08

Week of March 23rd, 2020

Binary Trees, AVL Trees, and Tree Traversals

Announcements

- Project 3 due **March 28th** at **11:59pm**.
- Lab 7 due **March 27th** at **11:59pm**.
- Lab 8 due **April 3rd** at **11:59pm.**
  - **Autograder** and **Quiz** for Lab 8
- Make sure to submit the written problem to Gradescope by Friday **3/27**!

Last Week's Handwritten Problem

Prefixes are words that can be followed by some other letters to form a longer word - let's call this final word the successor. For example, the prefix "an" followed by "other" forms the word "another".

Now, given a dictionary consisting of many prefixes and a sentence, you need to replace all the successors in the sentence with the prefix forming it. If a successor has many prefixes that can form it, replace it with the shortest length.

The input will only have lower-case letters. Return the new sentence in a vector of strings.
P prefixes, **N** words, **M** length: **O(PM + NM^2) (Hashing/looking up a string of length M costs O(M))**

Example:
Prefixes: ["cat", "bat", "rat"]
Sentence: ["the", "cattle", "was", "rattled", "by", "the", "battery"]
Output: ["the", "cat", "was", "rat", "by", "the", "bat"]

```
vector<string>
replace_words(const vector<string>& prefixes,
              const vector<string>& sentence);
```

Common Mistakes

- unordered_map instead of unordered_set
- not using range-based constructor
- making a new substring each time, rather than having a running substring to add to char by char
- forgetting to return result
- forgetting to add non-replaced words
- not correctly choosing the smallest root to replace
- modifying the sentence vector - it's CONST reference!

Handwritten Solution

```
vector<string> replace_words(const vector<string>& prefixes,
                             const vector<string>& sentence) {
    unordered_set<string> set(prefixes.begin(), prefixes.end()); // O(MR)
    vector<string> output;
    for (const string& word : sentence) {      // N iterations {
        string prefix;                          //
        for (char c : word) {                   //    M iterations {
            prefix.push_back(c);                //
            if (set.find(prefix) != set.end())  //       O(M)
                break;                          //
        }                                       //    }
        output.push_back(prefix);               //    O(M)
    }                                           // }
    return output;
}
```

Agenda

- Tree Traversals
- Binary Search Trees
- AVL Trees
- Programming Problem
- Handwritten Problem

- Slides on https://preetiramaraj.github.io/eecs_281/lab8.pdf
- Preeti's Lab 8 OH on Tuesday, 03/24/2020 from 3:30-5:30pm EDT.

Tree Traversals

Tree Terminology

- Root: node with no parents
- Leaf: node with no children
- Internal Node: node with children (including root)
- Depth: distance from a node to the root
- Height: distance from a node to the lowest leaf node
- Siblings: nodes with the same parent node

Warm-Up Question

Given a binary tree with following declaration, find the minimum depth of the binary tree (aka the depth of the shallowest leaf node)

```
struct Node {
   Node* left;
   Node* right;
   int val;
};

int minimum_depth(Node* root);
```

Warm-Up Question Solution

Given a binary tree with following declaration, find the minimum depth of the binary tree (aka the depth of the shallowest leaf node)

```
int minimum_depth(Node* root) {
    if (!root)
        return 0;
    else if (!root->left)
        return minimum_depth(root->right) + 1;
    else if (!root->right)
        return minimum_depth(root->left) + 1;
    else
        return min(minimum_depth(root->left),
                   minimum_depth(root->right)) + 1;
}
```

Tree Traversal

Parent = P, Left Child = L, Right Child = R

- Pre-order: PLR
- Post-order: LRP
- In-order: LPR
- Level-order: Traverse all nodes of a level starting at the root and descending in level, traversing from left to right

Tree Traversal

Parent = P, Left Child = L, Right Child = R

- Pre-order: PLR (Explore all nodes first - top-down recursion)
- Post-order: LRP (Explore all leaves first - bottom-up recursion)
- In-order: LPR (flatten back to original insertion sequence)
- Level-order: Traverse all nodes of a level starting at the root and descending in level, traversing from left to right

https://stackoverflow.com/questions/9456937/when-to-use-preorder-postorder-and-inorder-binary-search-tree-traversal-strate

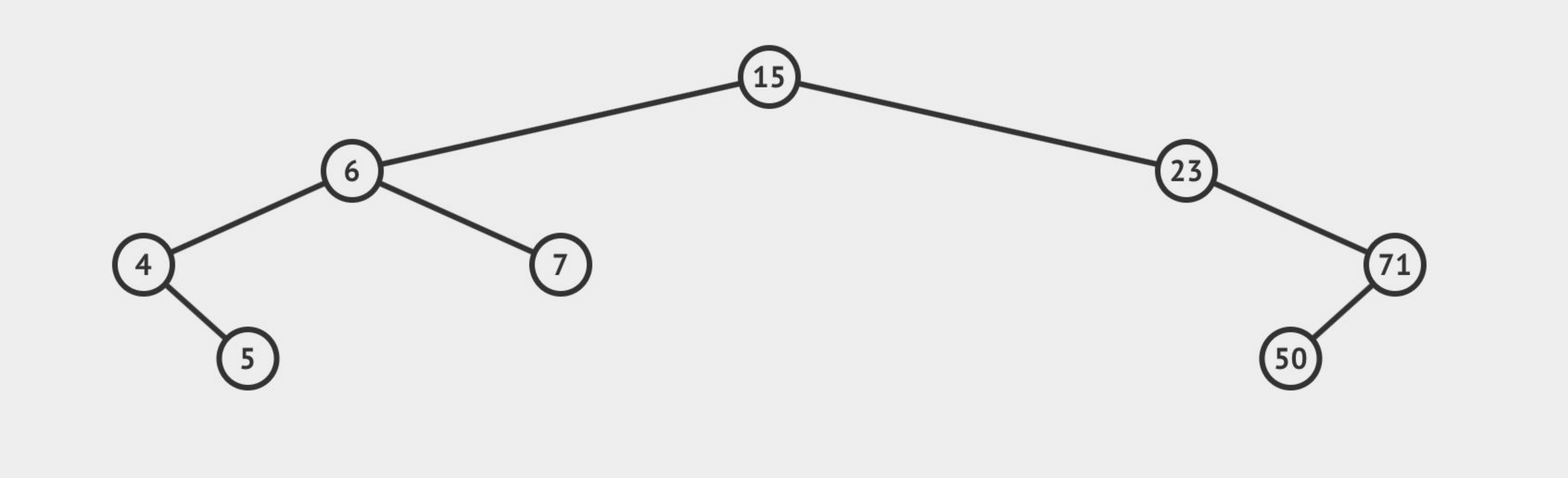
Recursive Tree Traversal

```
void traversal( Node * head ) {
    if( !head ) return;
    // code for pre-order: ( visit head node )
    traversal( head->left );
    // code for in-order: ( visit head node )
    traversal( head->right );
    // code for post-order: ( visit head node )
}
```

Pre-order Traversal

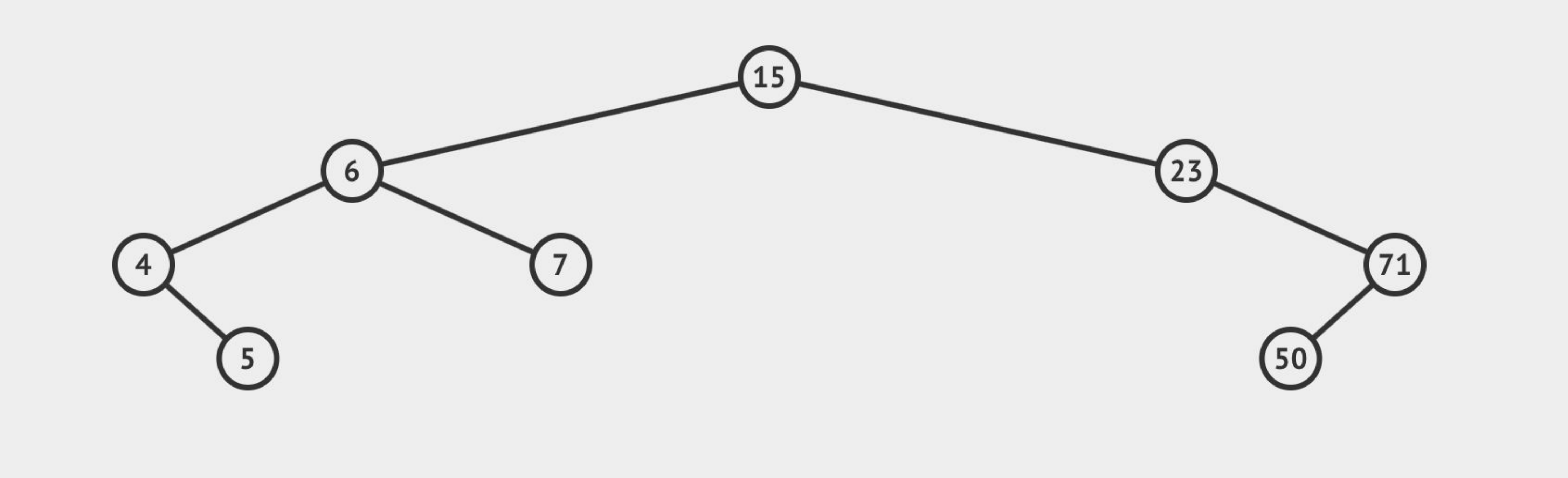
```
void traversal( Node * head ) {
    if( !head ) return;

    printNode( head );
    traversal( head->left );
    traversal( head->right );
}
```

Pre-order Traversal (PLR)



What is the pre-order traversal of this tree?

Pre-order Traversal (PLR)



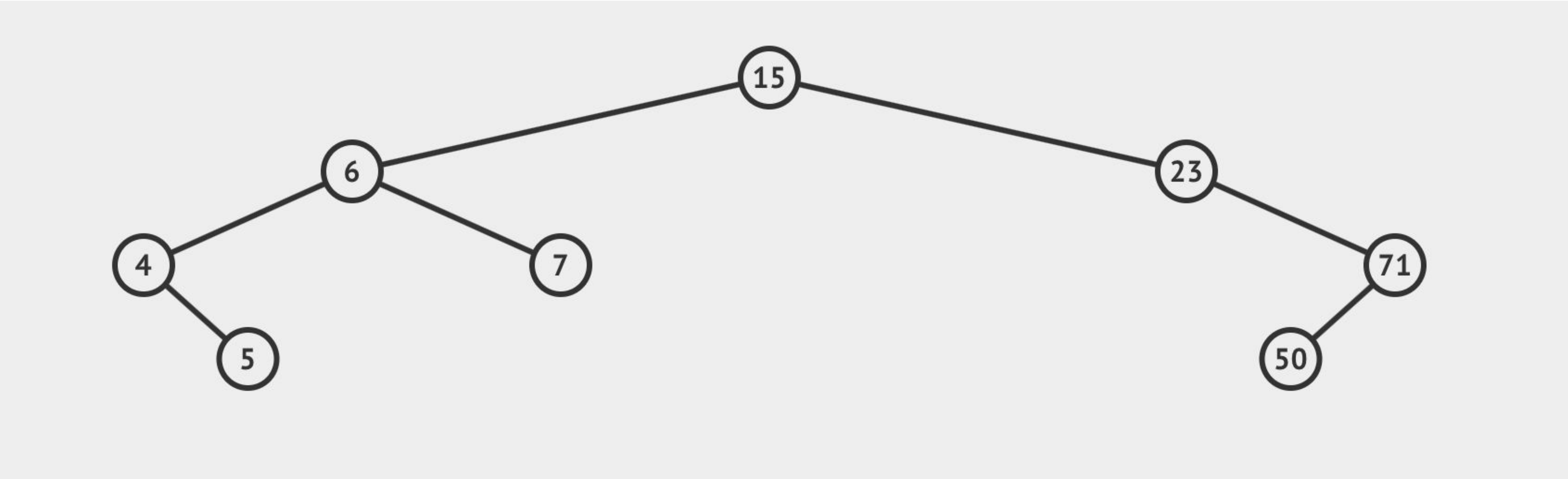
What is the pre-order traversal of this tree?

15, 6, 4, 5, 7, 23, 71, 50

Post-order Traversal

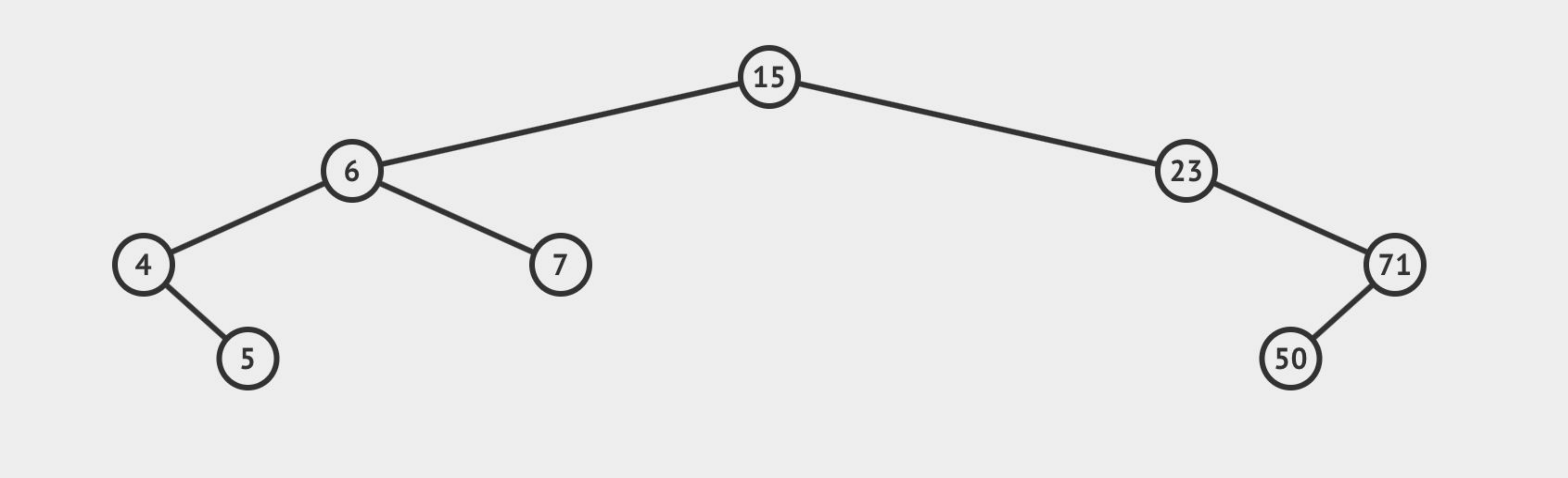
```
void traversal( Node * head ) {
    if( !head ) return;

    traversal( head->left );
    traversal( head->right );
    printNode( head );
}
```

Post-order Traversal (LRP)



What is the post-order traversal of this tree?

Post-order Traversal (LRP)



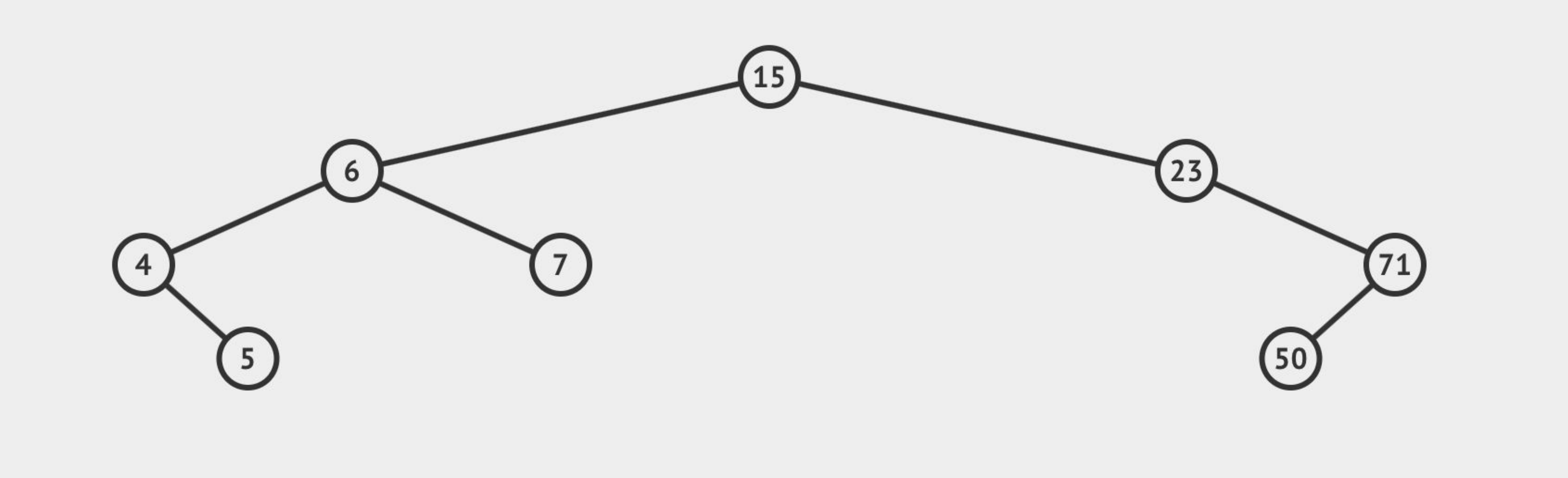
What is the post-order traversal of this tree?

5, 4, 7, 6, 50, 71, 23, 15

In-order Traversal

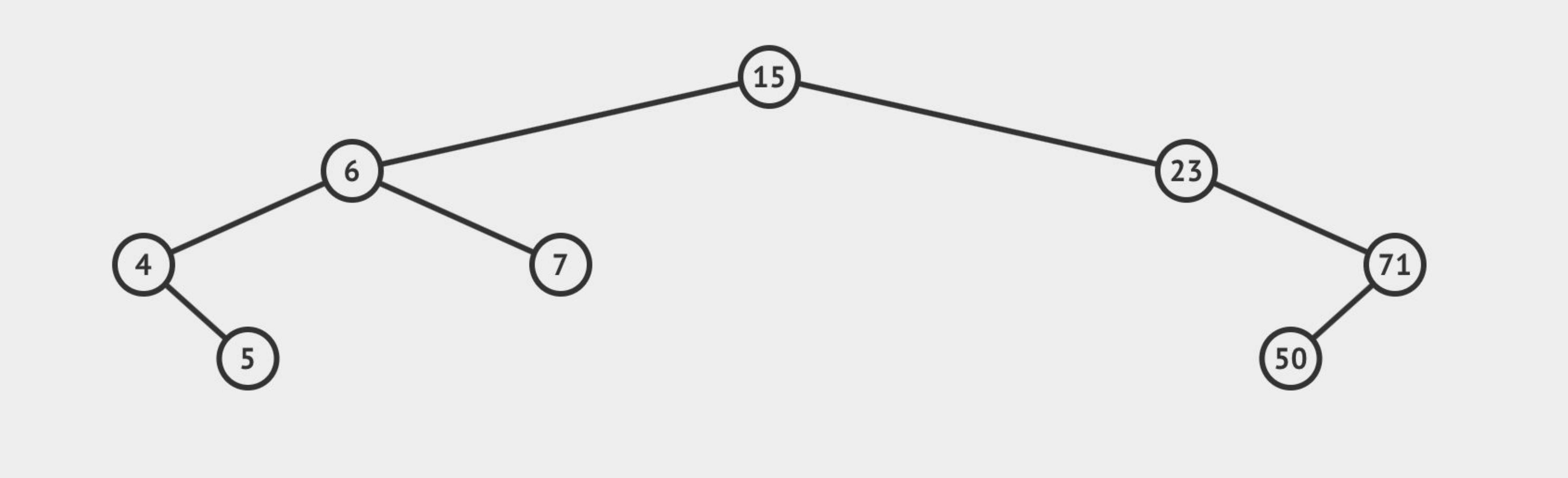
```
void traversal( Node * head ) {
    if( !head ) return;

    traversal( head->left );
    printNode( head );
    traversal( head->right );
}
```

In-order Traversal (LPR)



What is the in-order traversal of this tree?

In-order Traversal (LPR)



What is the in-order traversal of this tree?

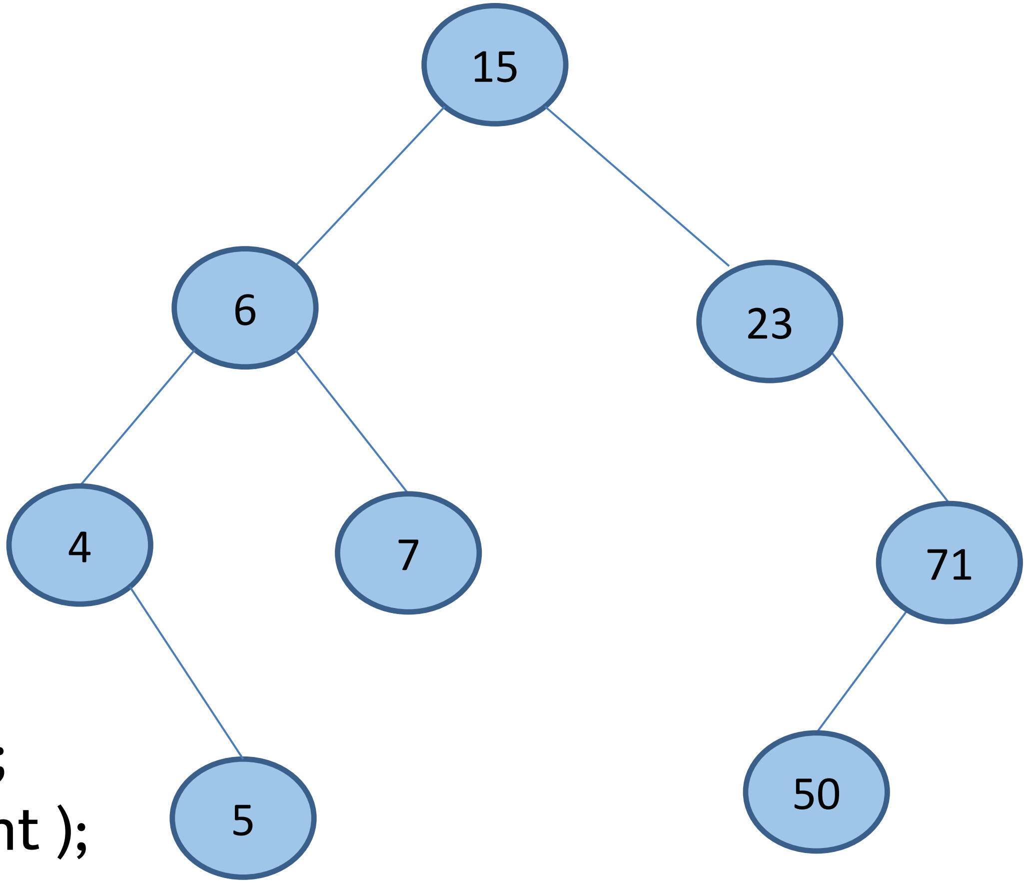
4, 5, 6, 7, 15, 23, 50, 71

Level-order Traversal

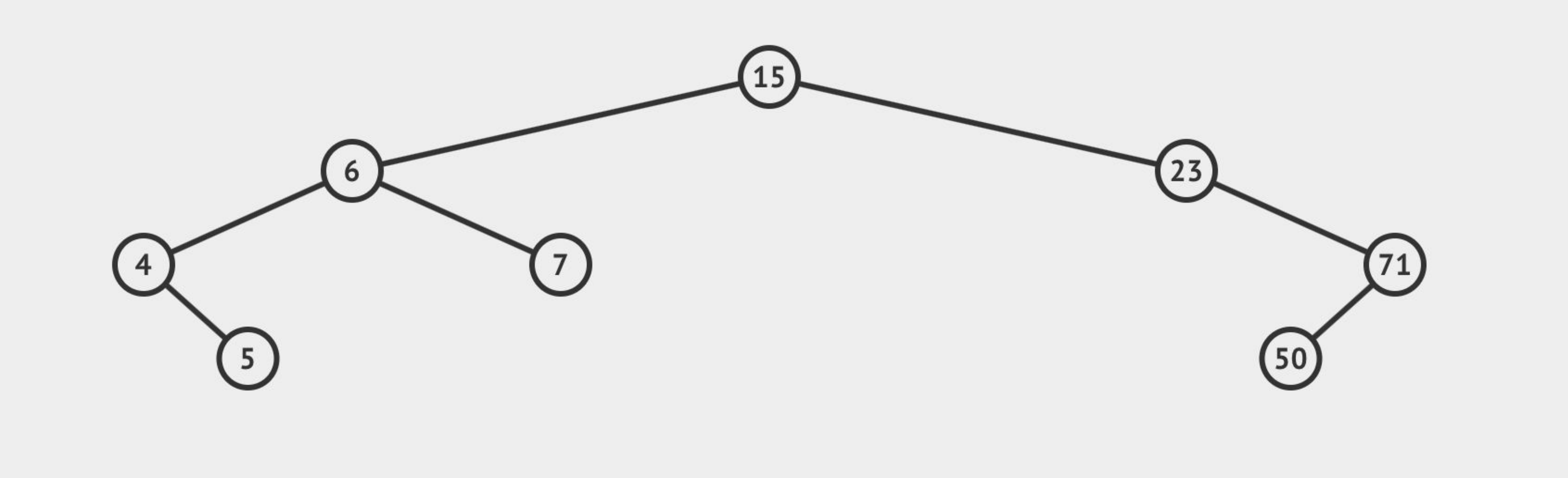
```
void traversal( Node * head ) {
    if( !head ) return;

    queue< Node * > level_queue;
    level_queue.push( head );

    while( !level_queue.empty() ){
        Node * top = level_queue.~~top~~();
                                front
        level_queue.pop();
        printNode( top );

        if( top->left ) level_queue.push( top->left );
        if( top->right ) level_queue.push( top->right );
    }
}
```

Level-order Traversal

```
void traversal( Node * head ) {
    if( !head ) return;

    queue< Node * > level_queue;
    level_queue.push( head );

    while( !level_queue.empty() ){
        Node * top = level_queue.~~top~~();
                                front
        level_queue.pop();
        printNode( top );

        if( top->left ) level_queue.push( top->left );
        if( top->right ) level_queue.push( top->right );
    }
}
```

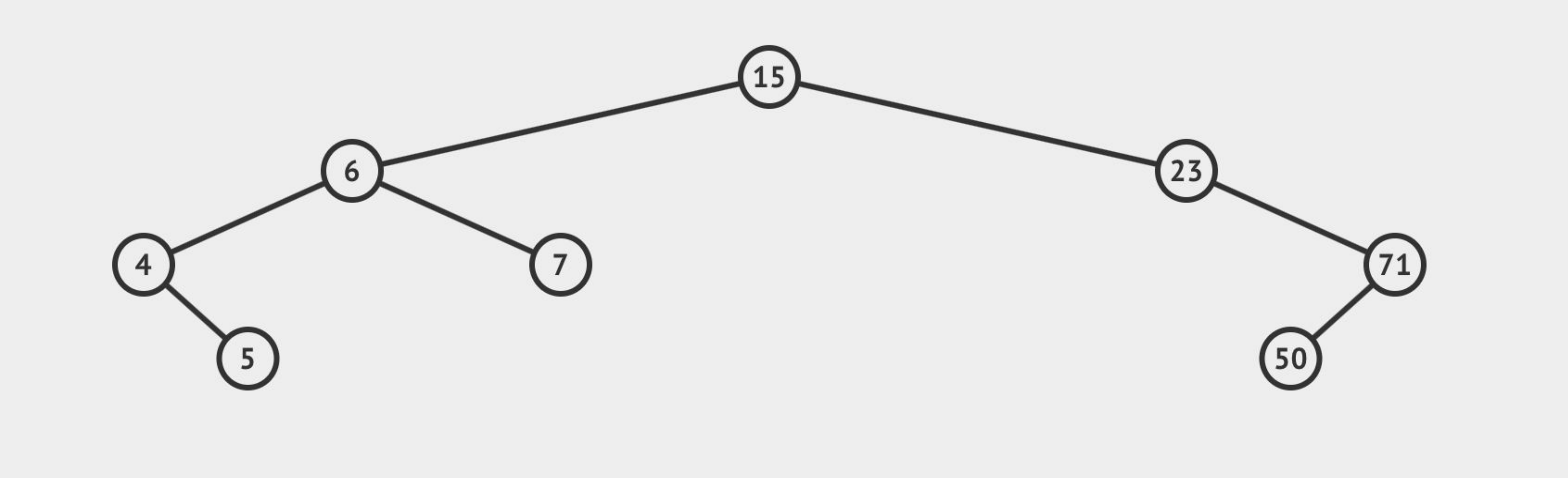


Level-order Traversal



What is the level-order traversal of this tree?

Level-order Traversal



What is the level-order traversal of this tree?

15, 6, 23, 4, 7, 71, 5, 50

Practice: Minimum Sum in a Tree

Given a root to a binary tree, find the **level** of the tree with the minimum sum. The binary tree is not guaranteed to be complete.

Time complexity: O(n)
Memory Complexity: O(logn) average, O(n) worst case

Example: Answer is level 1 (sum = 8)

```
      50
     /  \
    6    2
   / \  /
 30 80 7
```

```
int minimum_sum(Node * root) {
    int minimum_level = 0;
    int level = 0;
    int minimum_sum = std::numeric_limits<int>::max();       // start min at inf
    queue<Node *> q;

    q.push(root);

    while (!q.empty()) {
        int level_size = q.size();                        // snapshot of queue holds a full level
        int level_sum = 0;                                // reset level sum
        for (int i = 0; i < level_size; ++i) {
            Node * temp = q.front(); q.pop();
            level_sum += temp->elem;                          // add element to the level sum
            if (temp->left) q.push(temp->left);
            if (temp->right) q.push(temp->right);         // push on its children
        }
        if (level_sum < minimum_sum) {                        // update minimum
            minimum_sum = level_sum;
            minimum_level = level;
        }
        ++level;                                          // update level
    }
    return minimum_level;
}
```

Tree Reconstruction

Reconstruct a Tree

Given the following traversals, draw a tree that would match the traversal results.

In-order: 4, 8, 2, 5, 1, 6, 3, 7

Post-order: 8, 4, 5, 2, 6, 7, 3, 1

Reconstruct a Tree

In-order: 4, 8, 2, 5, 1, 6, 3, 7

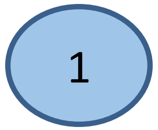
Post-order: 8, 4, 5, 2, 6, 7, 3, 1

What do we know about the last element in the post-order (or the first element in the pre-order)?

Reconstruct a Tree

In-order: 4, 8, 2, 5, 1, 6, 3, 7

Post-order: 8, 4, 5, 2, 6, 7, 3, 1



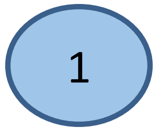
What do we know about the last element in the post-order (or the first element in the pre-order)?

It's the root!

Reconstruct a Tree

In-order: 4, 8, 2, 5, 1, 6, 3, 7

Post-order: 8, 4, 5, 2, 6, 7, 3, 1

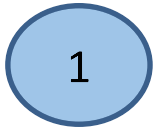


What do we know about the elements to the left and right of a node in the in-order traversal?

Reconstruct a Tree

In-order: 4, 8, 2, 5, 1, 6, 3, 7

Post-order: 8, 4, 5, 2, 6, 7, 3, 1



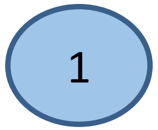
What do we know about the elements to the left and right of a node in the in-order traversal?

Elements to the left are in its left subtree
Elements to the right are in its right subtree

Reconstruct a Tree

In-order: 4, 8, 2, 5, 1, 6, 3, 7

Post-order: 8, 4, 5, 2, 6, 7, 3, 1

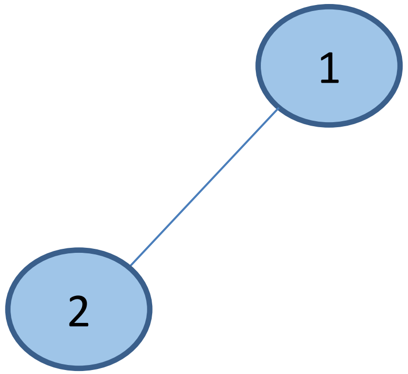


Let's just look at its left subtree for now
What is the root of its left subtree?

Reconstruct a Tree

In-order: 4, 8, 2, 5, 1, 6, 3, 7

Post-order: 8, 4, 5, 2, 6, 7, 3, 1



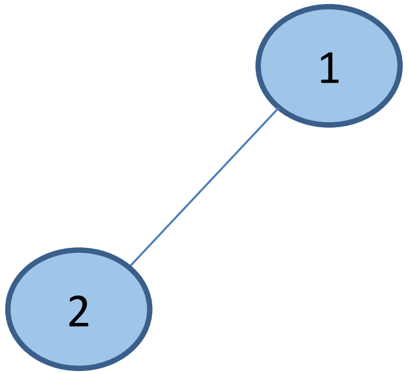
Let’s just look at its left subtree for now
What is the root of its left subtree?

2, because it’s the last of those elements in the post-order

Reconstruct a Tree

In-order: 4, 8, 2, 5, 1, 6, 3, 7

Post-order: 8, 4, 5, 2, 6, 7, 3, 1



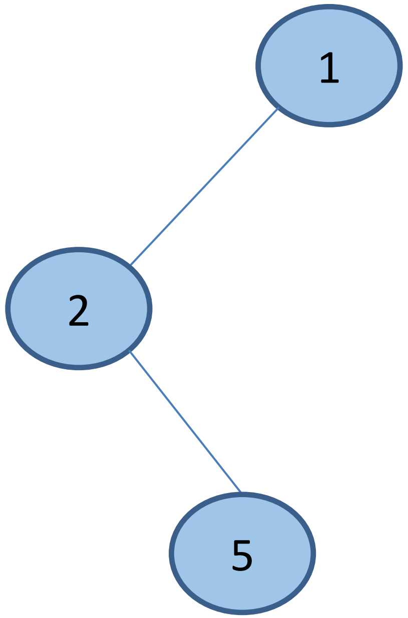
Split in-order at 2 and repeat!

Reconstruct a Tree

In-order: 4, 8, 2, 5, 1, 6, 3, 7

Post-order: 8, 4, 5, 2, 6, 7, 3, 1

5 is to the right of 2

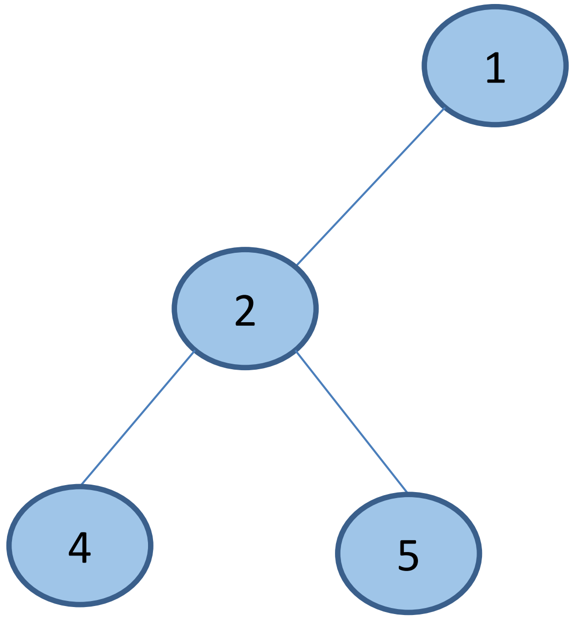


Reconstruct a Tree

In-order: 4, 8, 2, 5, 1, 6, 3, 7

Post-order: 8, 4, 5, 2, 6, 7, 3, 1

4 is after 8, so it's the root of 2's left subtree

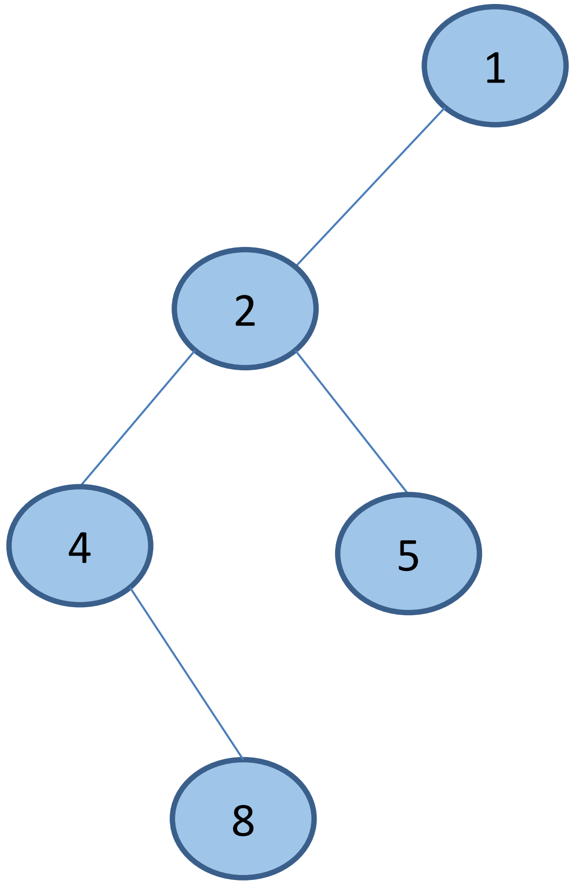


Reconstruct a Tree

In-order: 4, 8, 2, 5, 1, 6, 3, 7

Post-order: 8, 4, 5, 2, 6, 7, 3, 1

8 is to the right of 4

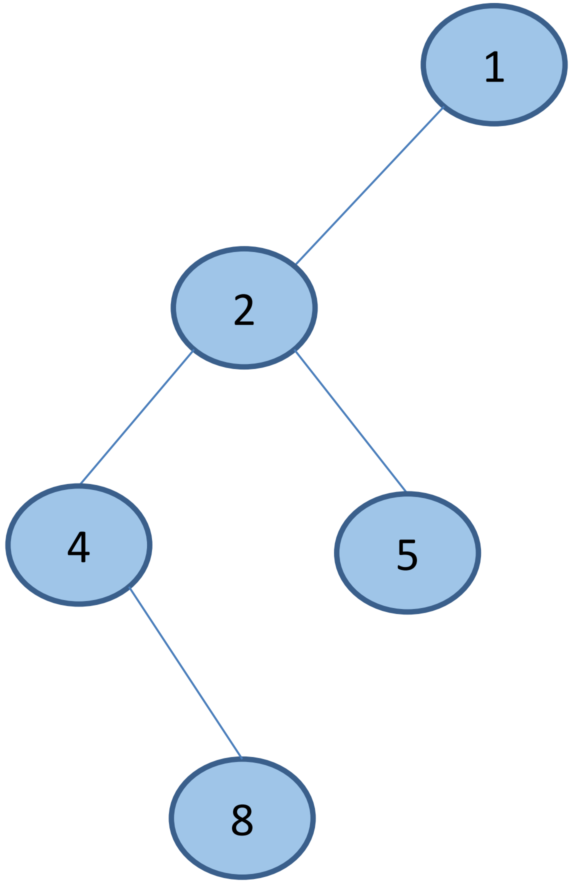


Reconstruct a Tree

In-order: 4, 8, 2, 5, 1, 6, 3, 7

Post-order: 8, 4, 5, 2, 6, 7, 3, 1

Done with 1's left subtree!
Let's grow its right one

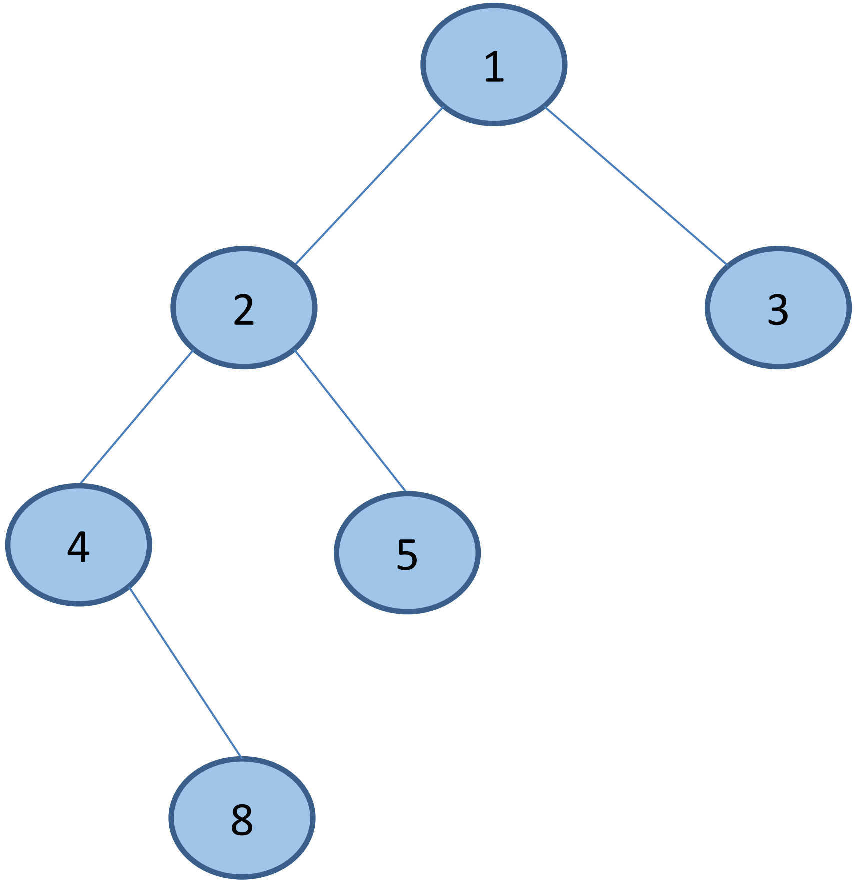


Reconstruct a Tree

In-order: 4, 8, 2, 5, 1, 6, 3, 7

Post-order: 8, 4, 5, 2, 6, 7, 3, 1

3 is the root node

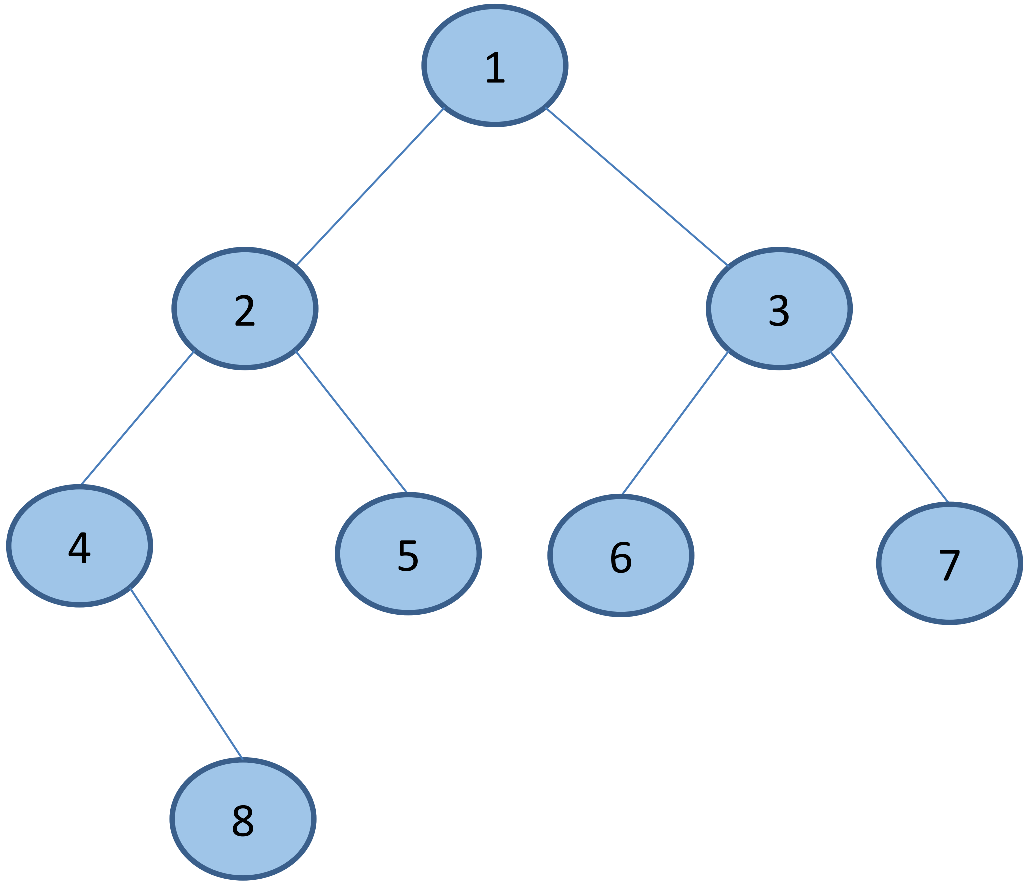


Reconstruct a Tree

In-order: 4, 8, 2, 5, 1, 6, 3, 7

Post-order: 8, 4, 5, 2, 6, 7, 3, 1

6 is to the left of 3 and 7 is to the right

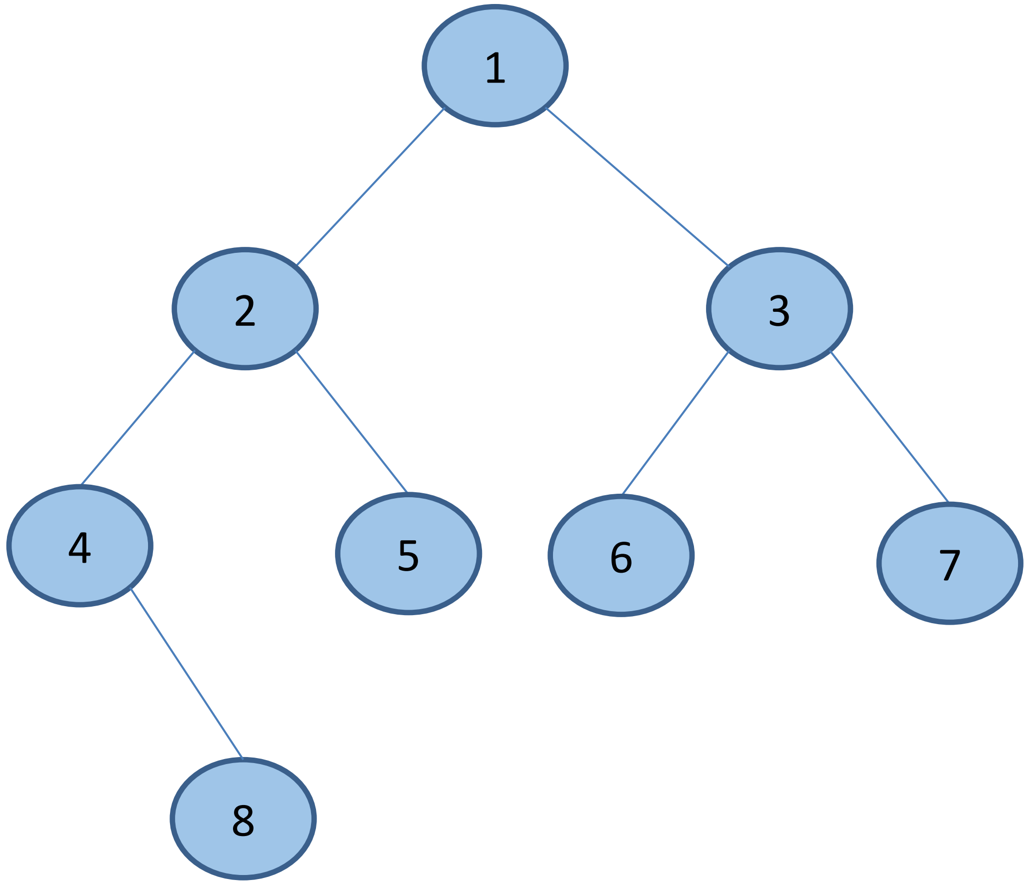


Reconstruct a Tree

In-order: 4, 8, 2, 5, 1, 6, 3, 7

Post-order: 8, 4, 5, 2, 6, 7, 3, 1

All done!



Binary Search Trees

Binary Search Trees

The key of any node is:

- > the keys of all nodes in its left subtree
- <= the keys of all nodes in its right subtree

Why do we use them? So that we can easily search for and insert items!

Binary Search Trees

The key of any node is:

- > the keys of all nodes in its left subtree
- <= the keys of all nodes in its right subtree

Why do we use them? So that we can easily search for and insert items!

Insertion time for best case/worst case/average case? O(1), O(n), O(log n)

Lookup time for best case/worst case/average case? O(1), O(n), O(log n)

BST Insertion & Deletion

Insertion - Average O(logn); Worst Case O(n)

Start at root and traverse downwards (based on node's value) until a spot to append the node is found

BST Insertion & Deletion

Insertion - Average O(logn); Worst Case O(n)

Start at root and traverse downwards (based on node's value) until a spot to append the node is found

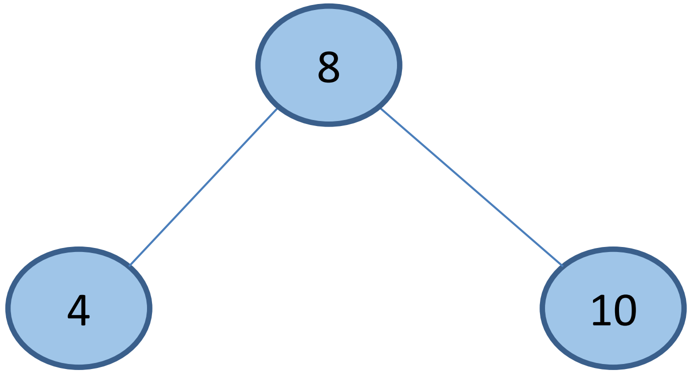
Deletion - Average O(logn); Worst Case O(n)

1. If the node has 1 child:
   - replace it with its child and delete child
2. If the node has 2 children:
   - replace it with its in-order successor (or predecessor)
   - remove the in-order node from its original spot in tree and replace it with its child if it has one

Binary Search Trees - Insert

Insert the following to the BST:

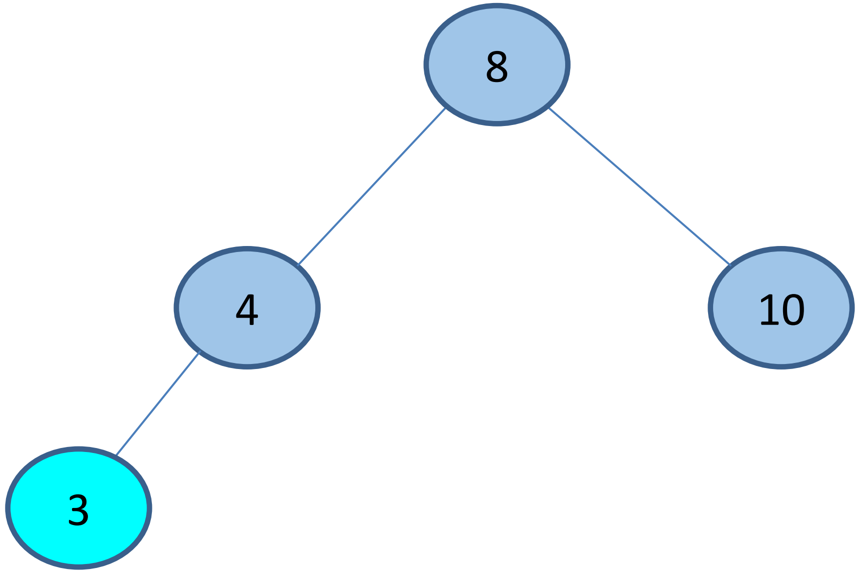
3 5 6 9



Binary Search Trees - Insert

Insert the following to the BST:

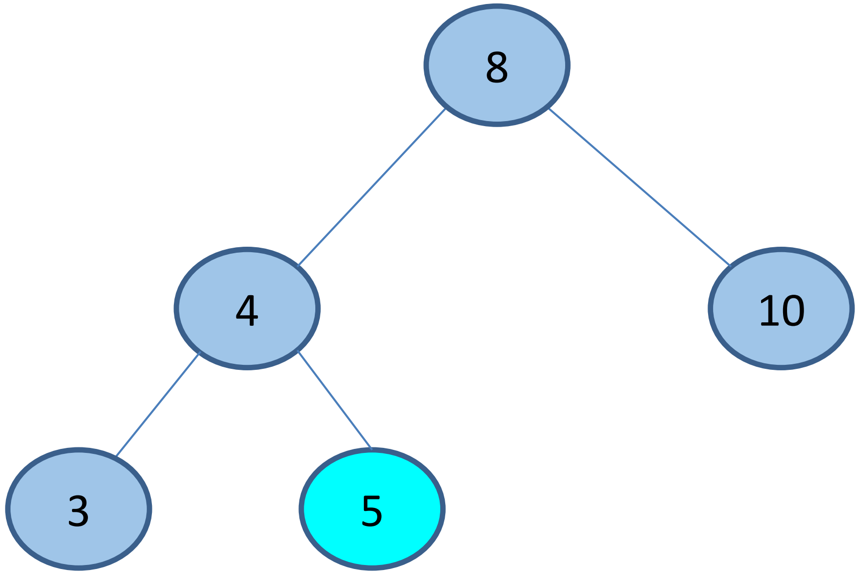
3 5 6 9



Binary Search Trees - Insert

Insert the following to the BST:

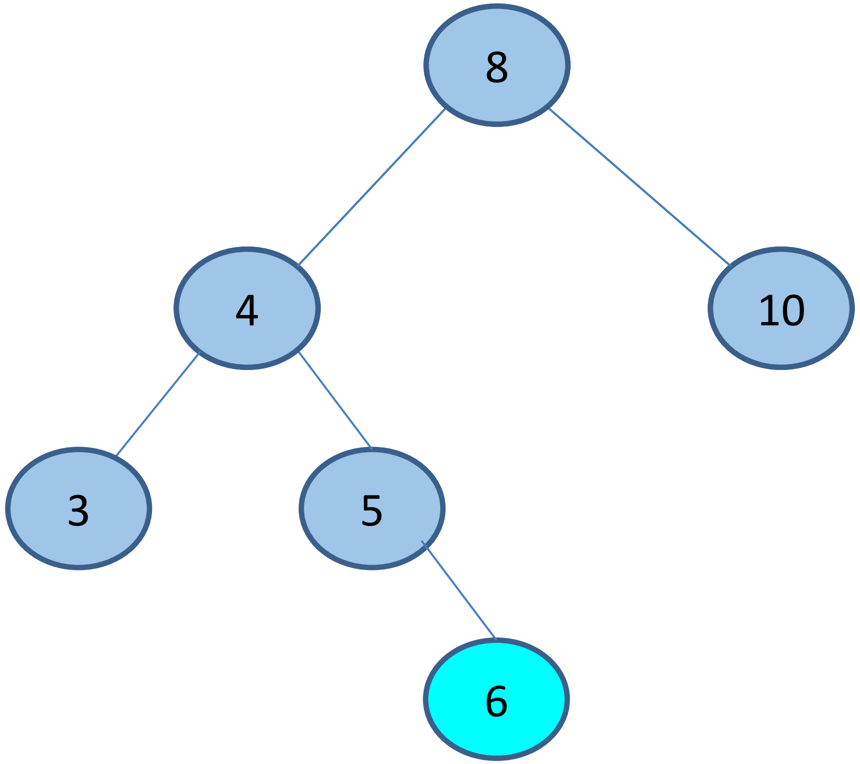
3 5 **6 9**



Binary Search Trees - Insert

Insert the following to the BST:

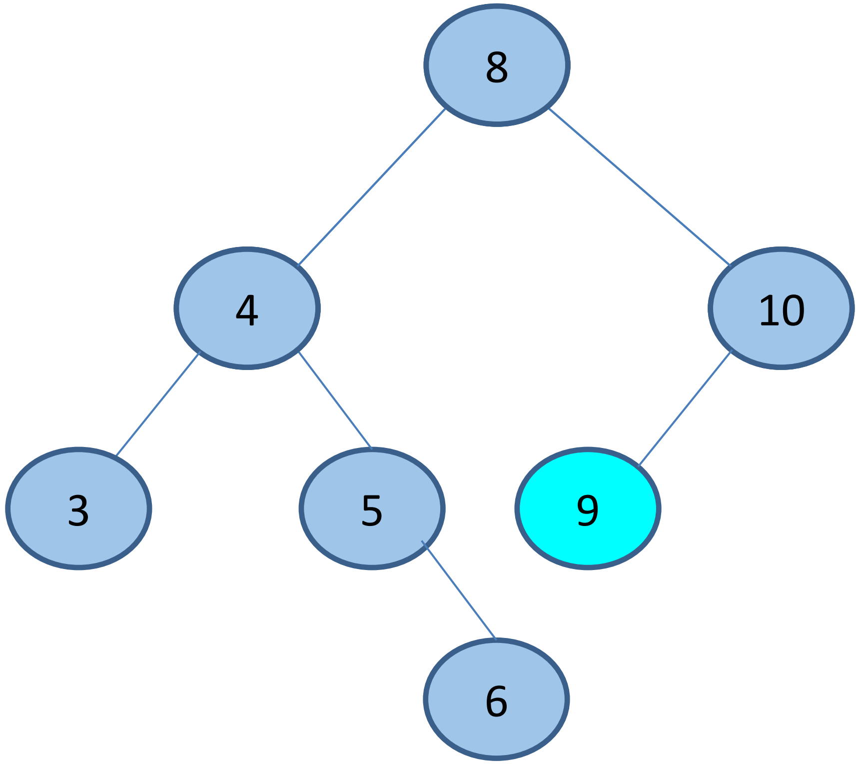
3 5 **6** **9**



Binary Search Trees - Insert

Insert the following to the BST:

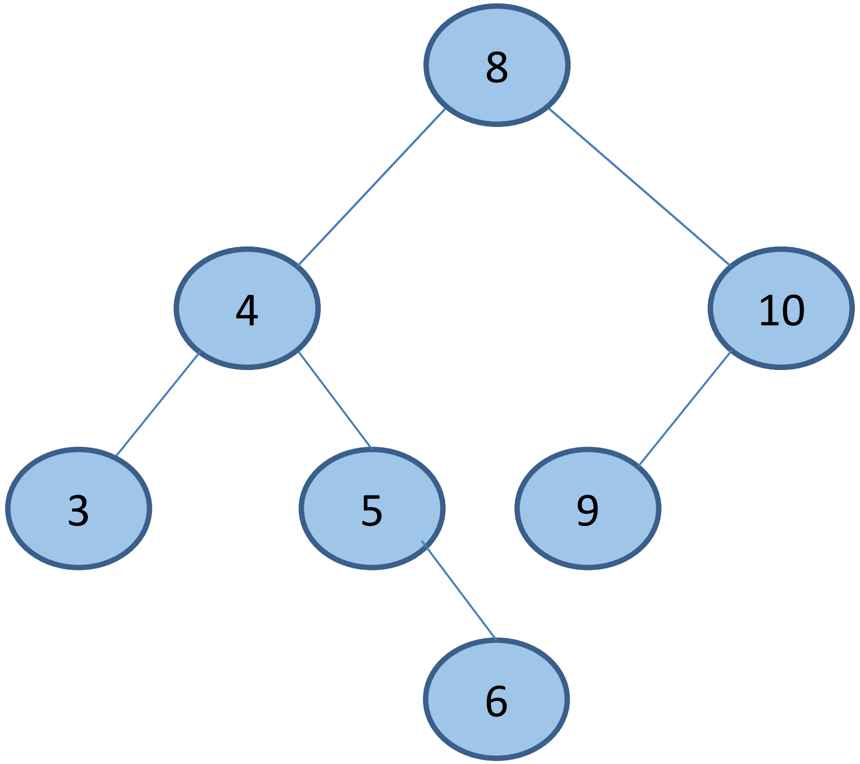
3 5 6 9



Binary Search Trees - Insert

Insert the following to the BST:

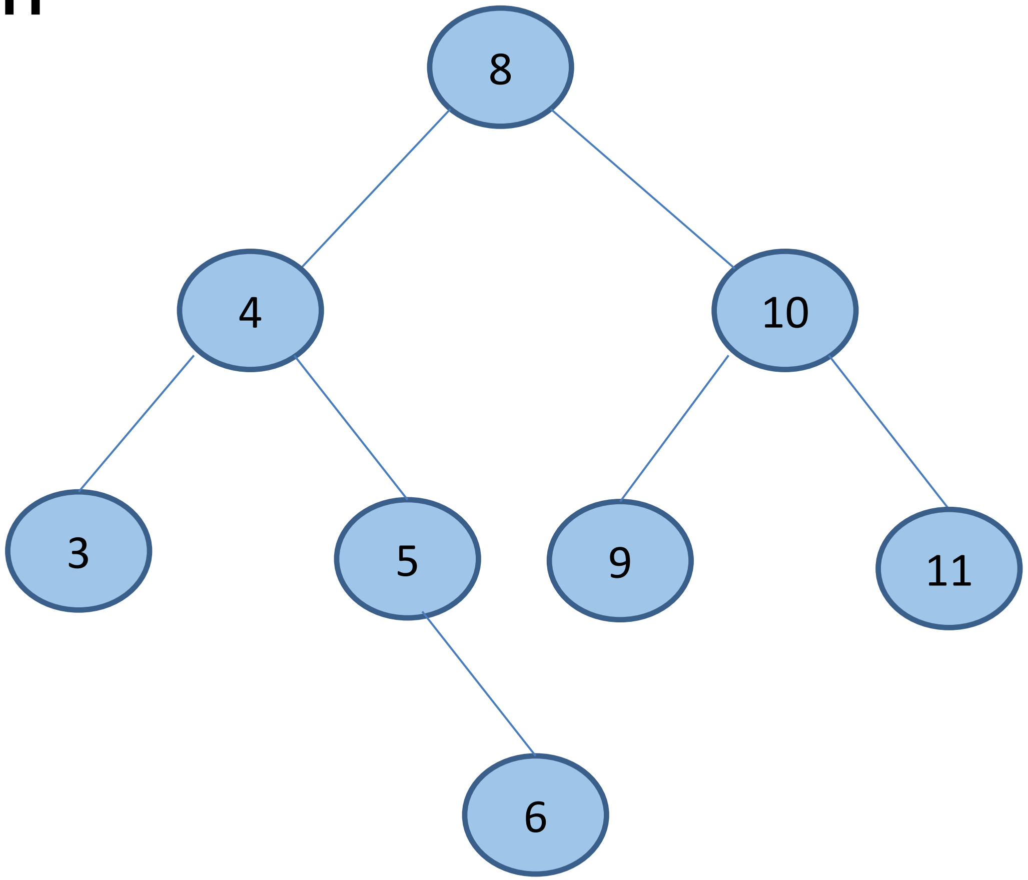
3 5 6 9



Binary Search Trees - Delete

Delete the following to the BST:

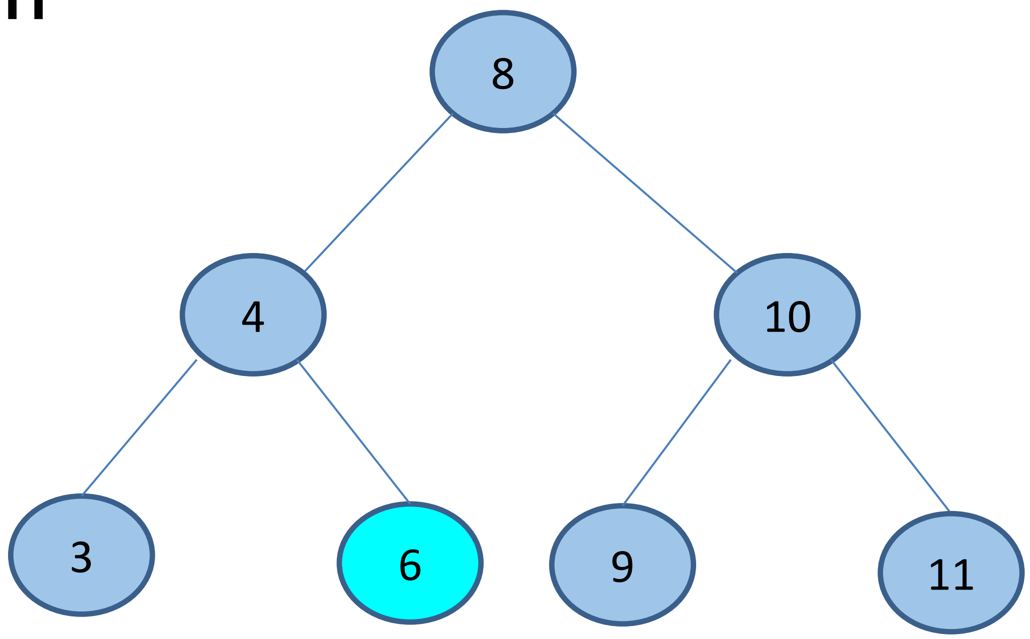
5 4 8 11



Binary Search Trees - Delete

Delete the following to the BST:

5 4 8 11

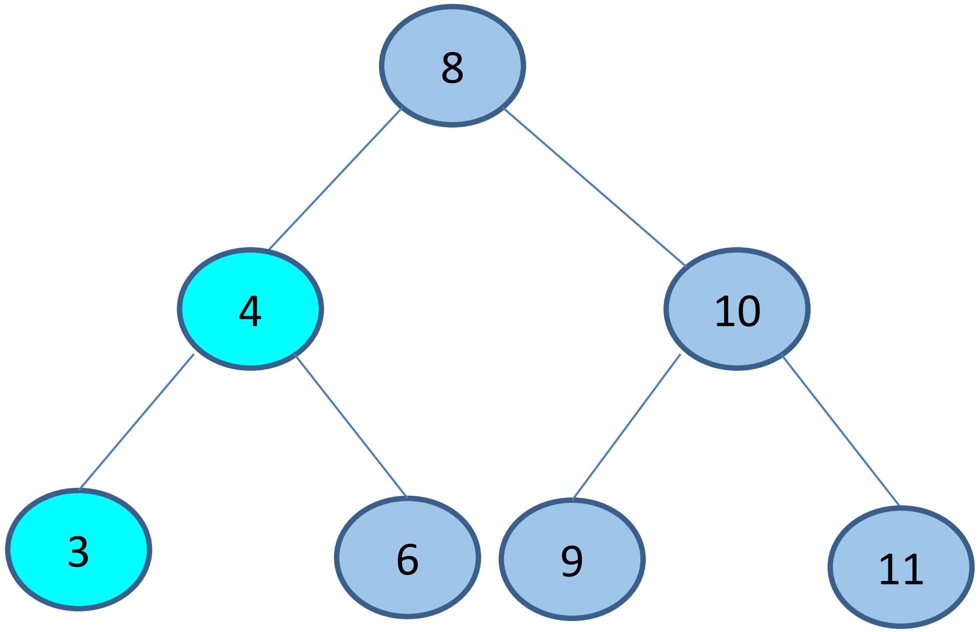


Just replace with 6!

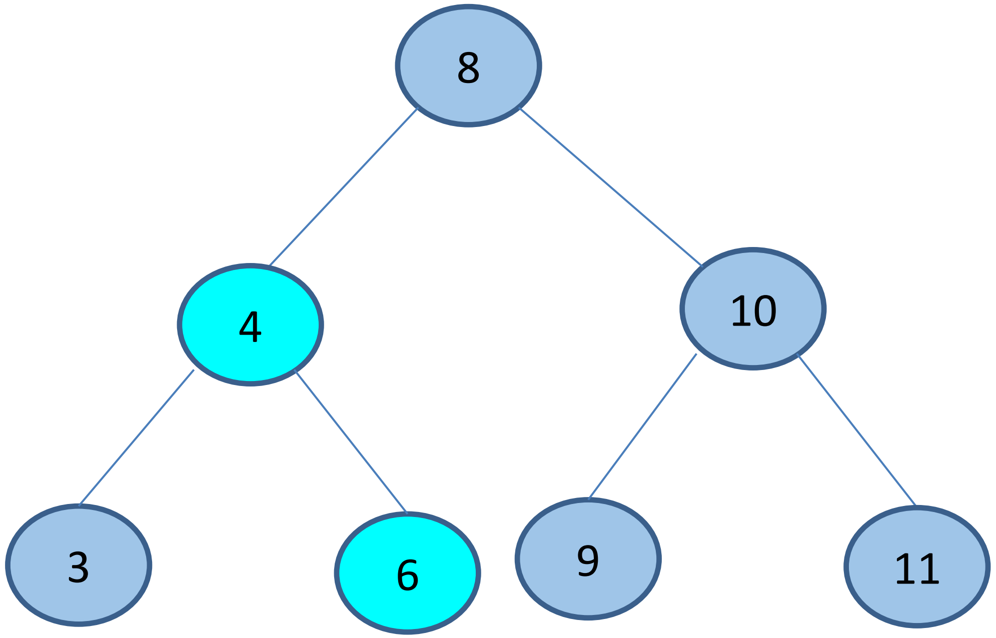
Binary Search Trees - Delete

Delete the following to the BST:
5 4 **8 11**

Replace with in order successor / in order predecessor



inorder predecessor

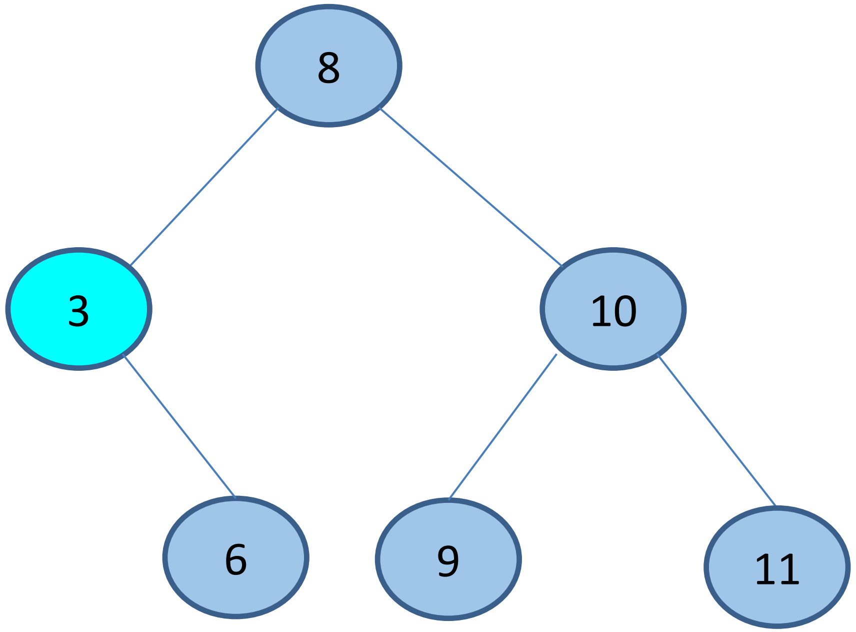


inorder successor

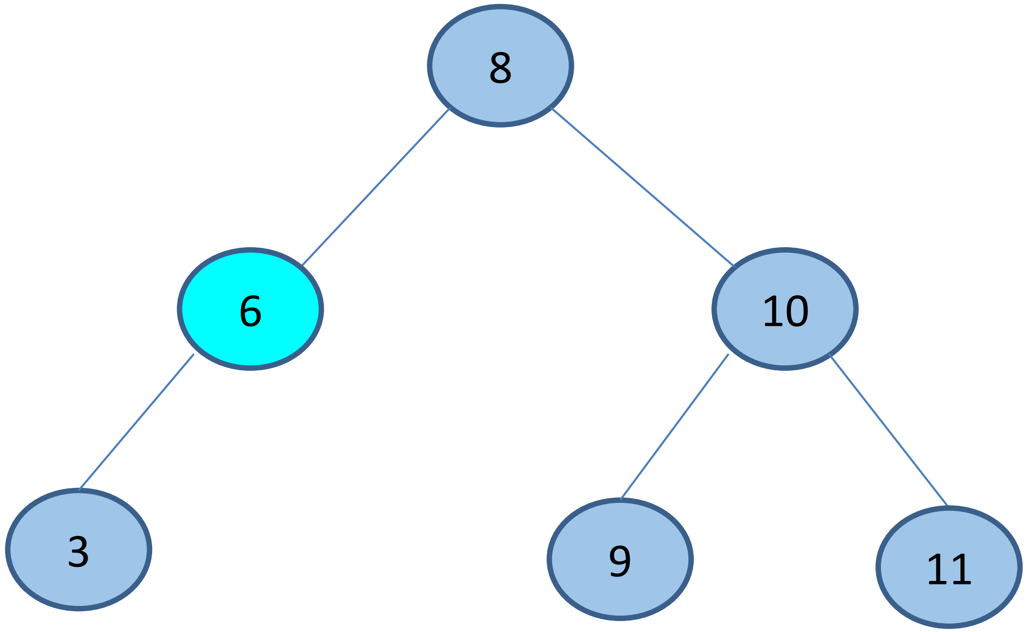
Binary Search Trees - Delete

Delete the following to the BST:
5 4 **8 11**

Replace with in order successor / in order predecessor



inorder predecessor

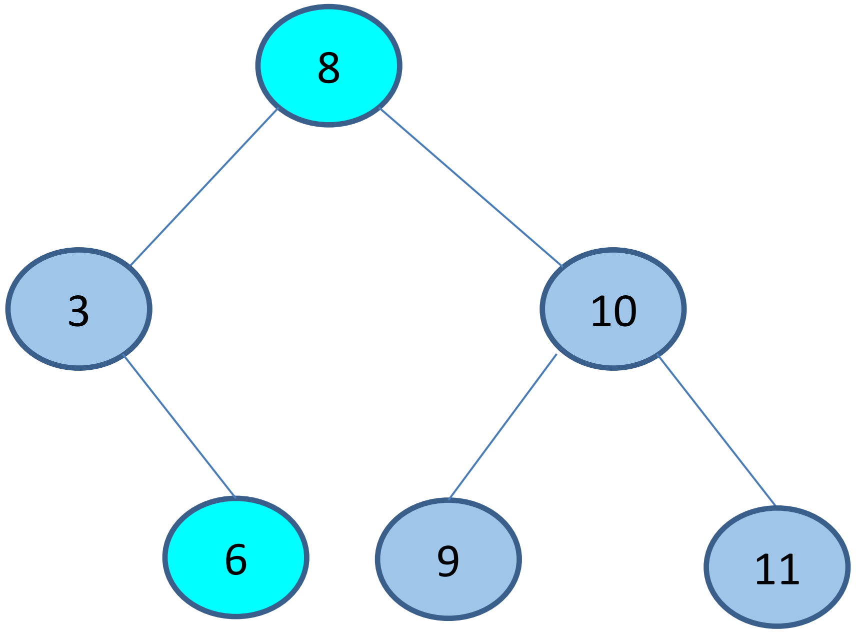


inorder successor

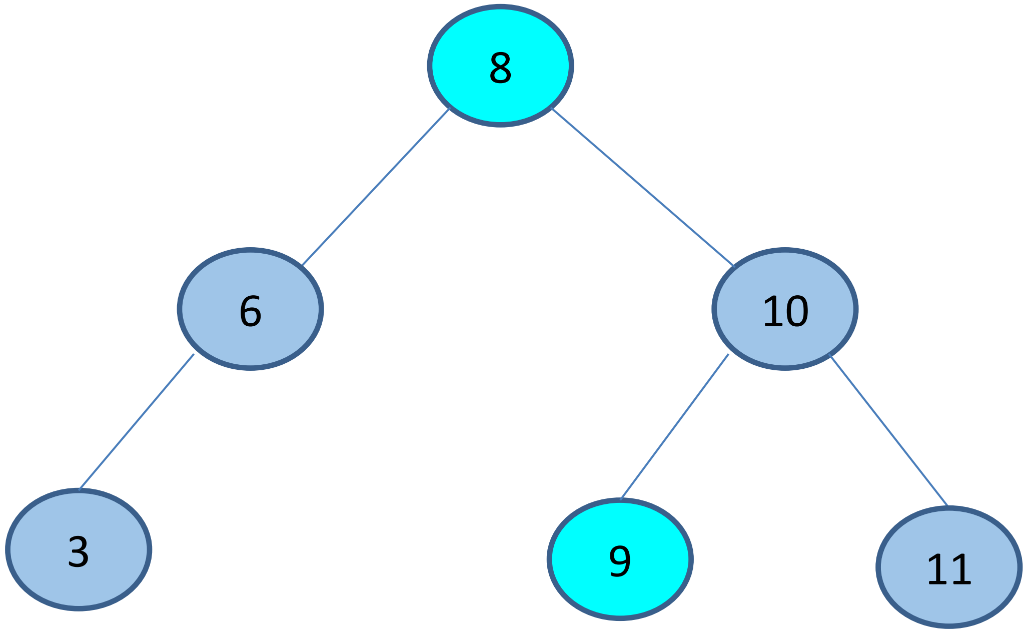
Binary Search Trees - Delete

Delete the following to the BST:
5 4 **8** **11**

Replace with in order successor / in order predecessor



inorder predecessor

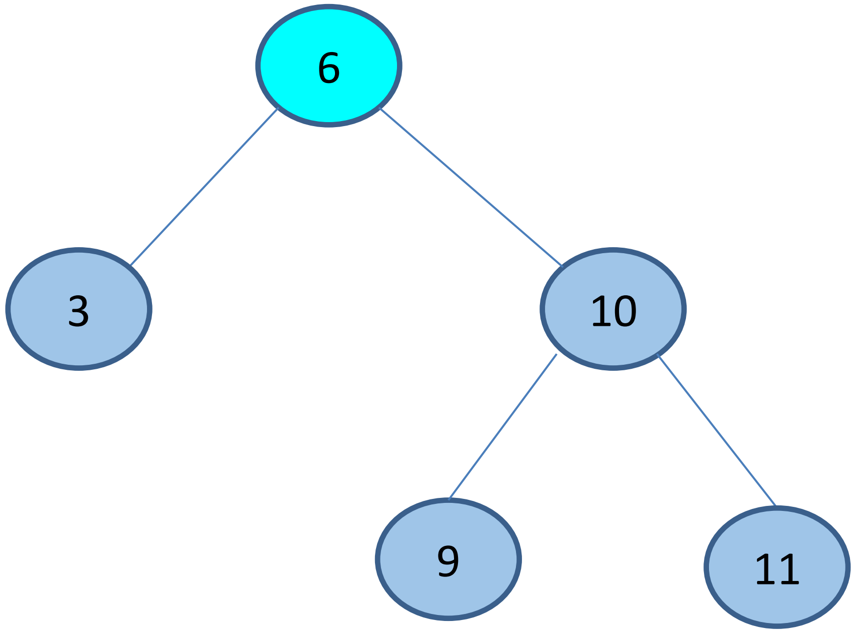


inorder successor

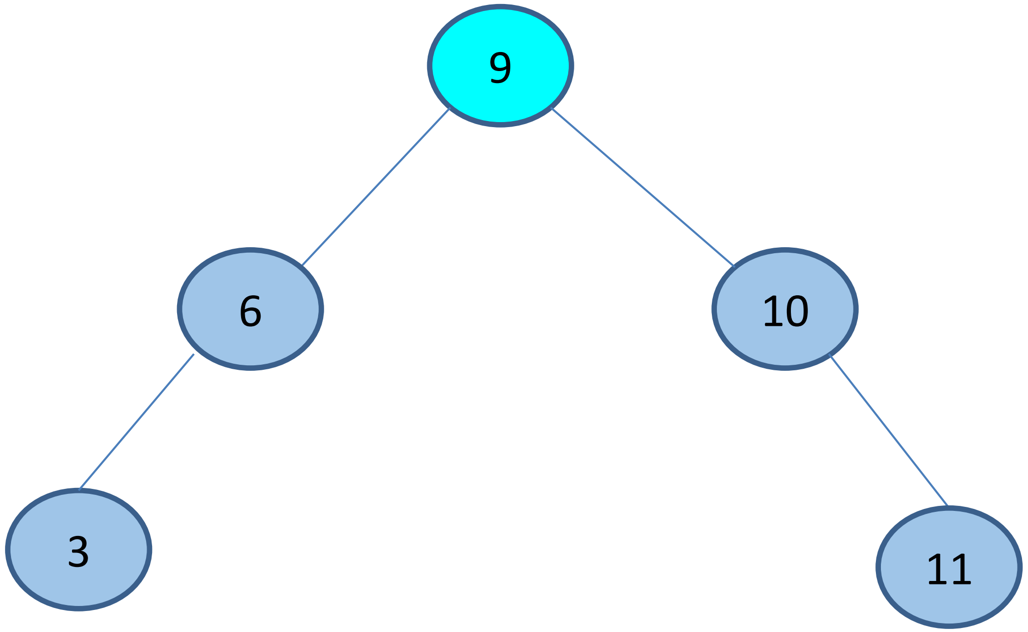
Binary Search Trees - Delete

Delete the following to the BST:
5 4 **8** **11**

Replace with in order successor / in order predecessor



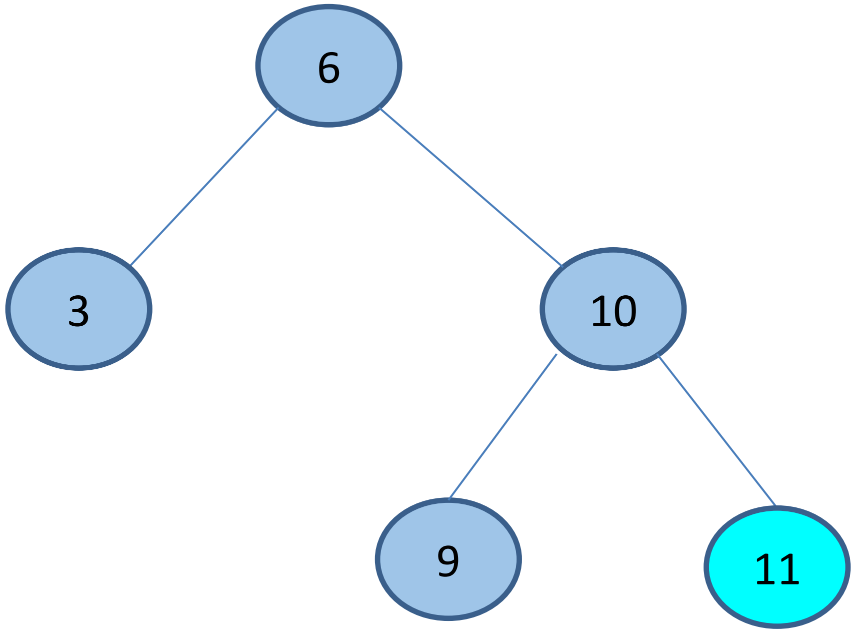
inorder predecessor



inorder successor

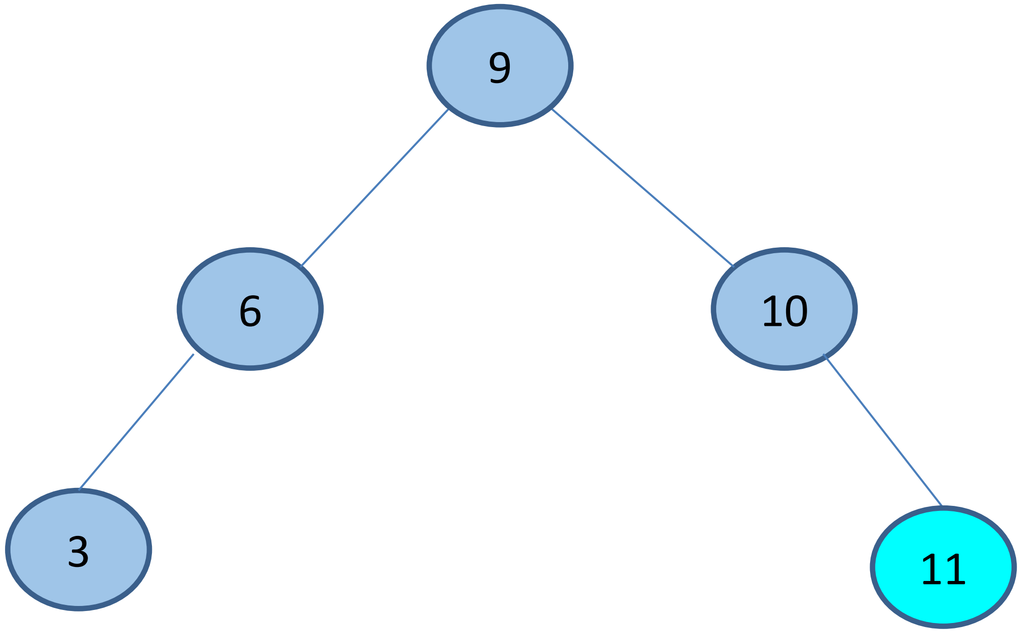
Binary Search Trees: Delete

Delete the following to the BST:
5 4 8 **11**



inorder predecessor

Replace with in order successor / in order predecessor

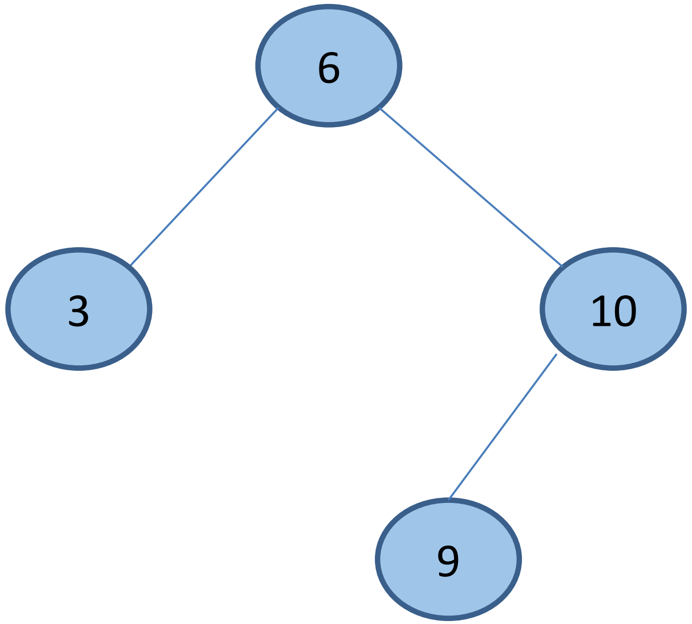


inorder successor

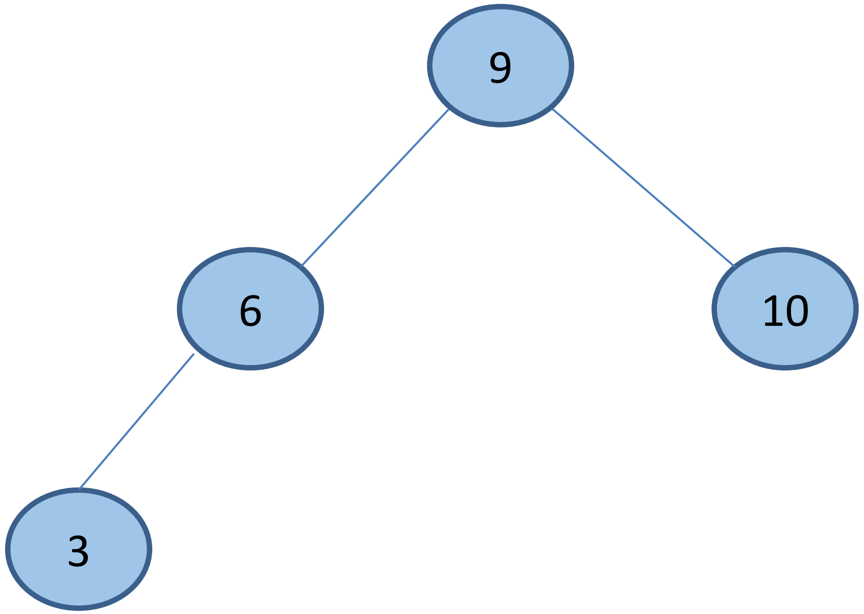
Binary Search Trees: Delete

Delete the following to the BST:
5 4 8 11

Replace with in order successor / in order predecessor



inorder predecessor



inorder successor

Binary Search Trees

What does it mean for a tree to be balanced or unbalanced?

What does this mean for the insert and search complexities?

Binary Search Trees

What does it mean for a tree to be balanced or unbalanced?

For every node k of T, the heights of the children of k differ by at most 1

What does this mean for the insert and search complexities (for balanced)?

Worst case becomes O(log n)

AVL Trees

AVL Trees

- Self-balancing BST
- Maintain balance with each insertion and deletion
- Have average and worst case search/insert/delete complexities of O(log n)

AVL Trees

- Self-balancing BST
- Maintain balance with each insertion and deletion
- Have average and worst case search/insert/delete complexities of O(log n)
- Invariants
    - The value of a node is > than the values of all its nodes in its left subtree and <= the values of all of the nodes in its right subtree (i.e. it is a BST!)

AVL Trees

- Self-balancing BST
- Maintain balance with each insertion and deletion
- Have average and worst case search/insert/delete complexities of O(log n)
- Invariants
  - The value of a node is > than the values of all its nodes in its left subtree and <= the values of all of the nodes in its right subtree (i.e. it is a BST!)
  - The balance factor of each node must be in the range [-1, 1]
    - Balance factor(node) = Height(left subtree) – Height(right subtree)

AVL Trees - Insertion & Deletion

Insertion - O(logn)

1. Insert the node in its appropriate location without considering imbalances (same as BST!)
2. Determine whether there is an imbalance in any node starting from the inserted node and moving up to the root and rotate if necessary. Once you've rotated "once" (might be a double rotation), you're done!

AVL Trees - Insertion & Deletion

Insertion - O(logn)

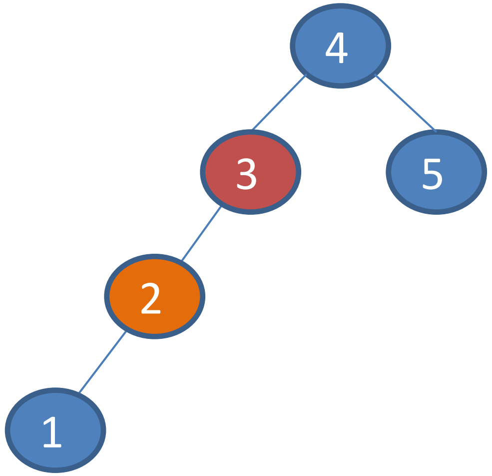
1. Insert the node in its appropriate location without considering imbalances (same as BST!)
2. Determine whether there is an imbalance in any node starting from the inserted node and moving up to the root and rotate if necessary. Once you've rotated "once" (might be a double rotation), you're done!

Deletion - O(logn)

1. Delete like a BST
2. Rearrange tree to balance height
    - Start at parent of deleted node and work up
    - At the first unbalanced node encountered, rotate as needed

AVL Rotation: Case 1 (+,+)

Left subtree causes imbalance and **left** side of that subtree has extra node

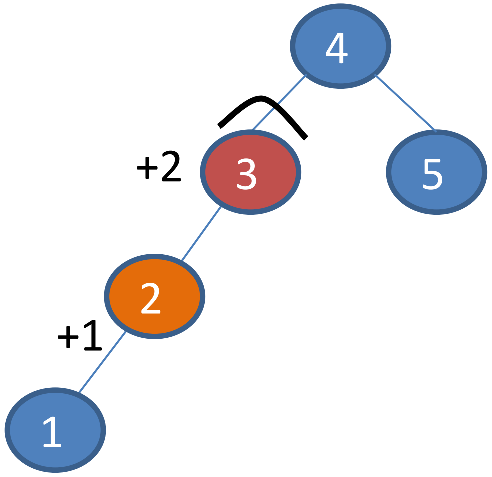


Insertion Order:

4, 3, 5, 2, 1

AVL Rotation: Case 1 (+,+)

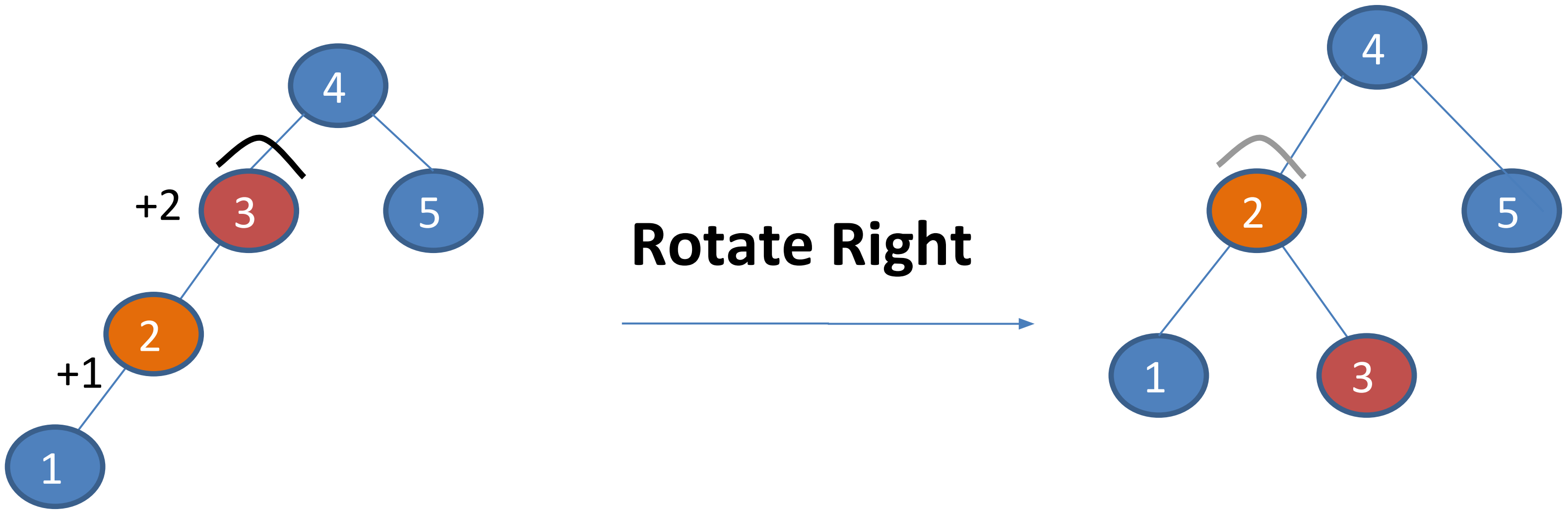
Left subtree causes imbalance and **left** side of that subtree has extra node



Rotate Right

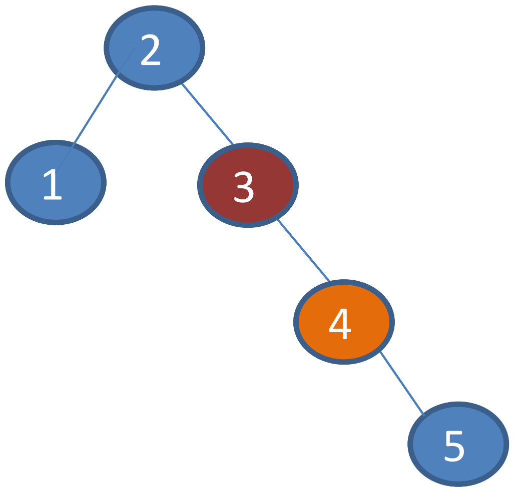
AVL Rotation: Case 1 (+,+)

Left subtree causes imbalance and **left** side of that subtree has extra node



AVL Rotation: Case 2 (-,-)

Right subtree causes imbalance, and **right** side of that subtree has extra node



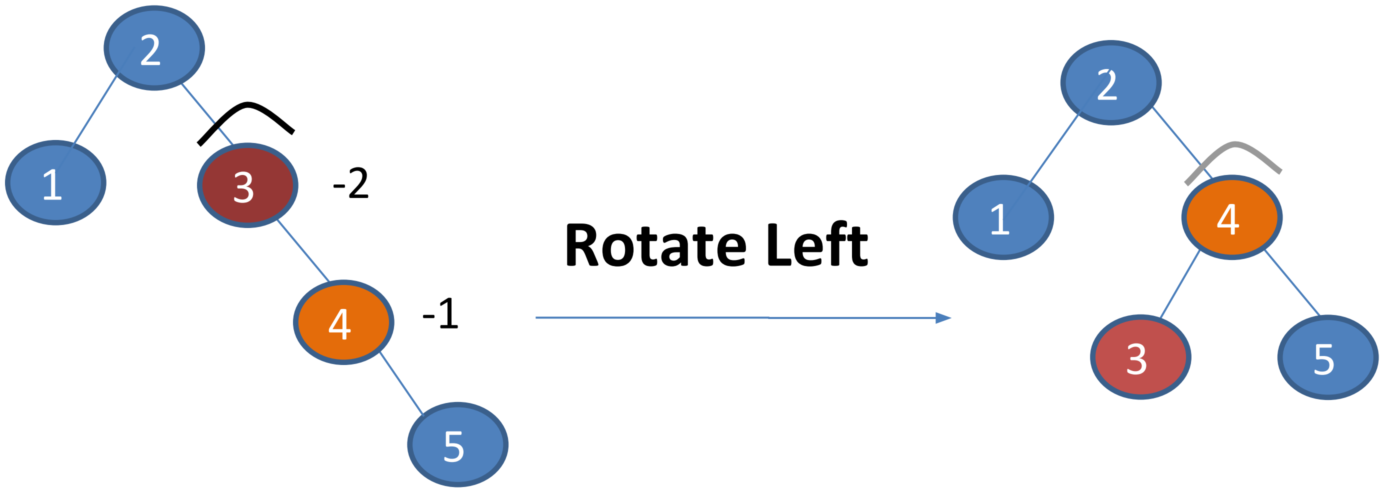
Insertion Order:

2, 1, 3, 4, 5

Rotate Left

AVL Rotation: Case 2 (-,-)

Right subtree causes imbalance, and **right** side of that subtree has extra node

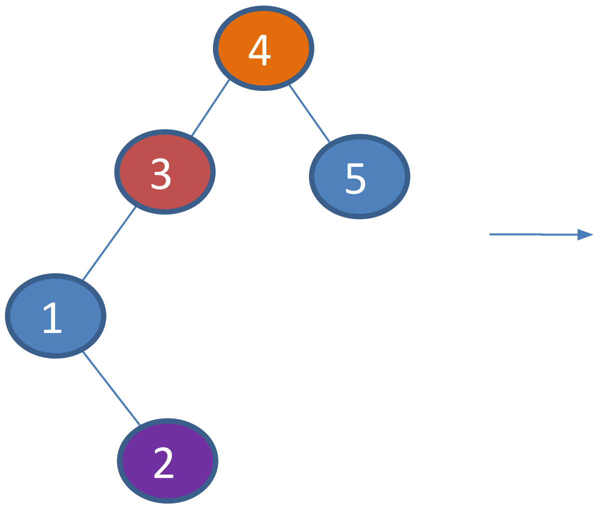


AVL Rotation: Case 3 (+,-)

Left subtree causes imbalance, and **right** side of that subtree has extra node

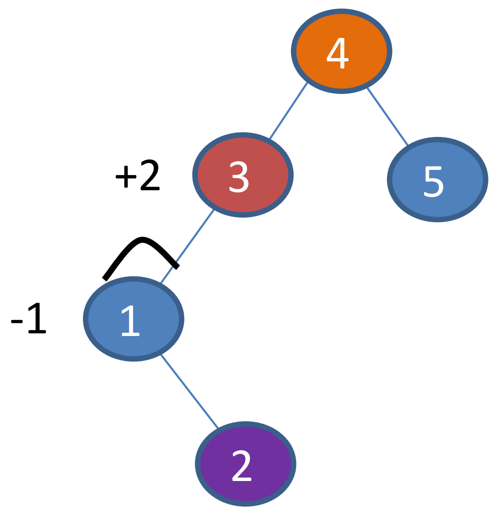
Insertion Order:

4, 5, 3, 1, 2



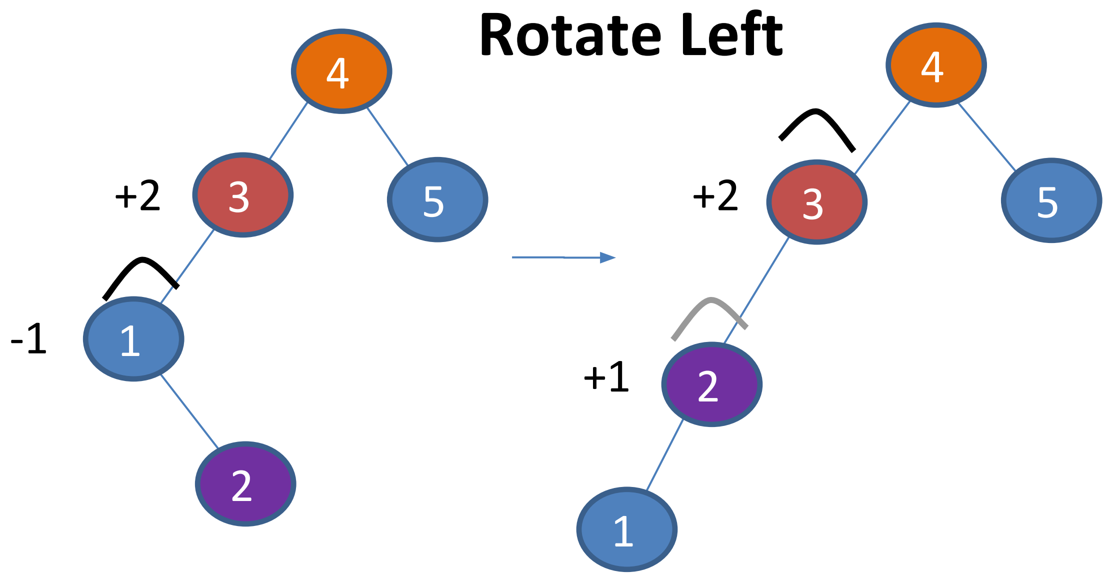
AVL Rotation: Case 3 (+,-)

Left subtree causes imbalance, and **right** side of that subtree has extra node



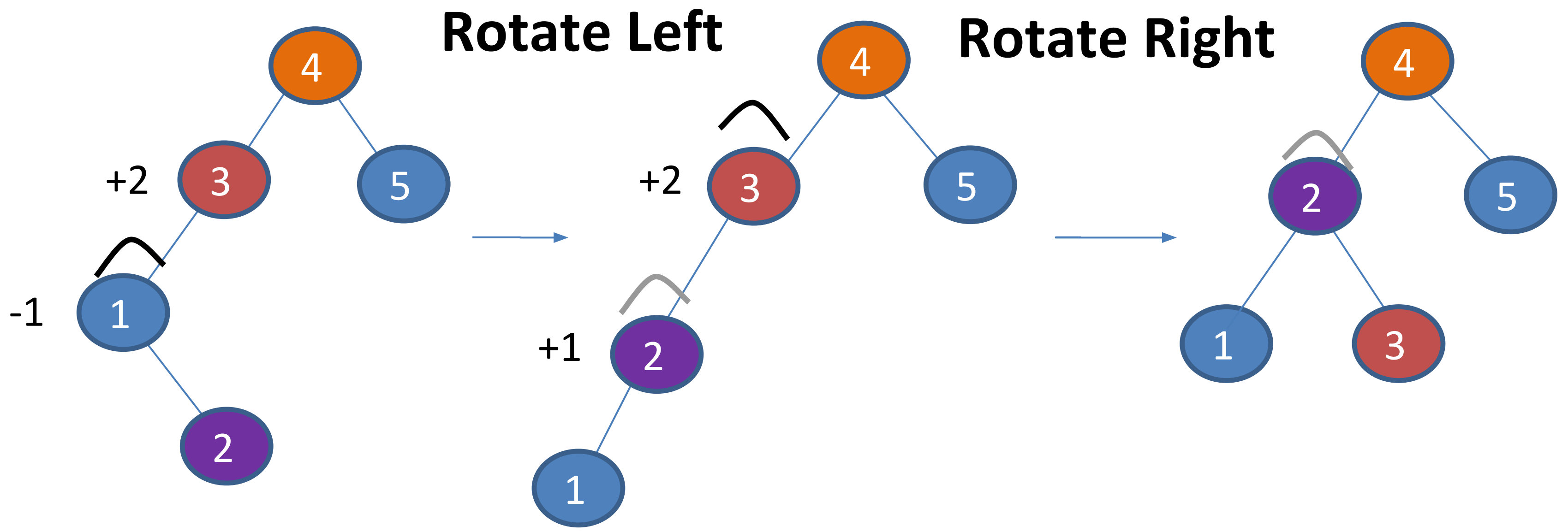
AVL Rotation: Case 3 (+,-)

Left subtree causes imbalance, and **right** side of that subtree has extra node



AVL Rotation: Case 3 (+,-)

Left subtree causes imbalance, and **right** side of that subtree has extra node

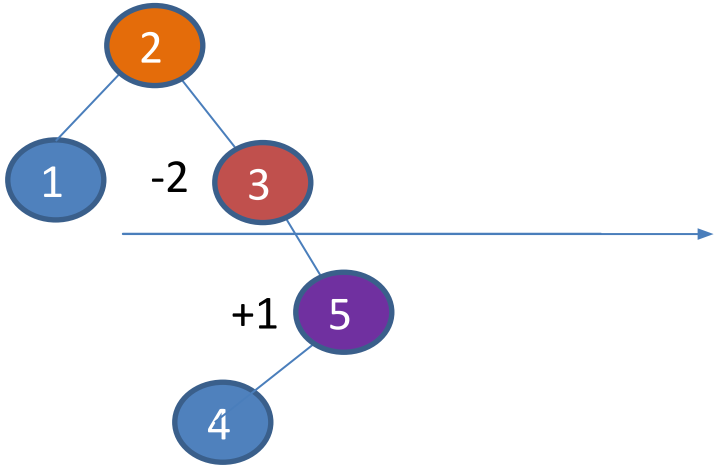


AVL Rotation: Case 4 (-,+)

Right subtree causes imbalance, and **left** side of that subtree has extra node

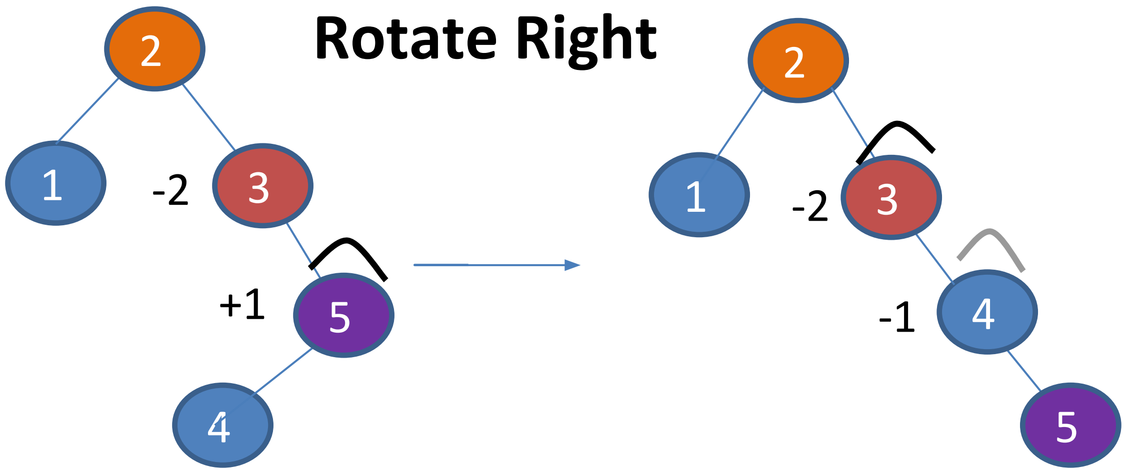
Insertion Order:

2, 1, 3, 5, 4



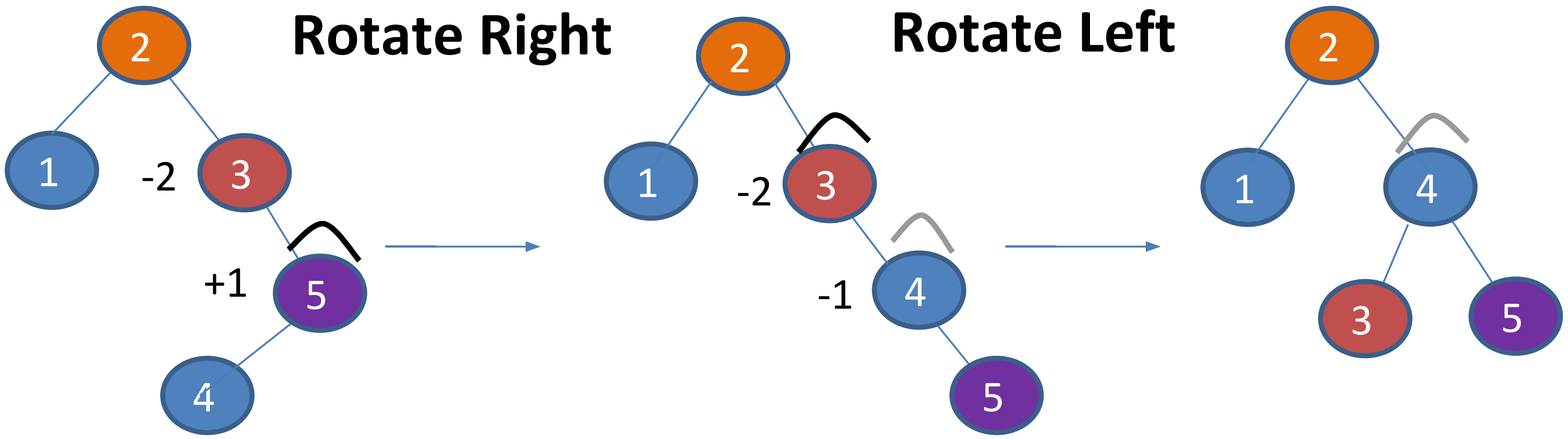
AVL Rotation: Case 4 (-,+)

Right subtree causes imbalance, and **left** side of that subtree has extra node



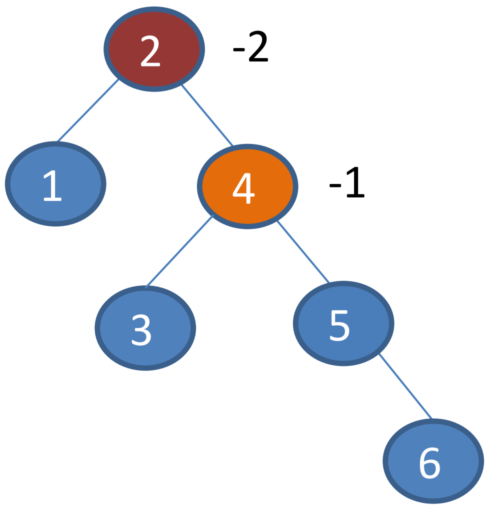
AVL Rotation: Case 4 (-,+)

Right subtree causes imbalance, and **left** side of that subtree has extra node



AVL Rotation: Special Case

Node that moves up has children!



Insertion Order:

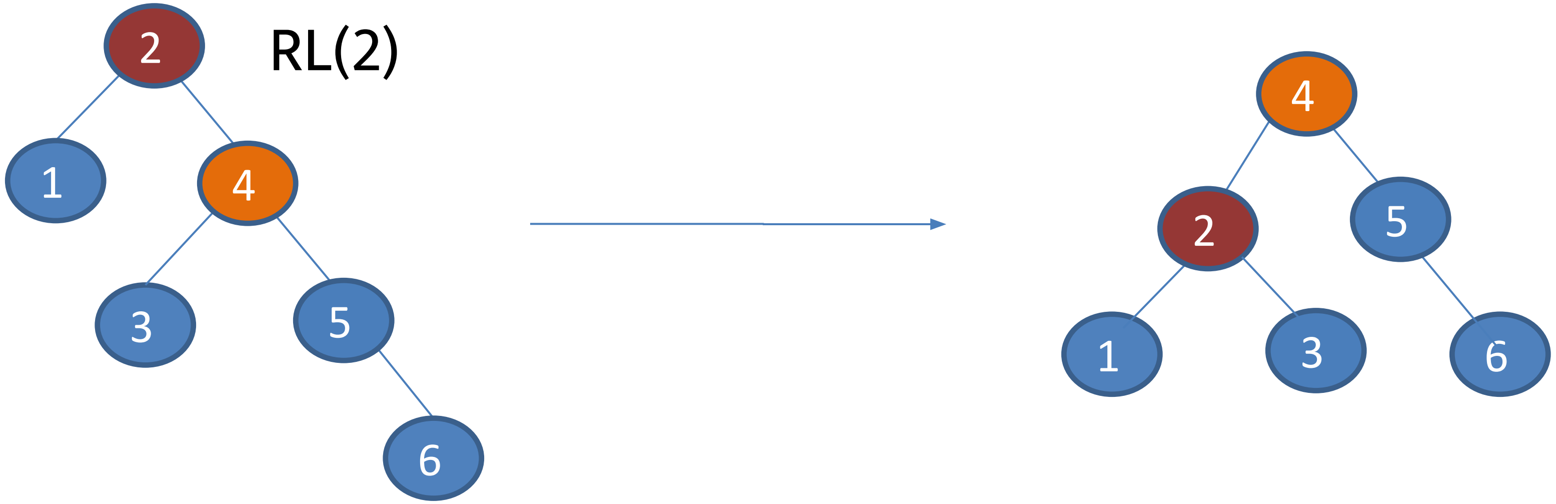
2, 1, 4, 3, 5, 6

AVL Rotation: Special Case

Node that moves up has 2 children! → The node that moves down gets the other child

If rotating left: node gets left child on its right side

If rotating right: node gets the right child on its left side

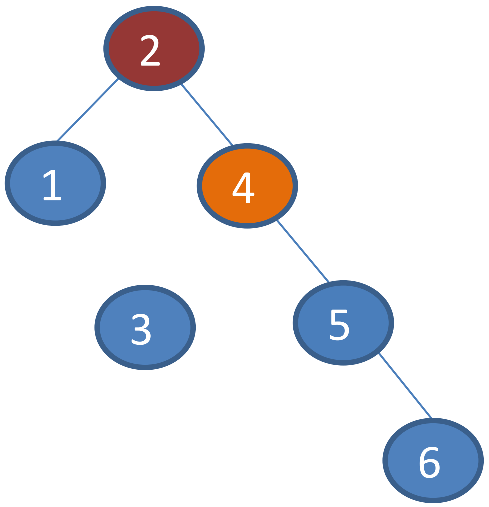


AVL Rotation: Special Case

Node that moves up has 2 children! $\longrightarrow$ The node that moves down gets the other child

If rotating left: node gets left child on its right side

If rotating right: node gets the right child on its left side



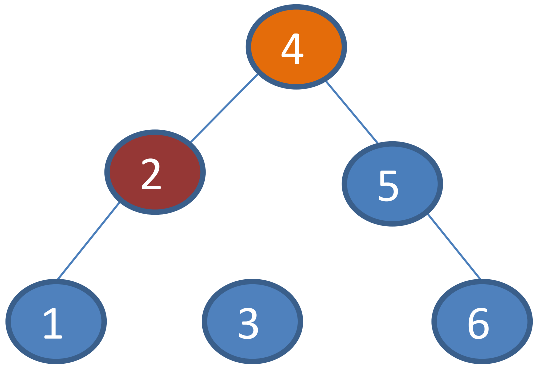
Disconnect left subtree so that parent can slide down

AVL Rotation: Special Case

Node that moves up has 2 children! $\longrightarrow$ The node that moves down gets the other child

If rotating left: node gets left child on its right side

If rotating right: node gets the right child on its left side



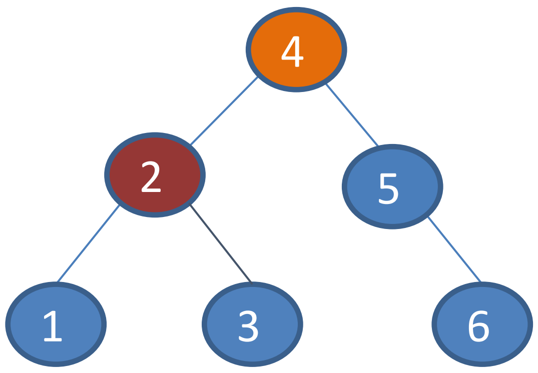
Slide 2 down to become left child of 4

AVL Rotation: Special Case

Node that moves up has 2 children! $\rightarrow$ The node that moves down gets the other child

If rotating left: node gets left child on its right side

If rotating right: node gets the right child on its left side



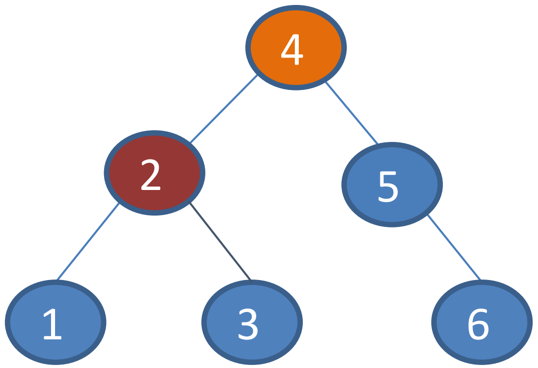
Make 4's previous left child (3), the right child of it's new left child (2)

AVL Rotation: Special Case

Node that moves up has 2 children! → The node that moves down gets the other child

If rotating left: node gets left child on its right side

If rotating right: node gets the right child on its left side



Make 4's previous left child (3), the right child of it's new left child (2)

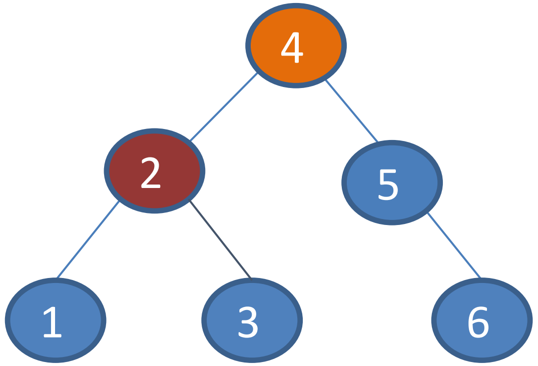
How do we know there is room for 3 there?
What about 2's right child?

AVL Rotation: Special Case

Node that moves up has 2 children! → The node that moves down gets the other child

If rotating left: node gets left child on its right side

If rotating right: node gets the right child on its left side



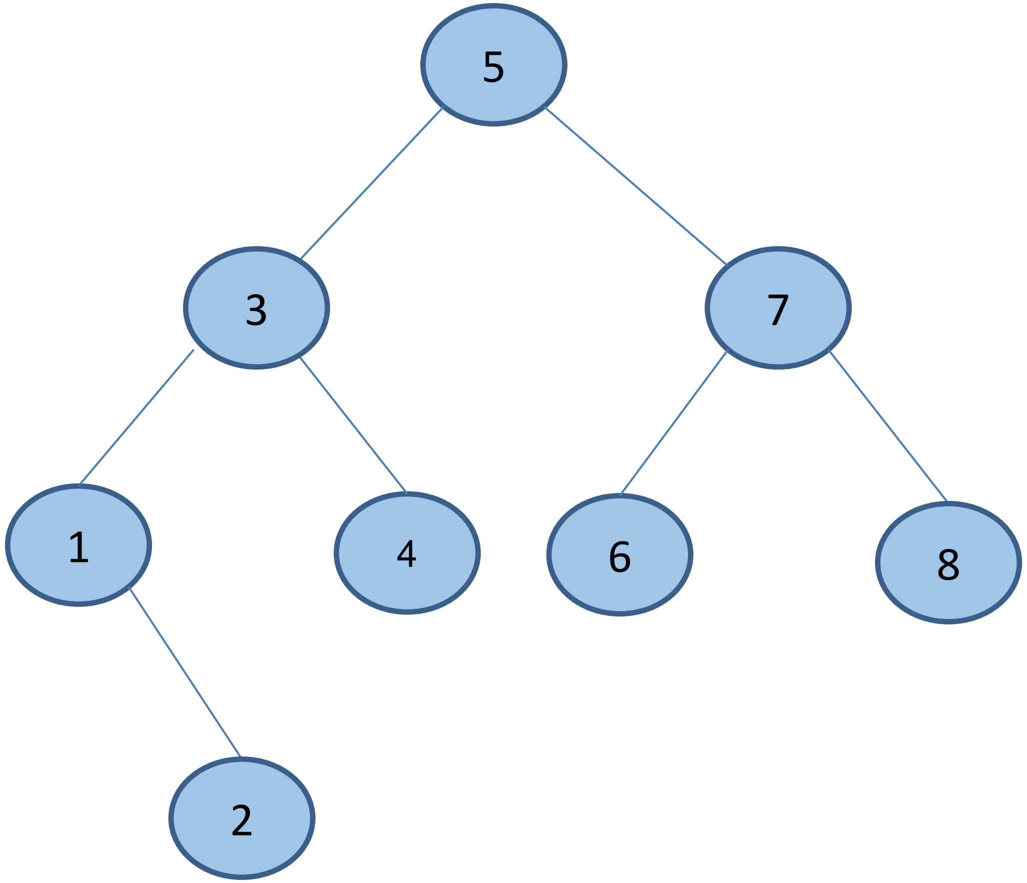
Make 4's previous left child (3), the right child of it's new left child (2)

How do we know there is room for 3 there?
What about 2's right child?

2's right child was 4, which we just rotated up to become its parent! The invariants of the BST hold because the lower node's left child will be greater than the node whose place it is taking

Practice Problem

1. AVL Trees: Practice Problem

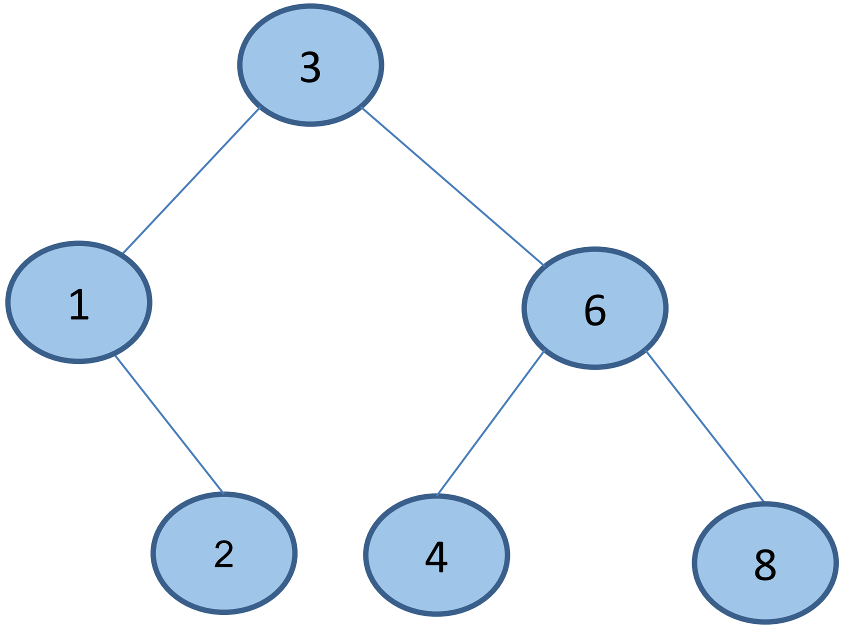


Delete node 5 and then delete node 7. What does the resulting tree look like?

Assume we use in-order successor

1. AVL Trees: Practice Problem

Answer:



2. Tree Traversal

What does the following function return?

```
int count(Node* curNode) {
 if (curNode == nullptr)
    return 0;
 else if (curNode->parent == nullptr)
    return 1 + count(curNode->left) + count(curNode->right);
 else if (curNode->left == nullptr && curNode->right == nullptr)
    return 1;
 else
   return count(curNode->left) + count(curNode->right);
 } // count()
```

A) The number of nodes in the tree
B) The number of internal nodes
C) The number of external (leaf) nodes
D) The number of external (leaf) nodes plus the root
E) The depth of the tree

2. Tree Traversal

What does the following function return?

```
int count(Node* curNode) {
 if (curNode == nullptr)
    return 0;
 else if (curNode->parent == nullptr)
    return 1 + count(curNode->left) + count(curNode->right);
 else if (curNode->left == nullptr && curNode->right == nullptr)
    return 1;
 else
   return count(curNode->left) + count(curNode->right);
 } // count()
```

A) The number of nodes in the tree
B) The number of internal nodes
C) The number of external (leaf) nodes
D) The number of external (leaf) nodes plus the root
E) The depth of the tree

3. Binary Search Tree Complexity

Suppose you have a binary search tree, where h represents the tree's height and n represents the number of nodes. What is the worst-case complexity of a search operation on this tree in terms of *h* and/or *n*?

A) Θ(n log h)
B) Θ(n^2)
C) Θ(log n)
D) Θ(h)
E) Θ(n log_n h)

3. Binary Search Tree Complexity

Suppose you have a binary search tree, where h represents the tree's height and n represents the number of nodes. What is the worst-case complexity of a search operation on this tree in terms of *h* and/or *n*?

A) Θ(n log h)
B) Θ(n^2)
C) Θ(log n)
D) Θ(h)
E) Θ(n log_n h)

4. Binary Search Tree vs. Binary Tree

Given **pointer-**based representations of both, what can be said about a binary search tree that cannot be said about a binary tree?

A) The inorder traversal of a tree is always a sorted list
B) The worst-case space complexity of a tree is O(2^n).
C) A tree balances itself when a single branch becomes uneven.
D) The worst-case time complexity of inserting an element is O(n).
E) It is always a complete binary tree

4. Binary Search Tree vs. Binary Tree

Given **pointer-**based representations of both, what can be said about a binary search tree that cannot be said about a binary tree?

A) The inorder traversal of a tree is always a sorted list
B) The worst-case space complexity of a tree is O(2^n).
C) A tree balances itself when a single branch becomes uneven.
D) The worst-case time complexity of inserting an element is O(n).
E) It is always a complete binary tree

Handwritten Problem

Handwritten Problem: Background

Let's say the *diameter* of a tree is the maximum number of edges on any path connecting two nodes of the tree. For example, here are three sample trees and their diameters. In each case the longest path is bolded and shown in purple. Note that there can be more than one longest path.

```
        A
      /   \
    B        C
   / \      / \
  D   E    F   G
 /   /        / \
H   I        J   K
     \          /
      L        M
```

Diameter: 8

```
    A
   / \
  B   C
 /   / \
D   E   F
   / \   \
  G   H   I
     /     \
    J       K
```

Diameter: 6

```
A
   \
       C
      /
     E
    / \
   G   H
      /
     J
```

Diameter: 4

Handwritten Problem

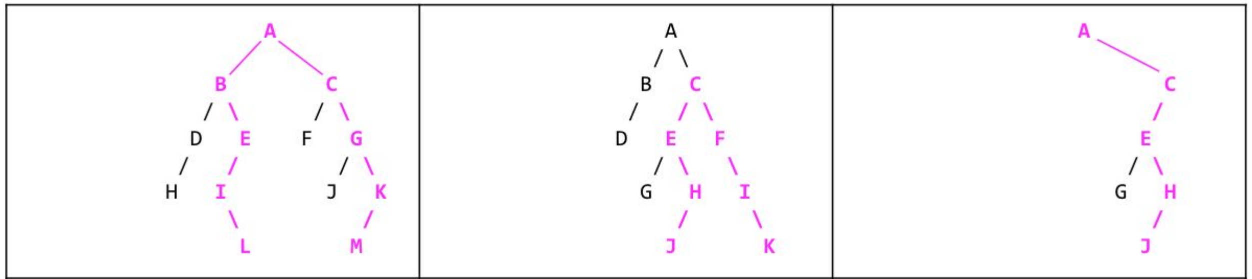
Consider the following Node definition of a binary tree:

```
class BinaryTreeNode {
public:
    BinaryTreeNode* left;
    BinaryTreeNode* right;
    int value;
    BinaryTreeNode(int n)
     : value(n), left(nullptr),
       right(nullptr) {}
};
```

Your task: Implement the function diameter that computes the diameter of a *binary* tree represented by a pointer to an object of BinaryTreeNode class. Assume that nullptr represents an empty tree or a missing child. Do not modify the definition of BinaryTreeNode class, but you may write helper functions.

Implement diameter in $O(n^2)$ or better time (it can be done in $O(n)$).

```
int diameter(const BinaryTreeNode* tree) {

}
```



Diameter: 8 | Diameter: 6 | Diameter: 4

Lab-related questions

Preeti's Lab 8 OH on Tuesday, 03/24/2020 from 3:30-5:30pm EDT.

https://us04web.zoom.us/j/4189761788

Handwritten Problem Review

```cpp
int heightOf(const BinaryTreeNode* tree) {
    // number of nodes on longest path leaf-to-root (edges is 1 less than this)
    if (tree == nullptr) {
        return 0;
    } else {
        return max(heightOf(tree->left), heightOf(tree->right)) + 1;
    }
}

int diameter(const BinaryTreeNode* tree) {
    if (tree == nullptr) {
        return 0;
    } else {
        // the diameter exists in left/right subtree:
        int childrenDiameters = max(diameter(tree->left), diameter(tree->right));
        // the diameter is a path through this node (each node has one edge up):
        int nodeDiameter = heightOf(tree->left) + heightOf(tree->right);
        return max(childrenDiameters, nodeDiameter);
    }
}
```

$O(n^2)$

Minor Optimizations

- You only need to check the diameter of the subtree with greater height
  - If the longest path doesn't go through the node, it can't occur in the shorter of the two subtrees

$O(n^2)$ instead of $O(n^2)$... does not help worst-case

```
int diameter(const BinaryTreeNode* tree)  {
    if (tree == nullptr)  {
        return 0;
    }
    int left_height = heightOf(tree->left);
    int right_height = heightOf(tree->right);
    int diam_taller = left_height >= right_height ? diameter(tree->left) : diameter(tree->right);
    return max(left_height + right_height, diam_taller);
}
```

```
struct Desc { int height; int diam; };
Desc helper(const BinaryTreeNode* tree) {
    if (tree == nullptr) {
        return Desc{/*height*/ 0, /*diam*/ 0};
    }
    Desc left  = helper(tree->left);
    Desc right = helper(tree->right);
    int diam_children = max(left.diam, right.diam); // diameter in left/right subtree
    int diam_self = left.height + right.height; // longest path through current node
    int diam_whole = max(diam_children, diam_self);
    return Desc {
        /*height*/ 1 + max(left.height, right.height),
        /*diam*/   diam_whole
    };
}
```

<-- **We can return a struct if we need to return multiple values from our helper!**

```
int diameter(const BinaryTreeNode* tree) {
    return helper(tree).diam;
}
```

O(n)