COMP - 285

Advanced Analysis of Algorithms

Welcome to COMP 285

Lecture 9: Introduction to Trees

Chris Lucas (cflucas@ncat.edu)

HW3 was released!

Due week from today @ 11:59pm ET

Tech. Mock Interviews!

10/10-10/13 EC Opportunity!

Career Days!

Every Wednesday! (resumes, interviews +0.5%, etc.)

Schedule time here

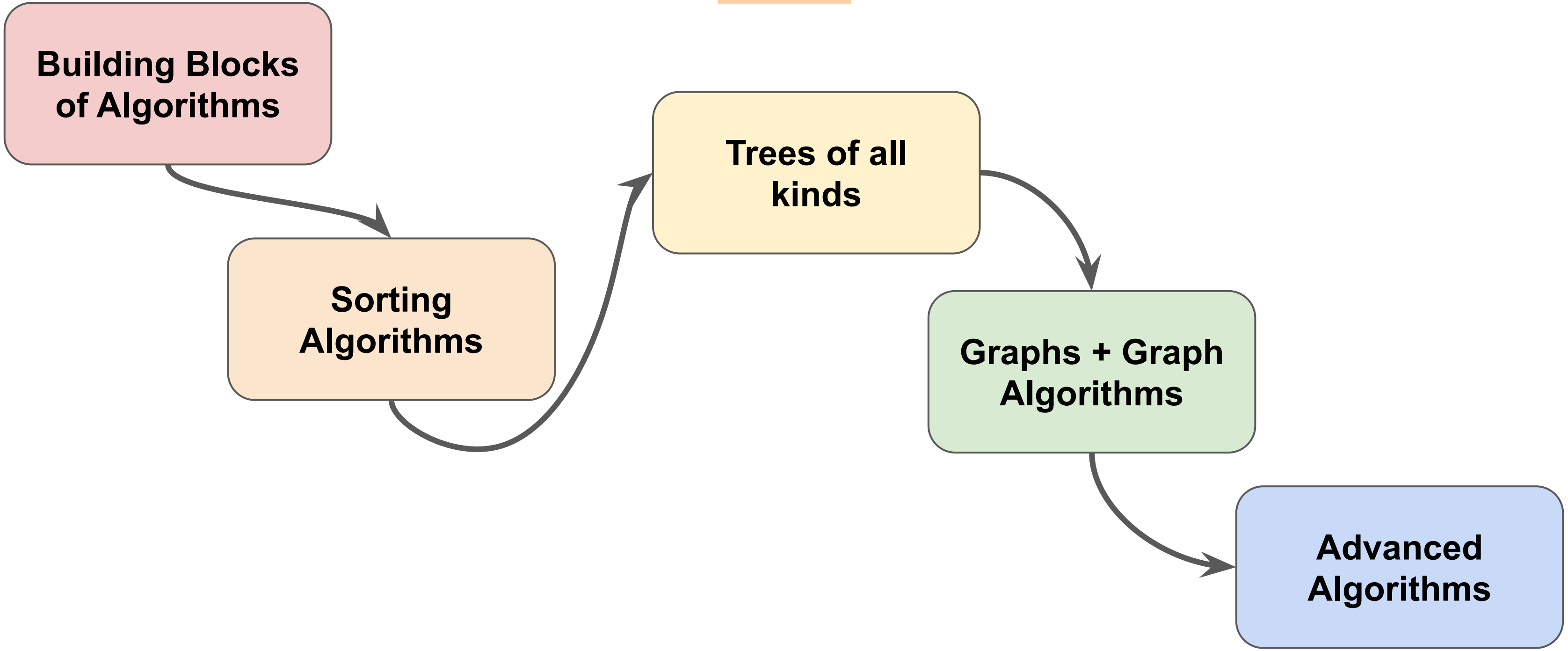
Quiz!

www.comp285-fall22.ml

Big Questions!

- What are trees again?
- How do we represent trees in C++?
- What are some algorithms with trees? (recursion!)
- How can I practice?

Our Course Map!



Building Blocks of Algorithms

Sorting Algorithms

Trees of all kinds

Graphs + Graph Algorithms

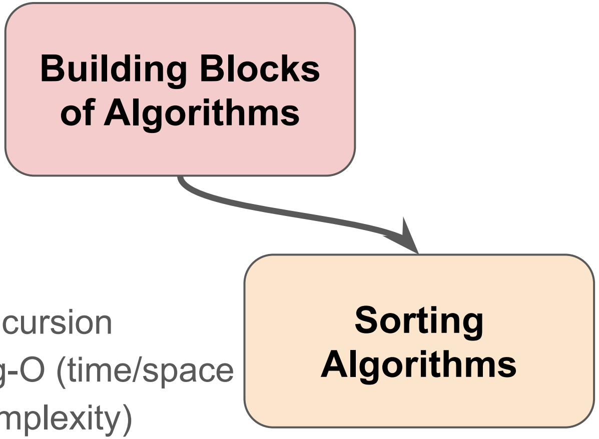
Advanced Algorithms

Our Course Map!

Building Blocks of Algorithms

- Recursion
- Big-O (time/space complexity)
- Data structures (stacks, queues, maps, sets, etc.)
- General problem solving methodology

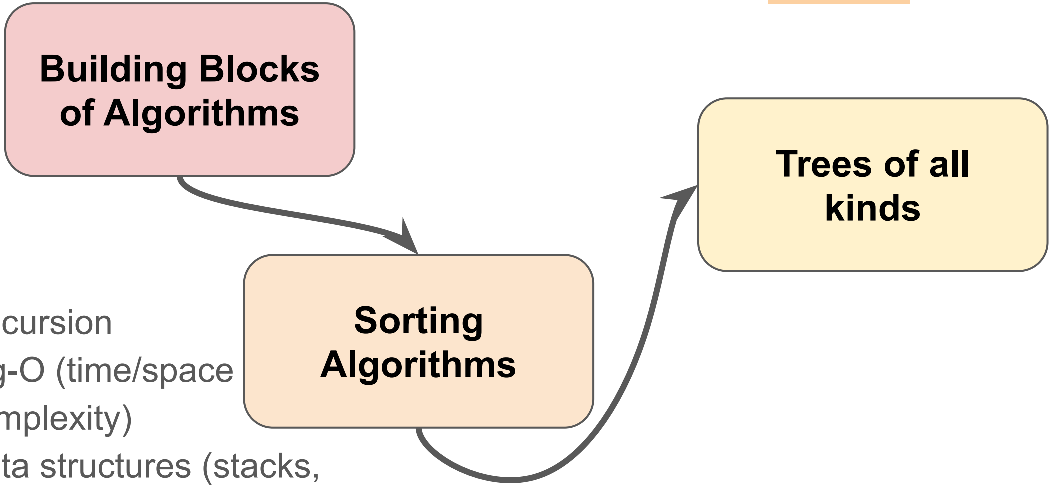
Our Course Map!



- Recursion
- Big-O (time/space complexity)
- Data structures (stacks, queues, maps, sets, etc.)
- General problem solving methodology

- How to arrange and “sort” data in data structures
- Mergesort, InsertionSort, QuickSort
- Master Theorem, linear sorting approaches

Our Course Map!



Building Blocks of Algorithms

- Recursion
- Big-O (time/space complexity)
- Data structures (stacks, queues, maps, sets, etc.)
- General problem solving methodology

Sorting Algorithms

- How to arrange and “sort” data in data structures
- Mergesort, InsertionSort, QuickSort
- Master Theorem, linear sorting approaches

Trees of all kinds

- What is a tree?
- BSTs
- Self-balancing trees
- Topological sorting

Recall where we ended last lecture…

Recurrence Relations and Master Theorem

Recurrence Relations and Master Theorem

- Suppose that $a \geq 1, b > 1,$ and d are constants (independent of n).

Recurrence Relations and Master Theorem

- Suppose that $a \geq 1, b > 1,$ and d are constants (independent of n).
- Suppose $T(n) = a \cdot T\left(\frac{n}{b}\right) + O\left(n^d\right)$. Then

Recurrence Relations and Master Theorem

- Suppose that $a \geq 1, b > 1,$ and d are constants (independent of n).

work needed to combine the solutions

- Suppose $T(n) = a \cdot T\left(\frac{n}{b}\right) + O\left(n^d\right)$. Then

number of subproblems

Factor by which input size shrinks

Recurrence Relations and Master Theorem

- Suppose that $a \geq 1, b > 1,$ and d are constants (independent of n).
- Suppose $T(n) = a \cdot T\left(\frac{n}{b}\right) + O\left(n^d\right)$. Then

work needed to combine the solutions

number of subproblems

Factor by which input size shrinks

$$T(n) = \begin{cases} \mathrm{O}\left(n^d \log(n)\right) & \text{if } a = b^d \\ \mathrm{O}\left(n^d\right) & \text{if } a < b^d \\ \mathrm{O}\left(n^{\log_b(a)}\right) & \text{if } a > b^d \end{cases}$$

The eternal struggle

Branching causes the number of problems to explode!
The most work is at the bottom of the tree!

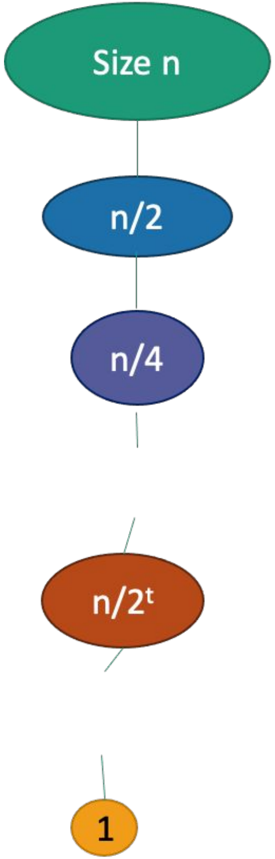
The problems lower in the tree are smaller!
The most work is at the top of the tree!

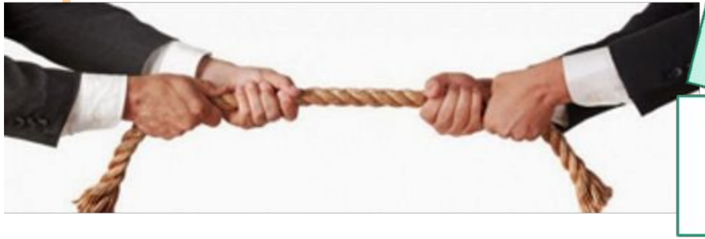
Tall and skinny tree

$$1.\ T(n) = T\left(\frac{n}{2}\right) + n, \qquad (a < b^d)$$

- The amount of work done at the top (the biggest problem) swamps the amount of work done anywhere else.

- T(n) = O(work at top) = O(n)

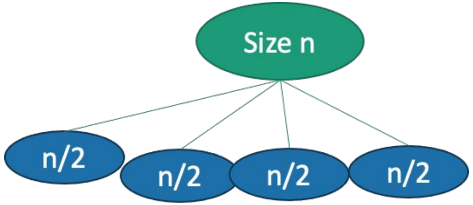




Most work at the top of the tree!

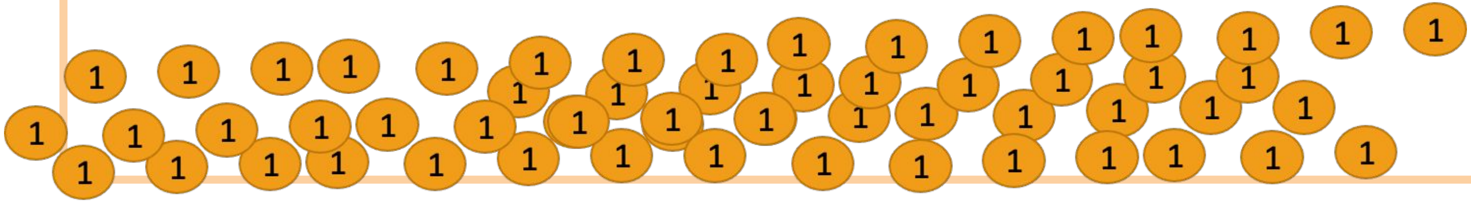
Needlessly recursive mult.: bushy tree

$$3.\ \ T(n) = 4 \cdot T\left(\frac{n}{2}\right) + n, \qquad (a > b^d)$$



- There are a HUGE number of leaves, and the total work is dominated by the time to do work at these leaves.
- T(n) = O(work at bottom) = O($4^{\text{depth of tree}}$) = O(n^2)

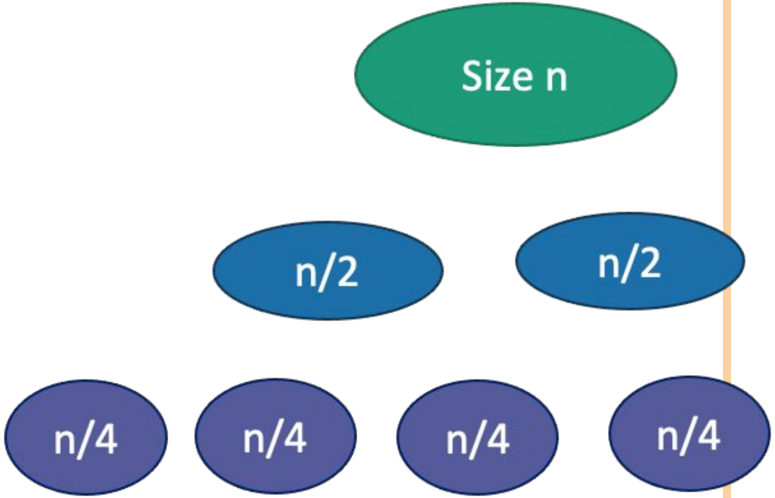
Most work at the bottom of the tree!



MergeSort: Just right

2. $T(n) = 2 \cdot T\left(\frac{n}{2}\right) + n, \qquad \left(a = b^d\right)$

- The branching **just** balances out the amount of work.
- The same amount of work is done at every level.



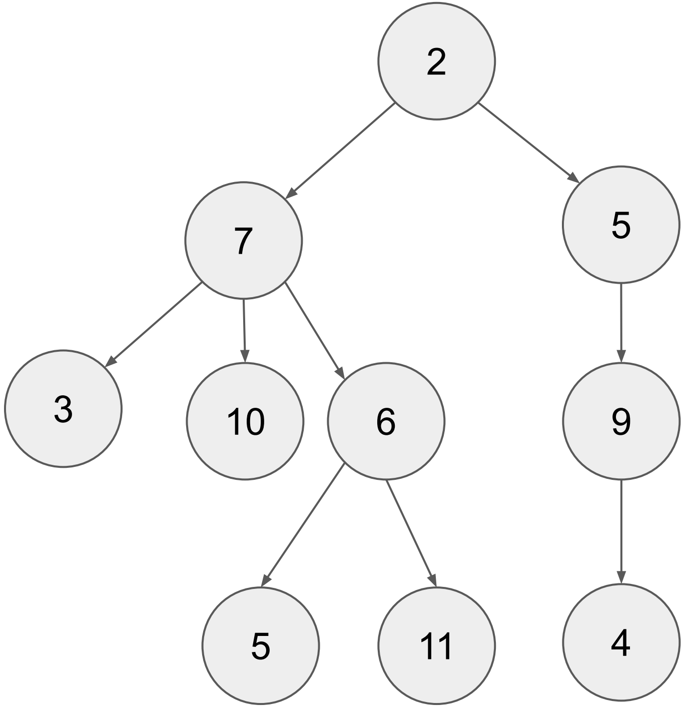
- T(n) = (number of levels) * (work per level)
- = log(n) * O(n) = O(n log(n))

Big Questions!

- What are trees again?
- How do we represent trees in C++?
- What are some algorithms with trees? (recursion!)
- How can I practice?

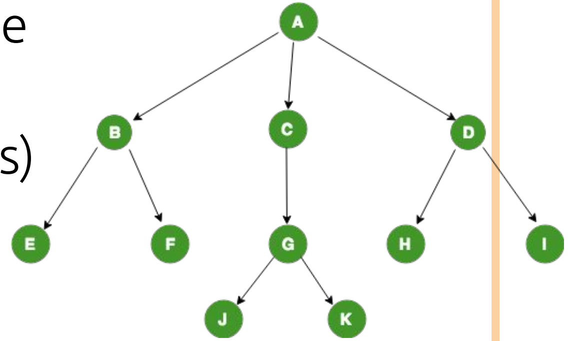
Tree

- A Tree is a **hierarchical** data structure that has a value and children. Each child is also a Tree, making this data structure **recursive** in nature.
- Don't confuse general N-ary Trees with Binary Trees (a special kind of tree where each node has at most two children) or Binary Search Trees (a special kind of binary tree where left subtree is less and right subtree is greater).



Motivation

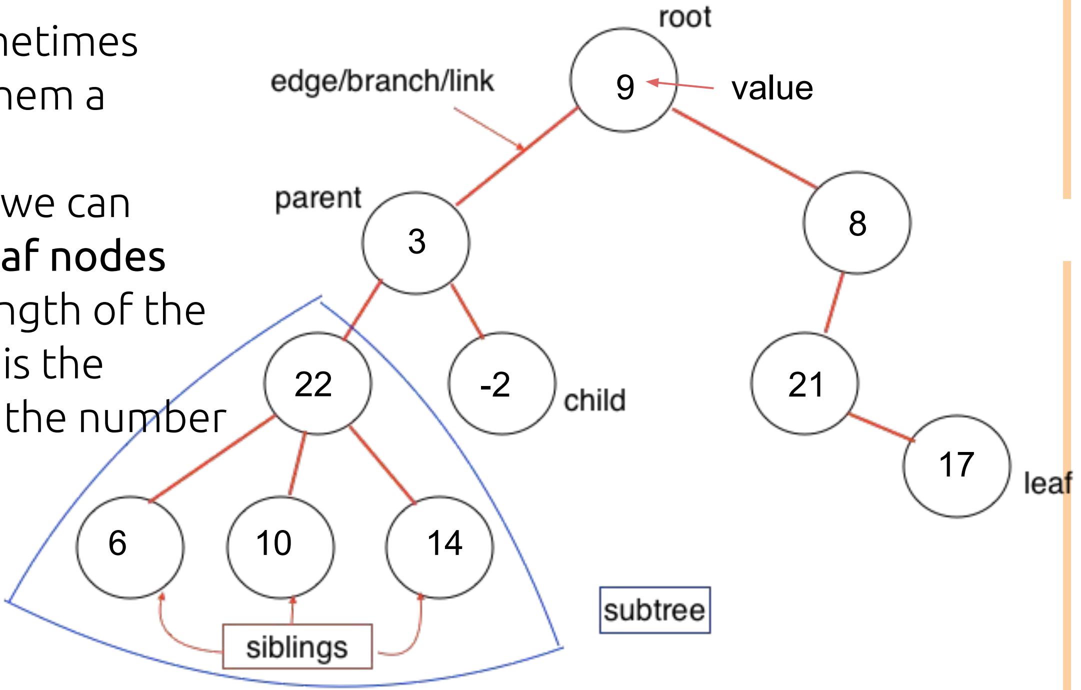
- In software engineering, trees are used everywhere
  - Heaps (priority queue implementation)
  - Text analysis: autocomplete & spell-check (tries)
  - Data systems (to internally represent data)
    - File systems, directories, etc.
  - Game AIs - Reinforcement Learning
  - Programming language compilers (Abstract Syntax Trees) or Parse Trees (HTML/XML)
    - ex
- Trees and writing algorithms on trees are reasonably common whiteboard coding interview questions.



Tree Terminology

height (and deepest level) = 3

- A tree is made up of **nodes**.
- Each node has a **value** and sometimes point to **child nodes,** making them a **parent node**.
- We start at the **root node** and we can follow pointers down to the **leaf nodes** which have no children. The length of the longest path from root to leaf is the **height**. The **depth** of a node is the number of hops away from root node
- If A can reach B downward:
  - B is a **descendant** of A
  - A is an **ancestor** of B



Polls

What is value of the root of this tree?

2

How many children does the root have?

2

How many leaves does this tree have?

5

How many nodes does this tree have?

10

What is the height of the tree?

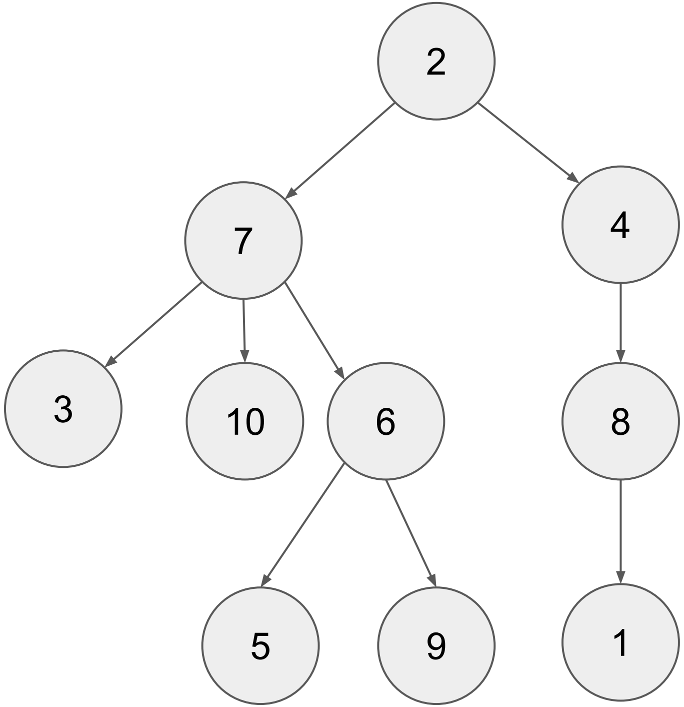
3

How many ancestors does 9 have?

3

Is the following a Binary Tree?

No



Big Questions!

- What are trees again?
- How do we represent trees in C++?
- What are some algorithms with trees? (recursion!)
- How can I practice?

Representing Trees in C++

This is a custom TreeNode class I have written for this class, but it is very similar to what is provided as an interface in most interview settings. It has private fields we can access with helpers:

- **`getValue()`:** returns value of node (generic type T)
- **`getChildren()`:** returns children pointers vector<TreeNode<T>*> (as we can have any # of children)

Example construction:

```
 new TreeNode<int>(
   20,  // value
   {new TreeNode<int>(2, {})})  // children
```

Representing Trees in C++

N-ary Tree (TreeNode.h)

- **`getValue()`**: returns value of node (generic type T)
- **`getChildren()`**: returns children pointers vector<TreeNode<T>*> (as we can have any # of children)

Constructor:

```
 TreeNode(
   T value,
   std::vector<TreeNode<T>*> children = {})
```

Binary Tree (BinaryTree.h)

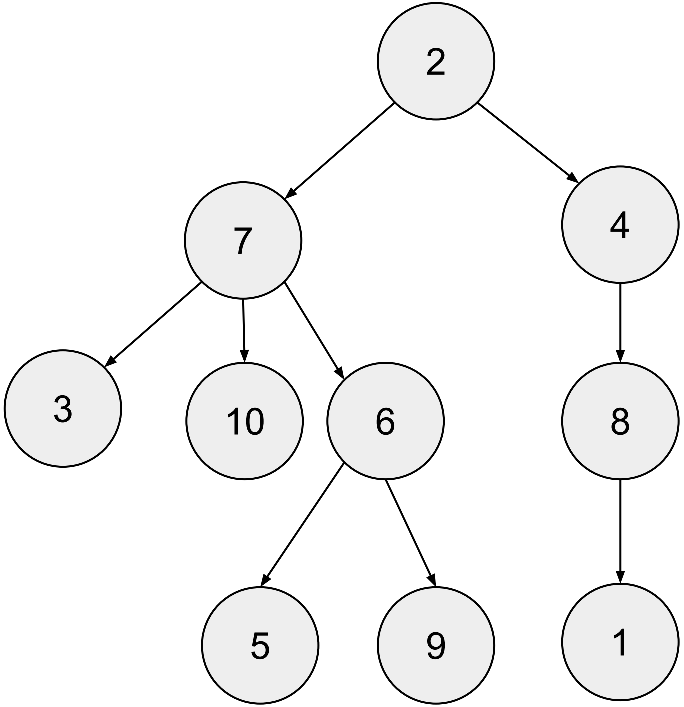
- **`getValue()`**: returns value of node (generic type T)
- **`getLeft()`**: returns left child (TreeNode<T>*)
- **`getRight()`**: returns right child (TreeNode<T>*)

Constructor:

```
 BinaryTree(
   T value,
   BinaryTree<T>* left = nullptr,
   BinaryTree<T>* right = nullptr)
```

Representing Trees in C++

```
TreeNode<int>* ex1 = new TreeNode<int>(
    2, {
      new TreeNode<int>(
        7, {
          new TreeNode<int>(3),
          new TreeNode<int>(10),
          new TreeNode<int>(
            6, {
              new TreeNode<int>(5),
              new TreeNode<int>(9)})}),
      new TreeNode<int>(
        4, {
          new TreeNode<int>(
            8, {
              new TreeNode<int>(1)})})});
```



Poll

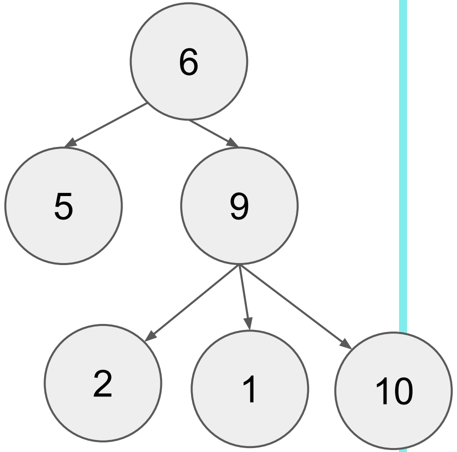
Which code snippet represents the tree to the right?

(Answer #1)

```
new TreeNode<int>(
     6, {
          new TreeNode<int>(5),
          new TreeNode<int>(9, {
               new TreeNode<int>(10),
               new TreeNode<int>(1),
               new TreeNode<int>(2)
          })
     })
```

(Answer #2)

```
new TreeNode<int>(
     6, {
          new TreeNode<int>(9),
          new TreeNode<int>(5, {
               new TreeNode<int>(10),
               new TreeNode<int>(1),
               new TreeNode<int>(2)
          })
     })
```



(Answer #3) None of the above

Poll

Which code snippet represents the tree to the right?

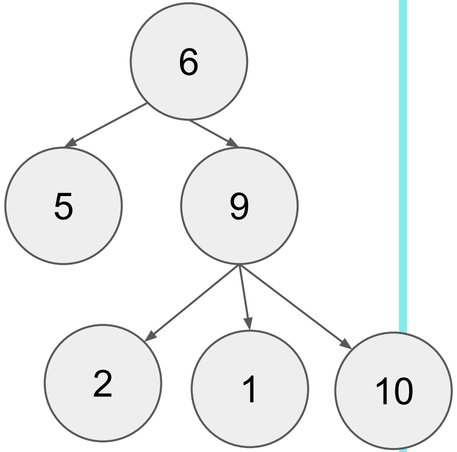
```
new TreeNode<int>(
     6, {
          new TreeNode<int>(5),
          new TreeNode<int>(9, {
               new TreeNode<int>(2),
               new TreeNode<int>(1),
               new TreeNode<int>(10)
          })
     })
```

(Answer #1)

```
new TreeNode<int>(
     6, {
          new TreeNode<int>(5),
          new TreeNode<int>(9, {
               new TreeNode<int>(10),
               new TreeNode<int>(1),
               new TreeNode<int>(2)
          })
     })
```

(Answer #2)

```
new TreeNode<int>(
     6, {
          new TreeNode<int>(9),
          new TreeNode<int>(5, {
               new TreeNode<int>(10),
               new TreeNode<int>(1),
               new TreeNode<int>(2)
          })
     })
```



(Answer #3) None of the above

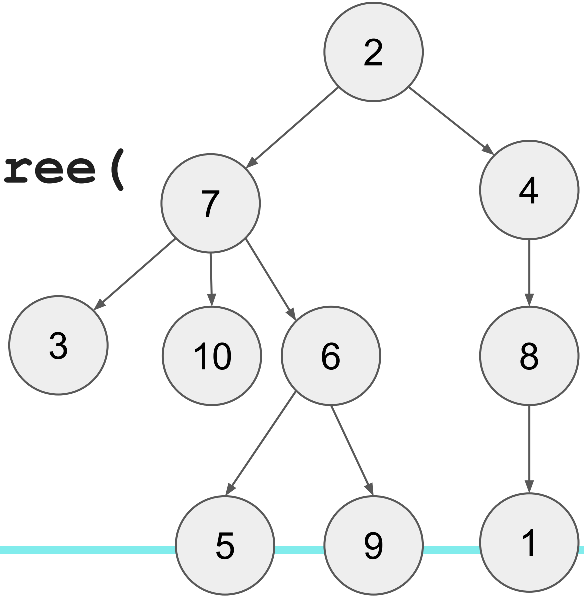
Big Questions!

- What are trees again?
- How do we represent trees in C++?
- What are some algorithms with trees? (recursion!)
- How can I practice?

`findSumOfTree`

Write an algorithm that takes in a tree of ints, and returns the sum of all the values within the tree.

`findSumOfTree(`



`) outputs 55`

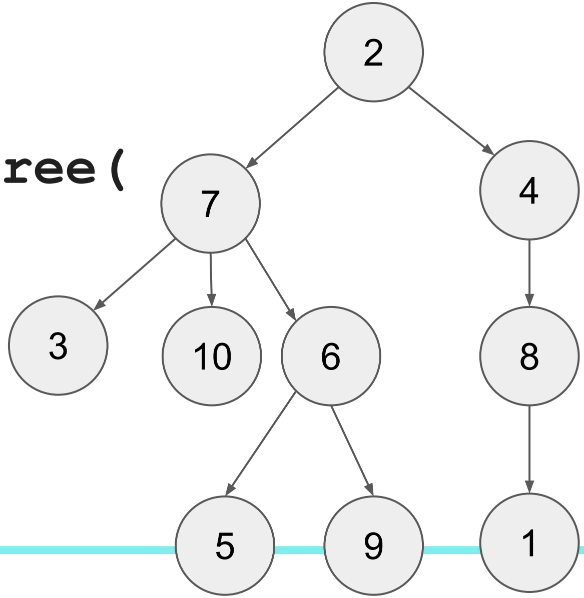
findSumOfTree

Base case: ?

Recursive case: ?

Combination: ?

findSumOfTree(



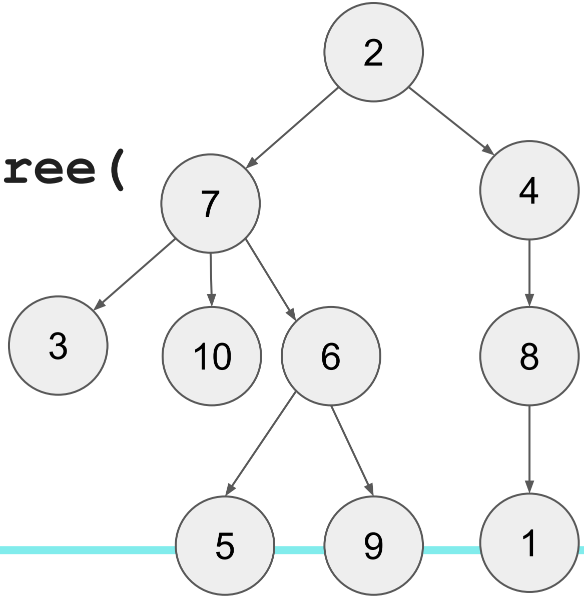
) outputs 55

`findSumOfTree`

Base case: Leaf node can return its value directly

Recursive case: ?

Combination: ?

`findSumOfTree(`  `) outputs 55`

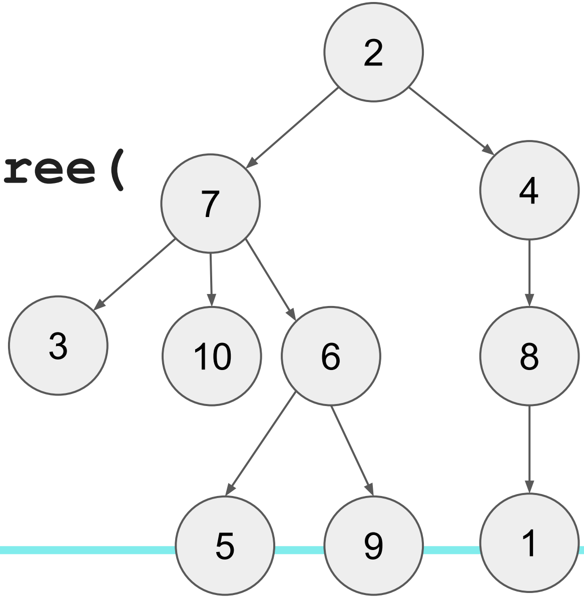
findSumOfTree

Base case: Leaf node can return its value directly

Recursive case: findSumOfTree on children

Combination: ?

findSumOfTree(



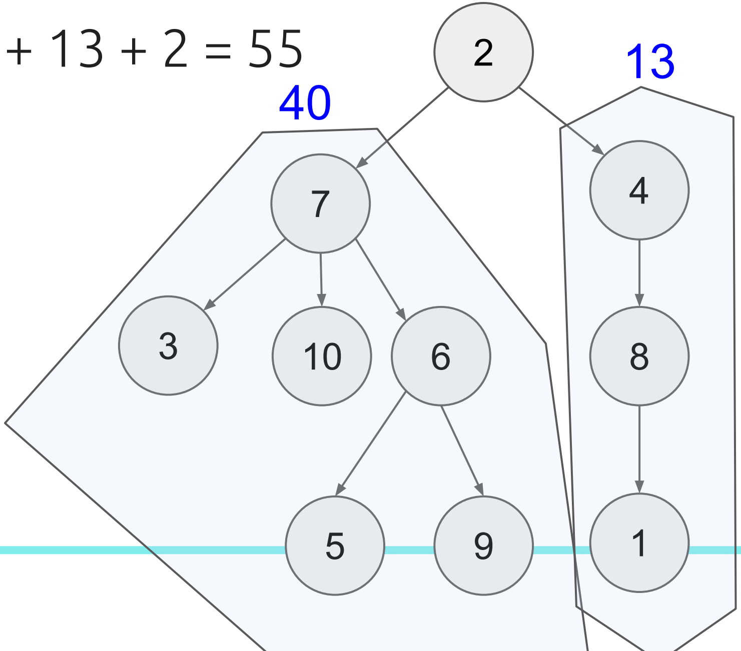
) outputs 55

findSumOfTree

Base case: Leaf node can return its value directly

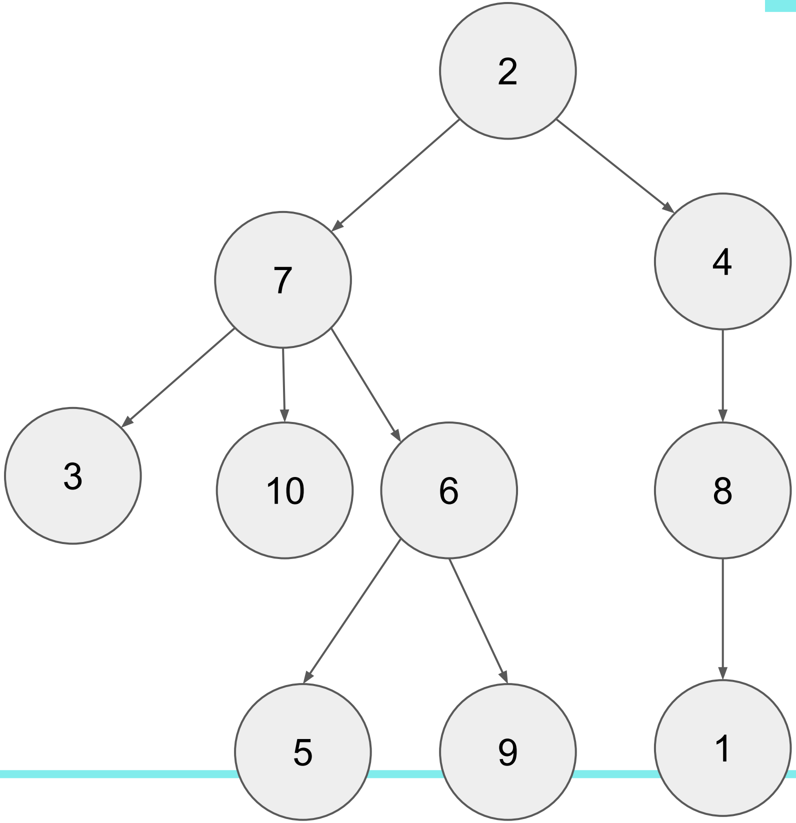
Recursive case: findSumOfTree on children

Combination: 40 + 13 + 2 = 55



Let’s code it!!!

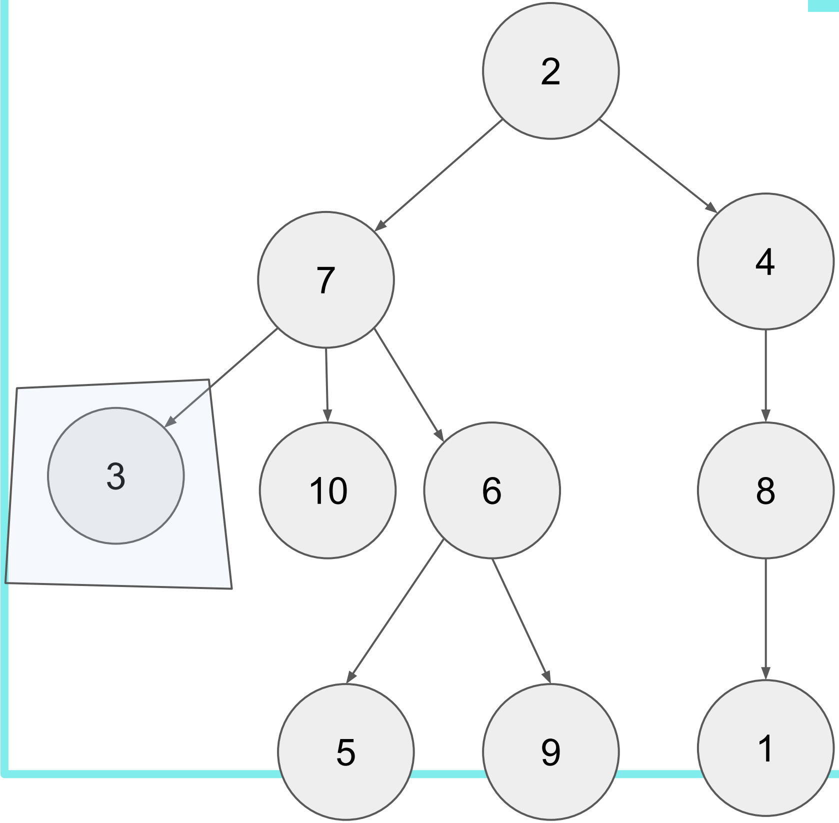
findSumOfTree



What actually happens...

```
int findSumOfTree(TreeNode<int>* root) {
 if (root->isLeaf()) {
   return root->getValue();
 }
 int sumSoFar = 0;
 for (auto n: root->getChildren()) {
   sumSoFar += findSumOfTree(n);
 }
 return sumSoFar + root->getValue();
}
```

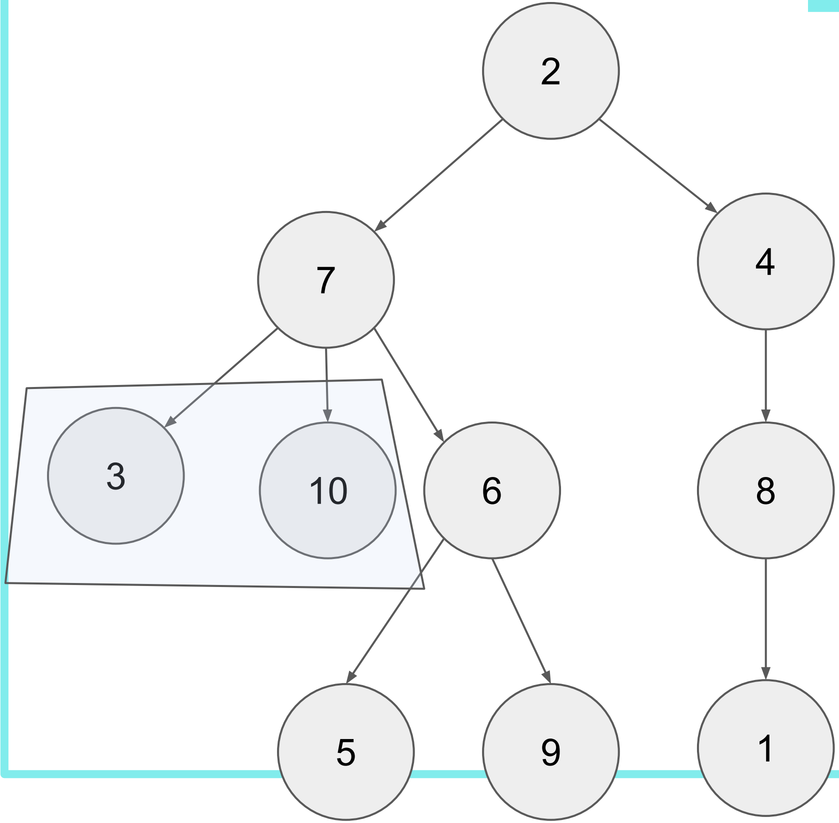
findSumOfTree



What actually happens...

```
int findSumOfTree(TreeNode<int>* root) {
 if (root->isLeaf()) {
   return root->getValue();
 }
 int sumSoFar = 0;
 for (auto n: root->getChildren()) {
   sumSoFar += findSumOfTree(n);
 }
 return sumSoFar + root->getValue();
}
```

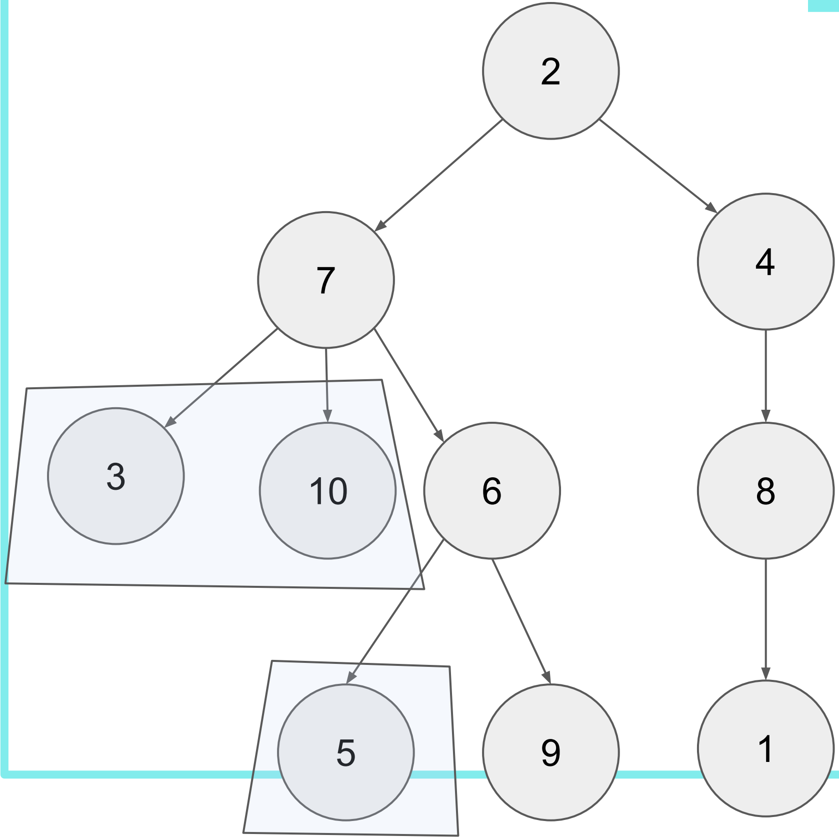
findSumOfTree



What actually happens...

```
int findSumOfTree(TreeNode<int>* root) {
 if (root->isLeaf()) {
   return root->getValue();
 }
 int sumSoFar = 0;
 for (auto n: root->getChildren()) {
   sumSoFar += findSumOfTree(n);
 }
 return sumSoFar + root->getValue();
}
```

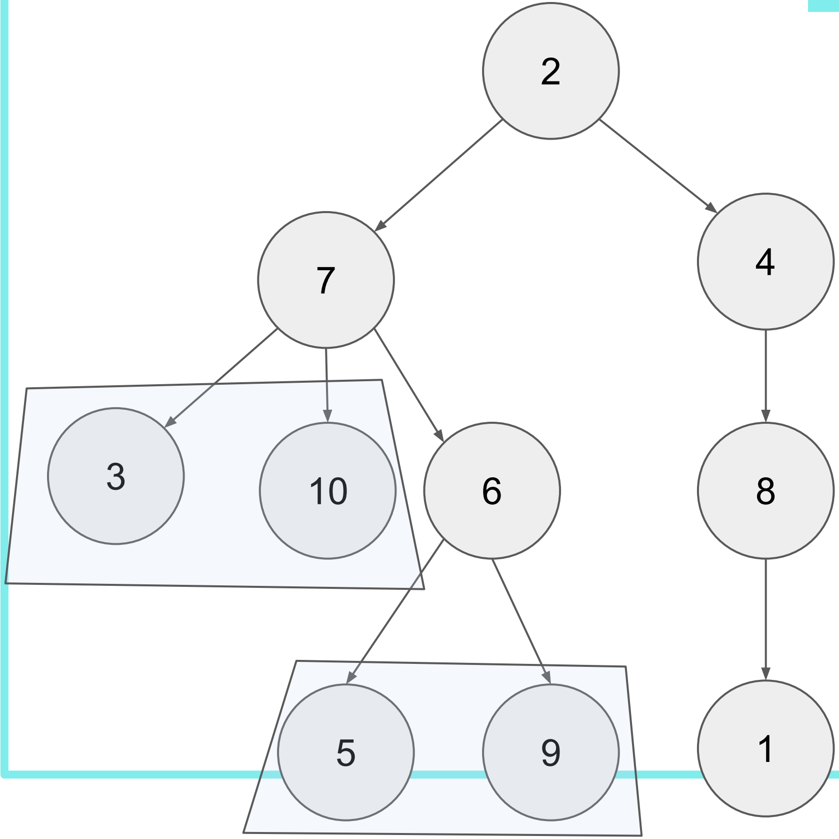
findSumOfTree



What actually happens...

```
int findSumOfTree(TreeNode<int>* root) {
 if (root->isLeaf()) {
   return root->getValue();
 }
 int sumSoFar = 0;
 for (auto n: root->getChildren()) {
   sumSoFar += findSumOfTree(n);
 }
 return sumSoFar + root->getValue();
}
```

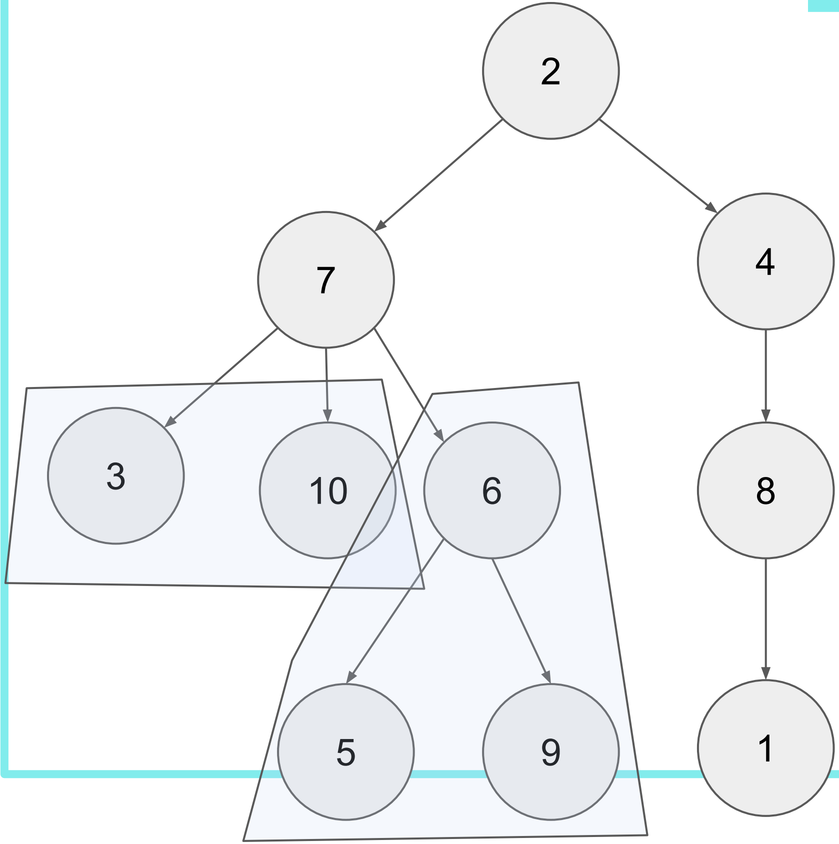
findSumOfTree



What actually happens...

```
int findSumOfTree(TreeNode<int>* root) {
 if (root->isLeaf()) {
   return root->getValue();
 }
 int sumSoFar = 0;
 for (auto n: root->getChildren()) {
   sumSoFar += findSumOfTree(n);
 }
 return sumSoFar + root->getValue();
}
```

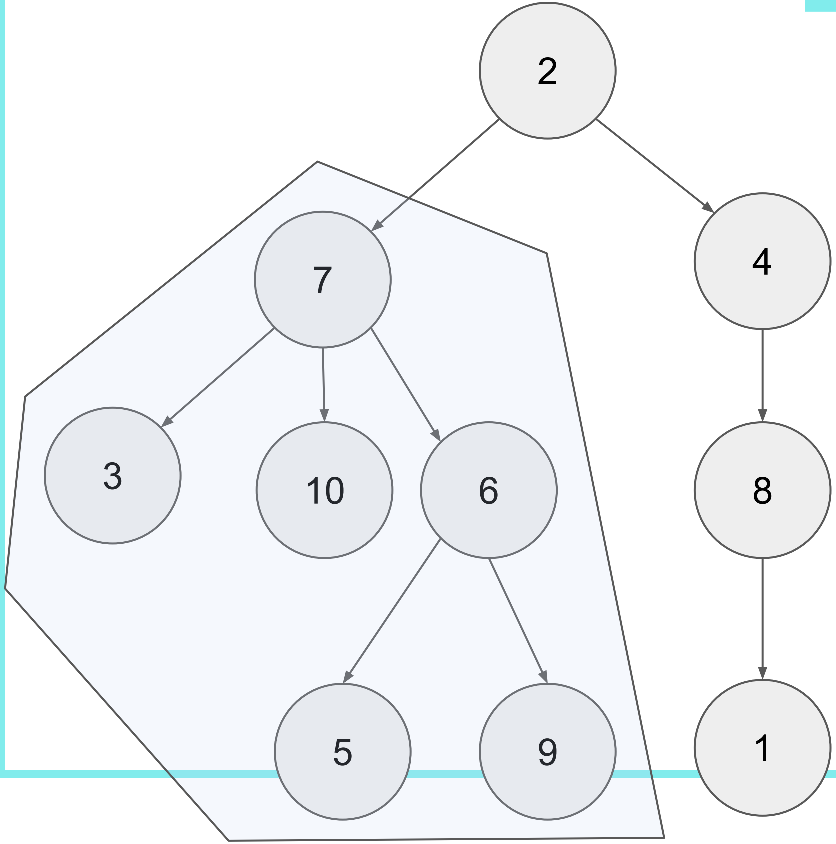
findSumOfTree



What actually happens...

```
int findSumOfTree(TreeNode<int>* root) {
 if (root->isLeaf()) {
   return root->getValue();
 }
 int sumSoFar = 0;
 for (auto n: root->getChildren()) {
   sumSoFar += findSumOfTree(n);
 }
 return sumSoFar + root->getValue();
}
```

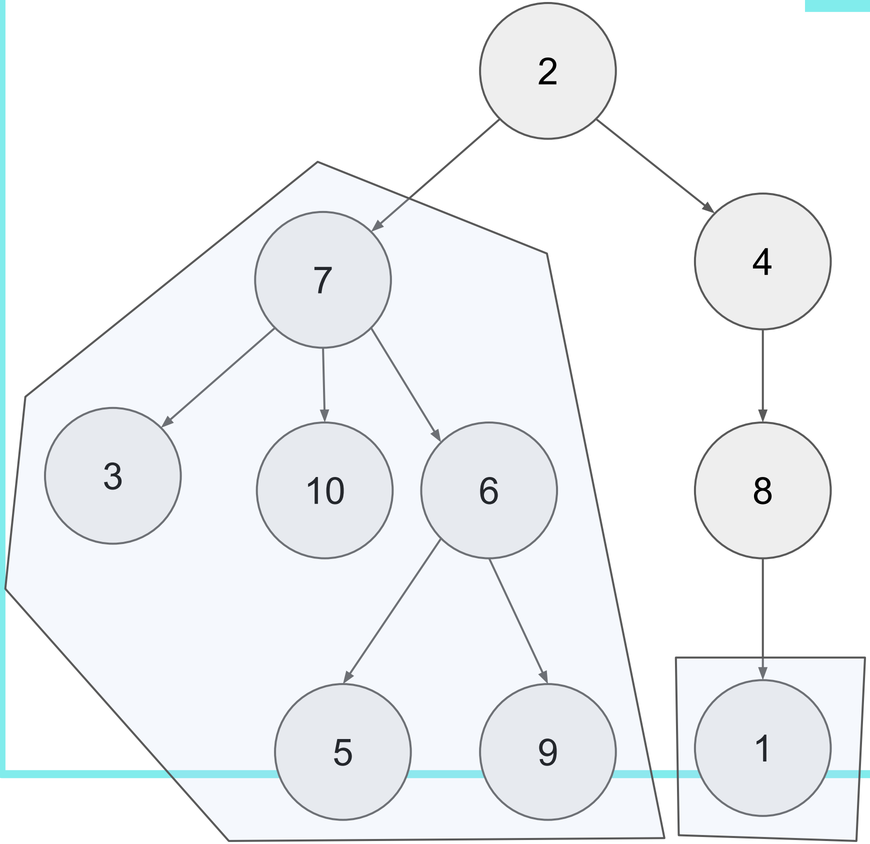
findSumOfTree



What actually happens...

```
int findSumOfTree(TreeNode<int>* root) {
 if (root->isLeaf()) {
   return root->getValue();
 }
 int sumSoFar = 0;
 for (auto n: root->getChildren()) {
   sumSoFar += findSumOfTree(n);
 }
 return sumSoFar + root->getValue();
}
```

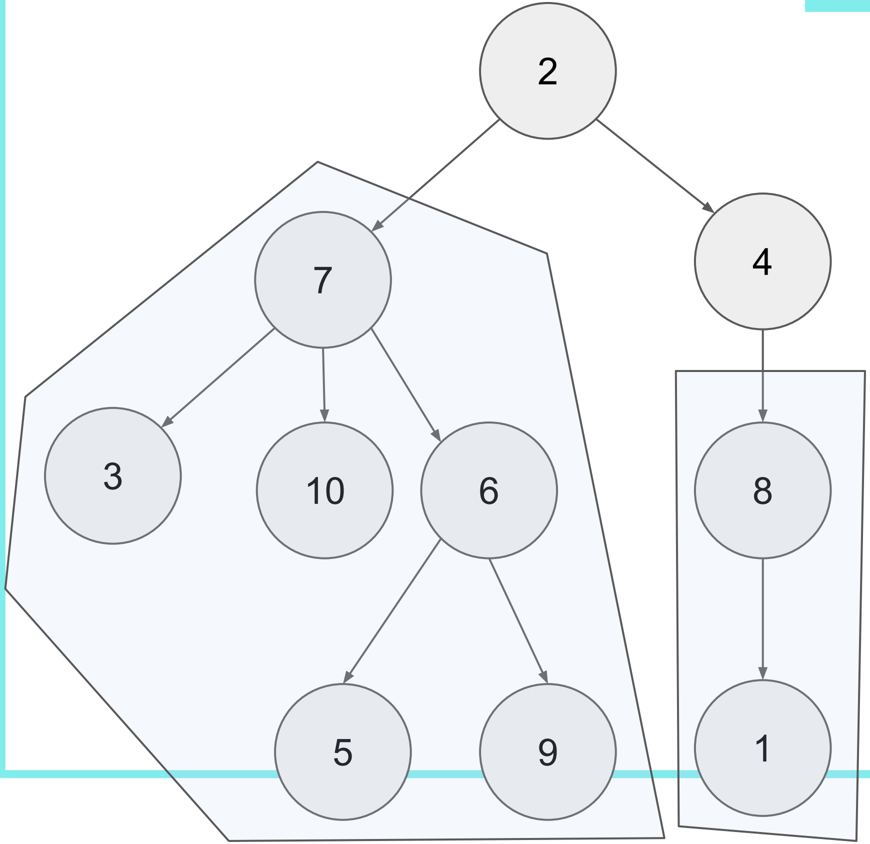
findSumOfTree



What actually happens...

```
int findSumOfTree(TreeNode<int>* root) {
 if (root->isLeaf()) {
   return root->getValue();
 }
 int sumSoFar = 0;
 for (auto n: root->getChildren()) {
   sumSoFar += findSumOfTree(n);
 }
 return sumSoFar + root->getValue();
}
```

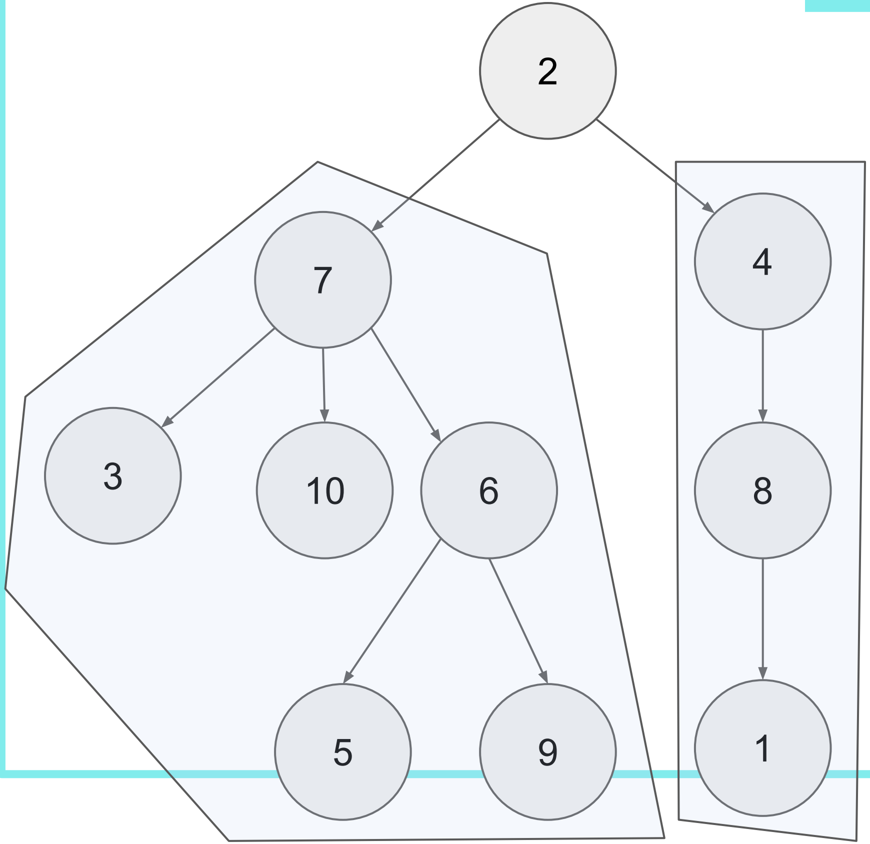
findSumOfTree



What actually happens...

```
int findSumOfTree(TreeNode<int>* root) {
 if (root->isLeaf()) {
   return root->getValue();
 }
 int sumSoFar = 0;
 for (auto n: root->getChildren()) {
   sumSoFar += findSumOfTree(n);
 }
 return sumSoFar + root->getValue();
}
```

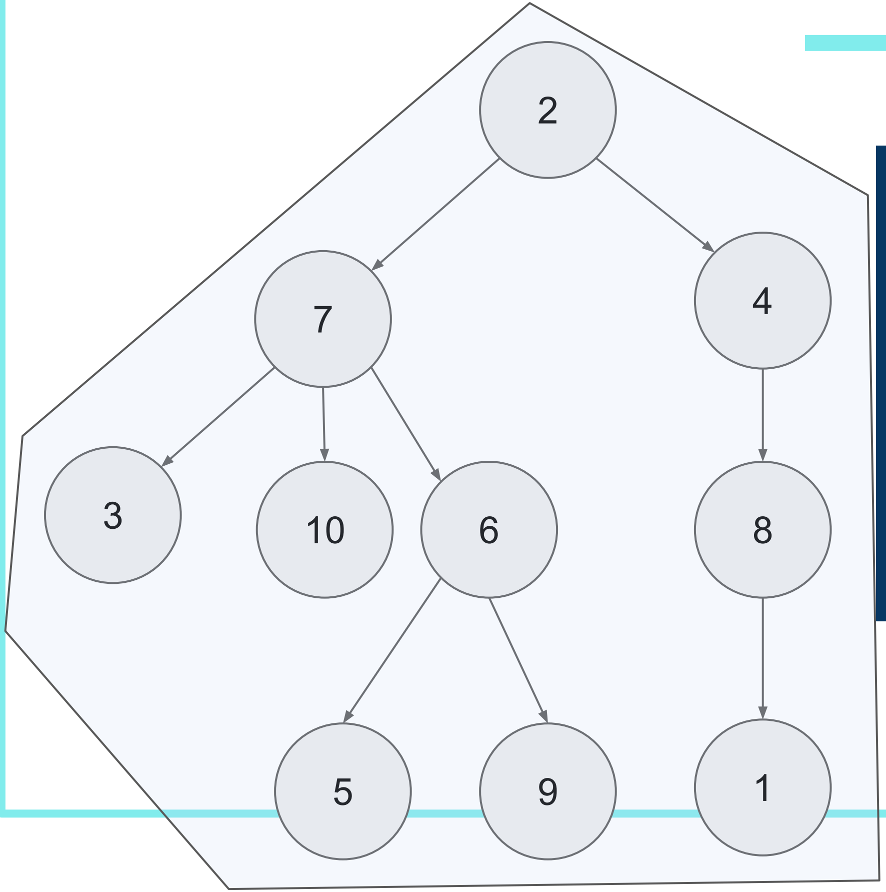
findSumOfTree



What actually happens...

```
int findSumOfTree(TreeNode<int>* root) {
 if (root->isLeaf()) {
   return root->getValue();
 }
 int sumSoFar = 0;
 for (auto n: root->getChildren()) {
   sumSoFar += findSumOfTree(n);
 }
 return sumSoFar + root->getValue();
}
```

findSumOfTree

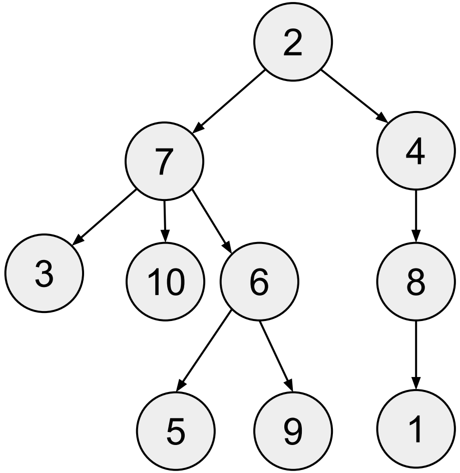


What actually happens...

```
int findSumOfTree(TreeNode<int>* root) {
 if (root->isLeaf()) {
   return root->getValue();
 }
 int sumSoFar = 0;
 for (auto n: root->getChildren()) {
   sumSoFar += findSumOfTree(n);
 }
 return sumSoFar + root->getValue();
}
```

findSumOfTree Time/Space Complexity

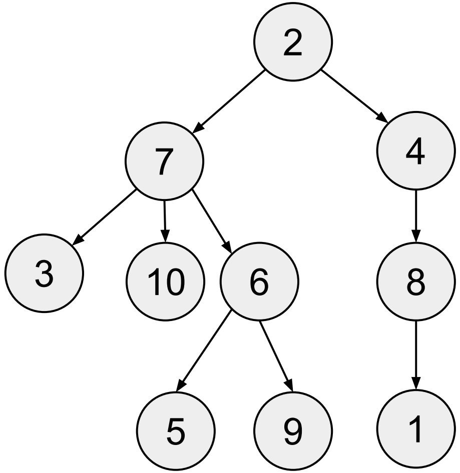
```
int findSumOfTree(TreeNode<int>* root) {
 if (root->isLeaf()) {
   return root->getValue();
 }
 int sumSoFar = 0;
 for (TreeNode<int>* n: root->getChildren()) {
   sumSoFar += findSumOfTree(n);
 }
 return sumSoFar + root->getValue();
}
```



- Base case
- Recursive call
- Building on recursive call

findSumOfTree Time/Space Complexity

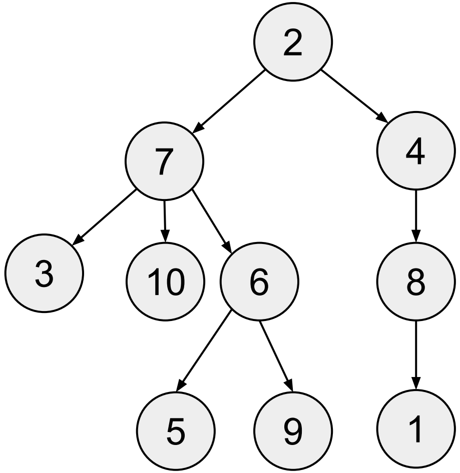
```
int findSumOfTree(TreeNode<int>* root) {
 if (root->isLeaf()) {
   return root->getValue();
 }
 int sumSoFar = 0;
 for (TreeNode<int>* n: root->getChildren()) {
   sumSoFar += findSumOfTree(n);
 }
 return sumSoFar + root->getValue();
}
```



- Best-case time complexity?
- Worst-case time complexity?
- Average-case time complexity?
- Space complexity?

findSumOfTree Time/Space Complexity

```
int findSumOfTree(TreeNode<int>* root) {
 if (root->isLeaf()) {
   return root->getValue();
 }
 int sumSoFar = 0;
 for (TreeNode<int>* n: root->getChildren()) {
   sumSoFar += findSumOfTree(n);
 }
 return sumSoFar + root->getValue();
}
```

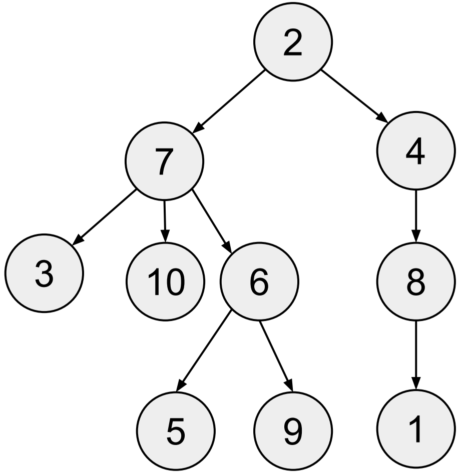


- Best-case time complexity?
- Worst-case time complexity?
- Average-case time complexity?
- Space complexity?

Hint: how many calls to findSumOfTree do we make, and how much work is being done in each call outside of the recursion?

findSumOfTree Time/Space Complexity

```
int findSumOfTree(TreeNode<int>* root) {
 if (root->isLeaf()) {
   return root->getValue();
 }
 int sumSoFar = 0;
 for (TreeNode<int>* n: root->getChildren()) {
   sumSoFar += findSumOfTree(n);
 }
 return sumSoFar + root->getValue();
}
```



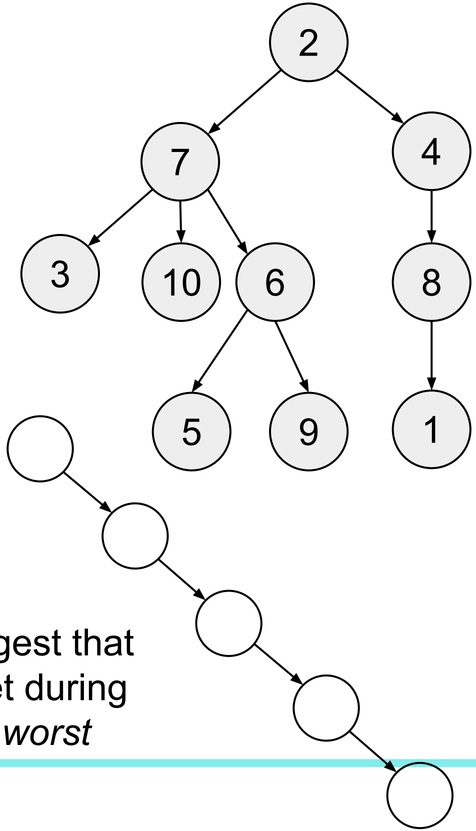
- Best-case time complexity? O(n)
- Worst-case time complexity? O(n)
- Average-case time complexity? O(n)
- Space complexity?

Hint: what is the largest that the call stack will get during the recursion *in the worst case?*

findSumOfTree Time/Space Complexity

```
int findSumOfTree(TreeNode<int>* root) {
 if (root->isLeaf()) {
   return root->getValue();
 }
 int sumSoFar = 0;
 for (TreeNode<int>* n: root->getChildren()) {
   sumSoFar += findSumOfTree(n);
 }
 return sumSoFar + root->getValue();
}
```

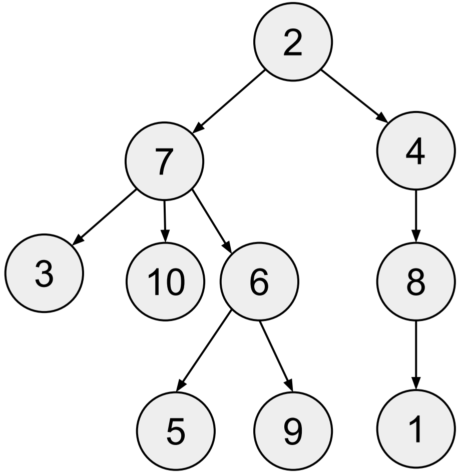
- Best-case time complexity? O(n)
- Worst-case time complexity? O(n)
- Average-case time complexity? O(n)
- Space complexity?



Hint: what is the largest that the call stack will get during the recursion *in the worst case?*

findSumOfTree Time/Space Complexity

```
int findSumOfTree(TreeNode<int>* root) {
 if (root->isLeaf()) {
   return root->getValue();
 }
 int sumSoFar = 0;
 for (TreeNode<int>* n: root->getChildren()) {
   sumSoFar += findSumOfTree(n);
 }
 return sumSoFar + root->getValue();
}
```



- Best-case time complexity? O(n)
- Worst-case time complexity? O(n)
- Average-case time complexity? O(n)
- Space complexity? O(n)

Big Questions!

- What are trees again?
- How do we represent trees in C++?
- What are some algorithms with trees? (recursion!)
- How can I practice?

Great Question!

With an in-class activity!

With an in-class activity!

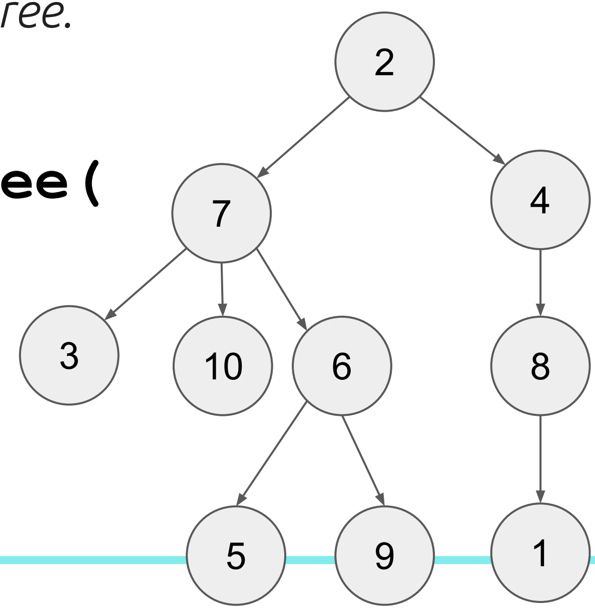
Head over to course website or Blackboard

findMaxOfTree

Instructions: Course Website or Blackboard -> Lectures -> Lecture 9

Write an algorithm that takes in a tree of ints, and returns the max of all the values within the tree.

findMaxOfTree(



) outputs 10

Motivation for Binary Search Trees

	Sorted Arrays	Linked Lists	(balanced) Binary Search Trees
Search	O(log(n))	O(n)	O(log(n))
Delete	O(n)	O(n)	O(log(n))
Insert	O(n)	O(1)	O(log(n))

How was the pace today?