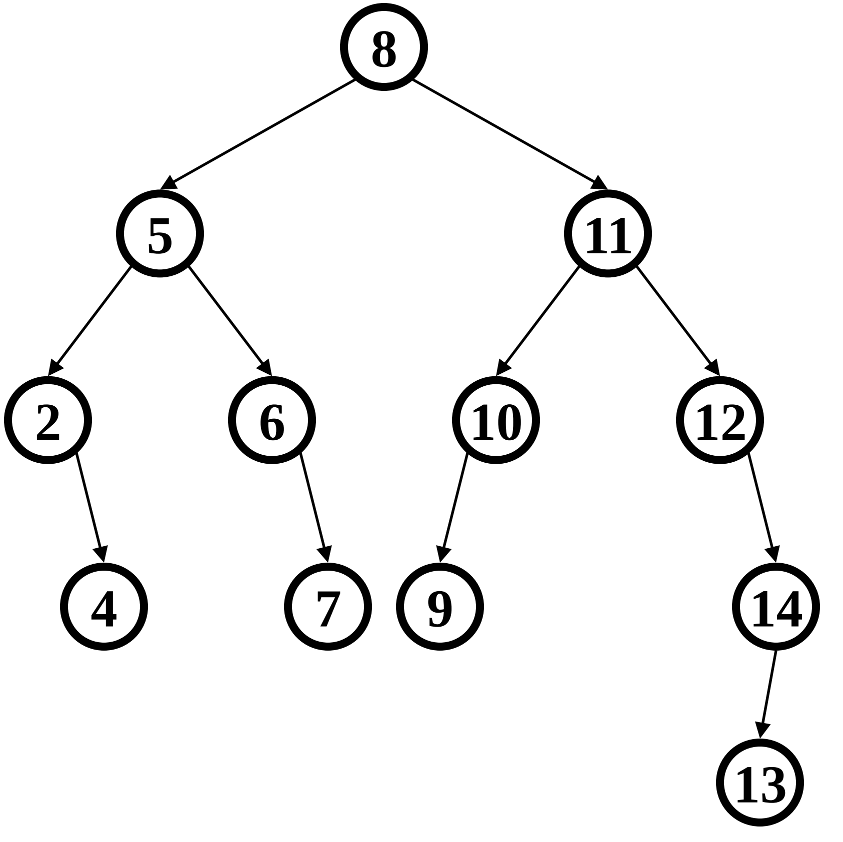


| | | | | | | | |
|---|---|---|---|---|---|---|---|
| **1** | **3** | **4** | **5** | **7** | **8** | **9** | **10** |

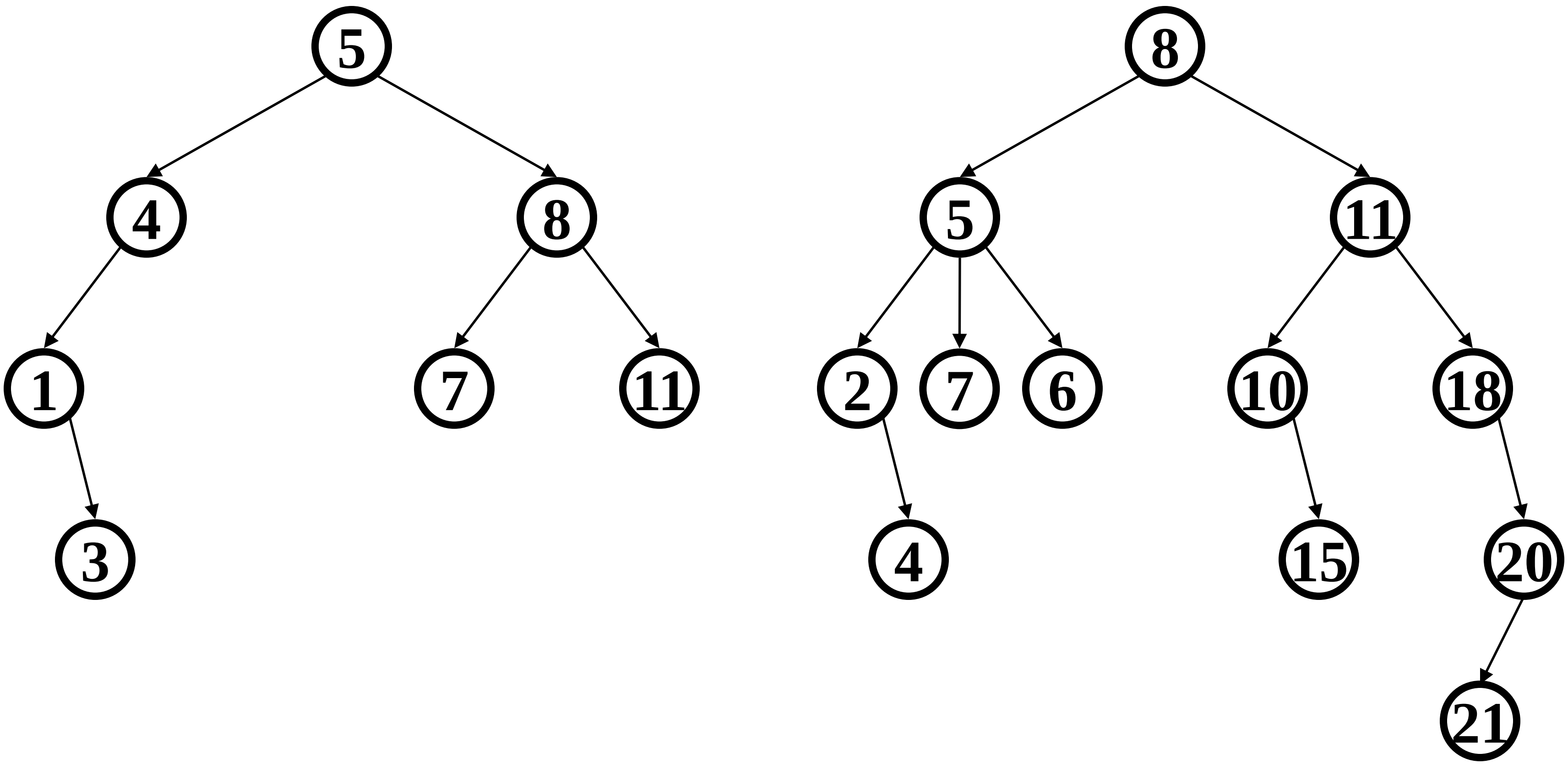
Our goal is the performance of binary search in a tree representation

# *Binary Search Tree*

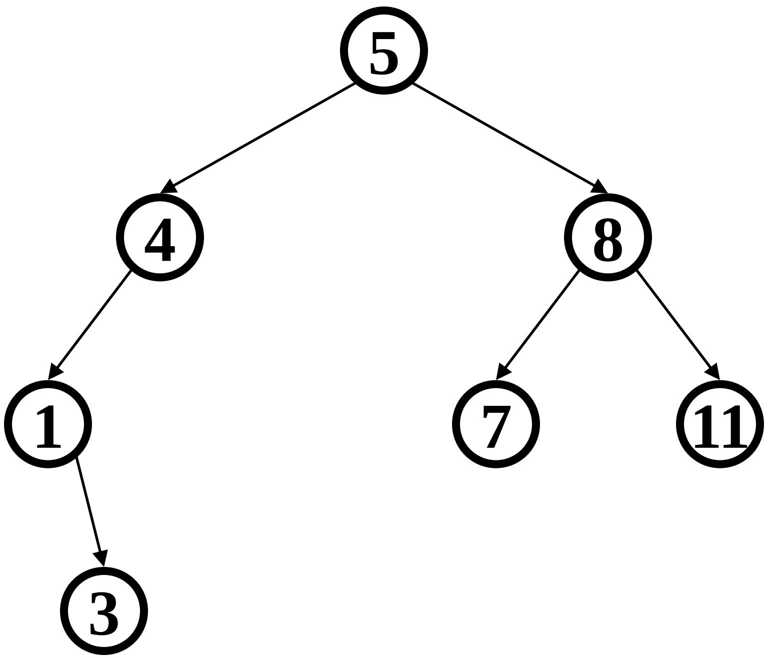
- Structure Property ("binary")
  - each node has ≤ 2 children
- Order Property
  - all keys in left subtree are smaller than node's key
  - all keys in right subtree are larger than node's key

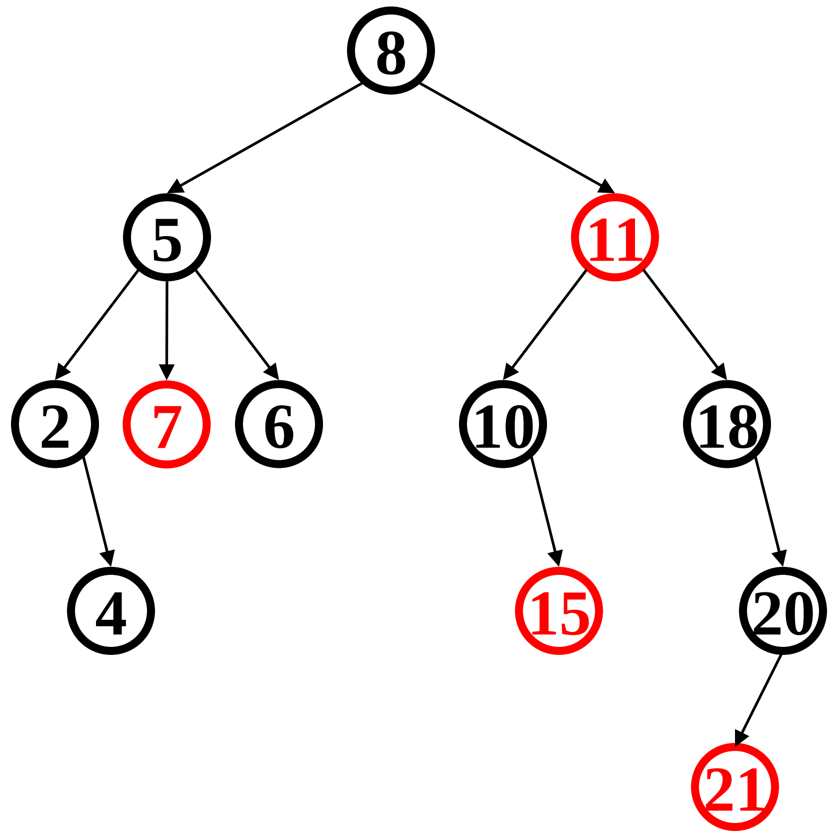


# *Are these BSTs?*

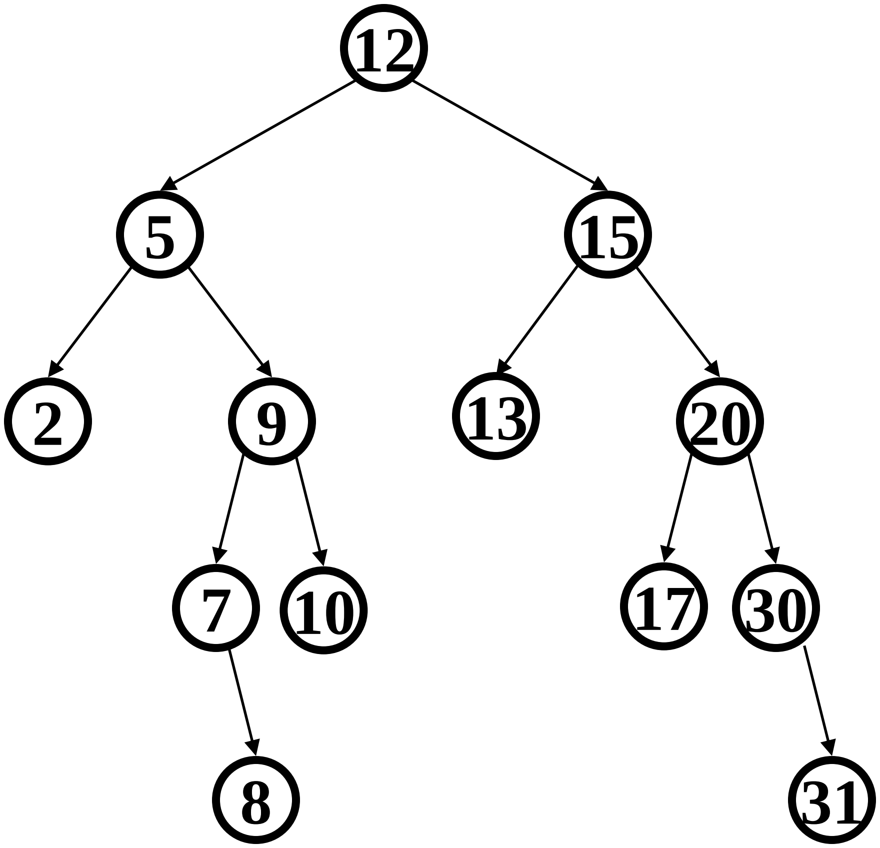


# *Are these BSTs?*



# *Insert and Find in BST*



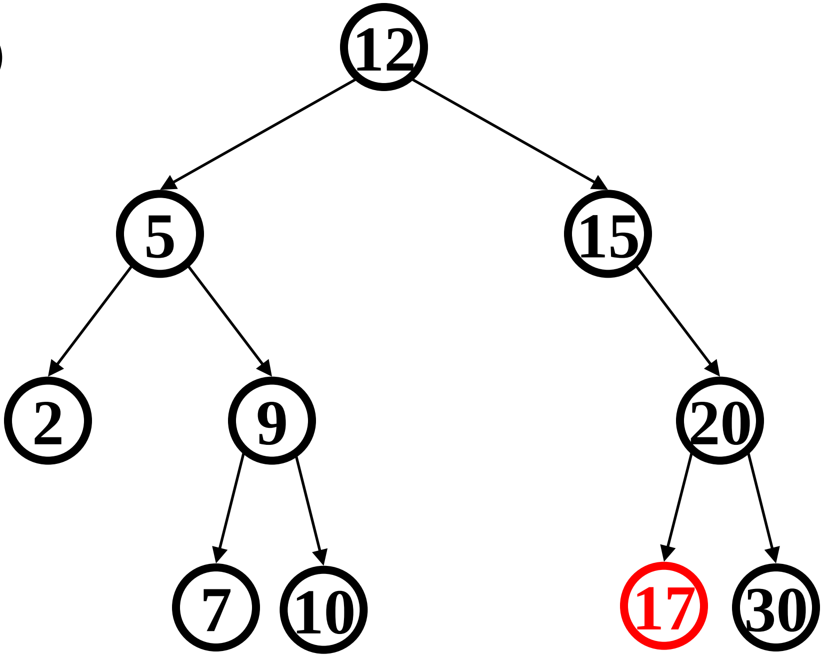

```
insert(13)
insert(8)
insert(31)
find(17)
find(11)
```

Insertion happens at leaves

Find walks down tree

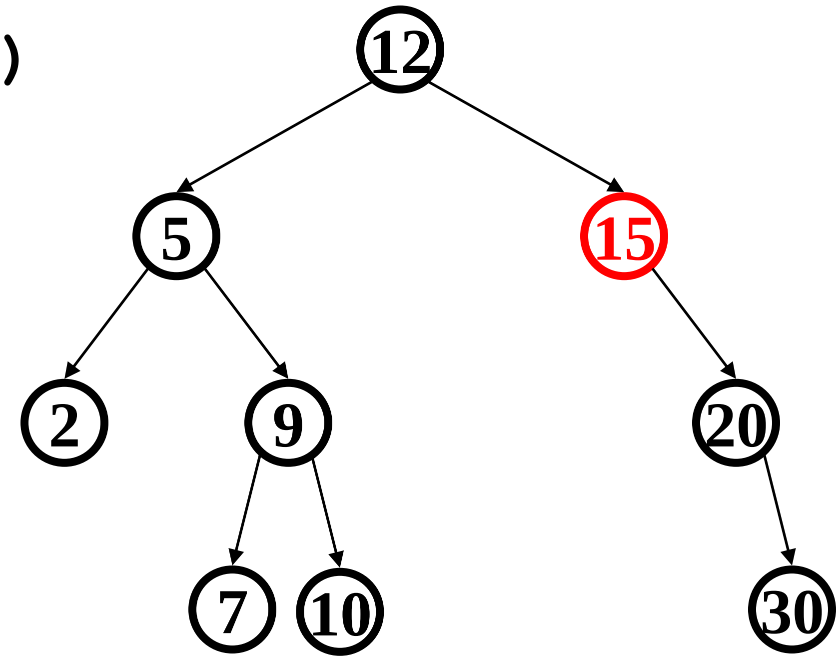
# Deletion – The Leaf Case

`delete(17)`



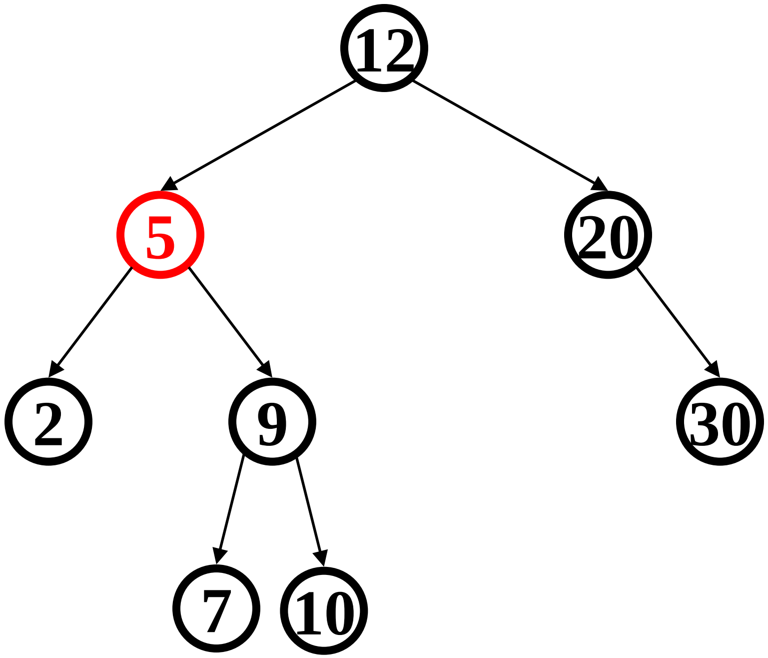
# *Deletion – The One Child Case*

**`delete(15)`**



# *Deletion – The Two Child Case*

**`delete(5)`**




What can we use to replace the 5?

– *successor* from right subtree: **`findMin(node.right)`**
– *predecessor* from left subtree: **`findMax(node.left)`**

# *The Need for a Balanced BST*

*Observation*

- BST is overall great
  - The shallower, the better!
- But worst case height is $O(n)$
  - Caused by simple cases, such as pre-sorted data

*Solution*

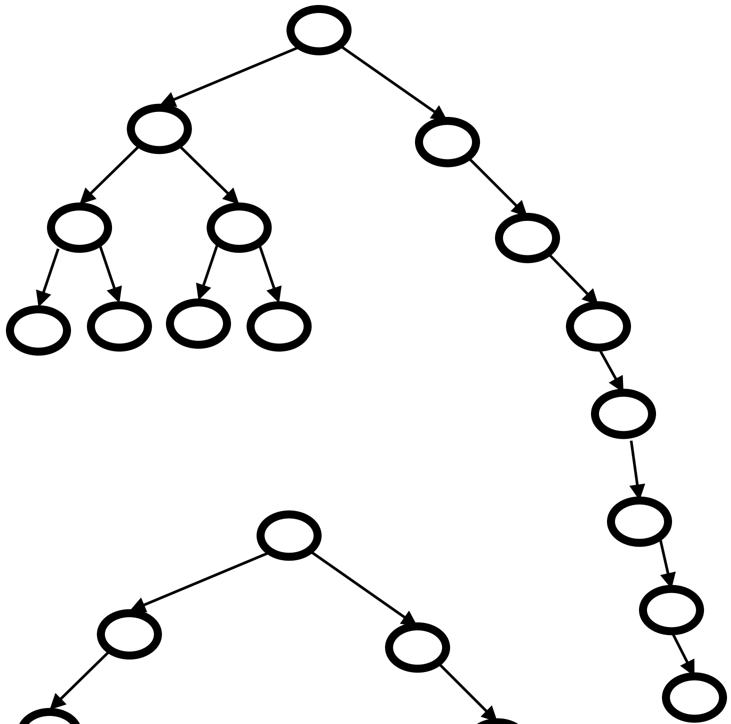
Require a **Balance Condition** that will:

1. ensure depth is always $O(\texttt{log}\ n)$ – strong enough!
2. be easy to maintain – not too strong!

# *Potential Balance Conditions*

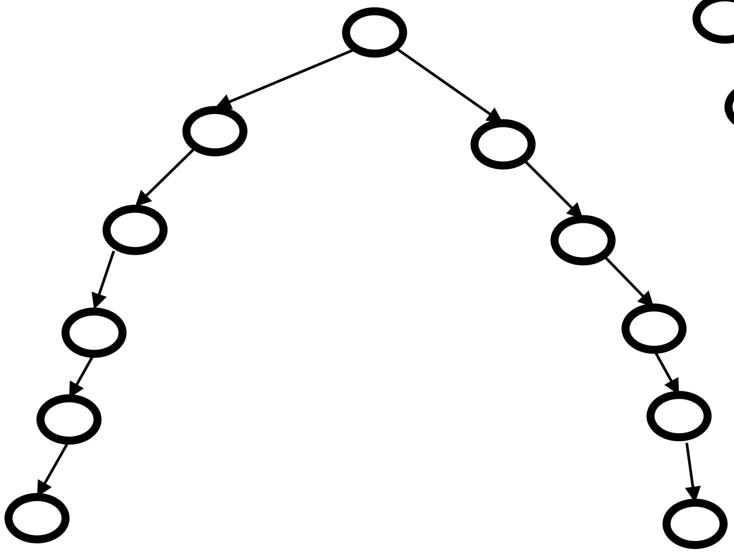
1. Left and right subtrees of the *root* have equal number of nodes

*Too weak!*
*Height mismatch example:*



2. Left and right subtrees of the *root* have equal *height*

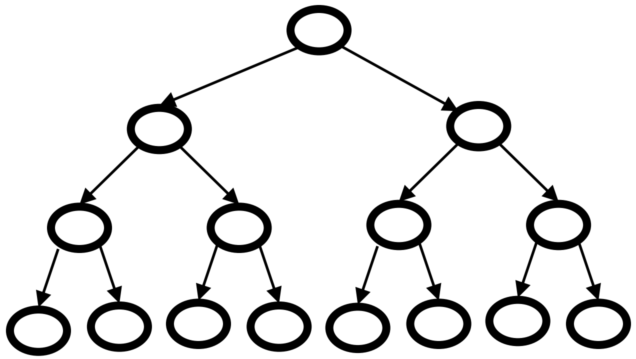
*Too weak!*
*Double chain example:*



# *Potential Balance Conditions*

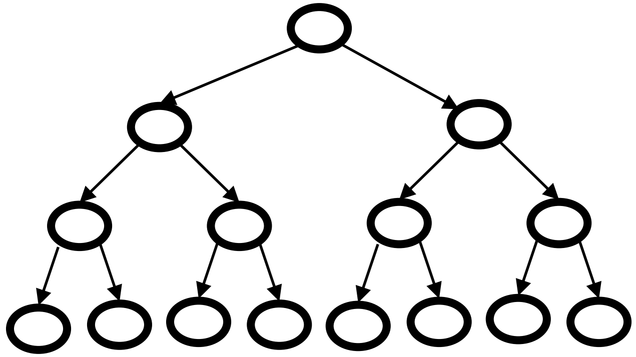
3. Left and right subtrees of every node have equal number of nodes

*Too strong!*
*Only perfect trees ($2^n$ – 1 nodes)*



4. Left and right subtrees of every node have equal *height*

*Too strong!*
*Only perfect trees ($2^n$ – 1 nodes)*



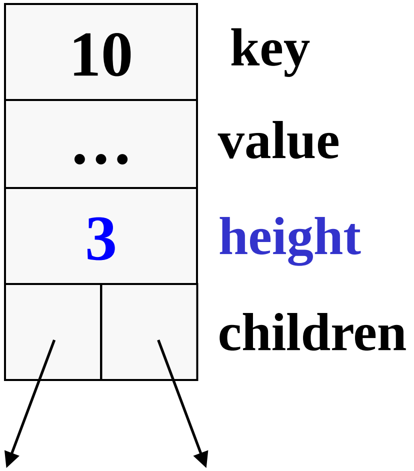
# *The AVL Balance Condition*

Left and right subtrees of *every node*
have *heights* **differing by at most 1**

*Definition*: **balance**(*node*) = height(*node*.left) – height(*node*.right)

AVL *property*: **for every node *x*, –1 ≤ balance(*x*) ≤ 1**

- Ensures small depth
  - Can prove by showing an AVL tree of height *h* must have nodes *exponential* in *h*
- Efficient to maintain
  - Using single and double rotations



# *Calculating Height*

What is the height of a tree with root **`r`**?

```
int treeHeight(Node root) {
  if(root == null)
    return -1;
  return 1 + max(treeHeight(root.left),
                 treeHeight(root.right));
}
```

Running time for tree with *n* nodes:

$O(n)$ – single pass over tree

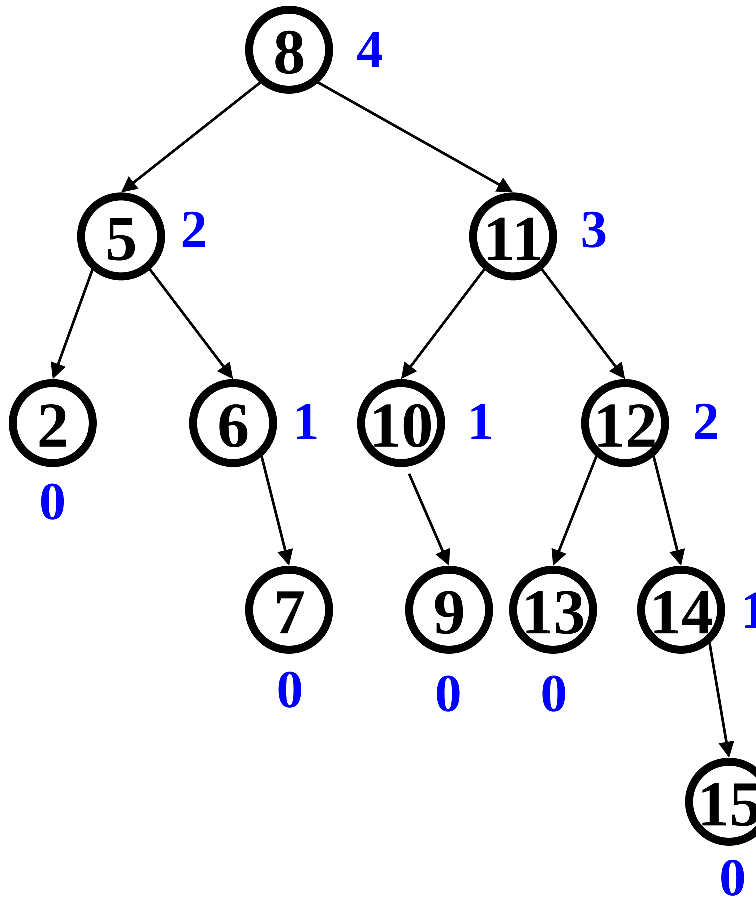
Very important detail of definition:

height of a null tree is -1, height of tree with a single node is 0

# *An AVL Tree?*

**This is the minimum AVL tree of height 4**

**Let S($h$) be the minimum nodes in height $h$**

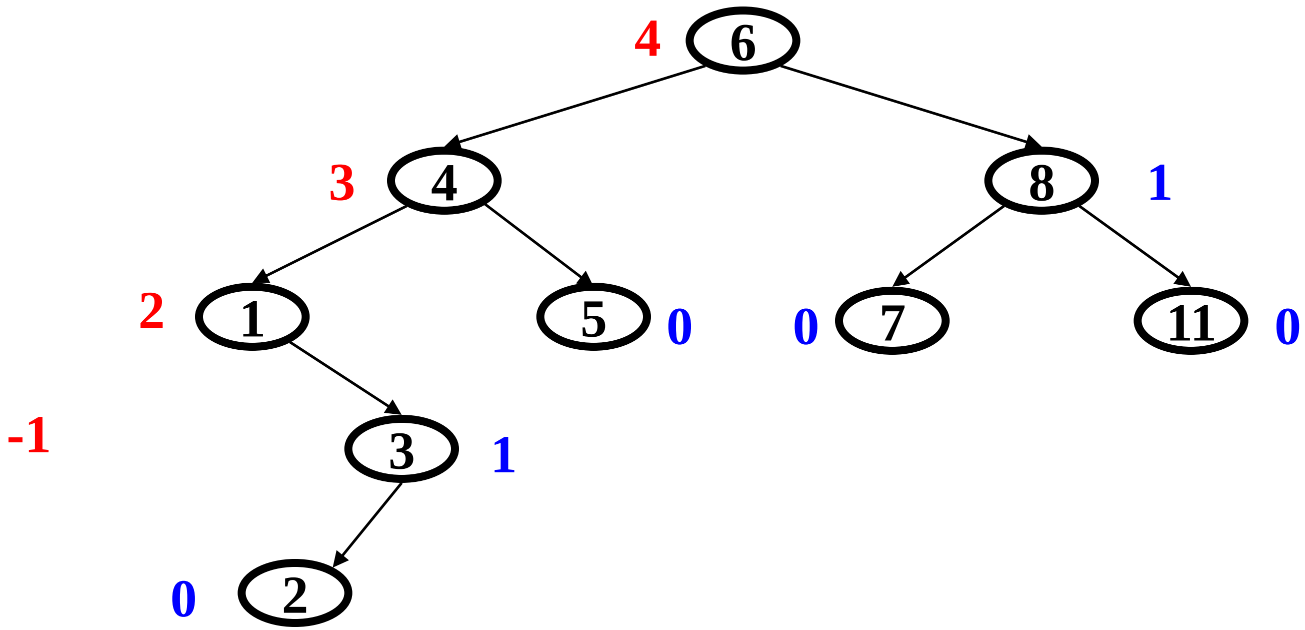



**$S(h) = S(h-1) + S(h-2) + 1$**

**$S(-1) = 0$**
**$S(0) = 1$**
**$S(1) = 2$**
**$S(2) = 4$**
**$S(3) = 7$**
**$S(4) = 12$**

**Solution of Recurrence: $S(h) \approx 1.62^h$**

# *An AVL Tree?*



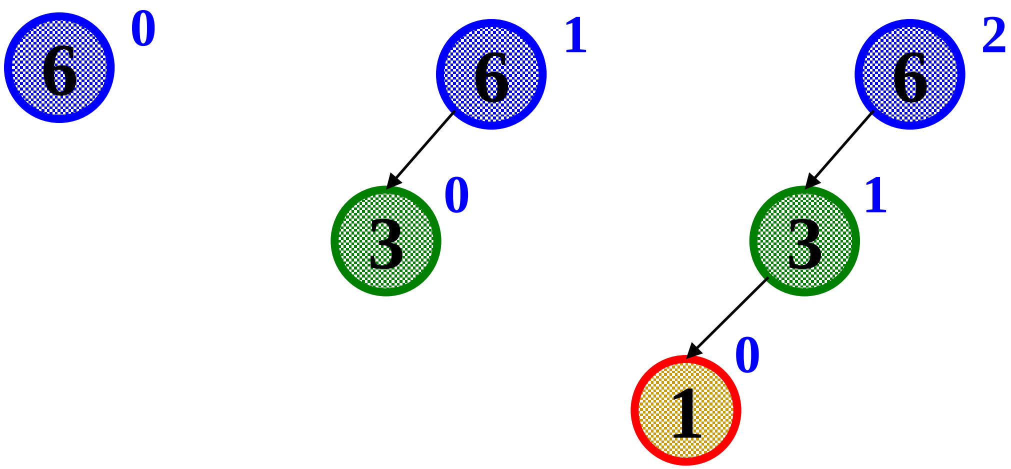
# *AVL Tree Operations*

- **AVL `find`**:
  - Same as BST **`find`**

- **AVL `insert`**:
  - Same as BST **`insert`**
    - then check balance and potentially fix the AVL tree
    - four different imbalance cases

- **AVL `delete`**:
  - As with insert, do the deletion and then handle imbalance

# *Example*




Insert(6)

Insert(3)

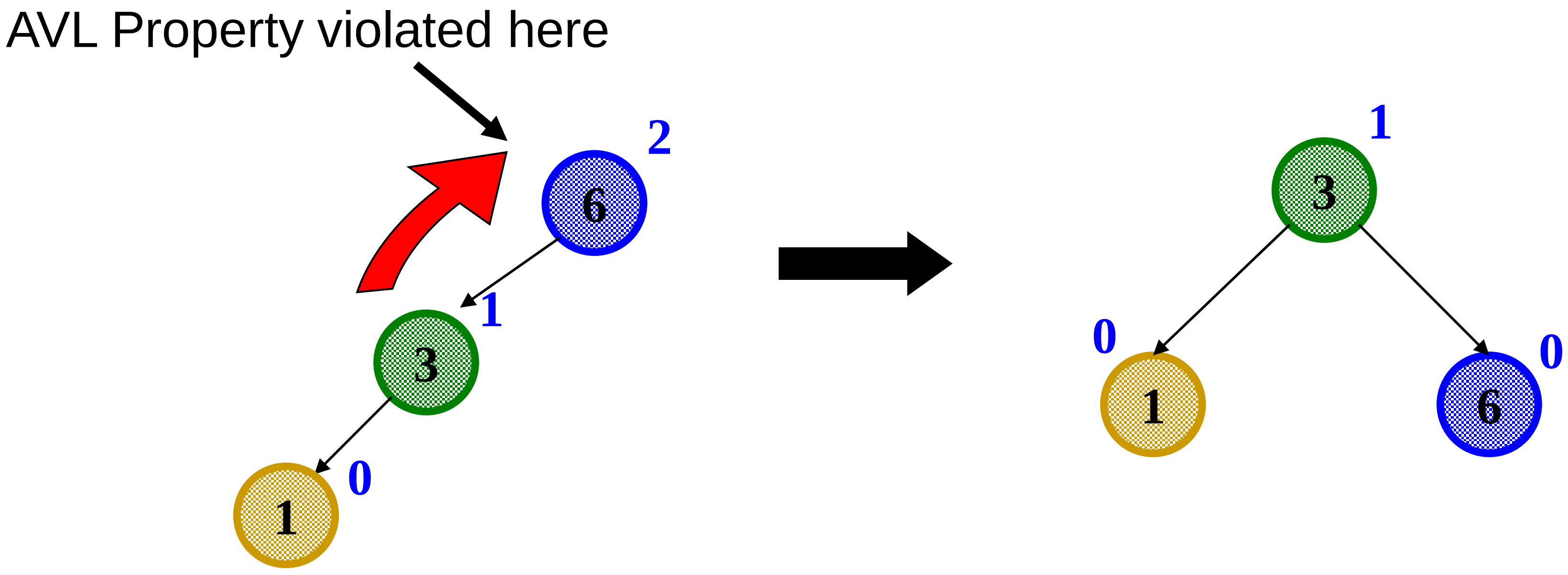
Insert(1)

Third insertion violates balance

What is the only way to fix this?

# *Single Rotation*

- *Single rotation:* The basic operation we use to rebalance
  - Move child of unbalanced node into parent position
  - Parent becomes a “other” child
  - Other subtrees move in the only way allowed by the BST



# *Insert and Detect Potential Imbalance*

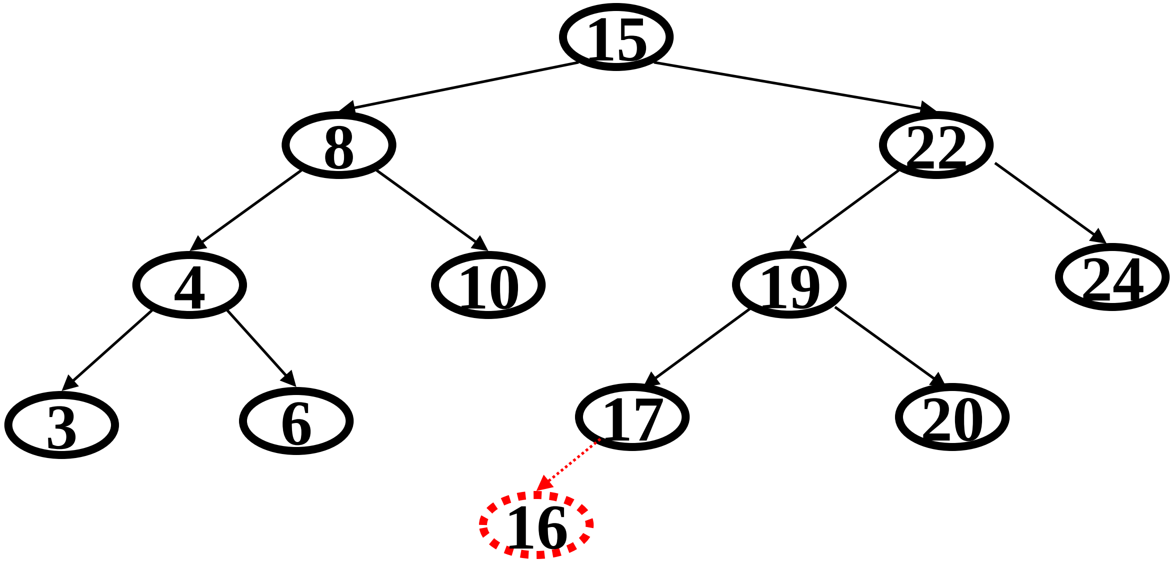
1. Insert the new node (at a leaf, as in a BST)
2. For each node on the path from the new leaf to the root
   the insertion may, or may not, have changed the node's height
3. After recursive insertion in a subtree
   detect height imbalance
   perform a *rotation* to restore balance at that node

*All the action is in defining the correct rotations to restore balance*

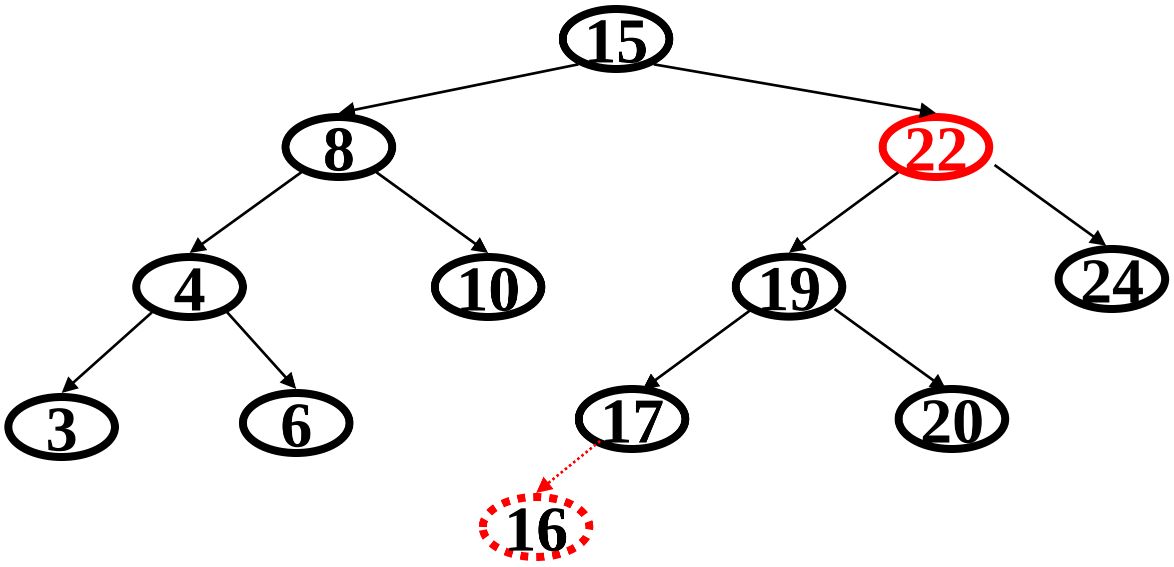
Fact that an implementation can ignore:

- There must be a deepest element that is imbalanced
- After rebalancing this deepest node, every node is balanced
- So at most one node needs to be rebalanced

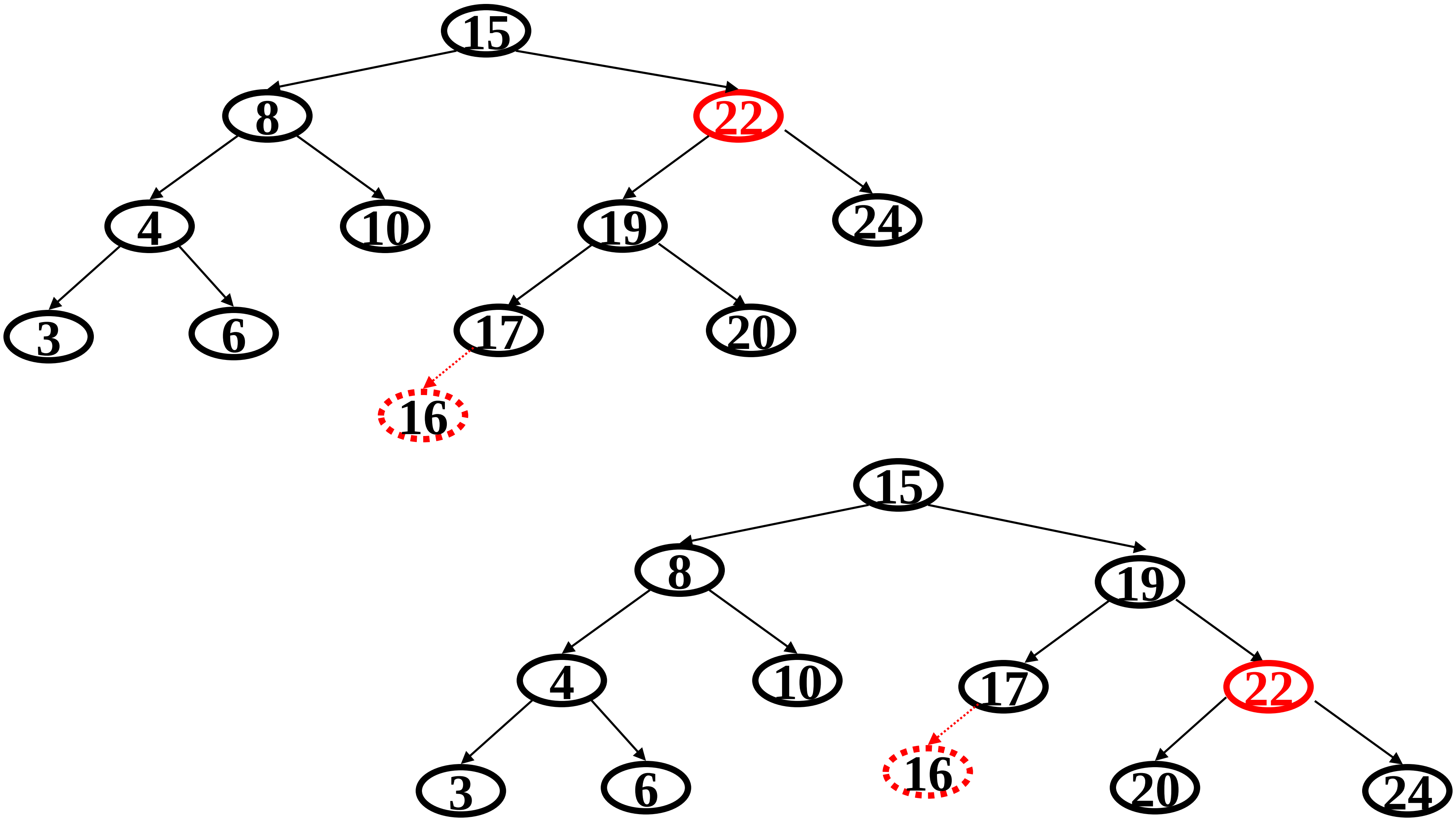
# Single Rotation Example: Insert(16)



# *Single Rotation Example: Insert(16)*



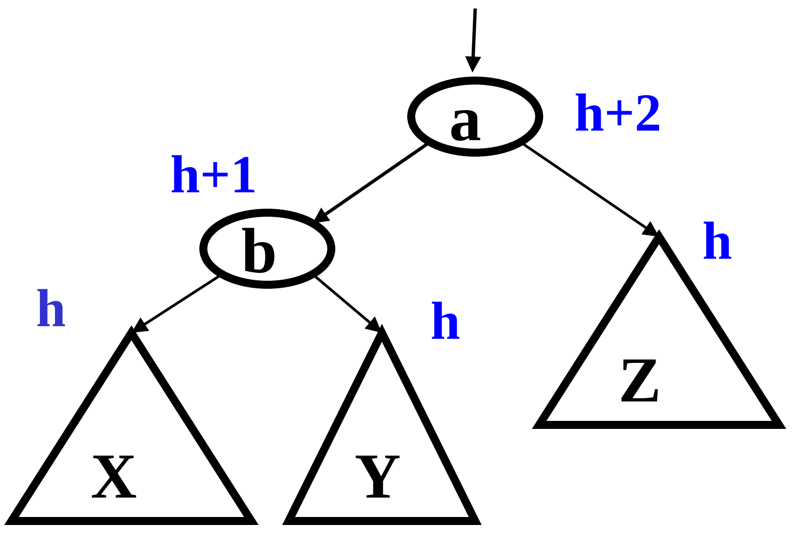
# *Single Rotation Example: Insert(16)*

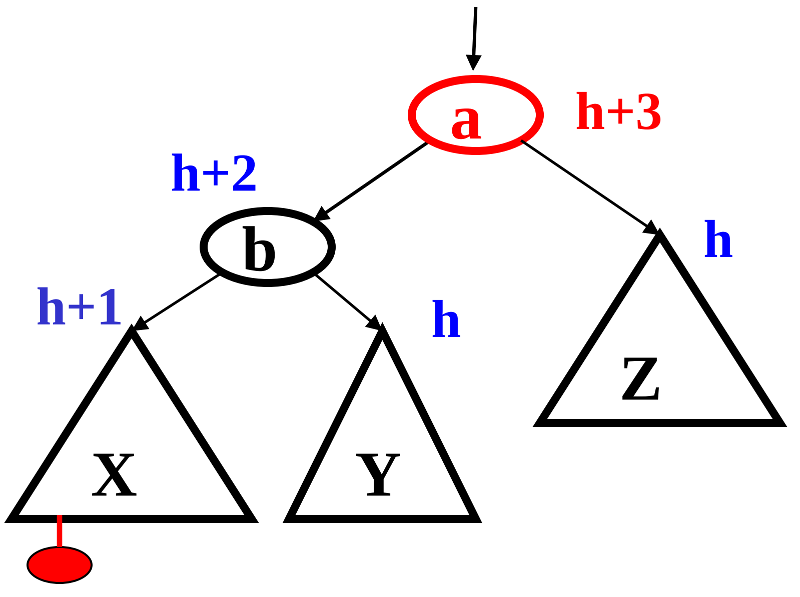


# *Left-Left Case*

- Node imbalanced due to insertion in **left-left grandchild**
  - This is 1 of 4 possible imbalance cases

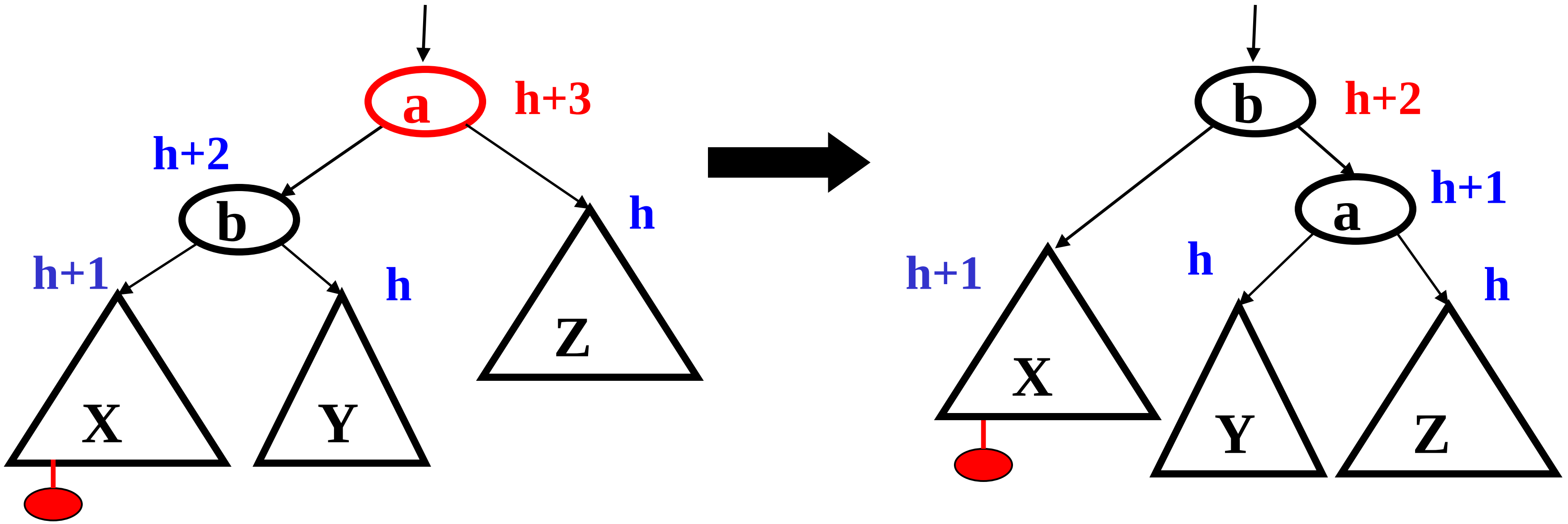
- First we did the insertion, which made *a* imbalanced



# *Left-Left Case*

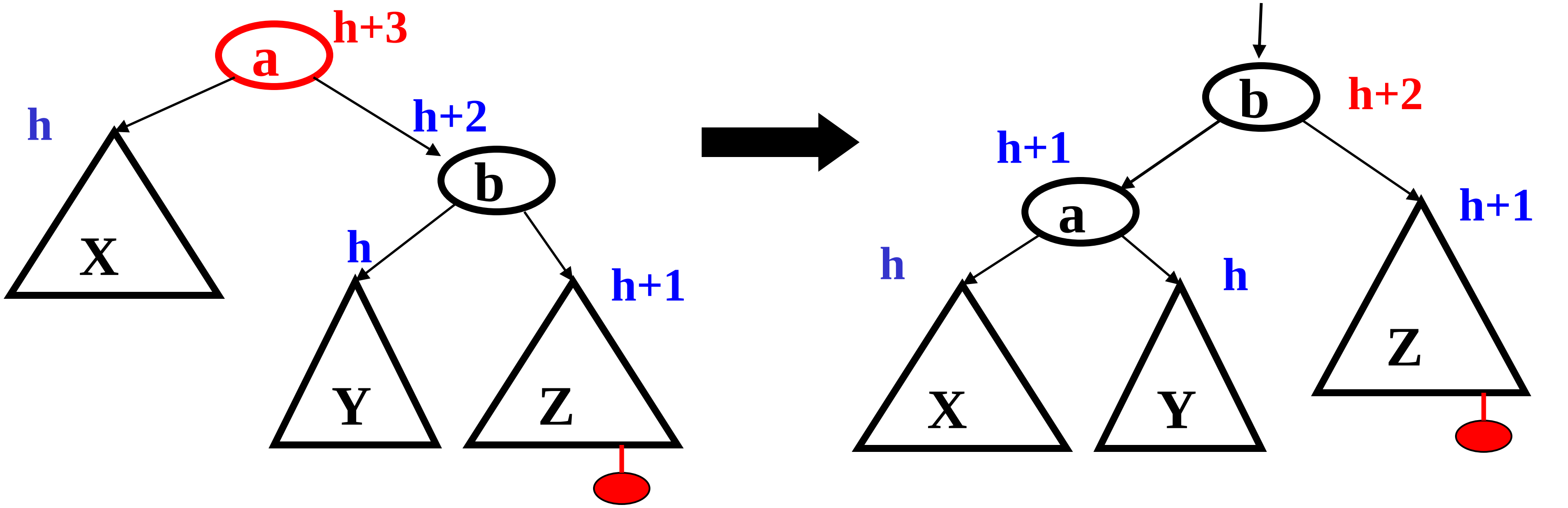
- So we rotate at ***`a,`*** using BST facts: X < b < Y < a < Z




- A single rotation restores balance at the node
  - Is same height as before insertion, so ancestors now balanced

# *Right-Right Case*

- Mirror image to left-left case, so you rotate the other way
  - Exact same concept, but need different code

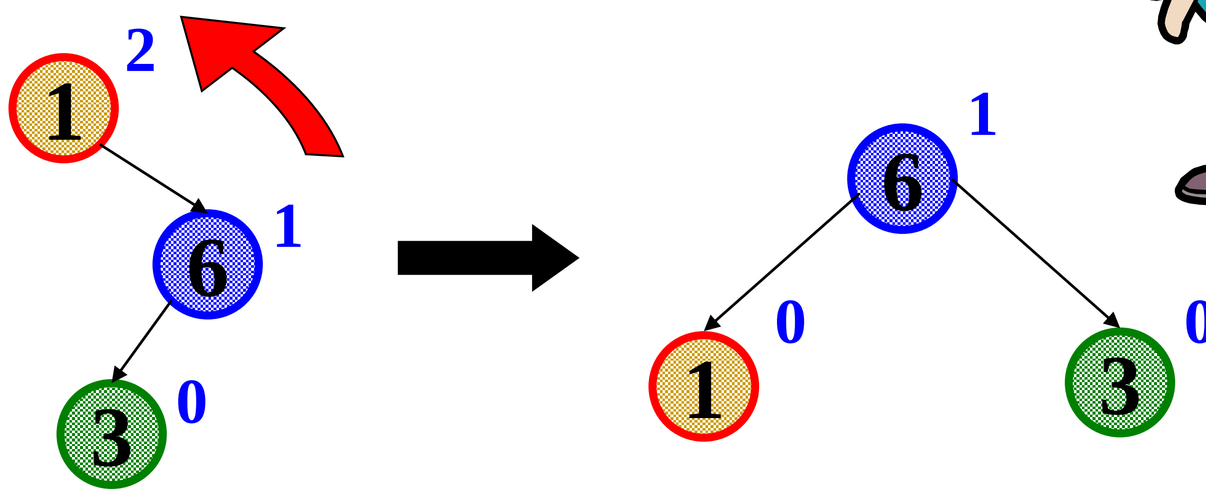


# *The Other Two Cases*

Single rotations not enough for insertions left-right or right-left subtree

Simple example: **`insert`**(1), **`insert`**(6), **`insert`**(3)

First wrong idea: single rotation as before

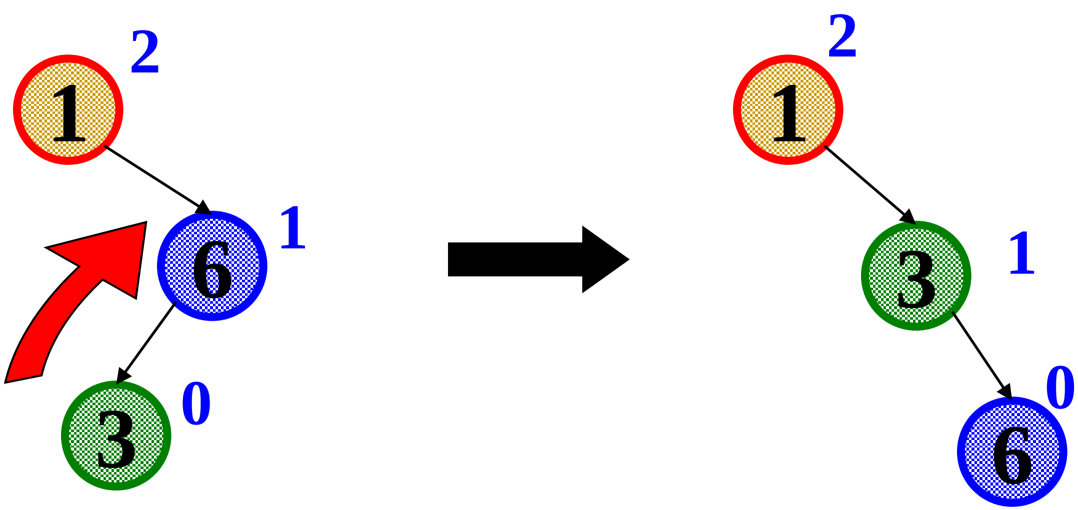



# *The Other Two Cases*

Single rotations not enough for insertions left-right or right-left subtree

Simple example: **`insert`**(1), **`insert`**(6), **`insert`**(3)

Second wrong idea: single rotation on child

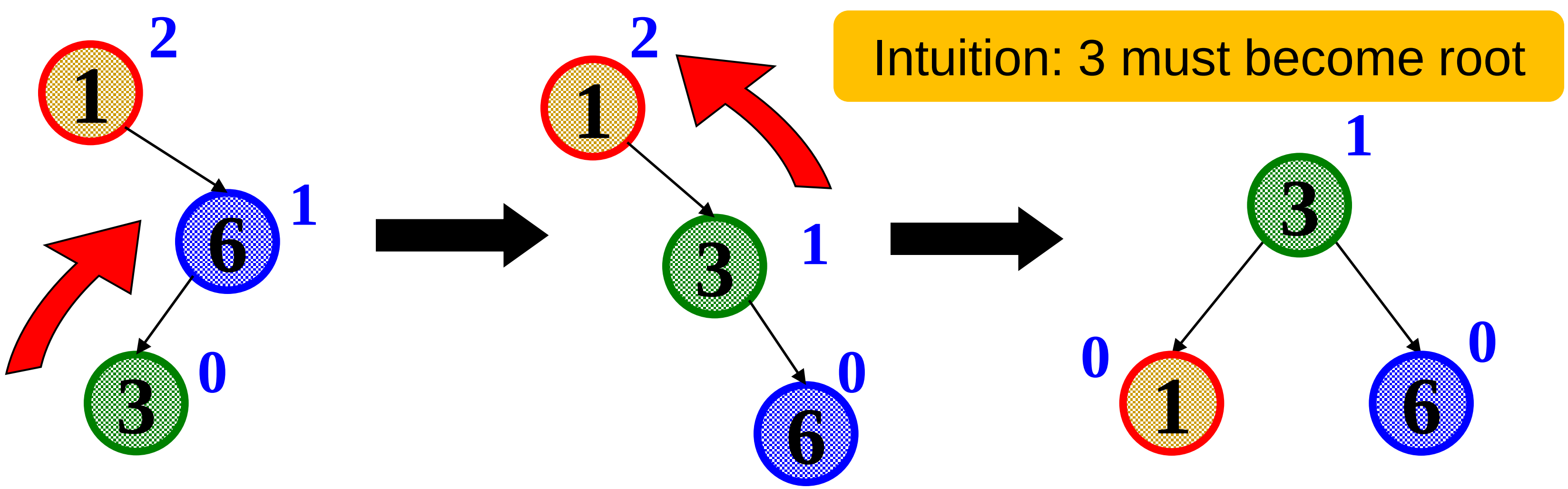



# *Double Rotation*

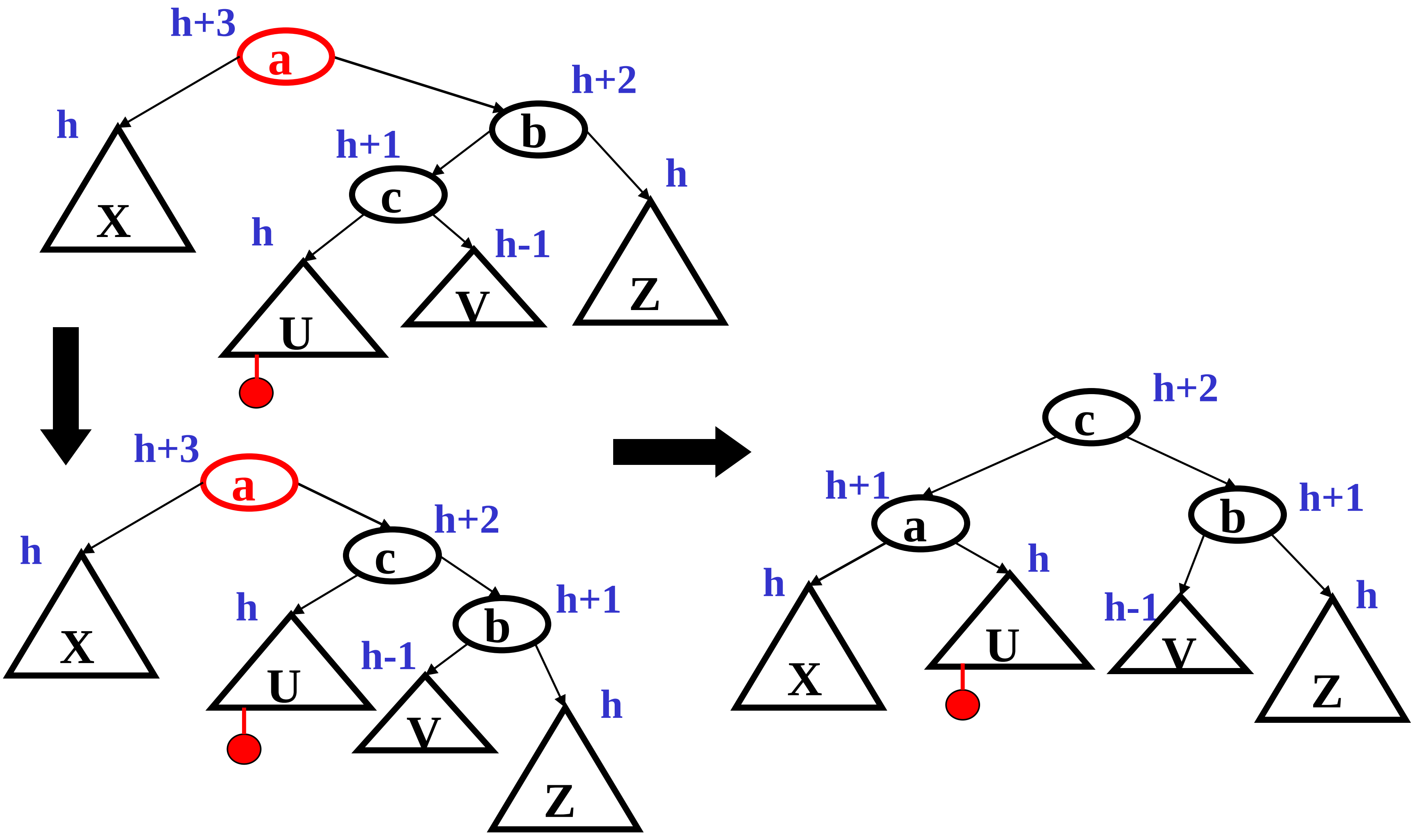
- First attempt at rotation violated the BST property
- Second attempt at rotation did not fix balance
- But if we do both, it works!

Double rotation:

1. Rotate problematic child and grandchild
2. Then rotate between self and new child

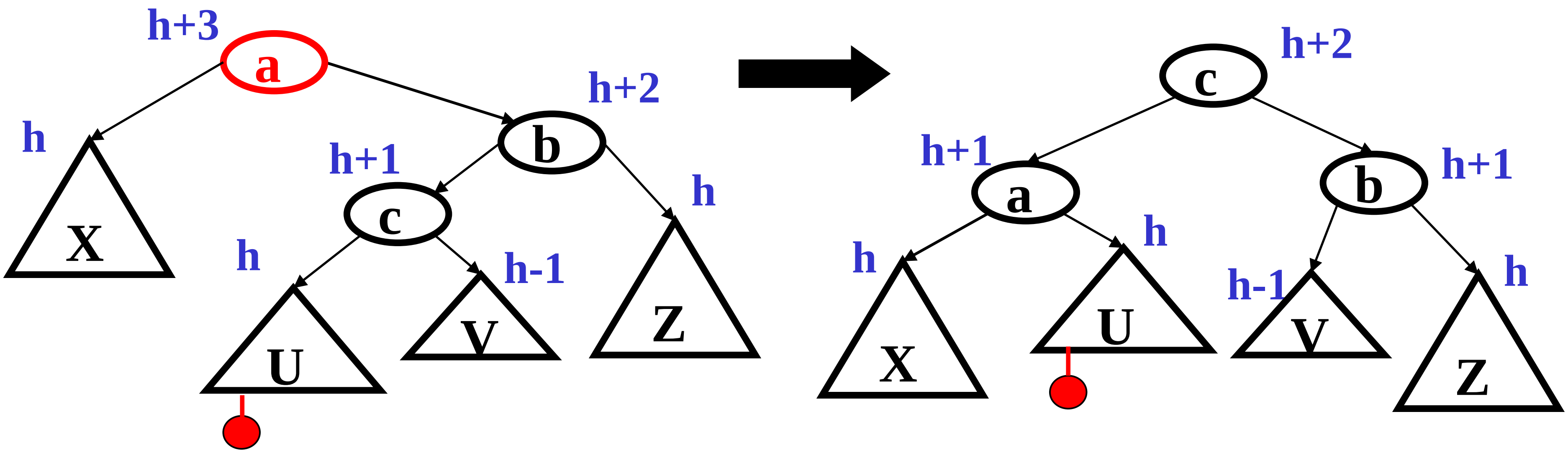


# *Right-Left Case*



# *Right-Left Case*

- Height of the subtree after rebalancing is the same as before insert
  - So no ancestor in the tree will need rebalancing
- Does not have to be implemented as two rotations; can just do:



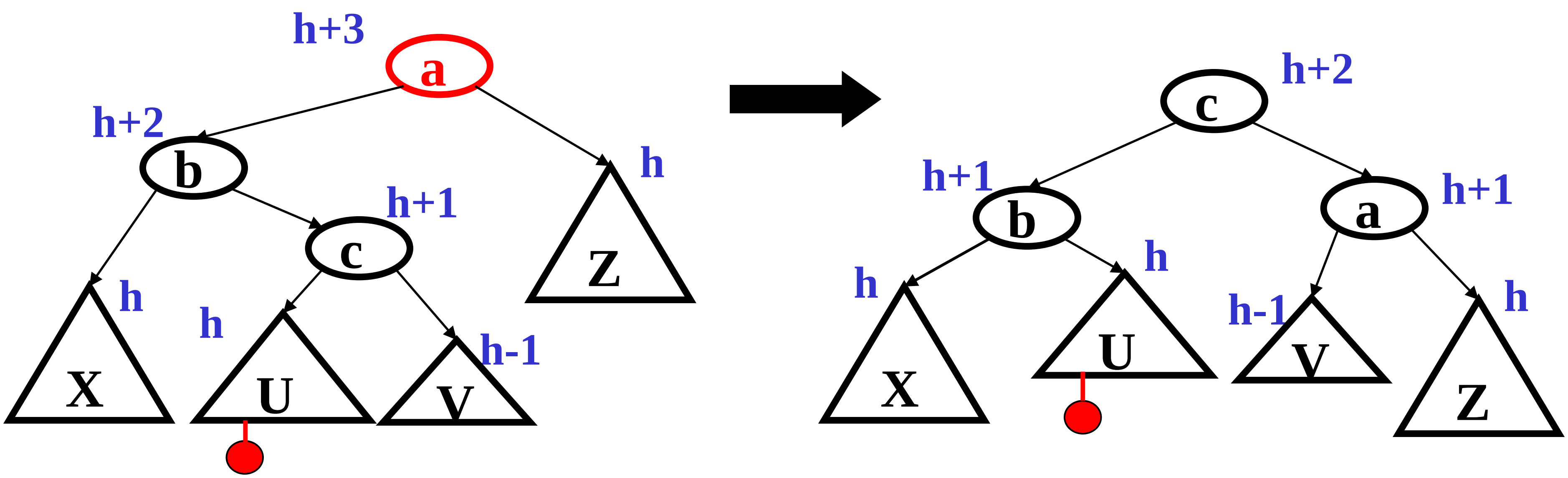

Easier to remember than you may think:

Move c to grandparent's position

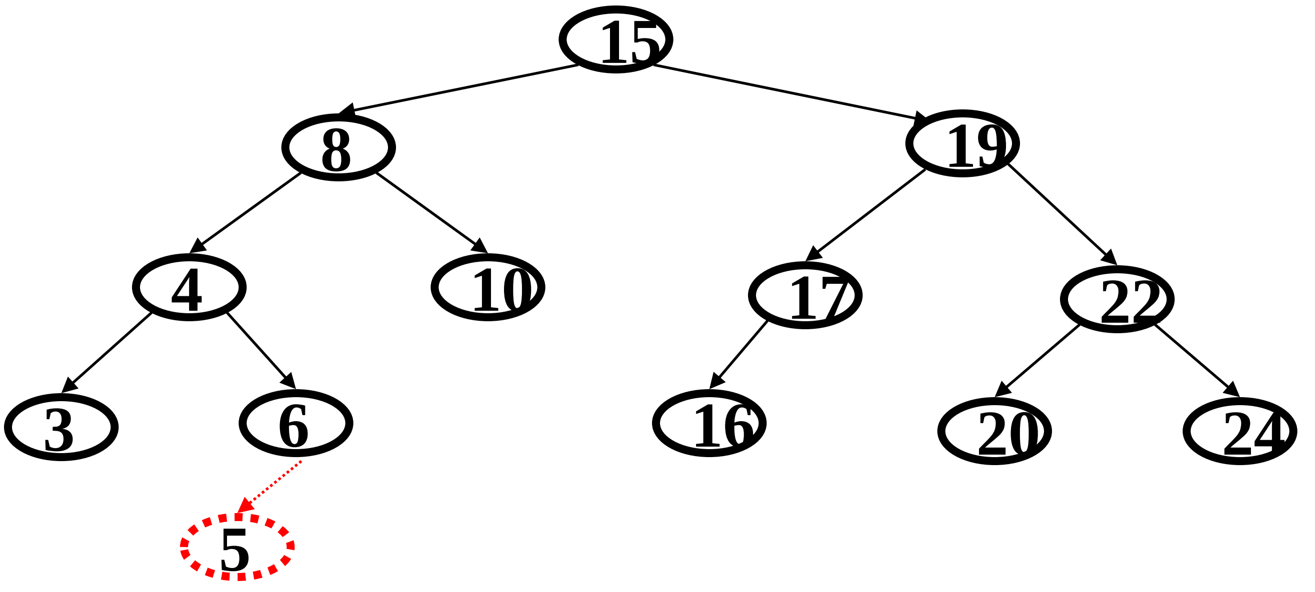
Put a, b, X, U, V, and Z in the only legal position for a BST

# *Left-Right Case*

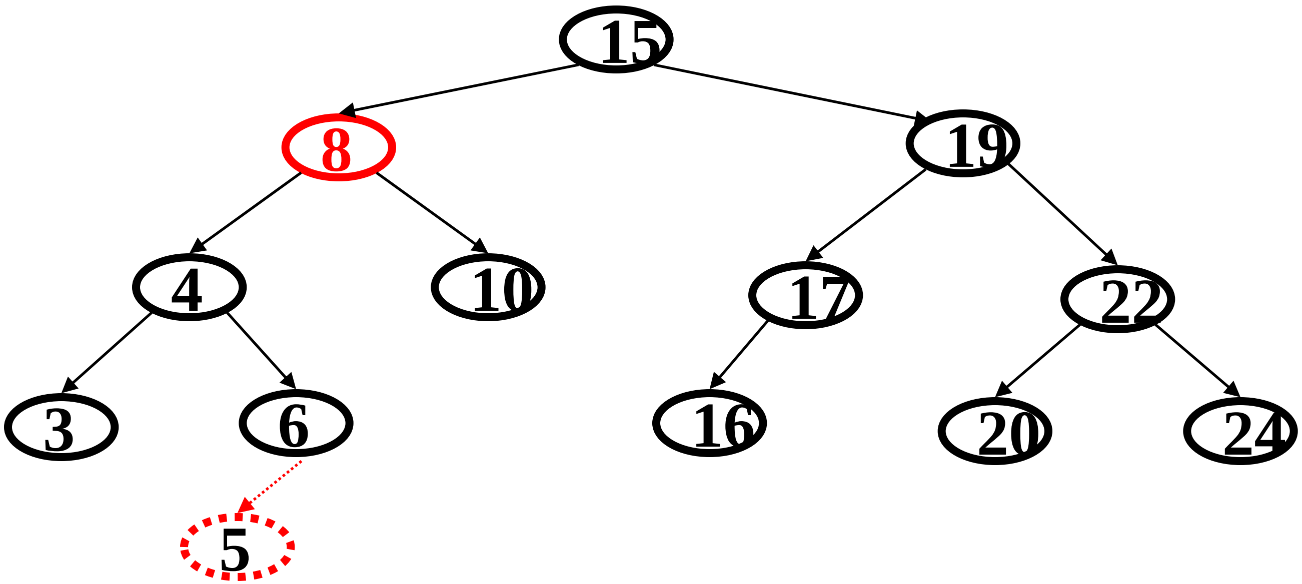
- Mirror image of right-left
  - No new concepts, just additional code to write



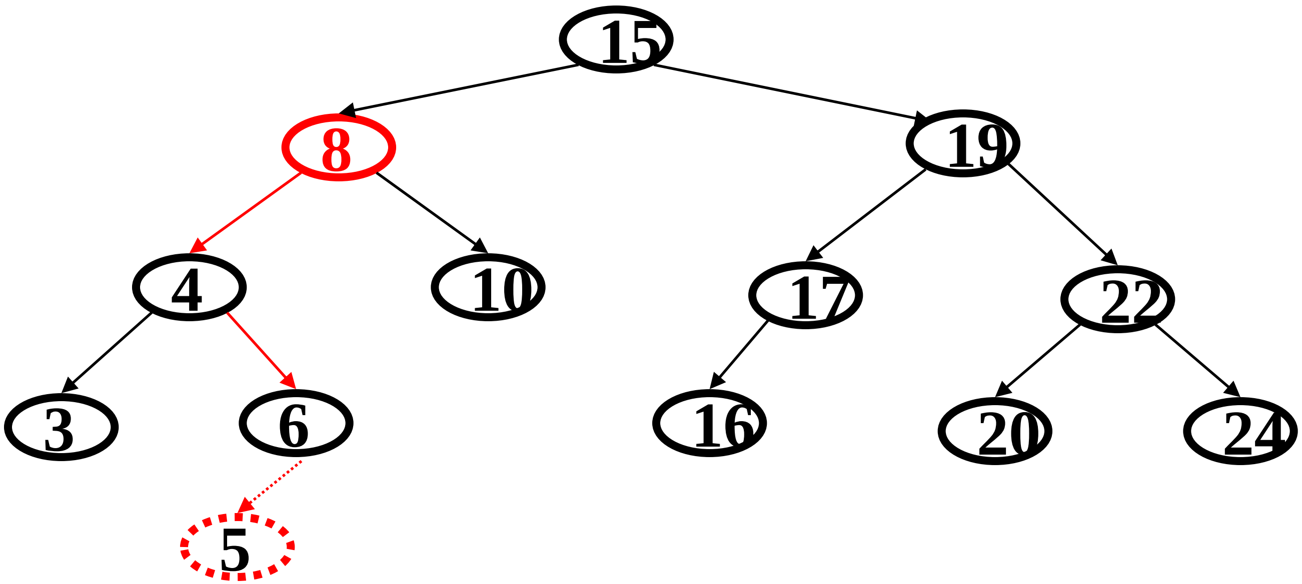
# *Double Rotation Example: Insert(5)*



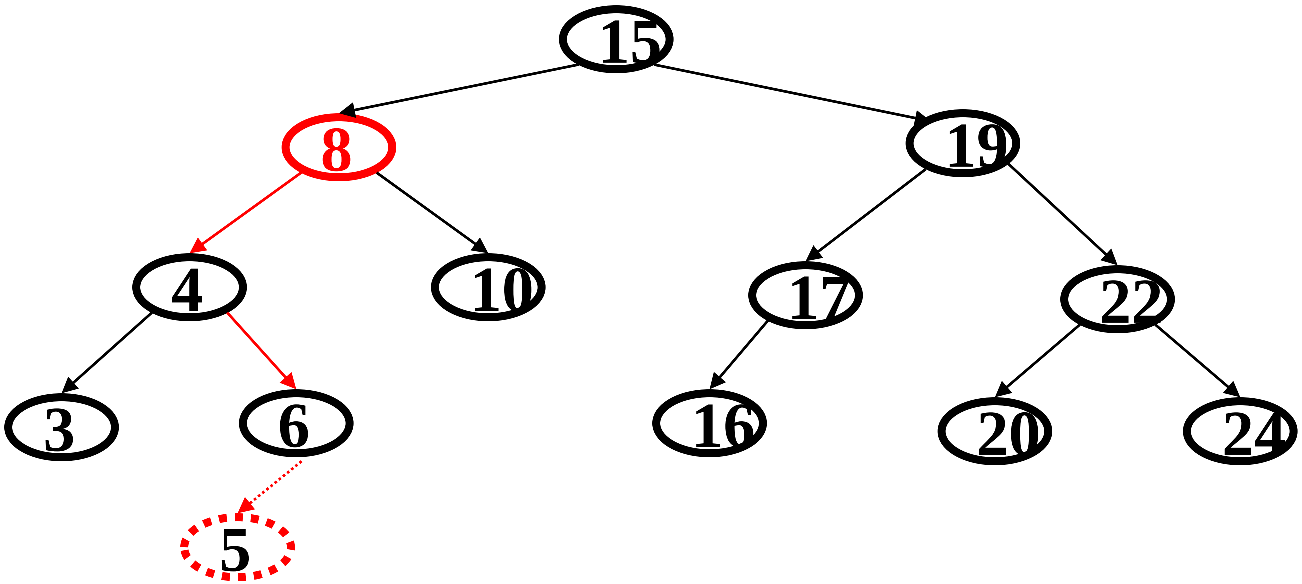
# *Double Rotation Example: Insert(5)*

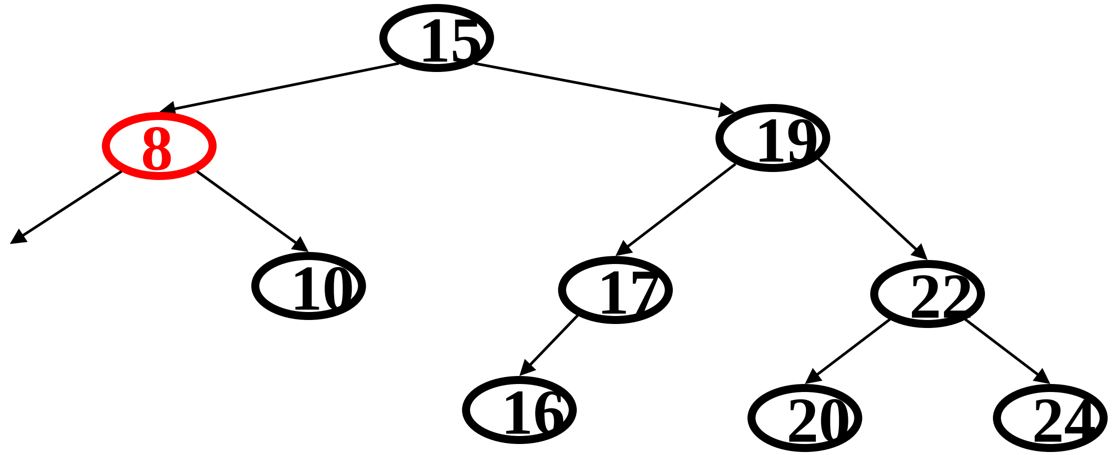


# *Double Rotation Example: Insert(5)*

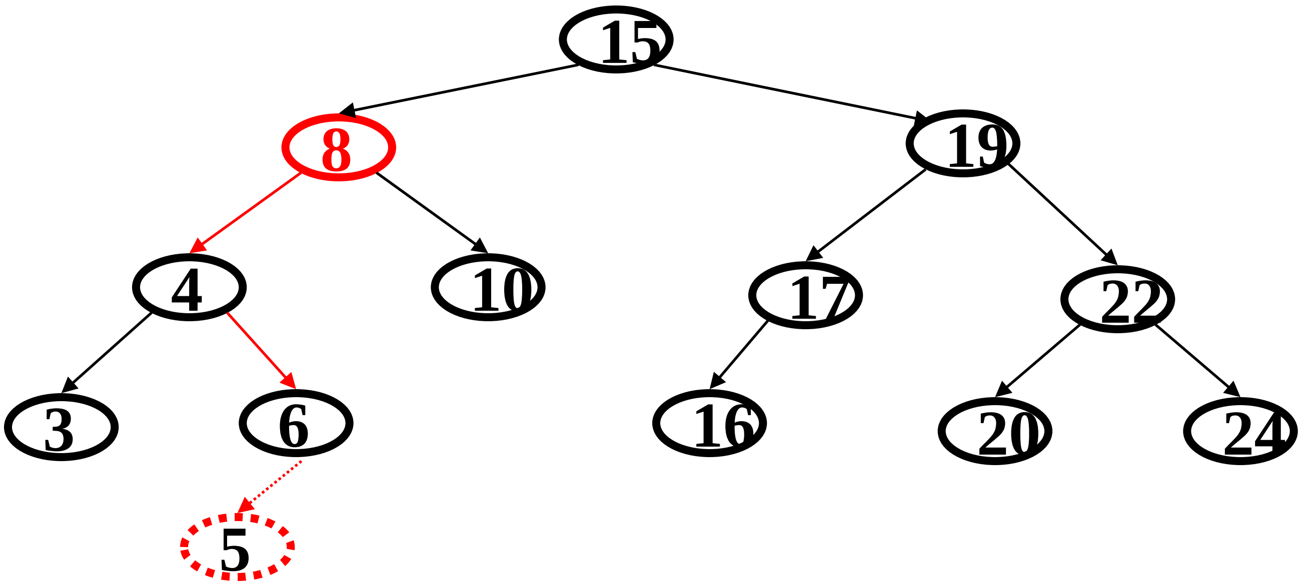


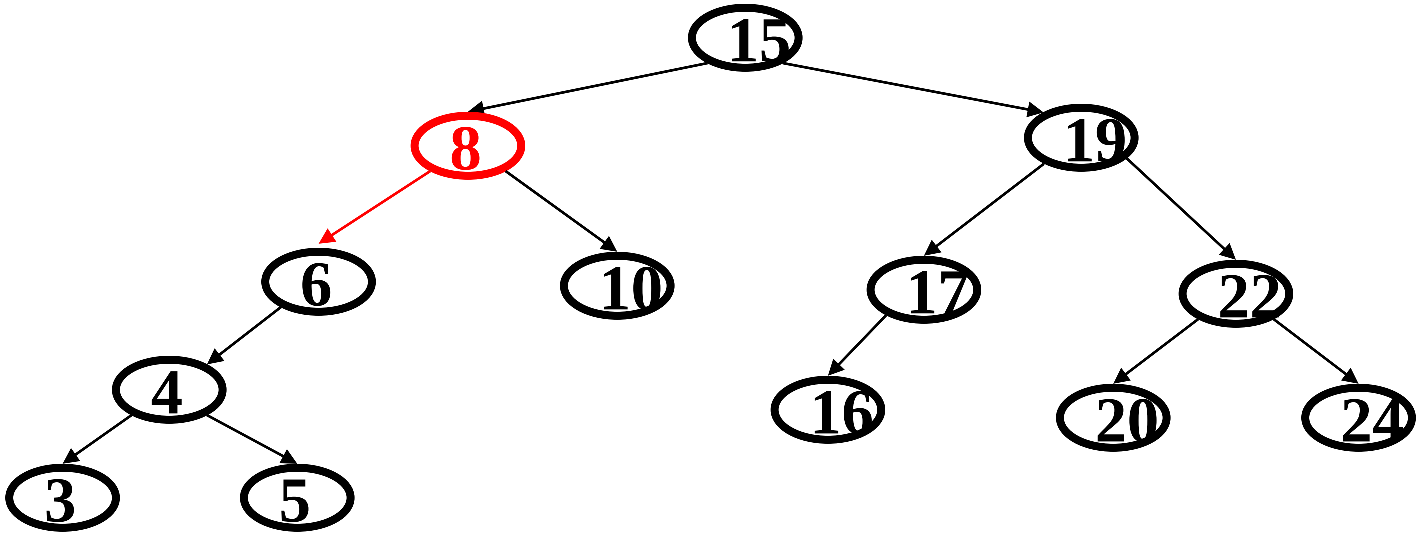
# Double Rotation Example: Insert(5)



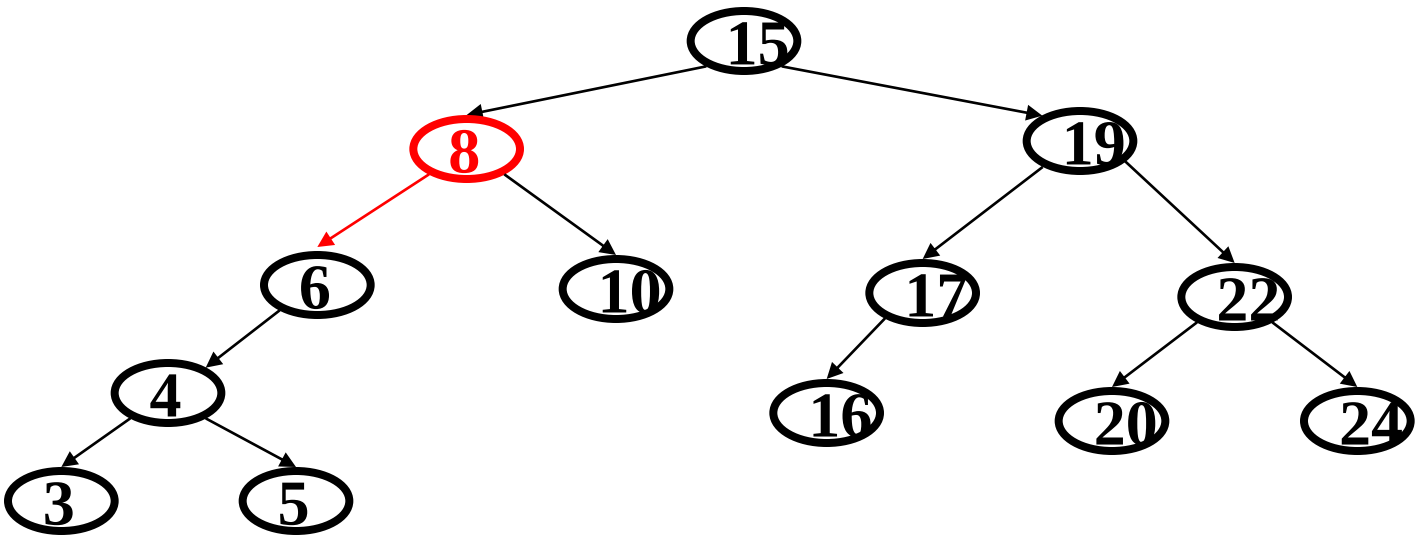



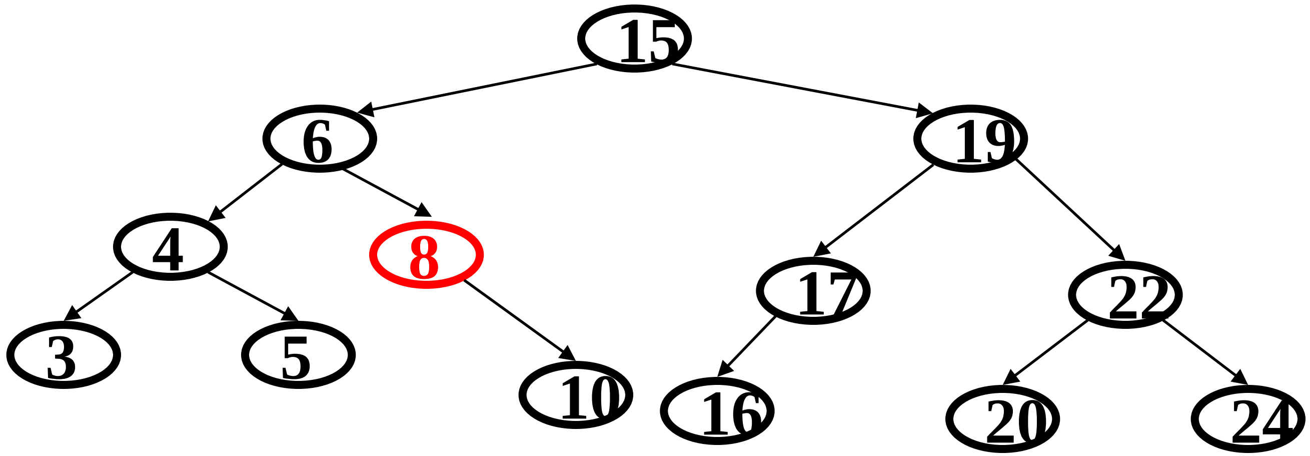
# *Double Rotation Example: Insert(5)*



# *Double Rotation Example: Insert(5)*



# *Summarizing Insert*

- Insert as in a BST
- Check back up path for imbalance, which will be 1 of 4 cases:
  - node's left-left grandchild is too tall
  - node's left-right grandchild is too tall
  - node's right-left grandchild is too tall
  - node's right-right grandchild is too tall
- Only one case can occur, because tree was balanced before insert
- After the single or double rotation, the smallest-unbalanced subtree now has the same height as before the insertion
  - So all ancestors are now balanced

# *Efficiency*

Worst-case complexity of **`find`**: $O(\mathtt{log}\ n)$

Worst-case complexity of **`insert`**: $O(\mathtt{log}\ n)$

- Rotation is $O(1)$ and there's an $O(\mathtt{log}\ n)$ path to root
- Same complexity even without "one-rotation-is-enough" fact

Worst-case complexity of **`buildTree`**: $O(n\ \mathtt{log}\ n)$

# *Delete*

We will not cover delete

- Multiple snow days, something has to give

Do the delete as in a BST, then balance path up from deleted node

- Which may be predecessor or successor

Single and double rotate based on height imbalance

- You are coming up the shorter subtree
- But need to pull up the taller subtree

Rotation reduces height of the tree

- So you need to check all the way to the root

**`delete`** is also $O($**`log`** $n)$