08

March 26-April 1, 2023

Binary Trees, AVL Trees, and Tree Traversals

Announcements

- Project 3 is due on **Tuesday, April 2nd** at **11:59pm.**
- Project 4 will be released on **Thursday, April 4th!**
- Lab 7 AG + quiz due **Monday, April 1st** at **11:59pm.**
- Lab 8 handwritten due in lab by **Monday, April 1st.**
- Lab 8 AG + quiz due **Monday, April 8th** at **11:59pm.**

Lab 7 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 prefix 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);
```

Handwritten Solution

```
vector<string> replace_words(const vector<string>& prefixes,
                             const vector<string>& sentence) {
    unordered_set<string> set(prefixes.begin(), prefixes.end()); // O(MP)
    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;
}
```

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!

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 == nullptr)
          return 0;
   else if (root->left == nullptr)
          return minimum_depth(root->right) + 1;
   else if (root->right == nullptr)
          return minimum_depth(root->left) + 1;
   else
          return min(minimum_depth(root->left),
                minimum_depth(root->right)) + 1;
}
```

Tree Traversals

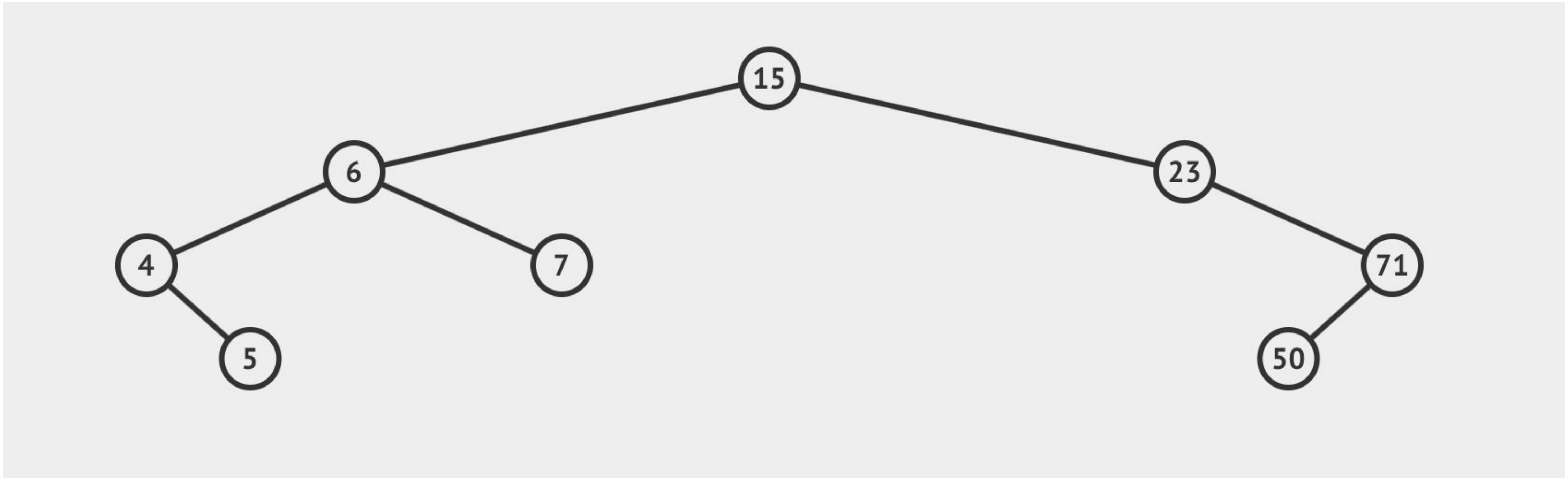
Parent = P, Left Child = L, Right Child = R

- 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

Preorder Traversal

```
void traversal(Node* root) {
  if(root == nullptr) return;

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

Preorder Traversal



What is the pre-order traversal of this tree?

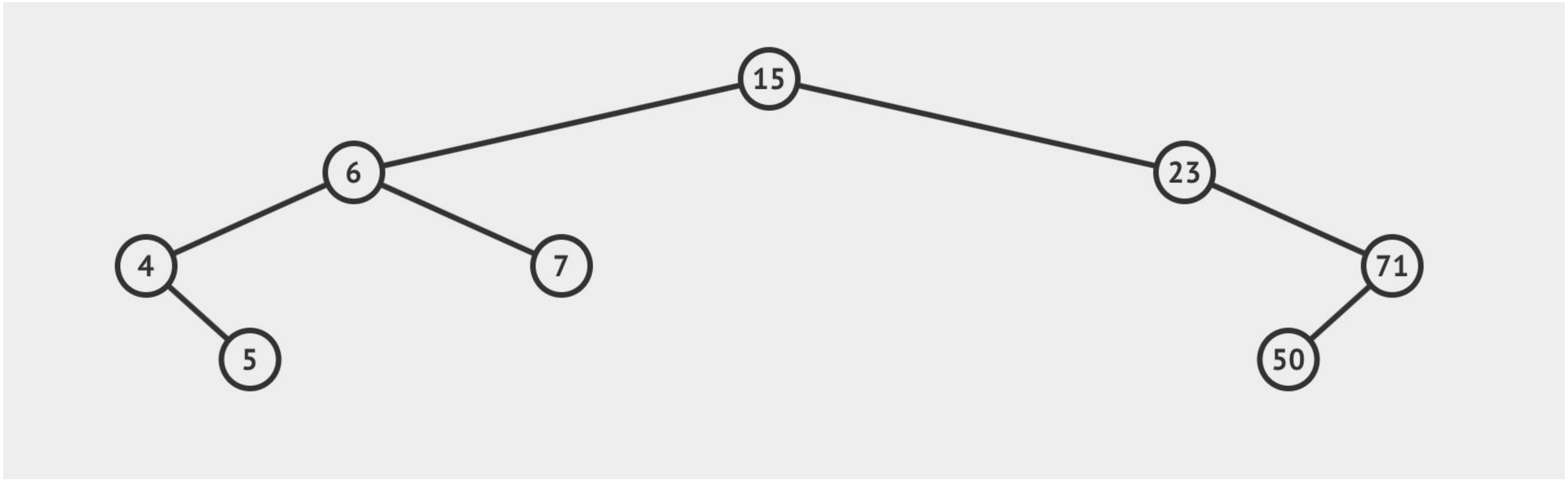
15, 6, 4, 5, 7, 23, 71, 50

```
void traversal(Node* root) {
   if(root == nullptr) return;

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

Postorder Traversal

```
void traversal(Node* root) {
   if(root == nullptr) return;

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

Postorder Traversal



```
void traversal(Node* root) {
    if(root == nullptr) return;

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

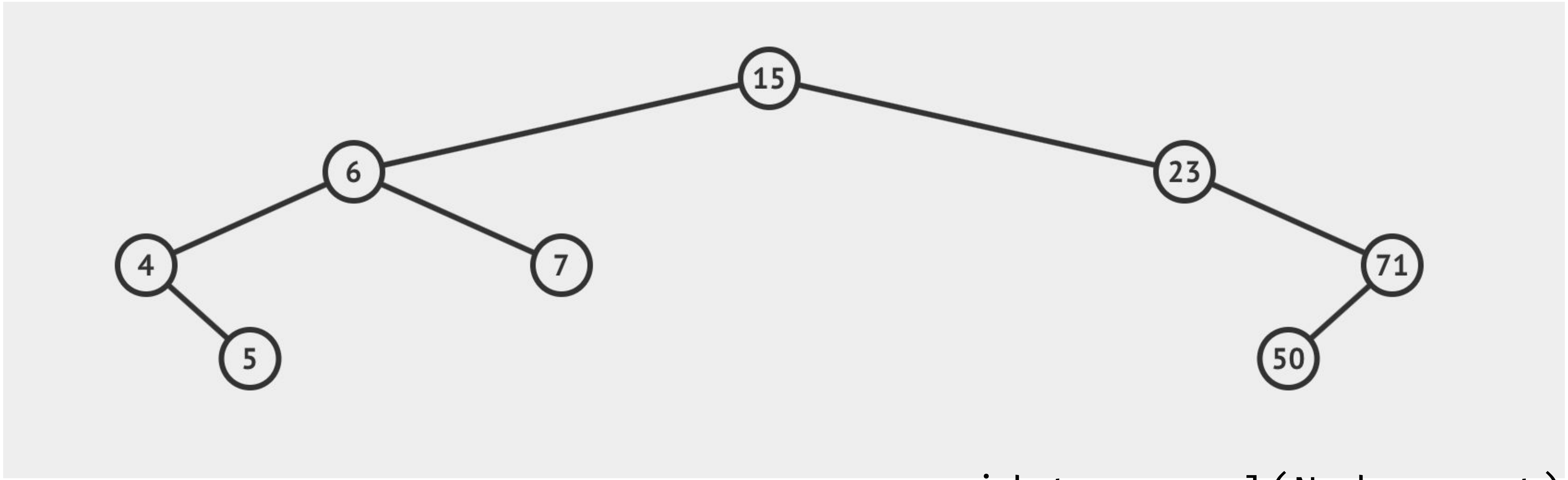
What is the post-order traversal of this tree?

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

Inorder Traversal

```
void traversal(Node* root) {
  if(root == nullptr) return;

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

Inorder Traversal



What is the in-order traversal of this tree?

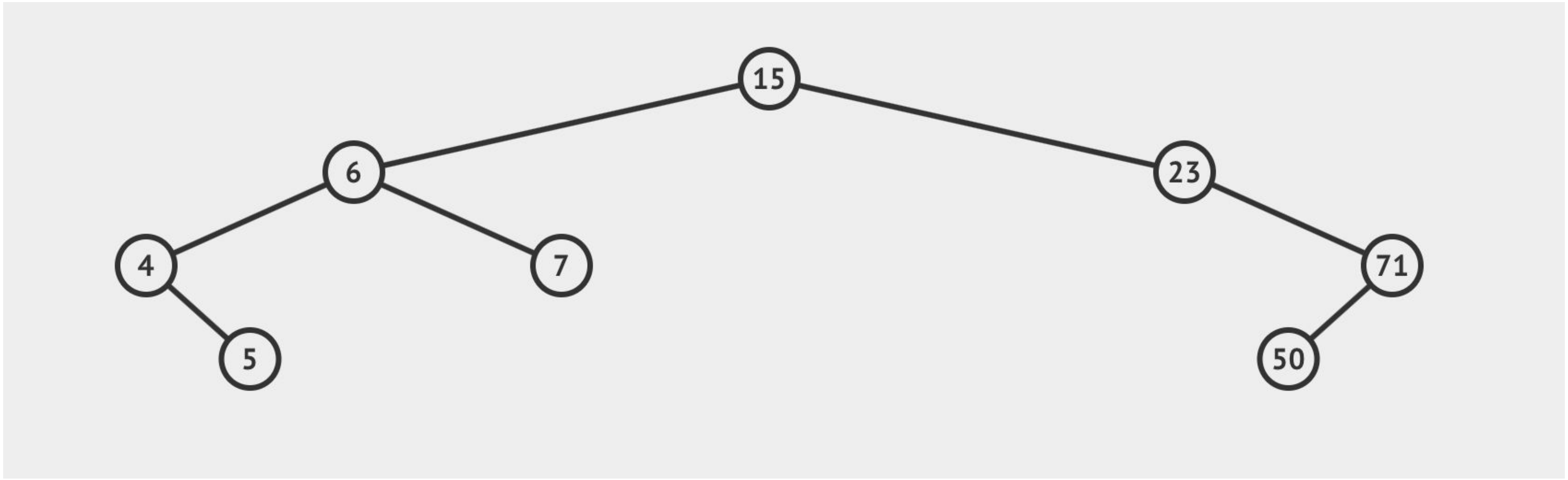
4, 5, 6, 7, 15, 23, 50, 71

```
void traversal(Node* root) {
    if(root == nullptr) return;

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

Level-Order Traversal

```
void traversal(Node* root) {
   if(root == nullptr) return;

   queue<Node*> discovered { { root } };
   while(not discovered.empty()){
      Node* node = discovered.front();
      discovered.pop();
      printNode(node);

      if(node->left != nullptr)
         discovered.push(node->left);
      if(node->right != nullptr)
         discovered.push(node->right);
   }
}
```

Level-Order Traversal



What is the level-order traversal of this tree?

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

Practice: Minimum Level Sum

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
```

```cpp
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 { { root } };
    while (not 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 != nullptr) q.push(temp->left);
            if (temp->right != nullptr) q.push(temp->right);     // push 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 Using Traversals

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

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

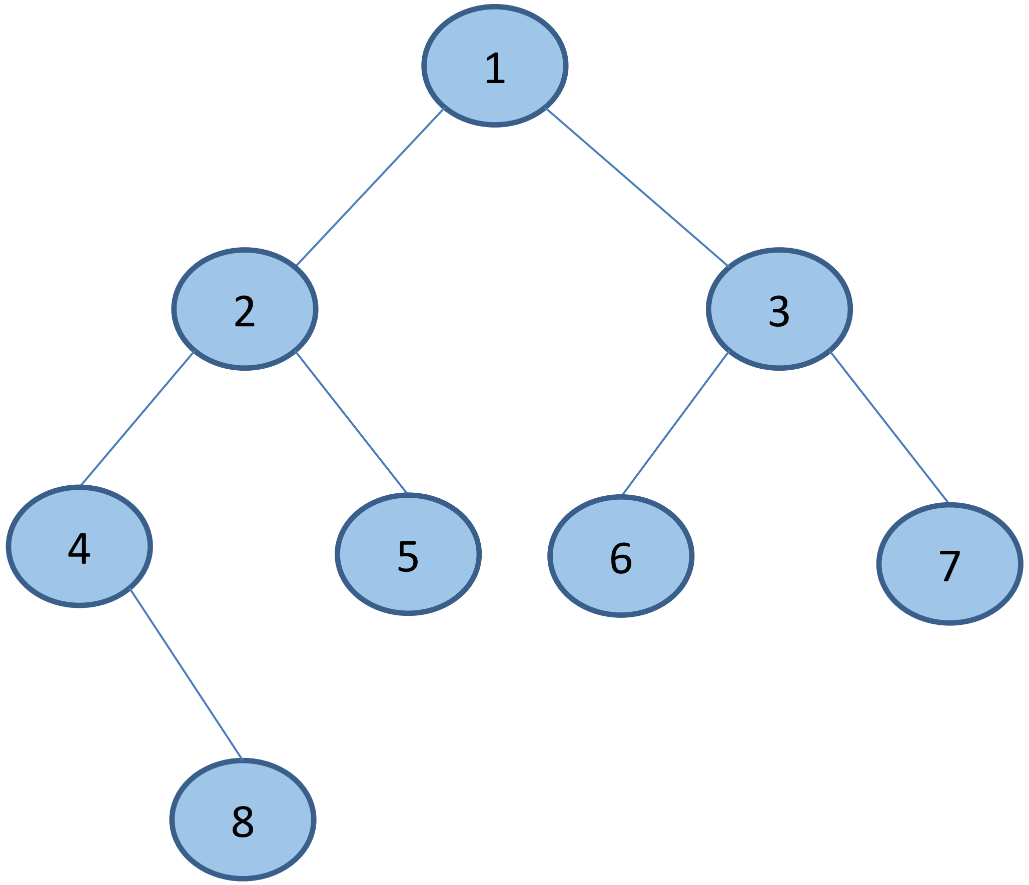
Post-order: 8, 4, 5, 2, 6, 7, 3, 1

Reconstruct a Tree Using Traversals

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

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

See the full version of the slides for a walkthrough!



Binary Search Trees

Binary Search Trees

The key of any node is:

- $\geq$[1,2] the keys of all nodes in its left child
- $\leq$[1] the keys of all nodes in its right child

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)

[1] These can be strengthened to strict relational inequality when no two keys in the tree can compare equivalent

[2] For the purpose of this lab's AG assignment, this is strict greater-than

BST Insertion and Deletion

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

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

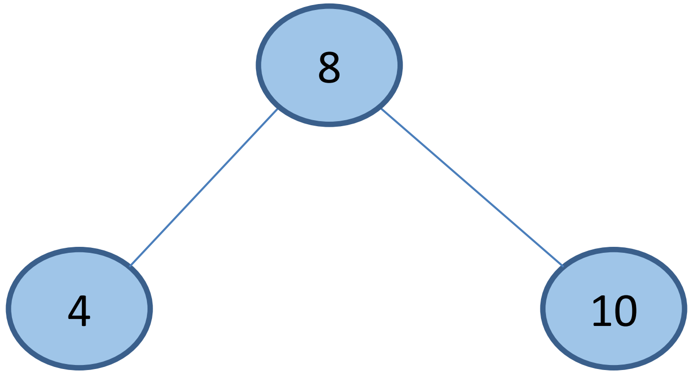
Deletion - Average O(logn); Worst Case O(n)

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 inorder 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:

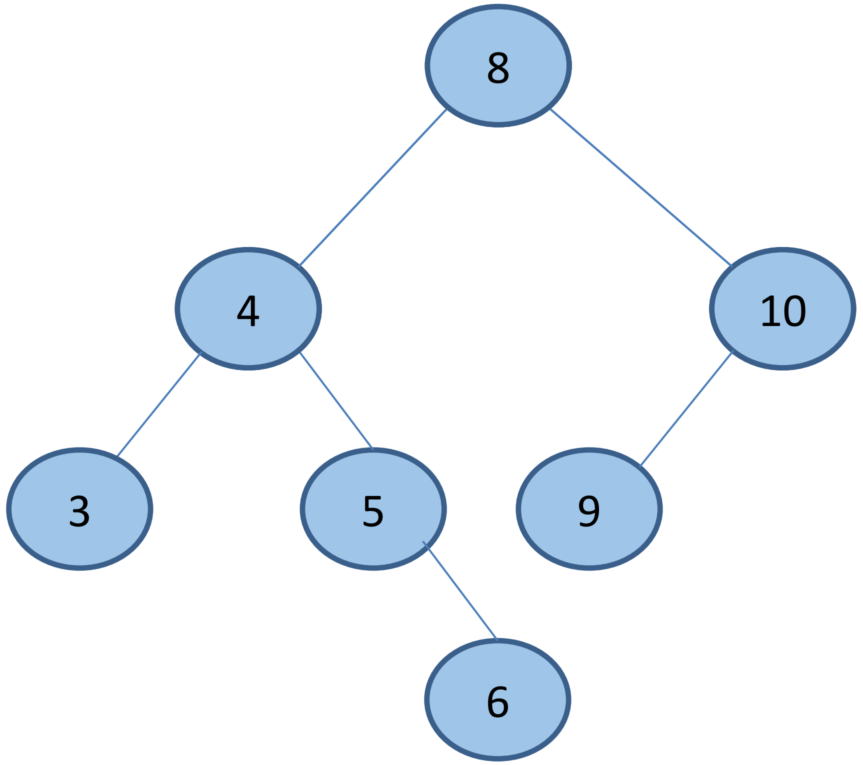
3 5 6 9



Binary Search Trees: Insert

Insert the following to the BST:

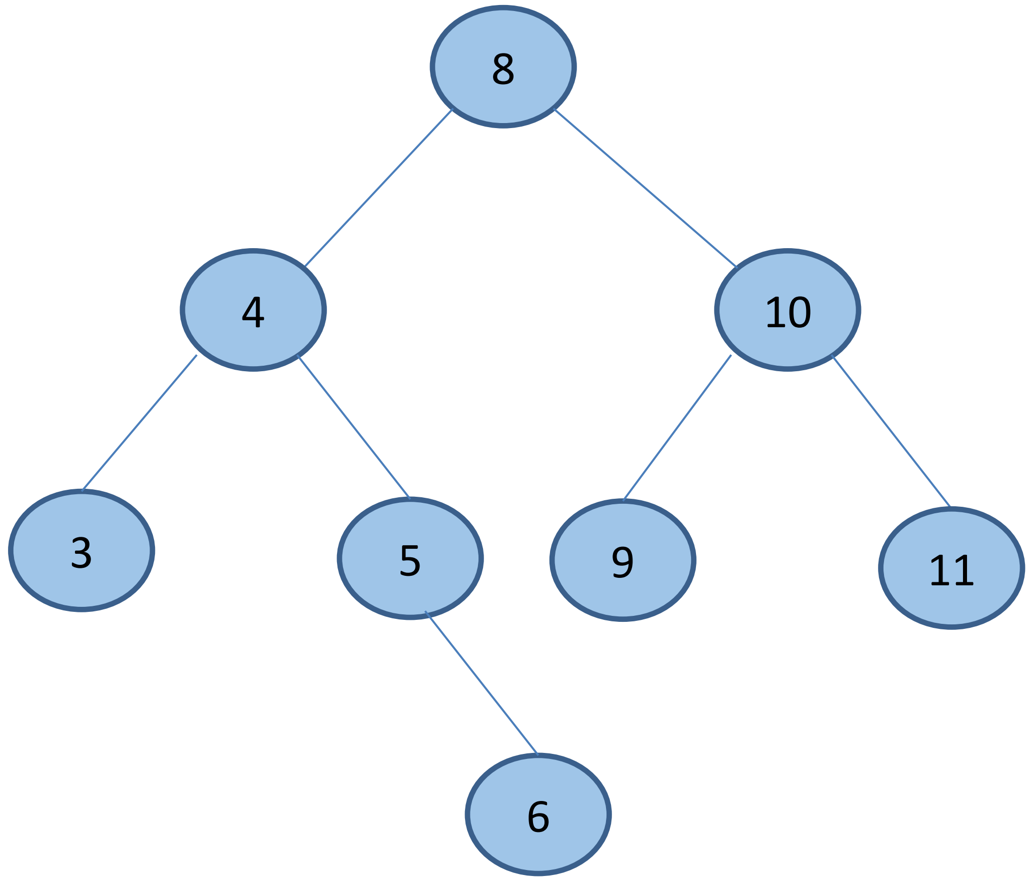
3 5 6 9



Binary Search Trees: Delete

Delete the following from the BST:

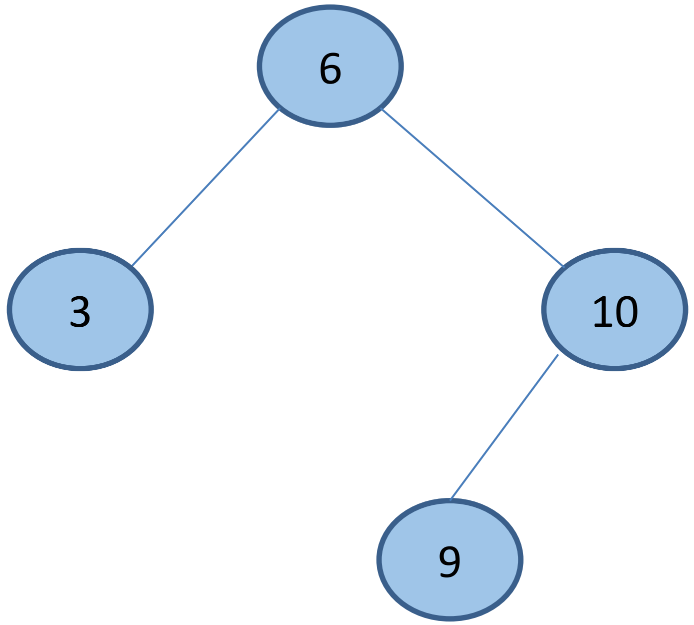
5 4 8 11



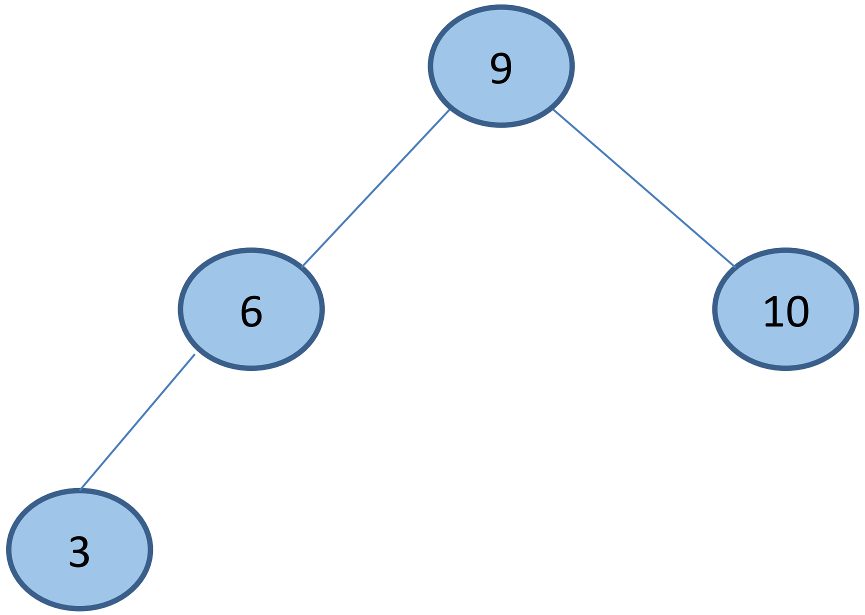
Binary Search Trees: Delete

Delete the following from 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 about a node k?

The heights of the children of k differ by at most 1

What does it mean for a tree to be balanced?

It's balanced about every node

In other words, it's balanced about the root and the root's children are balanced

What can we say about the complexities of insert and search in a balanced tree?

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)
- 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 and 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

Trees: Moving Up

Moving down in a tree is easy. Recurse to a child.

Moving up is not so straightforward... but we said in the last slide that it's necessary!

Recall postorder traversals. They visit the root last. How?

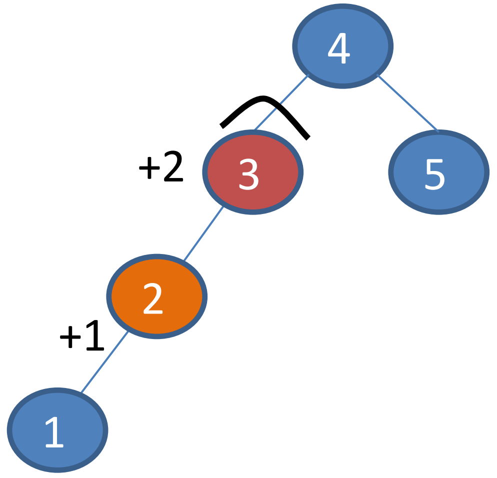
Visit the root *after* traversing a child, as recursion unwinds.

Information can be passed:

- down the tree as arguments to recursive calls
- up the tree as return values from recursive calls

AVL Rotation Case 1 (+,+)

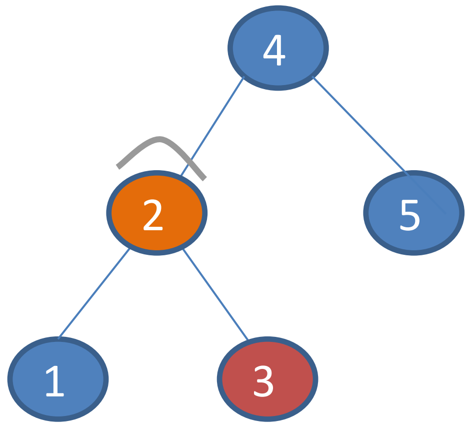
Left subtree causes imbalance and **left** side of that subtree has extra node



Insertion Order:

4, 3, 5, 2, 1

Rotate Right

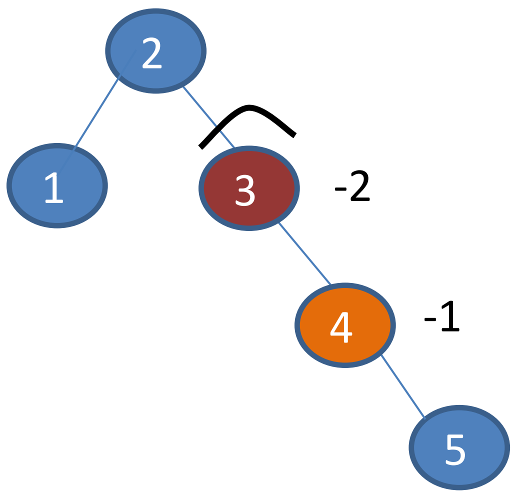


AVL Rotation Case 2 (–,–)

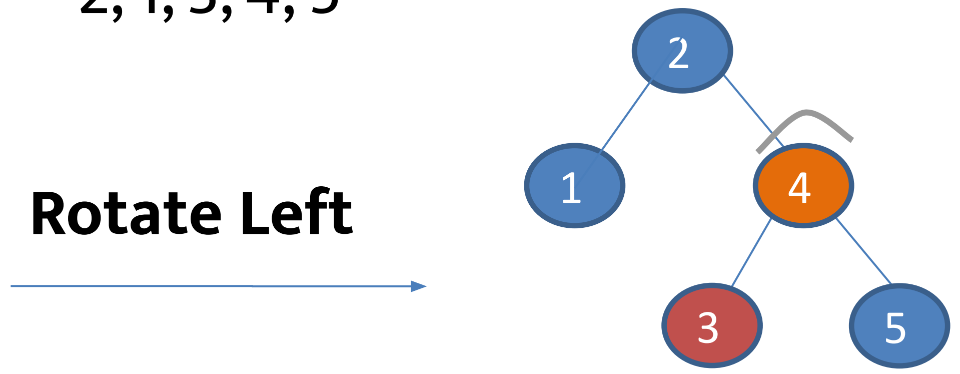
Right subtree causes imbalance, and **right** side of that subtree has extra node

Insertion Order:

2, 1, 3, 4, 5

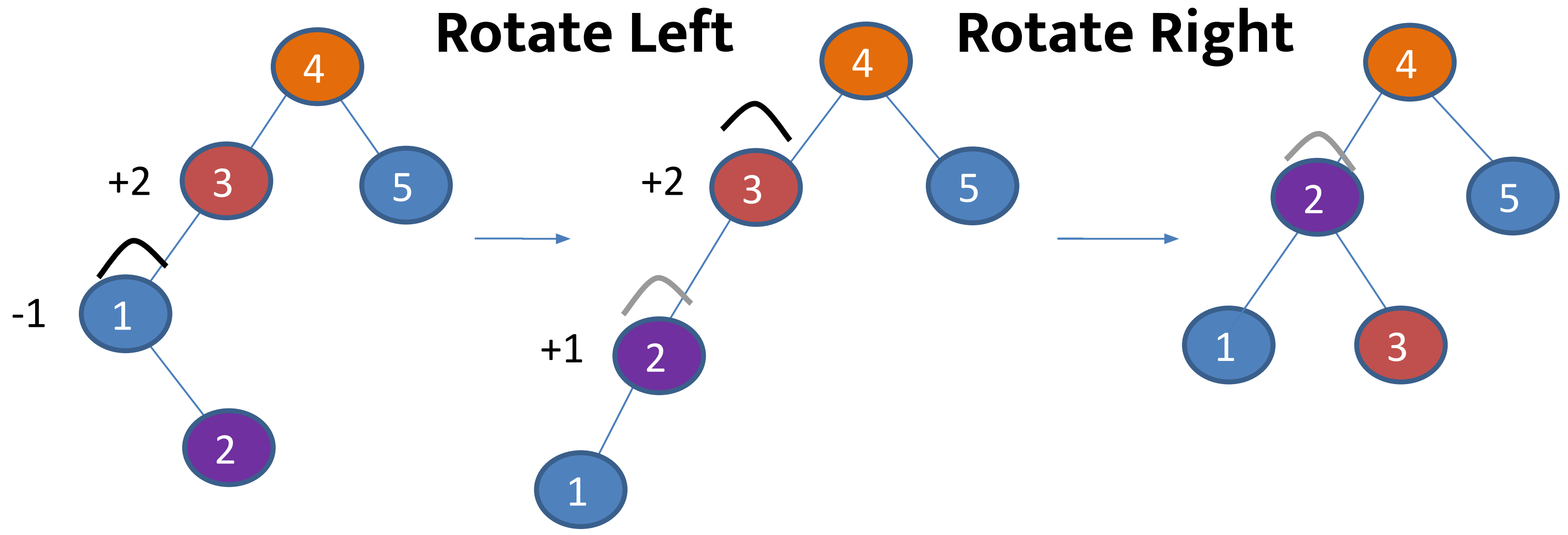


Rotate Left



AVL Rotation Case 3 (+,–)

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

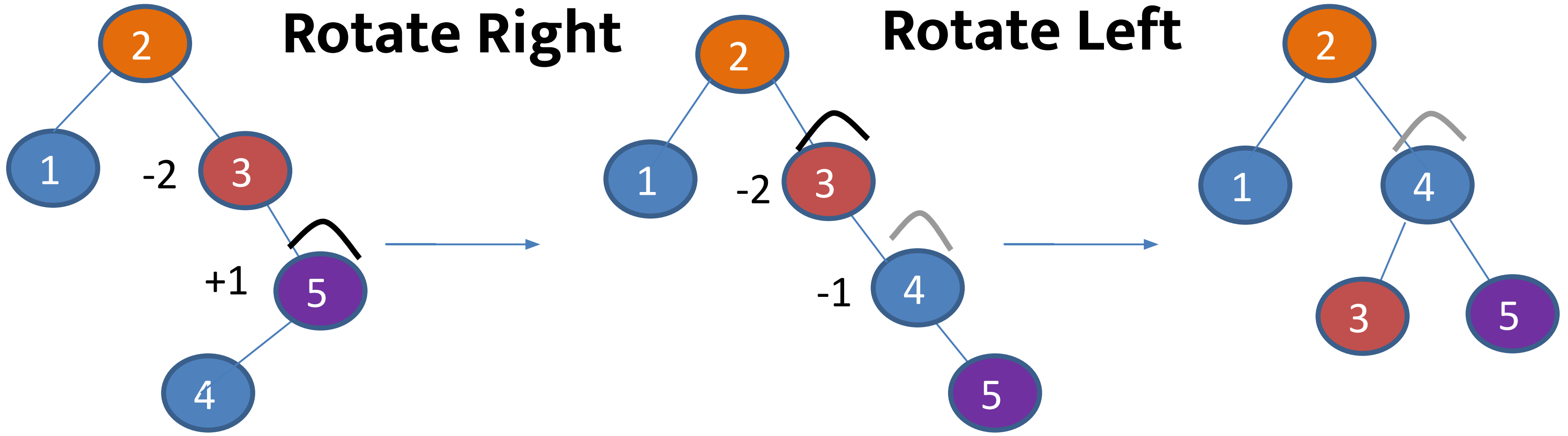


Insertion Order:

4, 5, 3, 1, 2

AVL Rotation Case 4 (–,+)

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

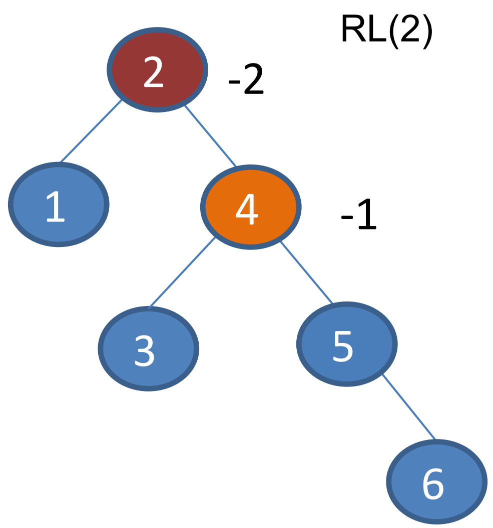


Insertion Order:

2, 1, 3, 5, 4

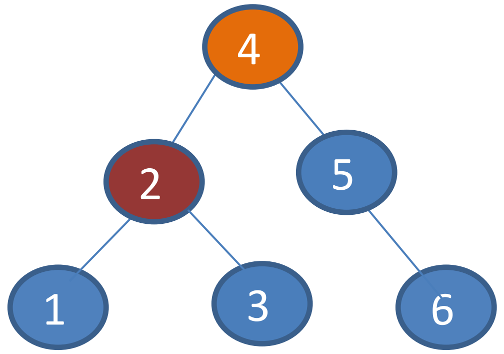
AVL Rotation

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



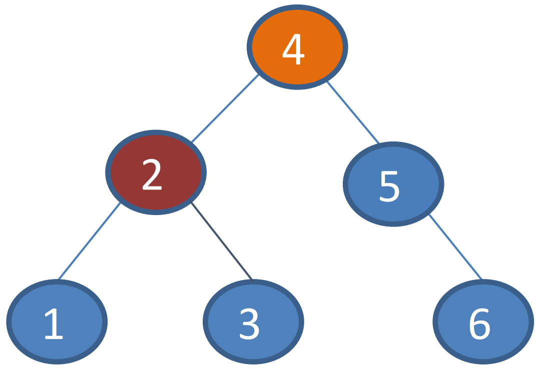
Insertion Order:

2, 1, 4, 3, 5, 6



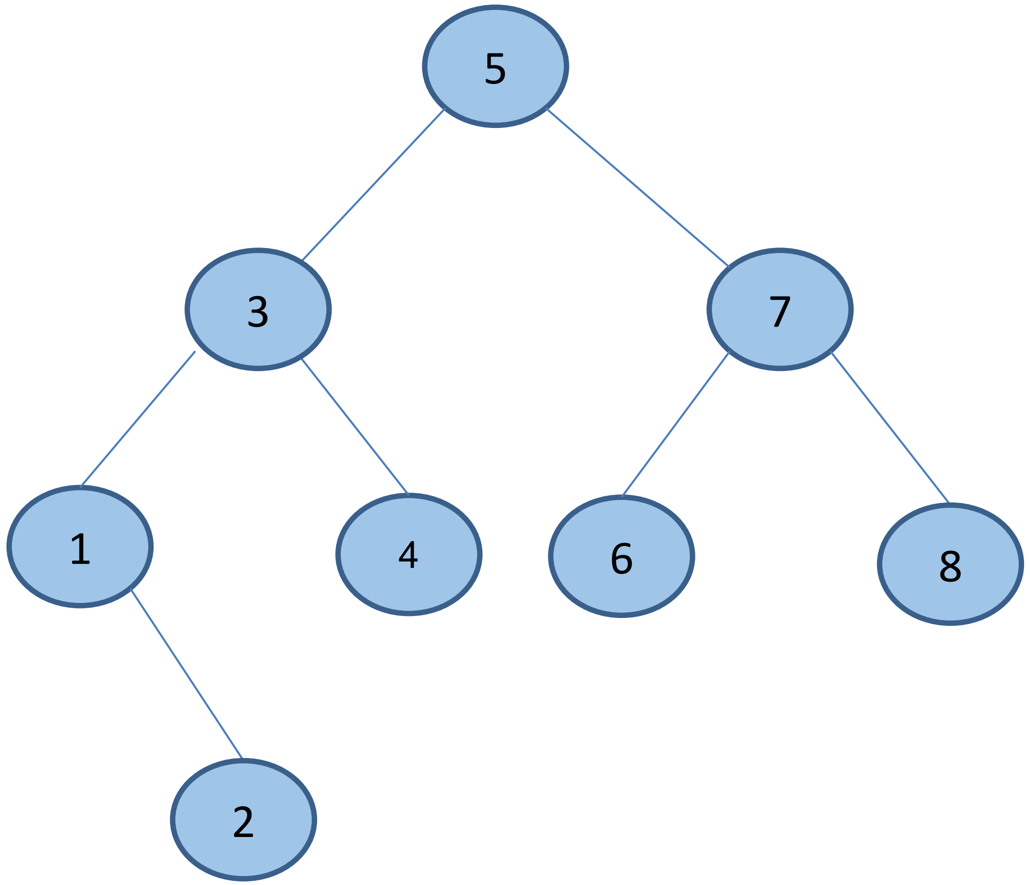
AVL Rotation

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



See the full version of the slides for a walkthrough!

AVL Trees Practice Problem

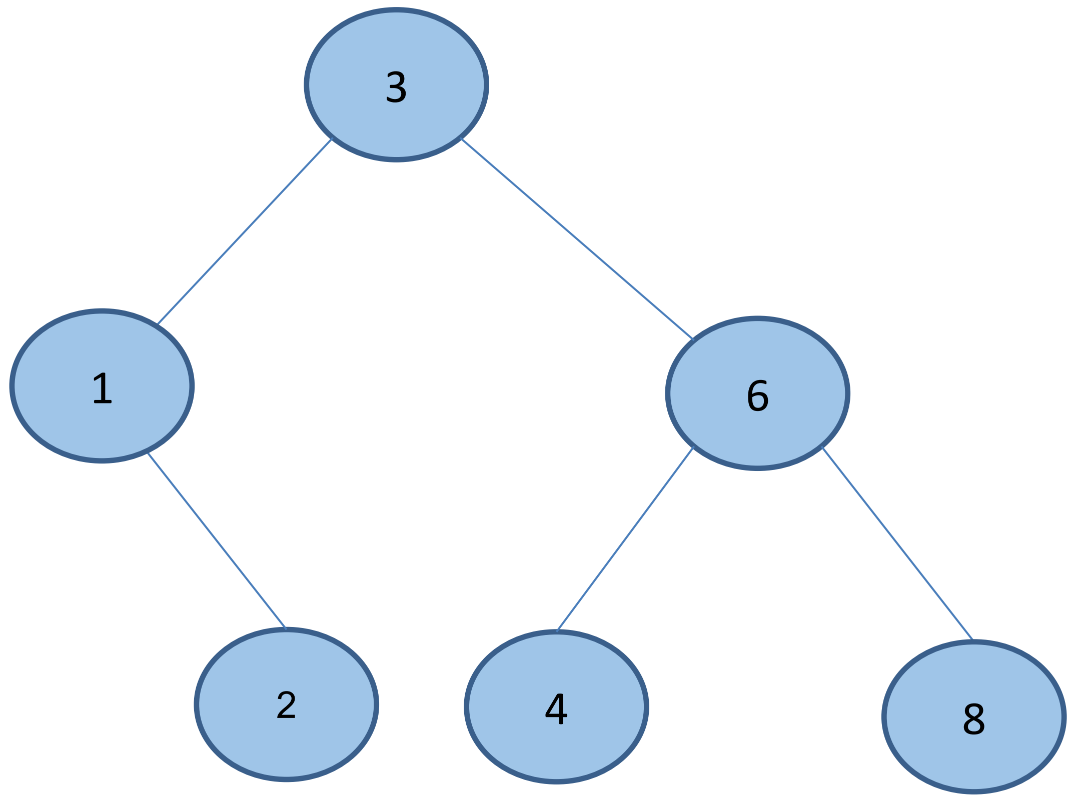


Delete node 5 and then delete node 7. What does the resulting tree look like?

Assume we use in-order successor.

AVL Trees Practice Problem

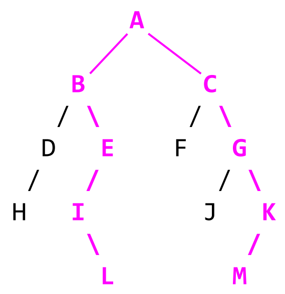
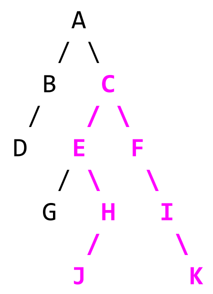
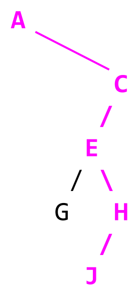
Answer:



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.

		
Diameter: 8	Diameter: 6	Diameter: 4

Handwritten Problem

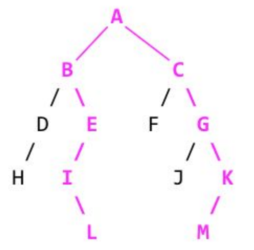
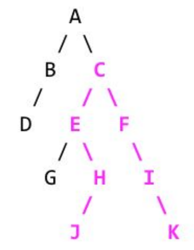
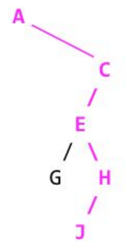
Consider the following Node definition of a binary tree:

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

Your task: Implement the `diameter` function that computes the diameter of a *binary* tree represented by a pointer to an object of type BinaryTreeNode. Assume that an empty tree or a missing child will be represented by `nullptr`. Do not modify the definition of the BinaryTreeNode class. You may write one or more helper functions if you need.

Implement diameter in O(n^2) or better time (try doing it in O(n)!).

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

}
```

 A B C D E F G H I J K L M		
Diameter: 8	Diameter: 6	Diameter: 4