Binary Search Trees

Kyunghan Lee
Networked Computing Lab (NXC Lab)
Department of Electrical and Computer Engineering
Seoul National University
https://nxc.snu.ac.kr
kyunghanlee@snu.ac.kr

Outline

- This topic covers binary search trees:
  - Abstract Sorted Lists
  - Background
  - Definition and examples
  - Implementation:
    - FindMin, FindMax, insert, erase
    - Previous smaller and next larger objects
    - Finding the k^{th} object

Abstract Sorted Lists

- Previously, we discussed Abstract Lists: the objects are linearly ordered by the programmer

- We will now discuss the Abstract Sorted List:
  - The relation is based on an implicit linear ordering

Abstract Sorted Lists (ASL)

- Queries that may be made about data stored in a Sorted List ADT include:
  - Finding the smallest and largest values
  - Finding the k^{th} largest value
  - Find the next larger or previous smaller objects of a given object
  - Iterate through objects within an interval [a, b]

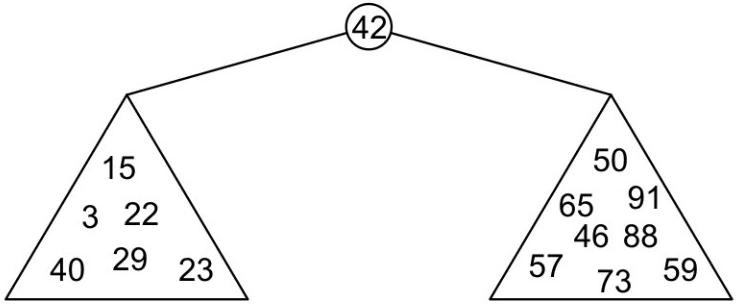
Limitation: ASL with Array or Linked Lists

- If we implement an Abstract Sorted List using an array or a linked list, some operations take O(n)
  - To perform insertion, we may either traverse or copy, on average, O(n) objects

Binary Search Trees

- Using a binary tree, we can dictate an order on the two children

- We will exploit this order:
  - All objects in the left sub-tree to be less than the object stored in the root node, and
  - All objects in the right sub-tree to be greater than the object in the root object



Recursive definition: Each of the two sub-trees will themselves be binary search trees

Binary Search Trees

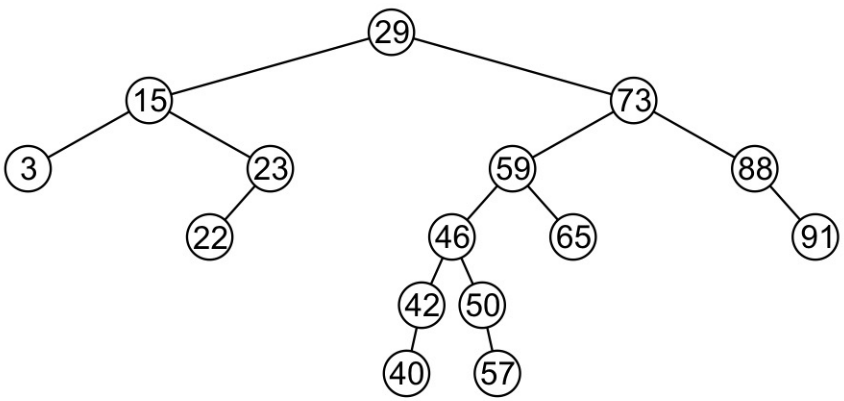
- We can use this structure for searching
- With a linear order, one of the following three must be true:

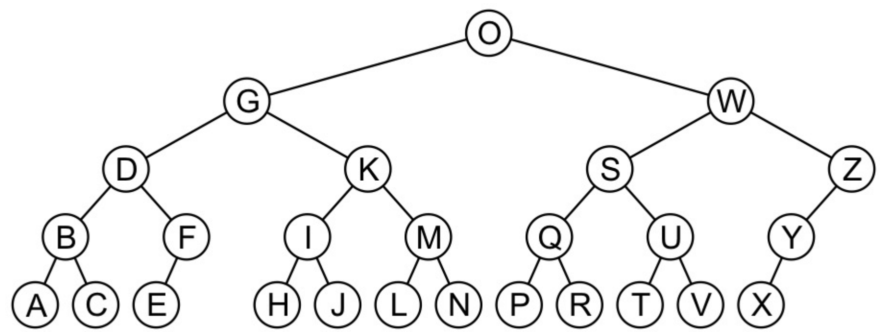
$$a < b \quad a = b \quad a > b$$

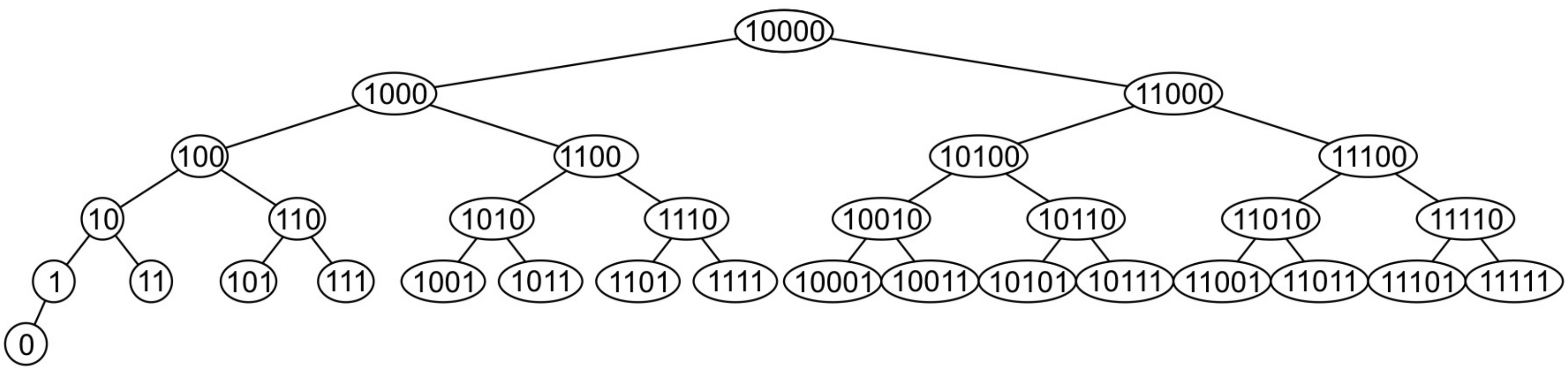
- Examine the root node and if we have not found what we are looking for:
  - If the object is less than what is stored in the root node, continue searching in the left sub-tree
  - Otherwise, continue searching the right sub-tree

Binary Search Trees: Good Example

- Here are other examples of binary search trees:

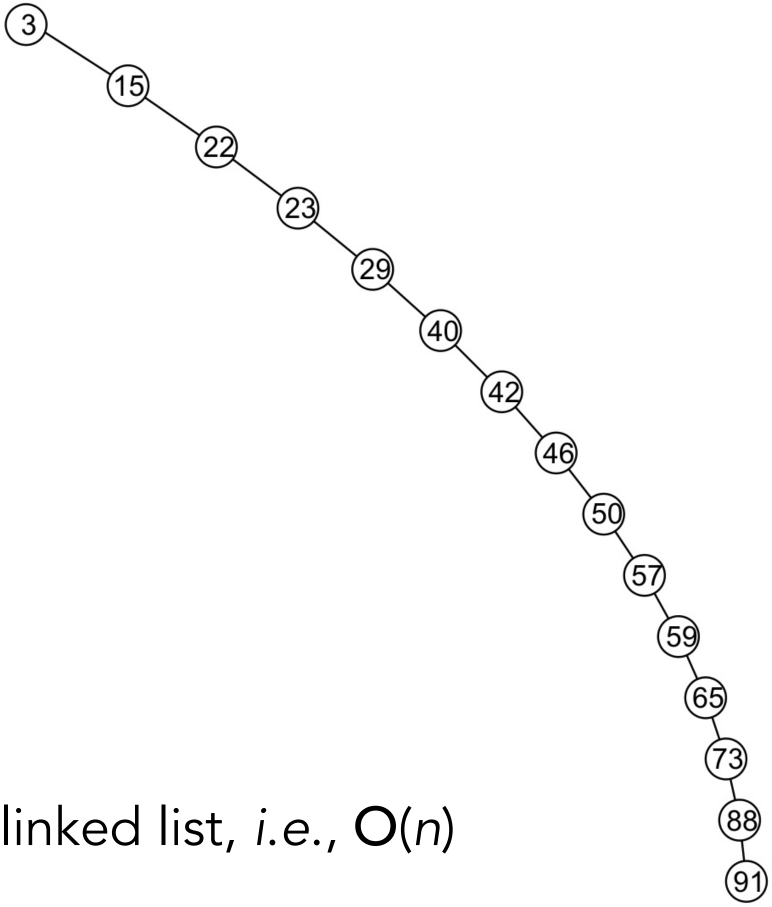






Binary Search Trees: Bad Example

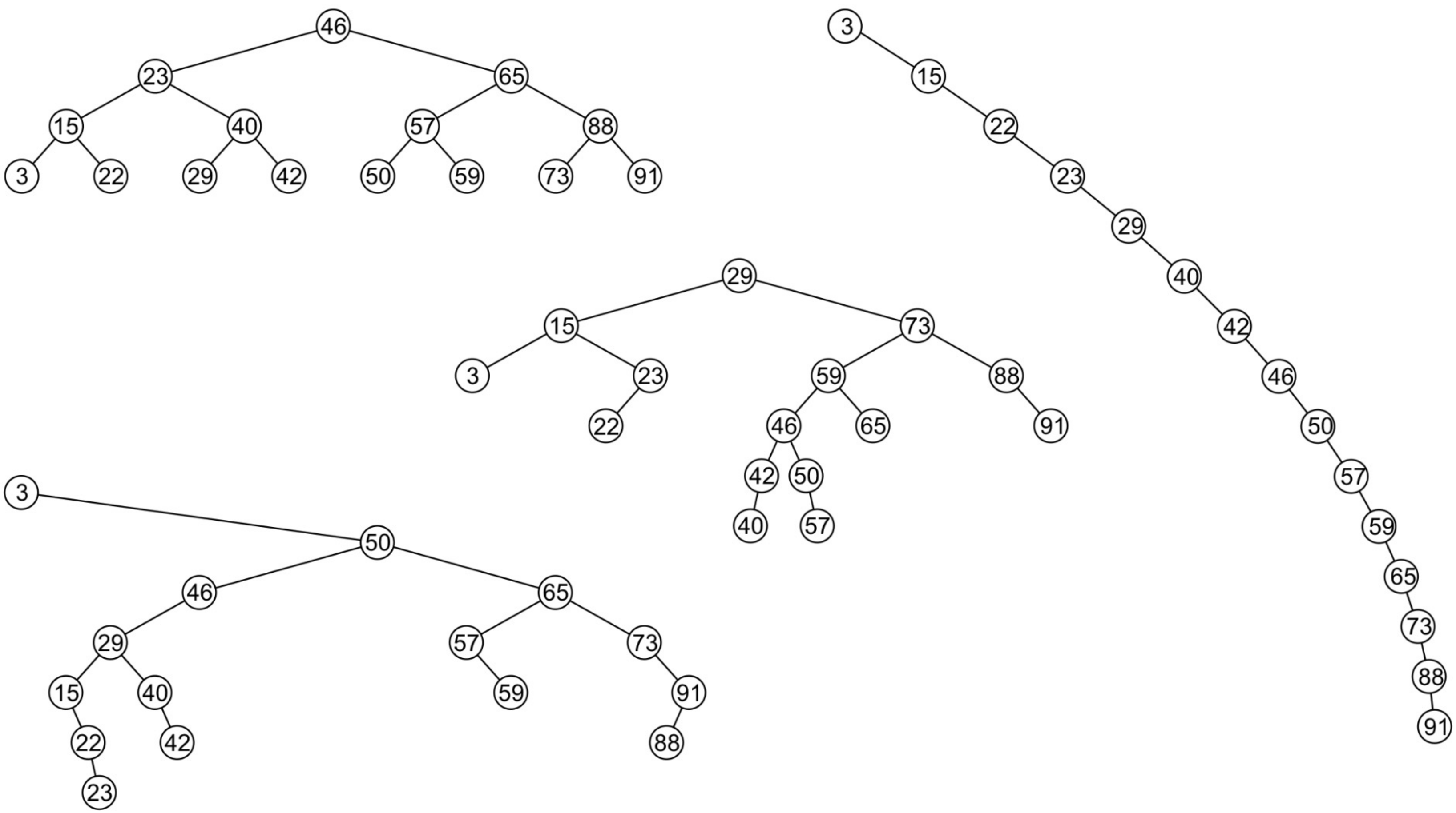
- Unfortunately, it is possible to construct *unbalanced* binary search trees



  - This is equivalent to a linked list, *i.e.*, O(n)

Binary Search Trees: More Examples

- All these binary search trees store the same data



Note: No Duplicate Values

- We will assume that in any binary tree, we are not storing duplicate values unless otherwise stated

- You can always consider duplicate values with modifications to the algorithms we will cover

Implementation: Binary Search Trees

- Design with two classes:

1) BinaryNode
   - Represent each node in the tree

2) BinarySearchTree
   - Represent the tree, which holds the root node (an instance of BinaryNode)

Implementation: class BinaryNode

```
template <typename T>
struct BinaryNode {
    T value;
    BinaryNode *left;
    BinaryNode *right;

    BinaryNode<T>(const T &value, BinaryNode<T> *left, BinaryNode<T> *right)
        : value{value}, left{left}, right{right} {}

    BinaryNode<T>(T &&value, BinaryNode<T> *left, BinaryNode<T> *right)
        : value{std::move(value)}, left{left}, right{right} {}
};
```

Recall the concept of reference variable from "T &value"

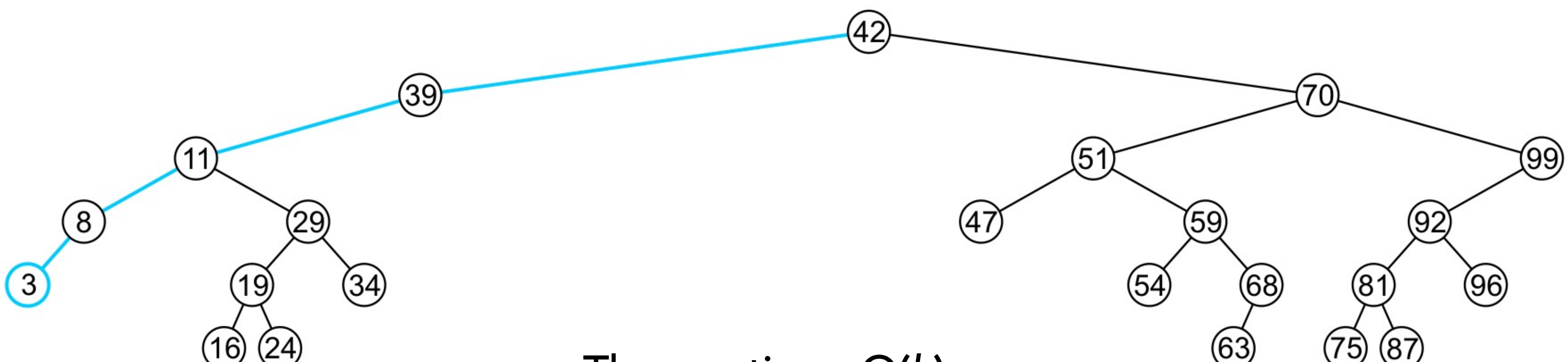
- A value has a template based type <T>
- If <T> is not comparable, you will need to override comparison operators

Implementation: class BinarySearchTree

```
template <typename T>
class BinarySearchTree {
public:
    BinarySearchTree() : root{nullptr} {}

    const T &findMin() const;
    const T &findMax() const;

    bool find(const T &x) const;
    void insert(const T &x);
    void remove(const T &x);
    // something more …

private:
    BinaryNode<T> *root;
    // something more …
};
```

Finding the Minimum Object

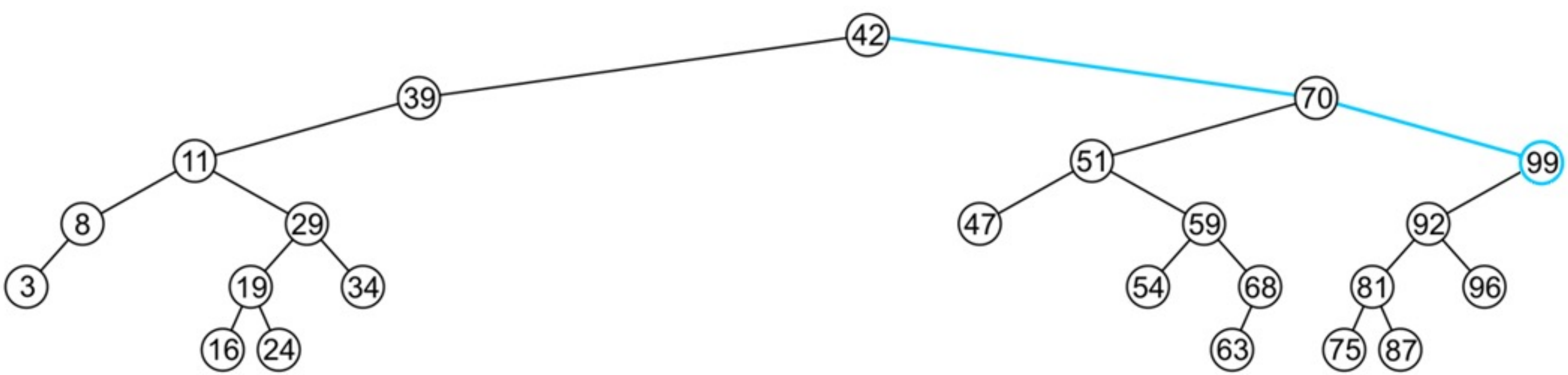
```
const T &findMin() const {
    if (isEmpty())
        throw std::exception{};
    return findMin(root)->value;
}

BinaryNode<T> *findMin(BinaryNode<T> *t) const {
    if (t == nullptr)
        return nullptr;
    if (t->left == nullptr)
        return t;
    return findMin(t->left);
}
```



The run time: $O(h)$

Finding the Maximum Object

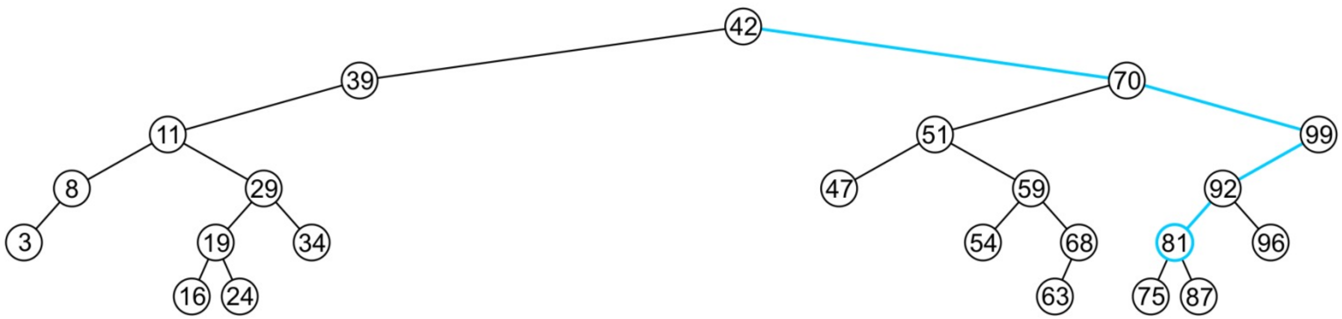
```
const T &findMax() const {
    if (isEmpty())
        throw std::exception{};
    return findMax(root)->value;
}

BinaryNode<T> *findMax(BinaryNode<T> *t) const {
    if (t != nullptr)
        while (t->right != nullptr)
            t = t->right;
    return t;
}
```



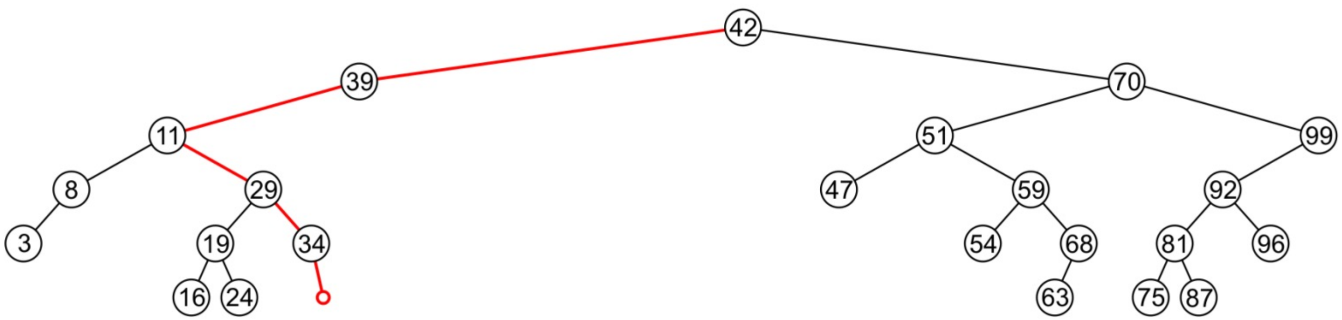
minimum/maximum values are not necessarily leaf nodes

Find

- To determine membership, traverse the tree based on the linear relationship:
  - If a node containing the value is found, *e.g.*, 81, return true



  - If an empty node is reached, *e.g.*, 36, return false:



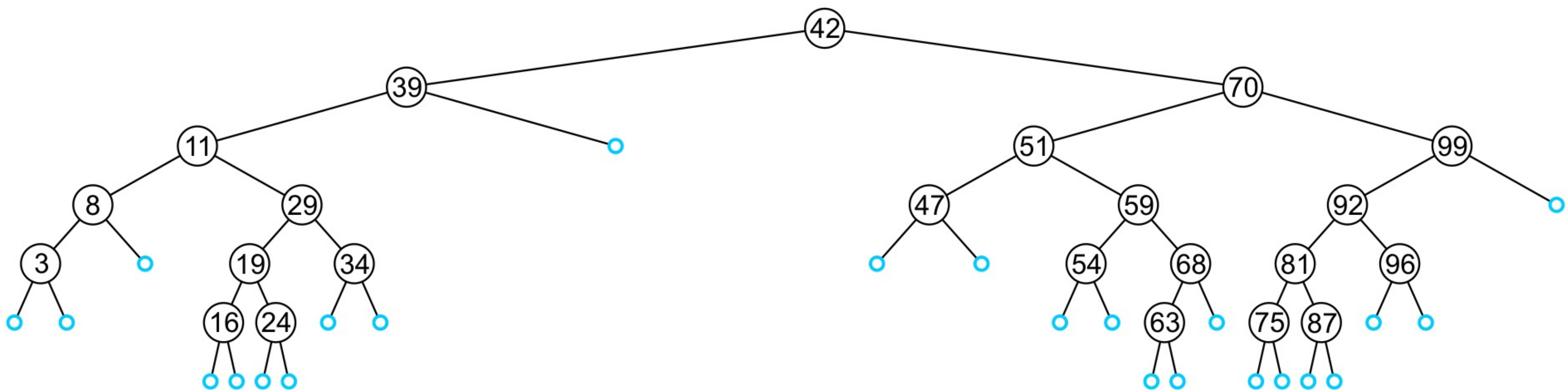
Find

- The implementation is similar to findMin and findMax:
  - The run time is O(h)

```
bool find(const T &x) const { return contains(x, root); }

bool find(const T &x, BinaryNode<T> *t) const {
    if (t == nullptr)
        return false;
    else if (x < t->value)
        return find(x, t->left);
    else if (t->value < x)
        return find(x, t->right);
    else
        return true; // Match
}
```

Insert

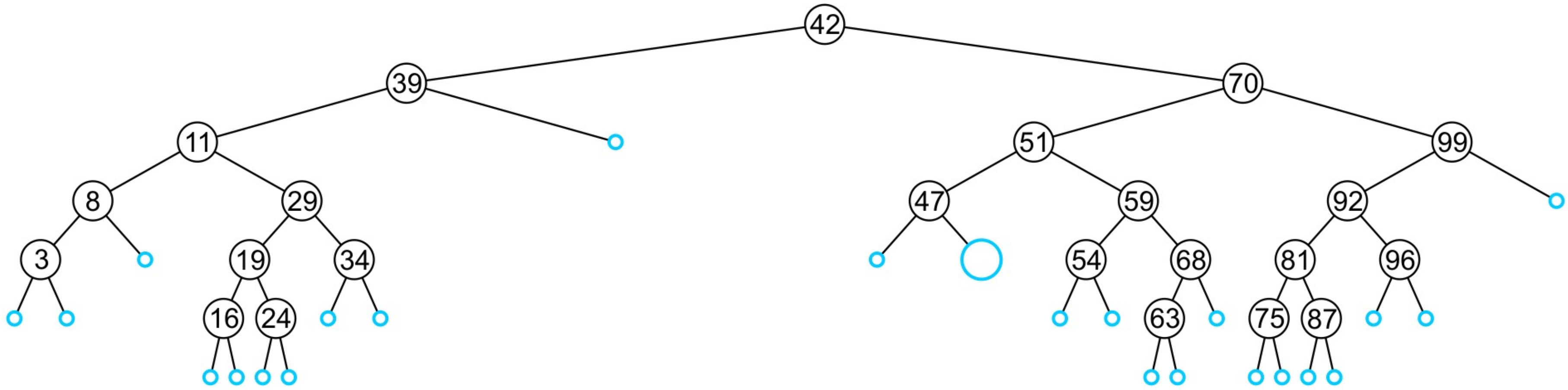
- An insertion will be performed at an empty node:
  - Any empty node is a possible location for an insertion



- The values which may be inserted at any empty node depend on the surrounding nodes

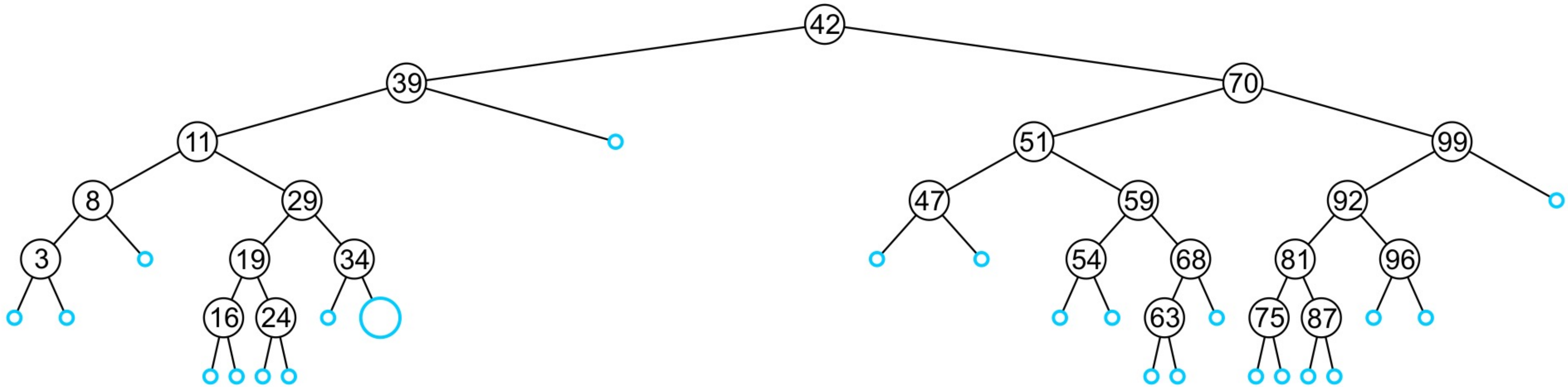
Insert

- For example, this node may hold 48, 49, or 50



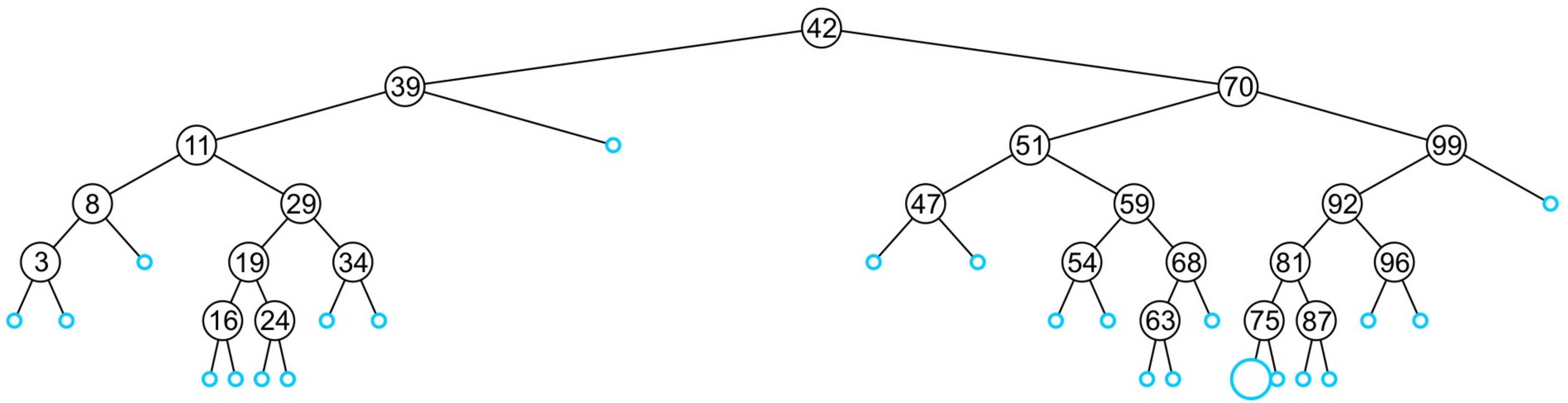
Insert

- An insertion at this location must be 35, 36, 37, or 38



Insert

- This empty node may hold values from 71 to 74

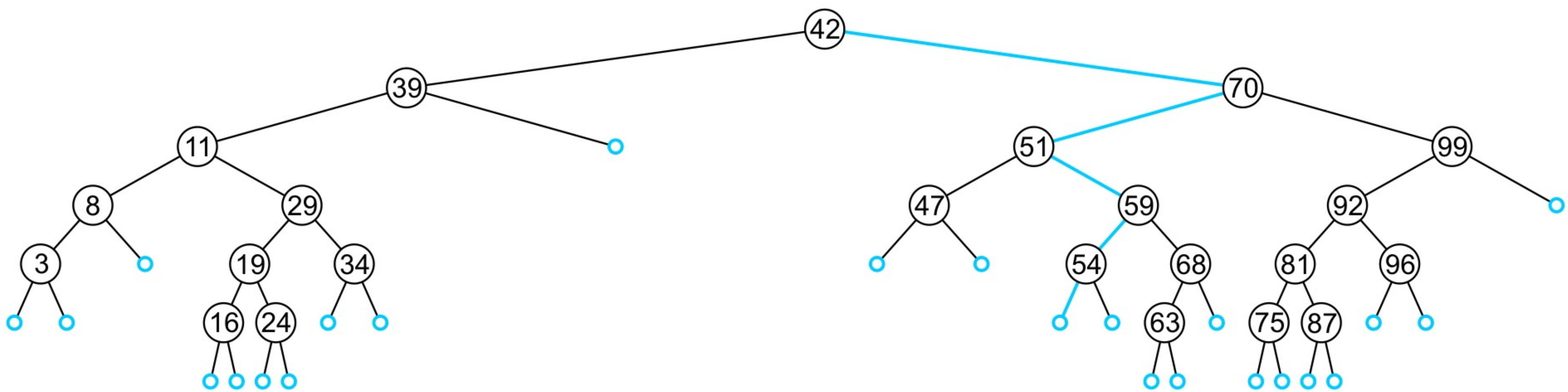


Insert

- Like find, we will step through the tree
  - If we find the object already in the tree, we will return
    - The object is already in the binary search tree (no duplicates)
  - Otherwise, we will arrive at an empty node
  - The object will be inserted into that location
  - The run time is $O(h)$

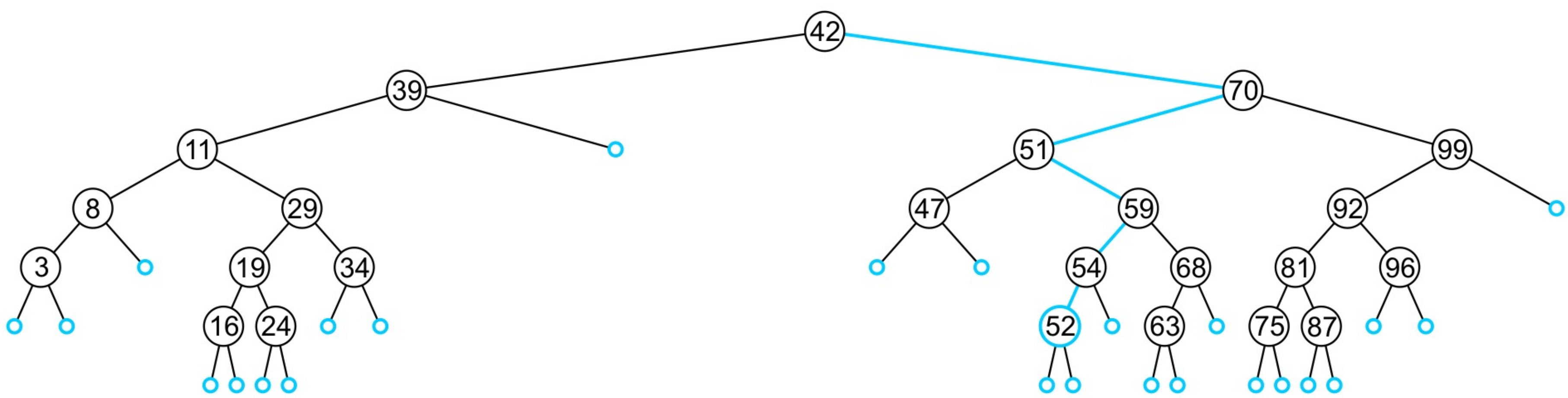
Insert 52

- In inserting the value 52, we traverse the tree until we reach an empty node
  - The `left` sub-tree of 54 is an empty node



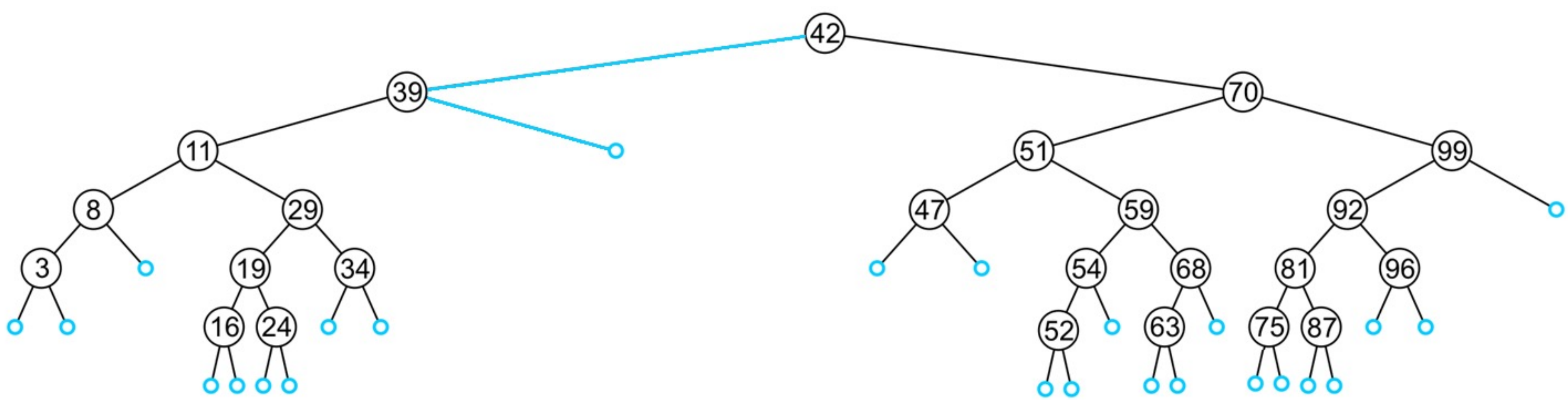
Insert 52

- A new leaf node is created and assigned to the member variable `left`



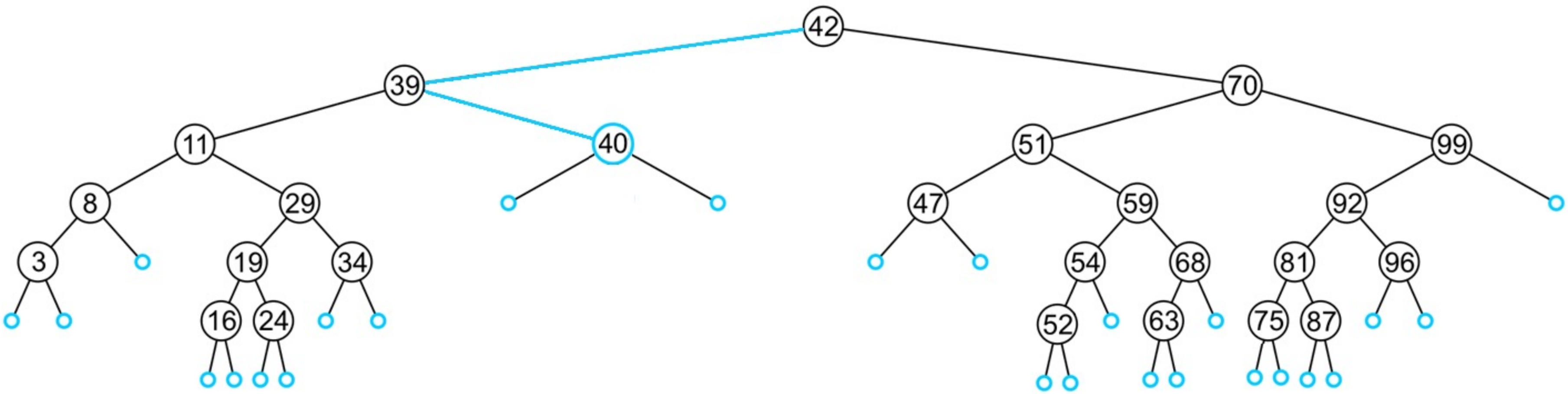
Insert 40

- In inserting 40, we determine the `right` sub-tree of 39 is an empty node



Insert 40

- A new leaf node storing 40 is created and assigned to the member variable `right`



Insert

```
    void insert(const T &x) { insert(x, root); }

    void insert(const T &x, BinaryNode<T> *&t) {
        if (t == nullptr)
            t = new BinaryNode<T>{x, nullptr, nullptr};
        else if (x < t->value)
            insert(x, t->left);
        else if (t->value < x)
            insert(x, t->right);
        else
            ; // Duplicate; do nothing
    }
```

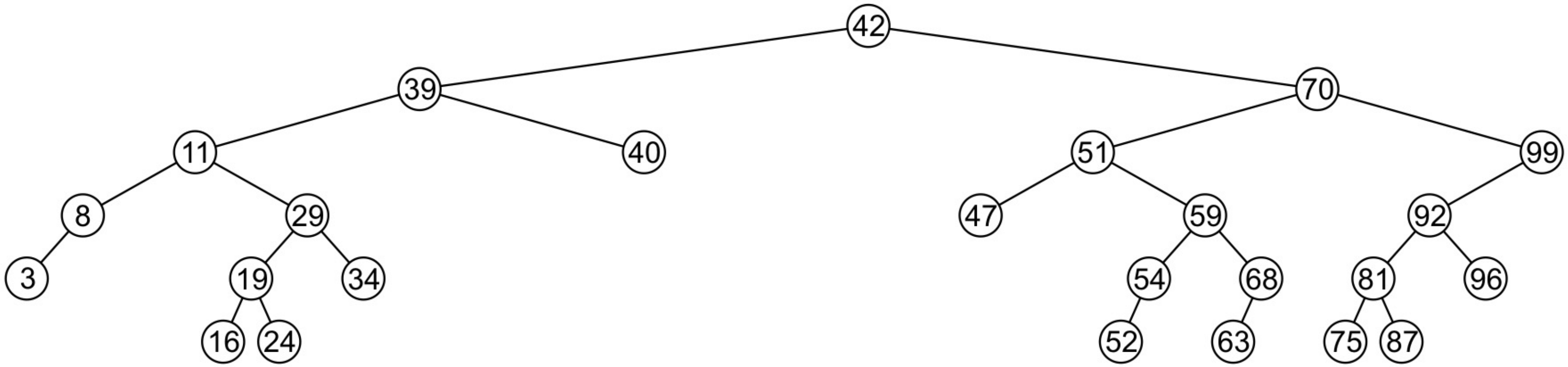
Insert

- Example questions:
  - In the given order, insert these objects into an initially empty binary search tree:

    31 45 36 14 52 42 6 21 73 47 26 37 33 8
  - What values could be placed:
    - To the left of 21?
    - To the right of 26?
    - To the left of 47?
  - How would we determine if 40 is in this binary search tree?
  - Which values could be inserted to increase the height of the tree?

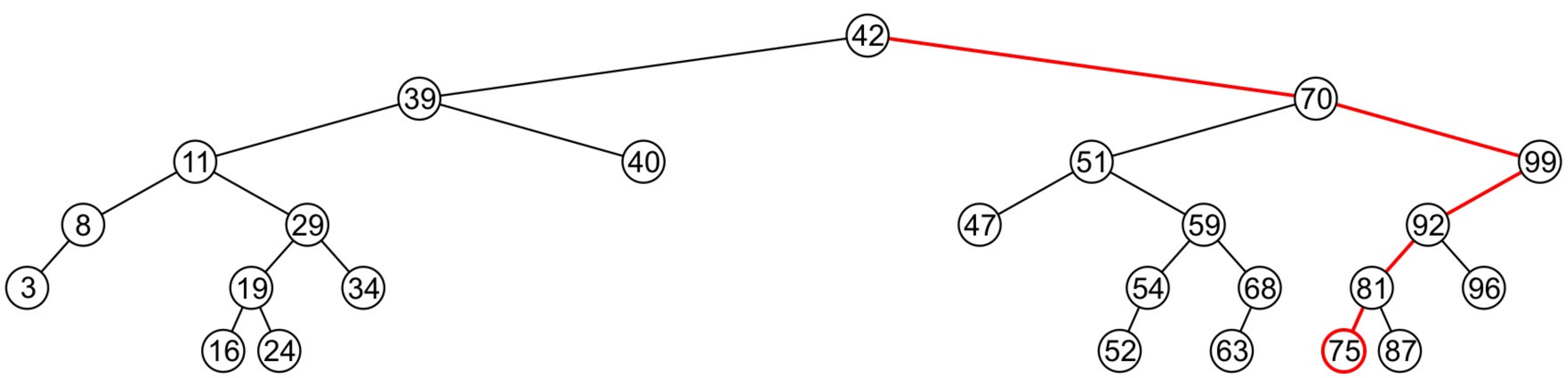
Erase

- A node being erased is not always going to be a leaf node
- There are three possible scenarios:
  - 1) The node is a leaf node,
  - 2) It has exactly one child, or
  - 3) It has two children (it is a full node)



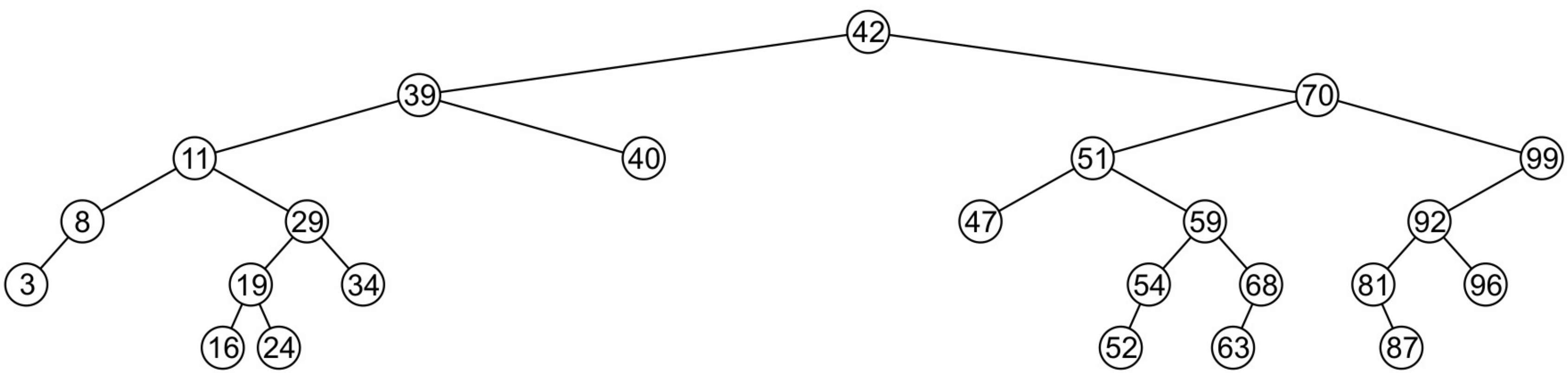
Erase: Leaf Node

- A leaf node must be removed and the appropriate member variable of the parent is set to **`nullptr`**
  - Consider removing 75



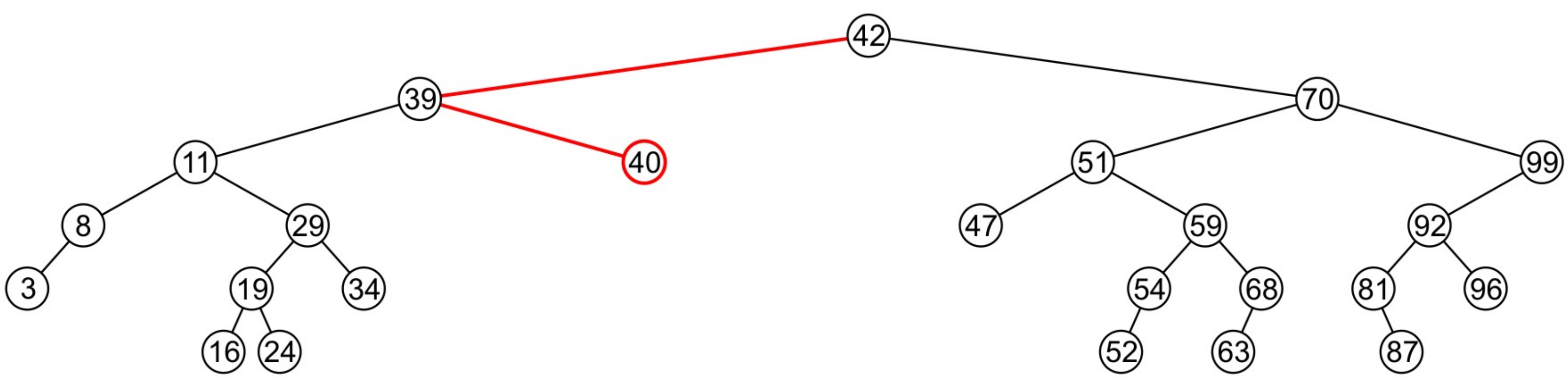
Erase: Leaf Node

- The node is deleted and `left` of 81 is set to `nullptr`



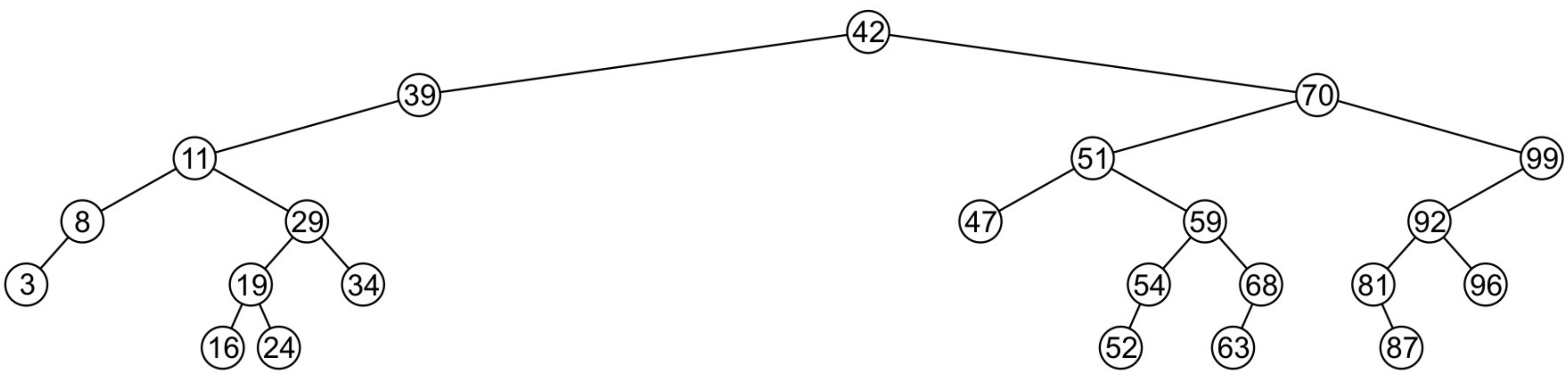
Erase: Leaf Node

- Erasing the node containing 40 is similar



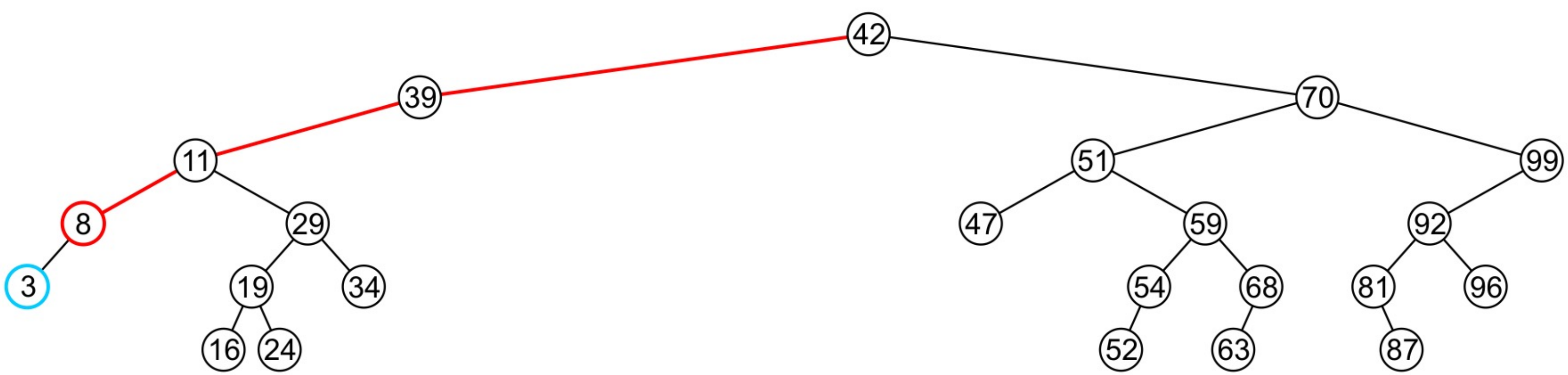
Erase: Leaf Node

- The node is deleted and `right` of 39 is set to `nullptr`



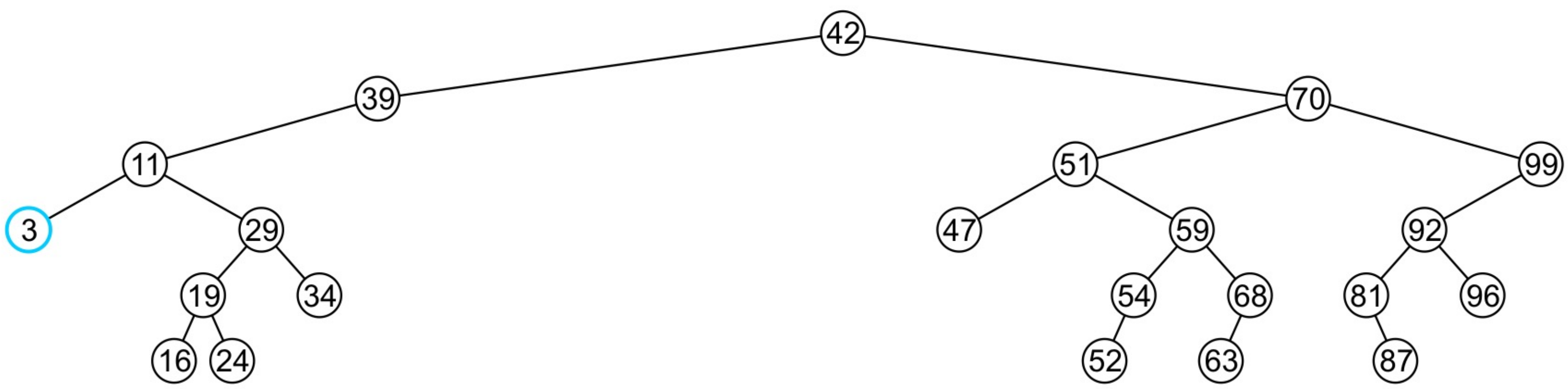
Erase: Non-leaf node w/ one child

- If a node has only one child, we can simply promote the sub-tree associated with the child
  - Consider removing 8 which has one left child



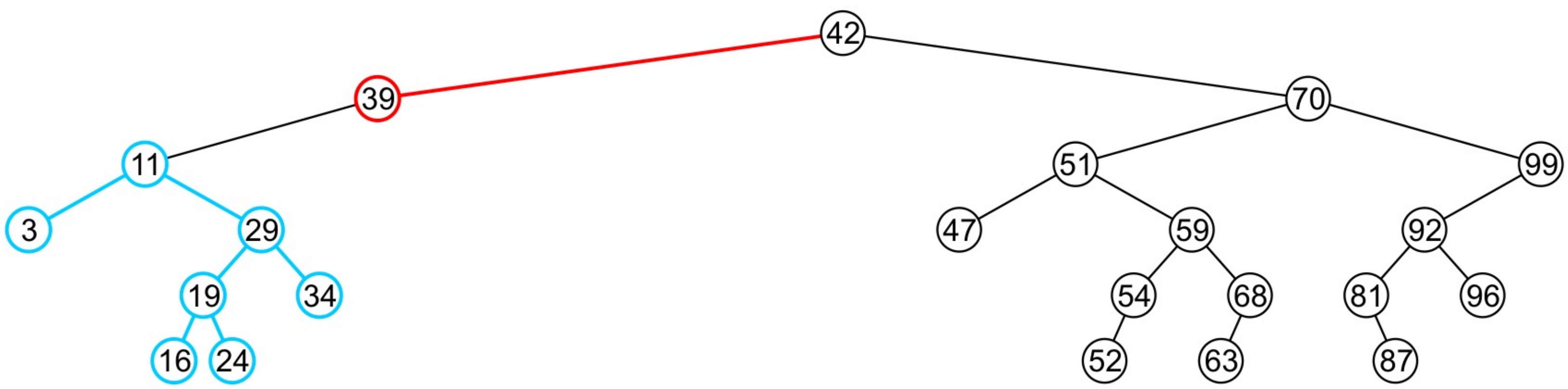
Erase: Non-leaf node w/ one child

- The node 8 is deleted and the `left` of 11 is updated to point to 3



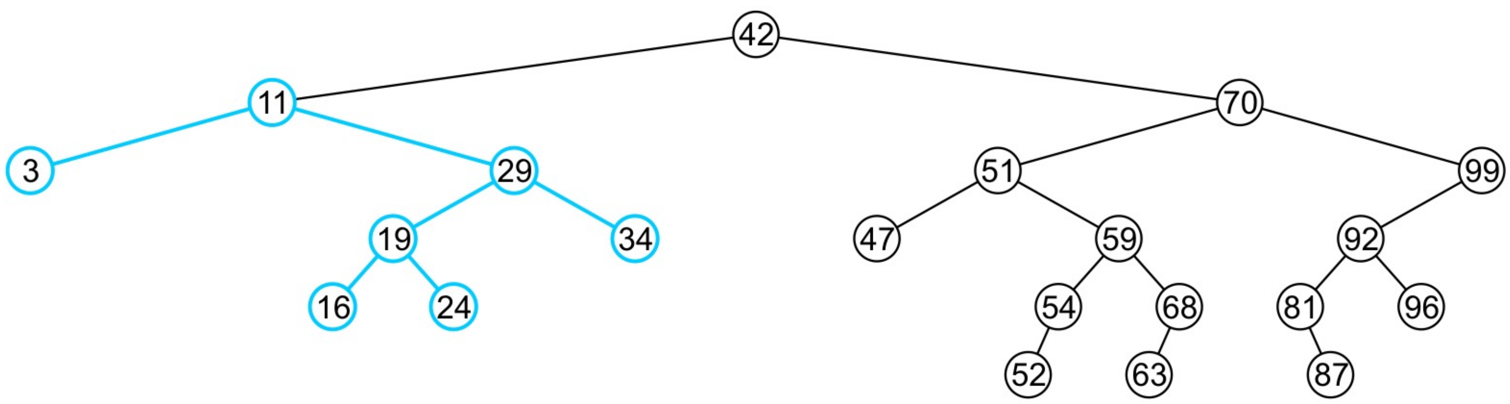
Erase: Non-leaf node w/ one child

- There is no difference in promoting a single node or a sub-tree
  - To remove 39, it has a single child 11



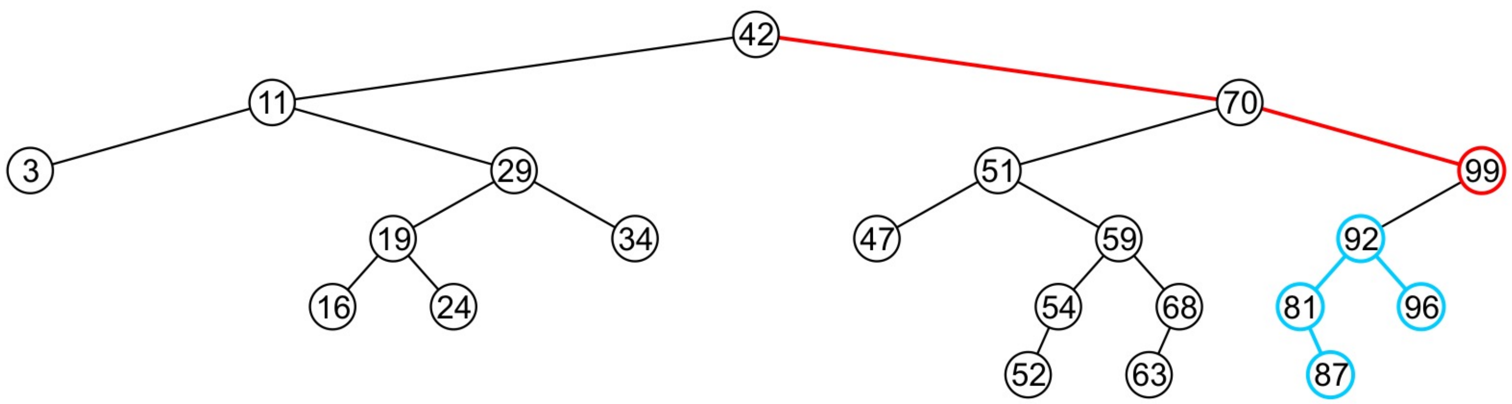
Erase: Non-leaf node w/ one child

- The node containing 39 is deleted and `left` of 42 is updated to point to 11
  - Notice that order is still maintained



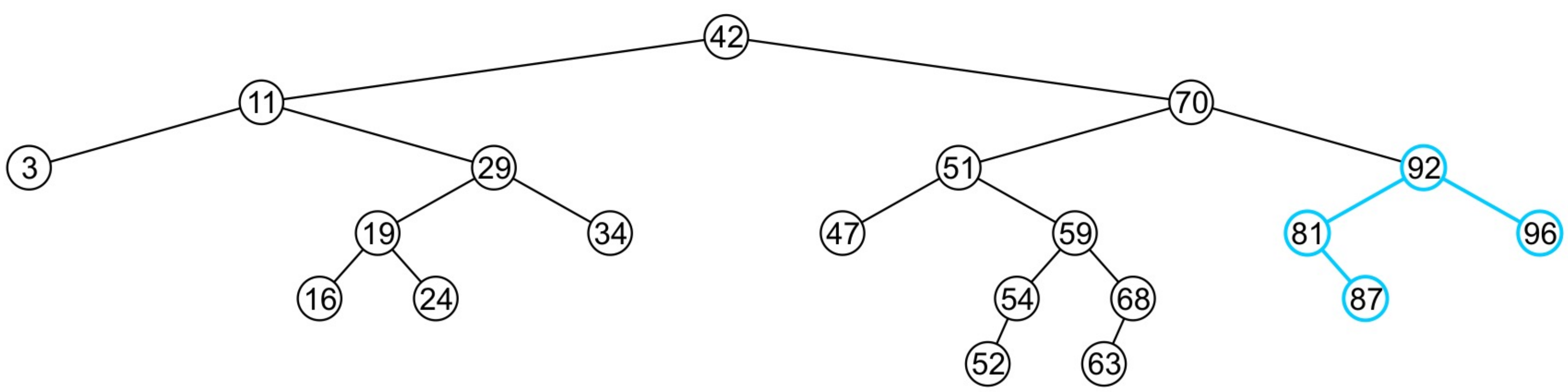
Erase: Non-leaf node w/ one child

- Consider erasing the node containing 99



Erase: Non-leaf node w/ one child

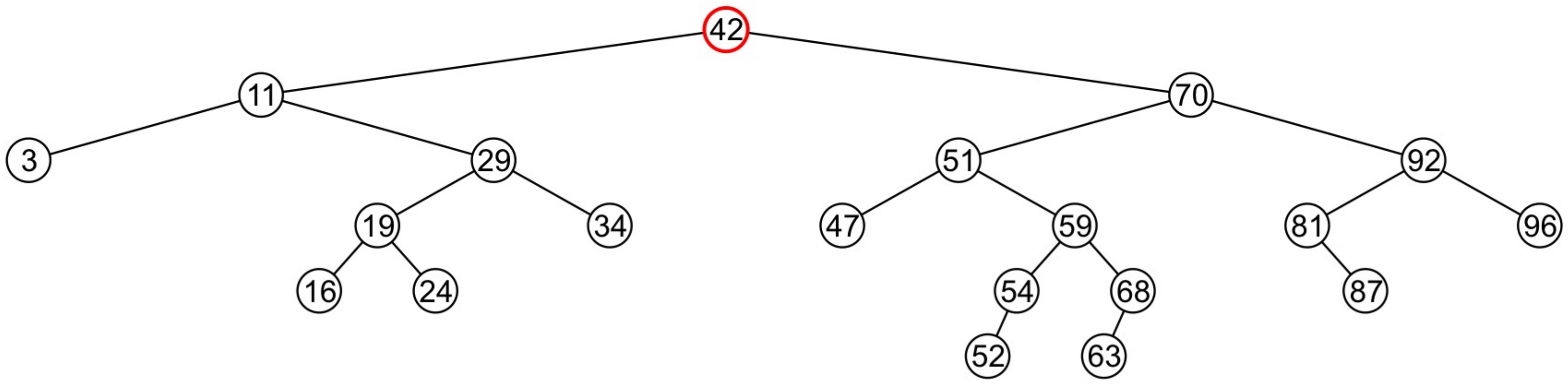
- The node is deleted and the **`left`** sub-tree is promoted:
  - The member variable **`right`** of 70 is set to point to 92
  - Again, the order of the tree is maintained



Erase: Full node

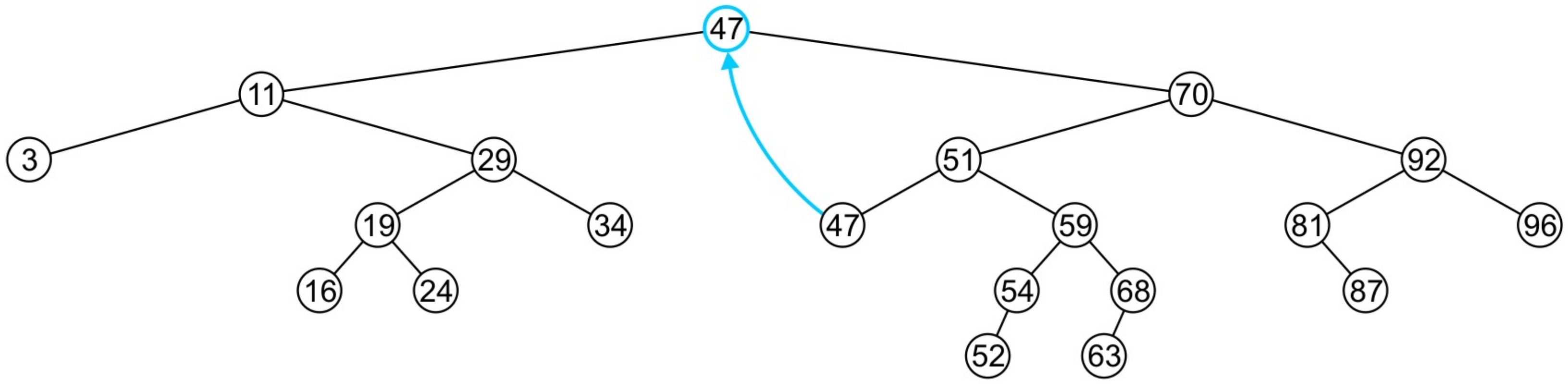
- Finally, we will consider the problem of erasing a full node, *e.g.*, 42

- We will perform two operations:
  - Replace 42 with the minimum object in the `right` sub-tree
  - Erase that object from the `right` sub-tree



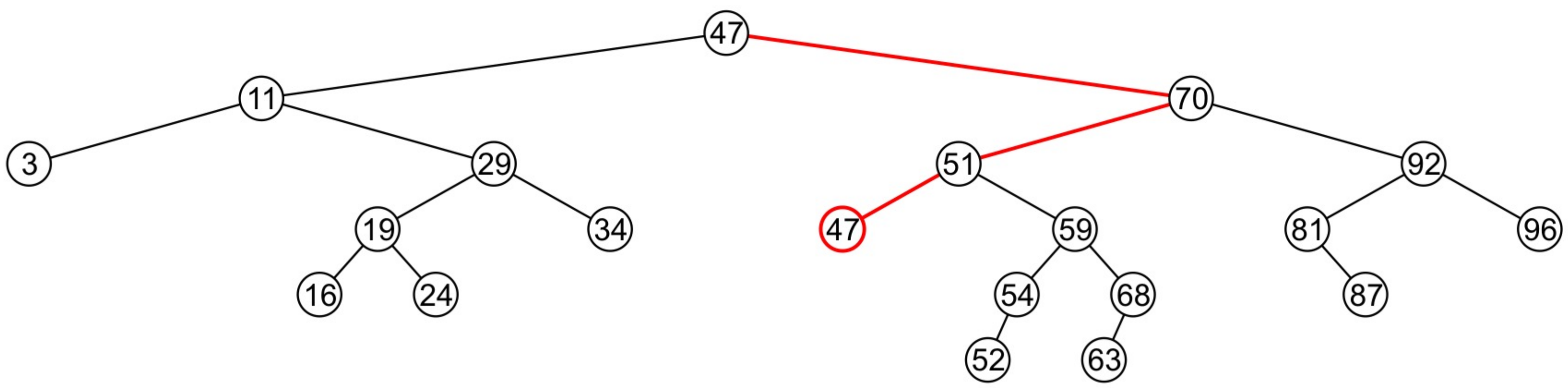
Erase: Full node

- In this case, we replace 42 with 47
  - We temporarily have two copies of 47 in the tree



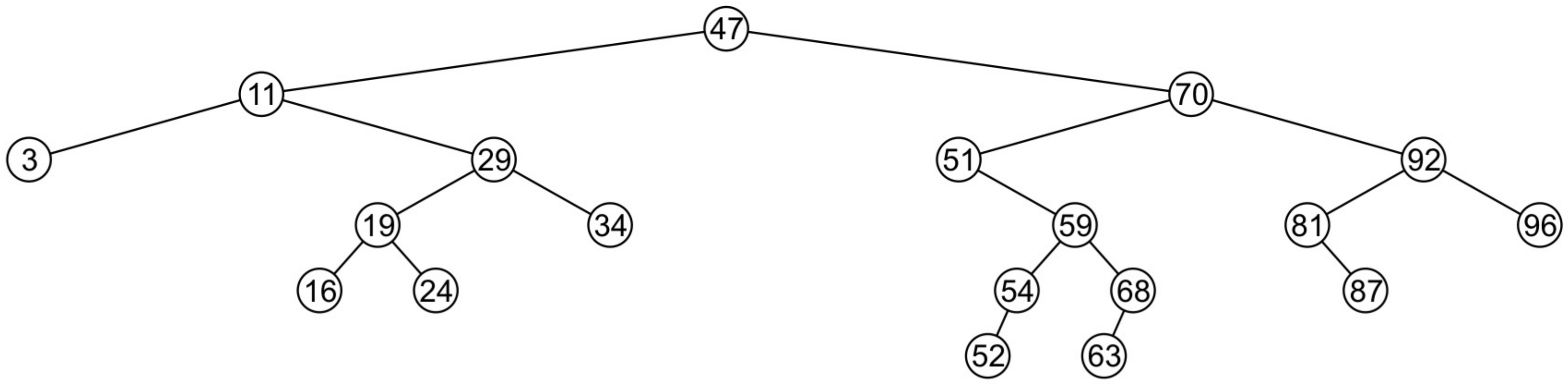
Erase: Full node

- We now recursively erase 47 from the `right` sub-tree
  - We note that 47 is a leaf node in the right sub-tree



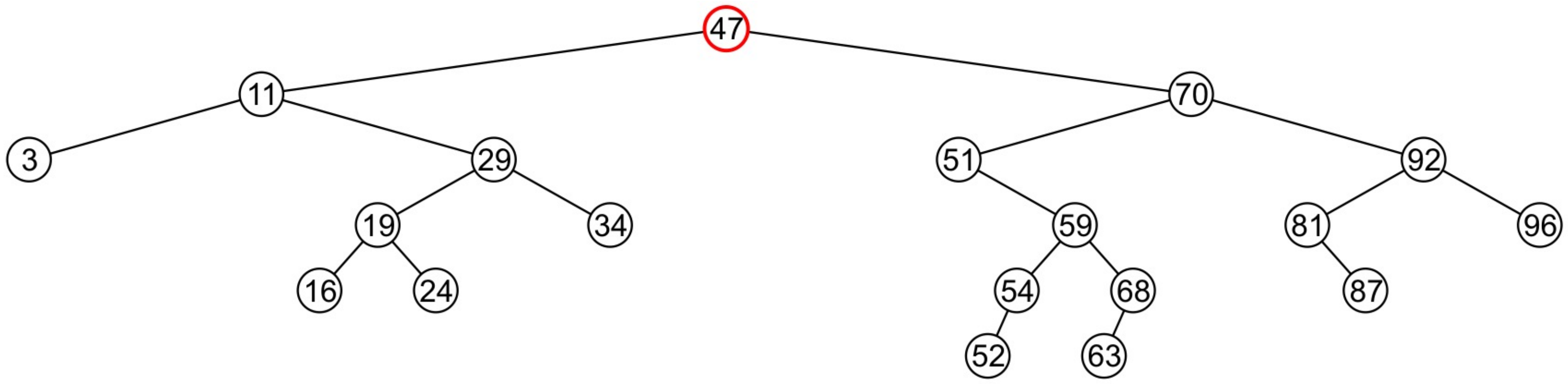
Erase: Full node

- Leaf nodes are simply removed and `left` of 51 is set to `nullptr`
  - Notice that the tree is still sorted



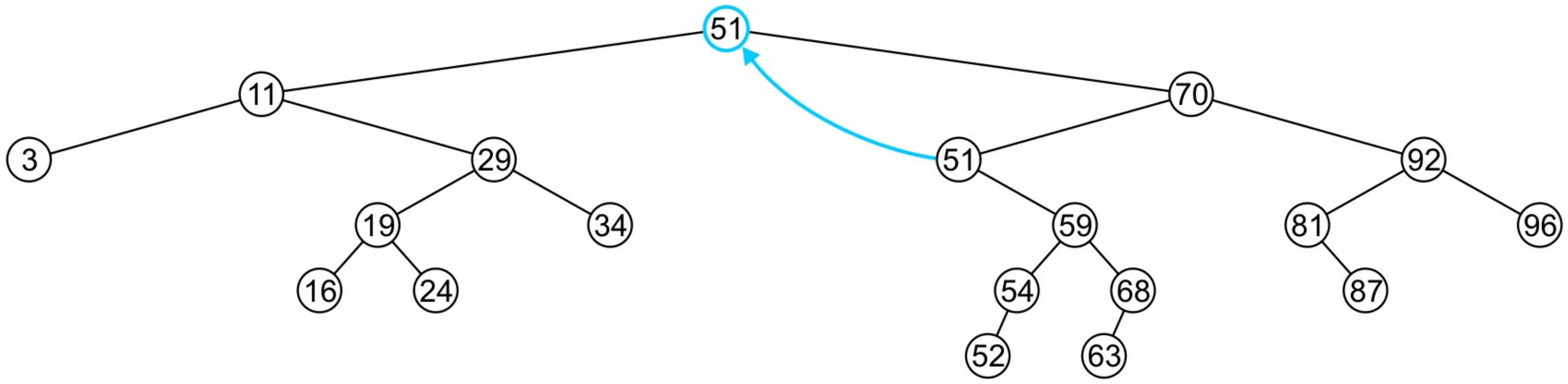
Erase: Full node

- Suppose we want to erase the root 47 again:
  - We must copy the minimum of the right sub-tree
  - We could promote the maximum object in the left sub-tree and achieve similar results



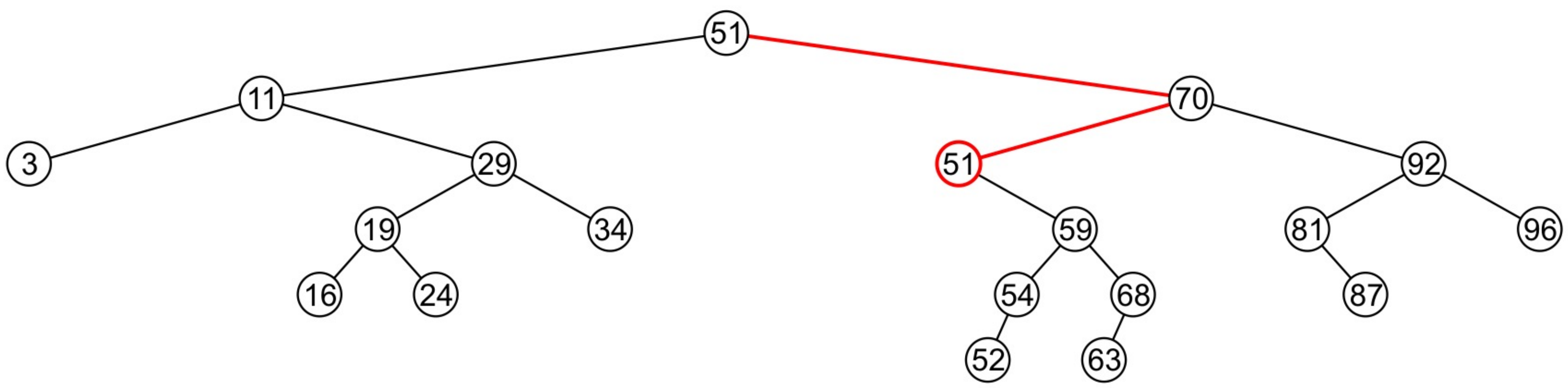
Erase: Full node

- We copy 51 from the right sub-tree



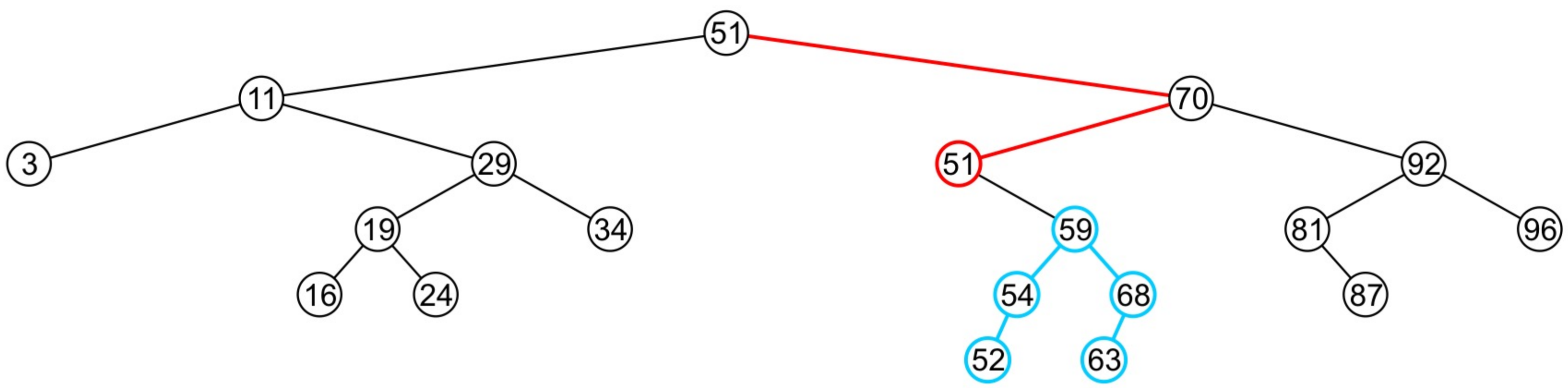
Erase: Full node

- We must proceed by delete 51 from the right sub-tree



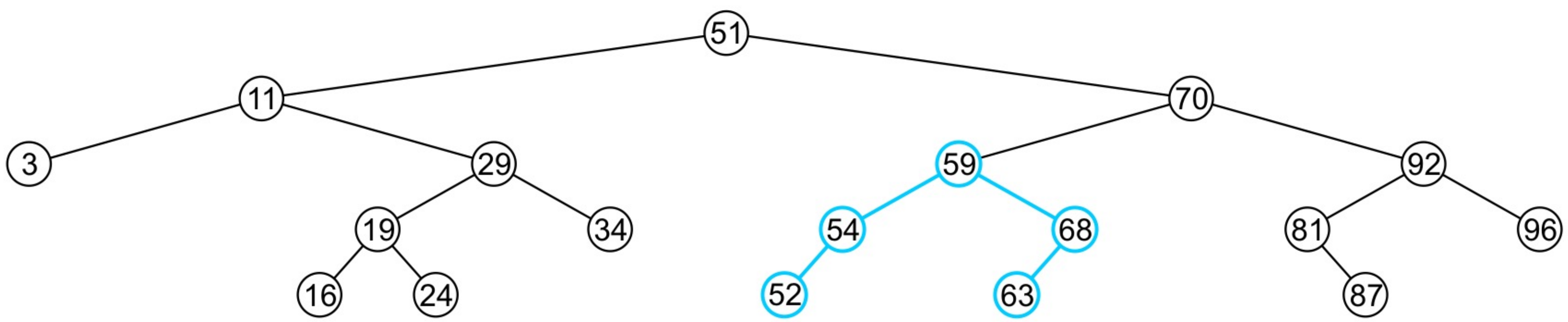
Erase: Full node

- In this case, the node storing 51 has just a single child



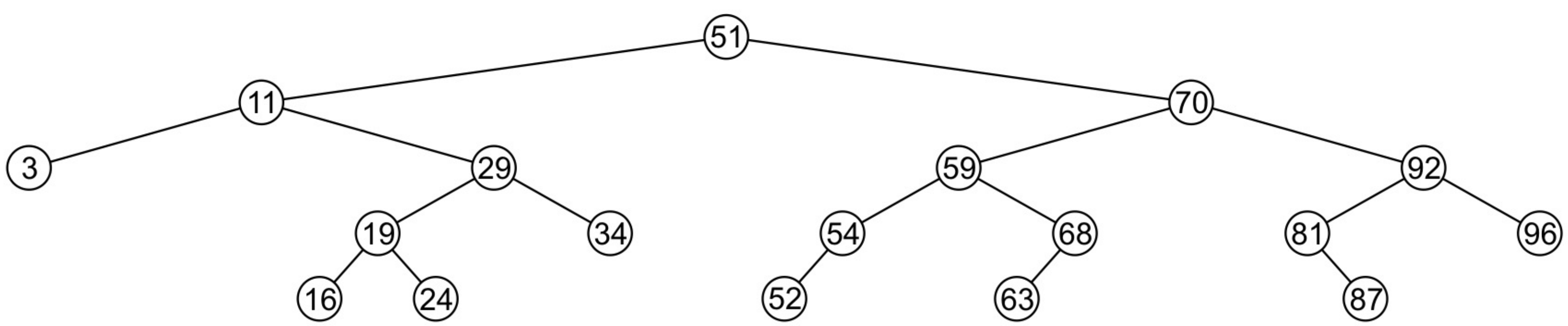
Erase: Full node

- We delete the node containing 51 and assign the member variable `left` of 70 to point to 59



Erase: Full node

- Note that after several removals, the remaining tree is still correctly sorted



Erase: Full node

```
void remove(const T &x) { remove(x, root); }

void remove(const T &x, BinaryNode<T> *&t) {
    if (t == nullptr)
        return; // Item not found; do nothing
    if (x < t->value)
        remove(x, t->left);
    else if (t->value < x)
        remove(x, t->right);
    else if (t->left != nullptr && t->right != nullptr) { // two children
            t->value = findMin(t->right)->value;
            remove(t->value, t->right);
    }
    else { // single child
        BinaryNode<T> *oldNode = t;
        t = (t->left != nullptr) ? t->left : t->right;
        delete oldNode;
    }
}
```

Other Relation-based Operations

- We will quickly consider two other relation-based queries that are very quick to calculate with an array of sorted objects:
  - Finding the previous and next values, and
  - Finding the k^{th} value

Previous and Next Objects

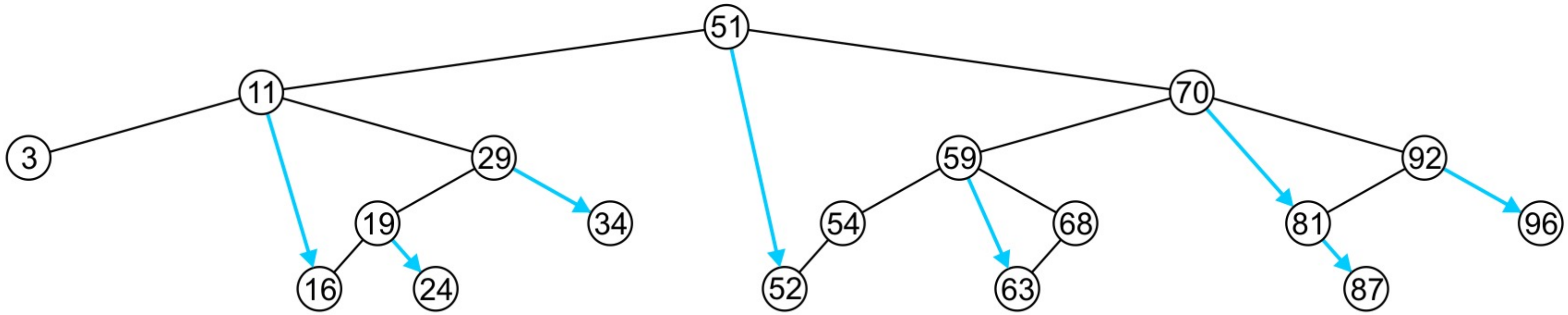
- All the operations up to now have been operations which work on any container: count, insert, *etc.*
  - If these are the only relevant operations, use a *hash table*

- Operations specific to linearly ordered data include:
  - Find the next larger and previous smaller objects of a given object which may or may not be in the container
  - Find the k^{th} value of the container
  - Iterate through those objects that fall on an interval [*a*, *b*]

- We will focus on finding the next largest object
  - The others will follow

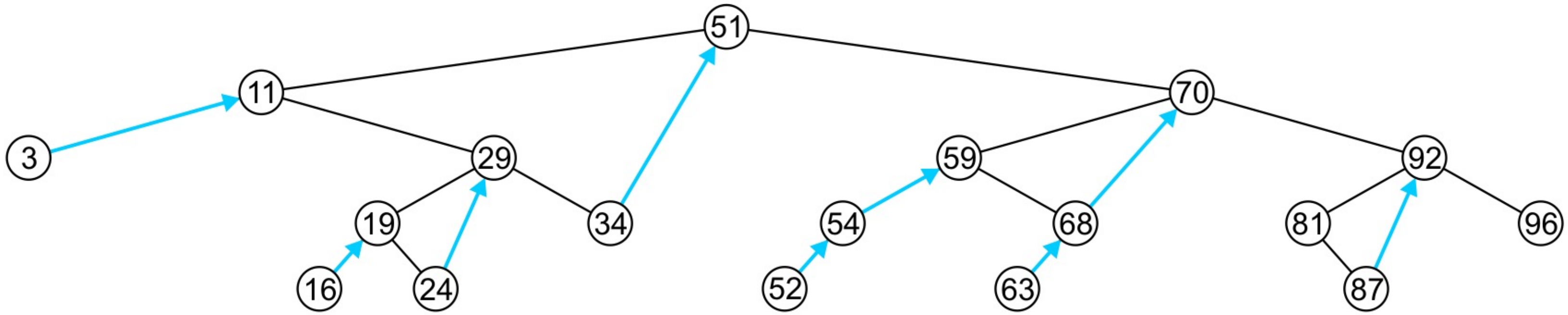
Previous and Next Objects

- To find the next largest object:
  - If the node has a right sub-tree, the minimum object in that sub-tree is the next-largest object



Previous and Next Objects

- If, however, there is no right sub-tree:
  - It is the next largest object (if any) that exists in the path from the root to the node

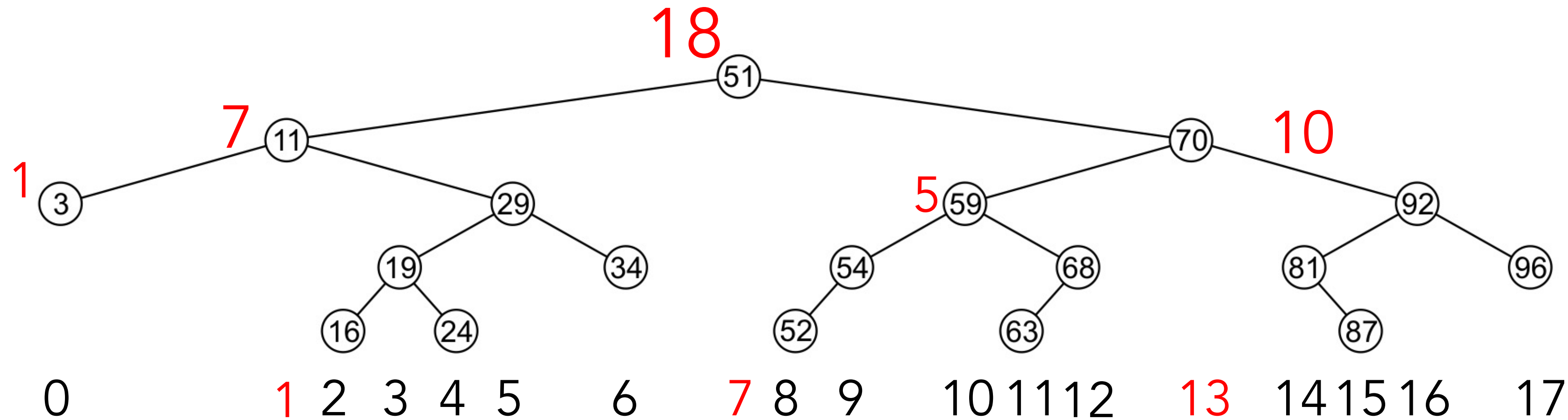


Previous and Next Objects

- More generally: what is the next largest value of an arbitrary object?
  - This can be found with a single search from the root node to one of the leaves — an O(*h*) operation
  - This function returns the object if it did not find something greater than it

Finding the k^{th} Object

- Another operation on sorted lists may be finding the k^{th} largest object
  - Recall that k goes from 0 to $n - 1$
  - If the left-sub-tree has $\ell = k$ values, return the current node,
  - If the left sub-tree has $\ell > k$ values, return the k^{th} value of the left sub-tree,
  - Otherwise, the left sub-tree has $\ell < k$ values, so return the $(k - \ell - 1)^{th}$ value of the right sub-tree



Run Time: O(h)

- Almost all of the relevant operations on a binary search tree are O(h)
  - If the tree is *close* to a linked list, the run times is O(n)
    - Insert 1, 2, 3, 4, 5, 6, 7, ..., n into a empty binary search tree
  - The best we can do is if the tree is perfect: O(ln(n))
  - Our goal will be to find tree structures where we can maintain a height of Θ(ln(n))

- We will look at
  - AVL trees
  - B+ trees

  both of which ensure that the height remains Θ(ln(n))

Summary

- In this topic, we covered binary search trees
  - Described Abstract Sorted Lists
  - Problems using arrays and linked lists
  - Definition a binary search tree
  - Looked at the implementation of:
    - Empty, size, height, count
    - FindMin, FindMax, insert, erase
    - Previous smaller and next larger objects

AVL Trees

Kyunghan Lee
Networked Computing Lab (NXC Lab)
Department of Electrical and Computer Engineering
Seoul National University
https://nxc.snu.ac.kr
kyunghanlee@snu.ac.kr

Outline

- Background
- Define height balancing
- Maintaining balance within a tree
  - AVL trees
  - Difference of heights
  - Maintaining balance after insertions and erases
  - Can we store AVL trees as arrays?

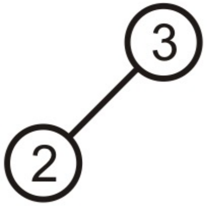
Background

- From previous lectures:
  - Binary search trees store linearly ordered data
  - Best case height: $\Theta(\ln(n))$
  - Worst case height: $O(n)$

- Requirement:
  - Define and maintain a *balance* to ensure $\Theta(\ln(n))$ height

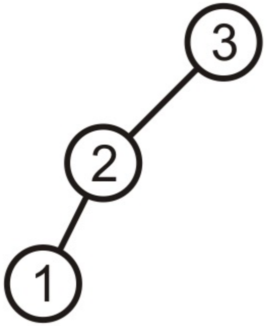
Prototypical Examples

- These two examples demonstrate how we can correct for imbalances: starting with this tree, add 1:



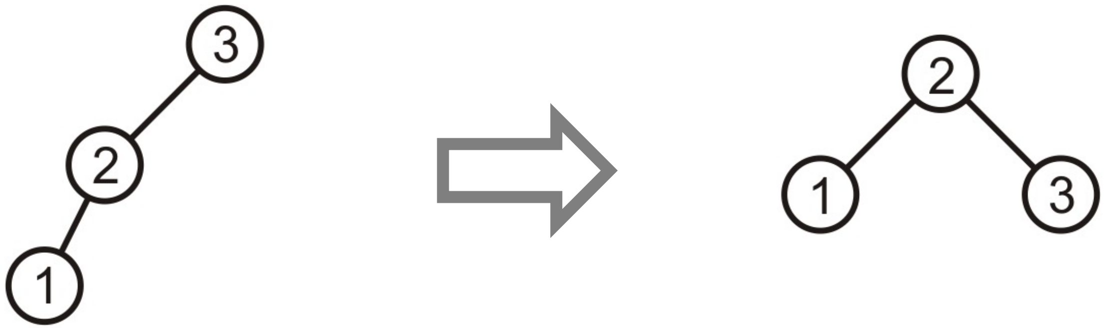
Prototypical Examples

- This is more like a linked list; however, we can fix this…



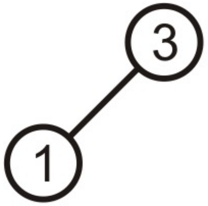
Prototypical Examples

- Promote 2 to the root, demote 3 to be 2's right child, and 1 remains the left child of 2.
- The result is a perfect, though trivial tree



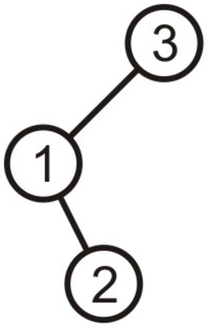
Prototypical Examples

- Alternatively, given this tree, insert 2



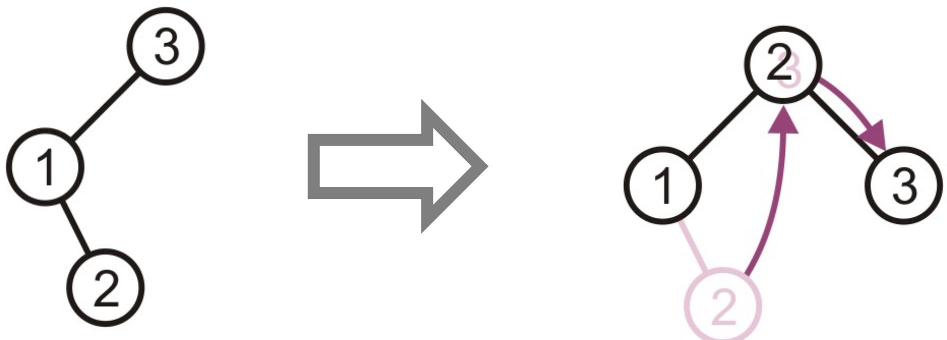
Prototypical Examples

- Again, the product is a linked list; however, we can fix this, too



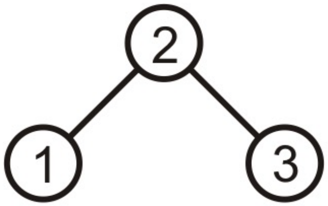
Prototypical Examples

- Promote 2 to the root, and assign 1 and 3 to be its children



Prototypical Examples

- The result is, again, a perfect tree



- These examples may seem trivial, but they are the basis for the corrections in the next data structure we will see: AVL trees

AVL Trees

- We will focus on the first strategy: AVL trees
  - Named after Adelson-Velsky and Landis

- Balance is defined by comparing the height of the two sub-trees

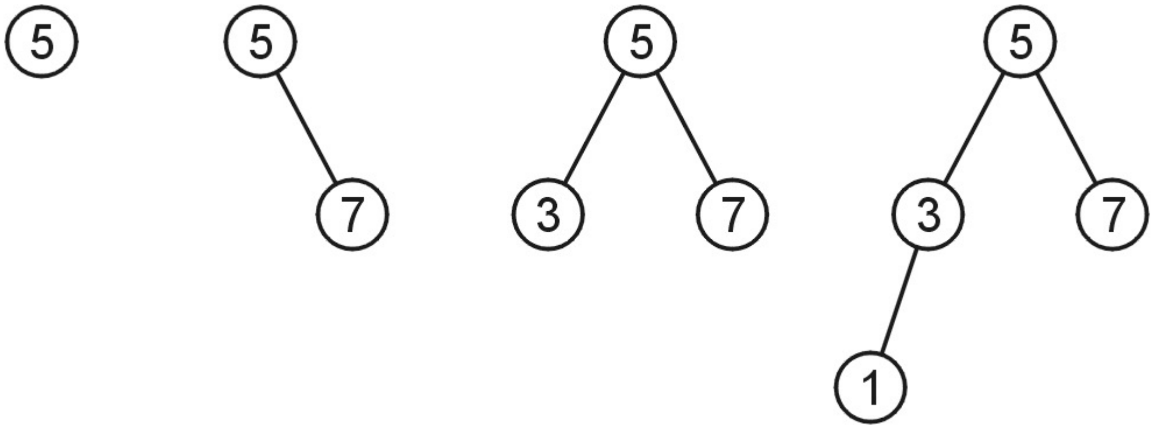
- Recall:
  - An empty tree has height –1
  - A tree with a single node has height 0

AVL Trees

- A binary search tree is said to be AVL balanced if:
  - The difference in the heights between the left and right sub-trees is at most 1, and
  - Both sub-trees are themselves AVL trees

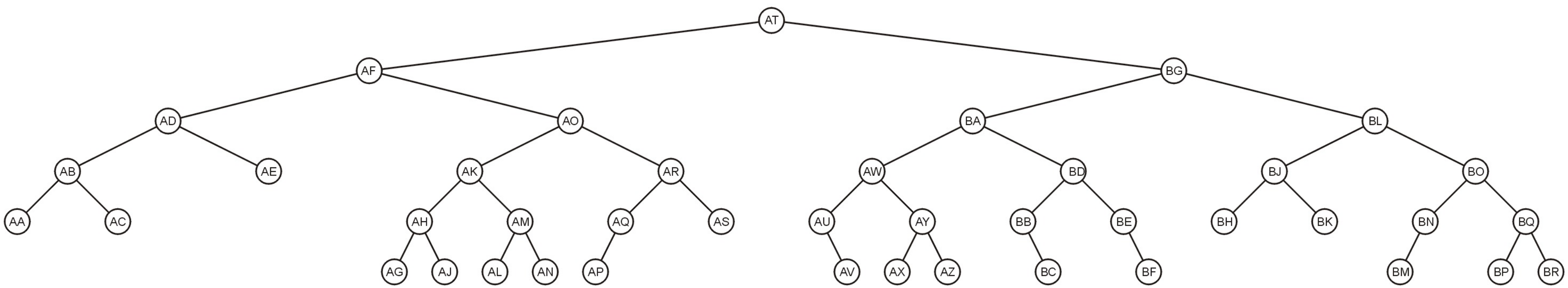
AVL Trees

- AVL trees with 1, 2, 3, and 4 nodes:



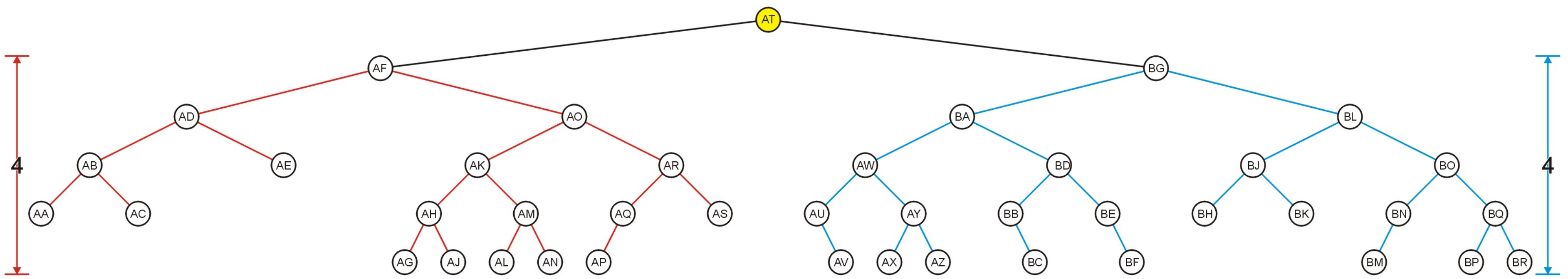
AVL Trees

- Here is a larger AVL tree (42 nodes):



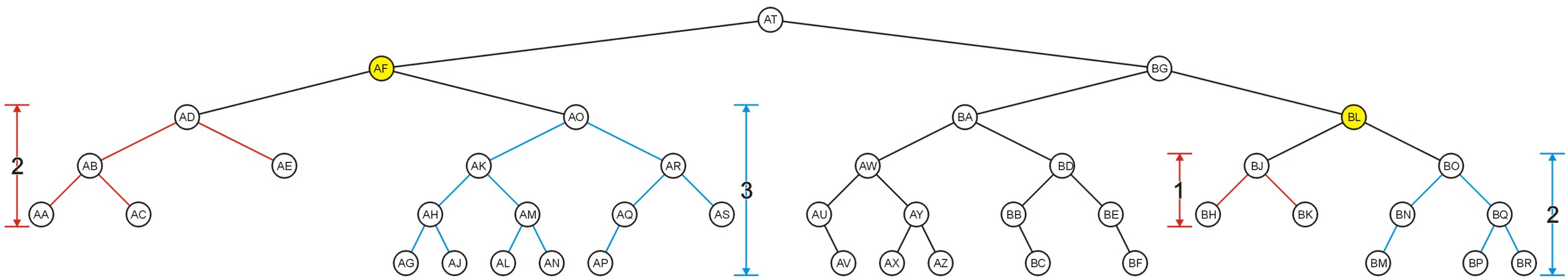
AVL Trees

- The root node is AVL-balanced:
  - Both sub-trees are of height 4:



AVL Trees

- All other nodes (*e.g.*, AF and BL) are AVL balanced
  - The sub-trees differ in height by at most one



Height of an AVL Tree

- By the definition of complete trees, any complete binary search tree is an AVL tree

- Thus an upper bound on the number of nodes in an AVL tree of height h is a perfect binary tree with $2^{h+1} - 1$ nodes
  - What is an lower bound?
  - This will be the worst case of an AVL tree

Height of an AVL Tree

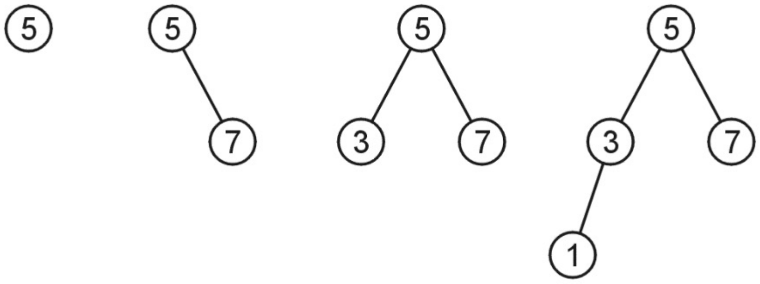
- Let F(h) be the fewest number of nodes in a tree of height h

- From a previous slide:

  F(0) = 1

  F(1) = 2

  F(2) = 4



- Can we find F(h)?

Height of an AVL Tree

- The worst-case AVL tree of height h would have:
  - A worst-case AVL tree of height $h - 1$ on one side,
  - A worst-case AVL tree of height $h - 2$ on the other, and
  - The root node

- We get: $F(h) = F(h-1) + 1 + F(h-2)$

Height of an AVL Tree

- This is a recurrence relation:

$$\mathrm{F}(h) = \begin{cases} 1 & h = 0 \\ 2 & h = 1 \\ \mathrm{F}(h-1) + \mathrm{F}(h-2) + 1 & h > 1 \end{cases}$$

- The solution?
  - Note that $F(h) + 1 = (F(h-1) + 1) + (F(h-2) + 1)$
  - Therefore, $F(h) + 1$ is a Fibonacci number:

$$\begin{array}{lcl} F(0) + 1 = 2 & \rightarrow & F(0) = 1 \\ F(1) + 1 = 3 & \rightarrow & F(1) = 2 \\ F(2) + 1 = 5 & \rightarrow & F(2) = 4 \\ F(3) + 1 = 8 & \rightarrow & F(3) = 7 \\ F(4) + 1 = 13 & \rightarrow & F(4) = 12 \\ F(5) + 1 = 21 & \rightarrow & F(5) = 20 \\ F(6) + 1 = 34 & \rightarrow & F(6) = 33 \end{array}$$

Height of an AVL Tree

- This is approximately

$$F(h) \approx 1.8944\, \phi^{h} - 1$$

where $\phi \approx 1.6180$ is the golden ratio

  - That is, $F(h) = \boldsymbol{\Omega}(\phi^{h})$
  - Check more: https://en.wikipedia.org/wiki/Golden_ratio#Relationship_to_Fibonacci_sequence

- Thus, we may find the maximum value of h for a given n:

$$\log_\phi\left(\frac{n+1}{1.8944}\right) = \log_\phi(n+1) - 1.3277 = 1.4404 \cdot \lg(n+1) - 1.3277$$

Height of an AVL Tree: Easier Proof?

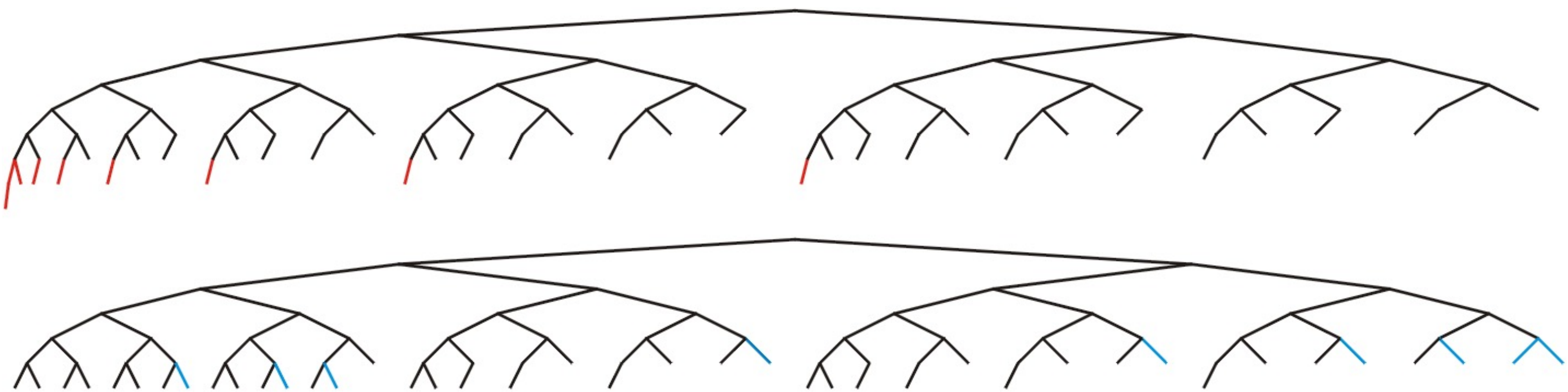
$F(h) > F(h-1) + F(h-2) > 2F(h-2)$

$F(h) > 2 * F(h-2) > 2 * 2 * F(h-4) > \ \dots > 2^{h/2}$

$n > 2^{h/2}$, so $h < 2\log(n)$

Height of an AVL Tree: Looking Good?

- In this example, n = 88, the worst- and best-case scenarios differ in height by only 2



Height of an AVL Tree: Looking Bad?

- If $n = 10^6$, the bounds on h are:
  - (best) The minimum height: $\log_2(10^6) - 1 \approx 19$
  - (worst) The maximum height : $\log_\phi(10^6 / 1.8944) < 28$

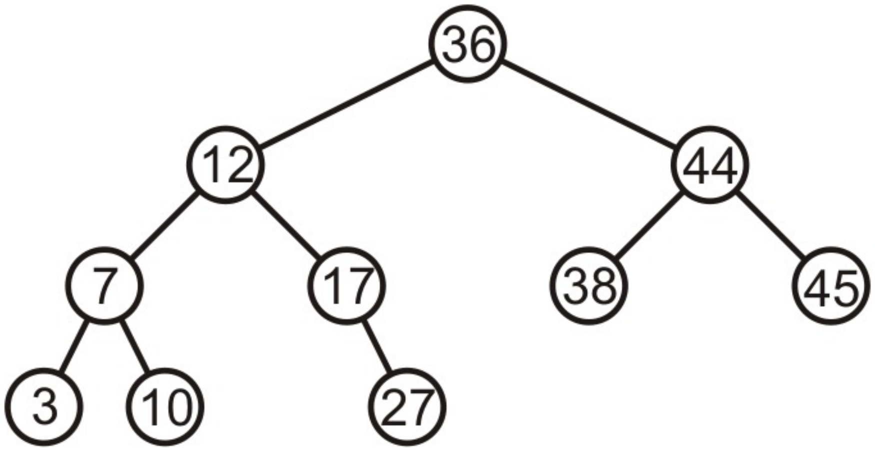
- The AVL Tree ensures the height balance, but such a balanced tree can be quite far from the ideal, perfect binary tree.

Maintaining Balance

- To maintain AVL balance, observe that:
  - Inserting a node can increase the height of a tree by at most 1
  - Removing a node can decrease the height of a tree by at most 1

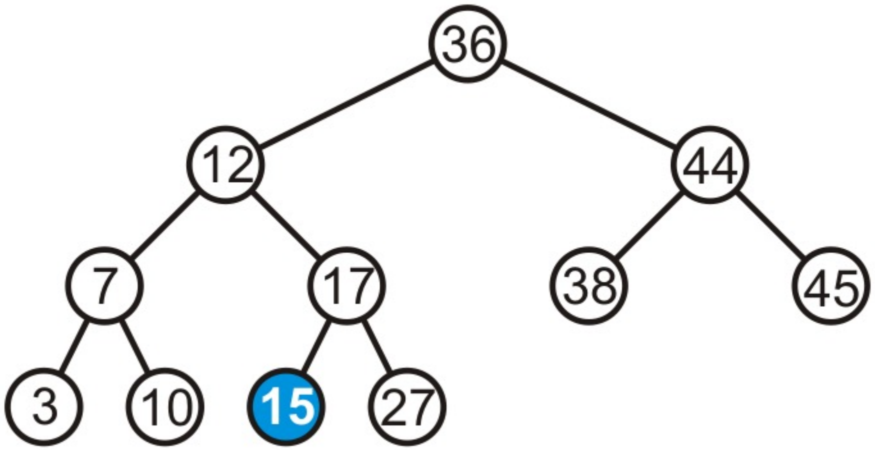
Maintaining Balance

- Consider this AVL tree



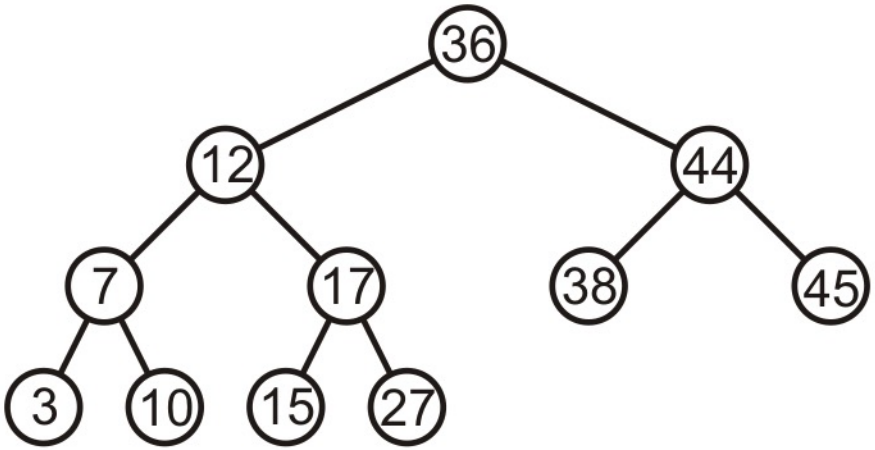
Maintaining Balance: Insert 15

- Consider inserting 15 into this tree
  - In this case, the heights of none of the trees change



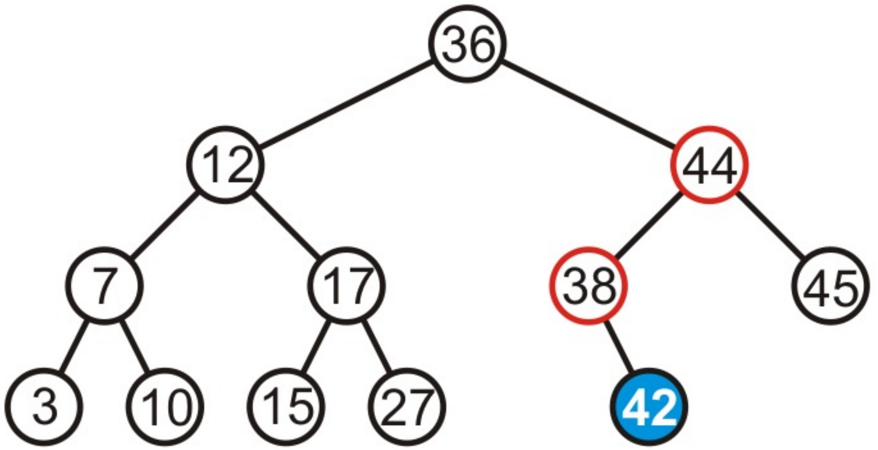
Maintaining Balance: Insert 15

- The tree remains balanced



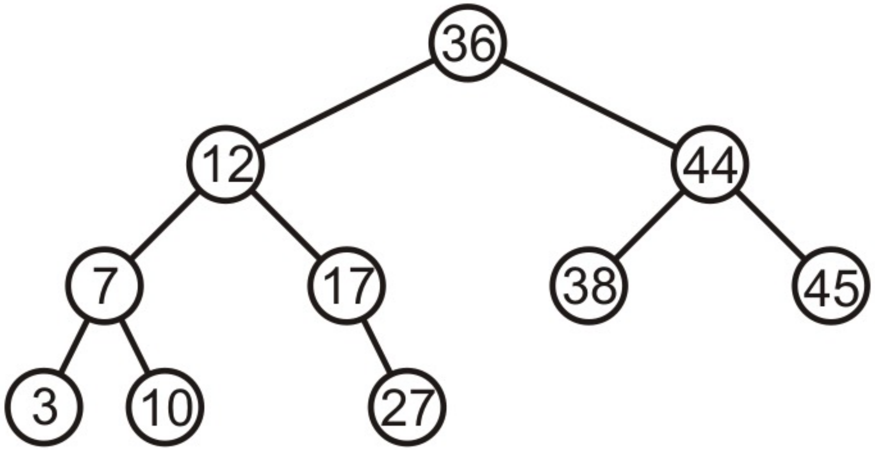
Maintaining Balance: Insert 42

- Consider inserting 42 into this tree
  - Now we see the heights of two sub-trees have increased by one
  - The tree is still balanced



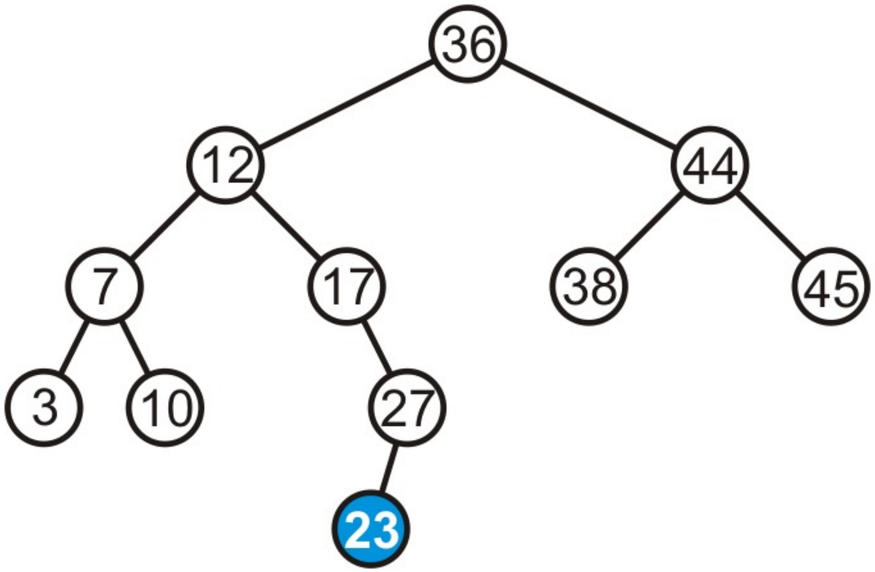
Maintaining Balance

- If a tree is AVL balanced, for an insertion to cause an imbalance:
  - The heights of the sub-trees must differ by 1
  - The insertion must increase the height of the deeper sub-tree by 1



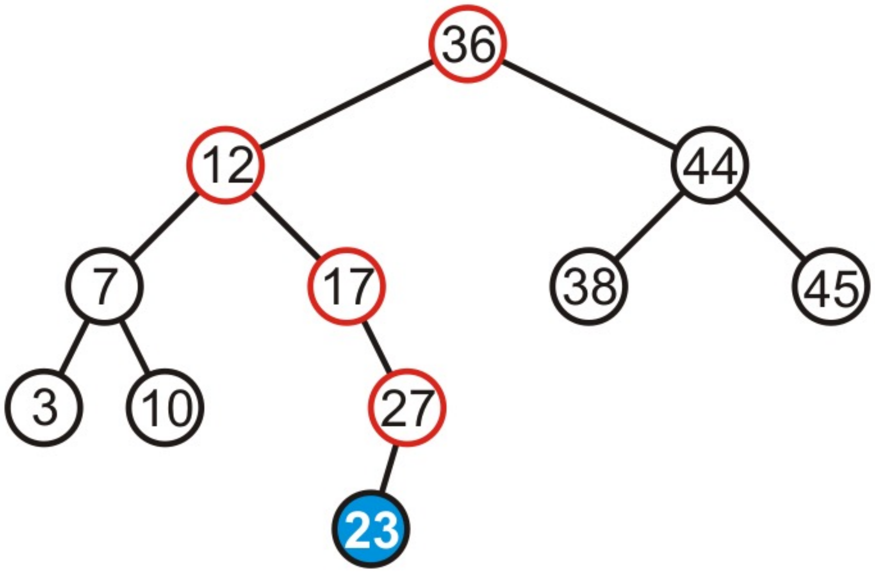
Maintaining Balance

- Suppose we insert 23 into our initial tree



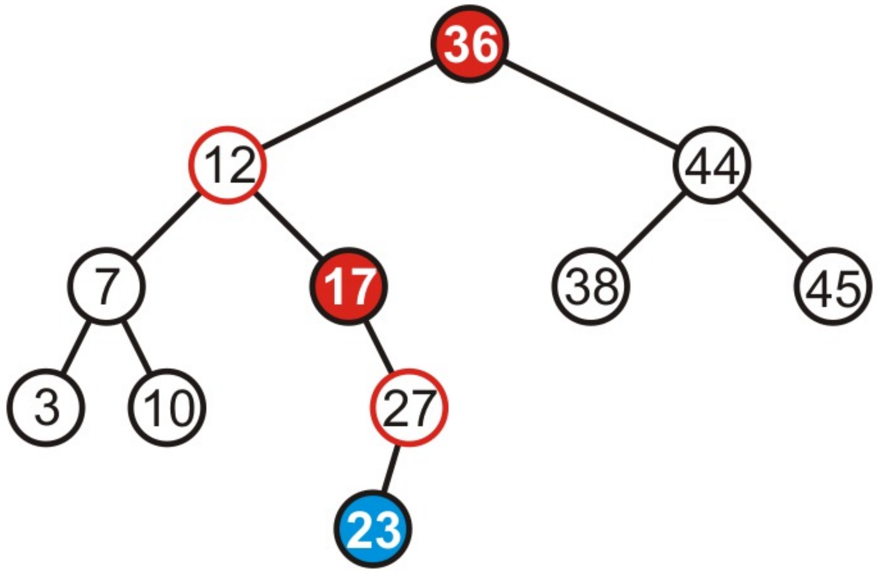
Maintaining Balance

- The heights of each of the sub-trees from here to the root are increased by one



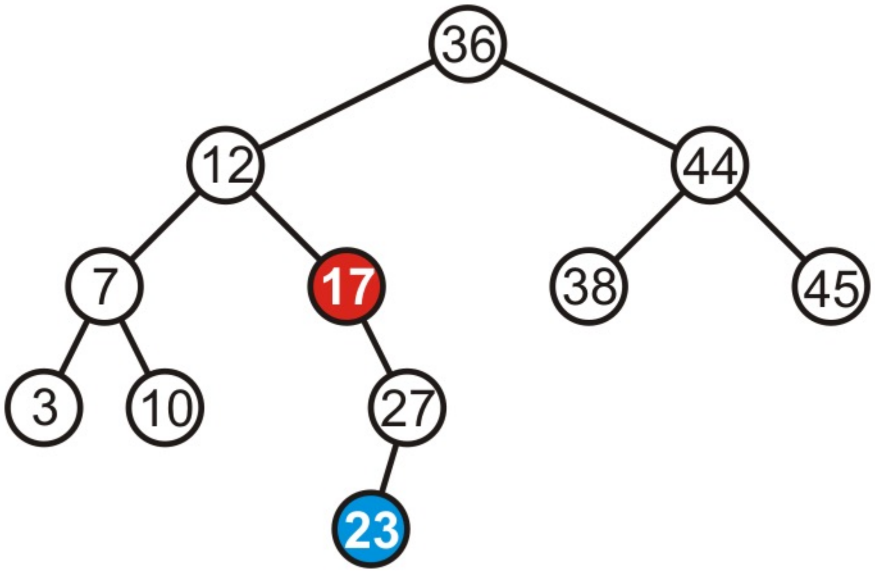
Maintaining Balance

- However, only two of the nodes are unbalanced: 17 and 36



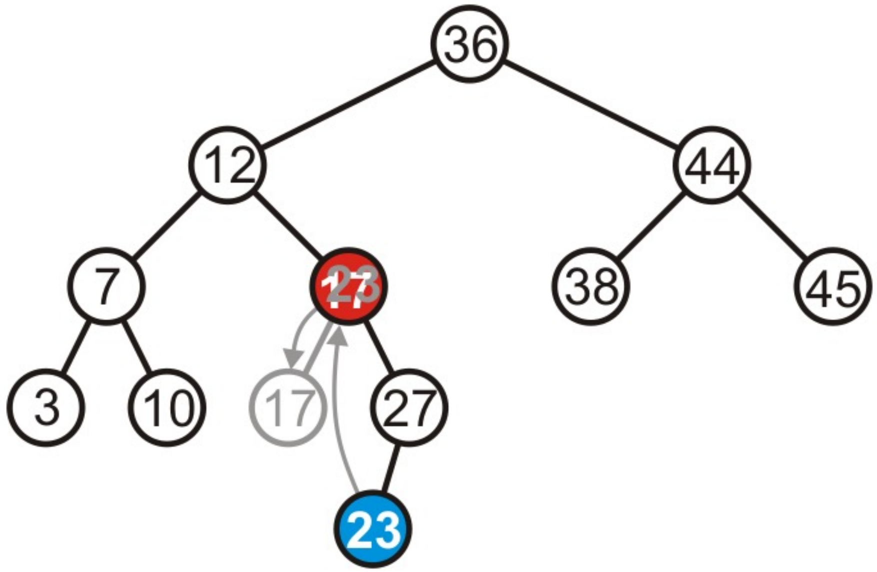
Maintaining Balance

- However, only two of the nodes are unbalanced: 17 and 36
  - We only have to fix the imbalance at the lowest node



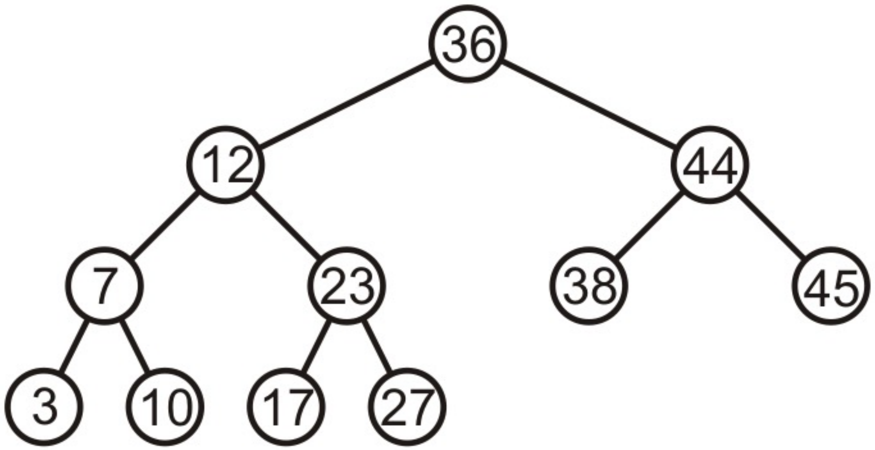
Maintaining Balance

- We can promote 23 to where 17 is, and make 17 the left child of 23



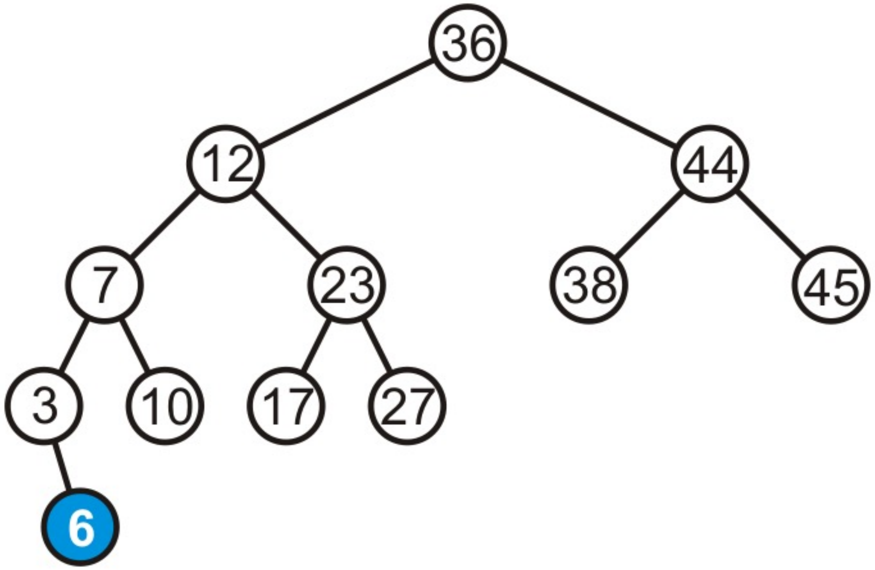
Maintaining Balance

- Thus, that node is no longer unbalanced
  - Incidentally, the root node is now balanced as well



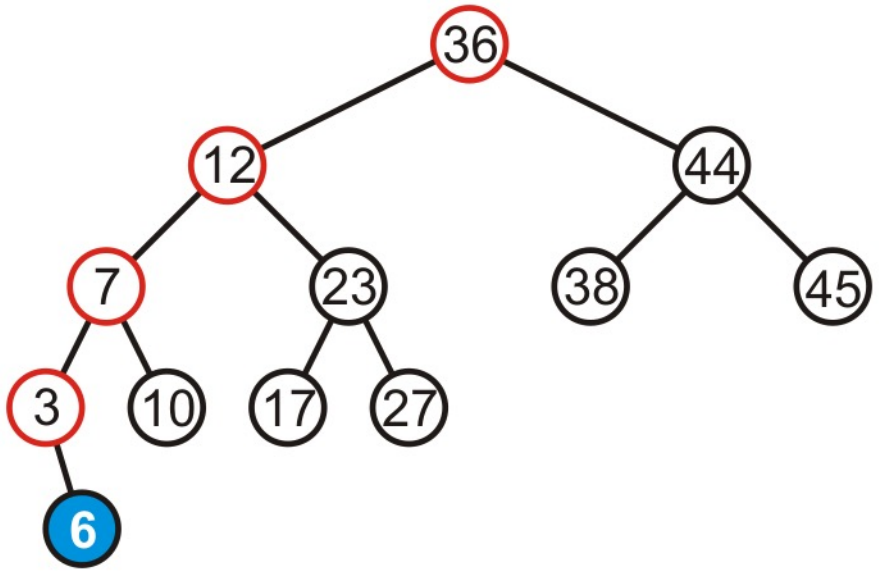
Maintaining Balance

- Consider adding 6:



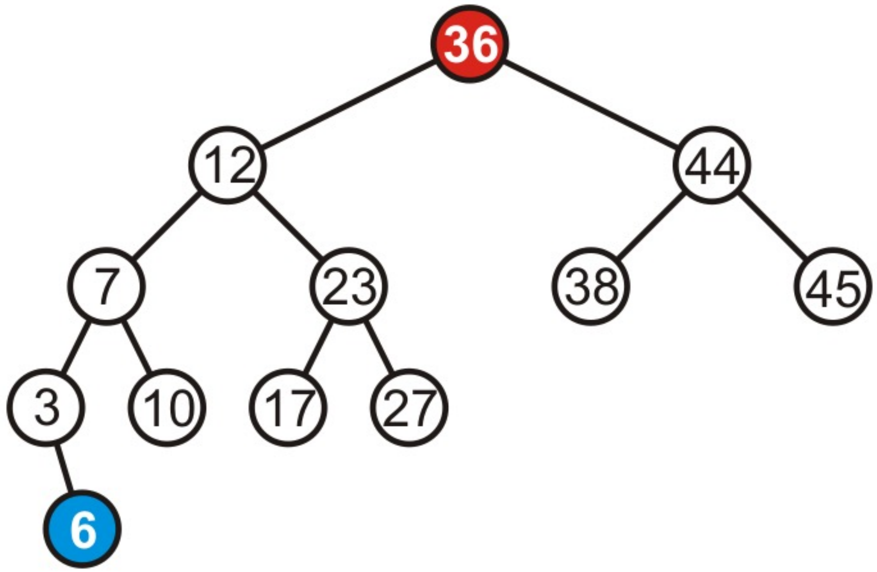
Maintaining Balance

- The height of each of the trees in the path back to the root are increased by one



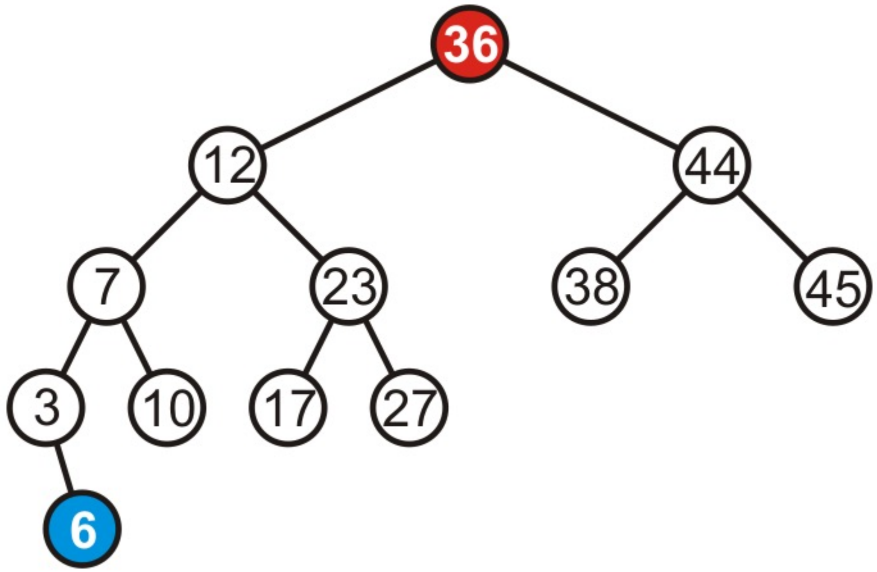
Maintaining Balance

- The height of each of the trees in the path back to the root are increased by one
  - However, only the root node is now unbalanced



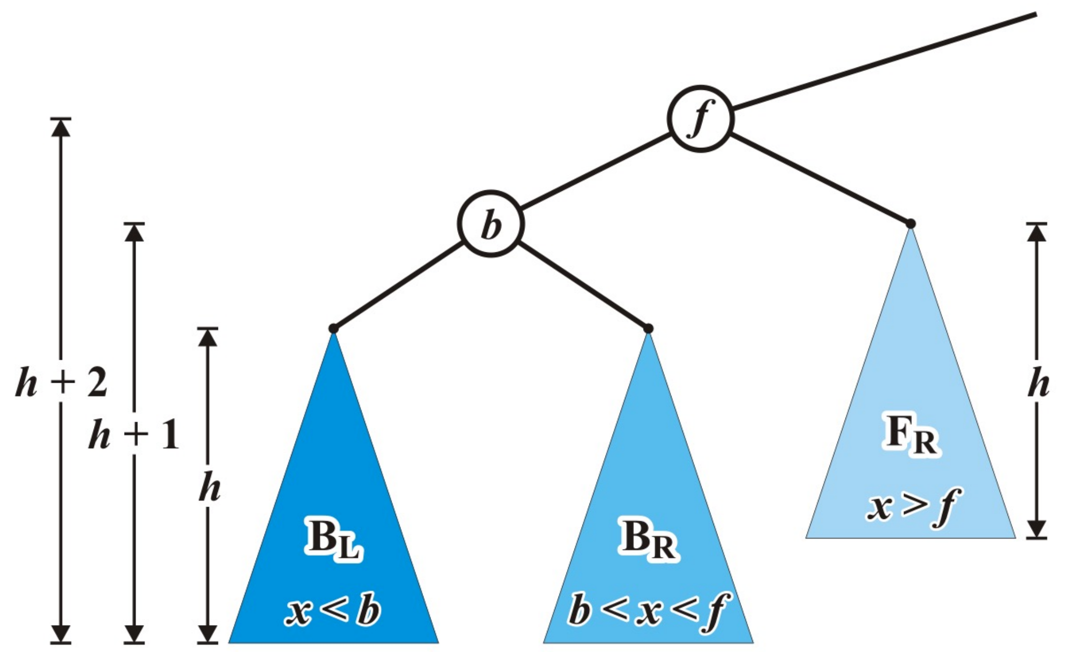
Maintaining Balance

- To fix this, we will look at the general case…



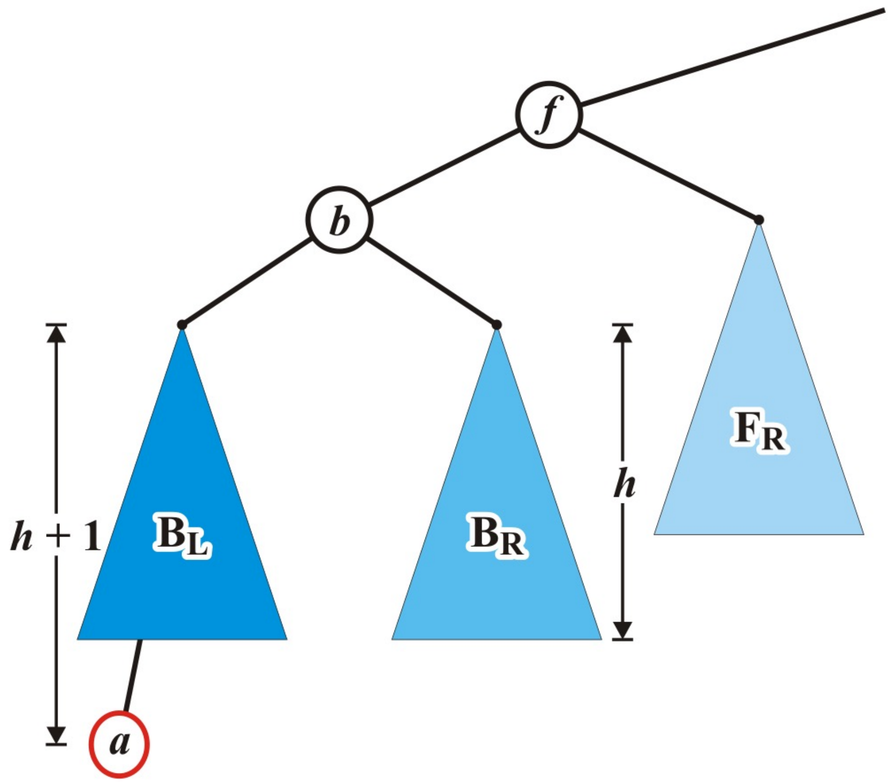
Maintaining Balance: Case 1

- Consider the following setup
  - Each blue triangle represents a tree of height *h*



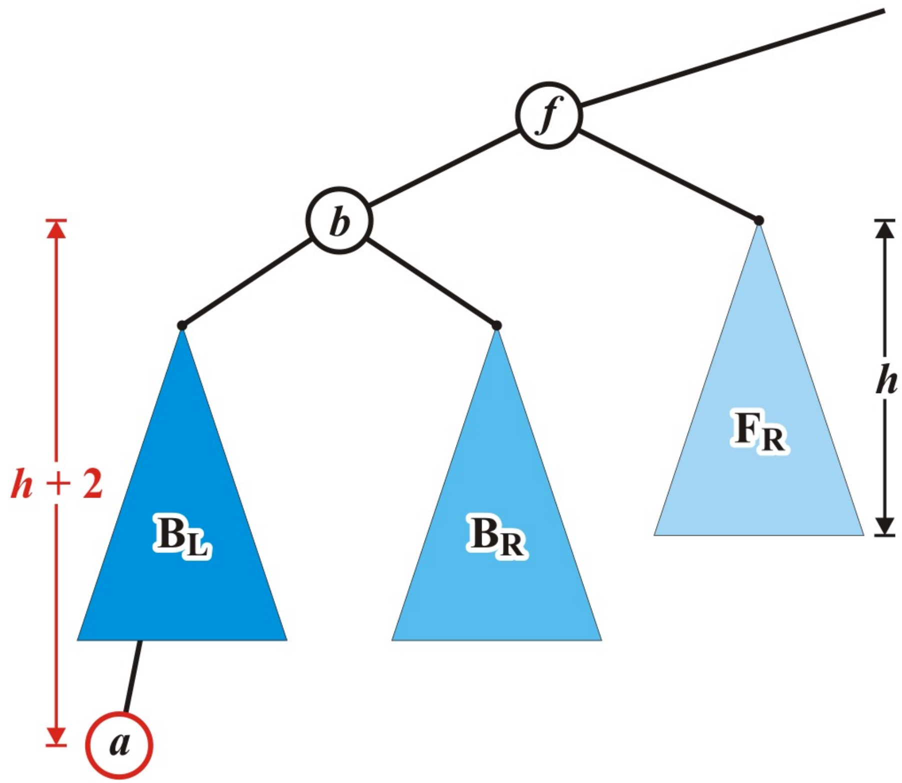
Maintaining Balance: Case 1

- Insert ***a*** into this tree: it falls into the left subtree B_L of ***b***
  - Assume B_L remains balanced
  - Thus, the tree rooted at ***b*** is also balanced



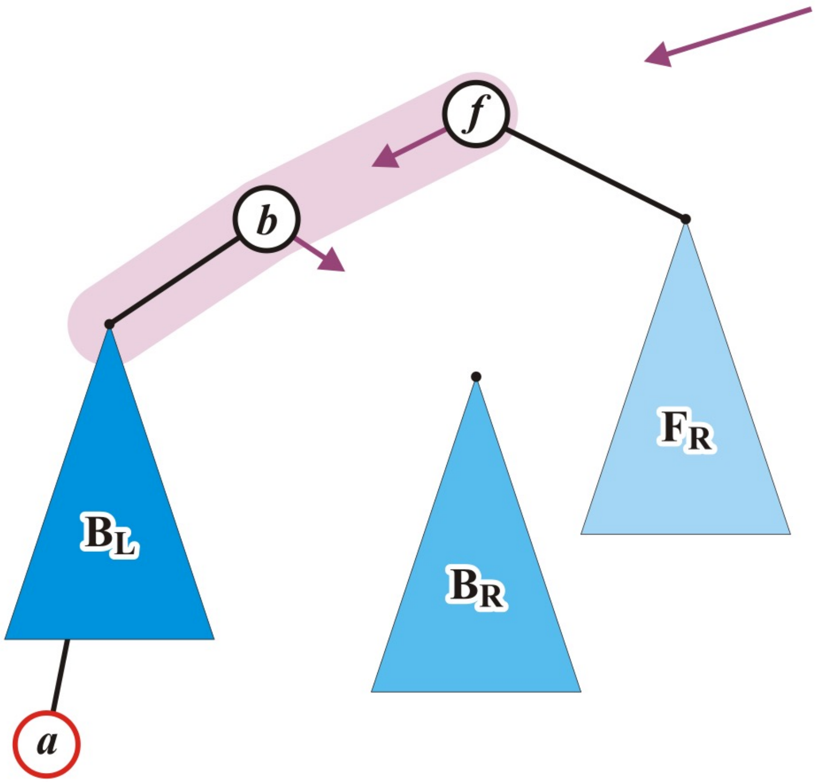
Maintaining Balance: Case 1

- The tree rooted at node f is now unbalanced
  - We will correct the imbalance at this node



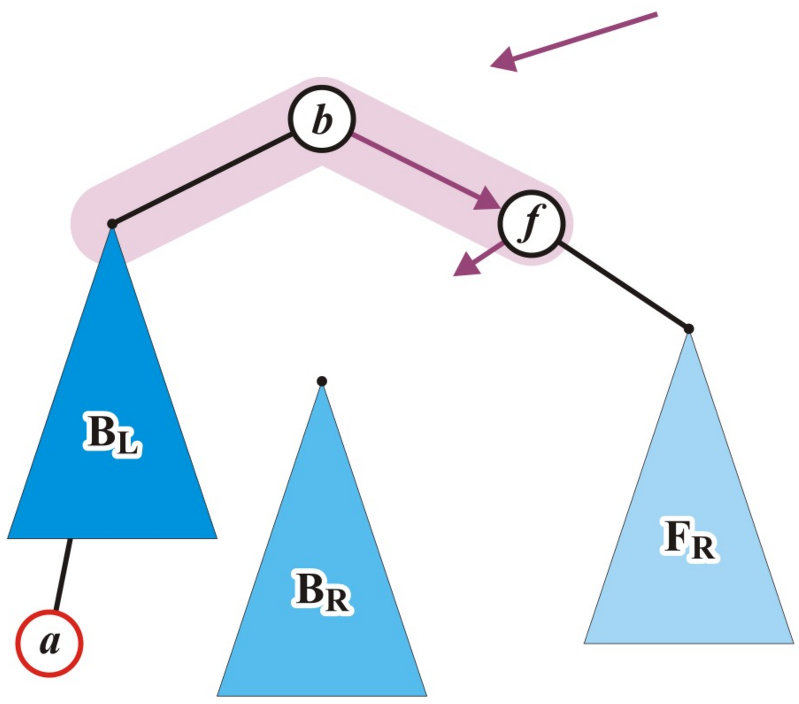
Maintaining Balance: Case 1

- Specifically, we will rotate these two nodes around the root:
  - Recall the first prototypical example
  - Promote node *b* to the root and demote node *f* to be the right child of *b*



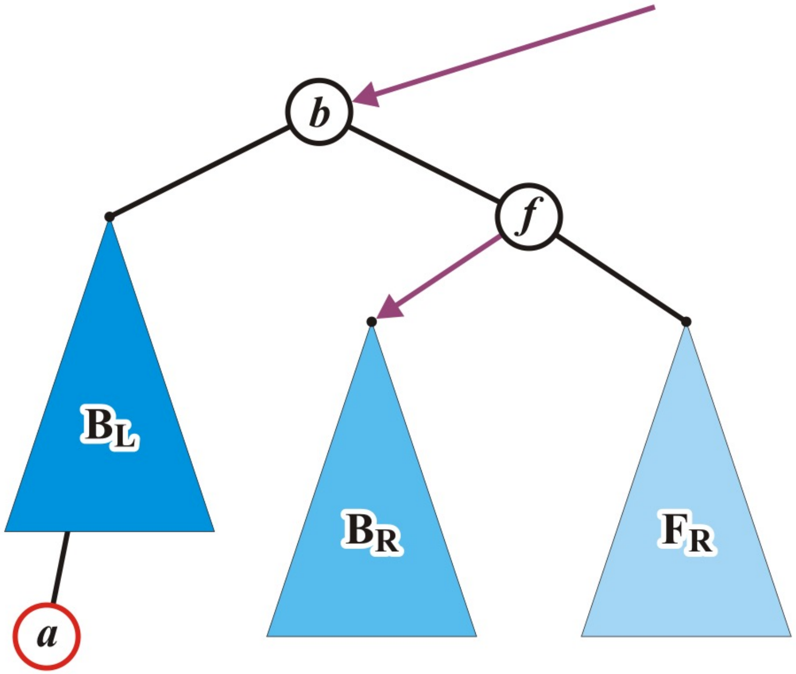
Maintaining Balance: Case 1

- This requires the address of node ***f*** to be assigned to the `p_right_tree` member variable of node ***b***



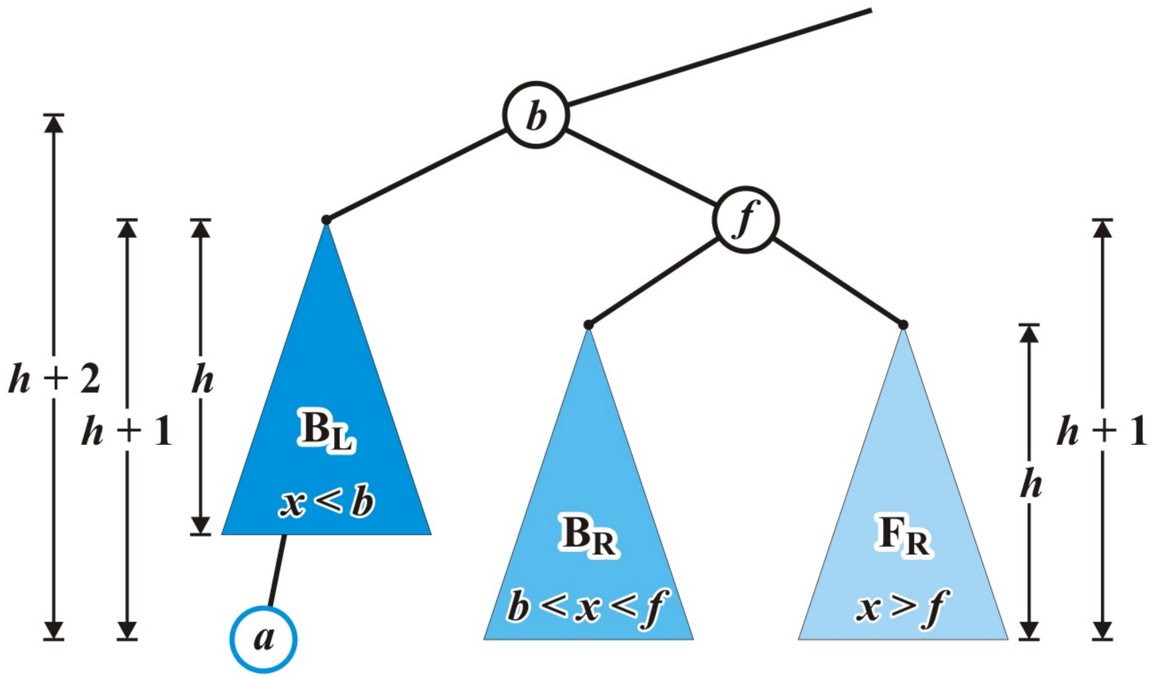
Maintaining Balance: Case 1

- Assign any former parent of node ***f*** to the address of node ***b***
- Assign the address of the tree B_R to `p_left_tree` of node ***f***



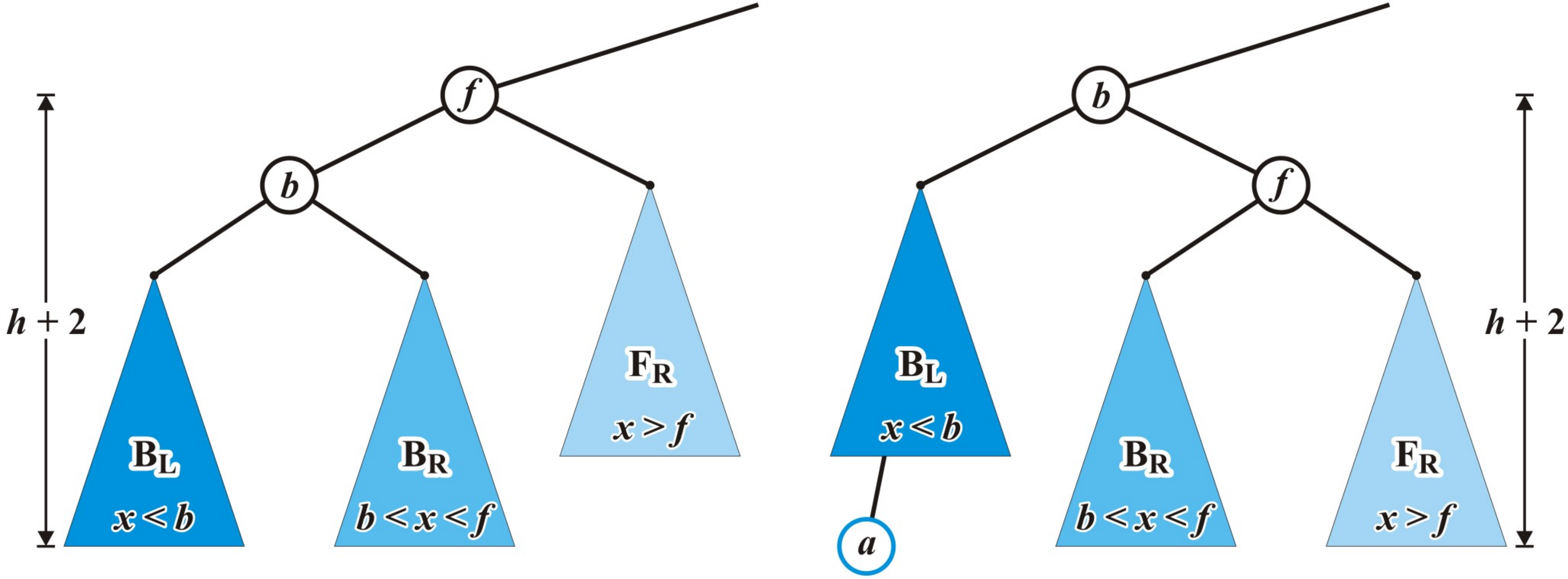
Maintaining Balance: Case 1

- The nodes ***b*** and ***f*** are now balanced and all remaining nodes of the subtrees are in their correct positions
  - The height of ***f*** is now *h* + 1 while ***b*** remains at height *h* + 2



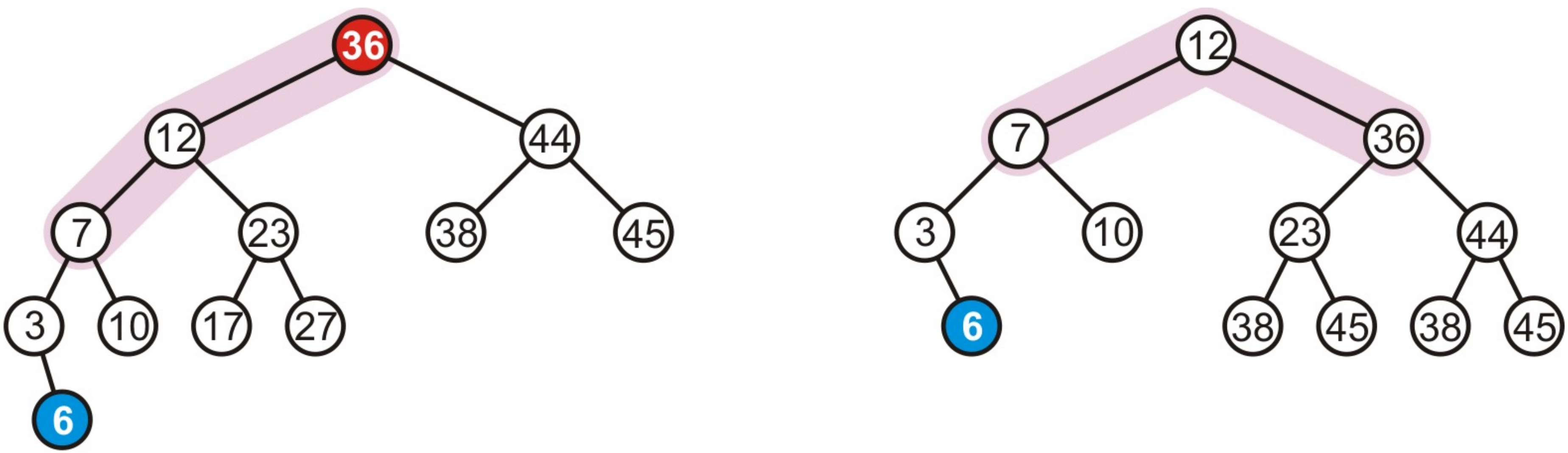
Maintaining Balance: Case 1

- Additionally, height of the corrected tree rooted at ***b*** equals the original height of the tree rooted at ***f***
  - Thus, this insertion will no longer affect the balance of any ancestors all the way back to the root



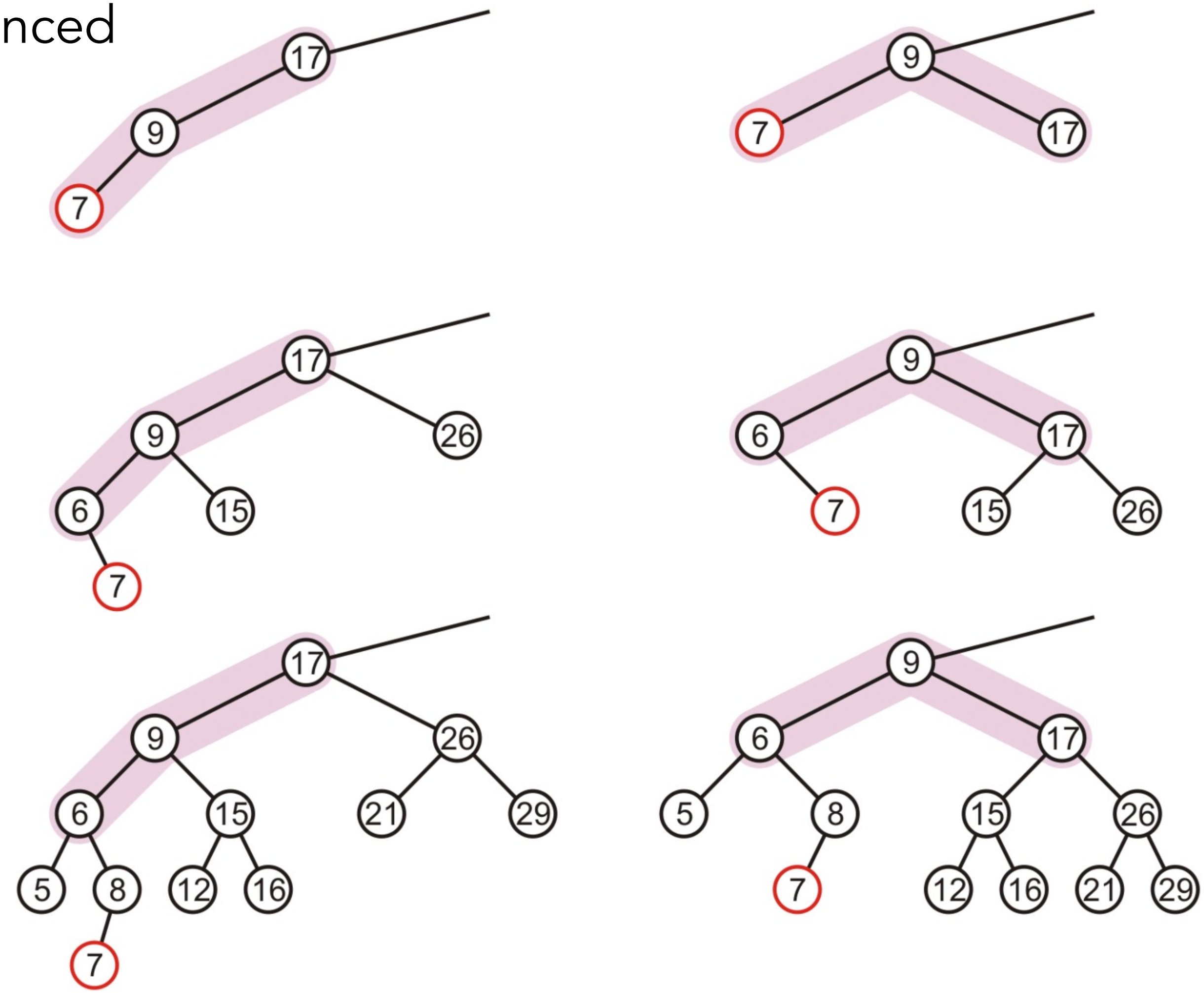
Maintaining Balance: Case 1

- In our example case, the correction



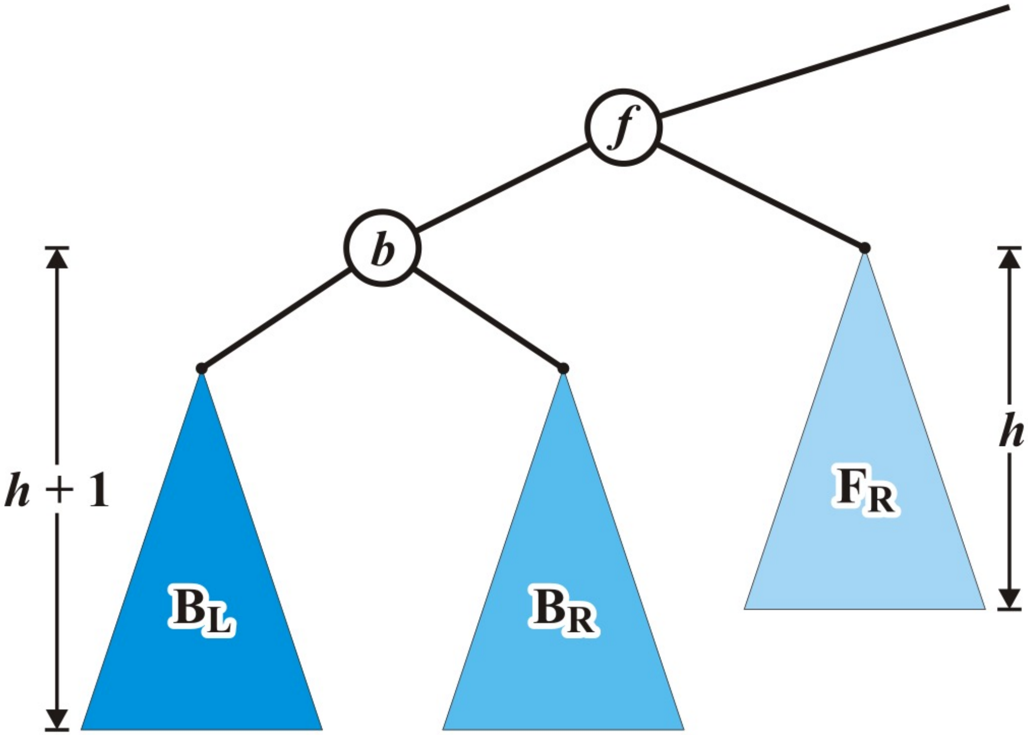
Maintaining Balance: Case 1

- In our three sample cases, the node is now balanced and the same height as the tree before the insertion



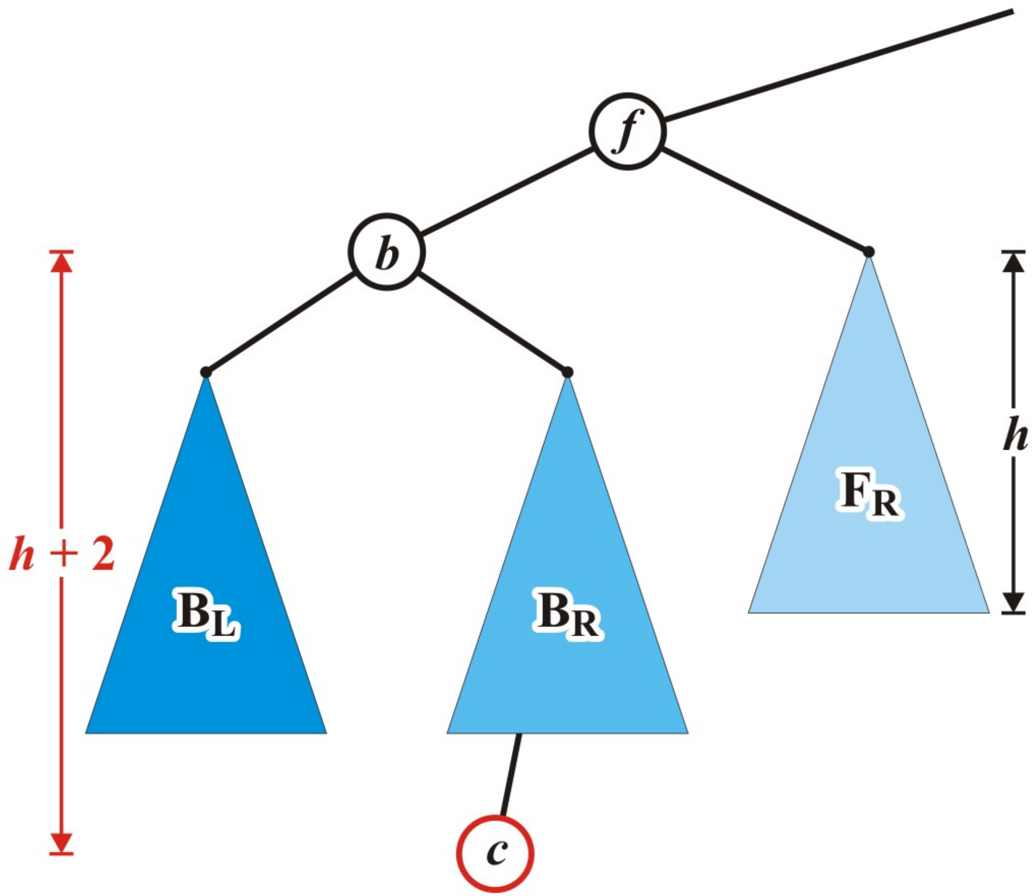
Maintaining Balance: Case 2

- Alternatively, consider the insertion of **c** where ***b*** < **c** < ***f*** into our original tree



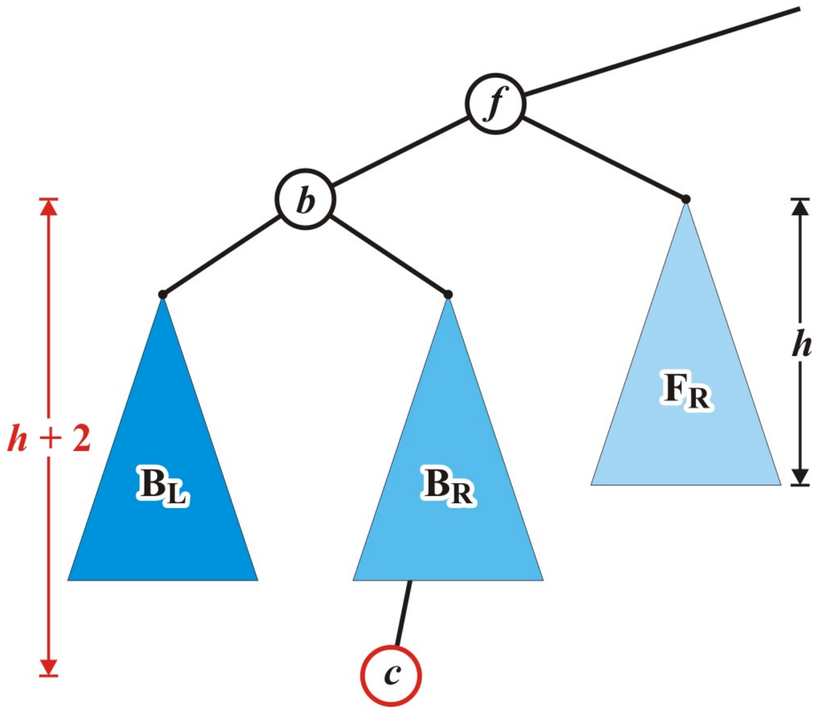
Maintaining Balance: Case 2

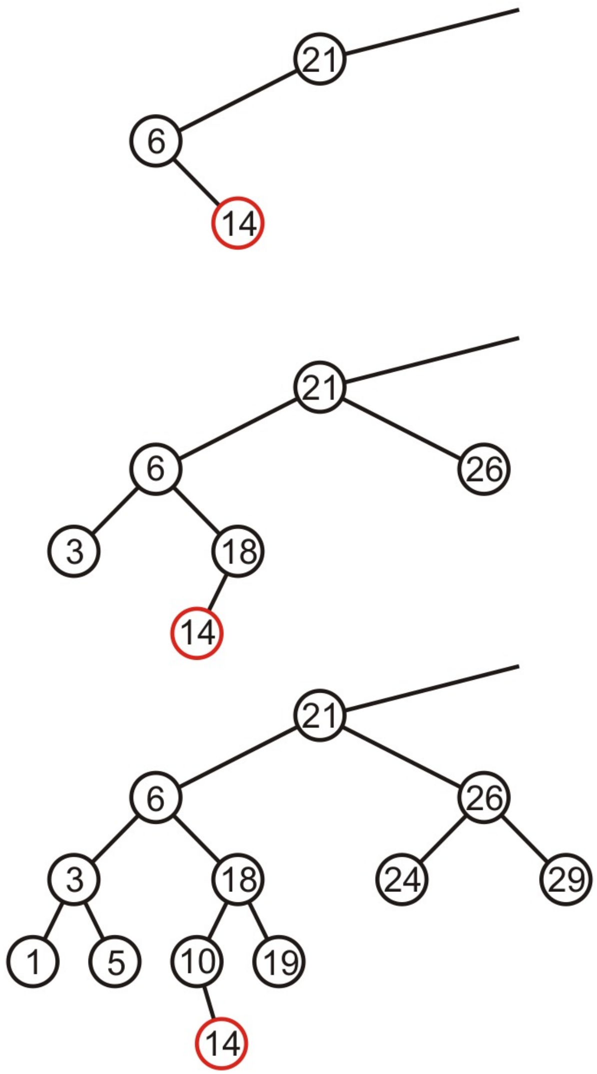
- Assume that the insertion of **c** increases the height of B_R
  - Once again, f becomes unbalanced



Maintaining Balance: Case 2

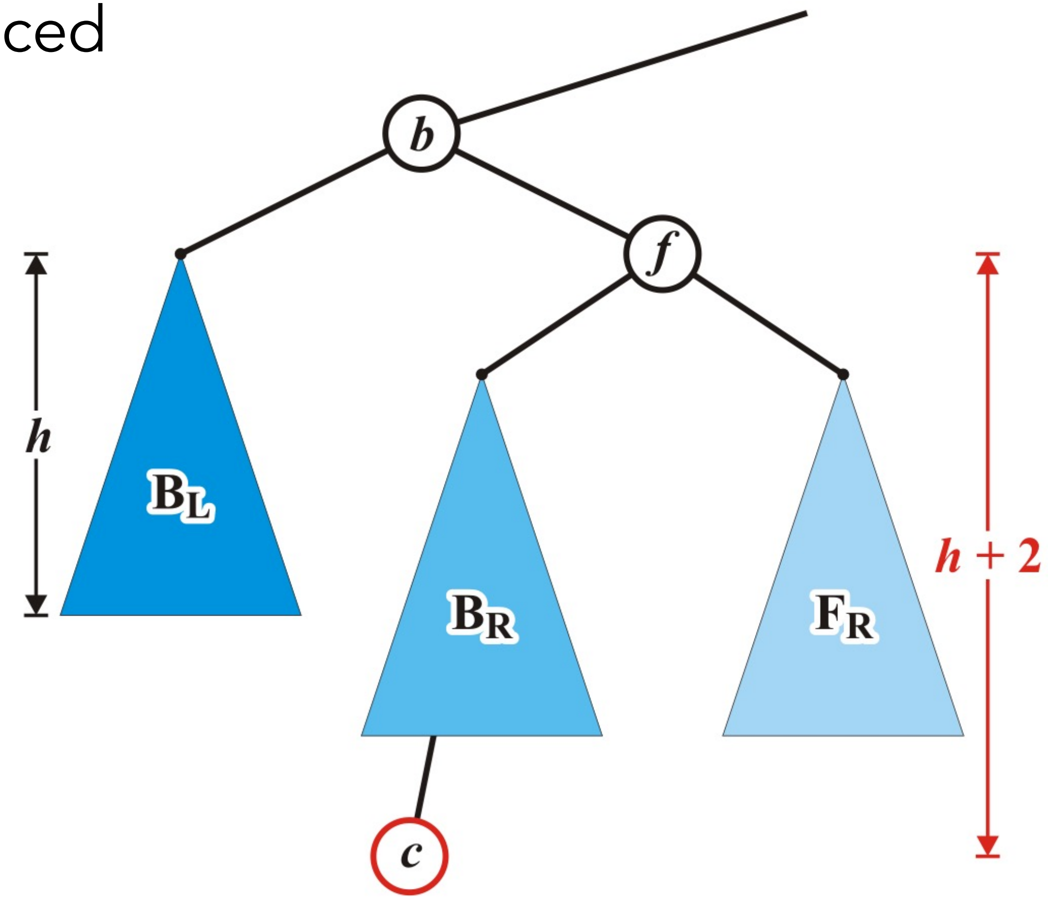
- Here are examples of when the insertion of 14 may cause this situation when $h = -1$, 0, and 1





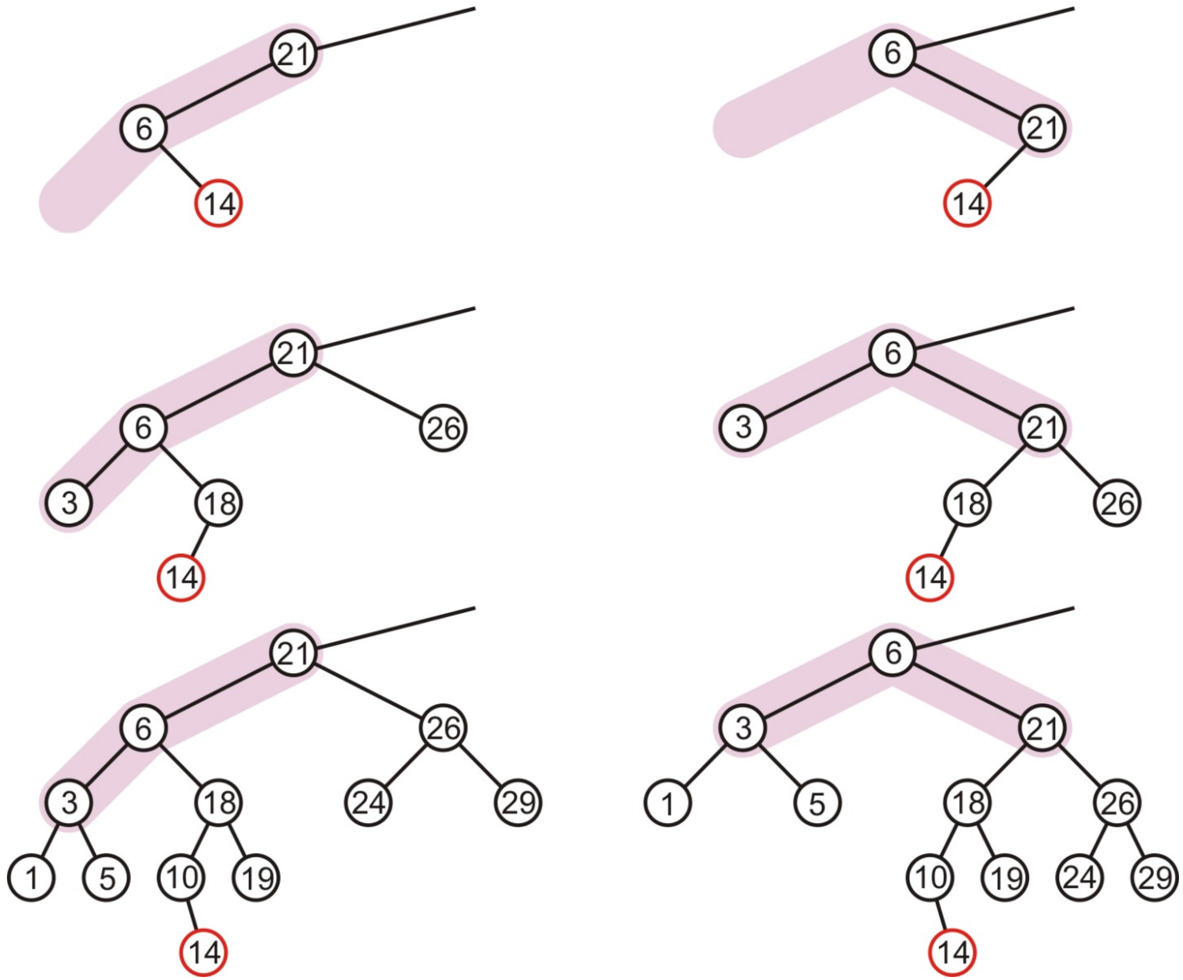
Maintaining Balance: Case 2

- Unfortunately, the previous correction does not fix the imbalance at the root of this sub-tree: the new root, ***b***, remains unbalanced



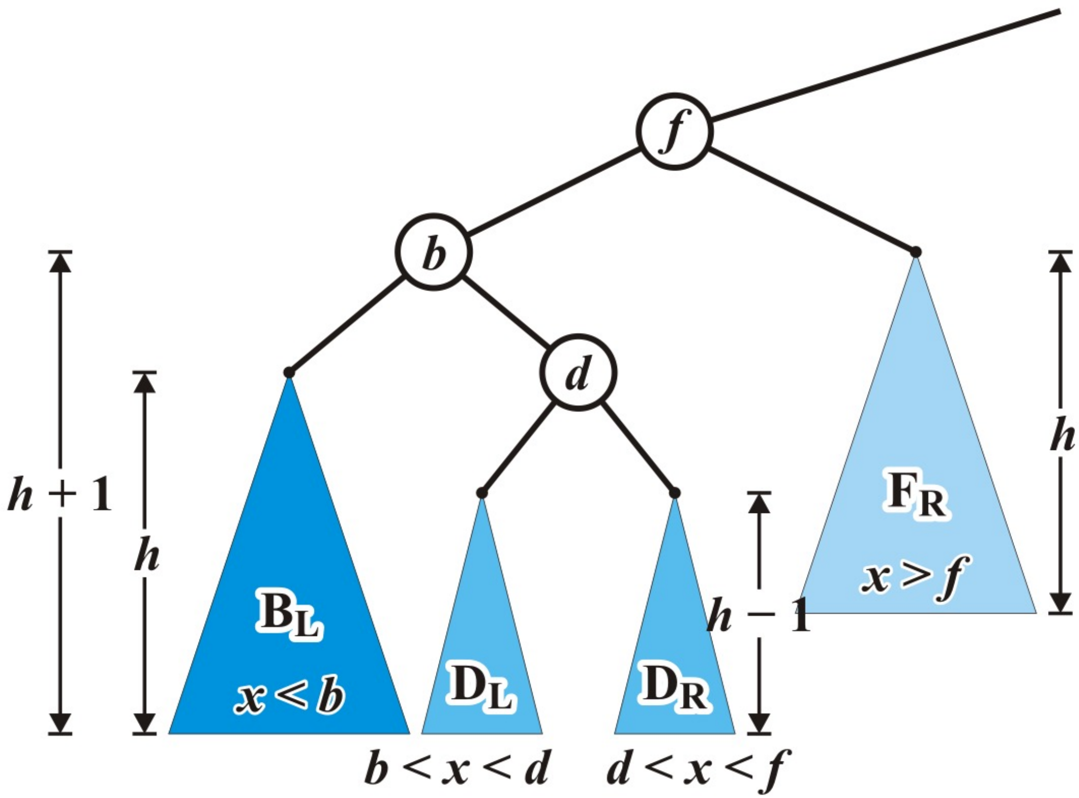
Maintaining Balance: Case 2

- In our three sample cases with $h = -1$, 0, and 1, doing the same thing as before results in a tree that is still unbalanced…
  - The imbalance is just shifted to the other side



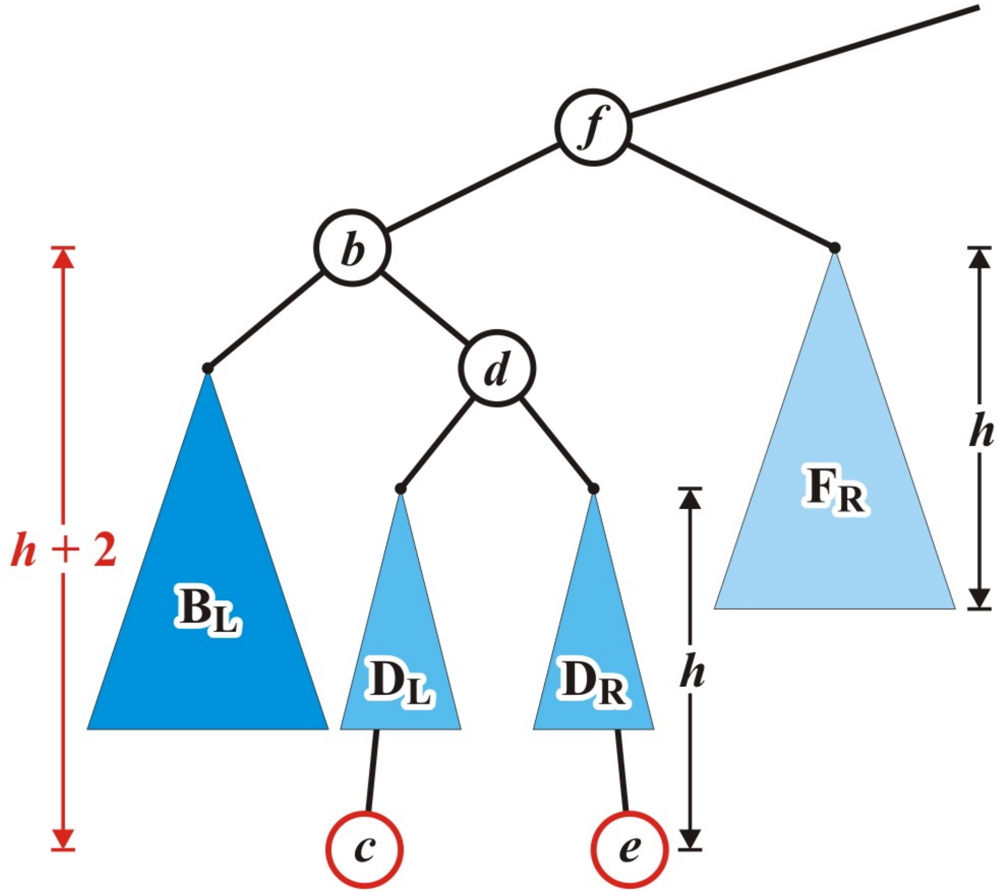
Maintaining Balance: Case 2

- Re-label the tree by dividing the left subtree of f into a tree rooted at d with two subtrees of height $h - 1$



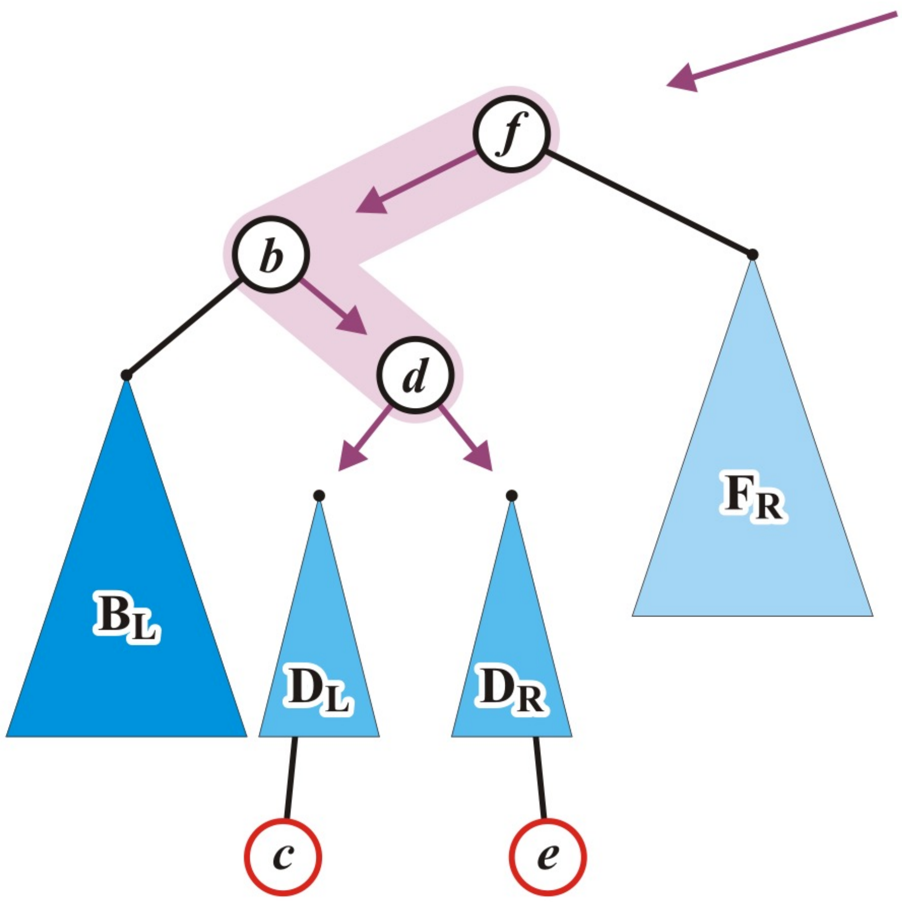
Maintaining Balance: Case 2

- Now an insertion causes an imbalance at ***f***
  - The addition of either **c** or **e** will cause this



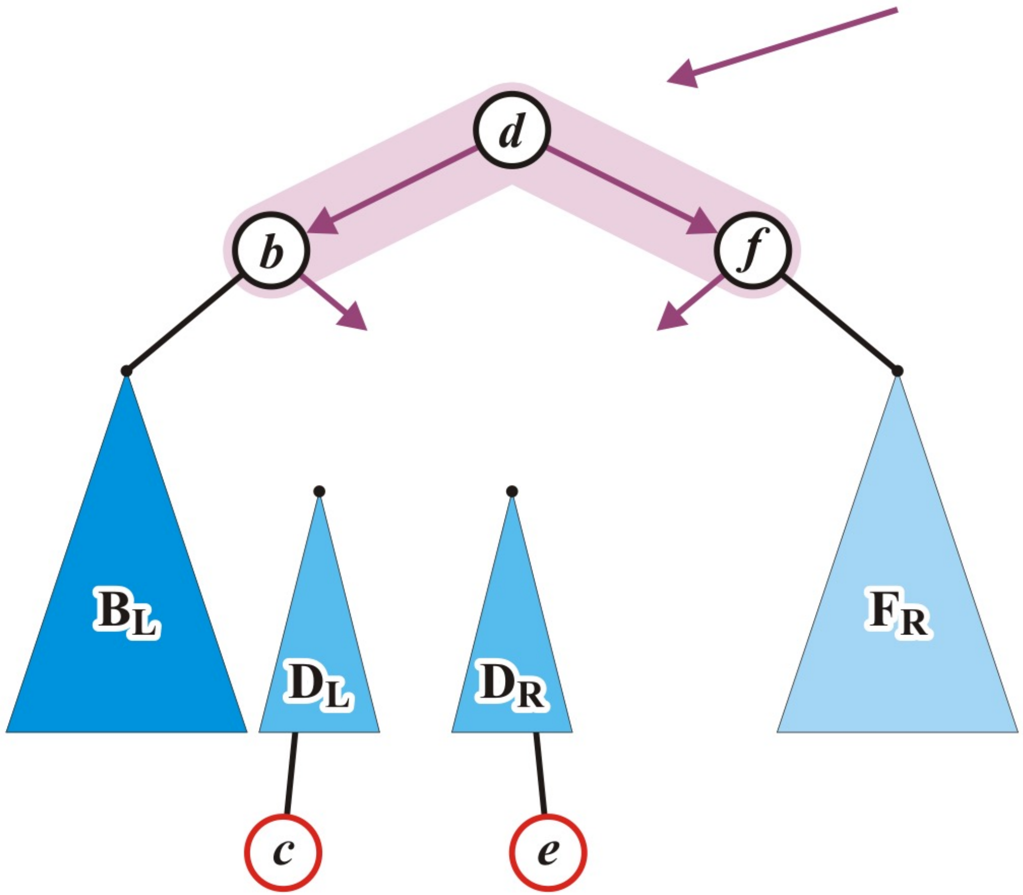
Maintaining Balance: Case 2

- We will rotate *d*, *b*, and *f*



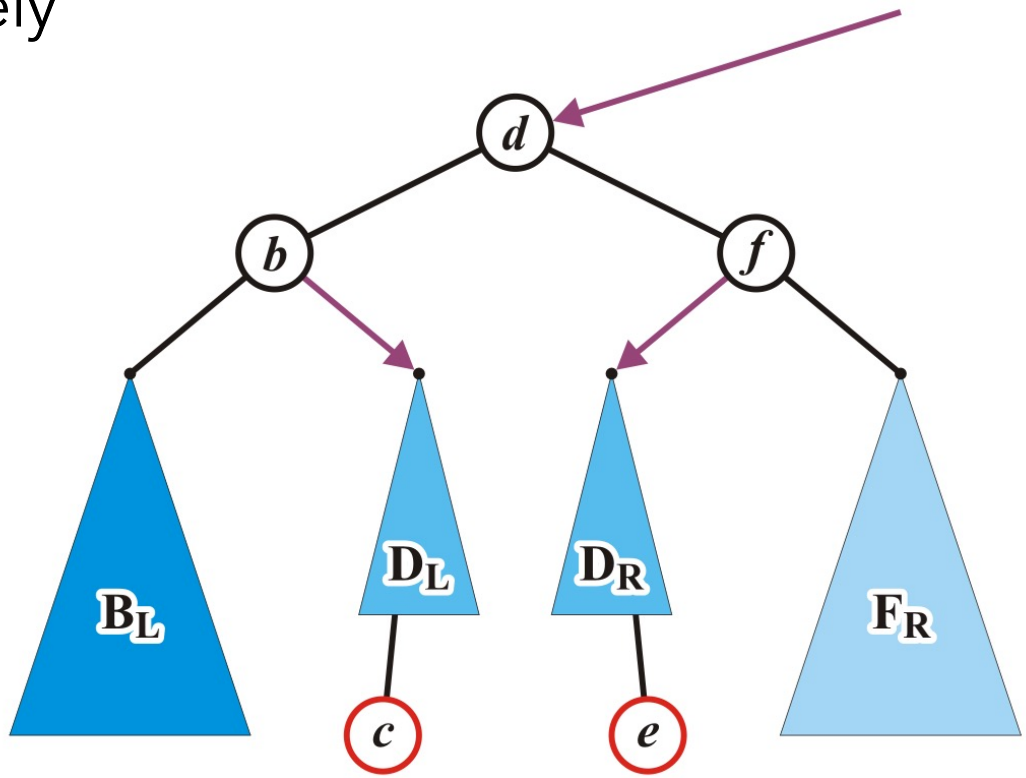
Maintaining Balance: Case 2

- We will first rotate ***d***, ***b***, and ***f***



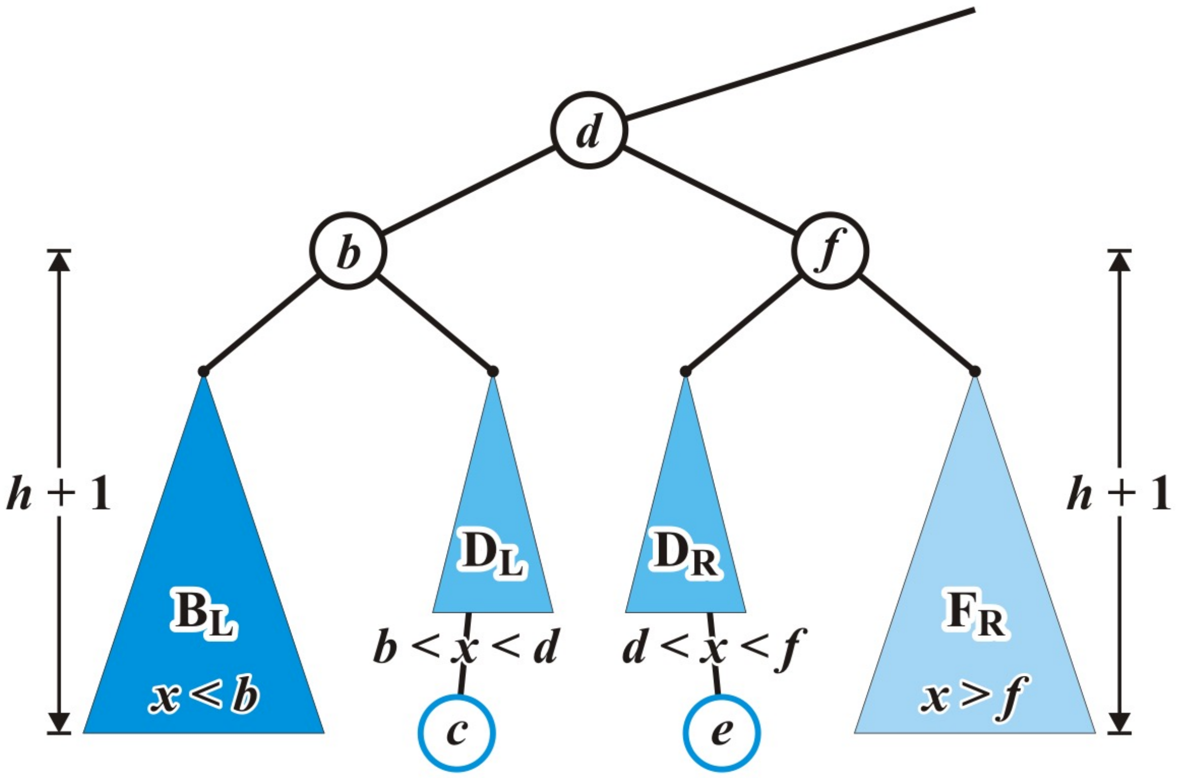
Maintaining Balance: Case 2

- Then connect D_L and D_R as a subtree of b and f, respectively



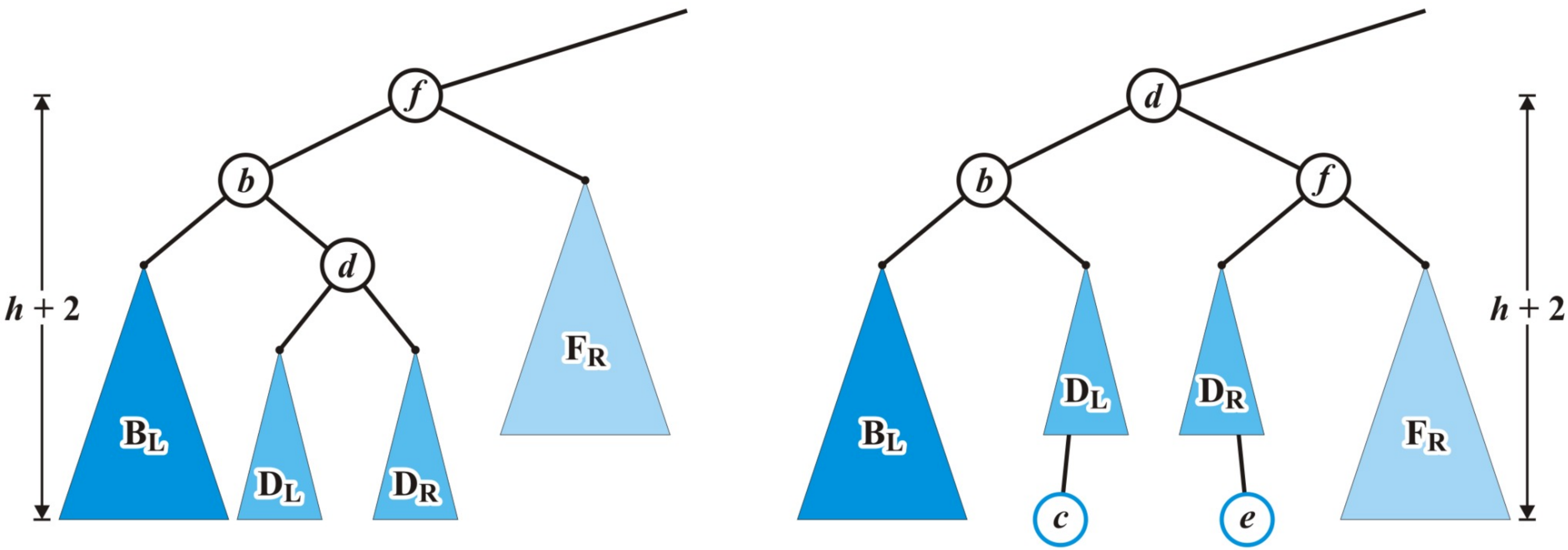
Maintaining Balance: Case 2

- Now the tree rooted at ***d*** is balanced
  - After the correction, height of ***b*** and ***f*** become h + 1 and ***d*** is h + 2



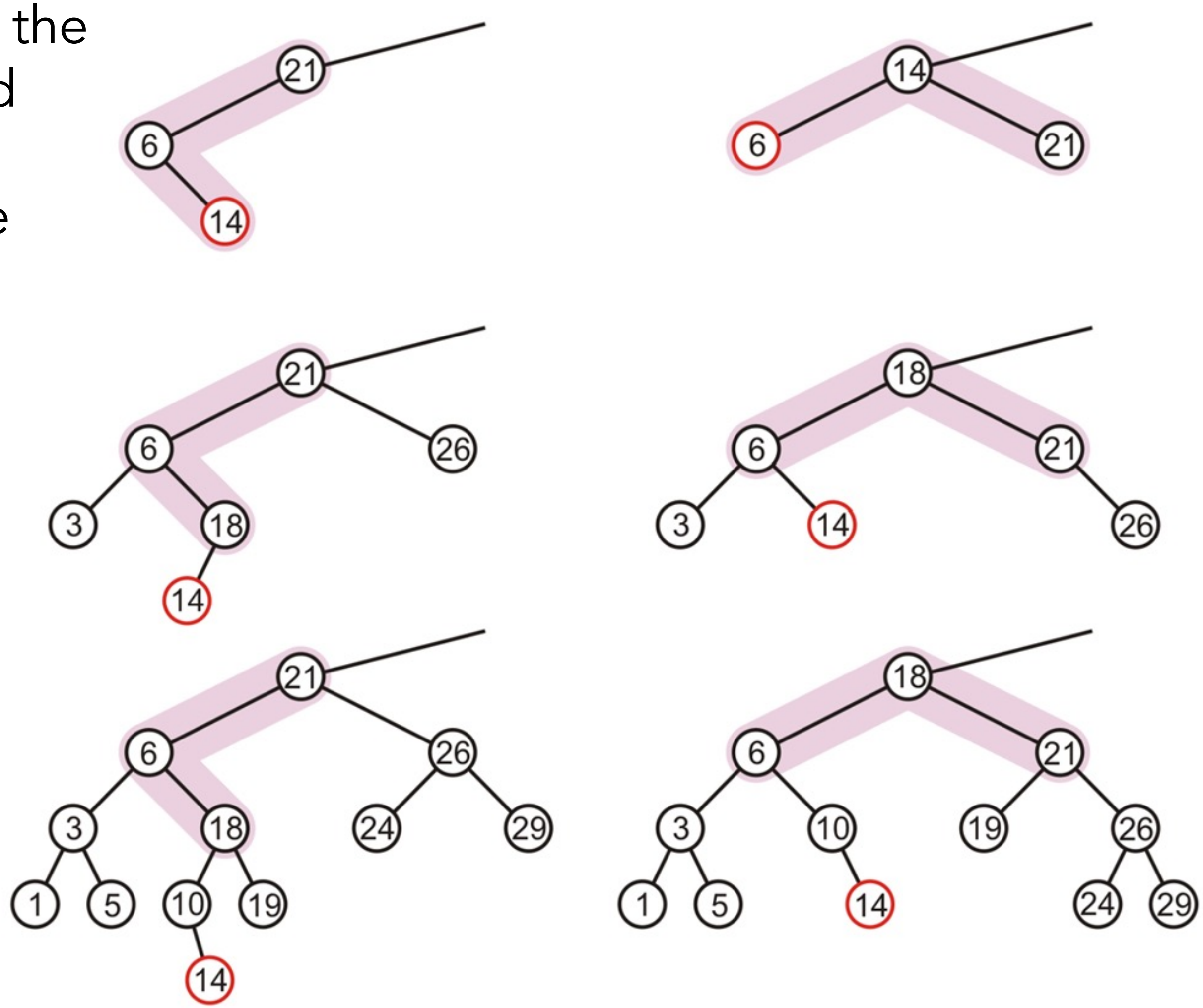
Maintaining Balance: Case 2

- Again, the height of the root did not change
  - The heights of all three nodes changed in this process



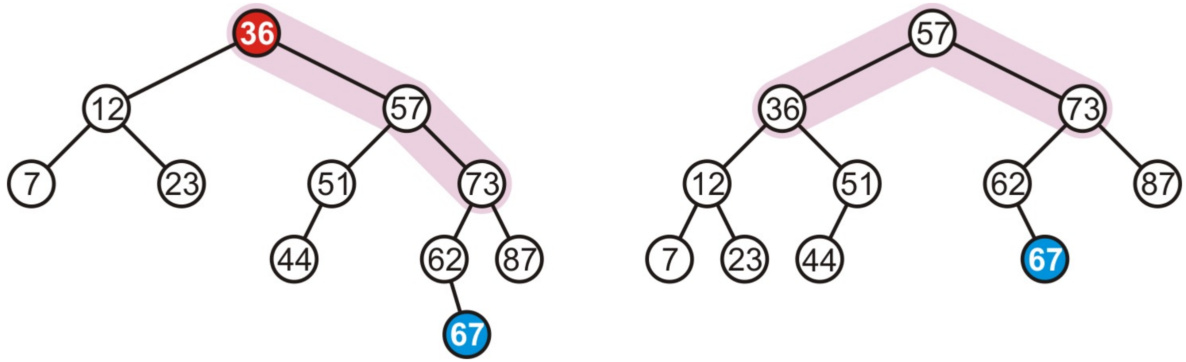
Maintaining Balance: Case 2

- In our three sample cases with $h = -1$, 0, and 1, the node is now balanced and the same height as the tree before the insertion

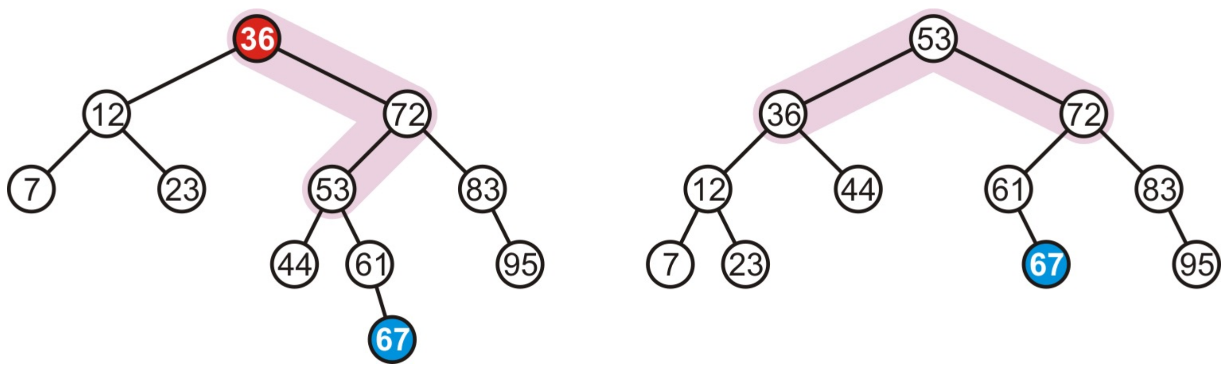


Maintaining balance: Summary

- There are two symmetric cases to those we have examined:
  - Insertions into the right-right sub-tree (Case 1)



  - Insertions into either the right-left sub-tree (Case 2)



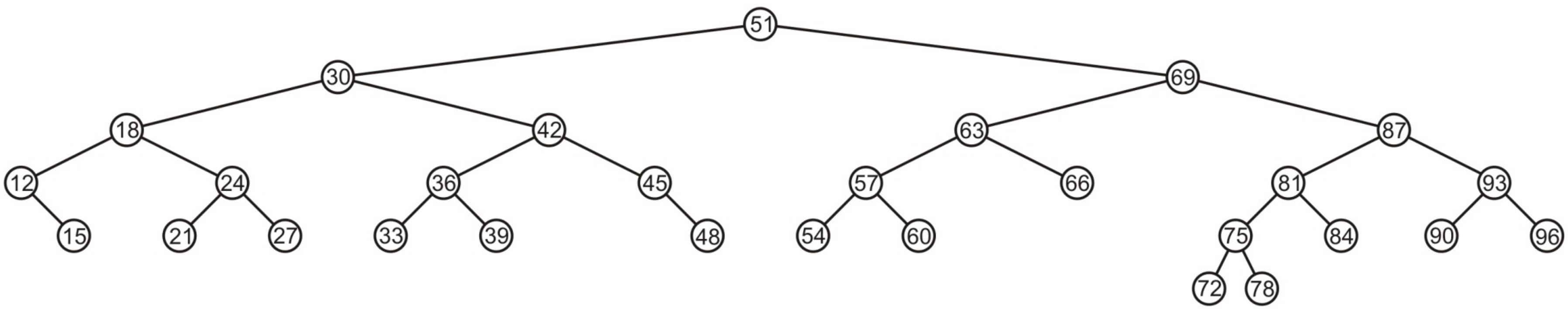
Time Complexity of Insertion

- Both balances (i.e., Case 1 and Case 2) are $\Theta(1)$
- All insertions are still $\Theta(\ln(n))$

Why?

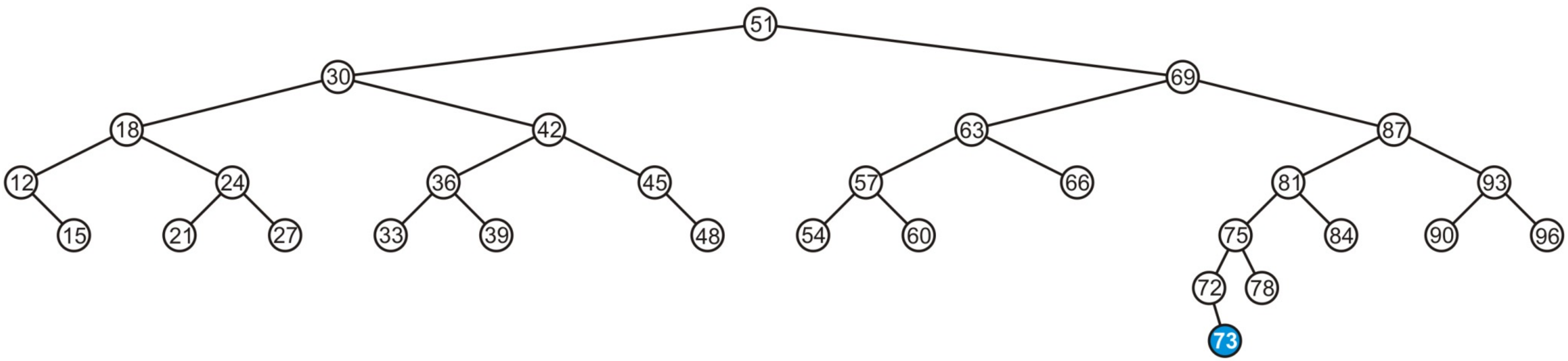
Insertion

- Consider this AVL tree



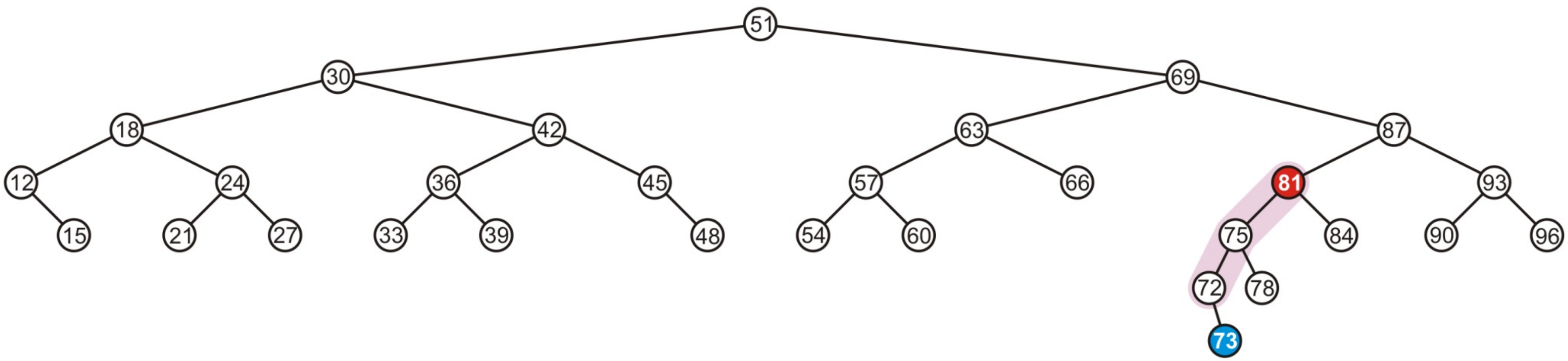
Insertion: 73

- Insert 73



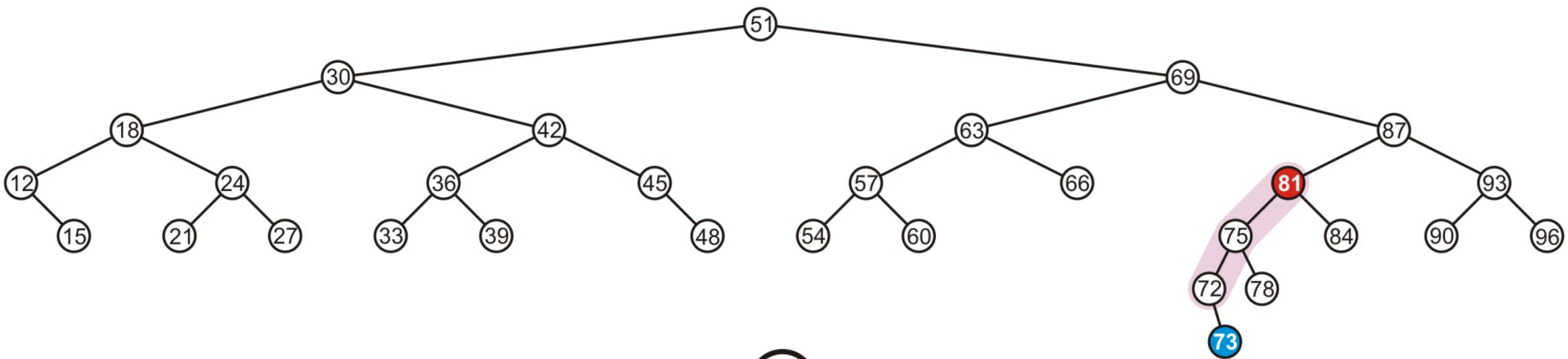
Insertion: 73

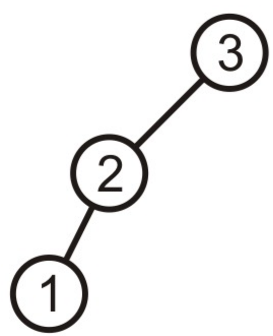
- The node 81 is unbalanced
  - A left-left imbalance



Insertion: 73

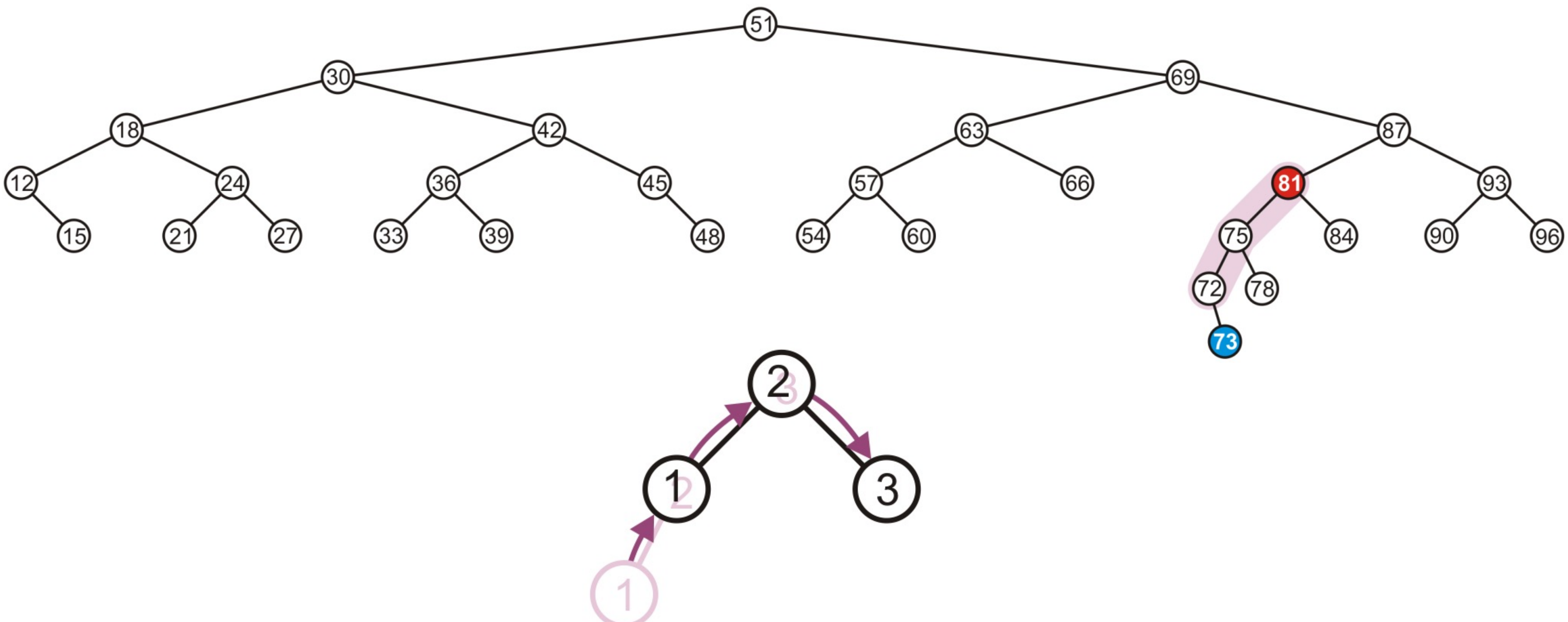
- The node 81 is unbalanced
  - A left-left imbalance





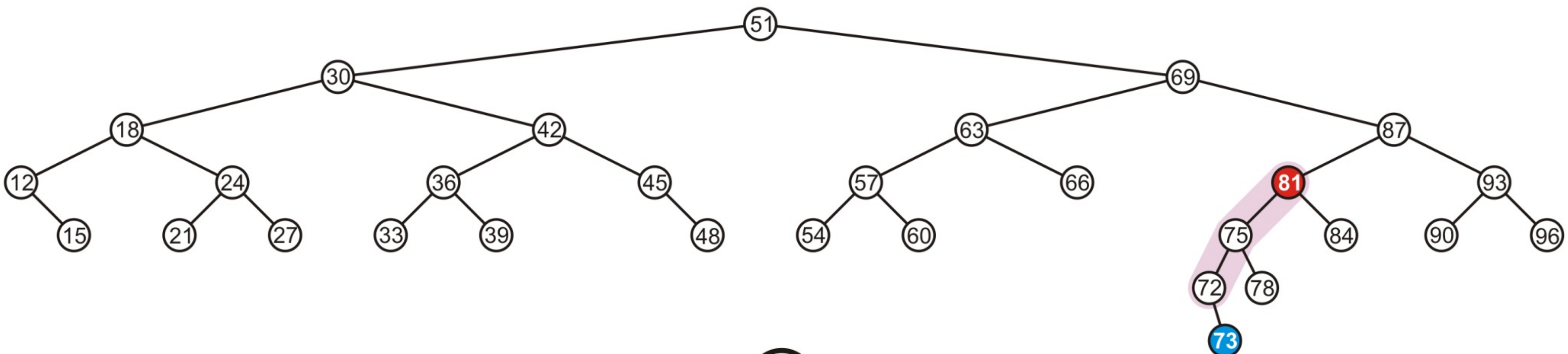
Insertion: 73

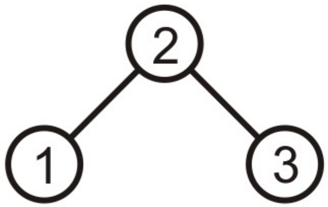
- The node 81 is unbalanced
  - A left-left imbalance
  - Promote the intermediate node to the imbalanced node



Insertion: 73

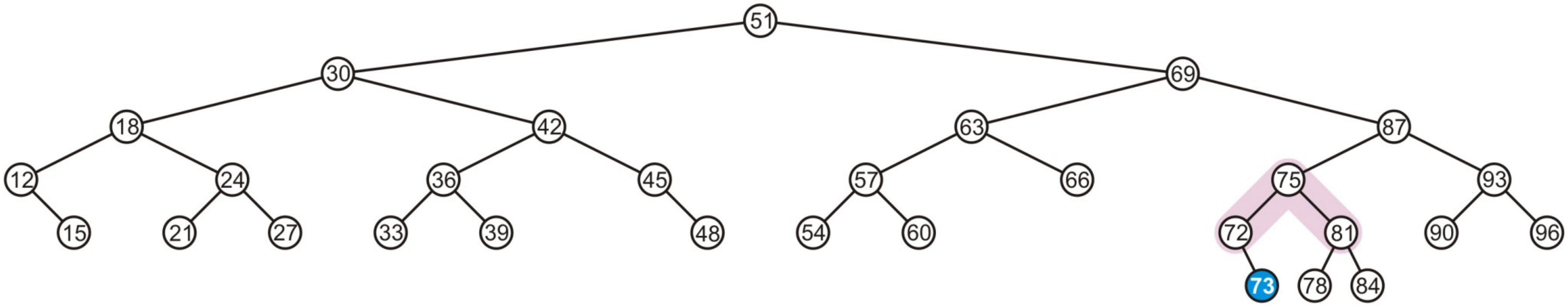
- The node 81 is unbalanced
  - A left-left imbalance
  - Promote the intermediate node to the imbalanced node
  - 75 is that node





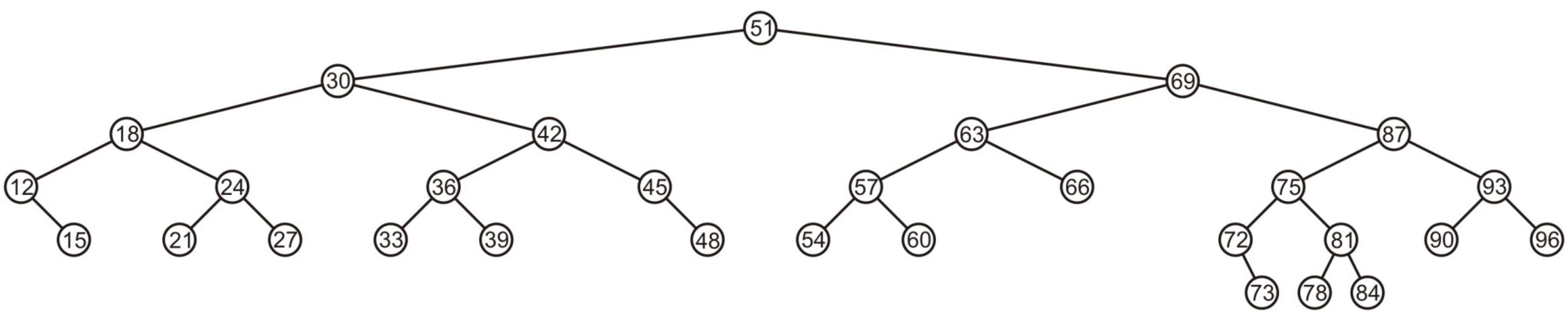
Insertion: 73

- The node 81 is unbalanced
  - A left-left imbalance
  - Promote the intermediate node to the imbalanced node
  - 75 is that node



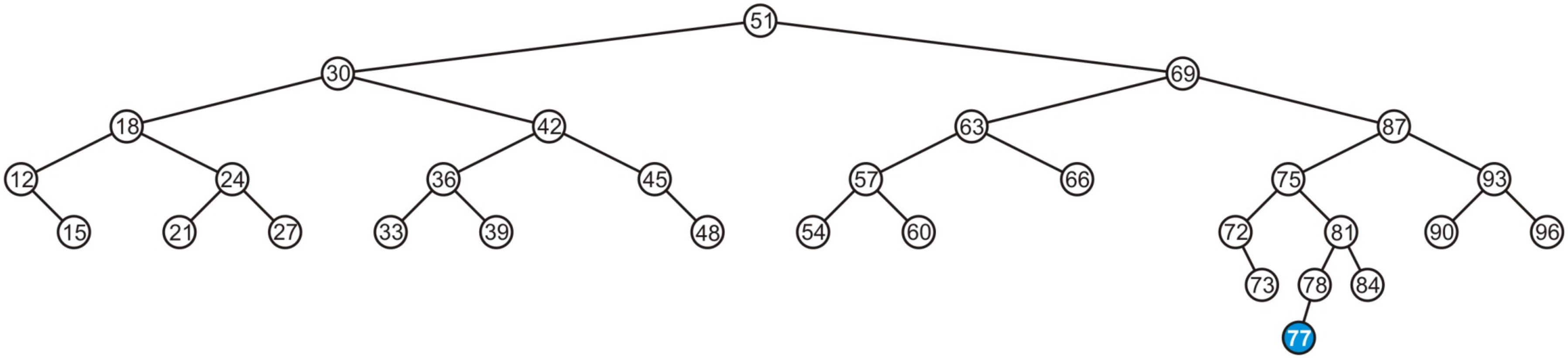
Insertion: 73

- The tree is AVL balanced



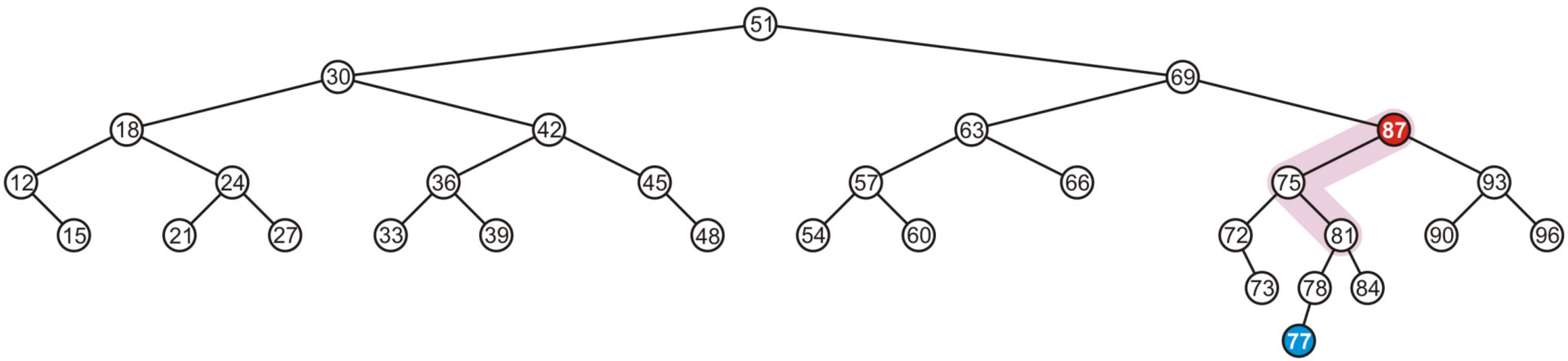
Insertion: 77

- Insert 77



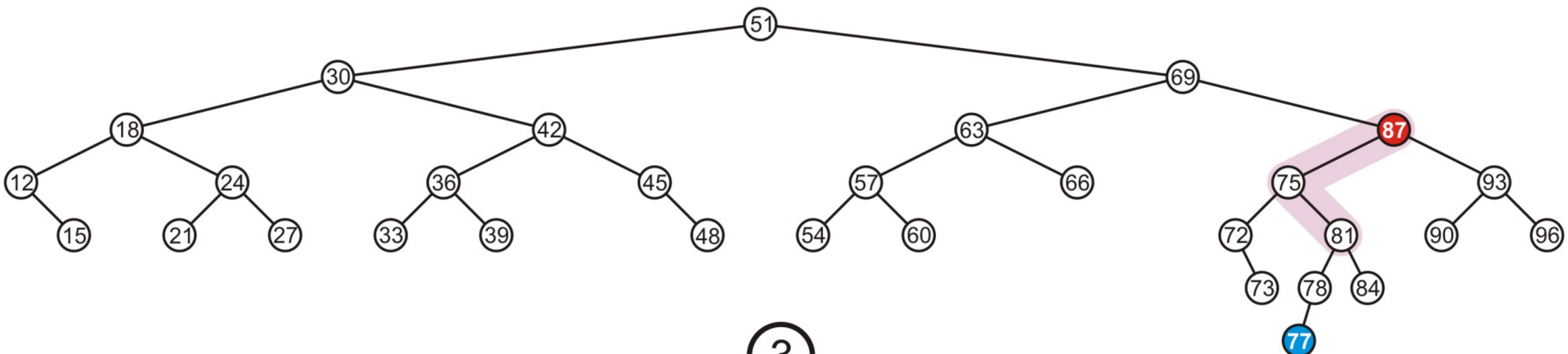
Insertion: 77

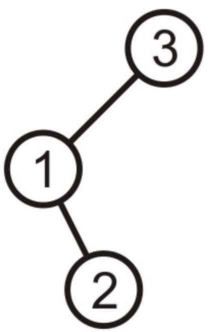
- The node 87 is unbalanced
  - A left-right imbalance



Insertion: 77

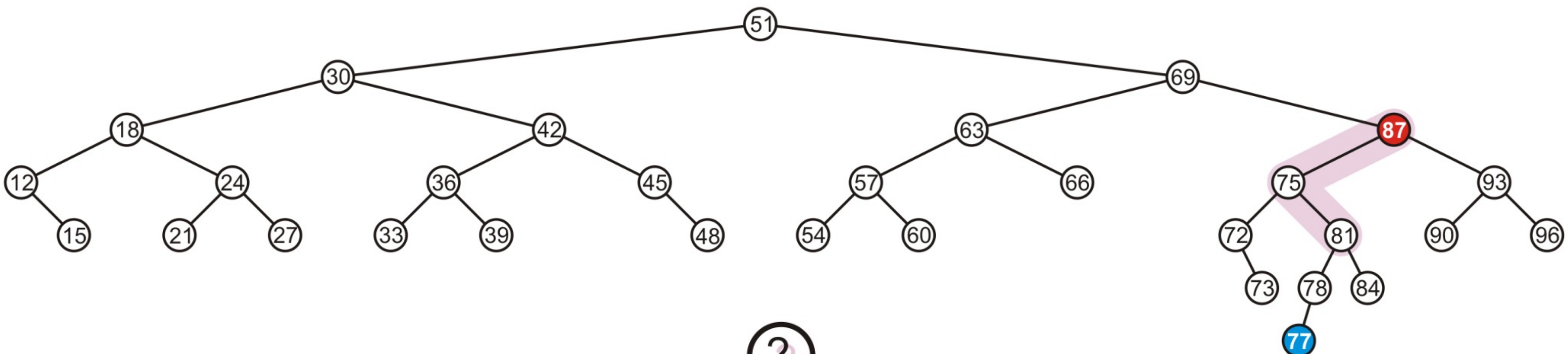
- The node 87 is unbalanced
  - A left-right imbalance

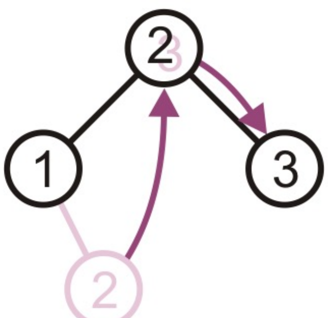




Insertion: 77

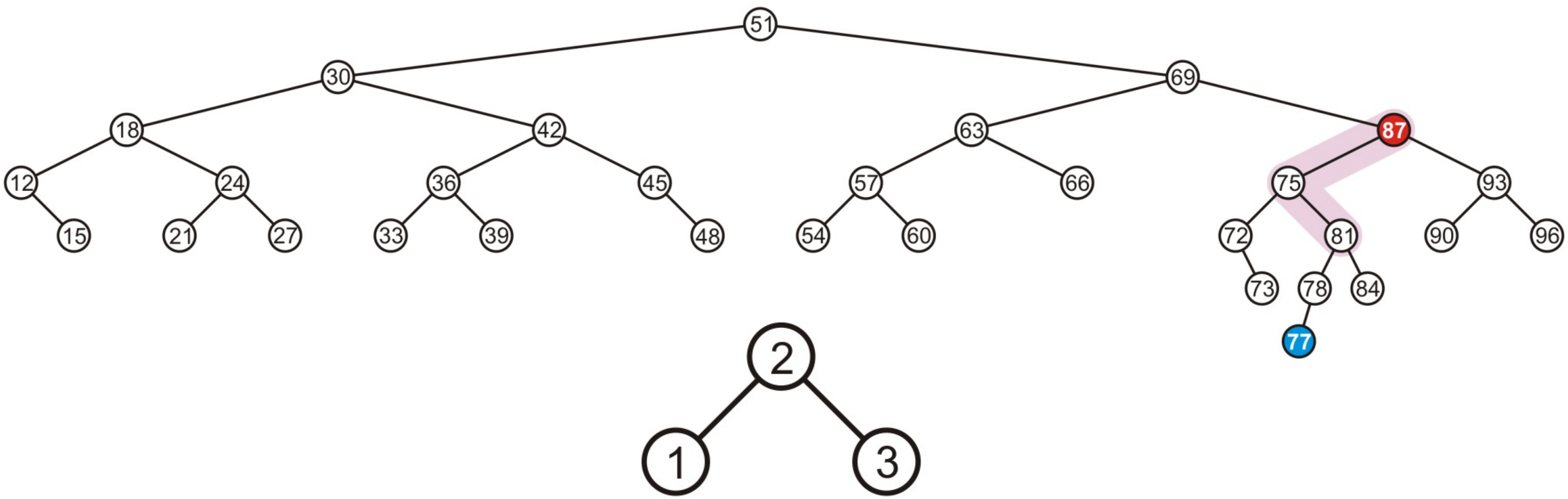
- The node 87 is unbalanced
  - A left-right imbalance
  - Promote the intermediate node to the imbalanced node





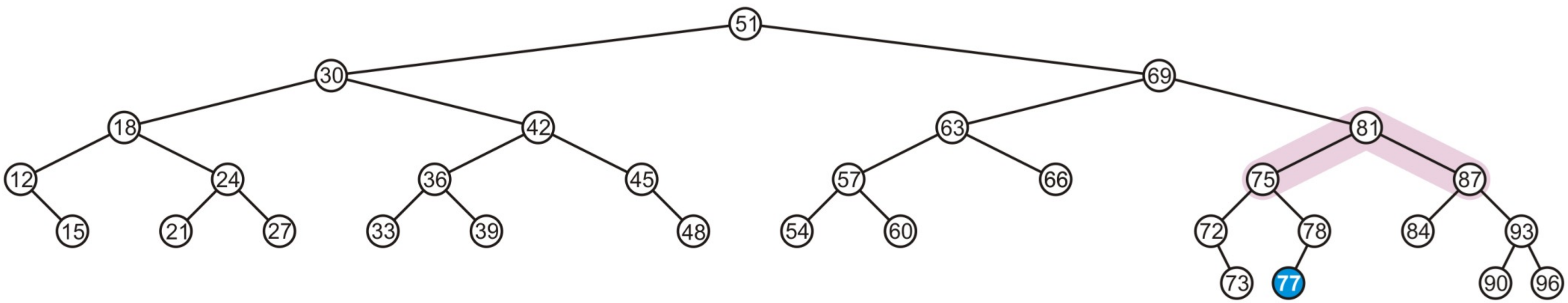
Insertion: 77

- The node 87 is unbalanced
  - A left-right imbalance
  - Promote the intermediate node to the imbalanced node
  - 81 is that value



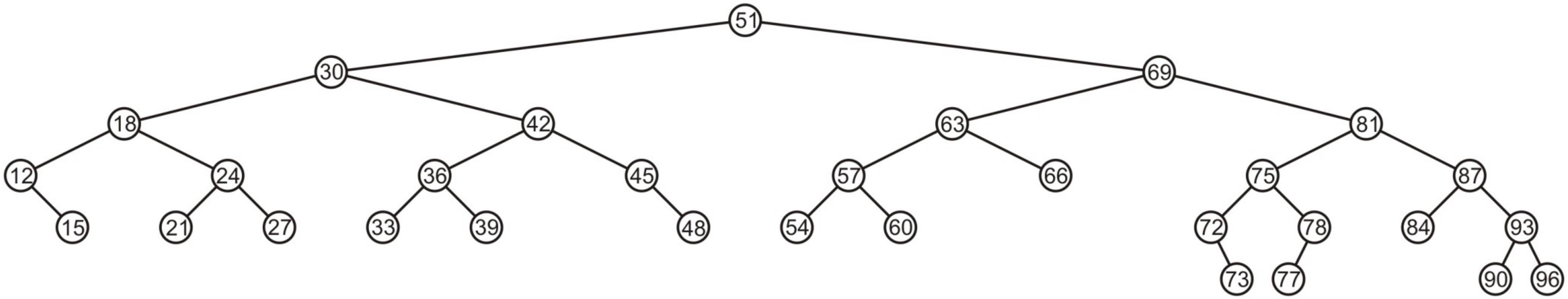
Insertion: 77

- The node 87 is unbalanced
  - A left-right imbalance
  - Promote the intermediate node to the imbalanced node
  - 81 is that value



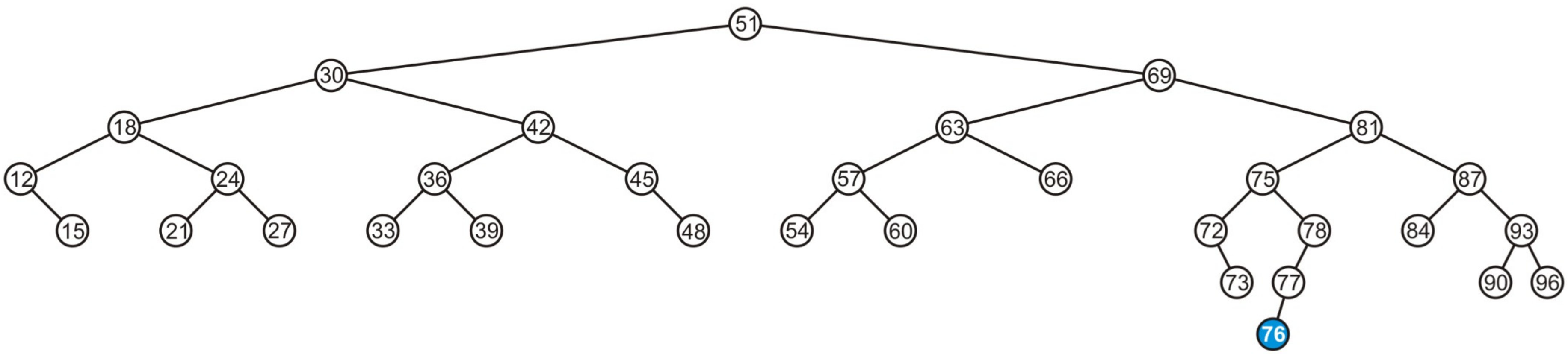
Insertion: 77

- The tree is balanced



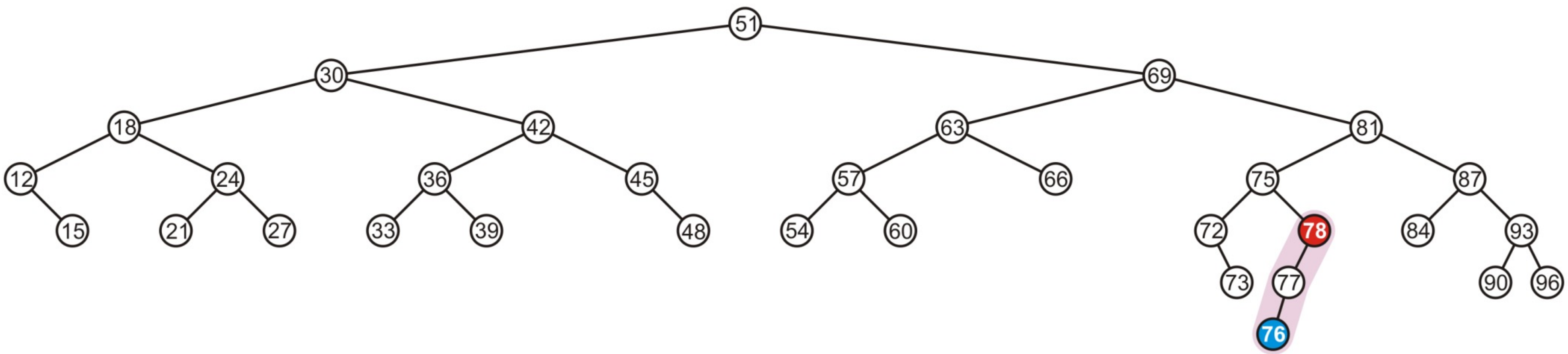
Insertion: 76

- Insert 76



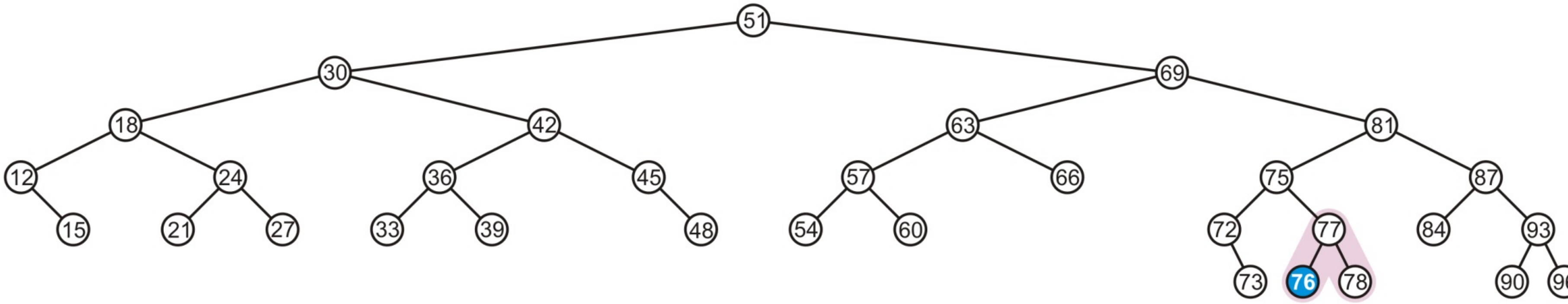
Insertion: 76

- The node 78 is unbalanced
  - A left-left imbalance



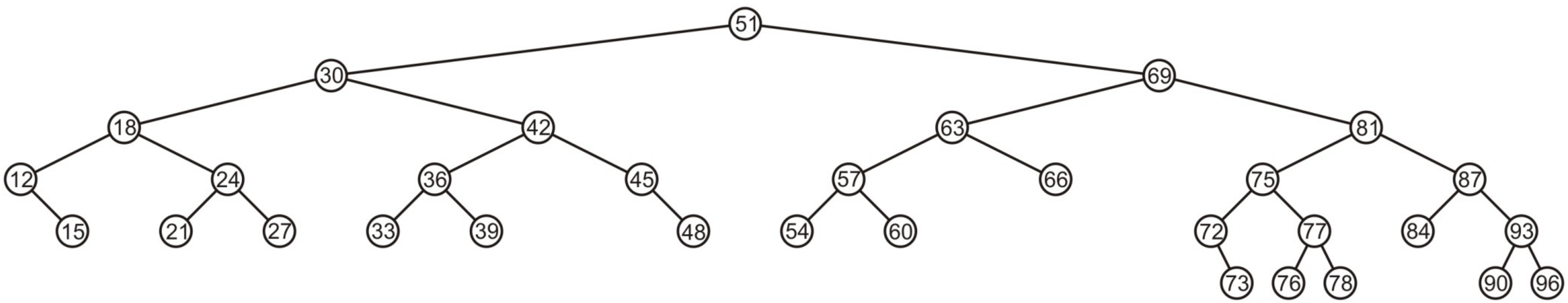
Insertion: 76

- The node 78 is unbalanced
  - Promote 77



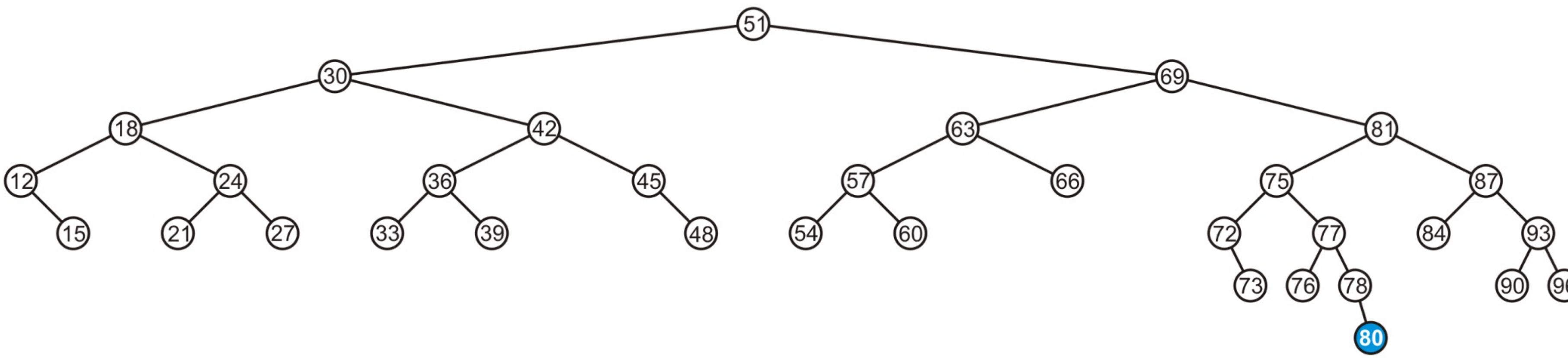
Insertion: 76

- Again, balanced



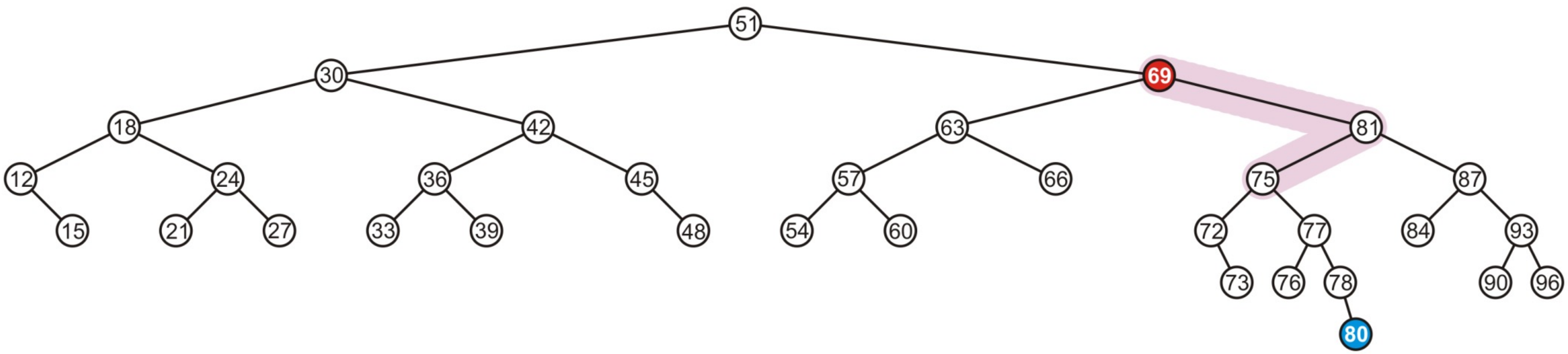
Insertion: 80

- Insert 80



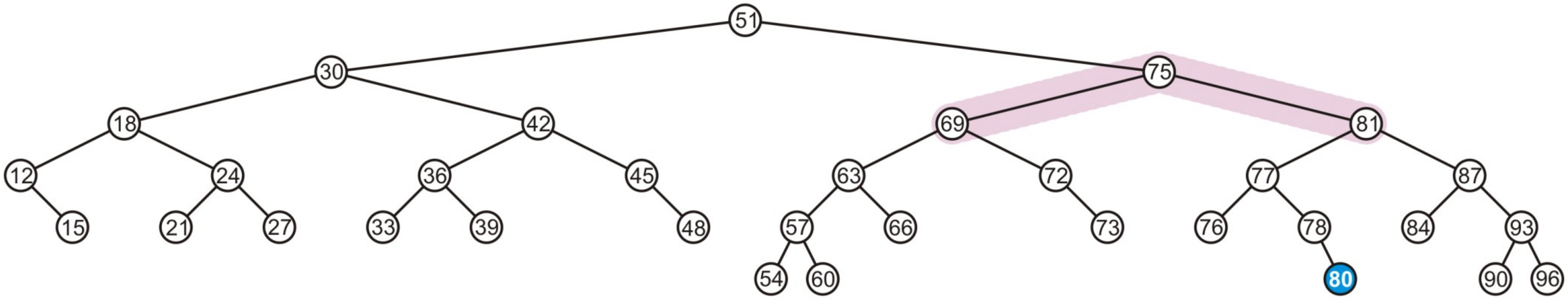
Insertion: 80

- The node 69 is unbalanced
  - A right-left imbalance
  - Promote the intermediate node to the imbalanced node



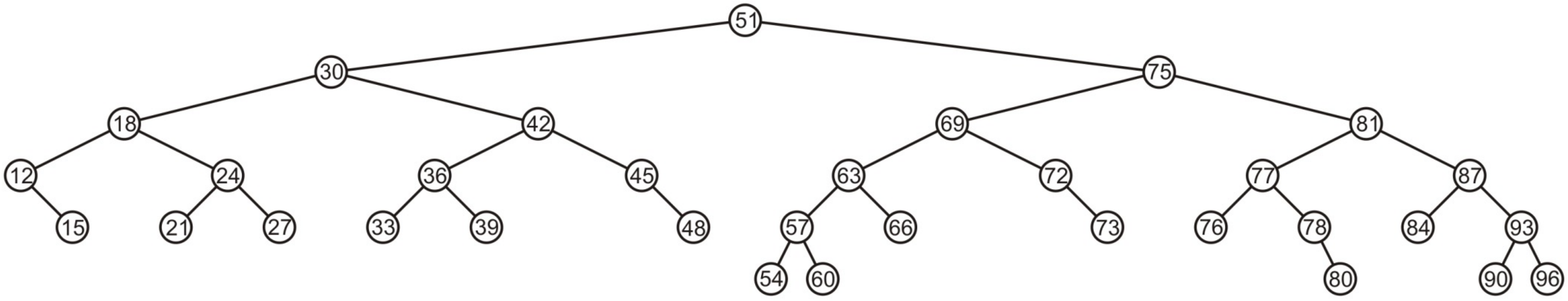
Insertion: 80

- The node 69 is unbalanced
  - A right-left imbalance
  - Promote the intermediate node to the imbalanced node
  - 75 is that value



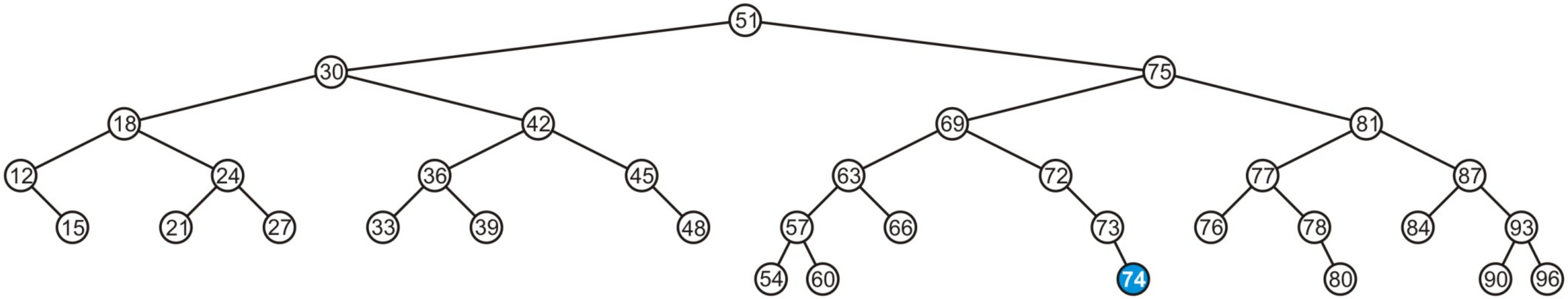
Insertion: 80

- Again, balanced



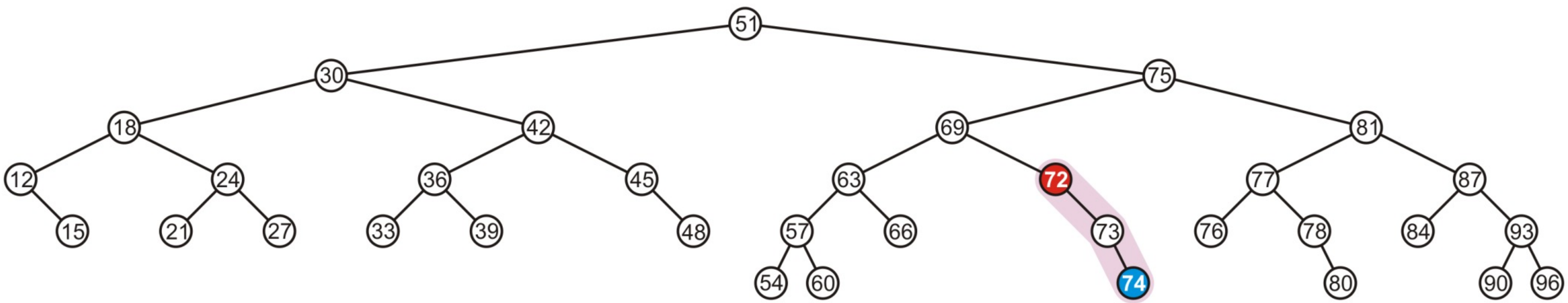
Insertion: 74

- Insert 74



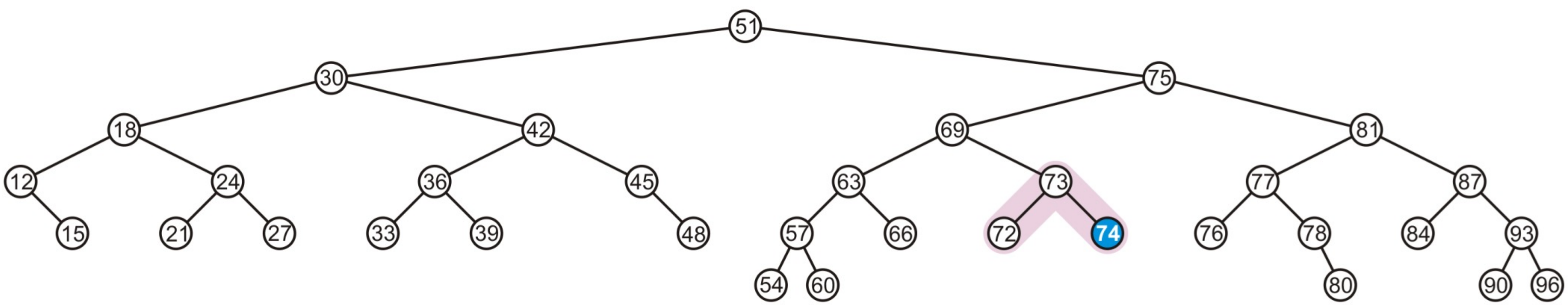
Insertion: 74

- The node 72 is unbalanced
  - A right-right imbalance
  - Promote the intermediate node to the imbalanced node
  - 73 is that value



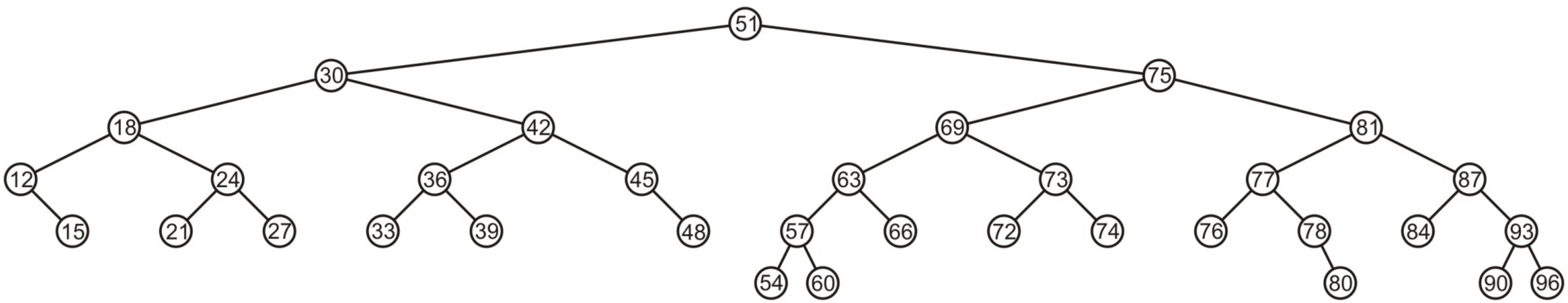
Insertion: 74

- The node 72 is unbalanced
  - A right-right imbalance
  - Promote the intermediate node to the imbalanced node



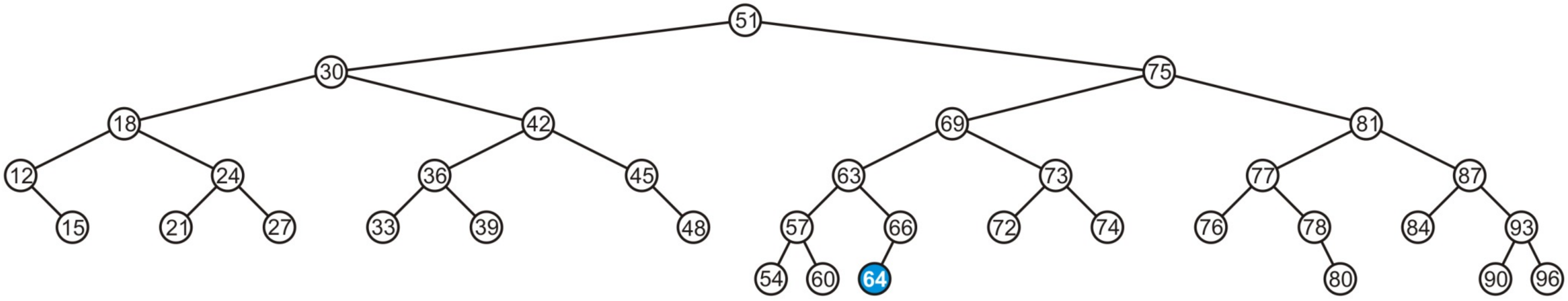
Insertion: 74

- Again, balanced



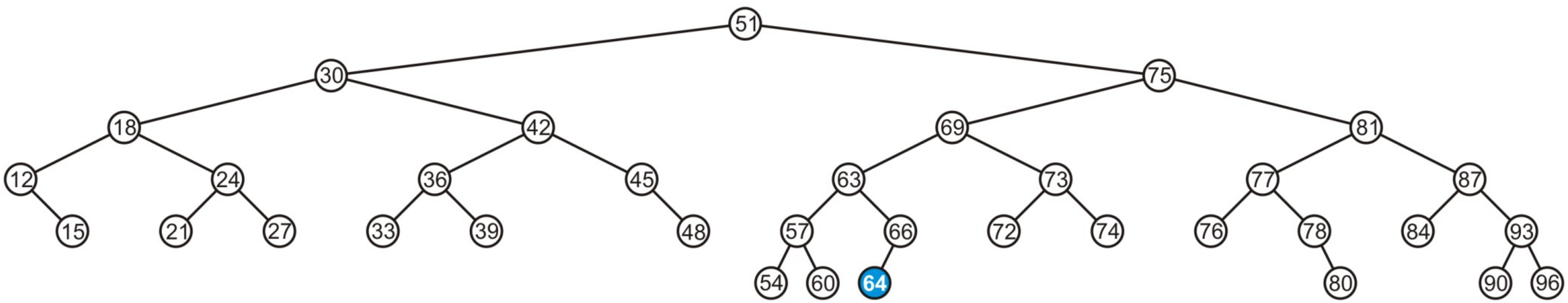
Insertion: 64

- Insert 64



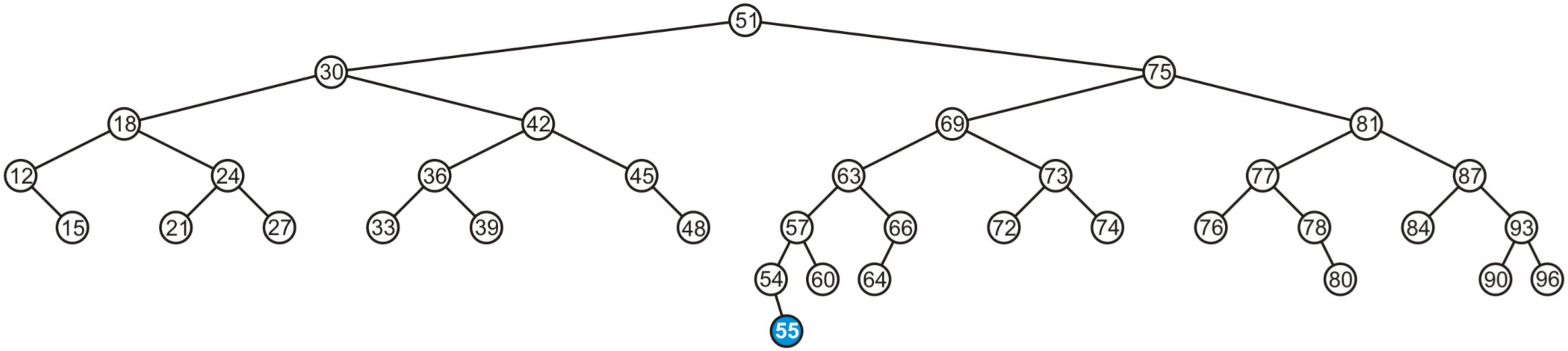
Insertion: 64

- This causes no imbalances



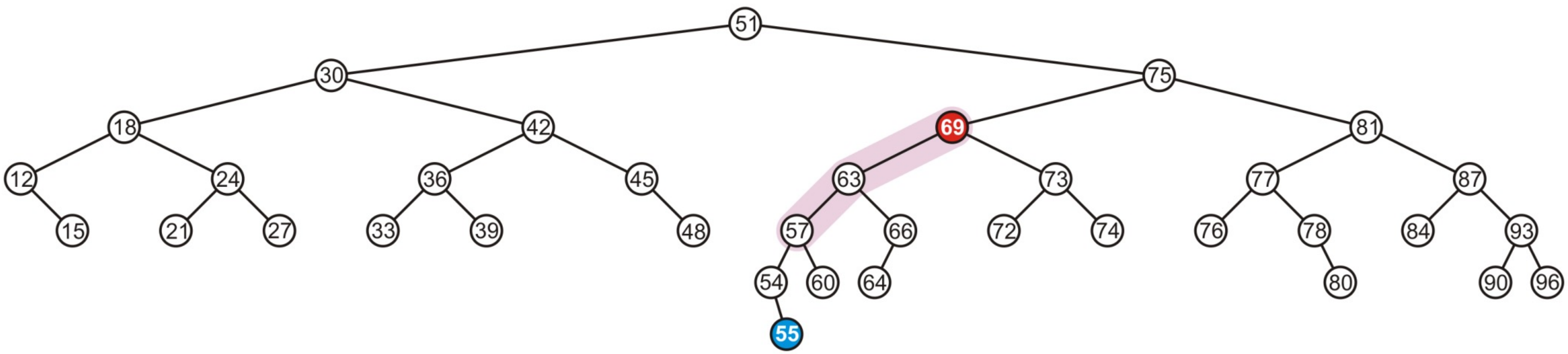
Insertion: 55

- Insert 55



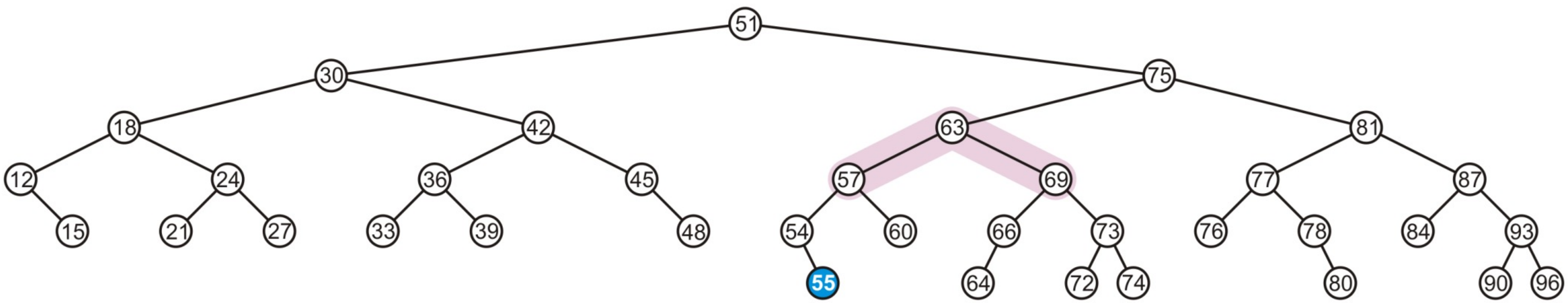
Insertion: 55

- The node 69 is imbalanced
  - A left-left imbalance
  - Promote the intermediate node to the imbalanced node



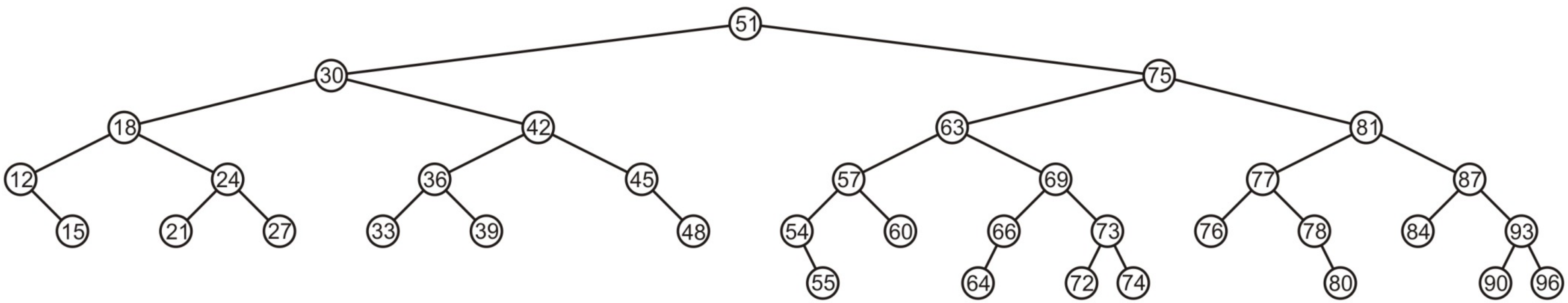
Insertion: 55

- The node 69 is imbalanced
  - A left-left imbalance
  - Promote the intermediate node to the imbalanced node
  - 63 is that value



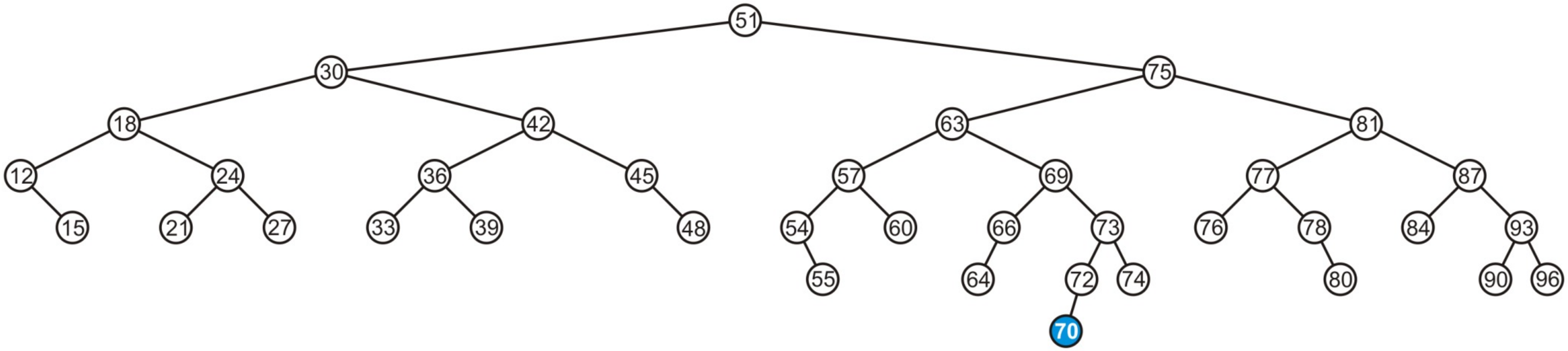
Insertion: 55

- The tree is now balanced



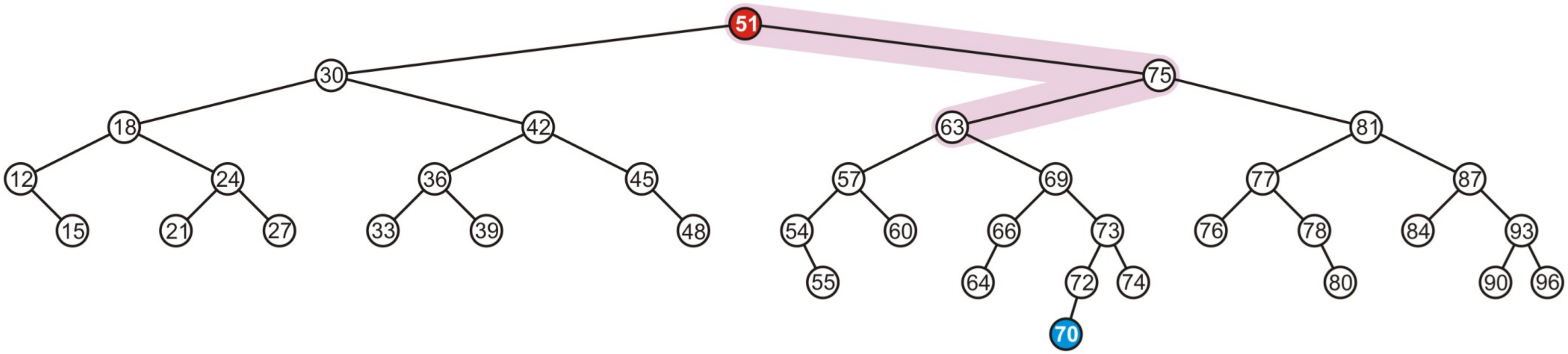
Insertion: 70

- Insert 70



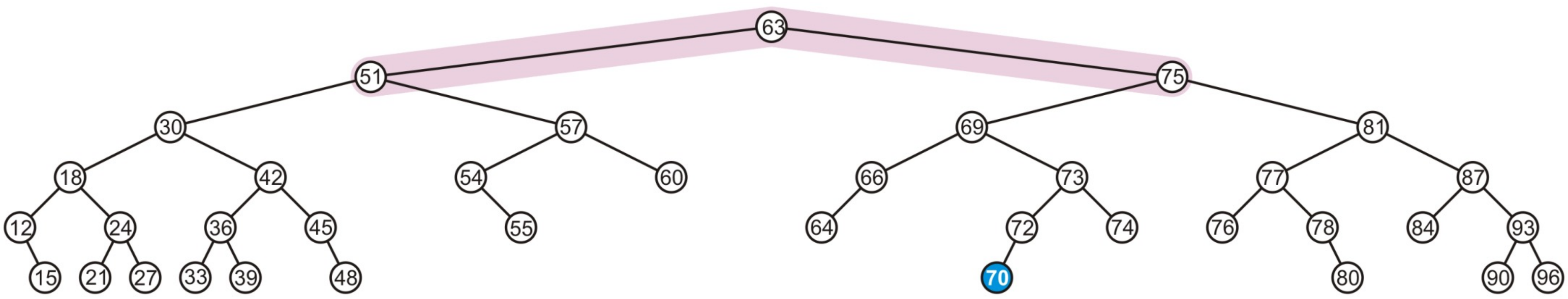
Insertion: 70

- The root node is now imbalanced
  - A right-left imbalance
  - Promote the intermediate node to the root
  - 63 is that value



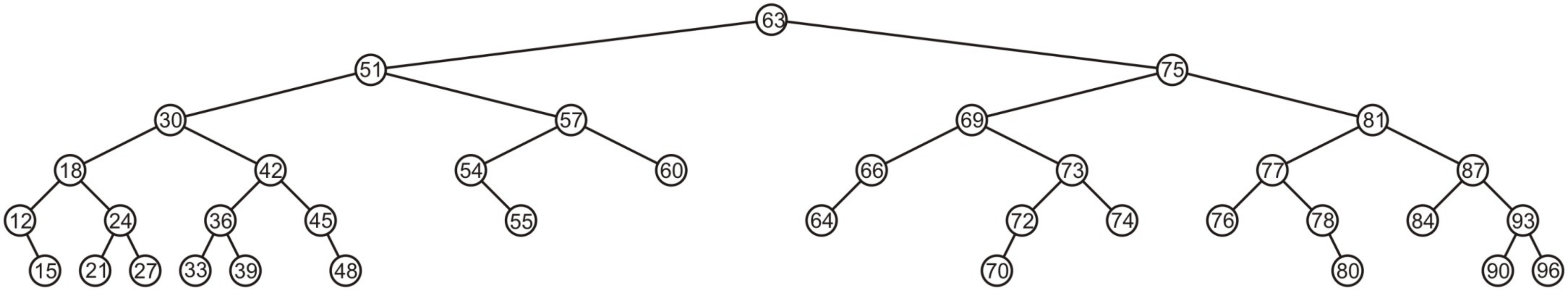
Insertion: 70

- The root node is imbalanced
  - A right-left imbalance
  - Promote the intermediate node to the root



Insertion: 70

- The result is AVL balanced

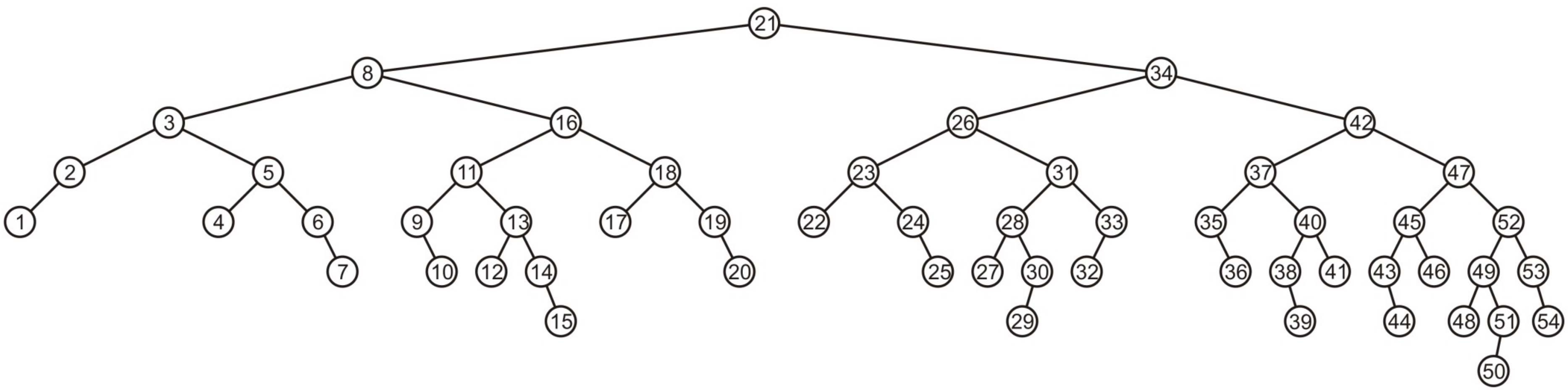


Erase

- Removing a node from an AVL tree may cause more than one AVL imbalance
  - Like insert, erase must check after it has been successfully called on a child to see if it caused an imbalance
  - Unfortunately, it may cause O(*h*) imbalances that must be corrected
    - Insertions will only cause one imbalance that must be fixed
  - Time complexity of deletion? Still O(*h*)
  - The movement of trees, however, may require that more than one node within the triplet has its height corrected

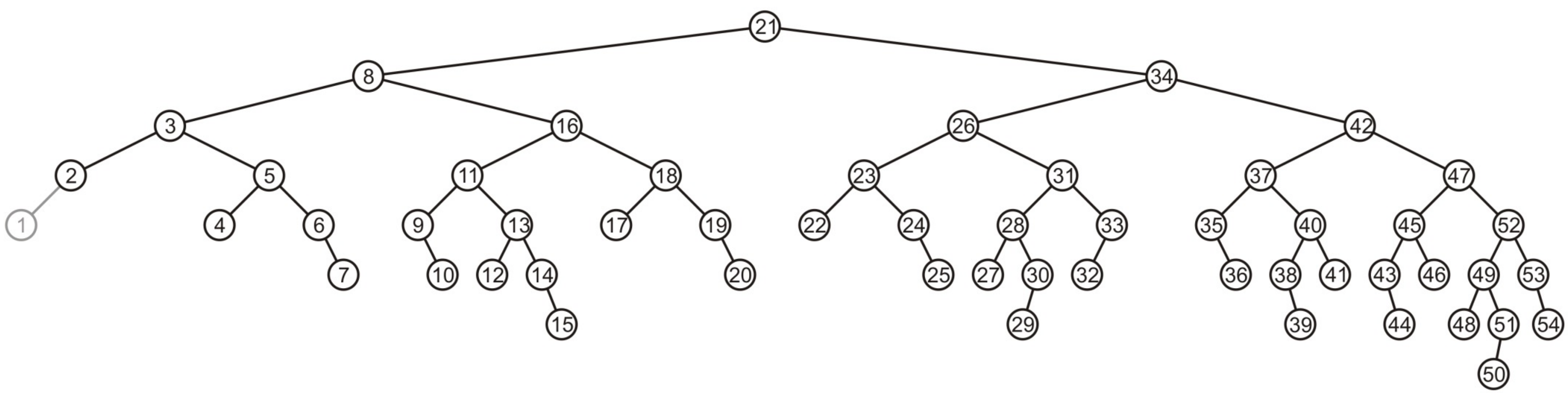
Erase

- Consider the following AVL tree



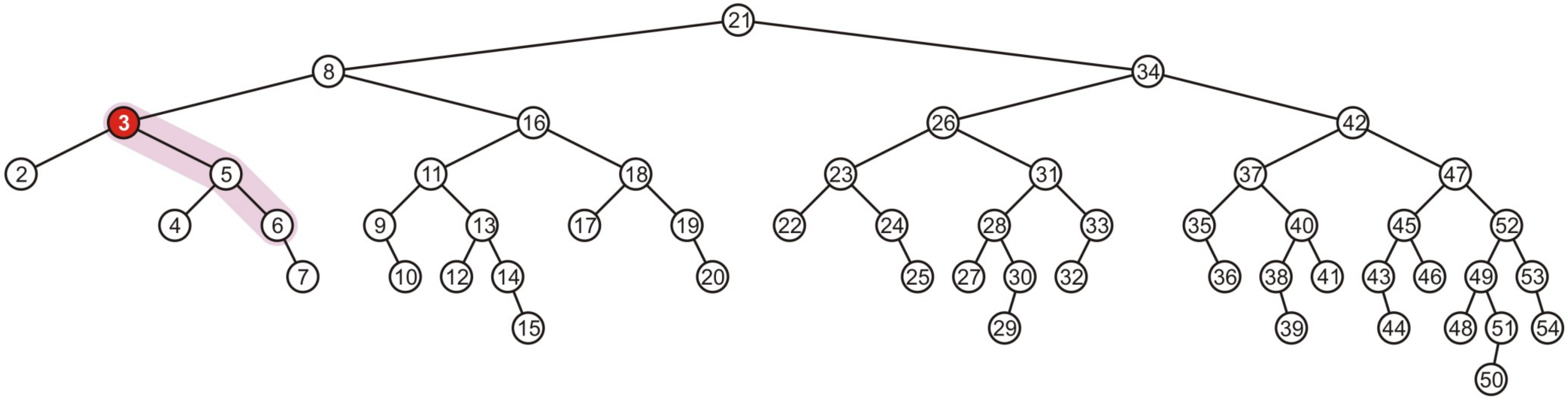
Erase: 1

- Suppose we erase the front node: 1



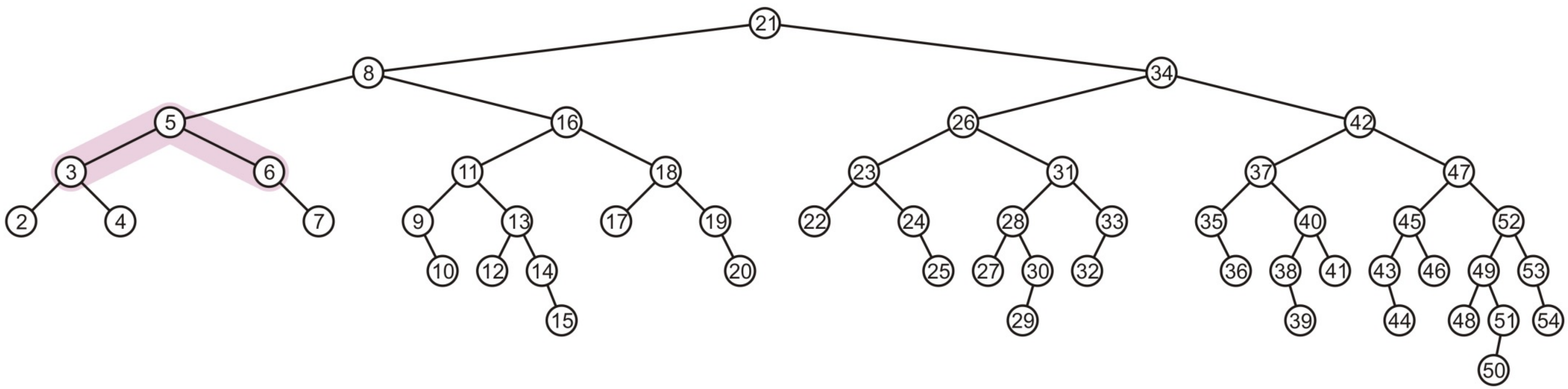
Erase: 1

- While its previous parent, 2, is not unbalanced, its grandparent 3 is
  - The imbalance is in the right-right subtree



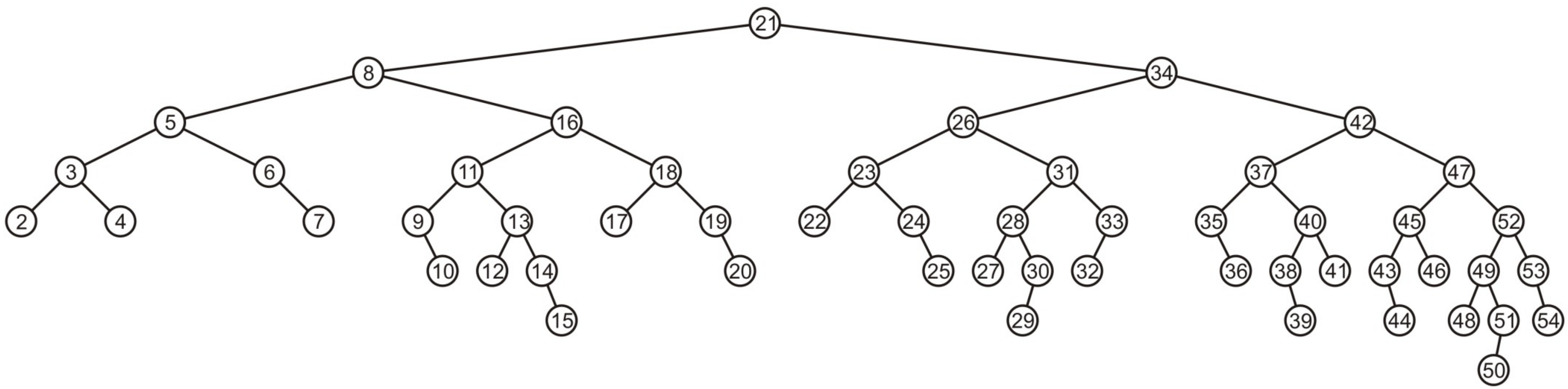
Erase: 1

- We can correct this with a simple balance



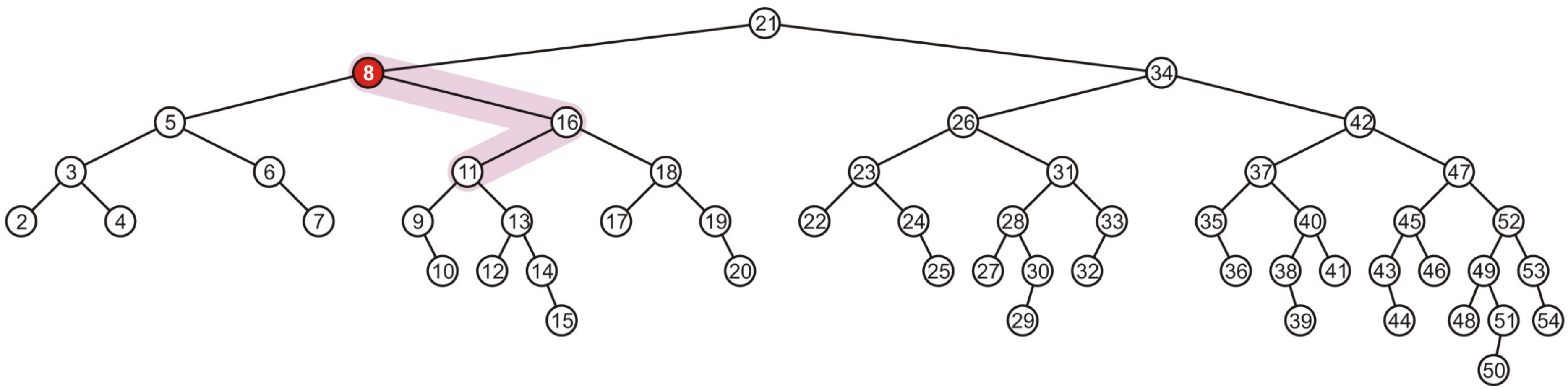
Erase: 1

- The node of that subtree, 5, is now balanced



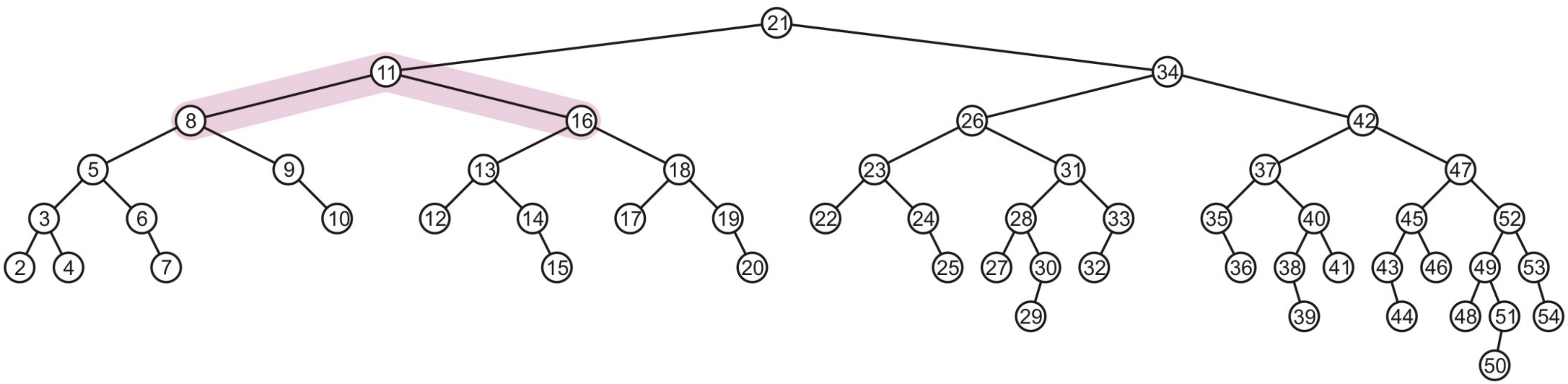
Erase: 1

- Recursing to the root, however, 8 is also unbalanced
  - This is a right-left imbalance



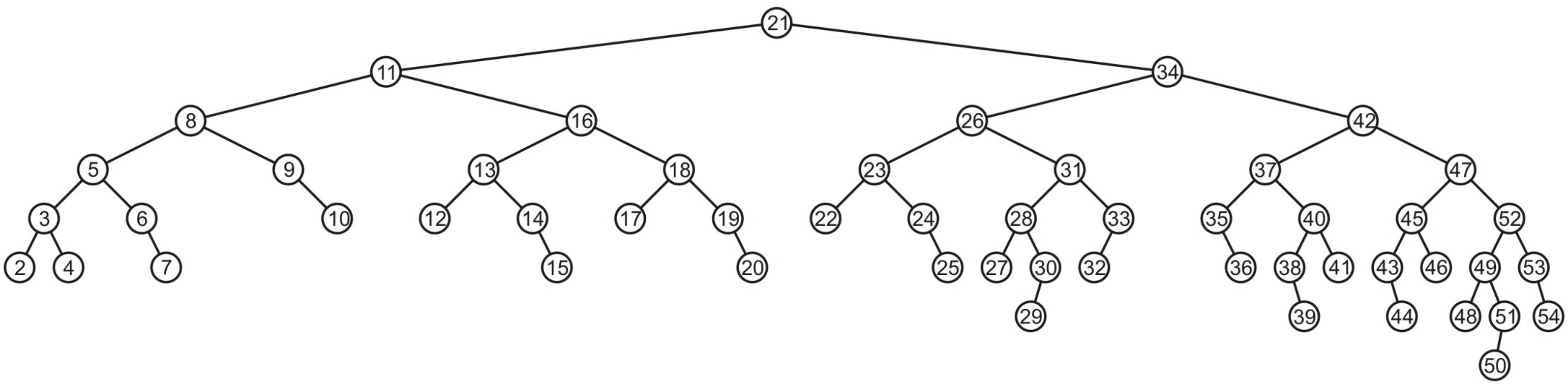
Erase: 1

- Promoting 11 to the root corrects the imbalance



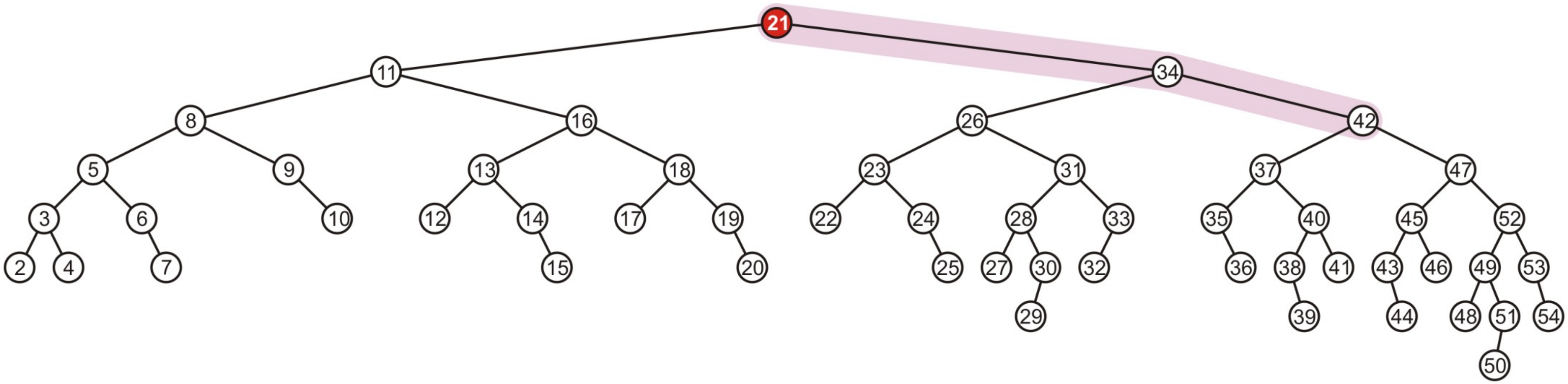
Erase: 1

- At this point, the node 11 is balanced



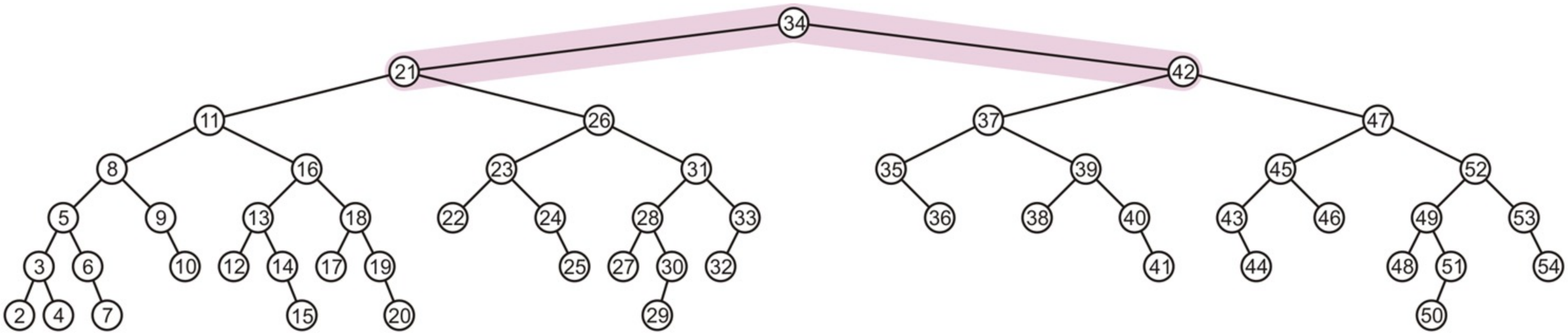
Erase: 1

- Still, the root node is unbalanced
  - This is a right-right imbalance



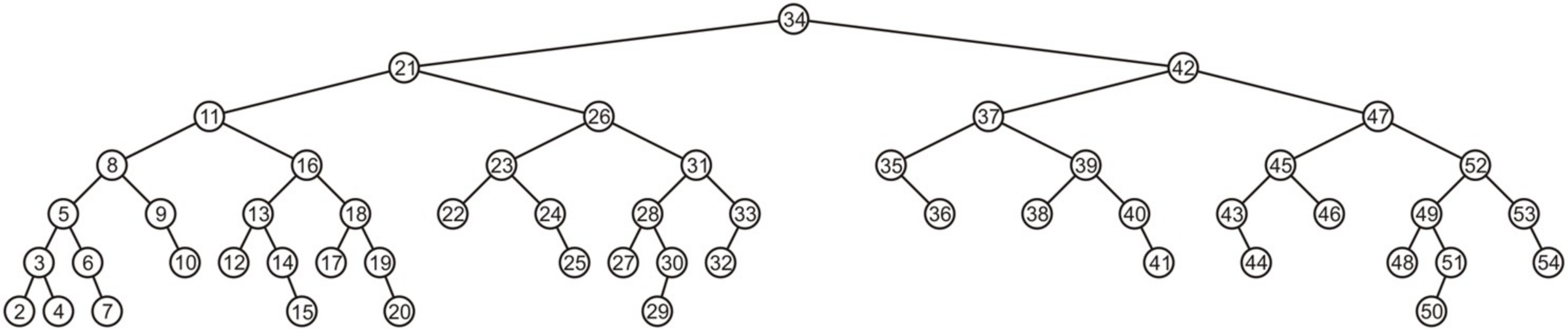
Erase: 1

- Again, a simple balance fixes the imbalance



Erase: 1

- The resulting tree is now AVL balanced



Summary

- In this topic we have covered:
  - AVL balance is defined by ensuring the difference in heights is 0 or 1
  - Insertions and erases are like binary search trees
  - Each insertion requires at least one correction to maintain AVL balance
  - Erases may require O(h) corrections
  - These corrections require Θ(1) time
  - Depth is Θ(ln(n))

  ∴ all O(h) operations are O(ln(n))

Red-Black Trees

Kyunghan Lee
Networked Computing Lab (NXC Lab)
Department of Electrical and Computer Engineering
Seoul National University
https://nxc.snu.ac.kr
kyunghanlee@snu.ac.kr

Outline

- In this topic, we will cover:
  - The idea behind a red-black tree
  - Defining balance
  - Insertions and deletions
  - The benefits of red-black trees over AVL trees

Red-Black Trees

- A red black tree "colors" each node within a tree either red or black
  - This can be represented by a single bit
  - In AVL trees, balancing restricts the difference in heights to at most one
  - For red-black trees, we have a different set of rules related to the colors of the nodes

AVL Vs. Red-Black Trees

	Average	Worst-case
Space	O(n)	O(n)
Lookup	O(log n)	O(log n)
Insert	O(log n)	O(log n)
Delete	O(log n)	O(log n)

AVL tree

	Average	Worst-case
Space	O(n)	O(n)
Lookup	O(log n)	O(log n)
Insert	O(log n)	O(log n)
Delete	O(log n)	O(log n)

Red-Black Tree

Asymptotic complexity
for lookup/insert/delete is the same!

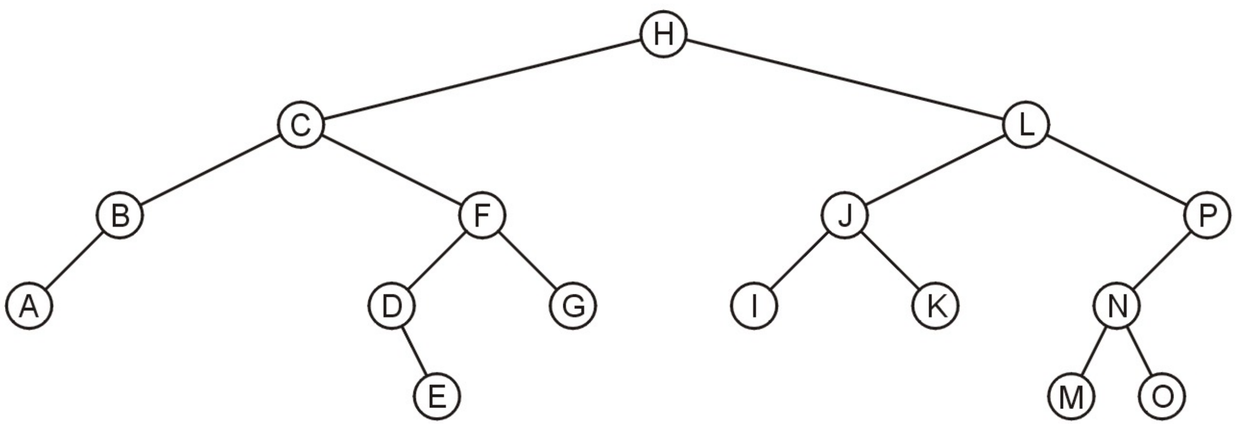
AVL Vs. Red-Black Trees

- AVL Vs. RBTree
  - AVL maintains its balance more tight than RBTree
    - Recall the definition
  - AVL performs better for lookup-intensive applications
  - RBTree provides faster worst-case performance for insert/delete

```
To quote Linux Weekly News:

    There are a number of red-black trees in use in the kernel.
    The deadline and CFQ I/O schedulers employ rbtrees to
    track requests; the packet CD/DVD driver does the same.
    The high-resolution timer code uses an rbtree to organize outstanding
    timer requests.  The ext3 filesystem tracks directory entries in a
    red-black tree.  Virtual memory areas (VMAs) are tracked with red-black
    trees, as are epoll file descriptors, cryptographic keys, and network
    packets in the "hierarchical token bucket" scheduler.
```

Red-Black Trees

- Define a *null path* within a binary tree as any path starting from the root where the last node is not a full node
  - Consider the following binary tree:



Red-Black Trees

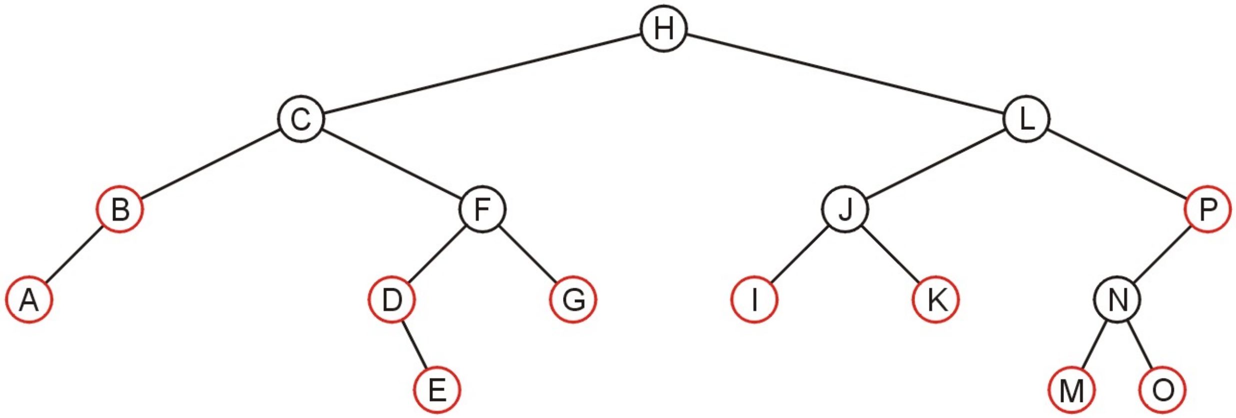
- All null paths include:

(H, C, B)
(H, C, B, A)

(H, C, F, D)
(H, C, F, D, E)
(H, C, F, G)

(H, L, J, I)
(H, L, J, K)

(H, L, P)
(H, L, P, N, M)
(H, L, P, N, O)

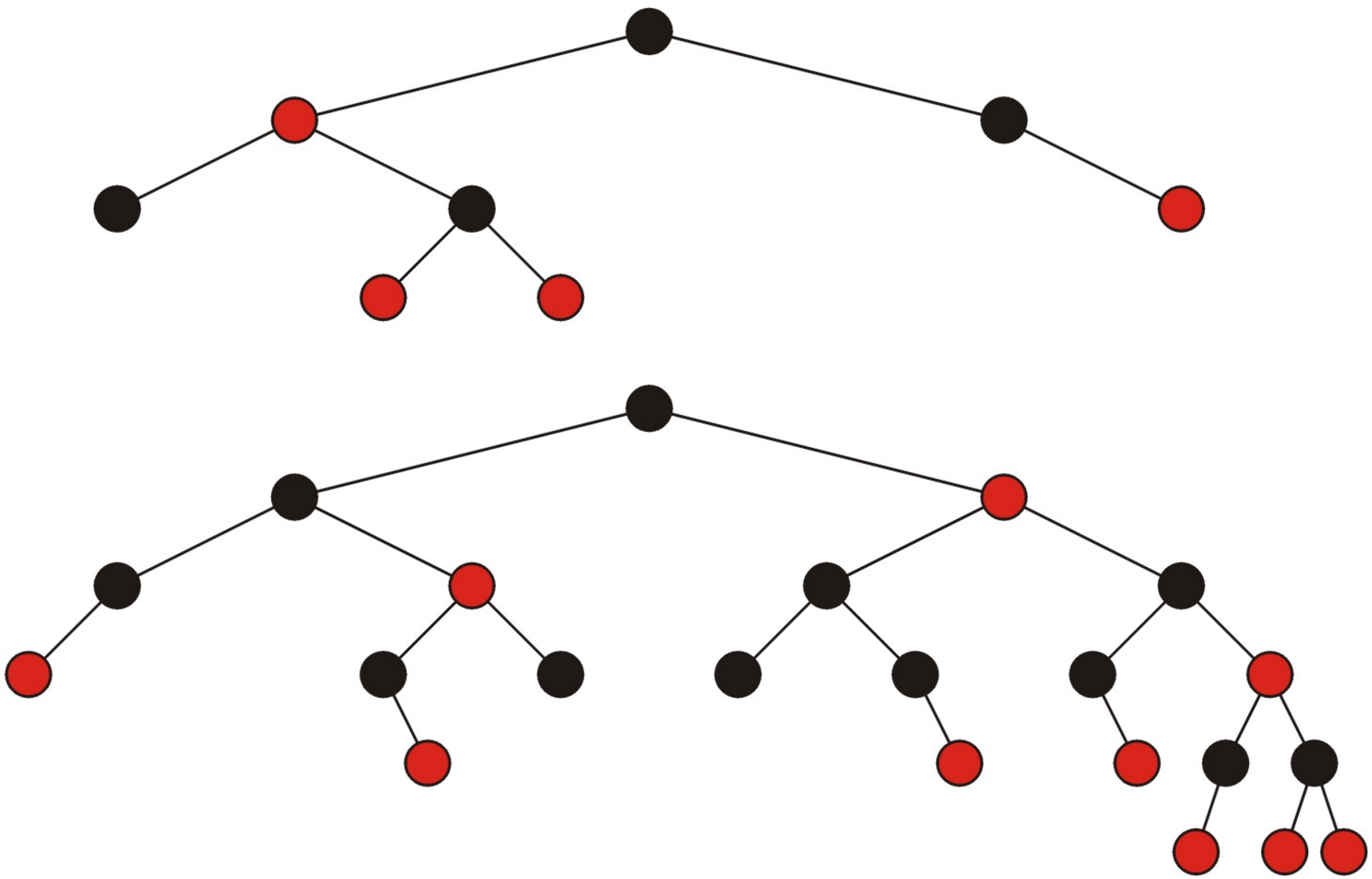


Red-Black Trees

- The three rules which define a red-black tree are
  1. The root must be black,
  2. If a node is red, its children must be black,
  3. Each null path must have the same number of black nodes

Red-Black Trees

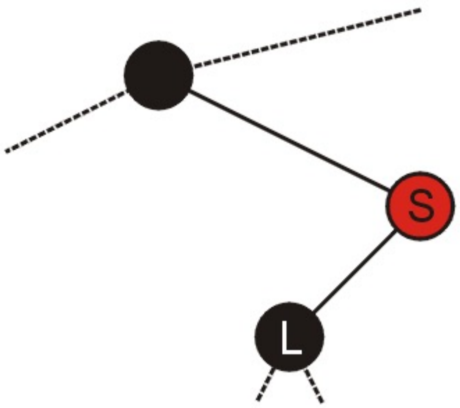
- These are two examples of red-black trees:



Red-Black Trees

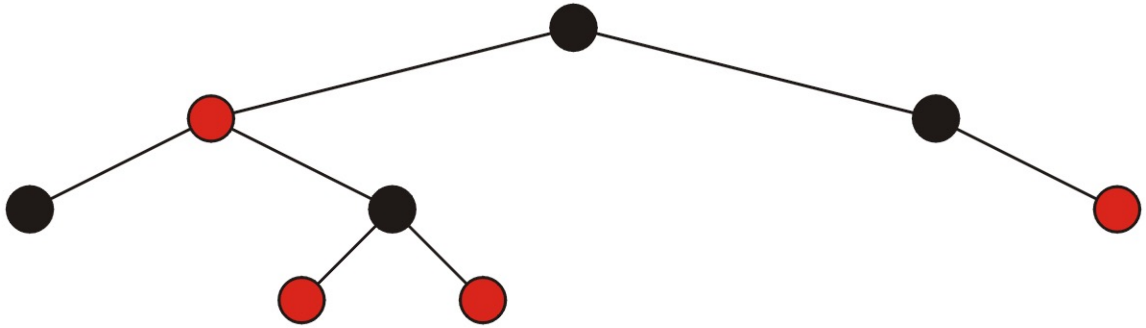
- Theorem:
  - Every red node must be either
    - A full node (with two black children), or
    - A leaf node

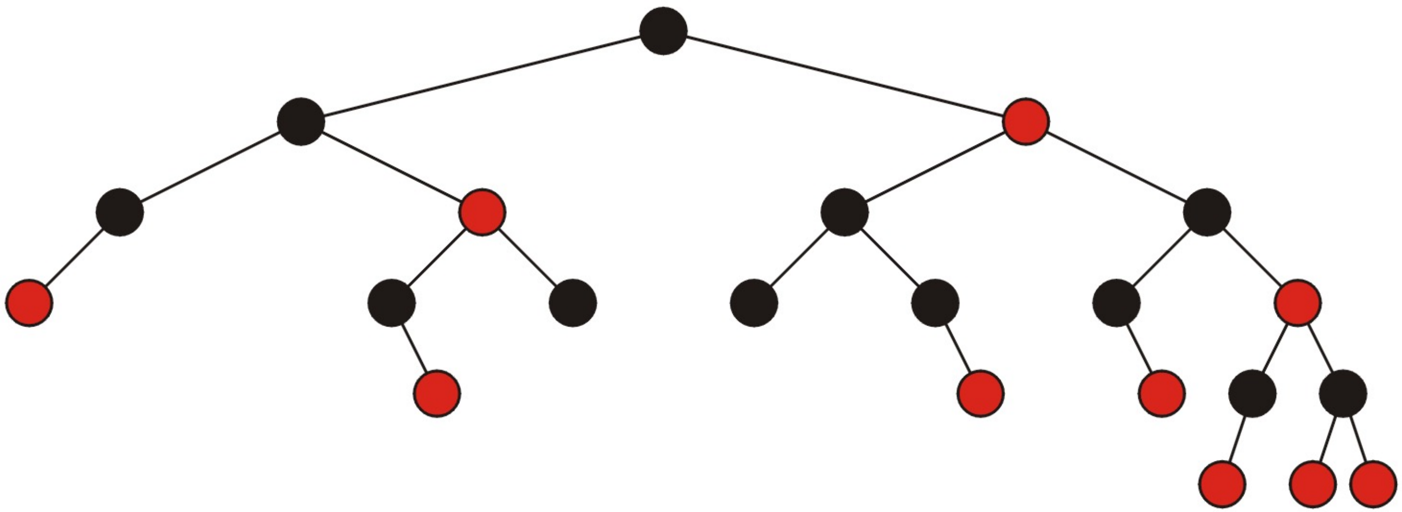
- Proof by contradiction:
  - Suppose node S has one child:
    - The one child L must be black
    - The null path ending at S has k black nodes
    - Any null path containing the node L will therefore have at least $k + 1$ black nodes



Red-Black Trees

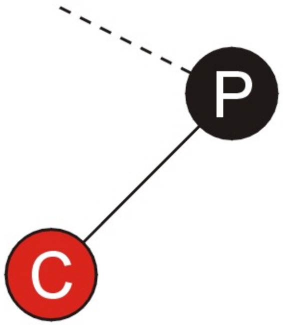
- In our two examples, you will note that all red nodes are either full or leaf nodes





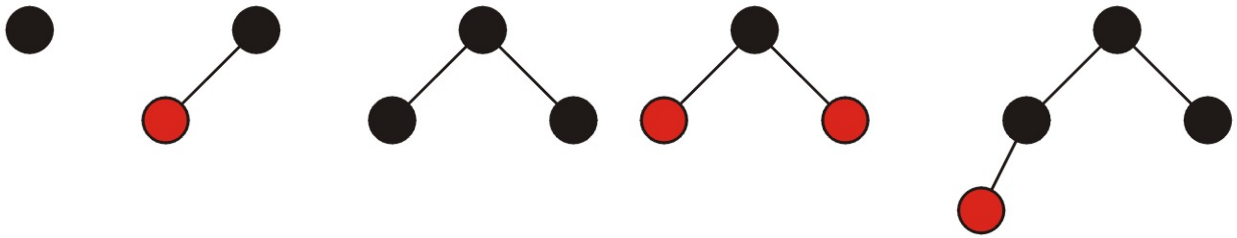
Red-Black Trees

- Another consequence is that if a node P has exactly one child:
  - The one child must be red,
  - The one child must be a leaf node, and
  - The node P must be black



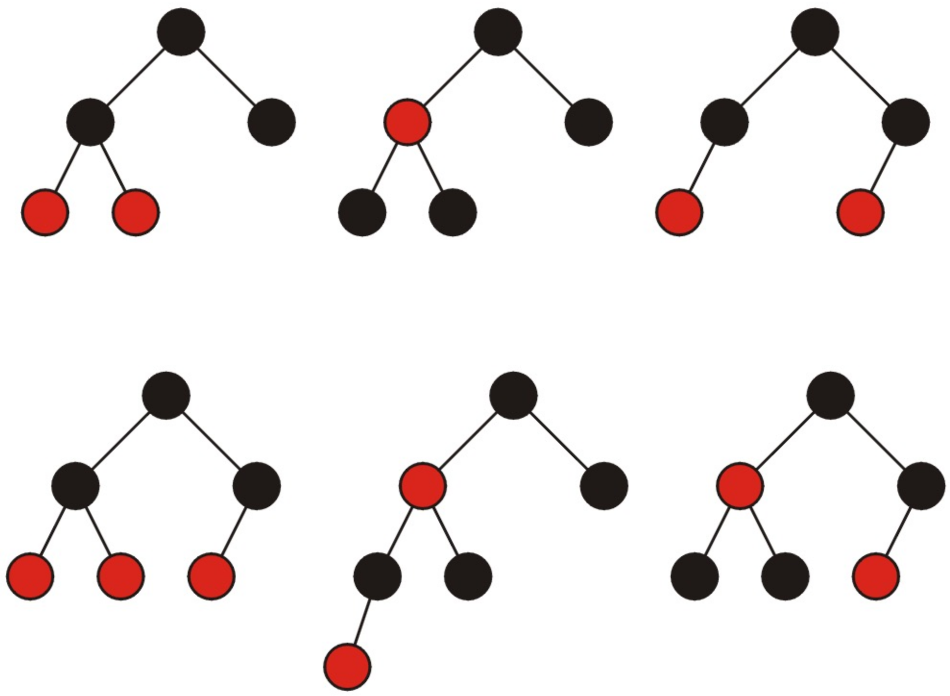
Red-Black Trees

- All red-black trees with 1, 2, 3, and 4 nodes:



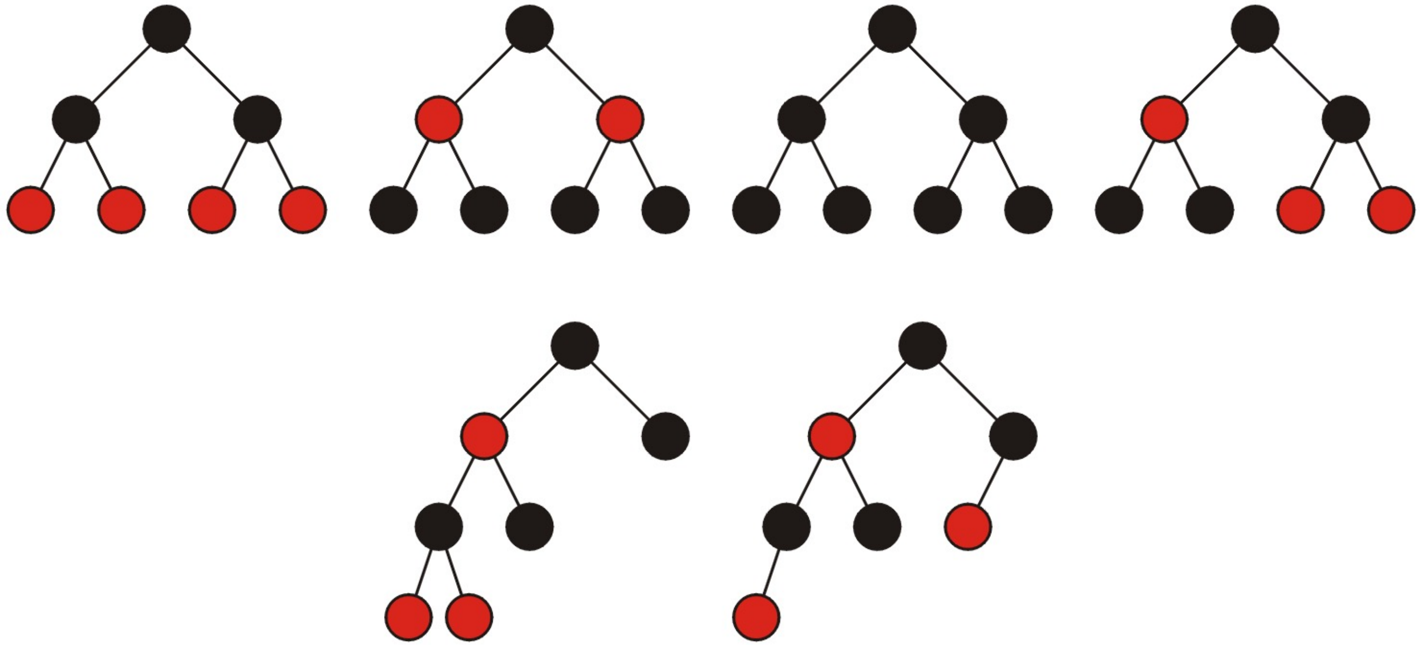
Red-Black Trees

- All red-black trees with 5 and 6 nodes:



Red-Black Trees

- All red-black trees with seven nodes—most are shallow:



Red-Black Trees

- Every perfect tree is a red-black tree if each node is colored black

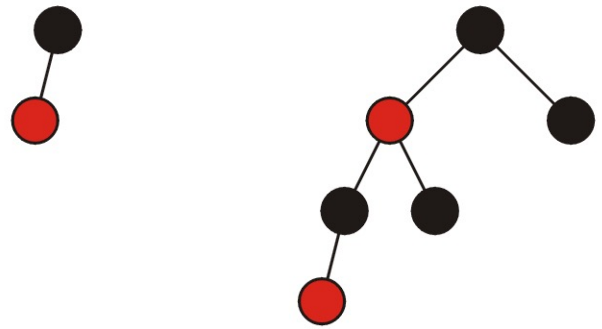
- A complete tree is a red-black tree if:
  - each node at the lowest depth is colored red, and
  - all other nodes are colored black

- What is the worst case?

Red-Black Trees

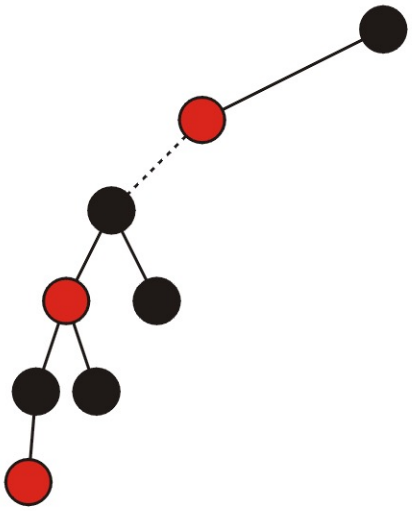
- Any worst-case red-black tree must have an alternating red-black pattern down one side

- The following are the worst-case red-black trees with 1 and 2 black nodes per null path (*i.e.*, heights 1 and 3)



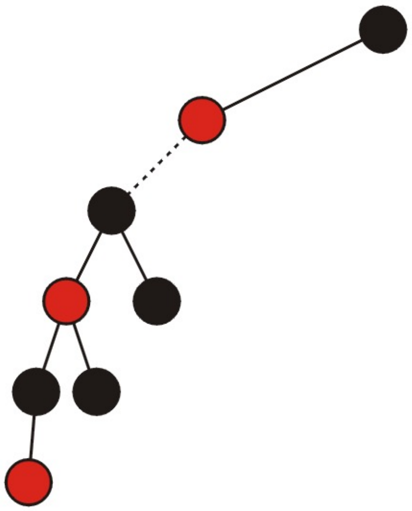
Red-Black Trees

- To create the worst-case for paths with 3 black nodes per path, start with a black and red node and add the previous worst-case for paths with 2 nodes



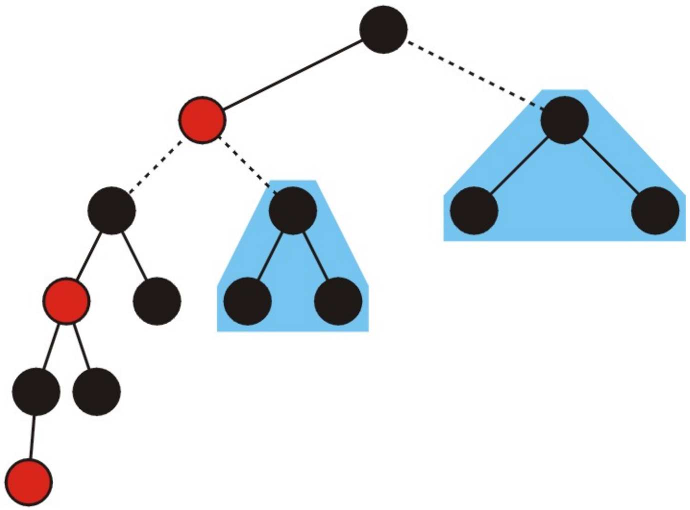
Red-Black Trees

- This, however, is not a red-black tree because the two top nodes do not have paths with three black nodes
  - To solve this, add the optimal red-black trees with two black nodes per path



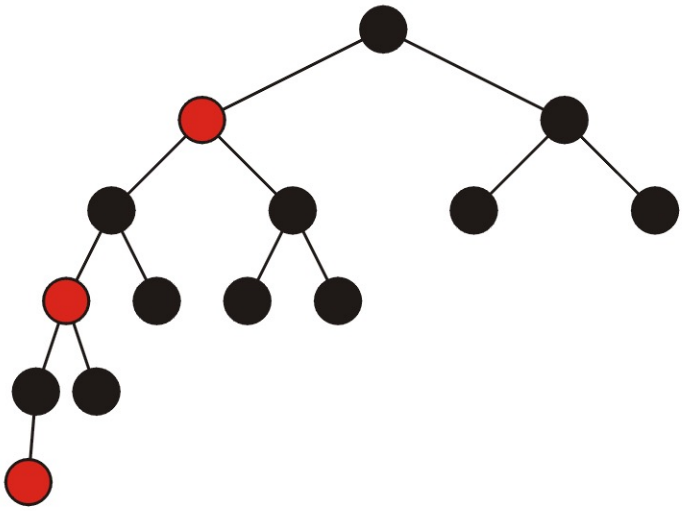
Red-Black Trees

- That is, add two perfect trees with height 1 to each of the missing sub-trees



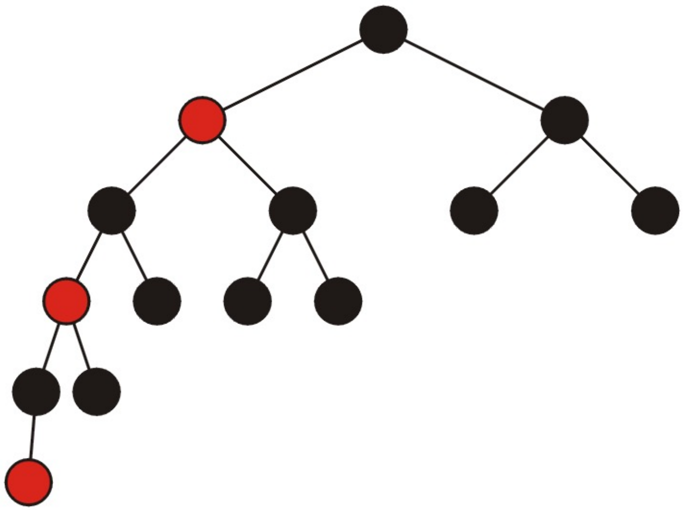
Red-Black Trees

- Thus, we have the worst-case for a red-black tree with three black nodes per path (or a red-black tree of height 5)



Red-Black Trees

- Note that the left sub-tree of the root has height 4 while the right has height 1
  - Thus, suggests that AVL trees may be better in maintaining "height balance"



Revisit: Red-Black Trees vs. AVL

- AVL trees are not as deep in the worst case as are red-black trees
  - Therefore, AVL trees will perform better when numerous searches are being performed,
  - However, insertions and deletions will require:
    - more rotations with AVL trees, and
    - require recursions from and back to the root
  - Thus, AVL trees will perform worse in situations where there are numerous insertions and deletions

Insertions

- We will consider two types of insertions:
  - bottom-up (insertion at the leaves), and
  - top-down (insertion at the root)

- The first will be instructional and we will use it to derive the second case

Bottom-Up Insertions

- After an insertion is performed, we must satisfy all the rules of a red-black tree:
  - #1. The root must be black,
  - #2. If a node is red, its children must be black, and
  - #3. Each path from a node to any of its descendants which are not a full node (*i.e.*, two children) must have the same number of black nodes

- #1 and #2 are local: they affect a node and its neighbors

- #3 is global: adding a new black node anywhere will cause all of its ancestors to become unbalanced

Bottom-Up Insertions

- Thus, when we add a new node, we will add a red node
  - Which breaks the local rule
  - But not breaking the global rule

- We will then travel up the tree to the root, while fixing the requirement #1 and #2

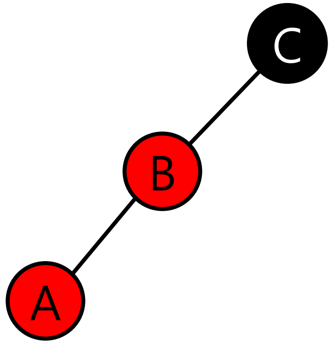
Bottom-Up Insertions

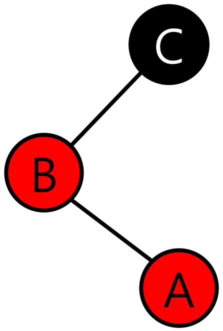
- If the parent of the inserted node is already black, we are done
  - Otherwise, we must correct the problem

- We will fix by following two steps:
  - Step #1) the initial insertion, and
  - Step #2) the recursive steps back to the root

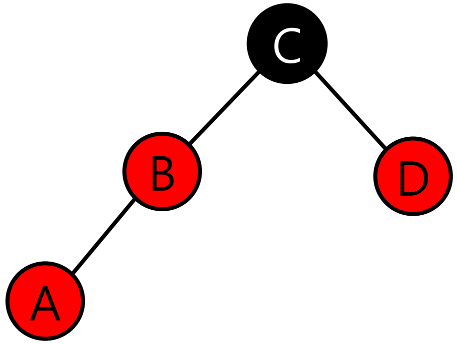
Bottom-Up Insertions: Step #1. Initial insertion

- For the initial insertion, there are two possible cases:
  - Case #1: the grandparent has one red child, or



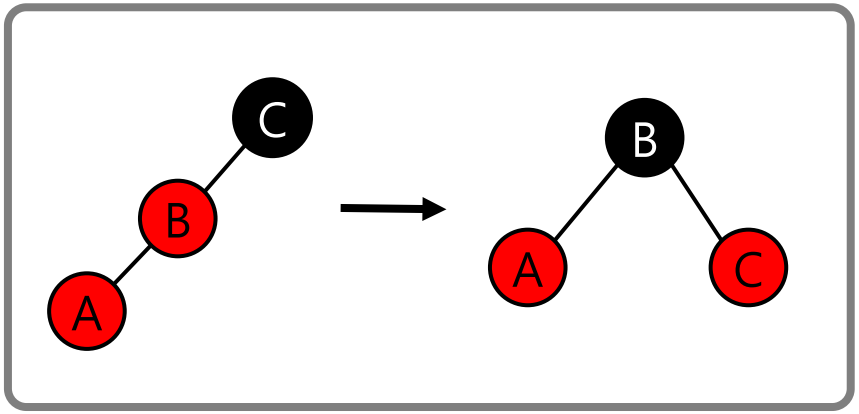


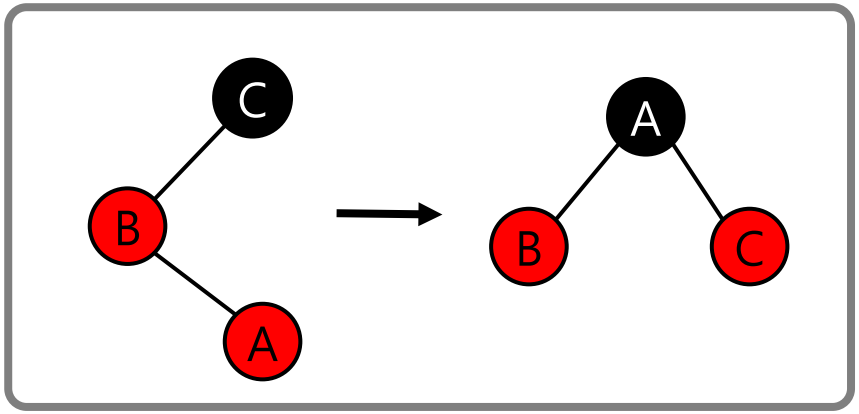
  - Case #2: the grandparent has two red children



Bottom-Up Insertions: Step #1. Initial insertion

- Case #1 can be fixed with a rotation.
  - Example: Inserting A

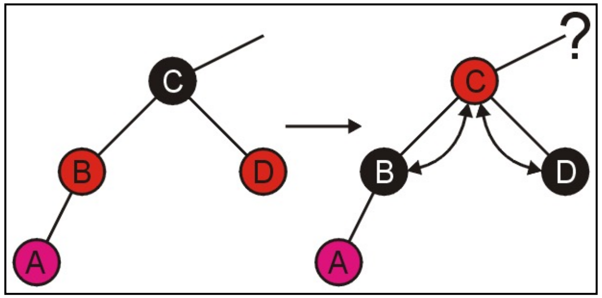




Consequently, we are finished...

Bottom-Up Insertions:
Step #1. Initial insertion

- Case #2 seems to be fixed by just swapping the colors:



- However, we now may cause a problem between the parent and the grandparent....

Bottom-Up Insertions:
Step #2. Recursive step back

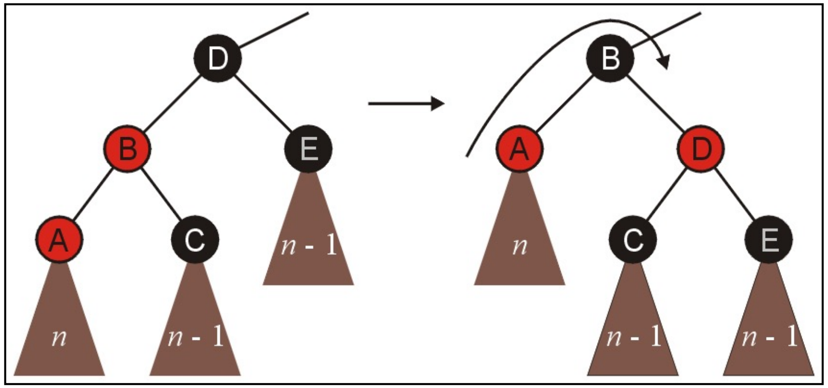
- Fortunately, dealing with problems caused within the tree are identical to the problems at the leaf nodes

- Like before, there are two cases:
  - the grandparent has one red child, or
  - the grandparent has two red children

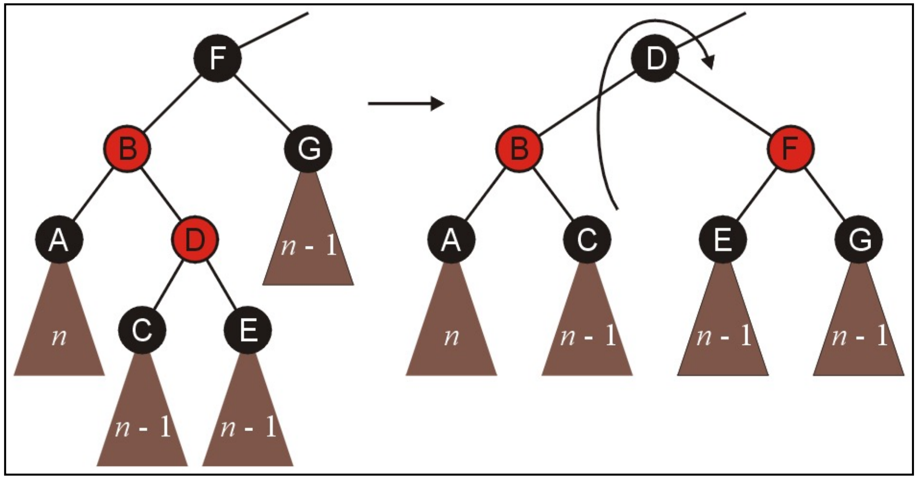
Bottom-Up Insertions: Step #2. Recursive step back

- Suppose that A and D, respectively were swapped.

- If the grand parent had one red child (Case #1), we perform similar rotations as we have done before.



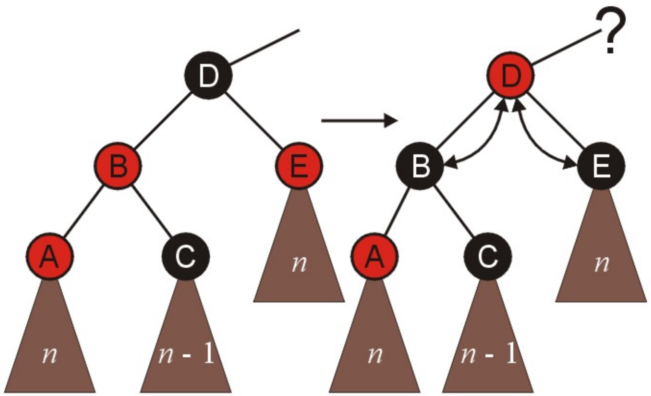
A was swapped, and the grand parent (D) has only one red child (B)



D was swapped, and the grand parent (F) has only one red child (B)

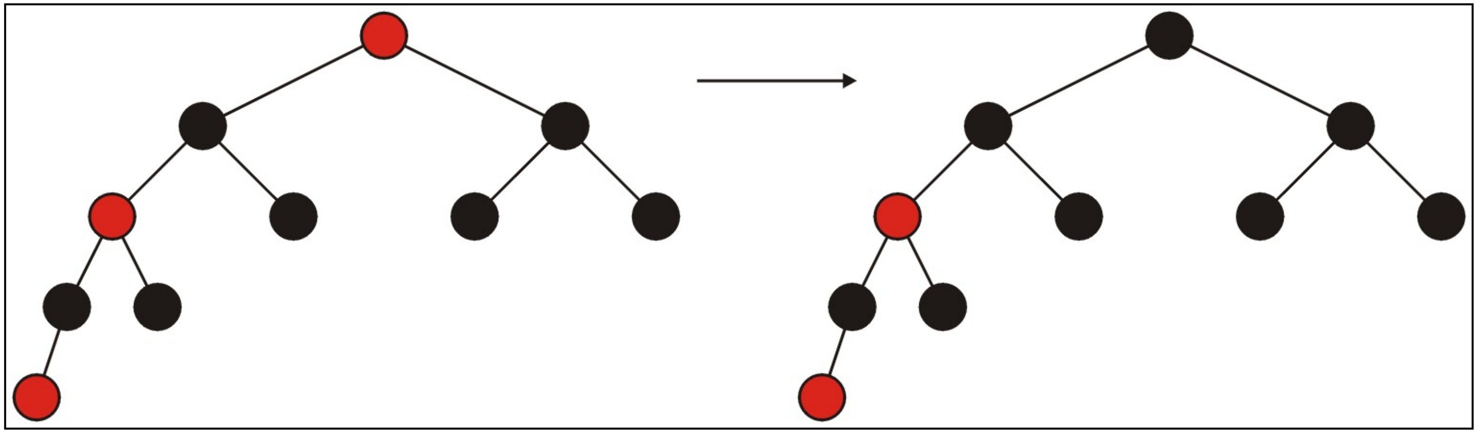
Bottom-Up Insertions:
Step #2. Recursive step back

- If both children of the grandparent are red (Case #2), we swap colors, and recurs back to the root



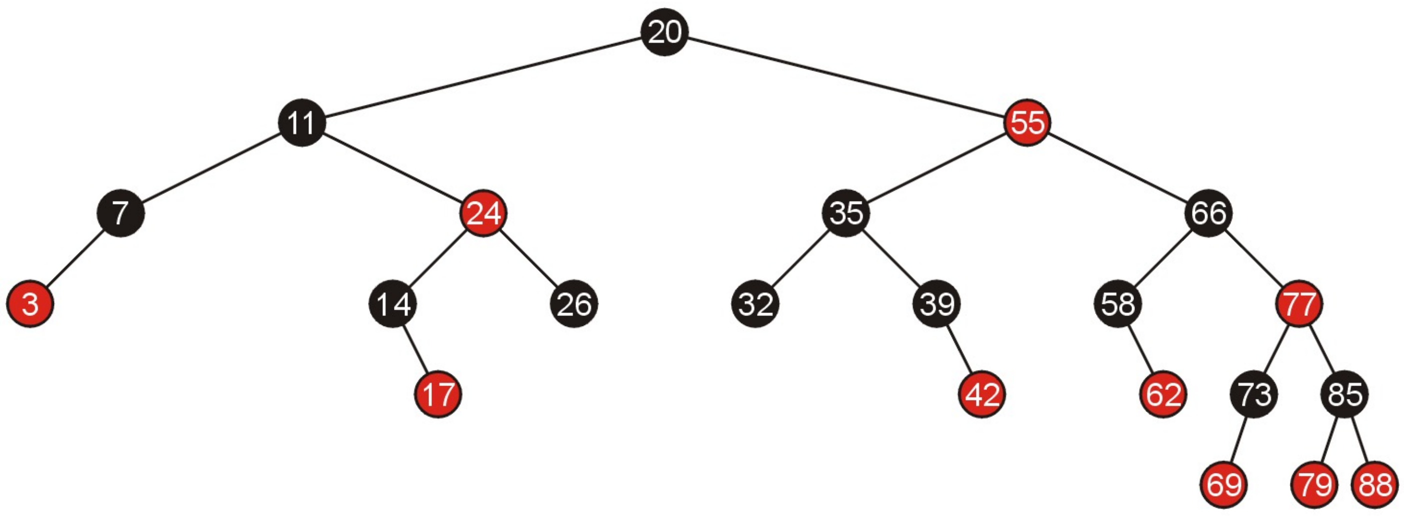
Bottom-Up Insertions: Step #2. Recursive step back

- If, at the end, the root is red, it can be colored black



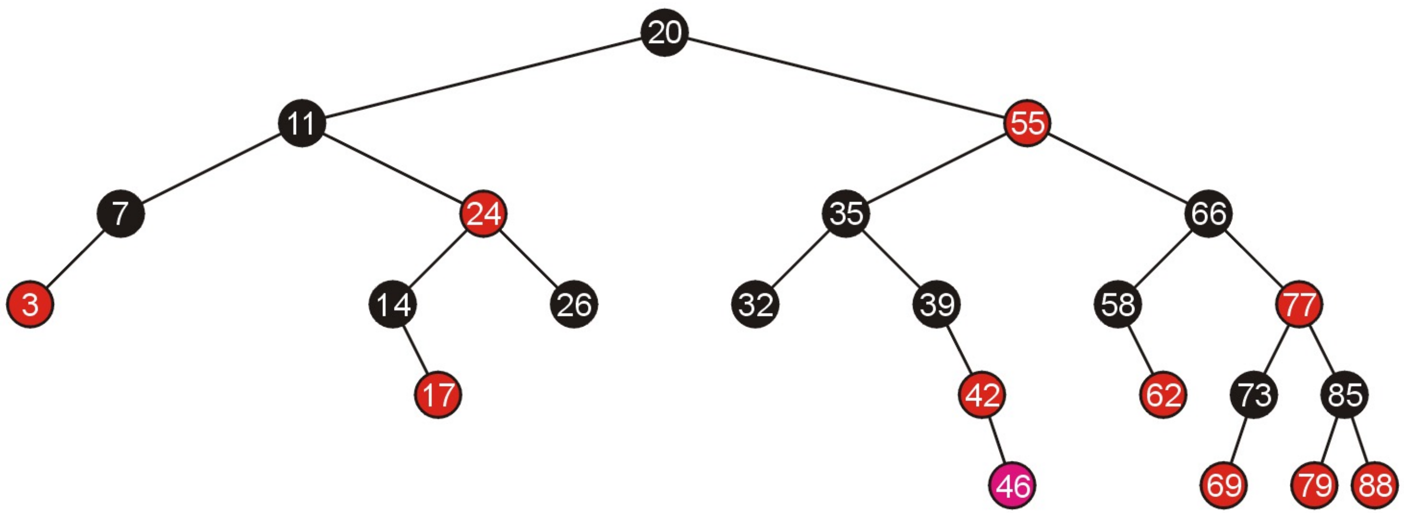
Examples of Insertions

- Given the following red-black tree, we will make a number of insertions



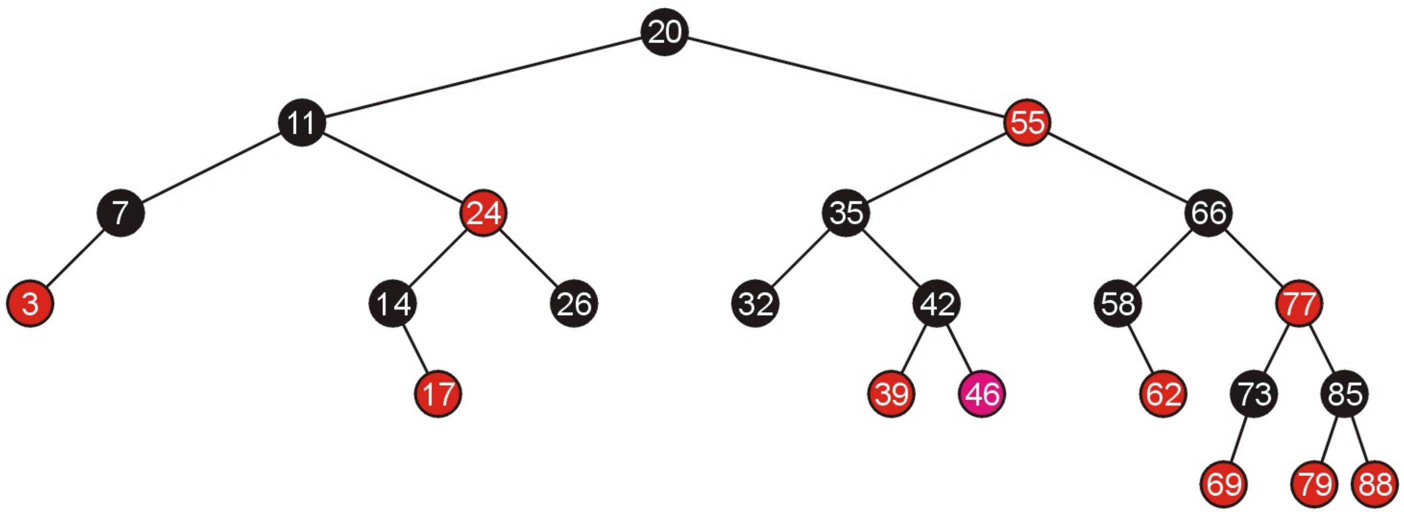
Examples of Insertions: Insert 46

- Adding 46 creates a red-red pair which can be corrected with a single rotation



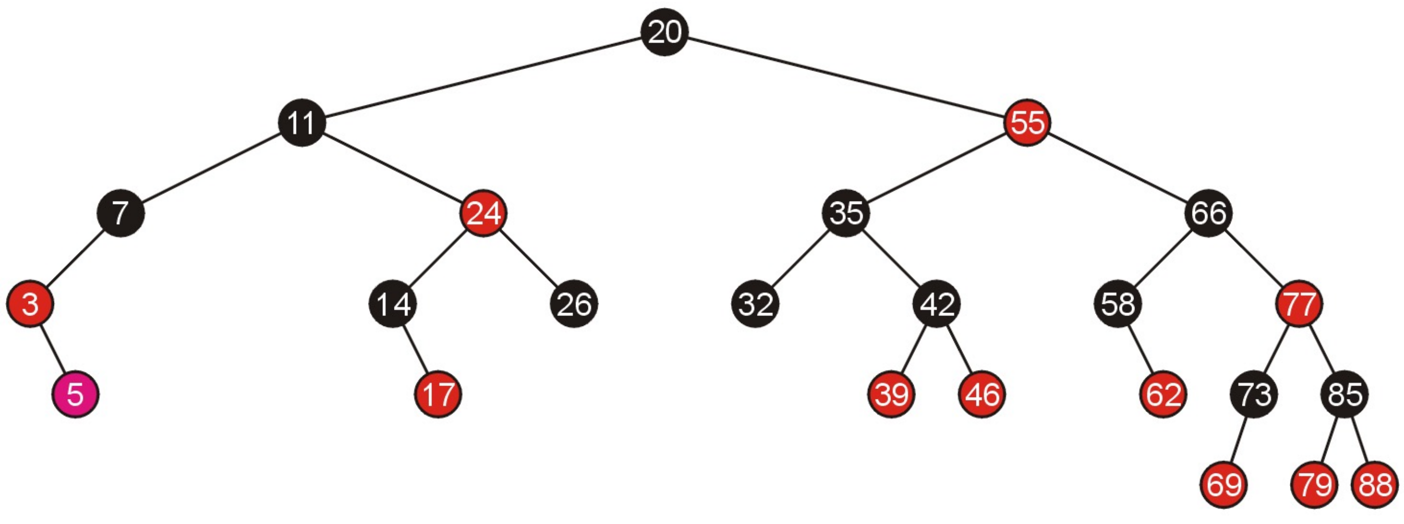
Examples of Insertions: Insert 46

- Because the pivot is still black, we are finished



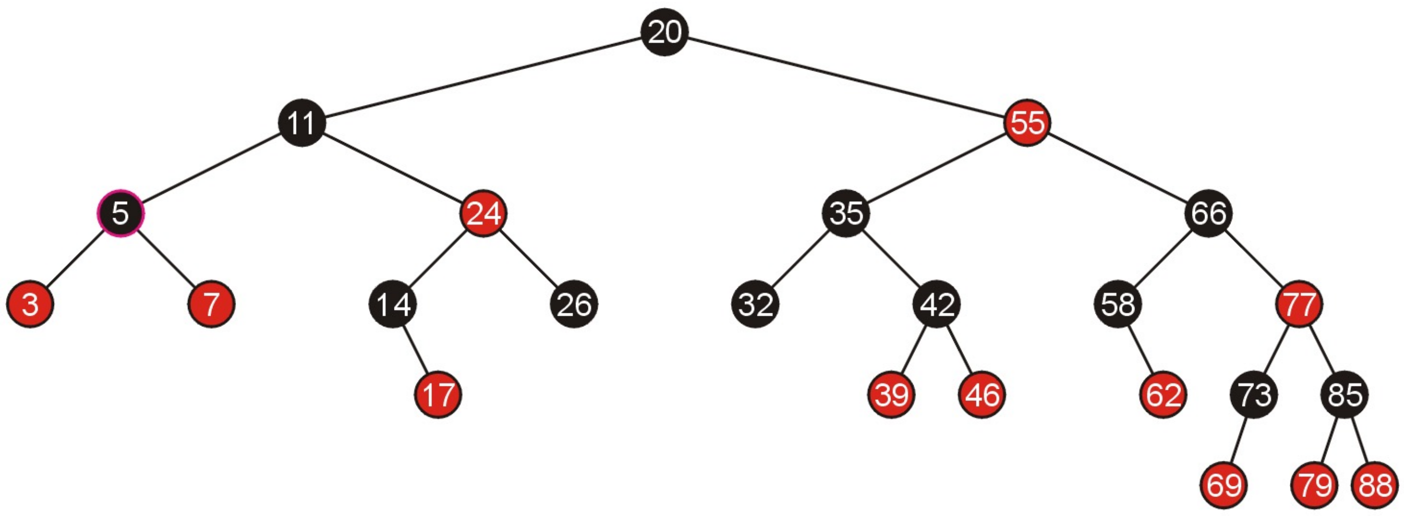
Examples of Insertions: Insert 5

- Similarly, adding 5 requires a single rotation



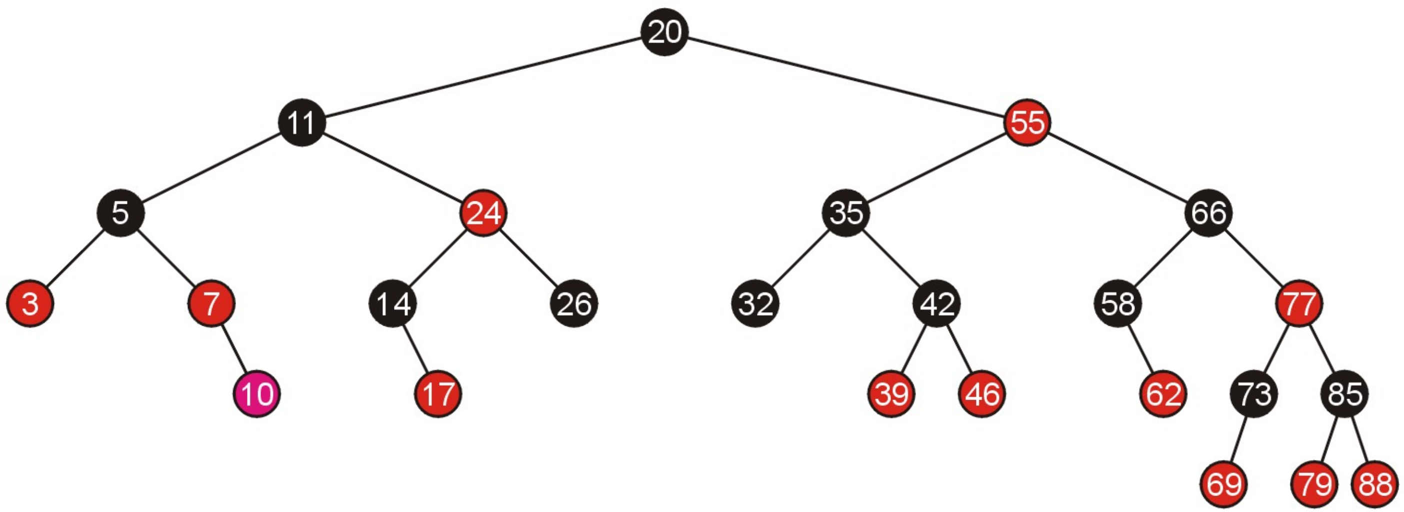
Examples of Insertions: Insert 5

- Which again, does not require any additional work



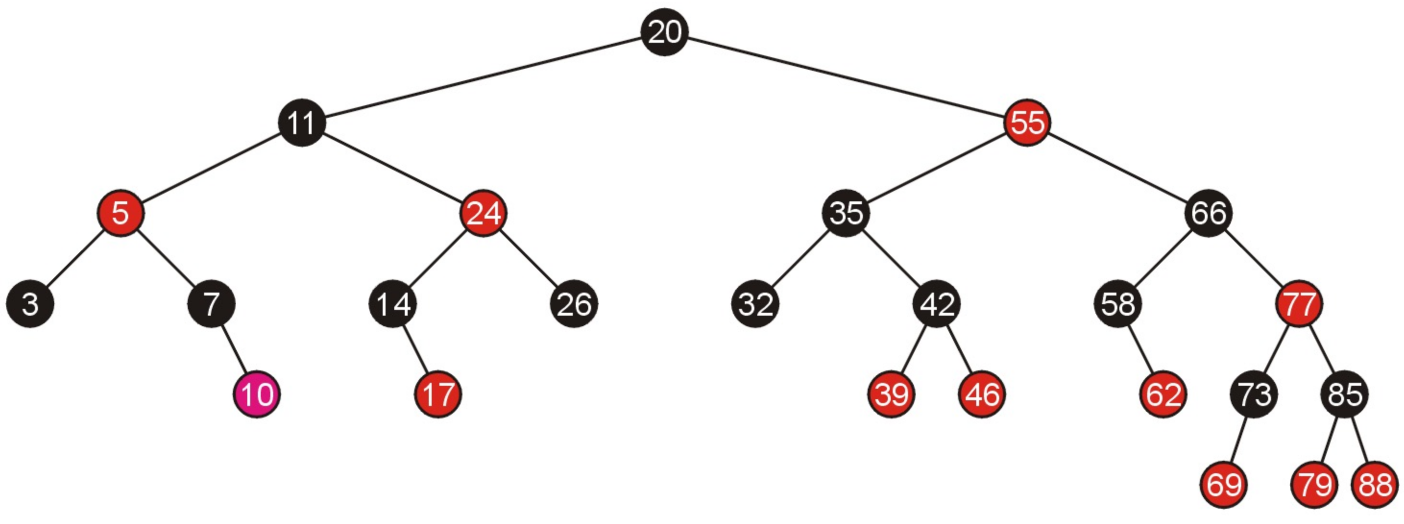
Examples of Insertions: Insert 10

- Adding 10 allows us to simply swap the color of the grand parent and the parent and the parent's sibling



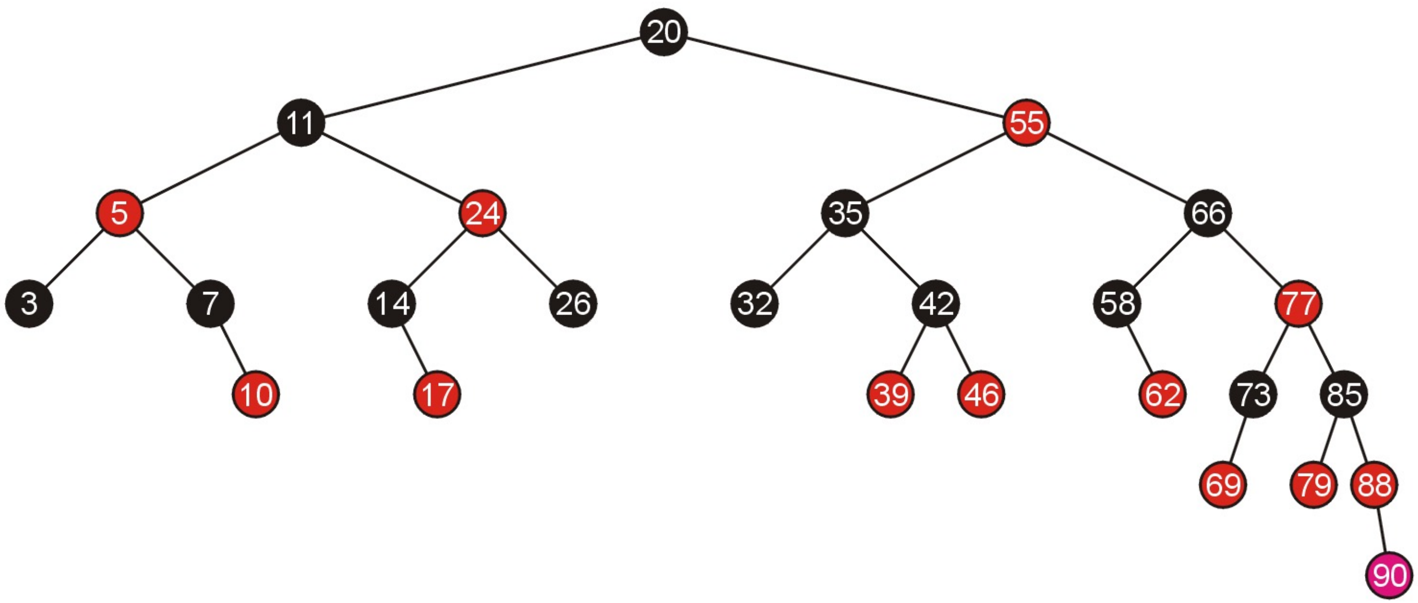
Examples of Insertions: Insert 10

- Because the parent of 5 is black, we are finished



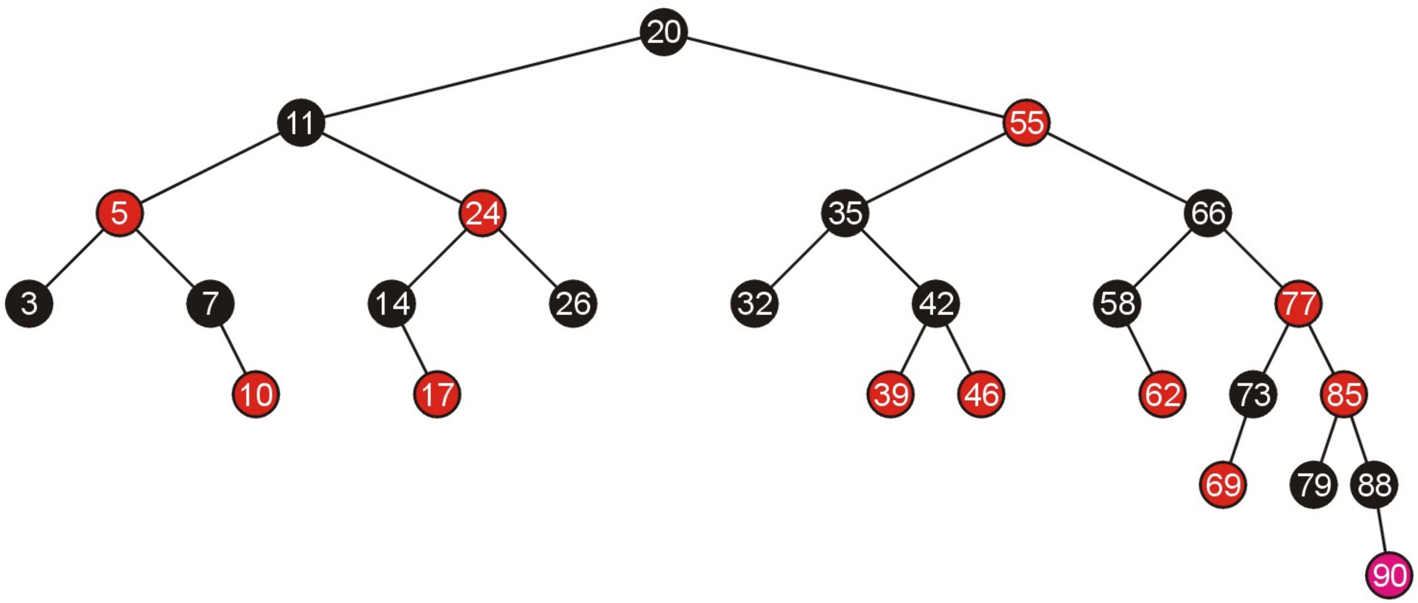
Examples of Insertions: Insert 90

- Adding 90 again requires us to swap the colors of the grandparent and its two children



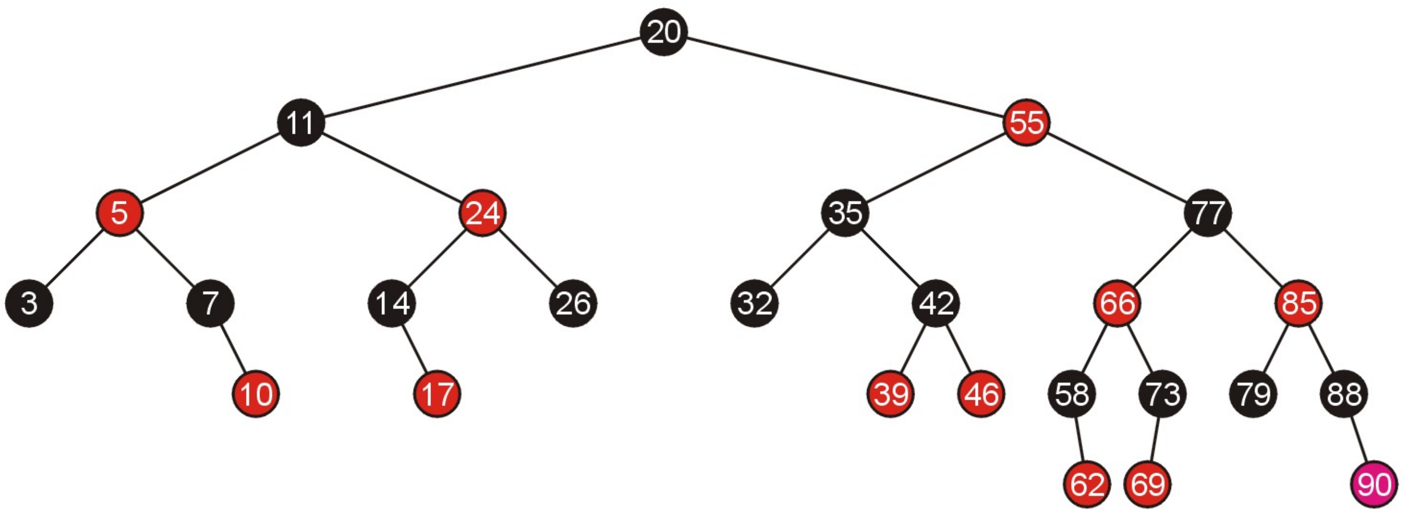
Examples of Insertions: Insert 90

- This causes a red-red parent-child pair, which now requires a rotation



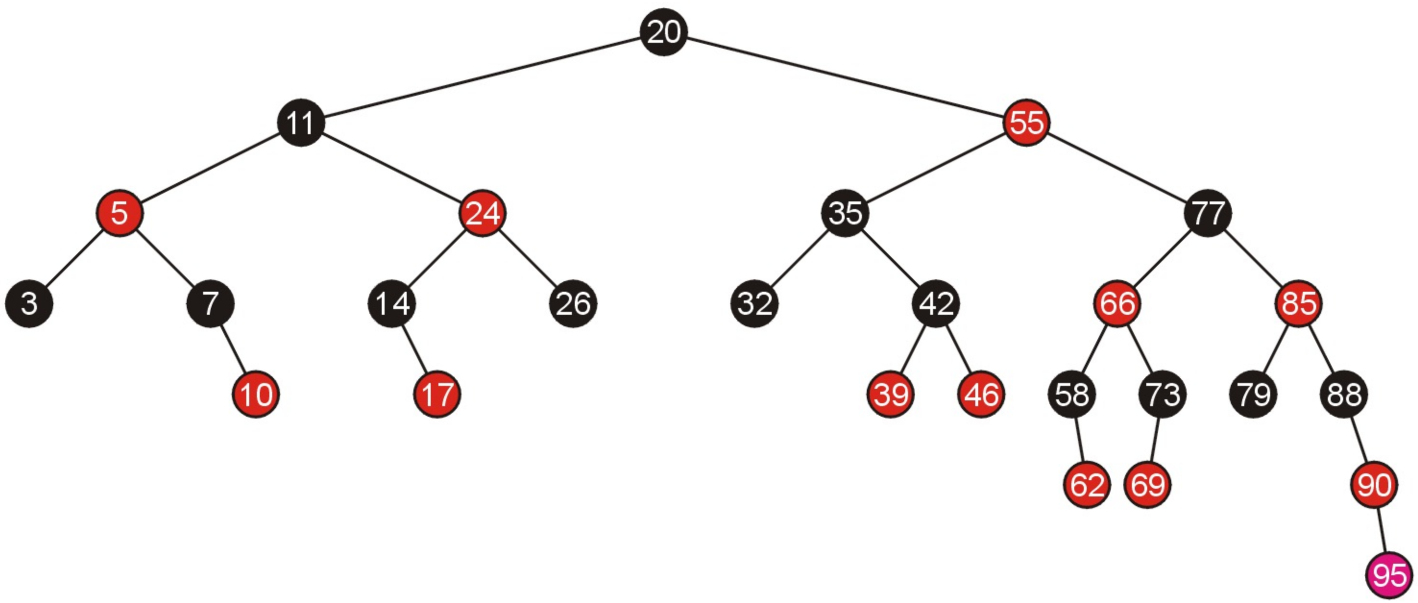
Examples of Insertions: Insert 90

- A rotation does not require any subsequent modifications, so we are finished



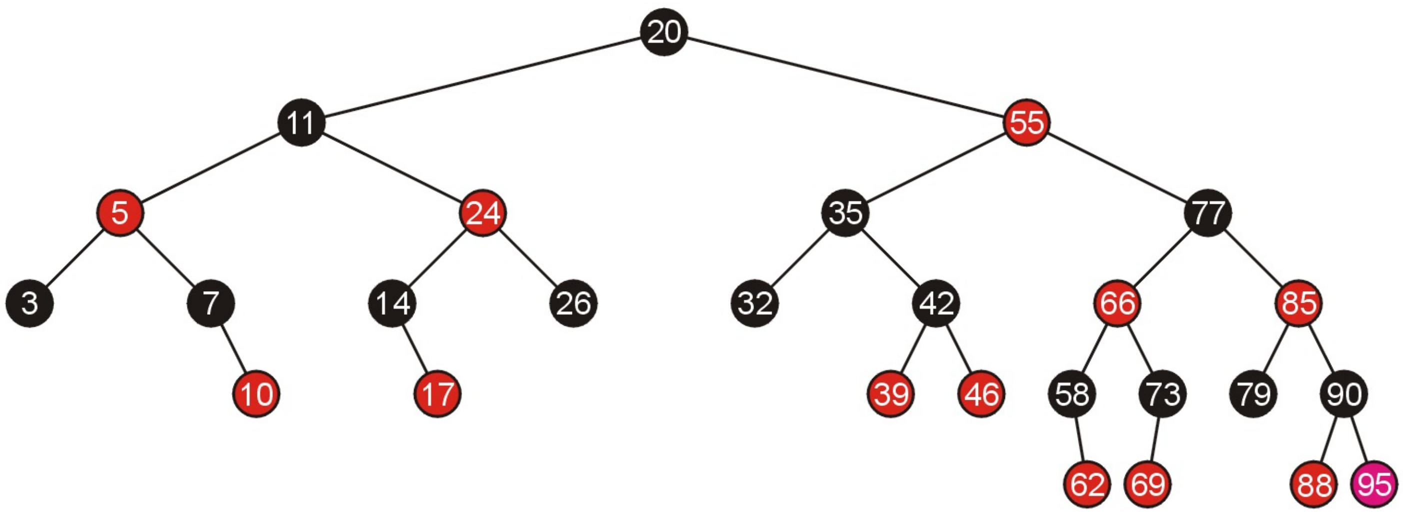
Examples of Insertions: Insert 95

- Inserting 95 requires a single rotation



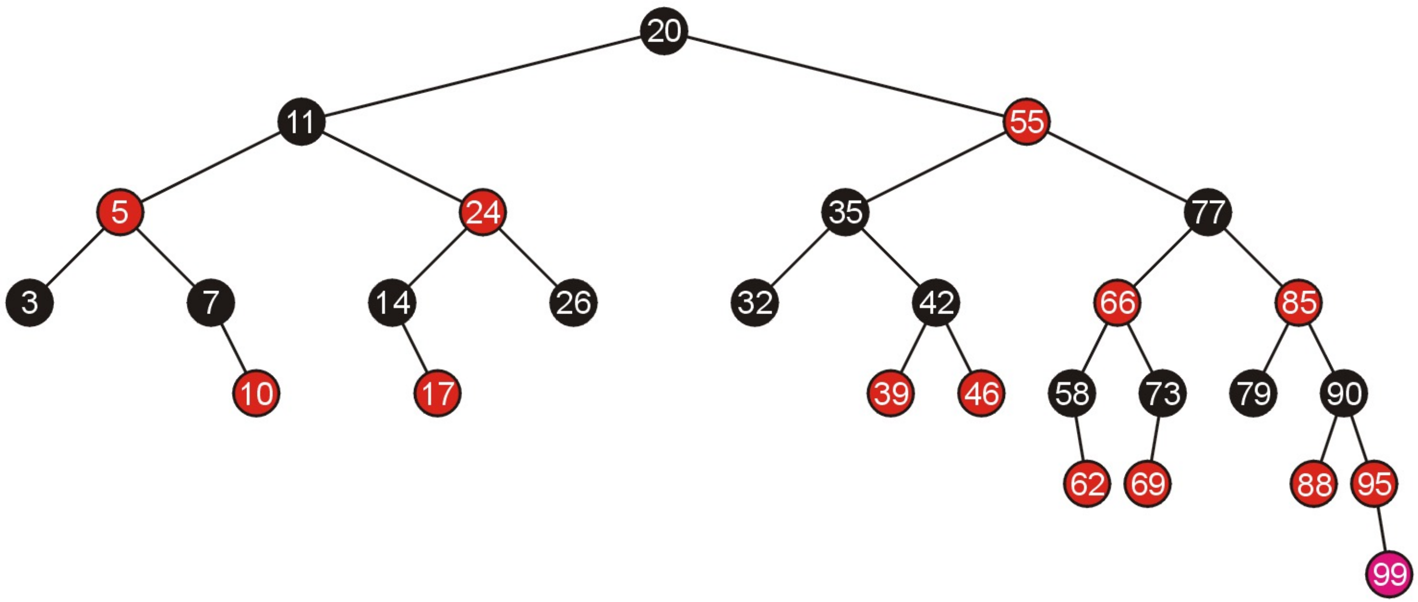
Examples of Insertions: Insert 95

- And consequently, we are finished



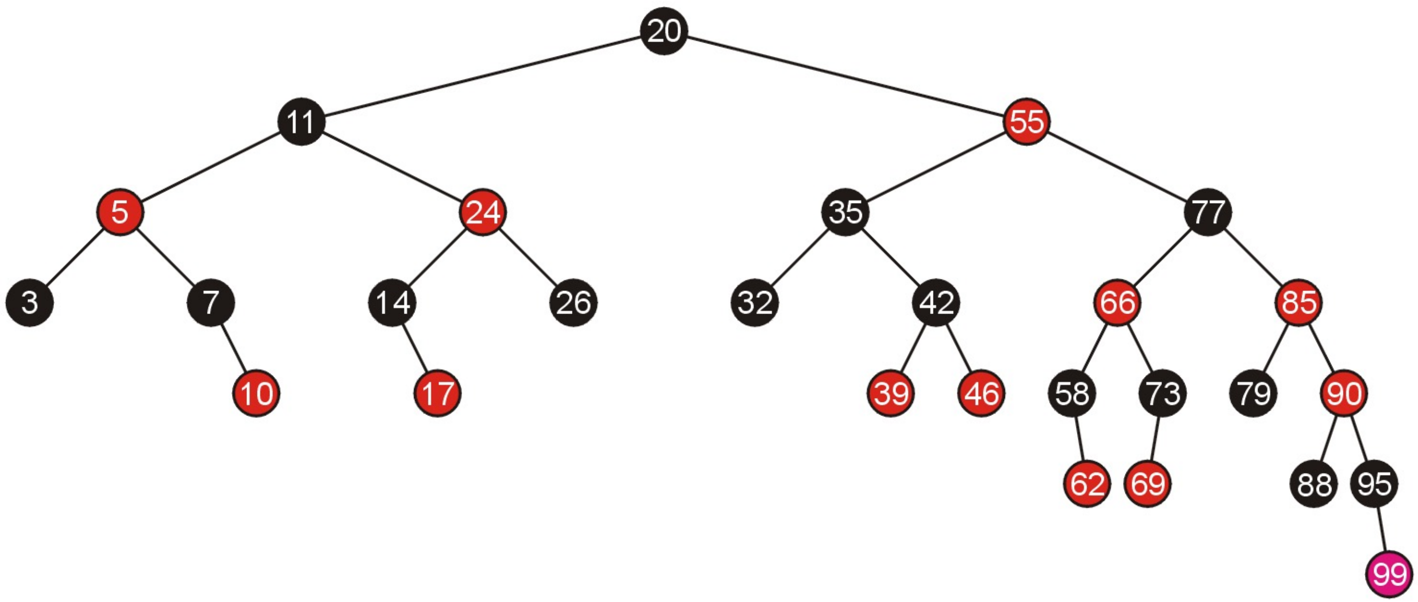
Examples of Insertions: Insert 99

- Adding 99 requires us to swap the colors of its grandparent and the grandparent's children



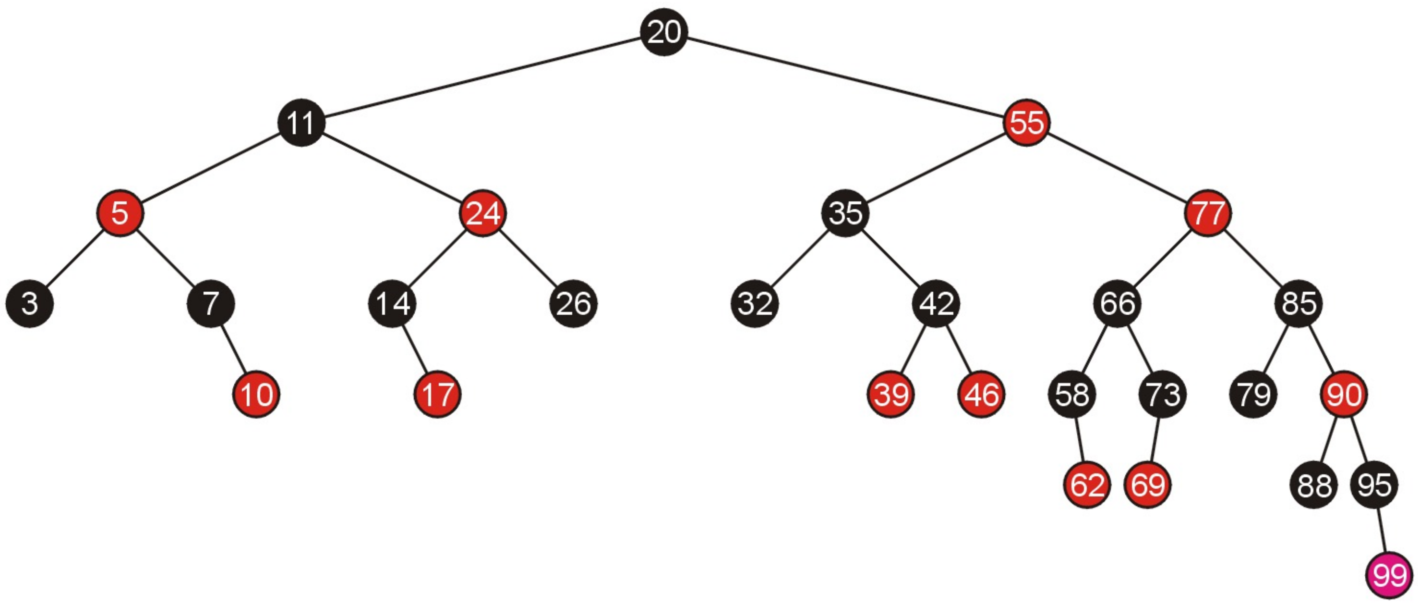
Examples of Insertions: Insert 99

- This causes another red-red child-parent conflict between 85 and 90 which must be fixed, again by swapping colors



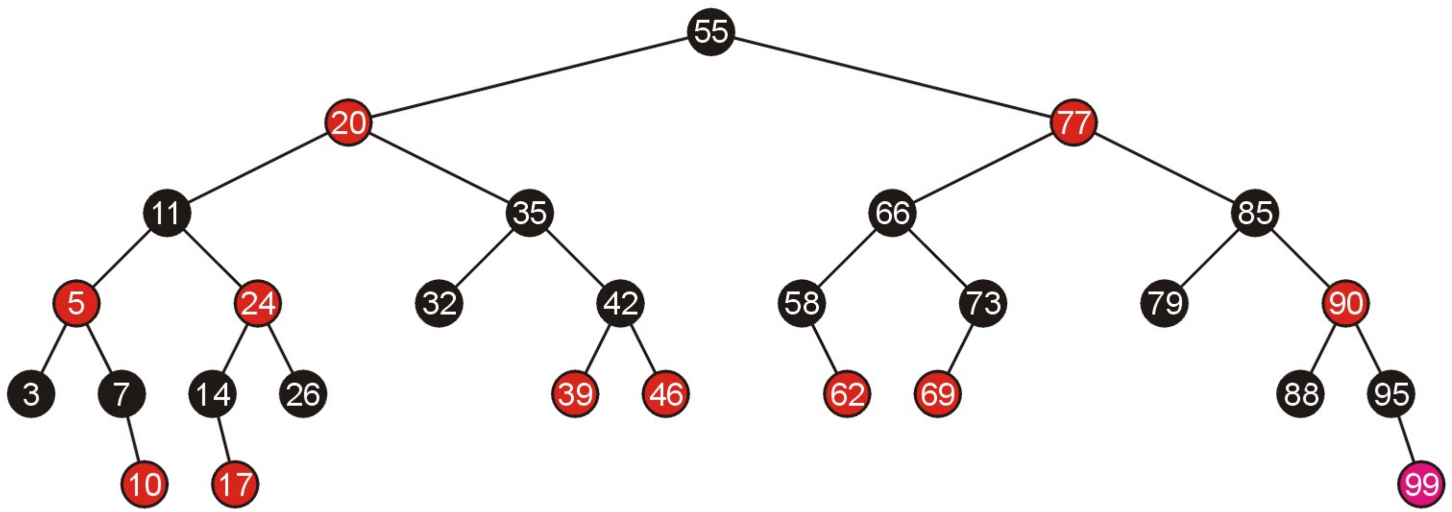
Examples of Insertions: Insert 99

- This results in another red-red parent-child conflict, this time, requiring a rotation



Examples of Insertions: Insert 99

- Thus, the rotation solves the problem



Top-Down Insertions and Deletions

- With a bottom-up insertion, it is first necessary to search the tree for the appropriate location, and only then recurs back to the root correcting any problems
  - This is similar to AVL trees

- With red-black trees, it is possible to perform both insertions and deletions strictly by starting at the root, but not requiring the recurs back to the root

Top-Down Insertions

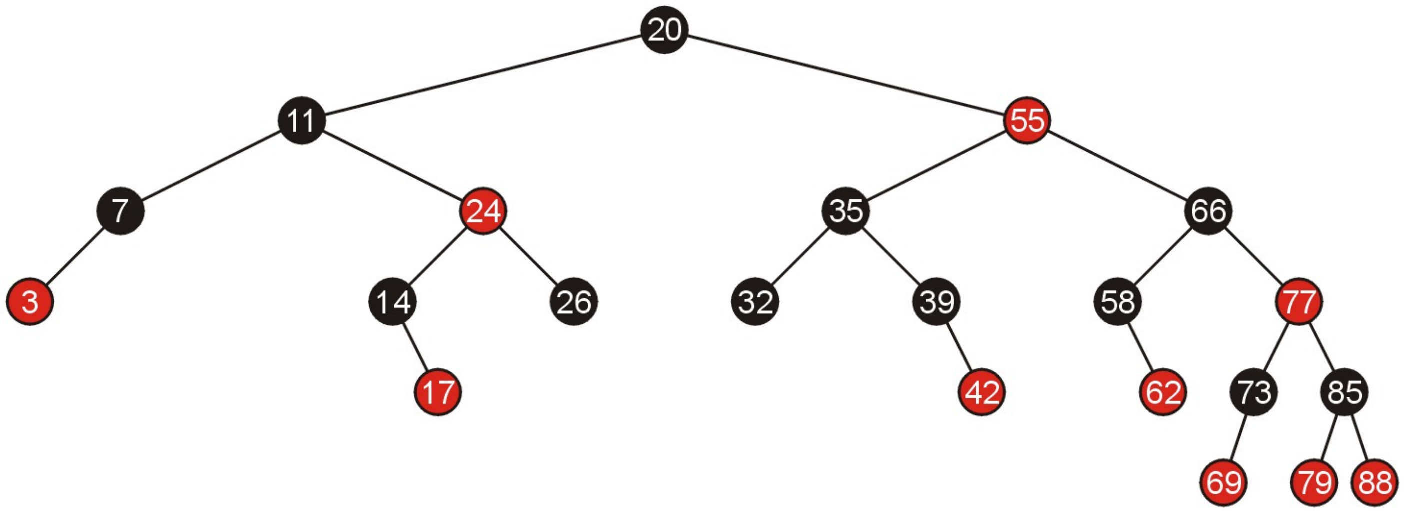
- The important observation is:
  - Rotations (Case #1) do not require recursive steps back to the root
  - Swapping (Case #2) may require recursive corrections going back all the way to the root

- Therefore, while moving down from the root, automatically swap the colors of any black node with two red children
  - this may require at most one rotation at the parent of the now-red node

NXC LAB

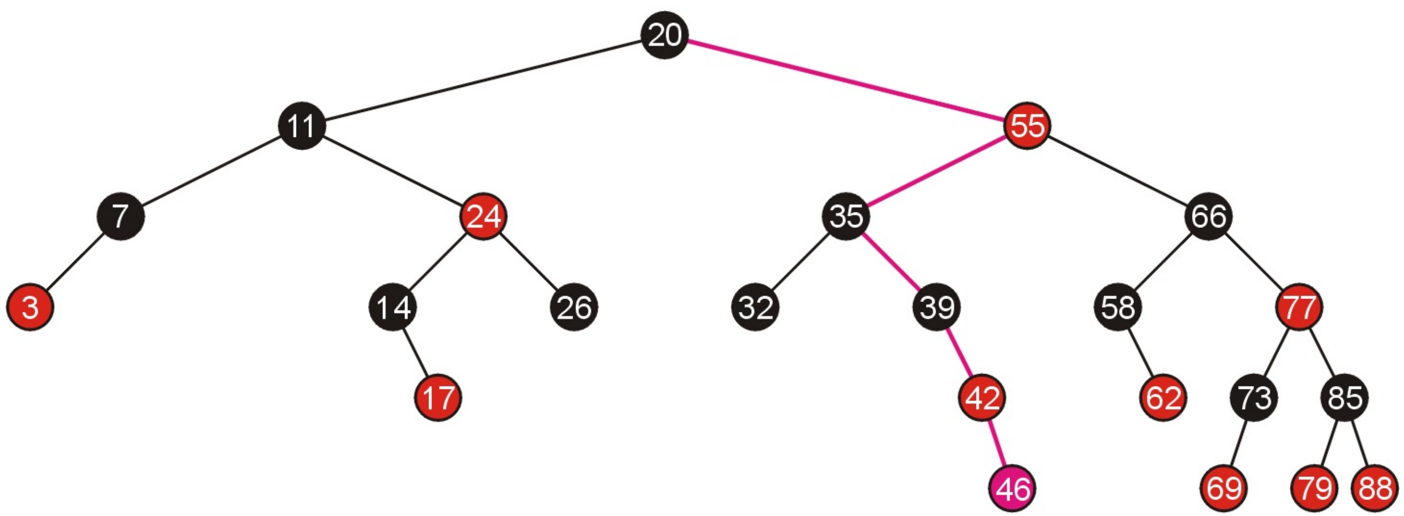
Examples of Top-Down Insertions

- We will start with the same red-black tree as before, but make top-down insertions (no recursion):



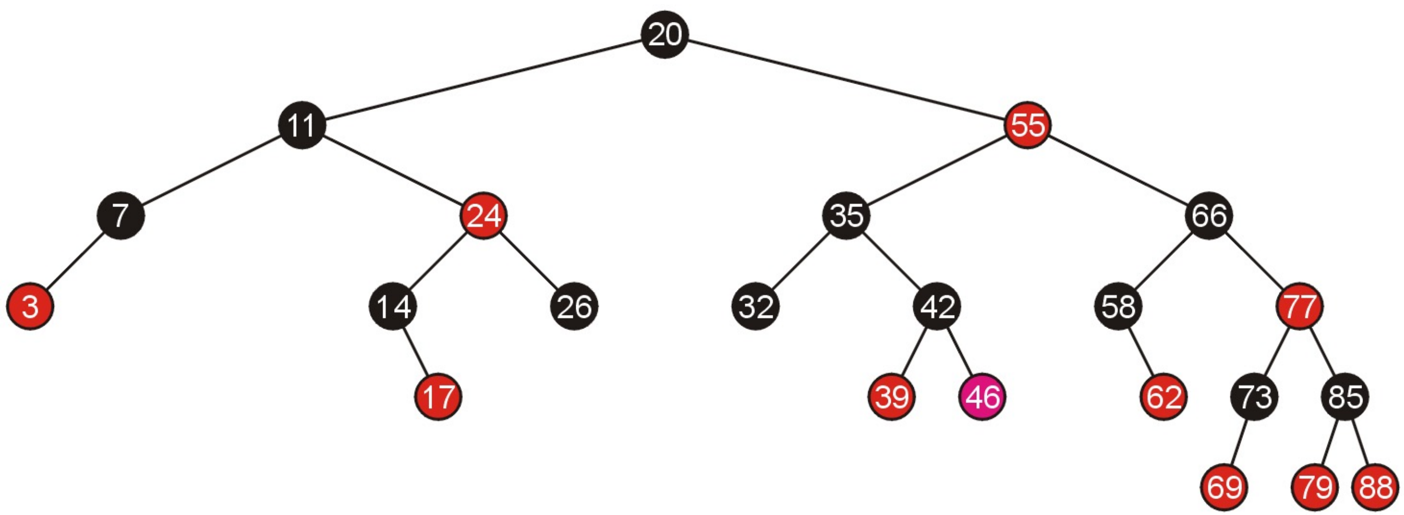
Examples of Top-Down Insertions: Insert 46

- Adding 46 does not find any (necessarily black) parent with two red children



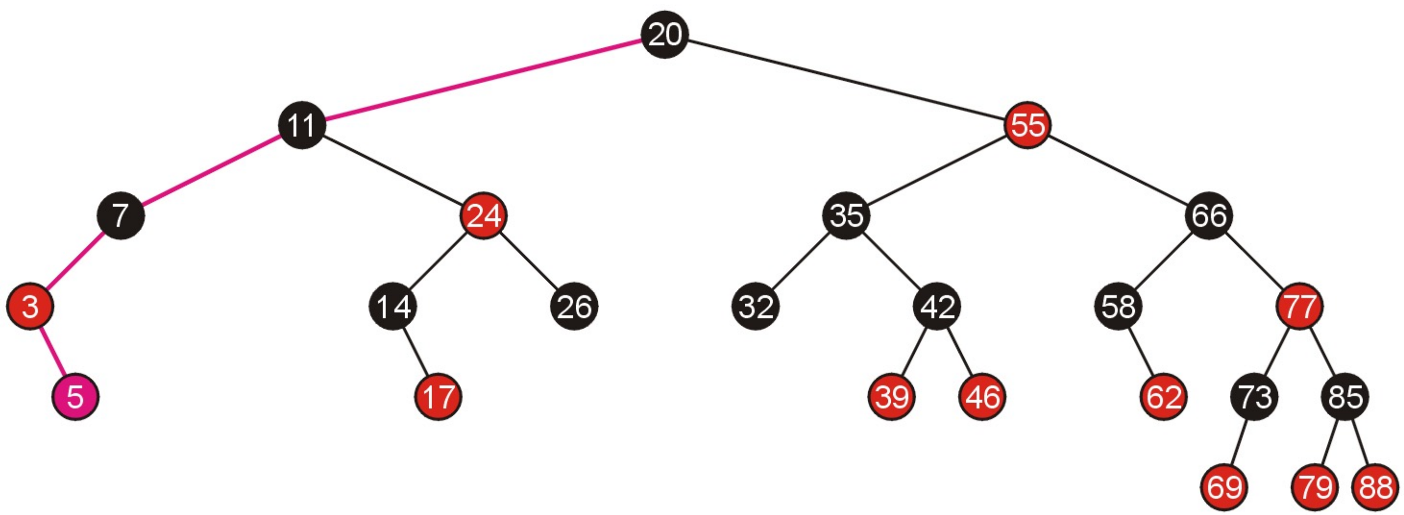
Examples of Top-Down Insertions: Insert 46

- However, it does require one rotation at the end



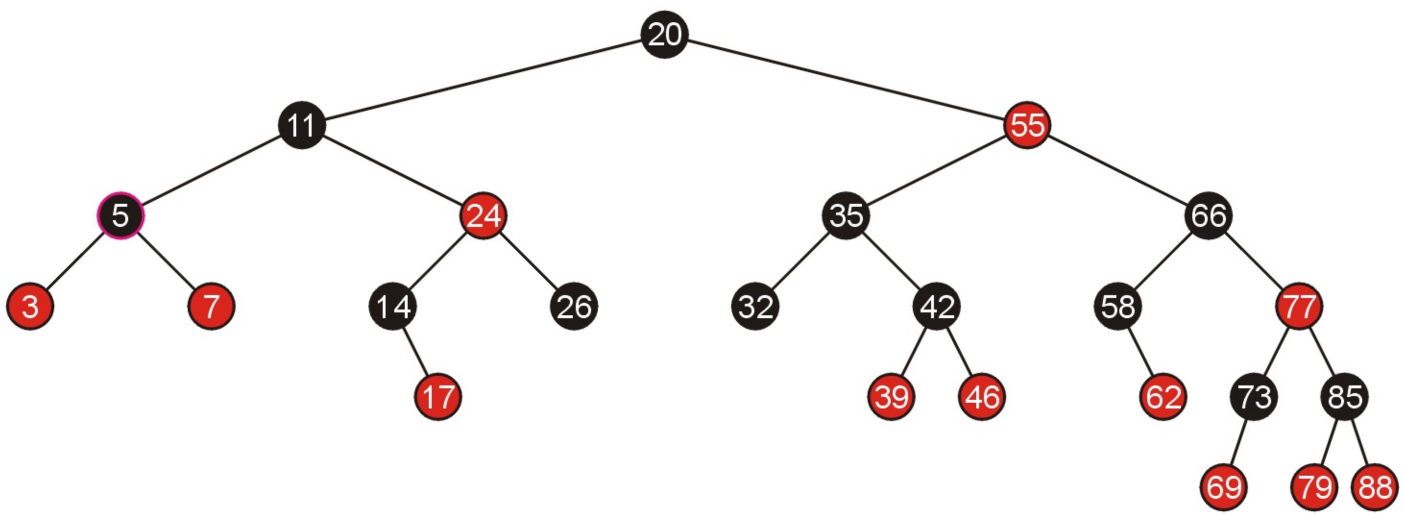
Examples of Top-Down Insertions: Insert 5

- Similarly, adding 5 does not meet any parent with two red children:



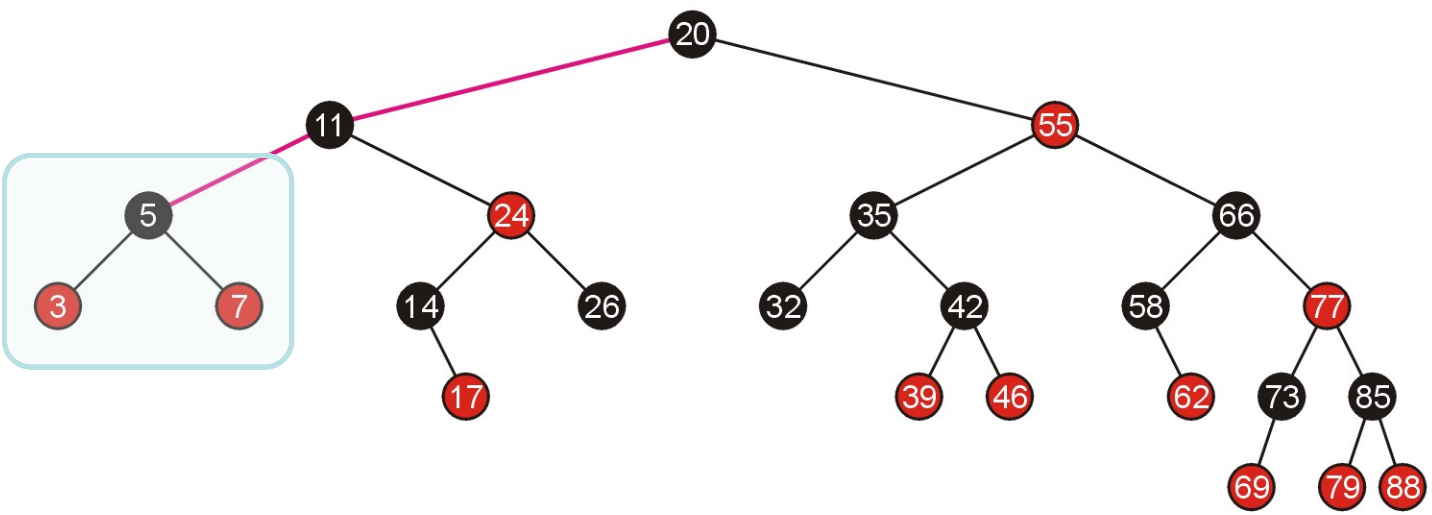
Examples of Top-Down Insertions: Insert 5

- A rotation solves the last problem



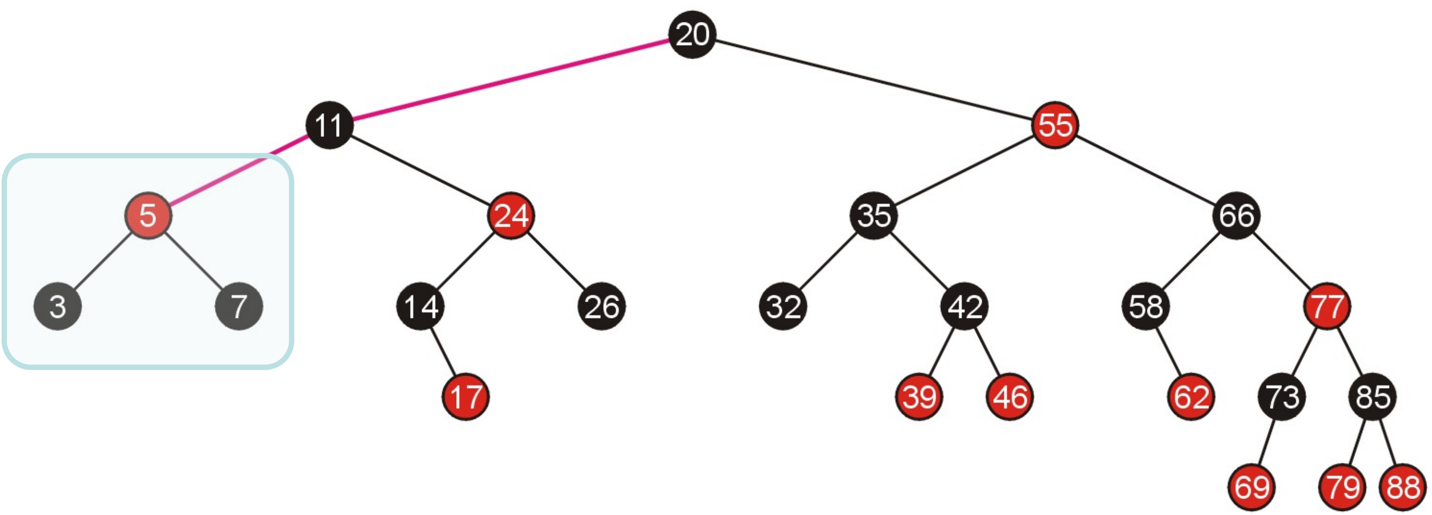
Examples of Top-Down Insertions: Insert 10

- To insert 10, we can spot that node 5 has two red children



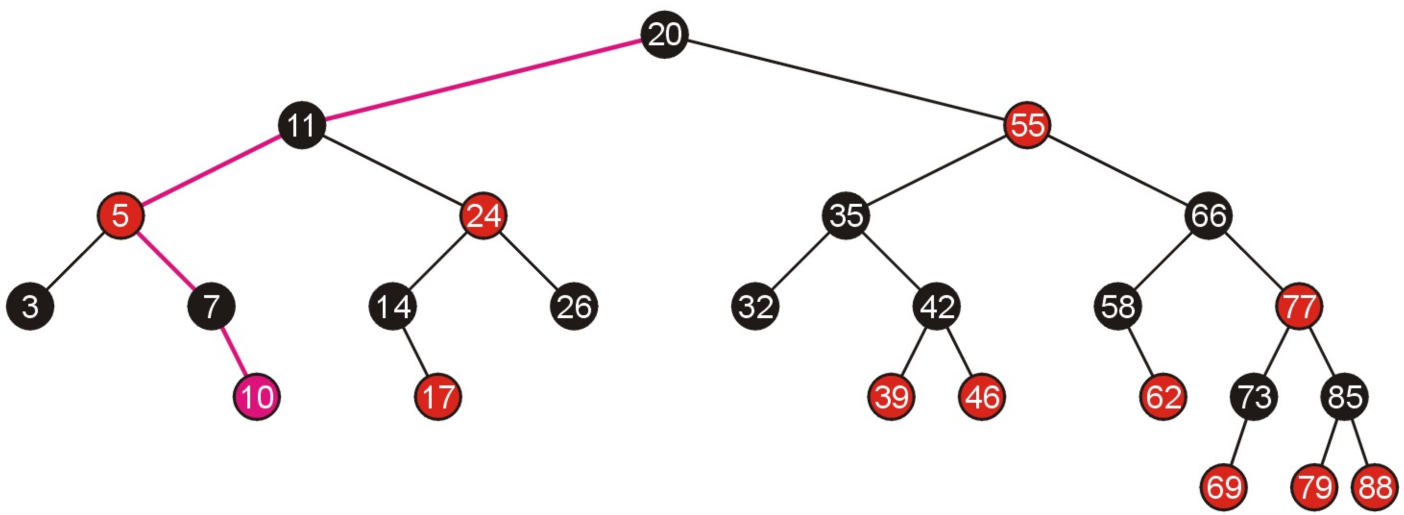
Examples of Top-Down Insertions: Insert 10

- We swap the colors, and this does not cause a problem between 5 and 11



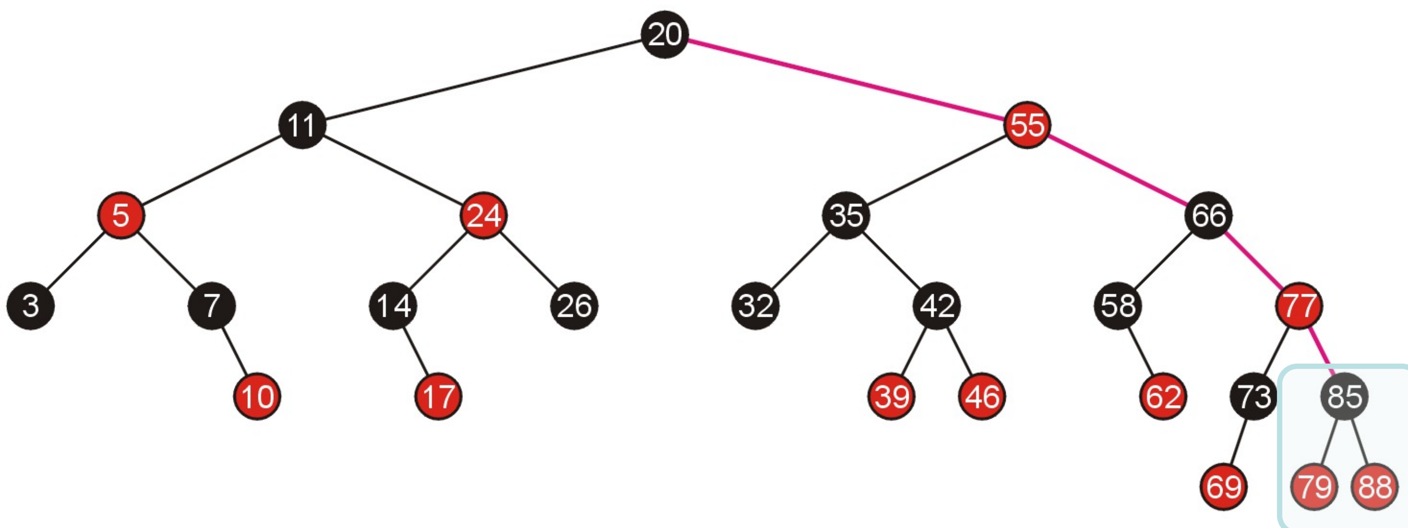
Examples of Top-Down Insertions: Insert 10

- We continue and place 10 in the appropriate location
  - No further rotations are required



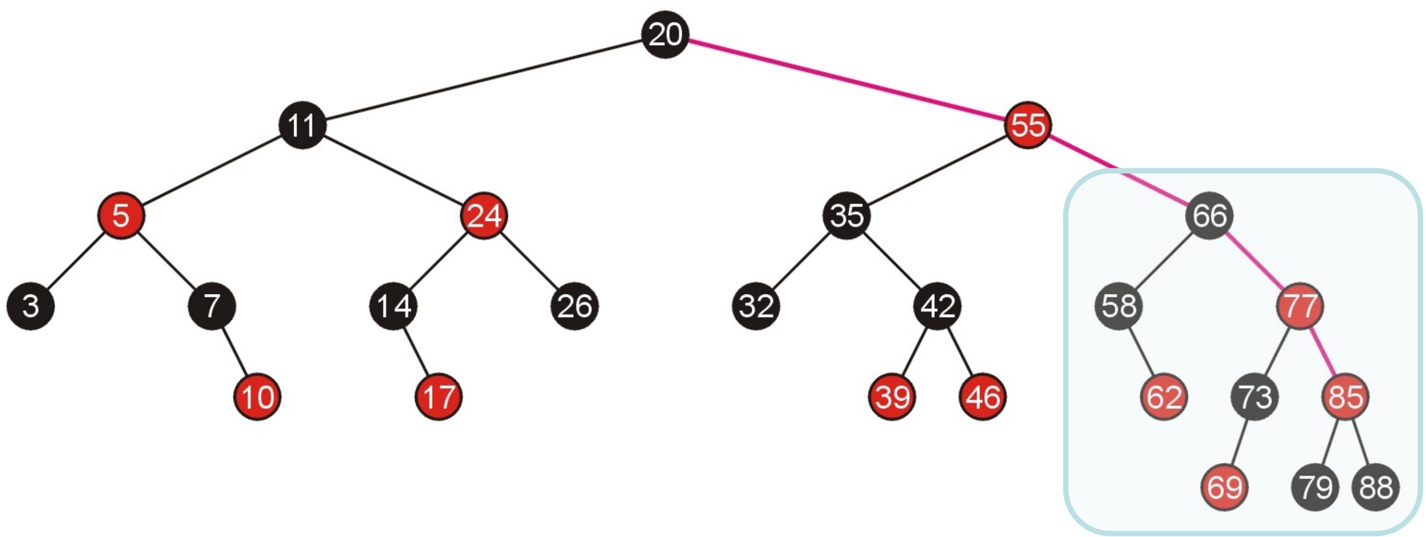
Examples of Top-Down Insertions: Insert 90

- To add the node 90, we traverse down the right tree until we reach 85 which has two red children



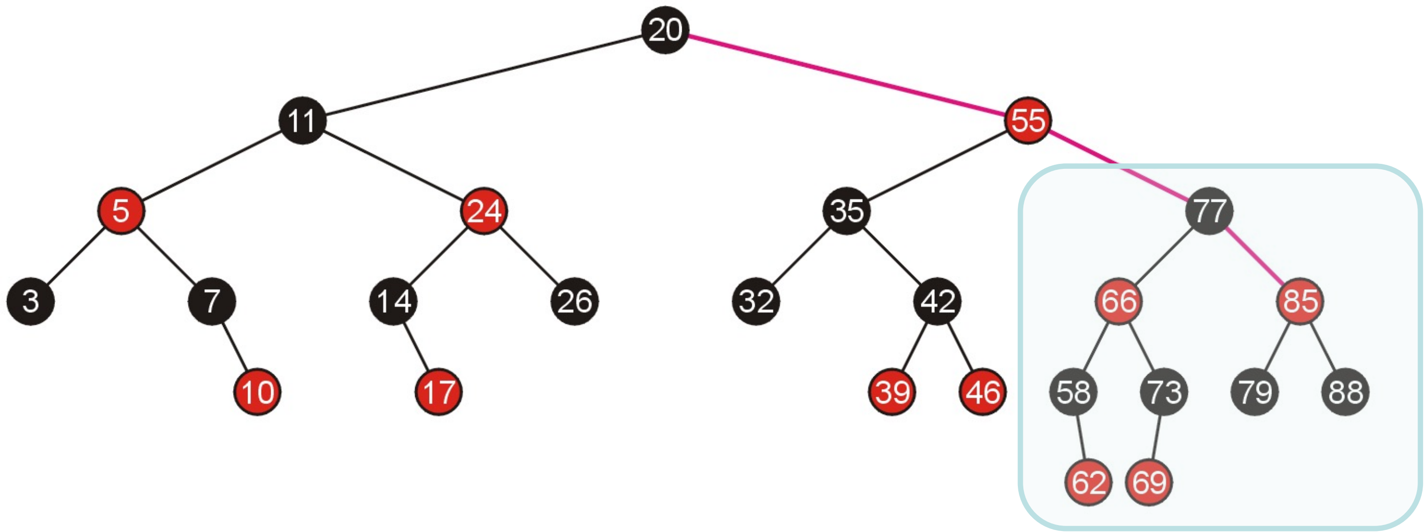
Examples of Top-Down Insertions: Insert 90

- We swap the colors, however this creates a red-red pair between 85 and its parent



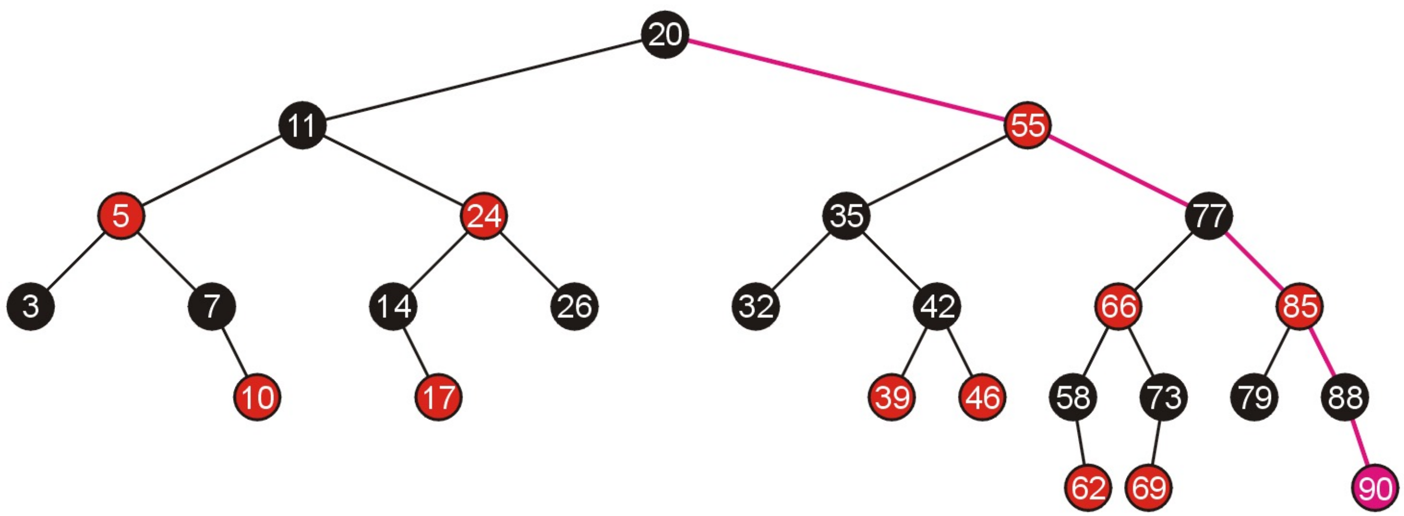
Examples of Top-Down Insertions: Insert 90

- We require only one rotation to solve this problem, and we are guaranteed that this will not cause any problem for its parents



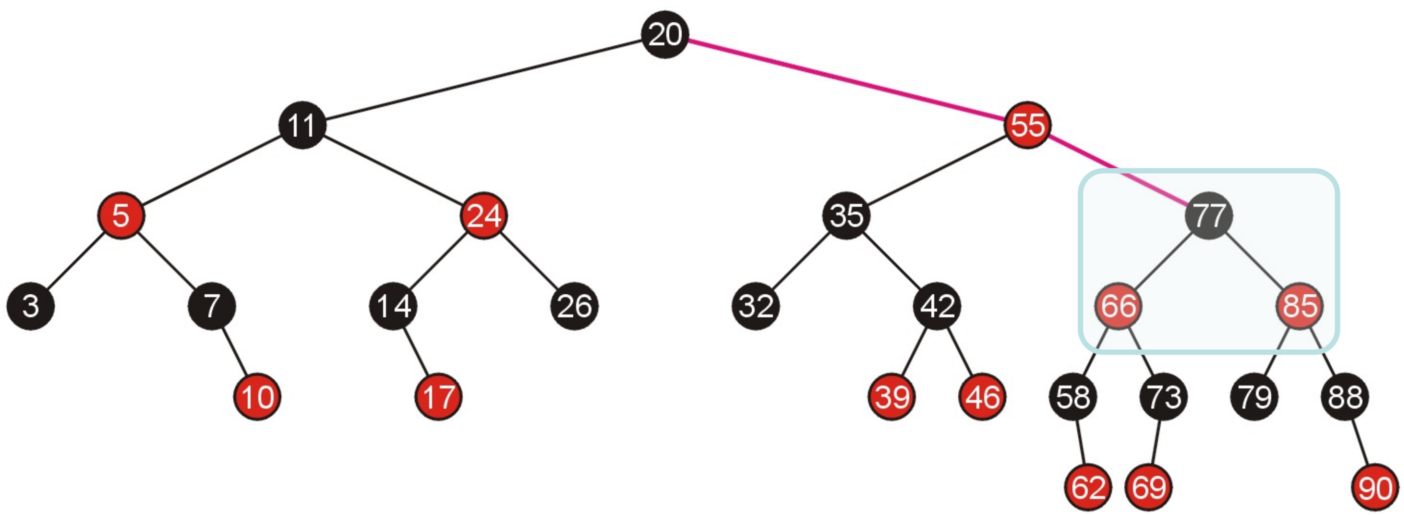
Examples of Top-Down Insertions: Insert 90

- We continue to search down the right path and add 90 in the appropriate location—no further corrections are required



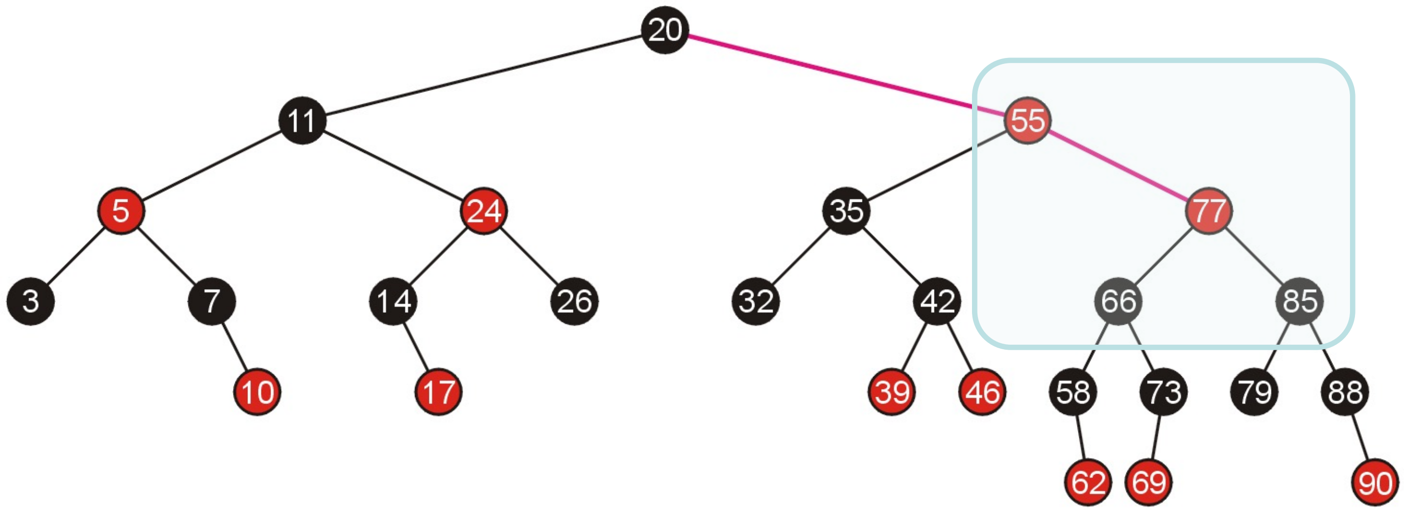
Examples of Top-Down Insertions: Insert 95

- Next, adding 95, we traverse down the right-hand until we reach node 77 which has two red children



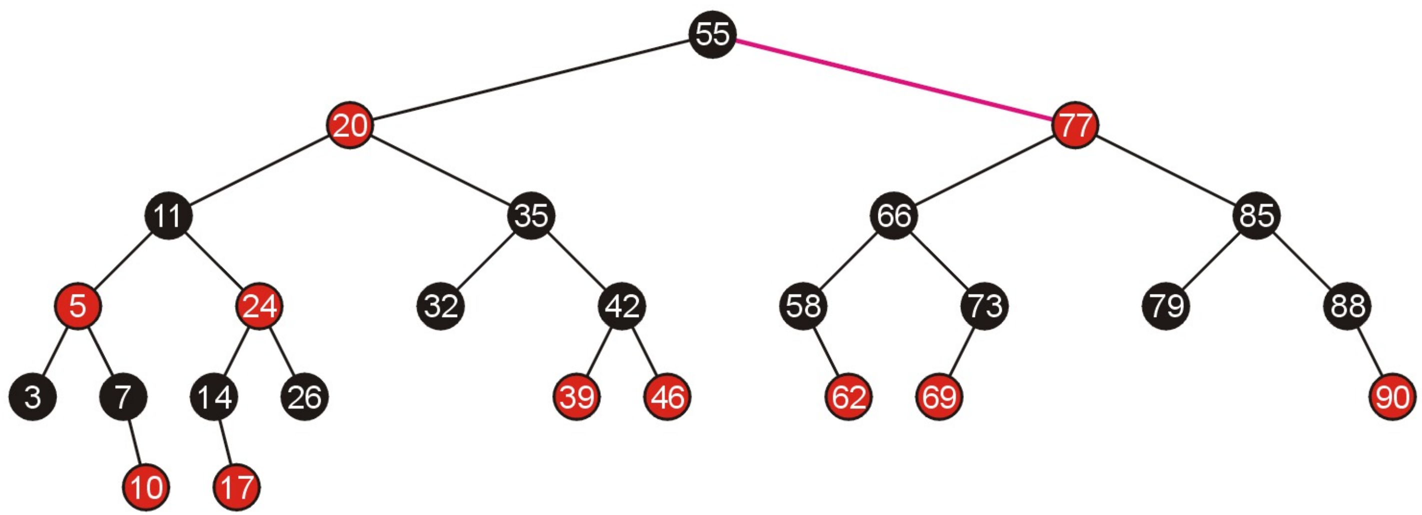
Examples of Top-Down Insertions: Insert 95

- We swap the colors, which causes a red-red parent-child combination which must be fixed by a rotation



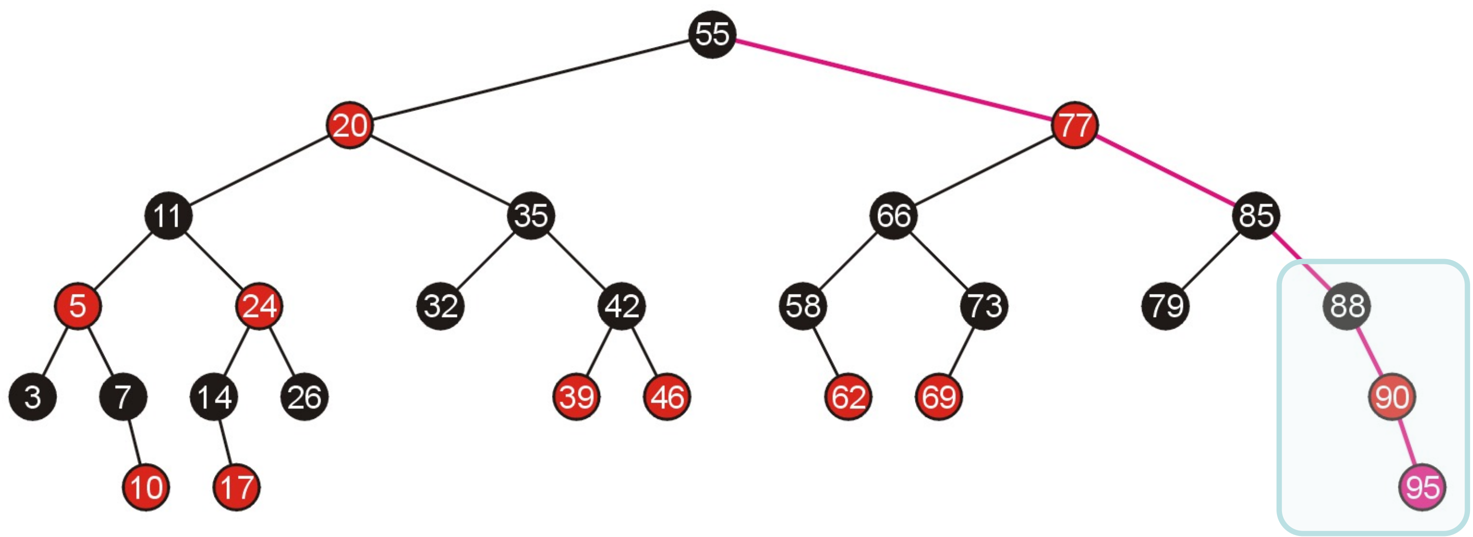
Examples of Top-Down Insertions: Insert 95

- The rotation is around the root
  - Note this rotation was not necessary with the bottom-up insertion of 95



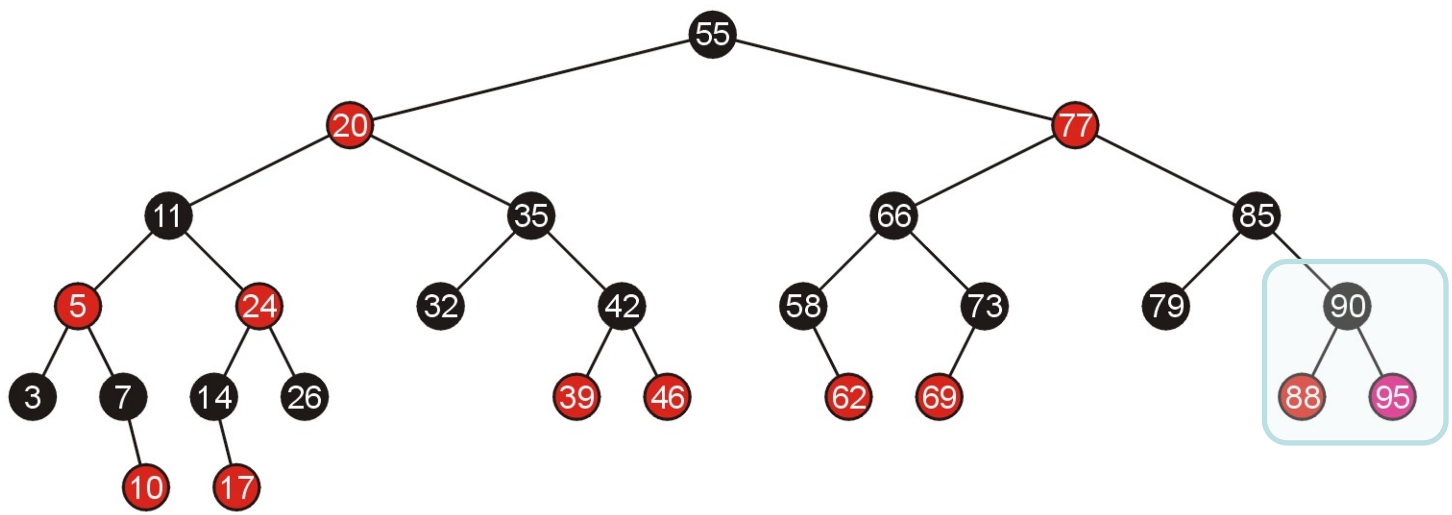
Examples of Top-Down Insertions: Insert 95

- We can now proceed to add 95 by following the right-hand branch, and the insertion causes a red-red parent-child combination



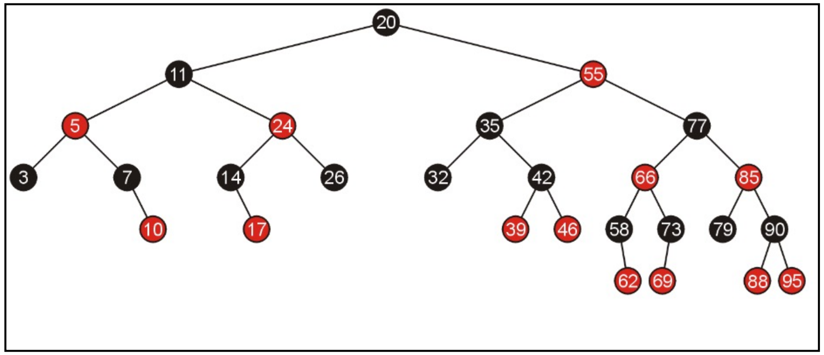
Examples of Top-Down Insertions: Insert 95

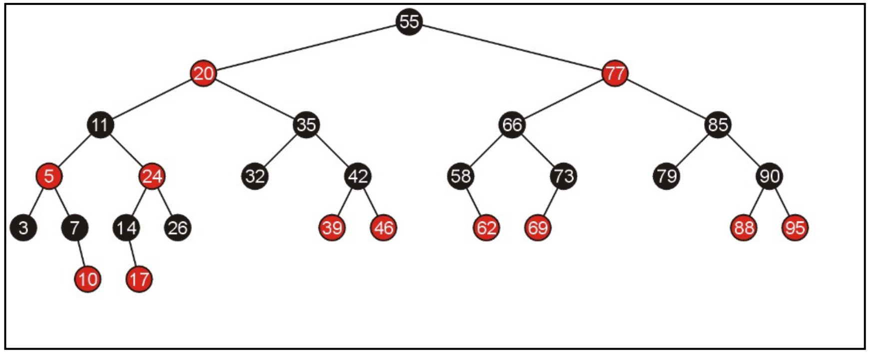
- This is fixed with a single rotation
  - We are guaranteed that this will not cause any further problems



Compare Top-Down and Bottom-up Insertions

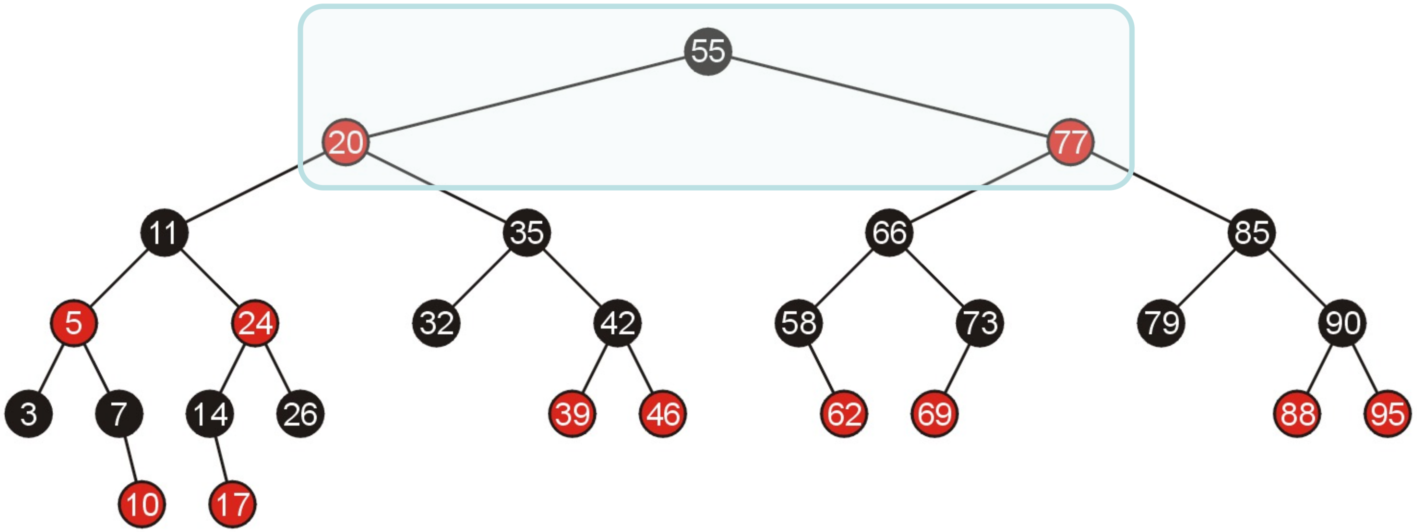
- If we compare the result of doing bottom-up insertions (left, seen previously) and top-down insertions (right), we note the resulting trees are different, but both are still valid red-black trees





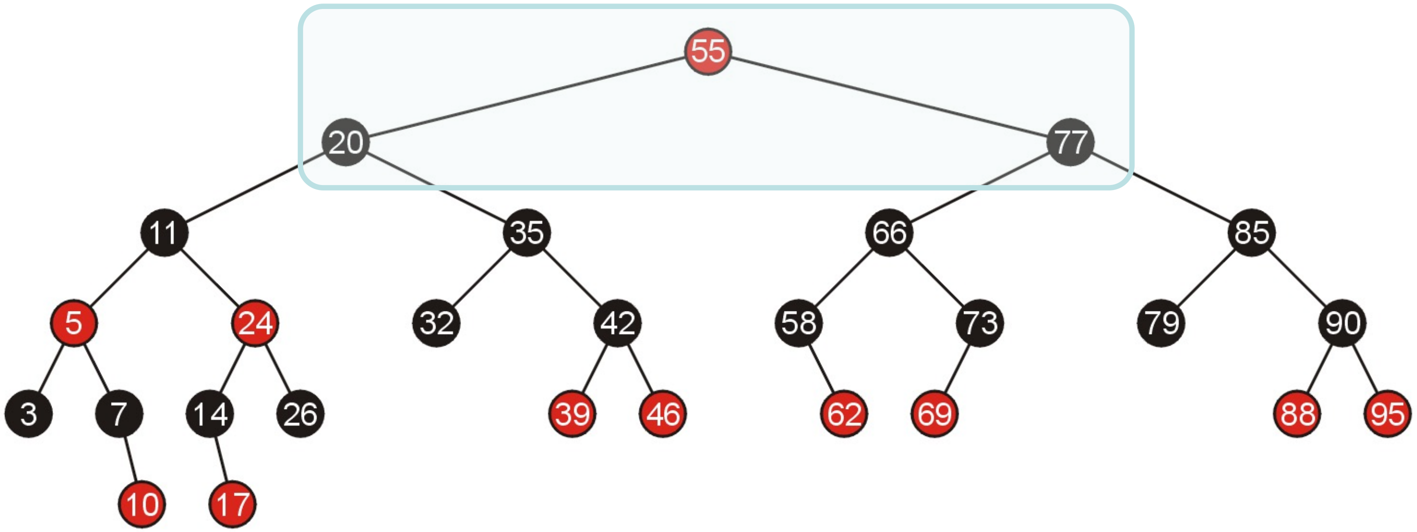
Examples of Top-Down Insertions: Insert 99

- If we add 99, the first thing we note is that the root has two red children, and therefore we swap the colors



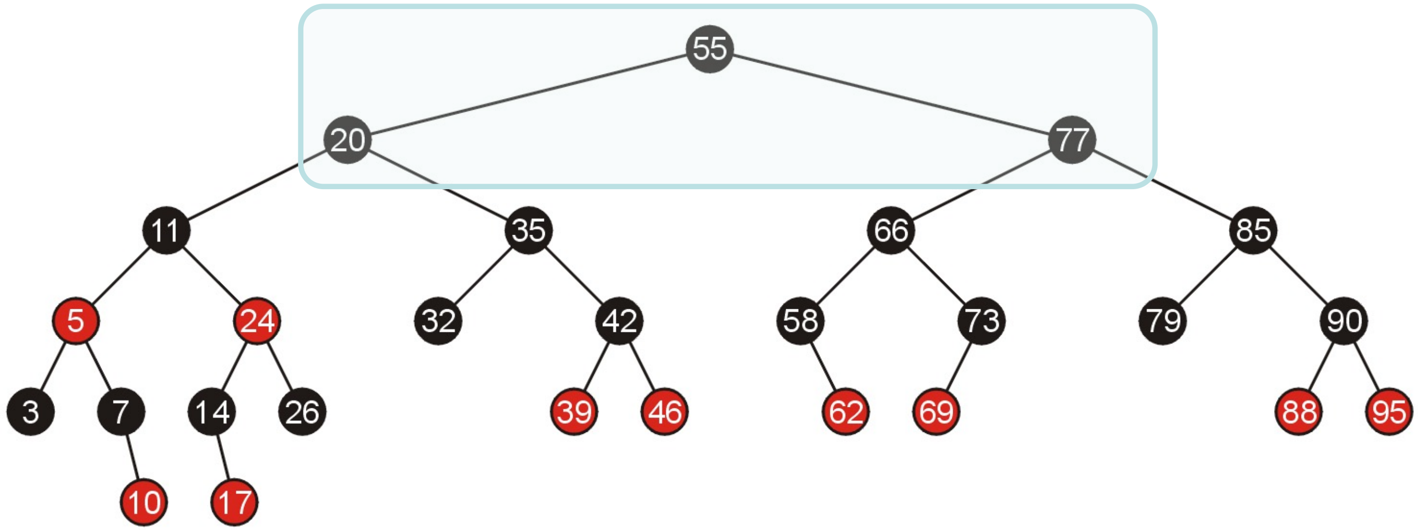
Examples of Top-Down Insertions: Insert 99

- At this point, each path to a non-full node still has the same number of black nodes, however, we violate the requirement that the root is black



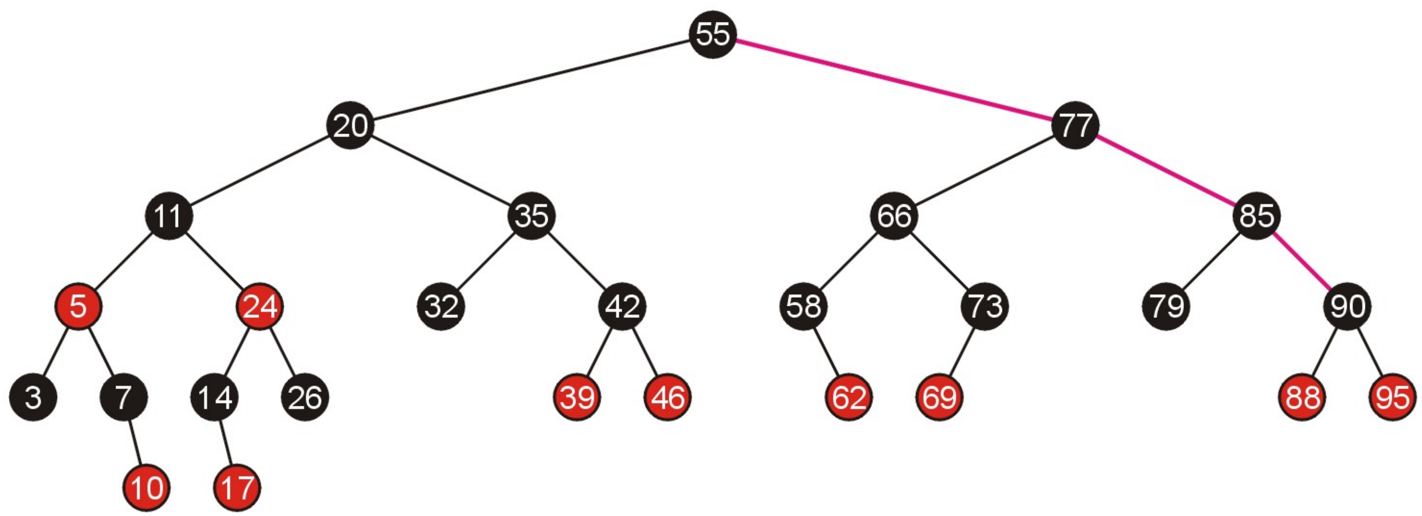
Examples of Top-Down Insertions: Insert 99

- We change the color of the root to black
  - This adds one more black node to each path



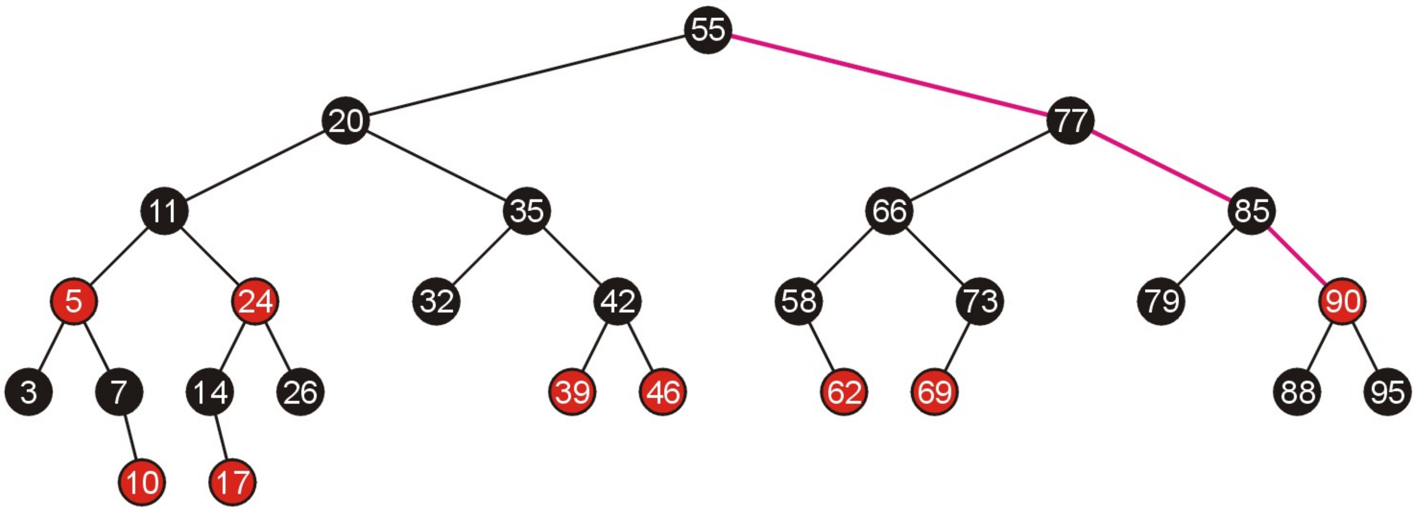
Examples of Top-Down Insertions: Insert 99

- Moving to the right, we now reach node 90 which has two red children and therefore we swap the colors



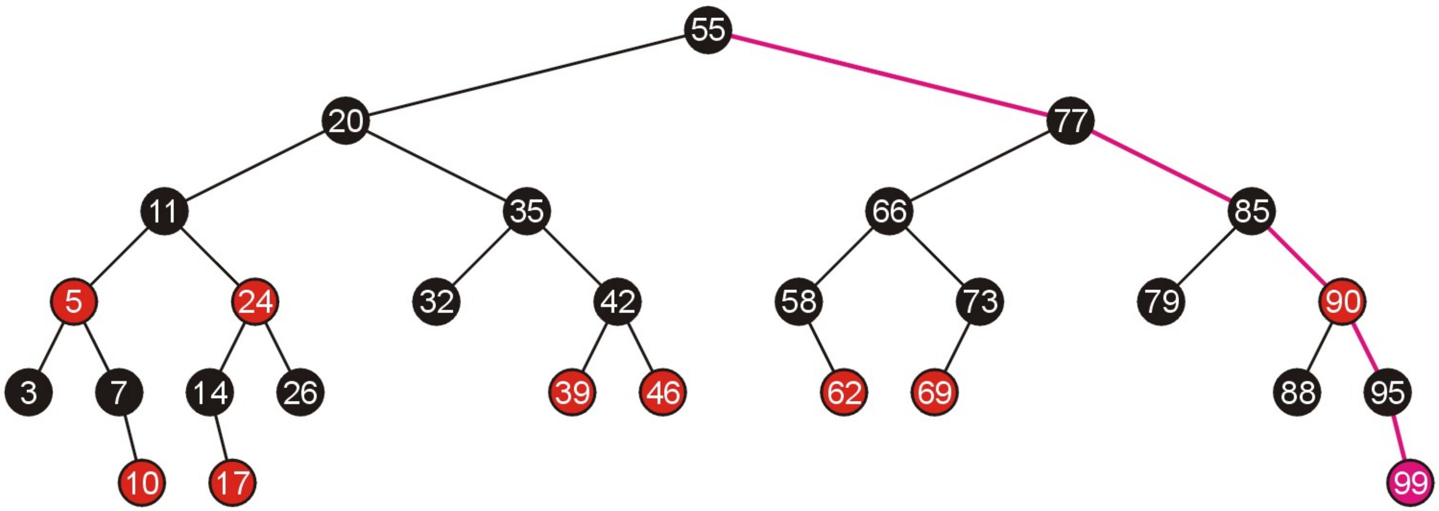
Examples of Top-Down Insertions: Insert 99

- We continue down the right to add 99



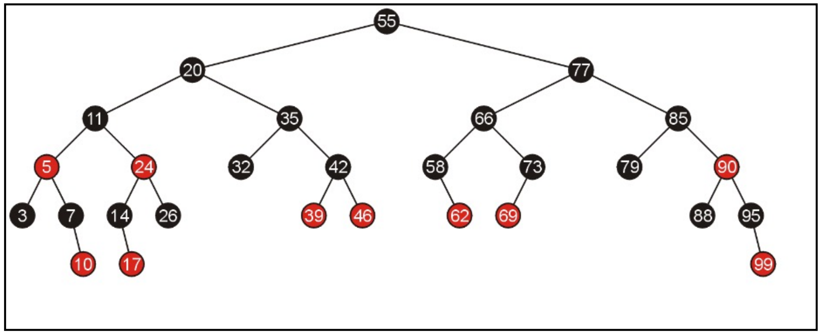
Examples of Top-Down Insertions: Insert 99

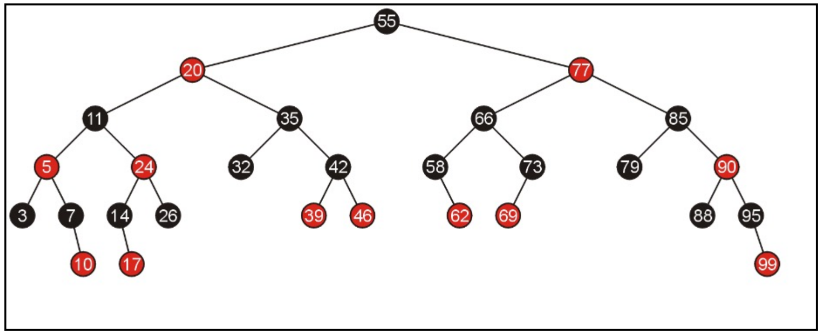
- This does not violate any of the rules of the red-black tree and therefore we are finished



Compare Top-Down and Bottom-up Insertions

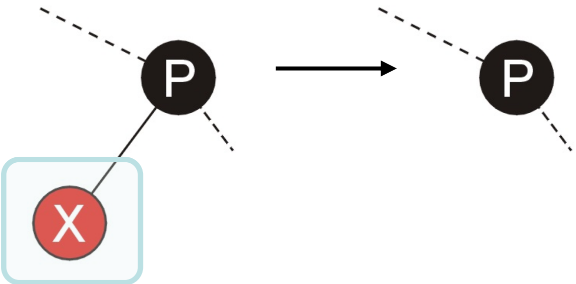
- Again, comparing the result of doing bottom-up insertions (left) and top-down insertions (right), we note the resulting trees are different, but both are still valid red-black trees



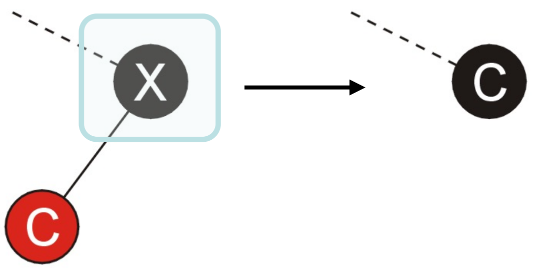


Top-Down Deletions: Easy cases

- If we are deleting a red leaf node X, then we are finished



- If we are deleting a node X with one child, we only need to replace the value of the deleted node with the value of the leaf node



Top-Down Deletions: Complex cases

- If we are deleting a full node, we use the same strategy used in standard binary search trees:
  - replace the node with the minimum element in the right sub-tree
  - then delete that element from the right sub-tree

Top-Down Deletions: Complex cases

- That minimum element must be:
  - Case #1: a red leaf node,
  - Case #2: a black node with a single red leaf node, or
  - Case #3: a black leaf *node*

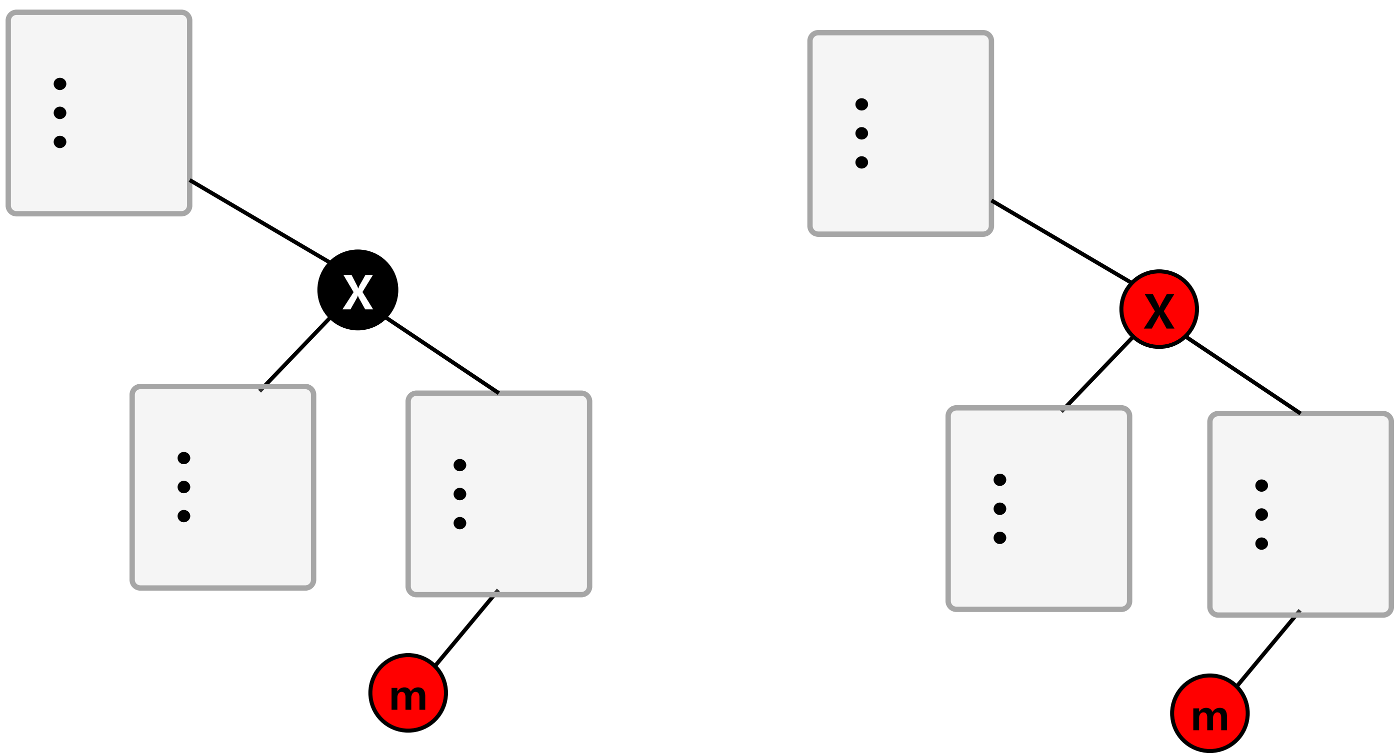
- The first two cases are easy to solve.

- For the last case, take the similar top-down insertion strategies.

- See why RBTree is difficult? You should handle all different cases (nicely).

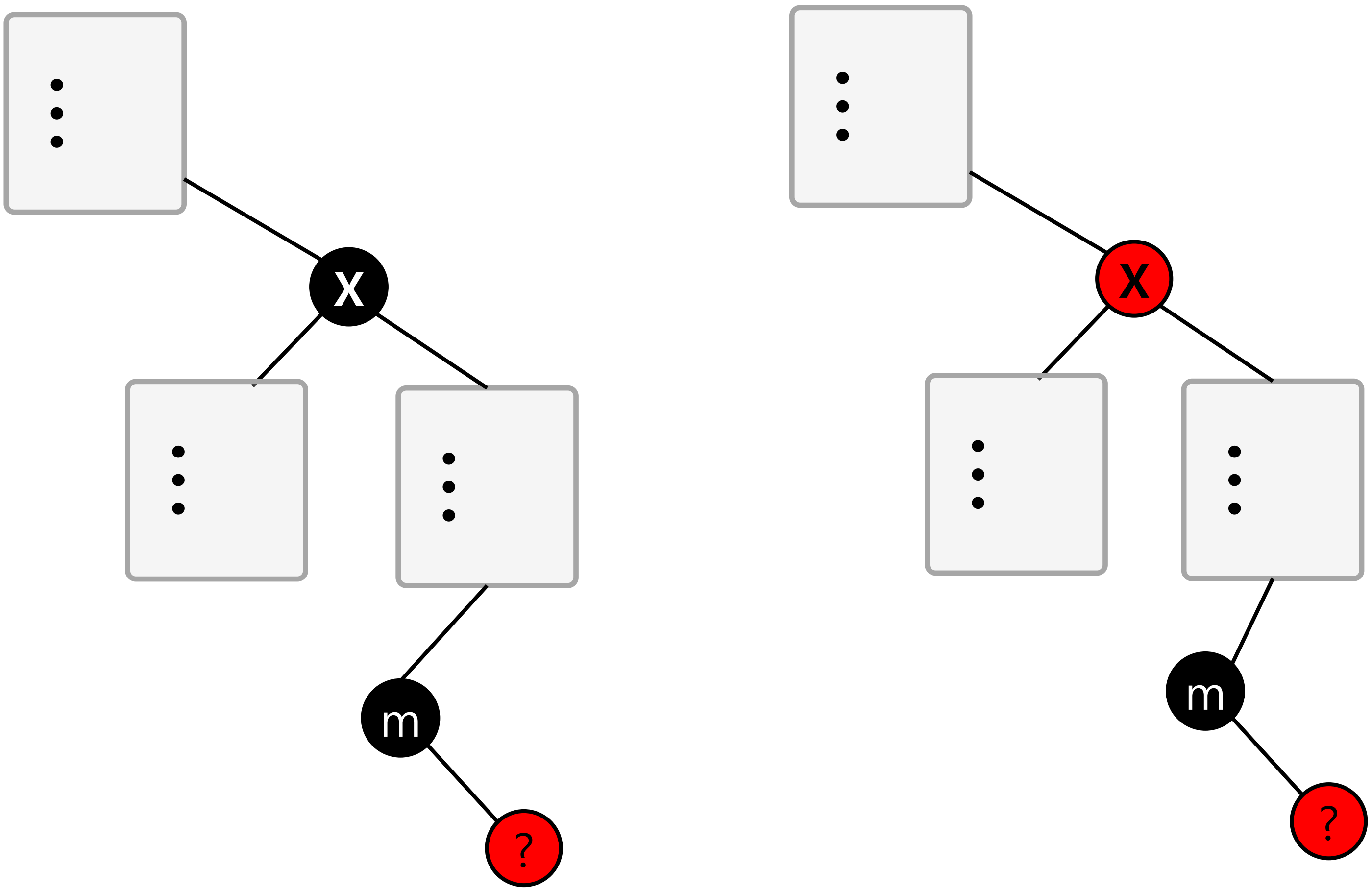
Top-Down Deletions: Complex cases

- That minimum element must be either:
  - Case #1: a red leaf node ➔ Easy to solve



Top-Down Deletions: Complex cases

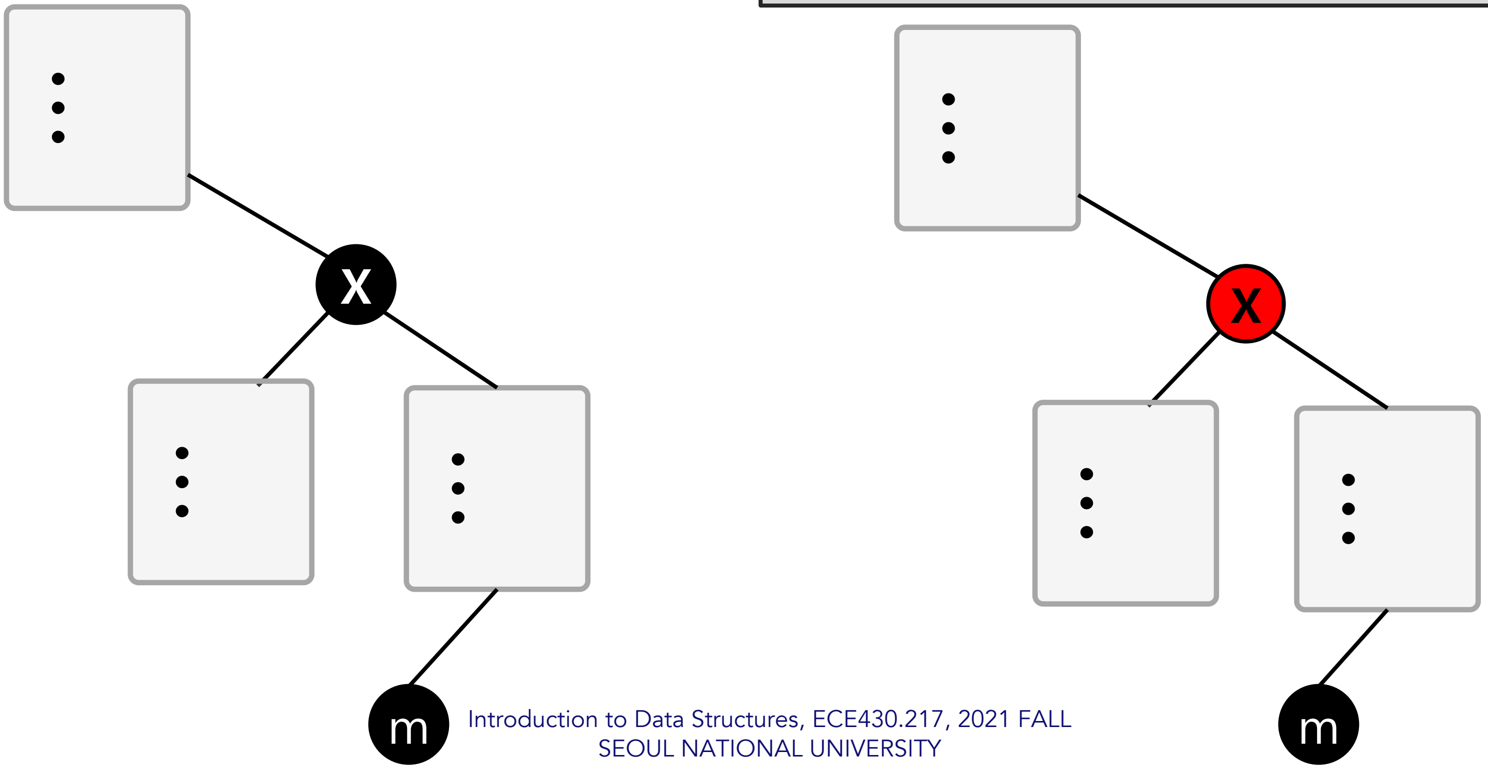
- That minimum element must be either:
  - Case #2: a black node with a single red leaf node ➔ Easy to solve



Top-Down Deletions: Complex cases

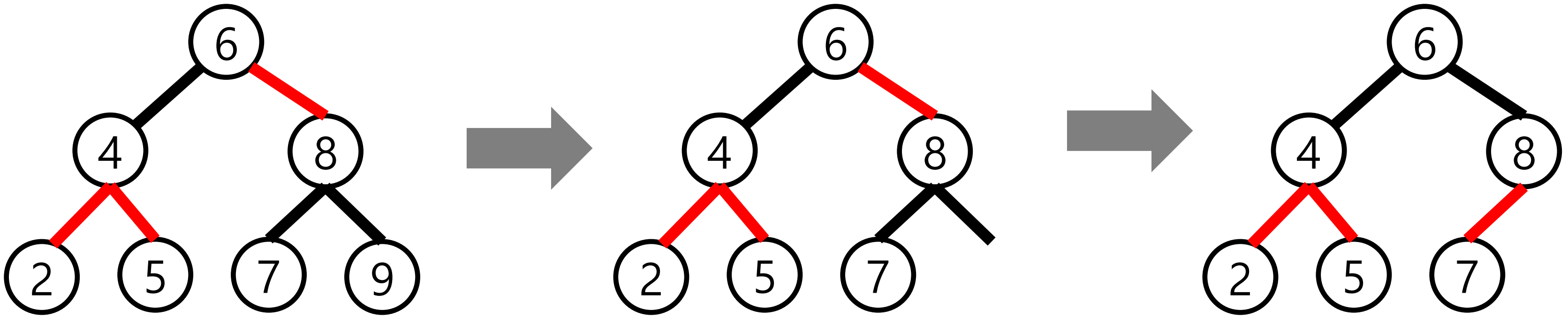
- That minimum element must be either:
  - Case #3: a black leaf *node*
    - ➔ take the similar top-down insertion strategies.

See why RBTree is difficult?
You should handle all different cases (nicely).



Top-Down Deletions: Complex cases

- Case # 3: Examples
  - Delete 9

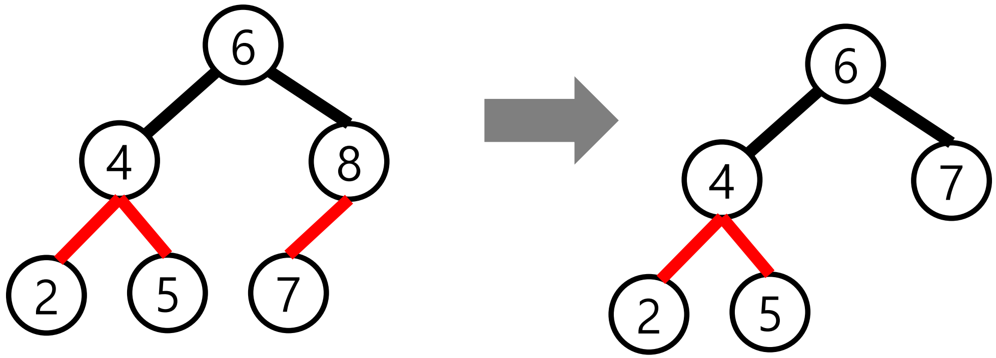


Remove 9, but the black height of node 8 becomes an issue

Swapping the color solves the problem

Top-Down Deletions: Complex cases

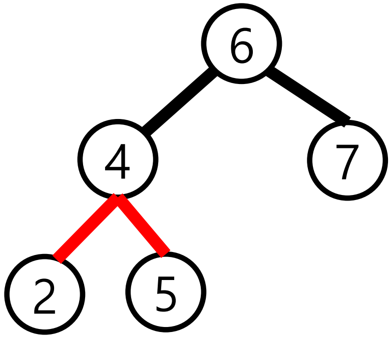
- Case # 3: Examples
  - Delete 8

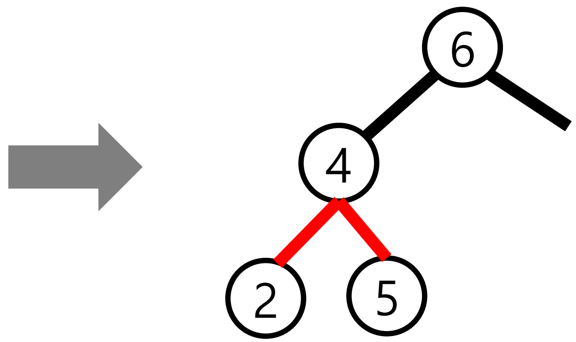


Deleting 8 is an easy case

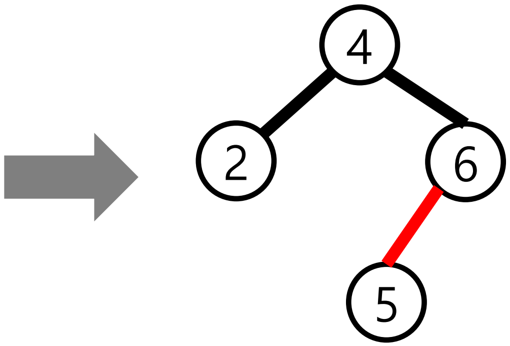
Top-Down Deletions: Complex cases

- Case # 3: Examples
  - Delete 7





Remove 7, then the black height of 6 becomes unbalanced



Rotate and recolor solves the problem

Red-Black Trees

- In this topic, we have covered red-black trees
  - simple rules govern how nodes must be distributed based on giving each node a color of either red or black
  - insertions and deletions may be performed without recursing back to the root
  - only one bit is required for the "color"
  - this makes them, under some circumstances, more suited than AVL trees

References

[1] Cormen, Leiserson, and Rivest, *Introduction to Algorithms*, McGraw Hill, 1990.
[2] Weiss, Data Structures and Algorithm Analysis in C++, 3rd Ed., Addison Wesley.

B-Tree and B+Tree

Kyunghan Lee
Networked Computing Lab (NXC Lab)
Department of Electrical and Computer Engineering
Seoul National University
https://nxc.snu.ac.kr
kyunghanlee@snu.ac.kr

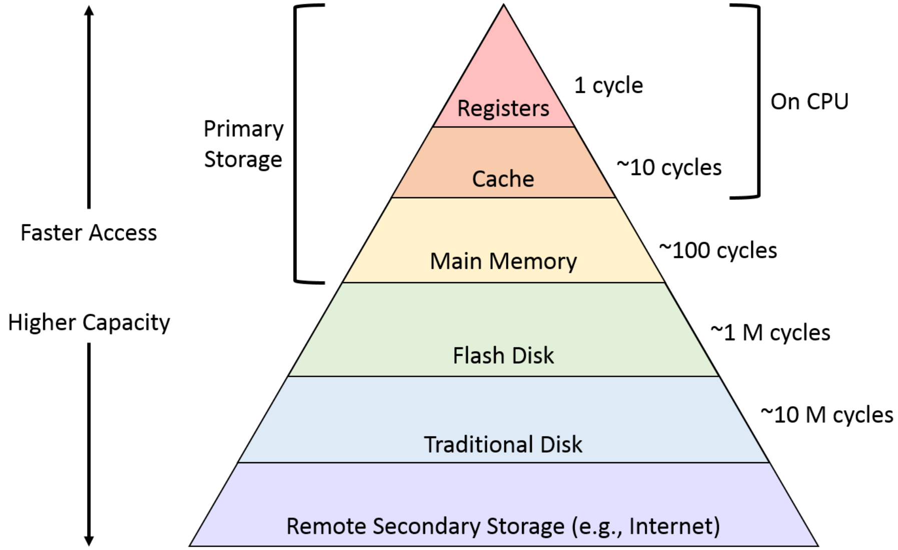
Outline

- Memory issues in designing data structures
- B-Tree
- B+Tree

Memory Considerations

- When we discuss data structures, we never specifically mention where the data would be stored.

- In fact, memory hierarchy suggests that the design of data structures should be well aware of it



The Memory Hierarchy

Memory and Data Structures

- Memory things to think about when designing data structures
  - Access speed
  - Cost per memory size
  - The unit size of access
  - Stream access vs. Random access

Tree for very large datasets

- Suppose you got very many pieces of information
  - e.g., $n = 2^{30}$
  - Suppose each piece has 1KB data
  - Examples
    - Student records, where each piece holds each student's report
    - Sales history, where each piece holds sale records per item

- If you design the tree to store such data
  - The number of nodes: 2^{30}
  - Each node will occupy at least 1KB
  - Total? At lest 1TB

```
class sales_node {
  ID item_id;
  RECORDS records; // 1KB
  sales_node *p_left_tree;
  sales_node *p_right_tree
  // ...
  // more
  // ...
}
```

Issues: Tree for very large datasets

- Two performance issues
  - #1: How many times do you need to access the memory?
    - Relevant to the height of a tree
    - Binary search tree
      - ✓ Best case
      - ✓ Worst case
    - AVL Tree
      - ✓ Best case
      - ✓ Worst case
  - #2: Can you store 1TB in the fast memory?
    - No, your main memory is (very likely) smaller than 1TB
    - So you will need to store the tree in the slow memory (i.e., a disk)

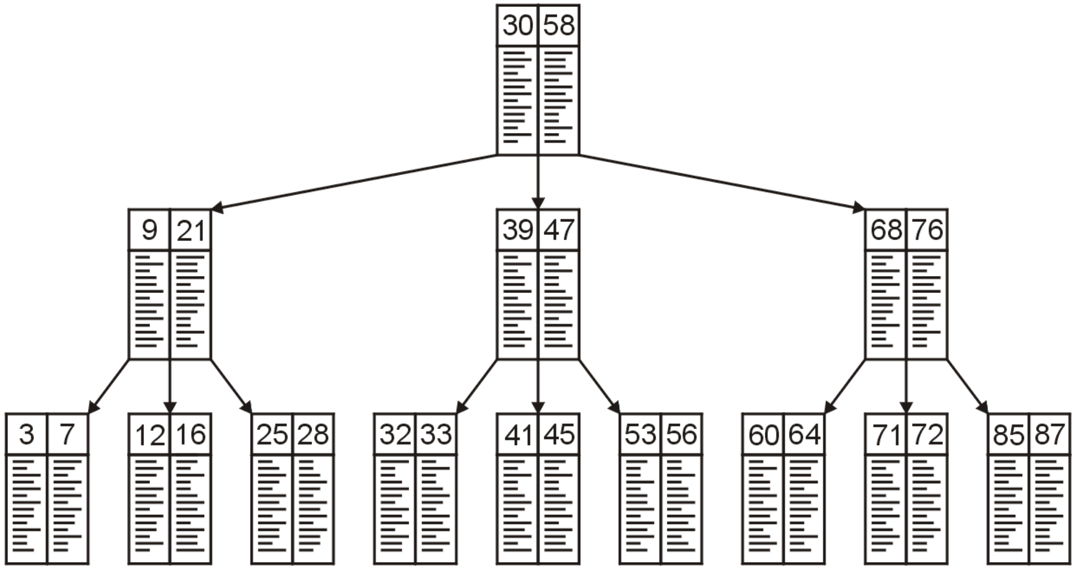
Ideas

- Solution to #1: Multiple keys per node
  - Load multiple keys at once
  - Reduce the height of the tree
  - Trees with this feature
    - M-Way Search Trees, B-Tree, B+Tree

- Solution to #2: No data in the internal nodes
  - Leaf nodes hold both keys/data, and internal nodes only hold keys
  - You "may" not need to access the slow memory when accessing the internal nodes
  - Trees with this feature
    - B+Tree

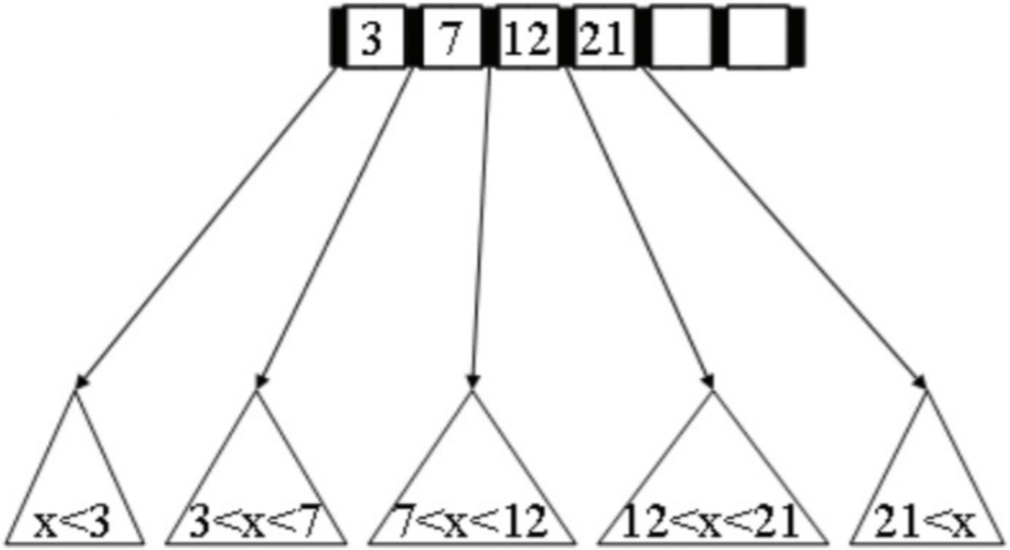
M-Way Search Trees

- M-Way Search Trees: A search tree with maximum branching factor M



B-Trees

- Each node has keys up to M-1 keys
- Order property
  - Subtree between two keys x and y contain leaves with values v such that $x < v < y$



B-Tree Structure Property ($M \geq 3$)

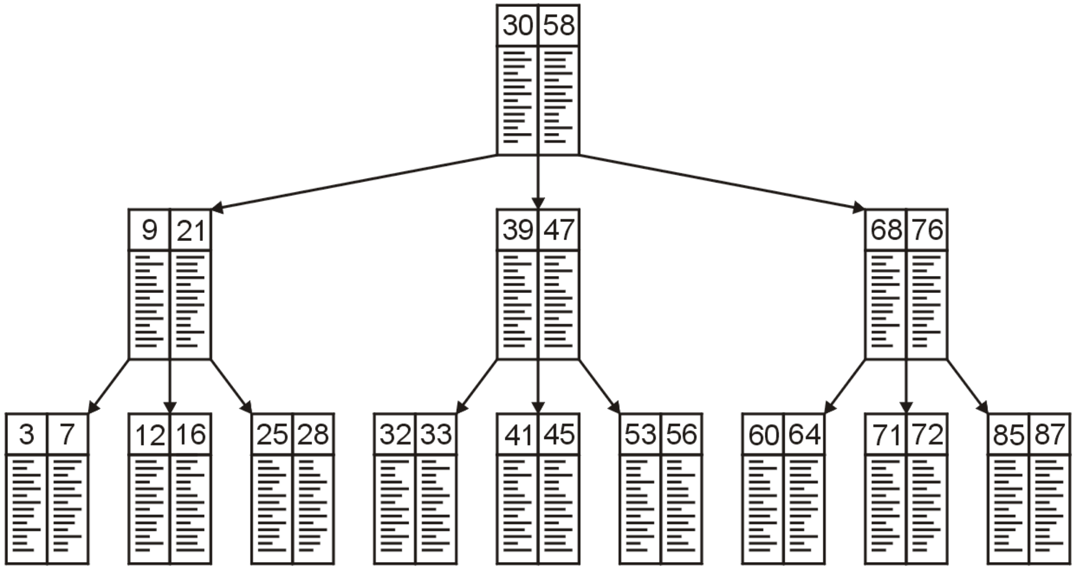
- Root (special case)
  - Has between 2 and M children (or root could be a leaf node)

- Internal nodes
  - Store up to $M - 1$ keys
  - Have between $\lceil M/2 \rceil$ and M children

- Leaf nodes
  - Store between $\lceil M/2 \rceil - 1$ and $M - 1$ sorted keys
  - All at the same depth

B-Tree: Example

- B-Tree with **M = 3**



B+Trees

- Internal nodes have no data
  - Only leaf nodes have data

- Each internal node still has (up to) $M - 1$ keys

- Order property
  - Subtree between two keys x and y contain leaves with values v such that $x \leq v < y$
  - Note the symbol, '≤'

- Leaf nodes have up to L sorted keys

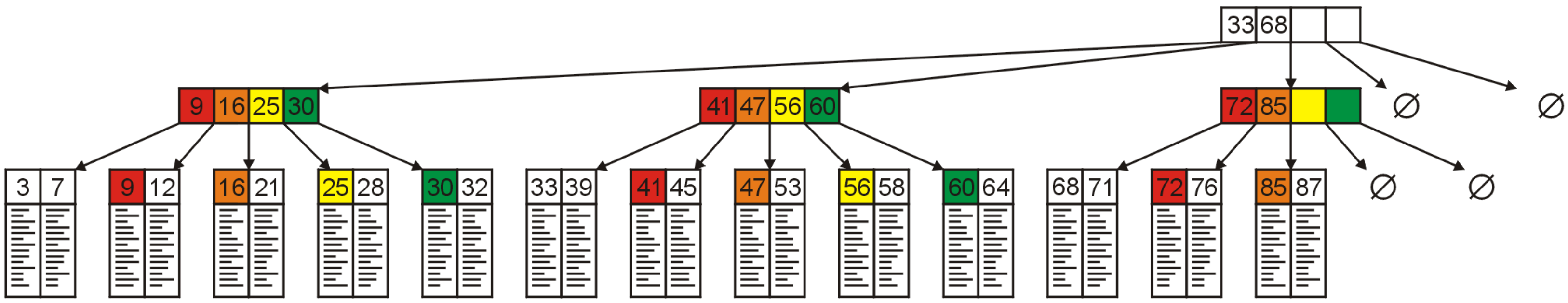
B+Tree Structure Property

- Root (special case)
  - Has between 2 and M children (or root could be a leaf node)

- Internal nodes
  - Store up to $M - 1$ keys
  - Have between $\lceil M/2 \rceil$ and M children

- Leaf nodes
  - Where data is stored
  - All at the same depth
  - Contain between $\lceil L/2 \rceil$ and L data items

B+Tree: Example



Disk Friendliness of B+Tree

- Many keys stored in a node
  - All brought to memory/cache in one disk access

- Internal nodes contain only keys
  - Only leaf nodes contain actual data
  - Much of tree structure can be loaded into memory irrespective of data object size
  - Data actually resides in disk

Comparison: B+Tree vs. AVL Tree

- Suppose again you have $n = 2^{30}$ items
  - AVL Tree
    - Height: 43
  - B+Tree where M=256, L=256
    - Height: 4.3

- If you consider other factors, things are getting more interesting
  - The size of each item
  - The size of Cache
  - The size of DRAM
  - …

- We never talked about the costs to balance the tree though

Maintain the Balance of B+ Tree

- How to make B+ Tree balanced?
  - Insertion idea (bottom-up approach)
    - Step 1: Insert an item to the leaf
    - Step 2: If the node overflows, 1) split the node and 2) add the key to the parent
    - Step 3: If the parent overflows, go back to step 2
    - You may need to increase the height
  - Deletion idea (bottom-up approach)
    - Step 1: Remove an item from the leaf
    - Step 2: If the node underflows, 1) adopt from (or merge with) the neighbor and 2) update the parent
    - Step 3: If the parent underflows, go back to step 2
    - You may need to decrease the height

Applications

- Databases
  - Index structure for MySQL
  - A hash table can be a better option (will cover later)

- File systems
  - Apple's HFS+, Microsoft NTFS, Linux's EXT4 and btrfs

- Real-world challenges in designing and implementing trees
  - Parallel access: Multi-core processors are everywhere
  - Distributed storage: Too large to store in a single computing node
  - You will learn more from advanced courses: operating systems, computer architecture, database, etc.

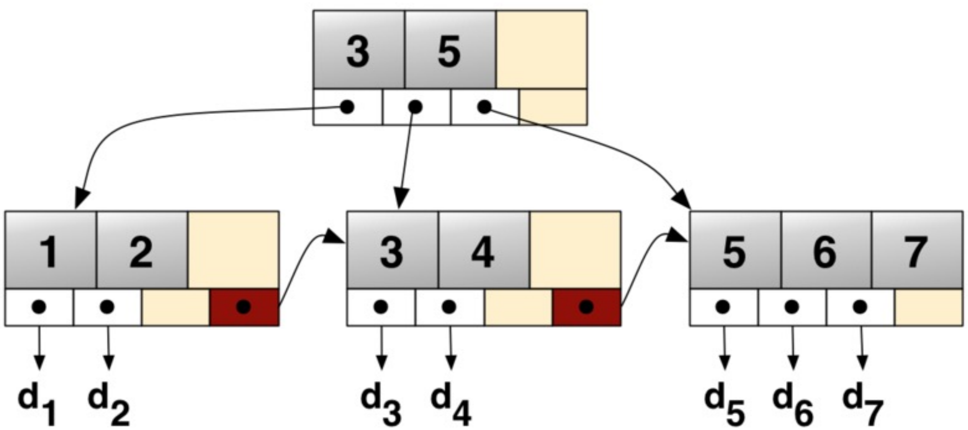
References

[1] Wikipedia, http://en.wikipedia.org/wiki/B+_tree

[2] Cormen, Leiserson, and Rivest, *Introduction to Algorithms*, McGraw Hill, 1990, Ch. 19, p.381-99.

[3] Weiss, *Data Structures and Algorithm Analysis in C++, 3rd Ed.*, Addison Wesley, §4.7, p.159-64.

[4] Lecture slides by Brian Curless, https://courses.cs.washington.edu/courses/cse326/08sp/lectures/markup/11-b-trees-markup.pdf



B+Tree: Leafs are linked listed [1]