Binary Search Trees

Elyse Cornwall

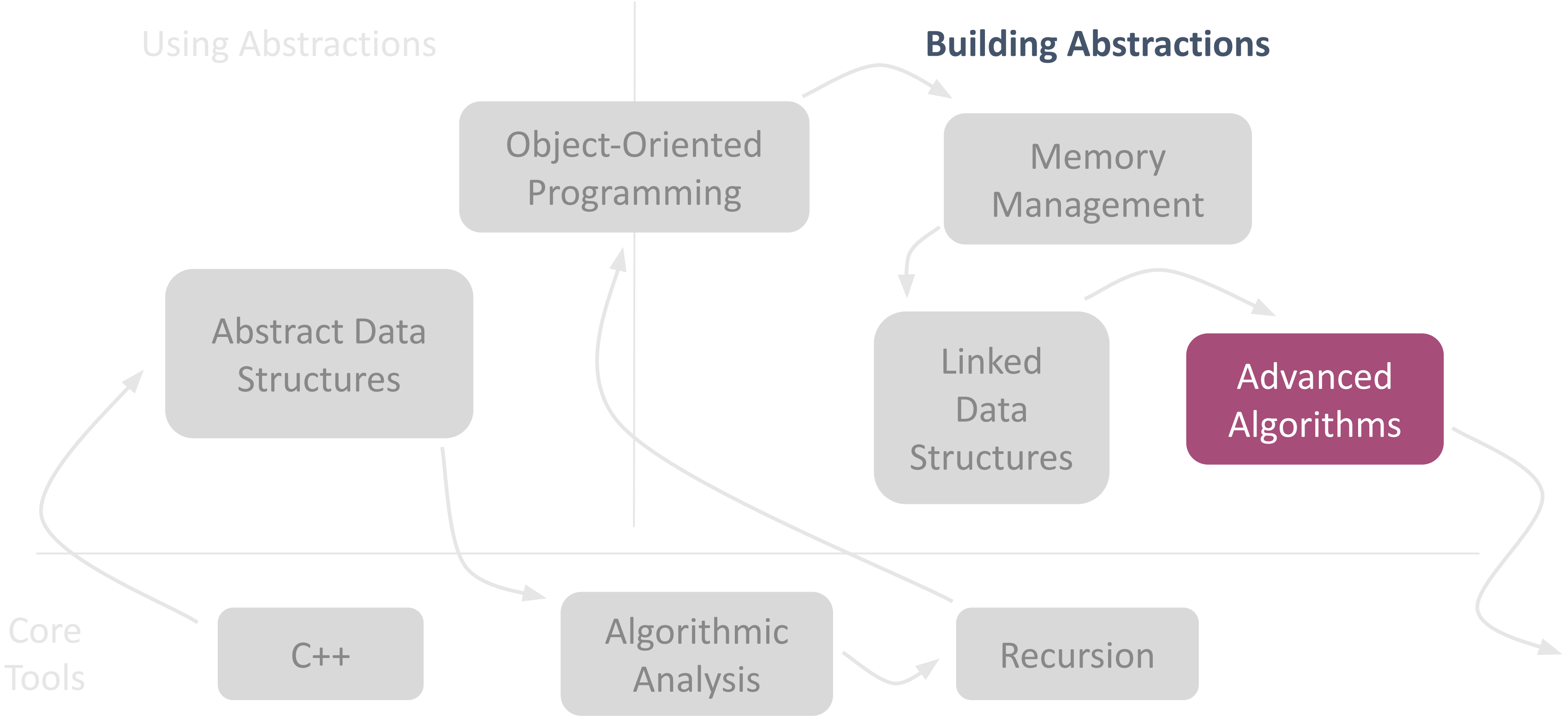
August 7, 2023

Contributions made from previous CS106B Instructors

Announcements

- This week is our final section
- Exam next Friday (8/18) from 3:30-6:30pm
    - Final exam info will be published this afternoon
    - Final review session next Tuesday in class

Roadmap

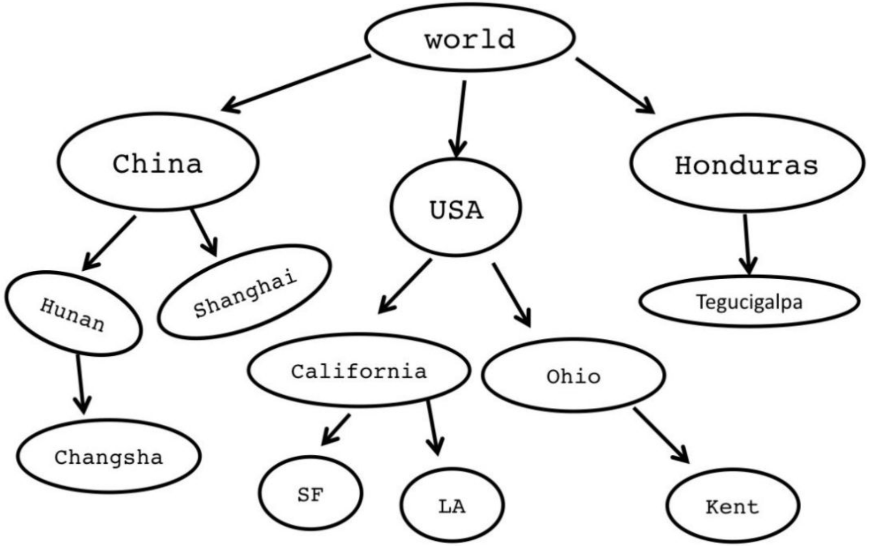


Using Abstractions
Building Abstractions
Object-Oriented Programming
Memory Management
Abstract Data Structures
Linked Data Structures
Advanced Algorithms
Core Tools
C++
Algorithmic Analysis
Recursion

Recap: Trees

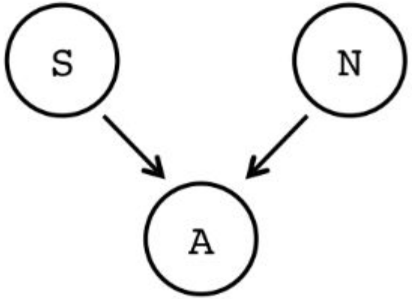
Uses

- Trees are useful in other ways besides visualizing recursion and modeling priority
  - Describe hierarchies

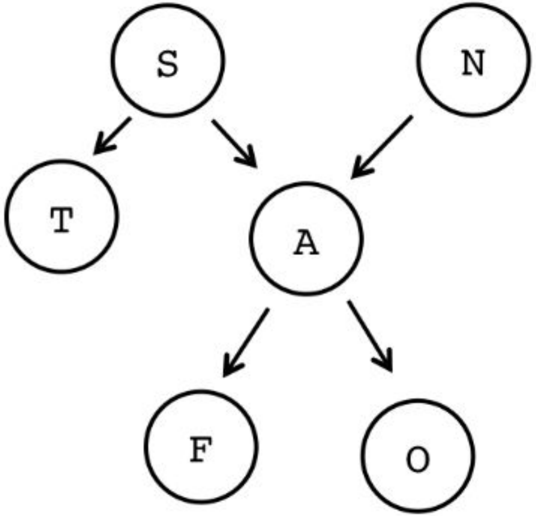


Tree Properties

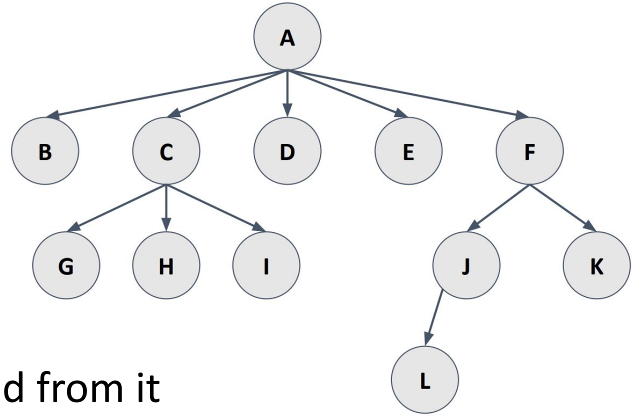
- Any node in a tree can only have one parent



Not trees!



Tree Terminology



Types of nodes

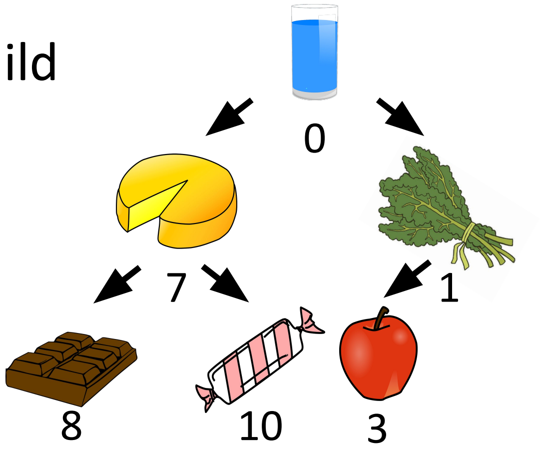
- The **root** node defines the "top" of the tree
- Every node has 0 or more **children** nodes descended from it
- Nodes with no children are called **leaf** nodes
- Every node in a tree has exactly one **parent** node (except for the root node)

Terminology for quantifying trees

- The **length** of a path between two nodes is the number of edges between them
- The **depth** of a node is the length of the path from the root to that node
- The **height** of a tree is the number of nodes in the longest path through the tree (i.e. the number of levels in the tree)

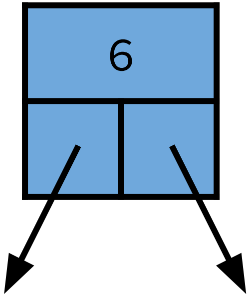
Binary Trees

- Most common trees in CS
  - We’ve seen these before, Binary Heaps!
- Every node has either 0, 1, or 2 children
- Children are referred to as left child and right child



Building Binary Trees

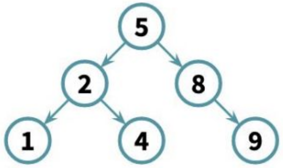
- A binary tree is composed of nodes
- Each node is a struct that contains:
  - A piece of data (like an int, or string)
  - A pointer to the left child
  - A pointer to the right child




```
struct TreeNode {
    int data;
    TreeNode* left;
    TreeNode* right;
};
```

Tree Traversal Recap

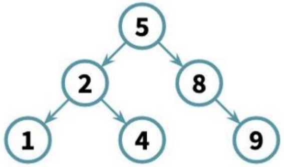
Pre-order



do something (aka cout)
traverse left subtree
traverse right subtree

5 2 1 4 8 9

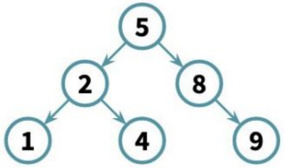
In-order



traverse left subtree
do something (aka cout)
traverse right subtree

1 2 4 5 8 9

Post-order



traverse left subtree
traverse right subtree
do something (aka cout)

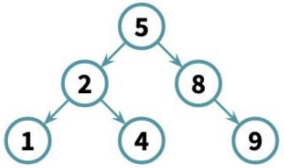
1 4 2 9 8 5

Demo: Freeing a Tree

Traverse a tree and free its nodes

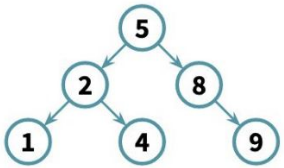
👥 Which Method Should We Use?

Pre-order



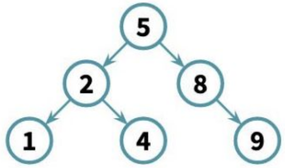
do something **(aka delete)**
traverse left subtree
traverse right subtree

In-order



traverse left subtree
do something **(aka delete)**
traverse right subtree

Post-order

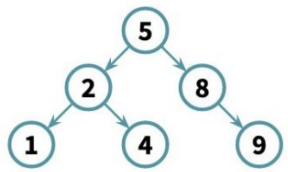


traverse left subtree
traverse right subtree
do something **(aka delete)**

Which Method Should We Use?

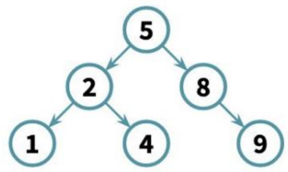
If we delete a node before deleting its children, we'll lose access to its children

Pre-order



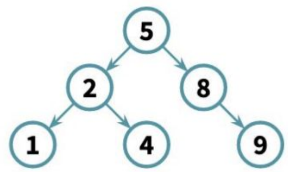
do something **(aka delete)**
traverse left subtree
traverse right subtree

In-order



traverse left subtree
do something **(aka delete)**
traverse right subtree

Post-order



traverse left subtree
traverse right subtree
do something **(aka delete)**

Let’s code it up!

Traverse a tree and free its nodes

Solution Code - Freeing a Tree

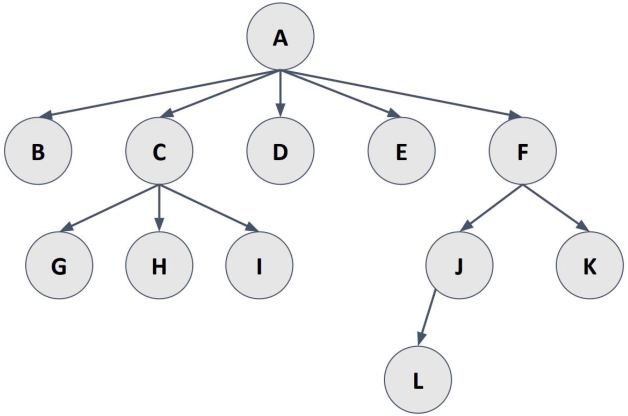
```
void freeTree(TreeNode* node) {
    if (node == nullptr) {
        return;
    }
    freeTree(node->left);
    freeTree(node->right);
    delete node;
}
```

Binary Search Trees

Trees optimized for binary search!

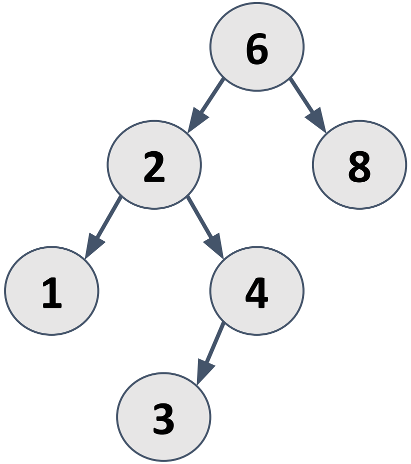
Why Trees?

- The distance from each node in a tree to root is small, even if there are many elements
- How can we take advantage of trees to structure and efficiently manipulate data?



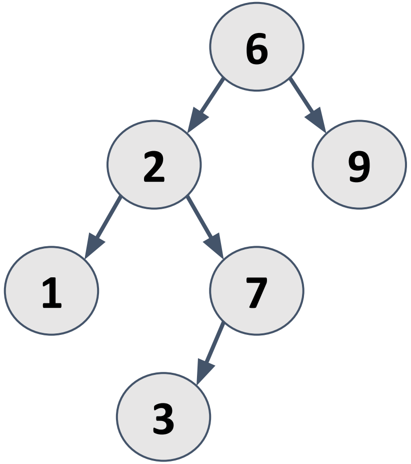
Binary Search Trees (BSTs)

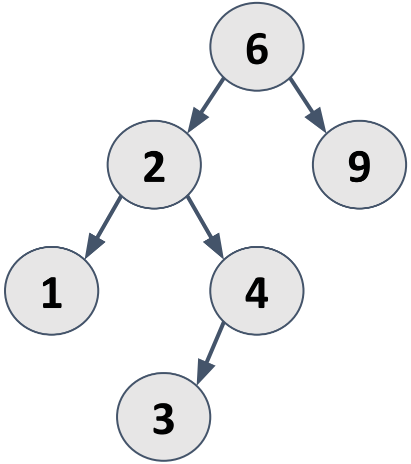
1. Binary tree (each node has 0, 1, or 2 children)
2. For a node with value X:
   a. All nodes in its left subtree must be less than X
   b. All nodes in its right subtree must be greater than X



Spot the Valid BST

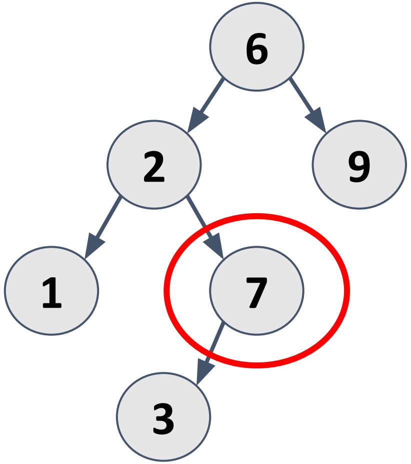
1. Binary tree (each node has 0, 1, or 2 children)
2. For a node with value X:
    a. All nodes in its left subtree must be less than X
    b. All nodes in its right subtree must be greater than X



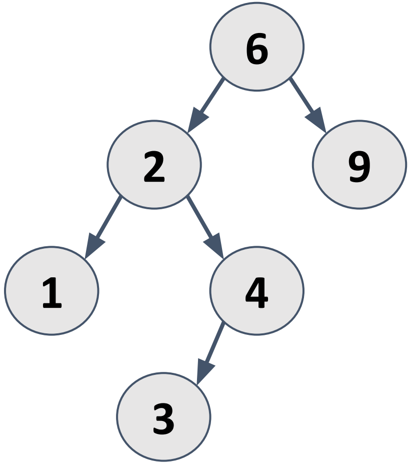


Spot the Valid BST

There's a node in the left subtree of 6 that is greater than 6

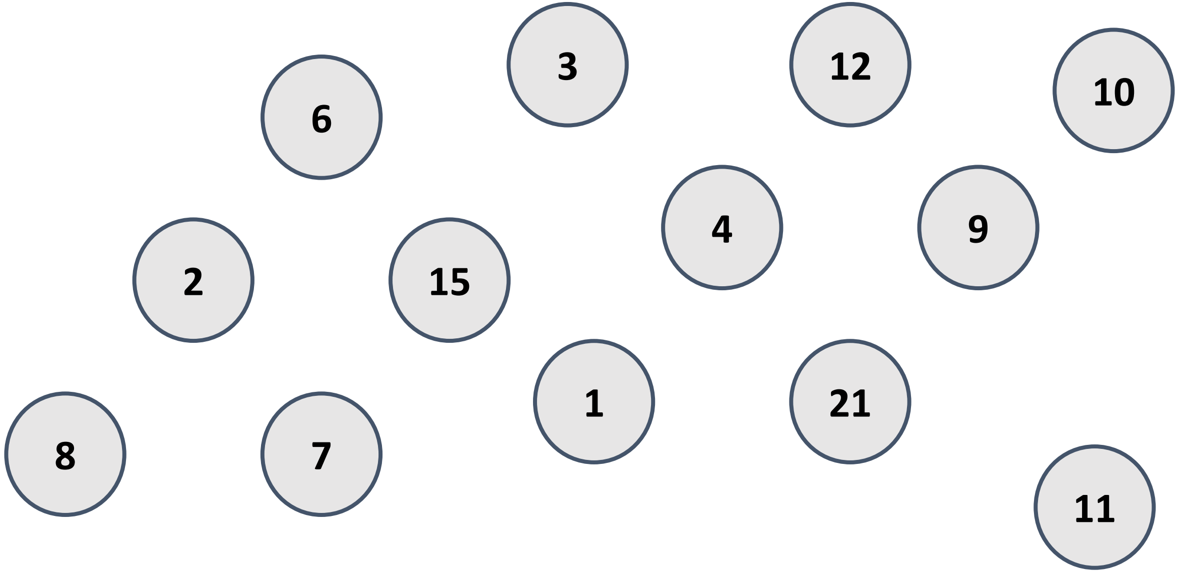


YOU'RE VALID 😎



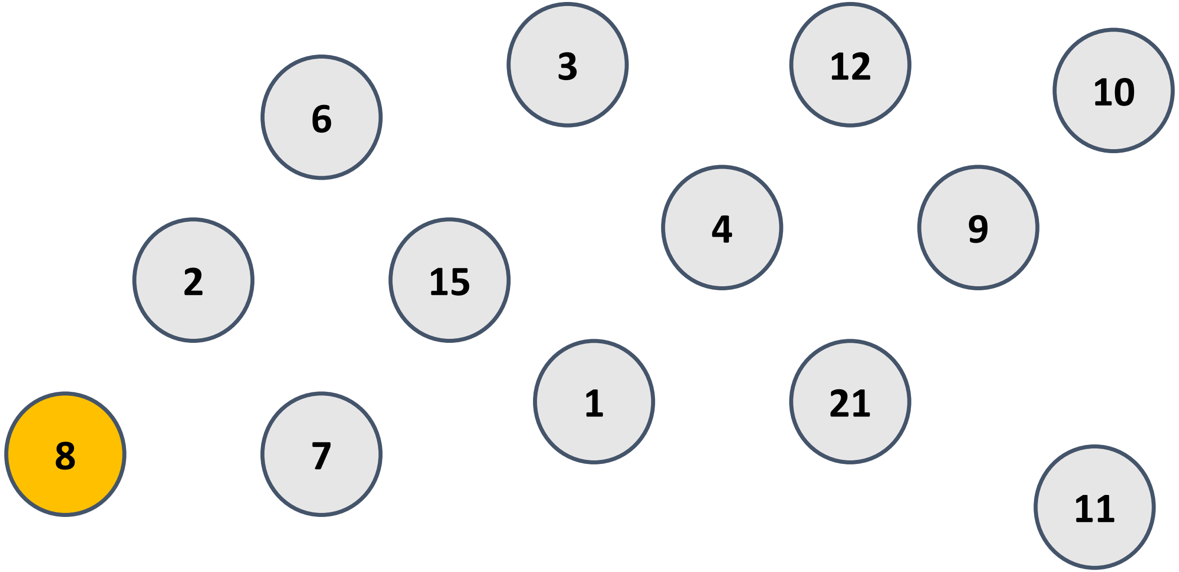
Turning Data into a BST

Let's say we wanted to store the following numbers in a BST:



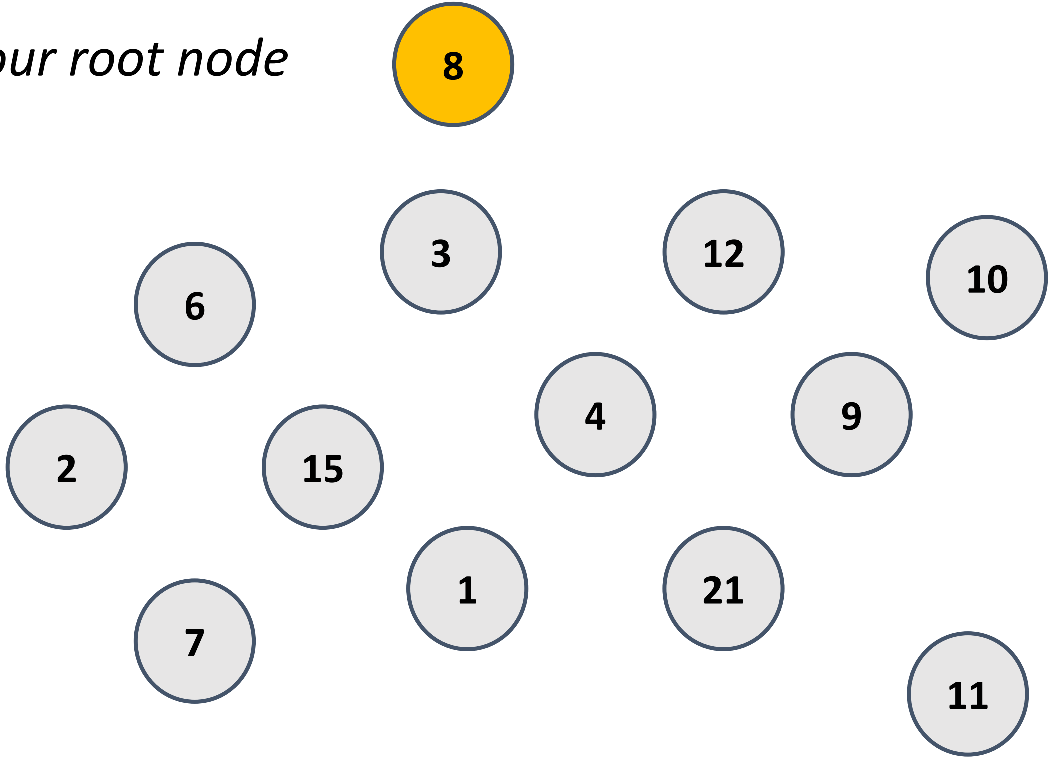
Turning Data into a BST

To build a BST, we choose the median element



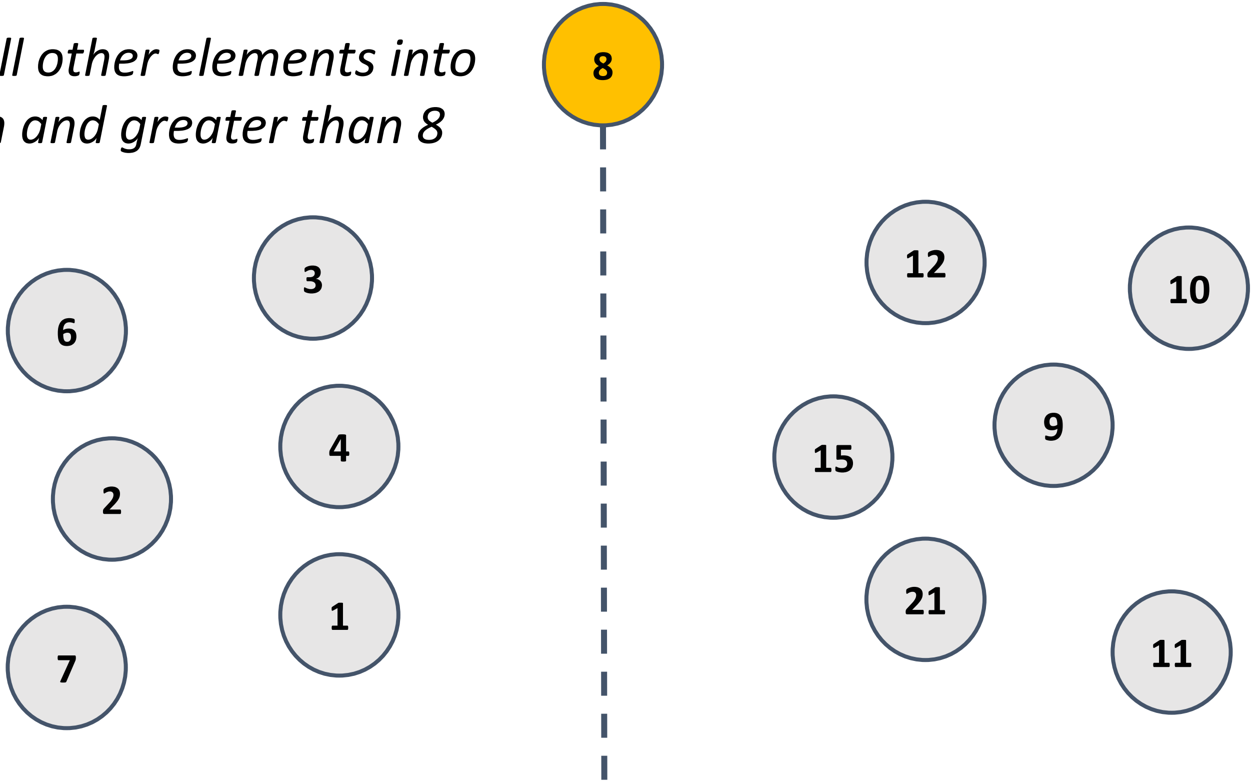
Turning Data into a BST

This becomes our root node



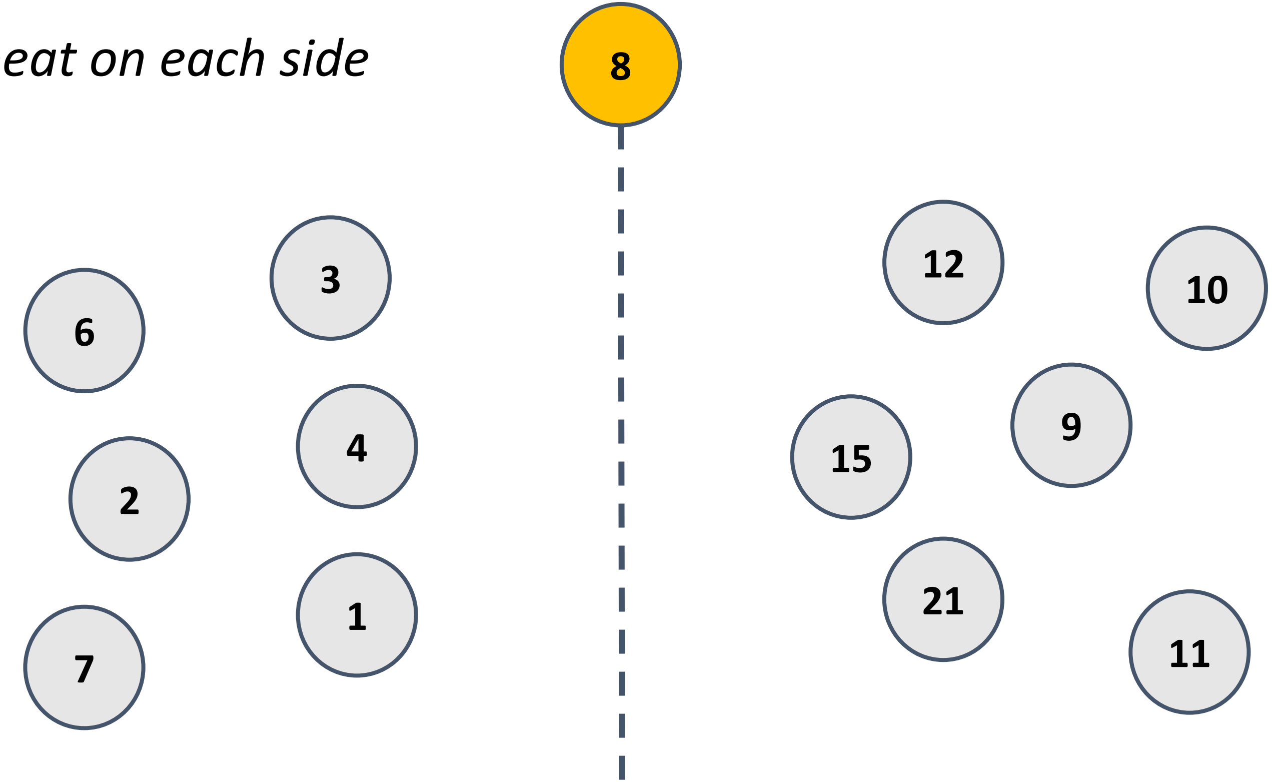
Turning Data into a BST

We split all other elements into less than and greater than 8



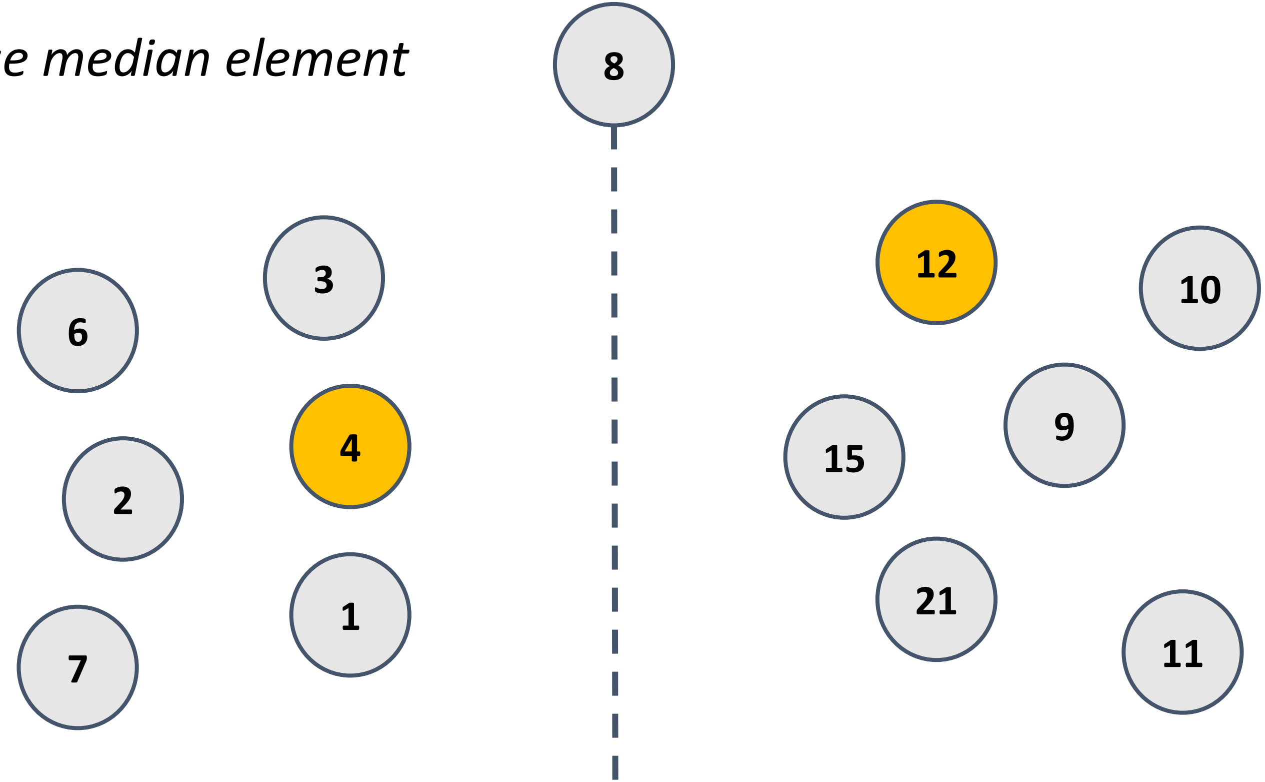
Turning Data into a BST

Repeat on each side



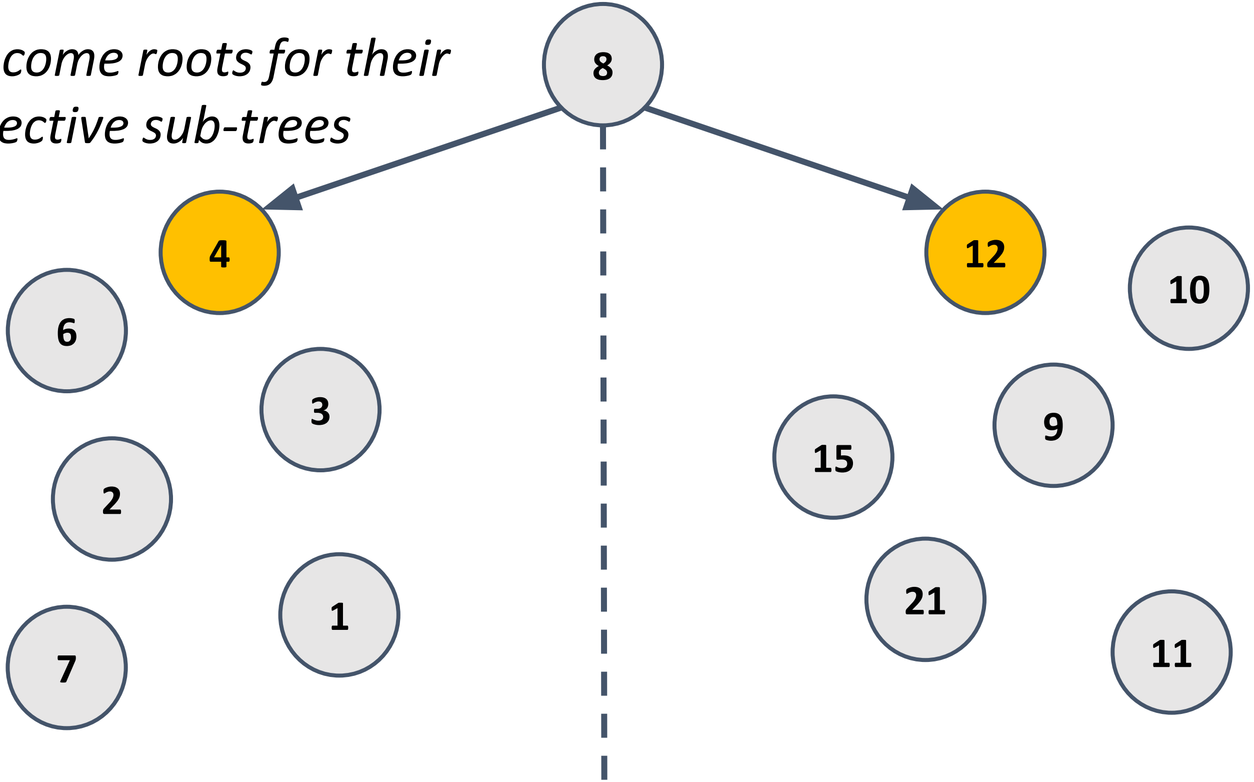
Turning Data into a BST

Choose median element



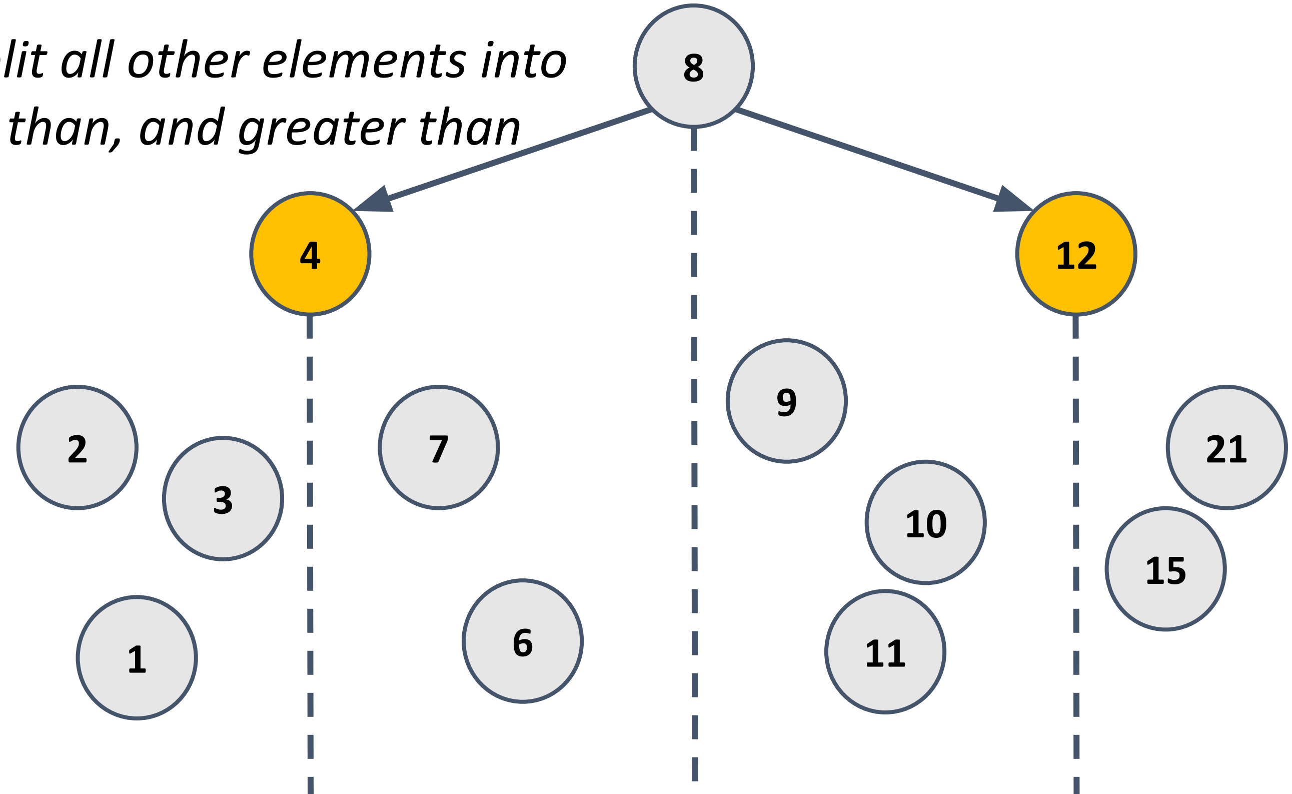
Turning Data into a BST

These become roots for their respective sub-trees

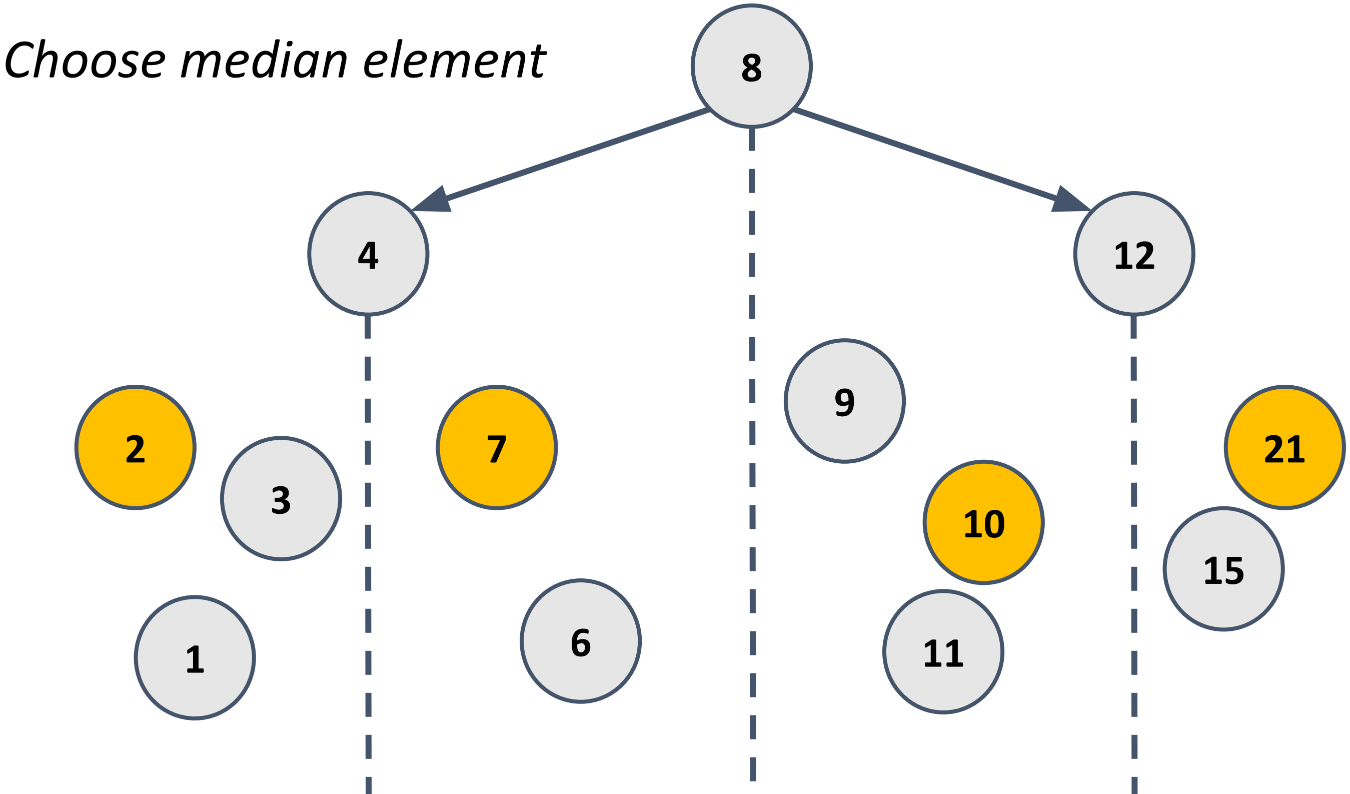


Turning Data into a BST

We split all other elements into less than, and greater than

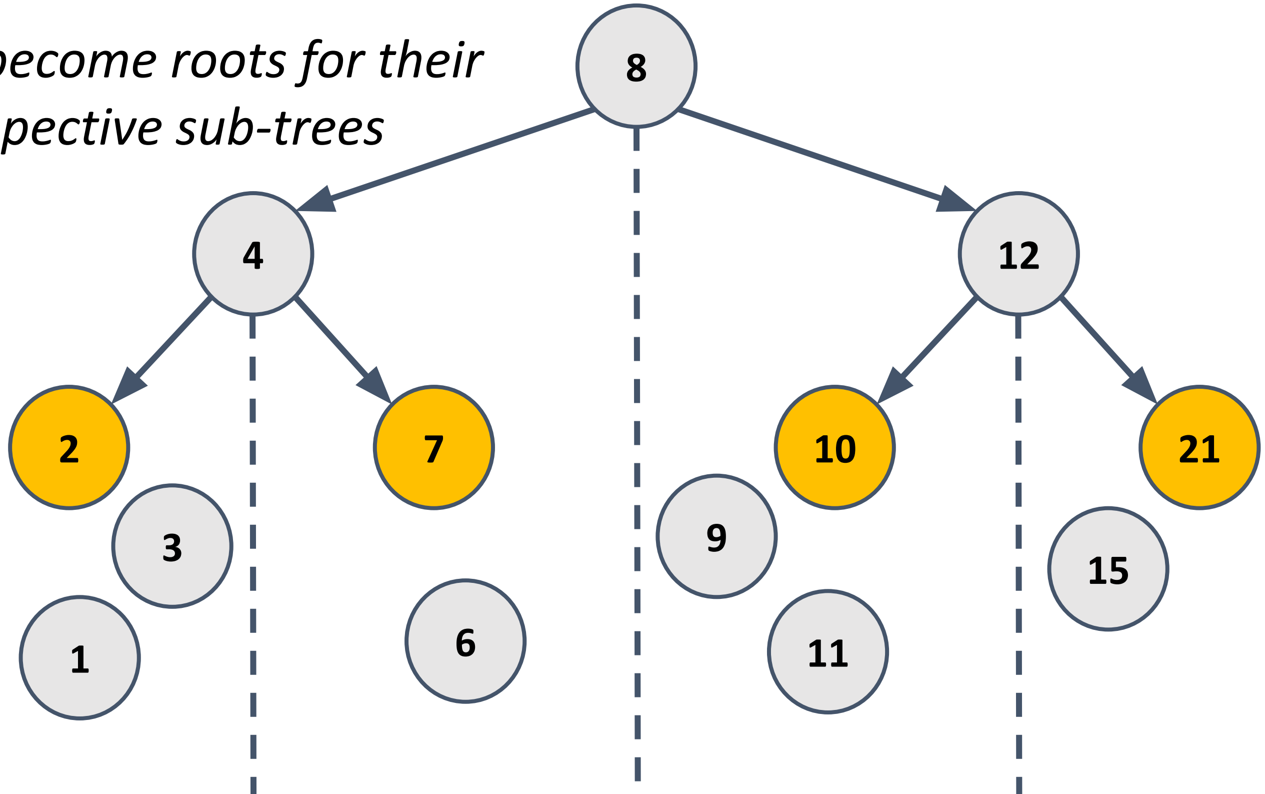


Turning Data into a BST



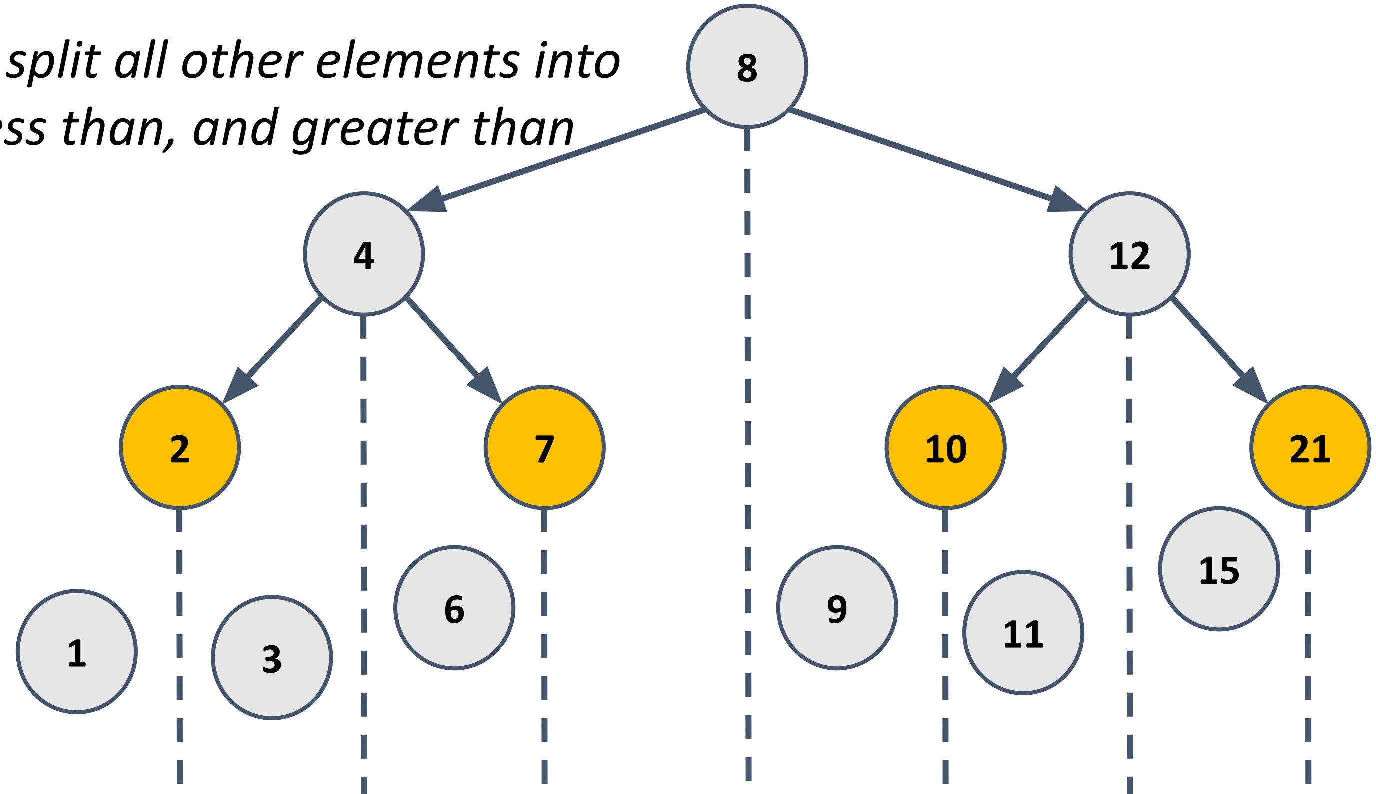
Turning Data into a BST

These become roots for their respective sub-trees



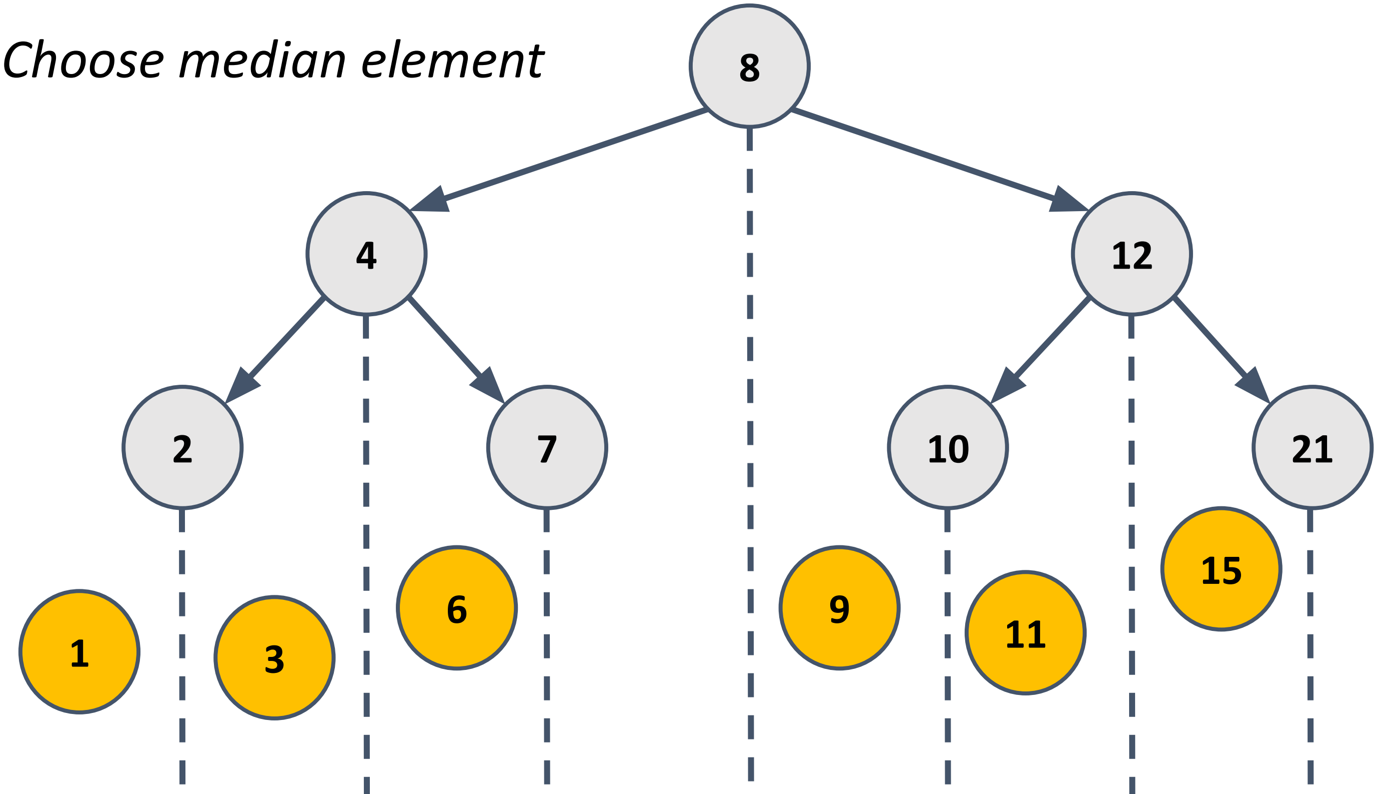
Turning Data into a BST

We split all other elements into less than, and greater than



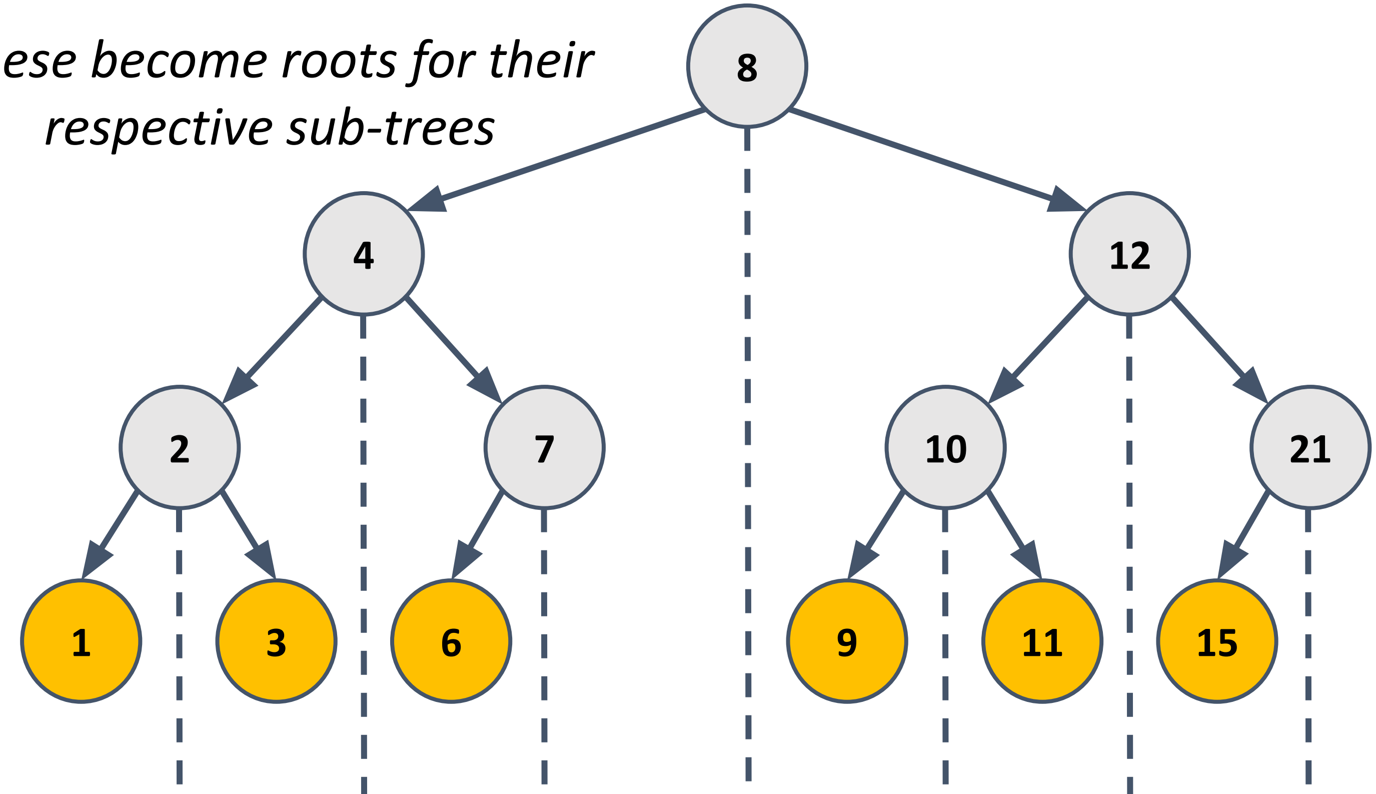
Turning Data into a BST

Choose median element

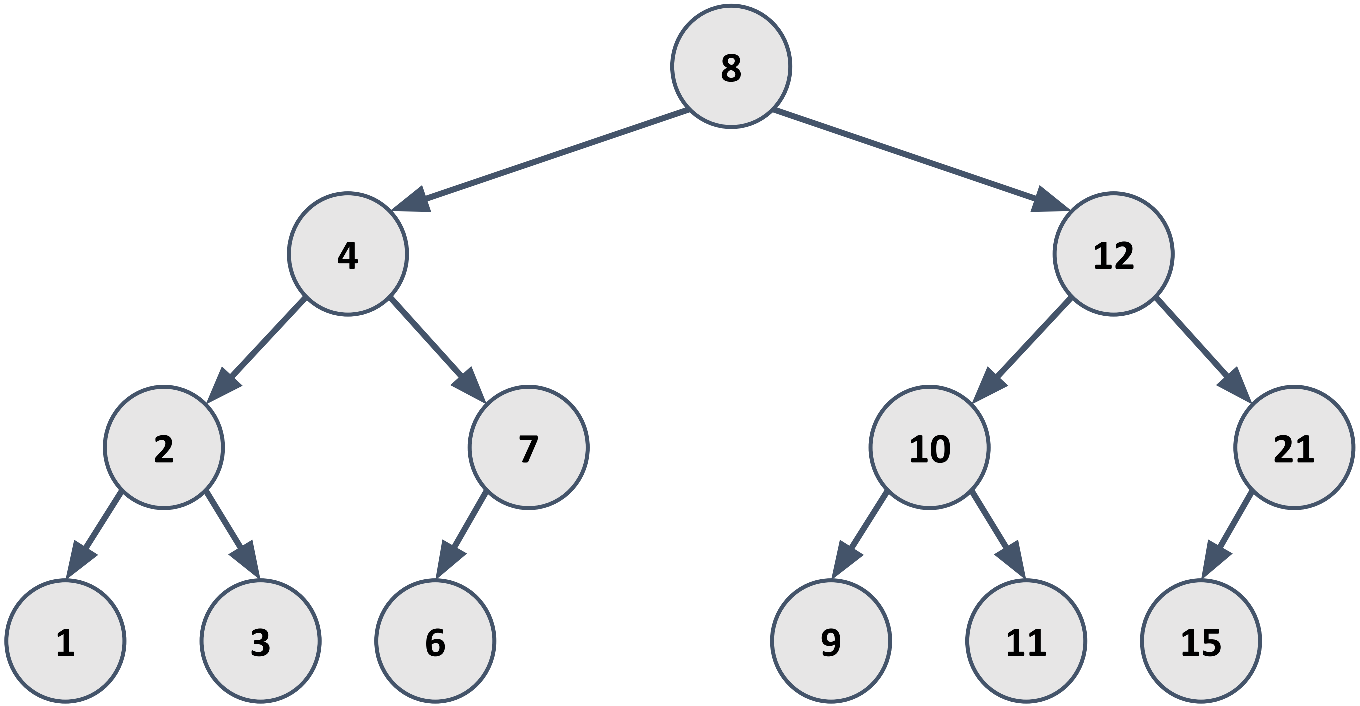


Turning Data into a BST

These become roots for their respective sub-trees



Turning Data into a BST

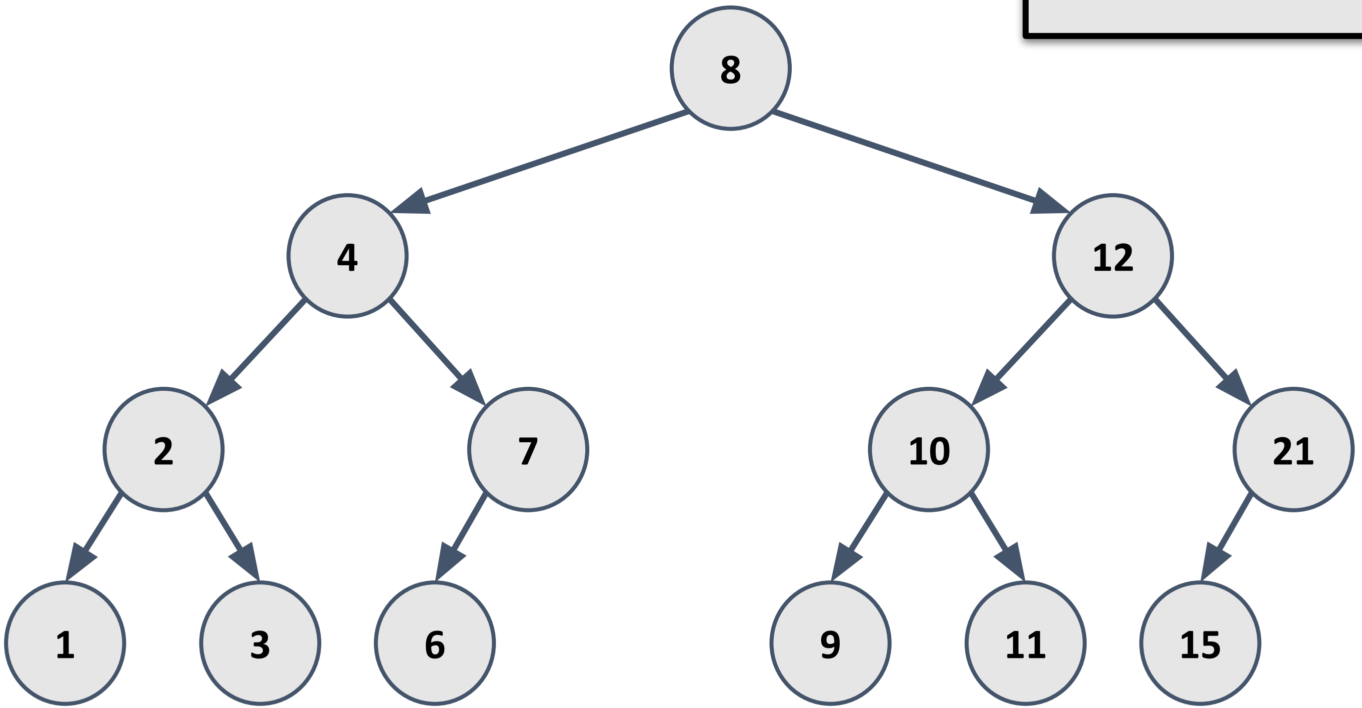


BST Lookups

These data structures are designed for fast lookups!

BST Lookups

Is 11 in this BST?

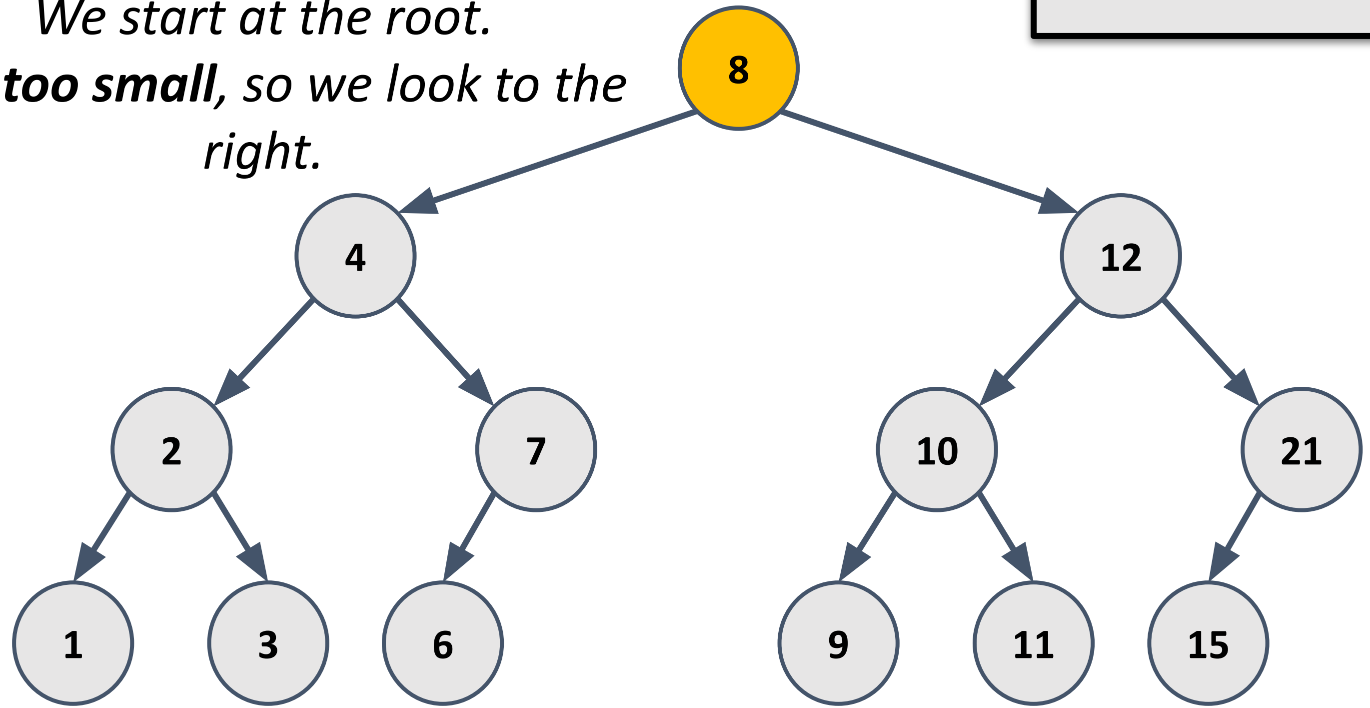


BST Lookups

Is 11 in this BST?

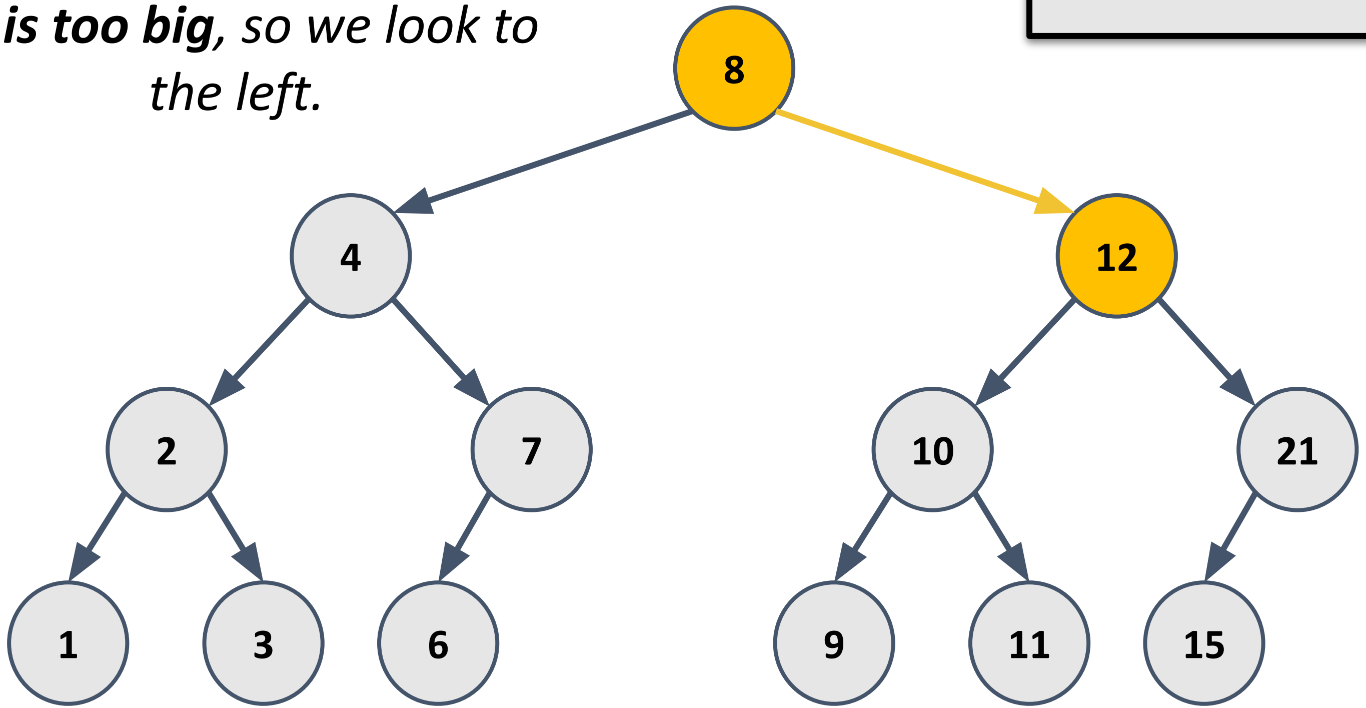
We start at the root.

***8 is too small**, so we look to the right.*



BST Lookups

***12 is too big**, so we look to the left.*

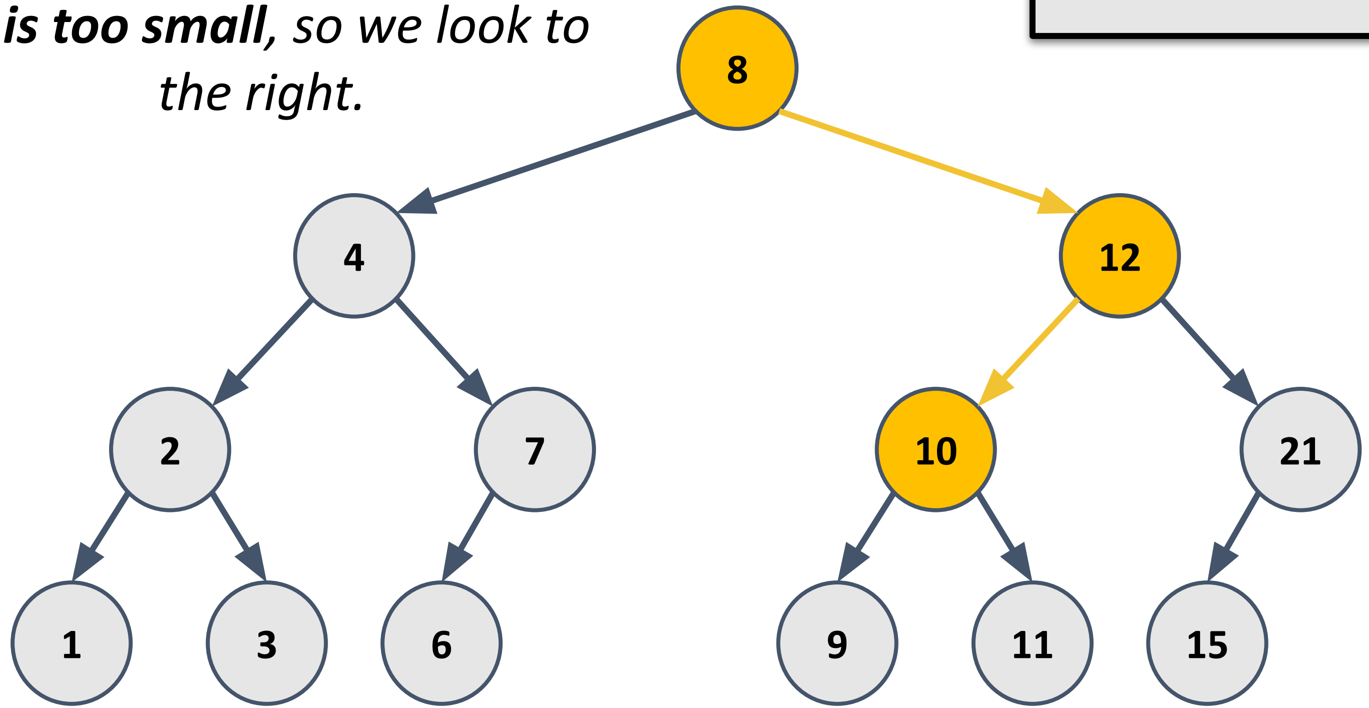


Is 11 in this BST?

BST Lookups

10 is too small*, so we look to the right.*

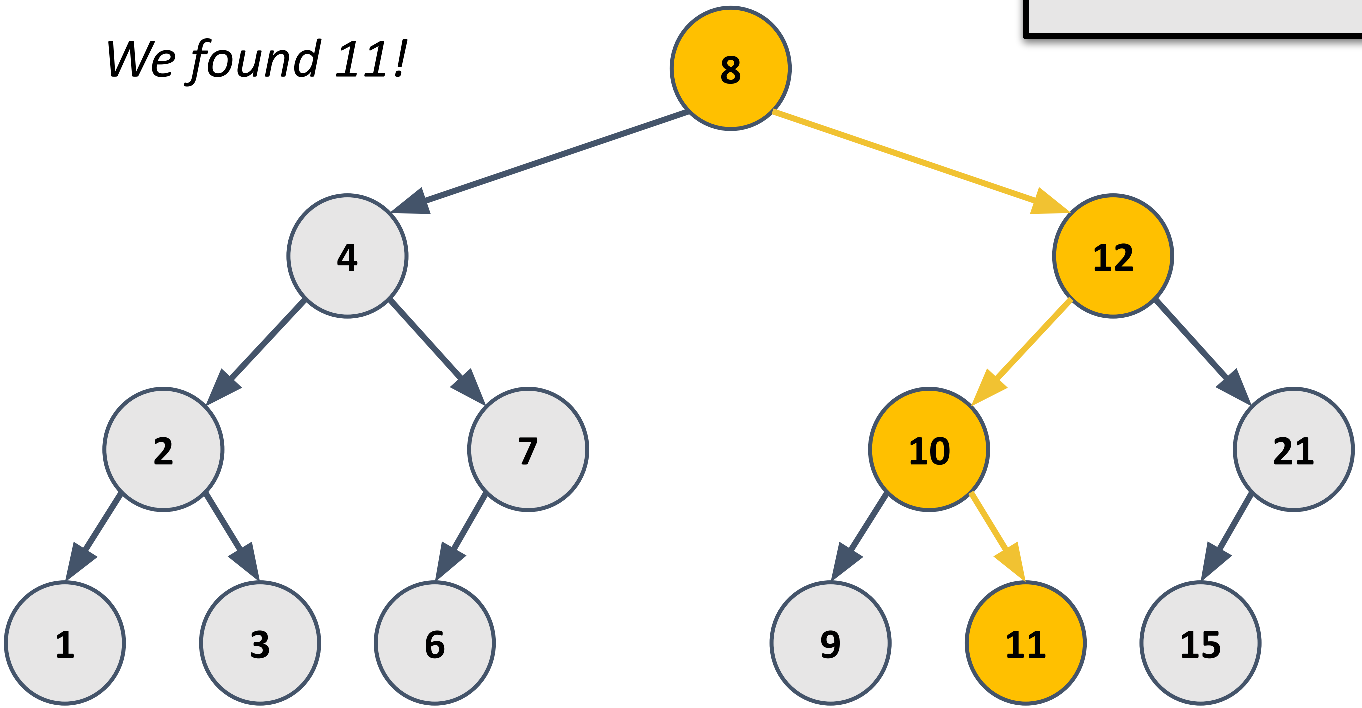
Is 11 in this BST?



BST Lookups

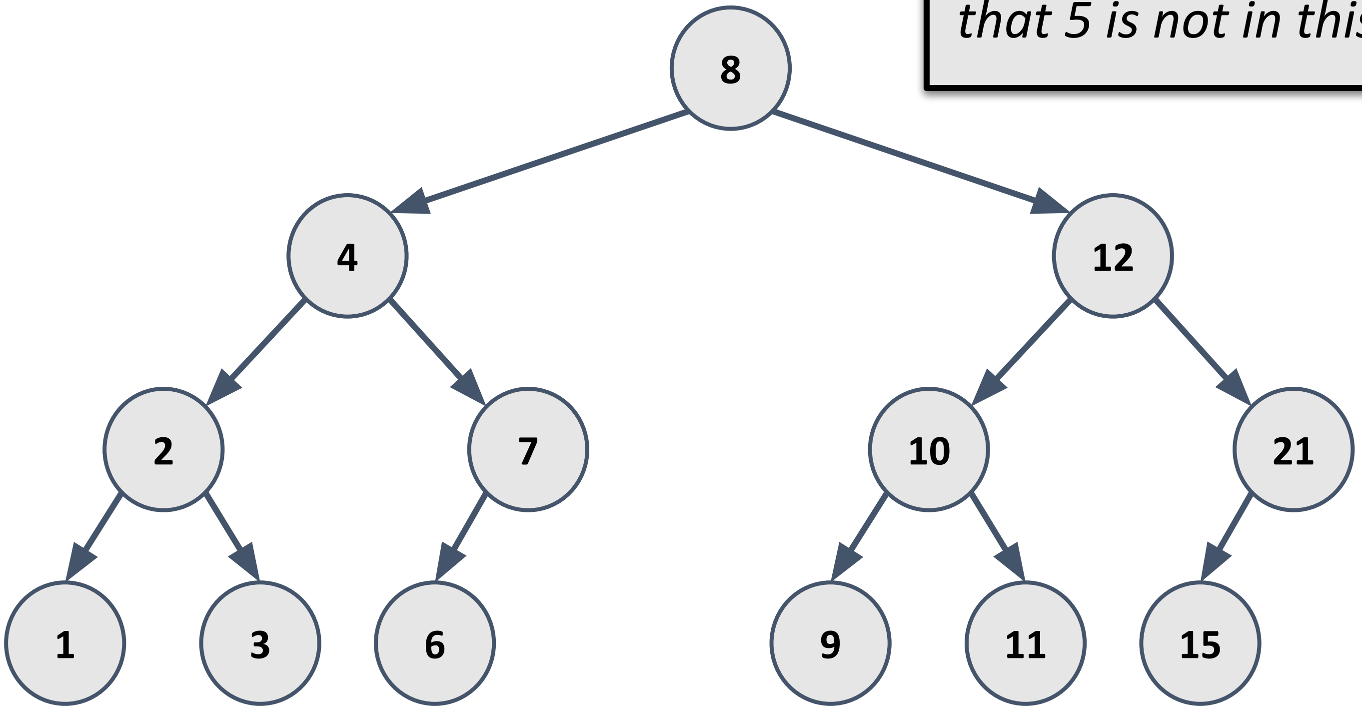
Is 11 in this BST?

We found 11!



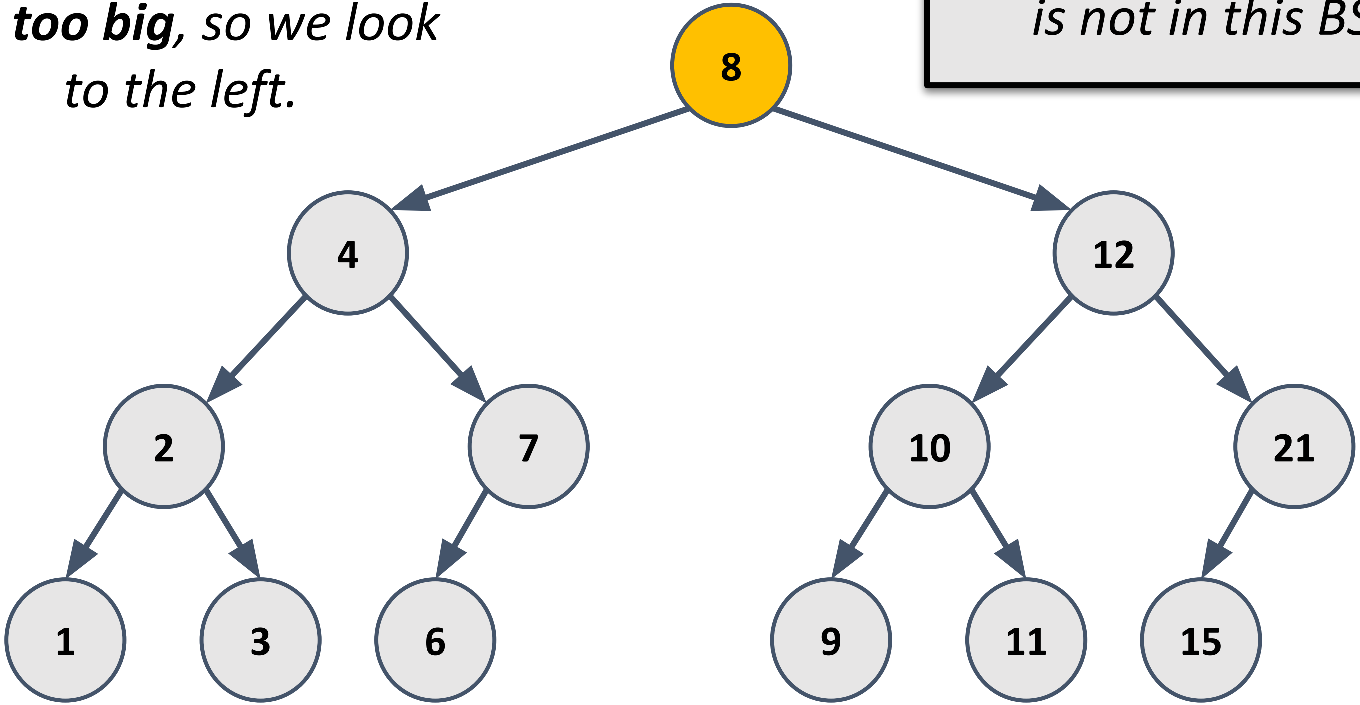
BST Lookups

👥 *How do we know that 5 is not in this BST?*



BST Lookups

***8 is too big**, so we look to the left.*

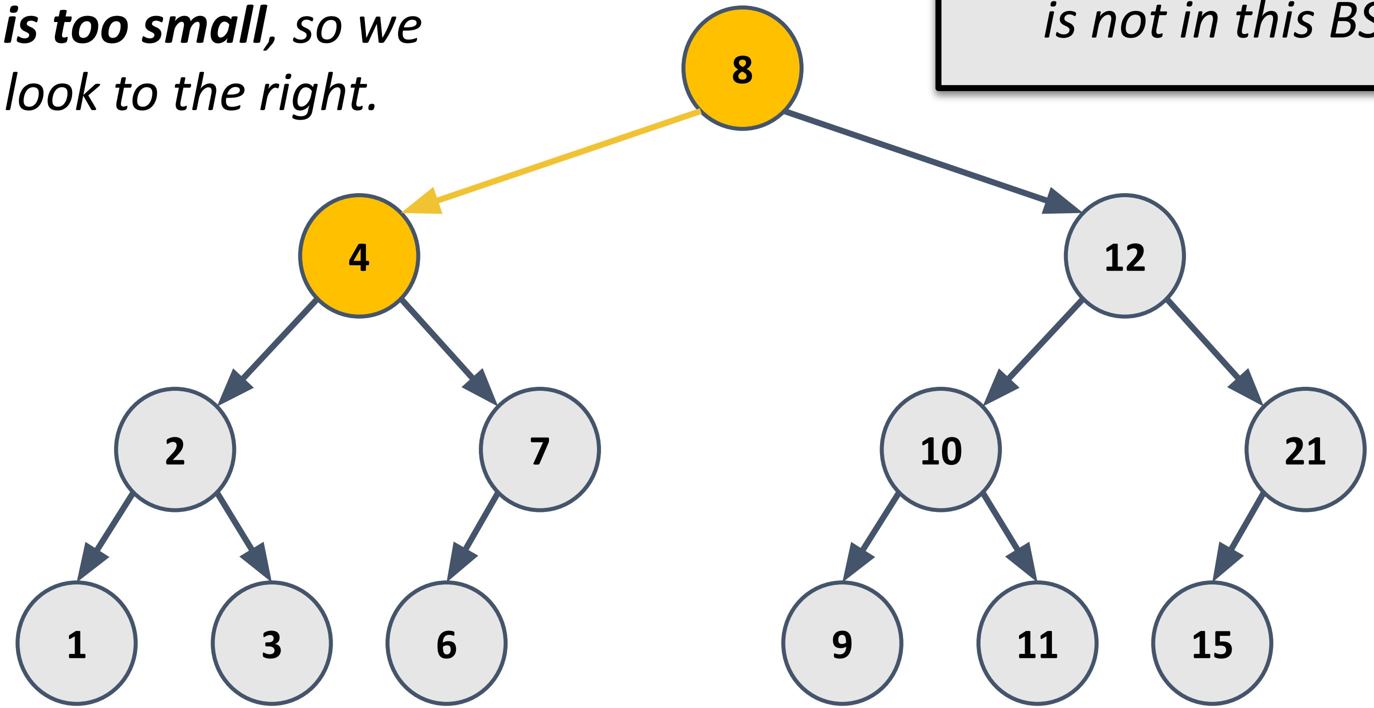


How do we know that 5 is not in this BST?

BST Lookups

***4 is too small**, so we look to the right.*

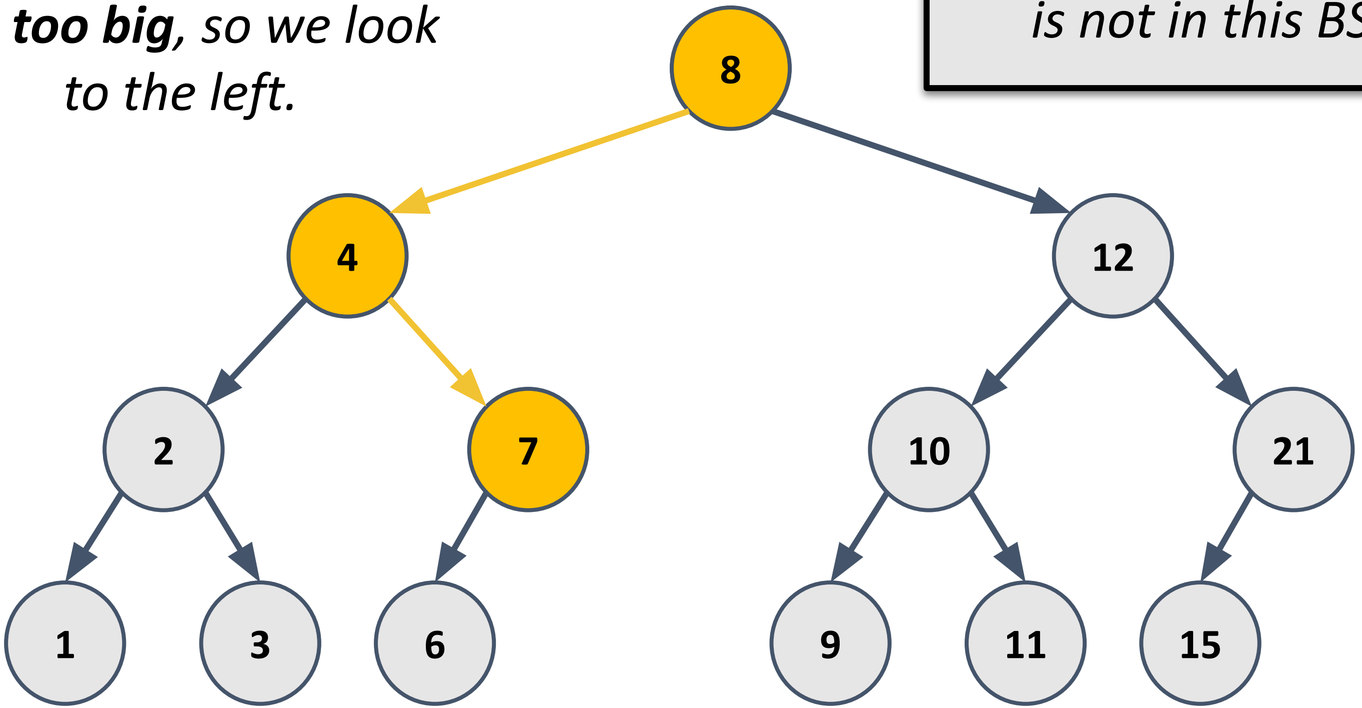
How do we know that 5 is not in this BST?



BST Lookups

***7 is too big**, so we look to the left.*

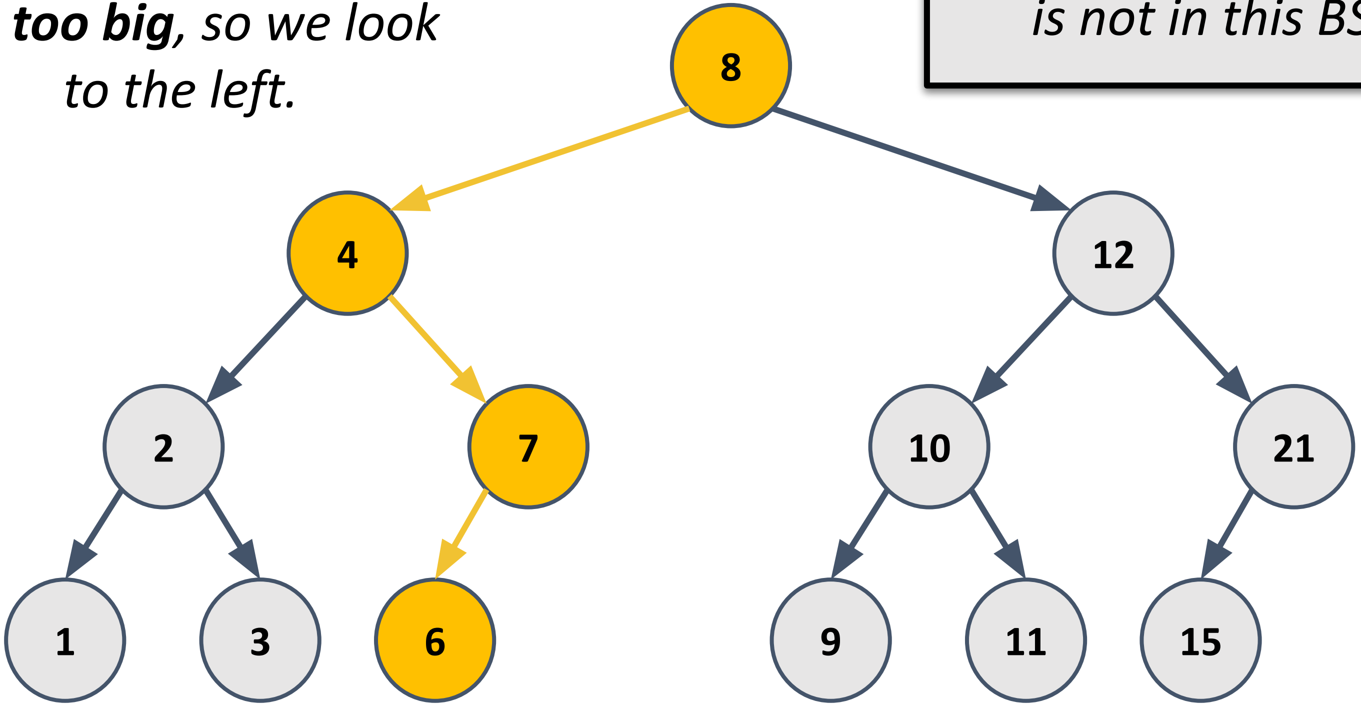
How do we know that 5 is not in this BST?



BST Lookups

***6 is too big**, so we look to the left.*

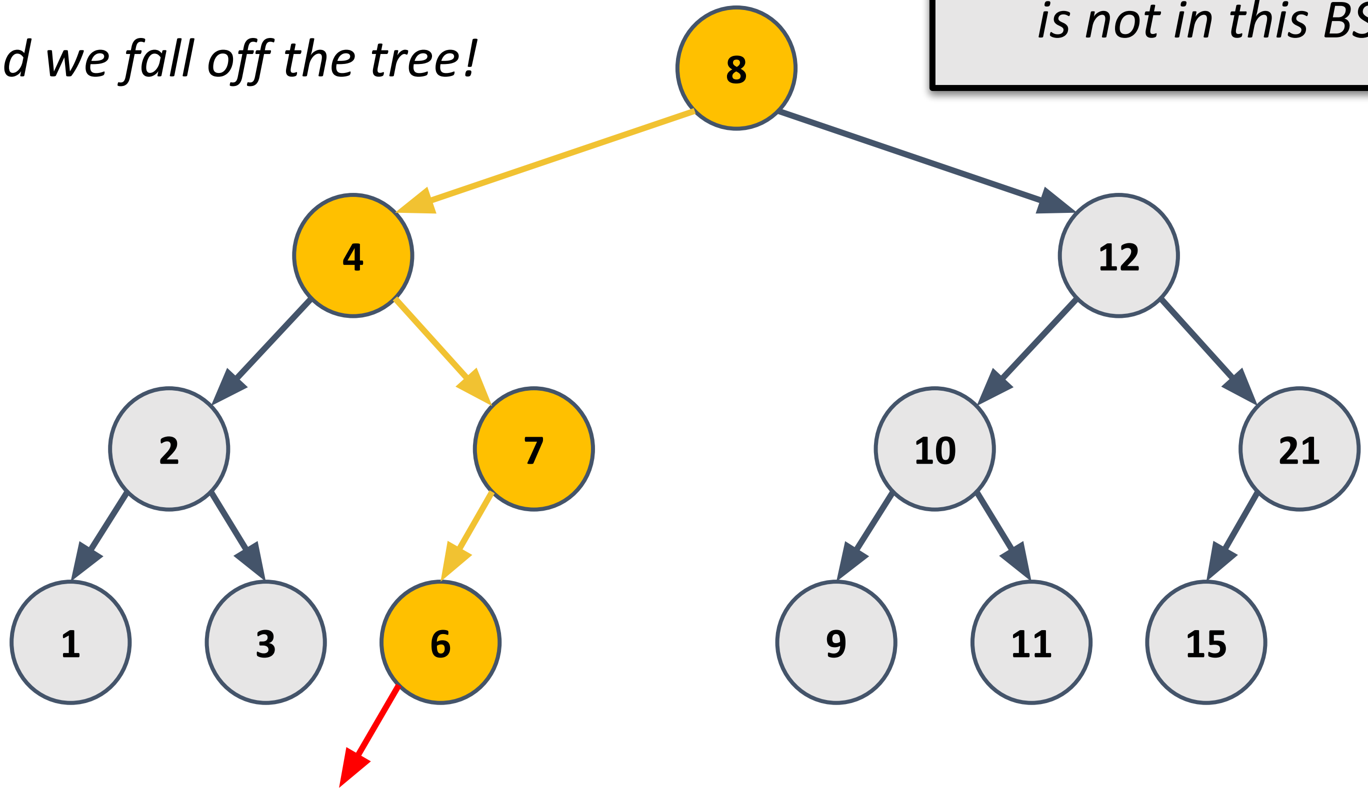
How do we know that 5 is not in this BST?



BST Lookups

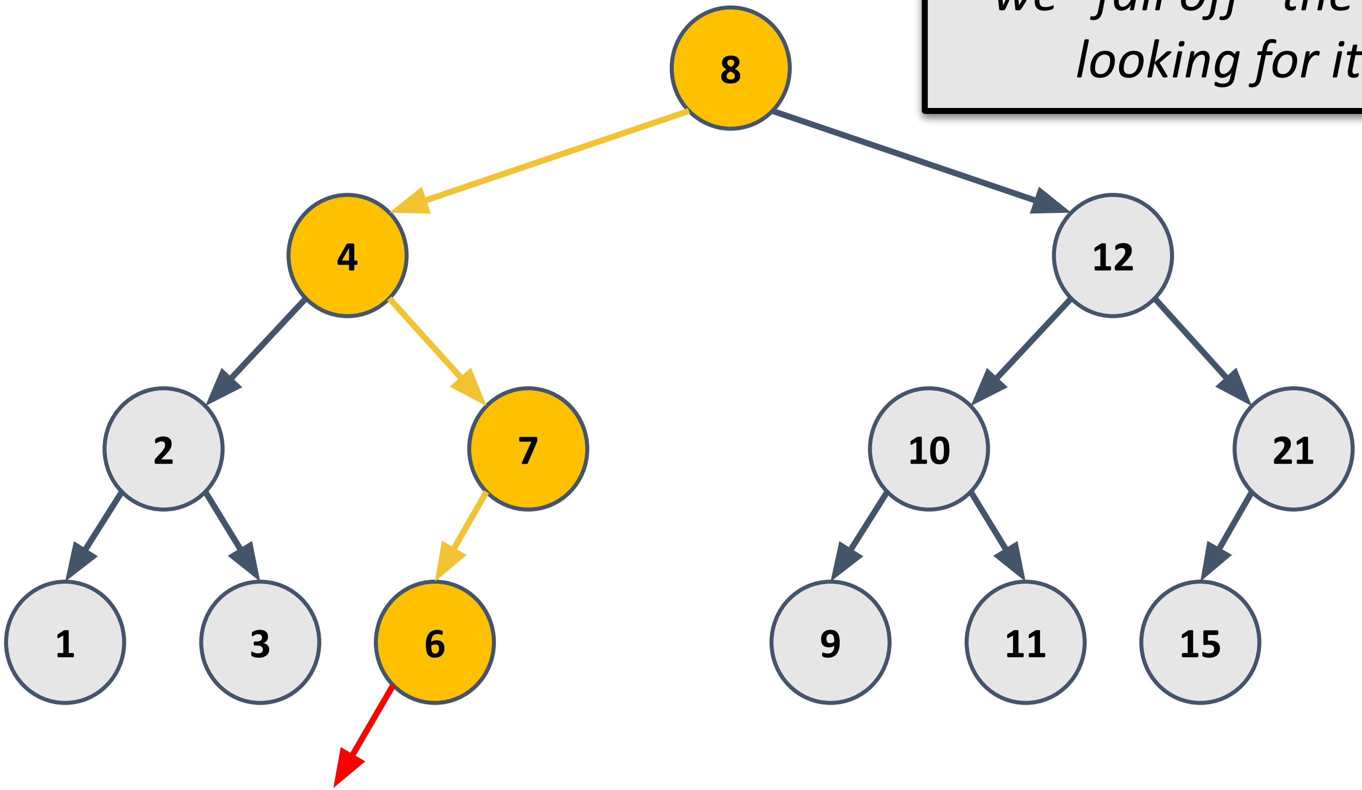
And we fall off the tree!

How do we know that 5 is not in this BST?



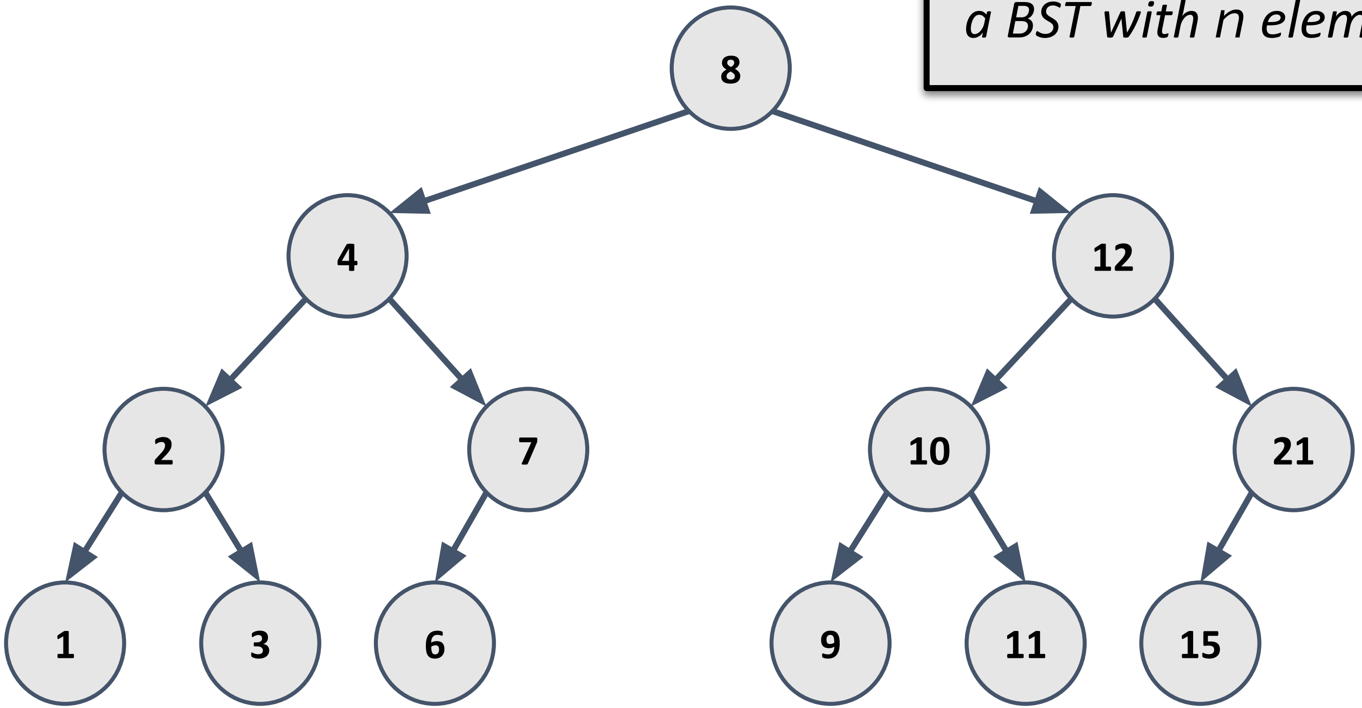
BST Lookups

A value isn't in our BST if we "fall off" the tree looking for it.



BST Lookups

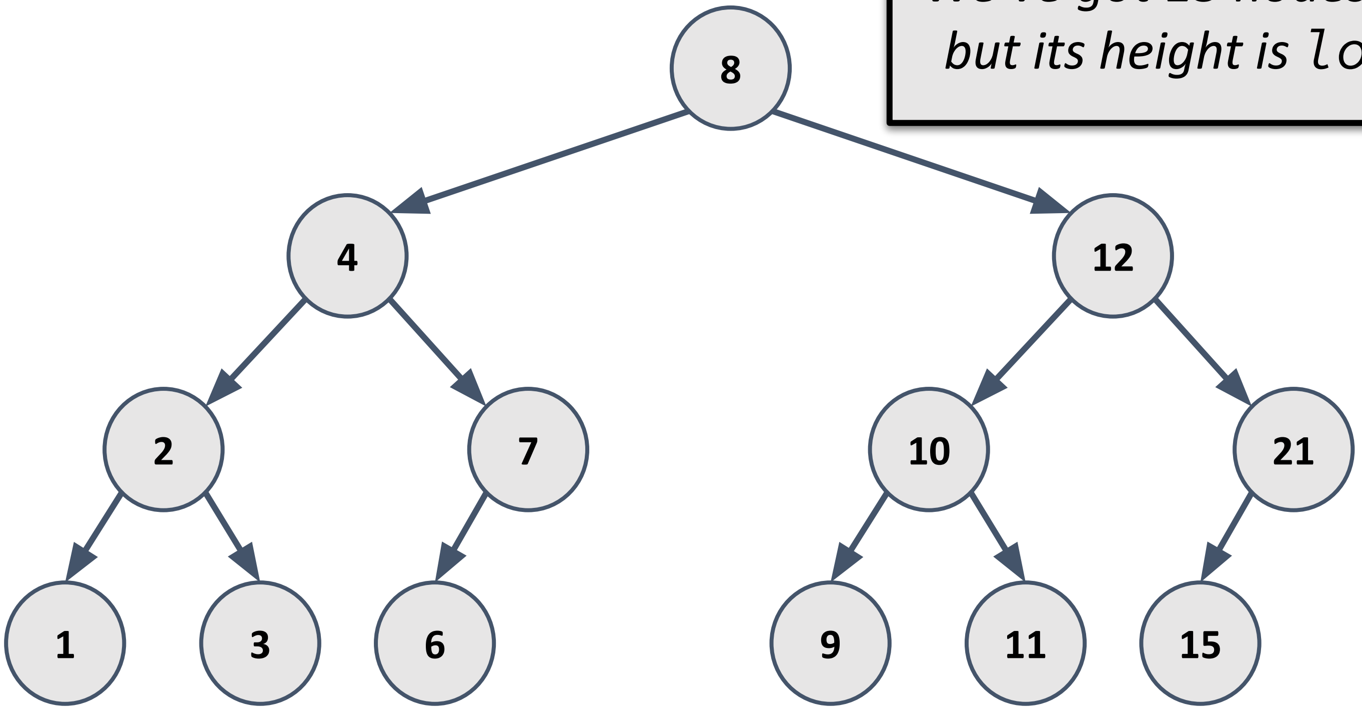
> 🤔 *What's the height of a BST with n elements?*



BST Lookups

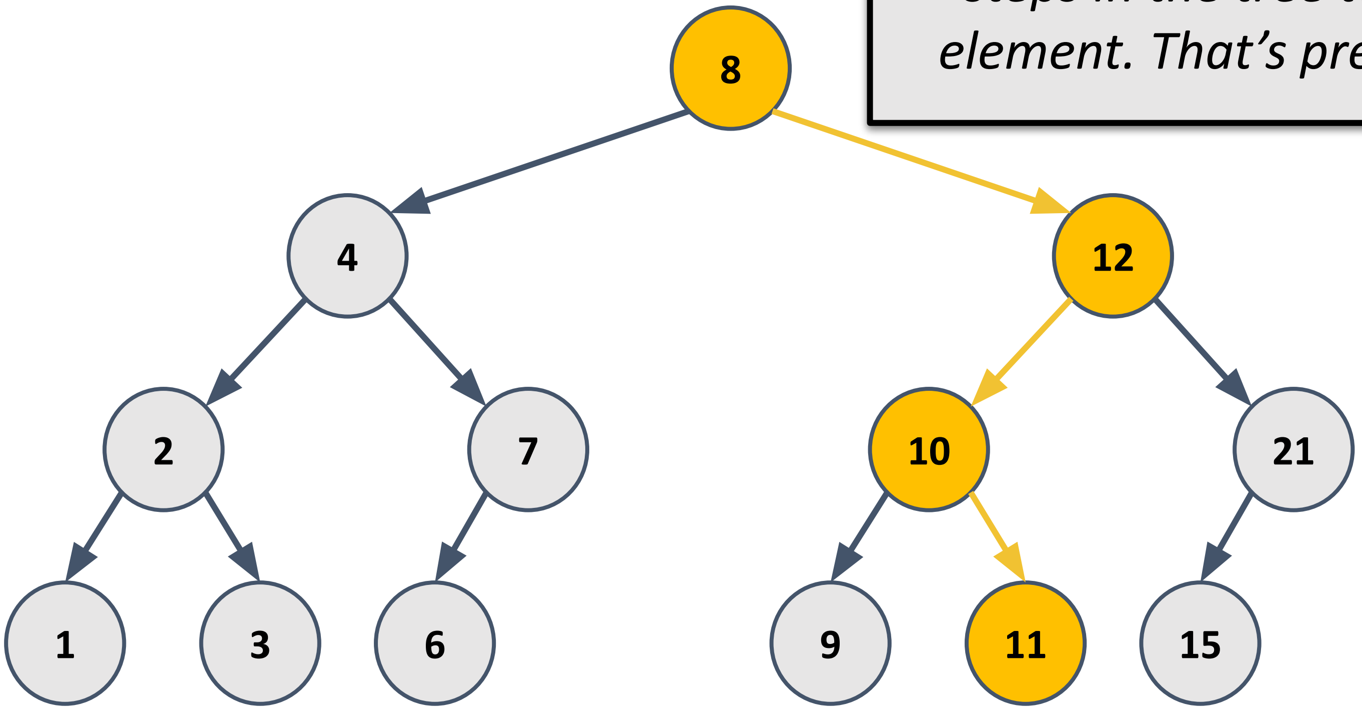
$O(\log_2 n)$

We've got 13 nodes in this tree, but its height is $\log_2 13 \approx 4$.



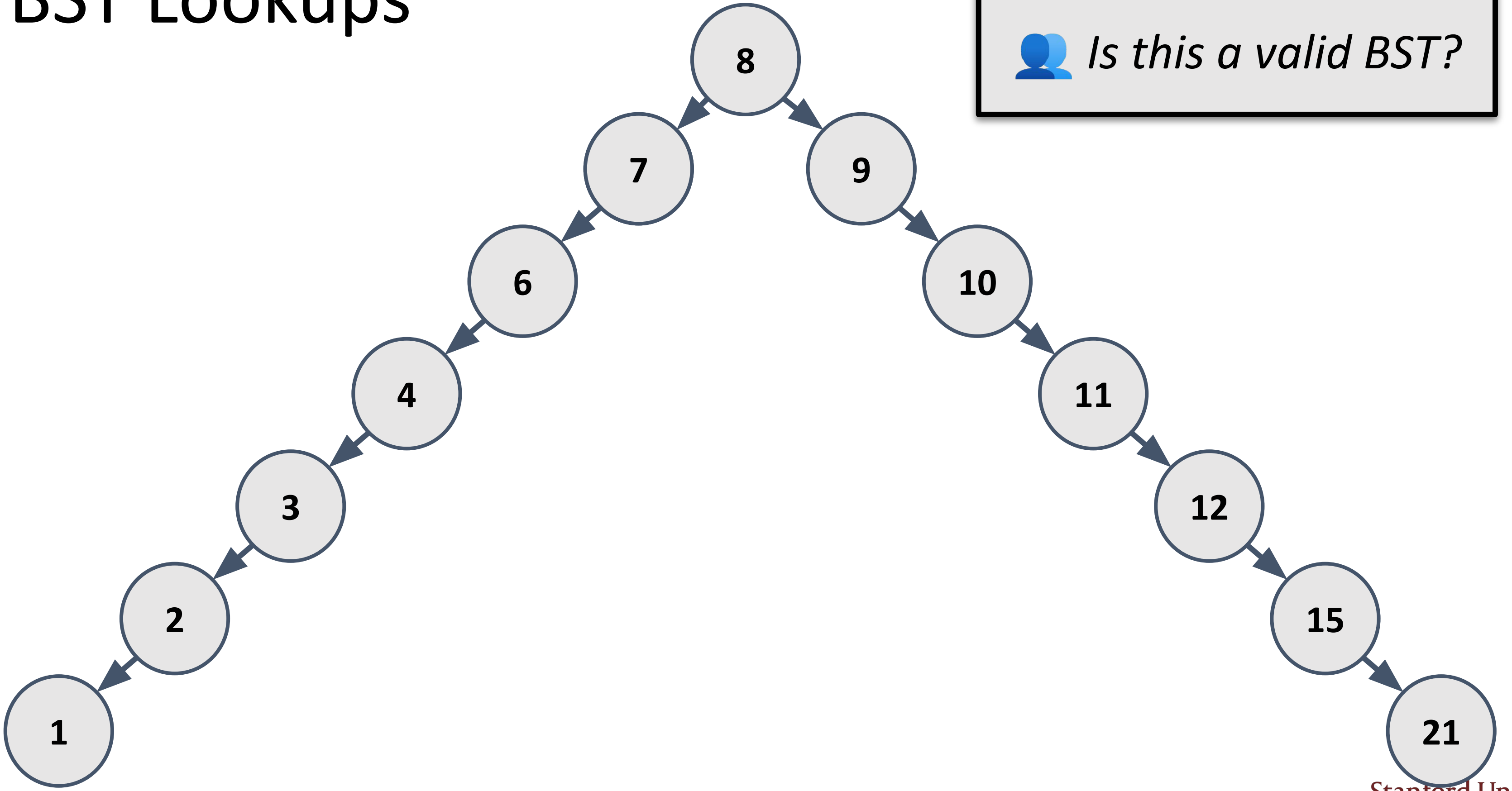
BST Lookups

Worst case, we have to take 4 steps in the tree to find an element. That's pretty good!



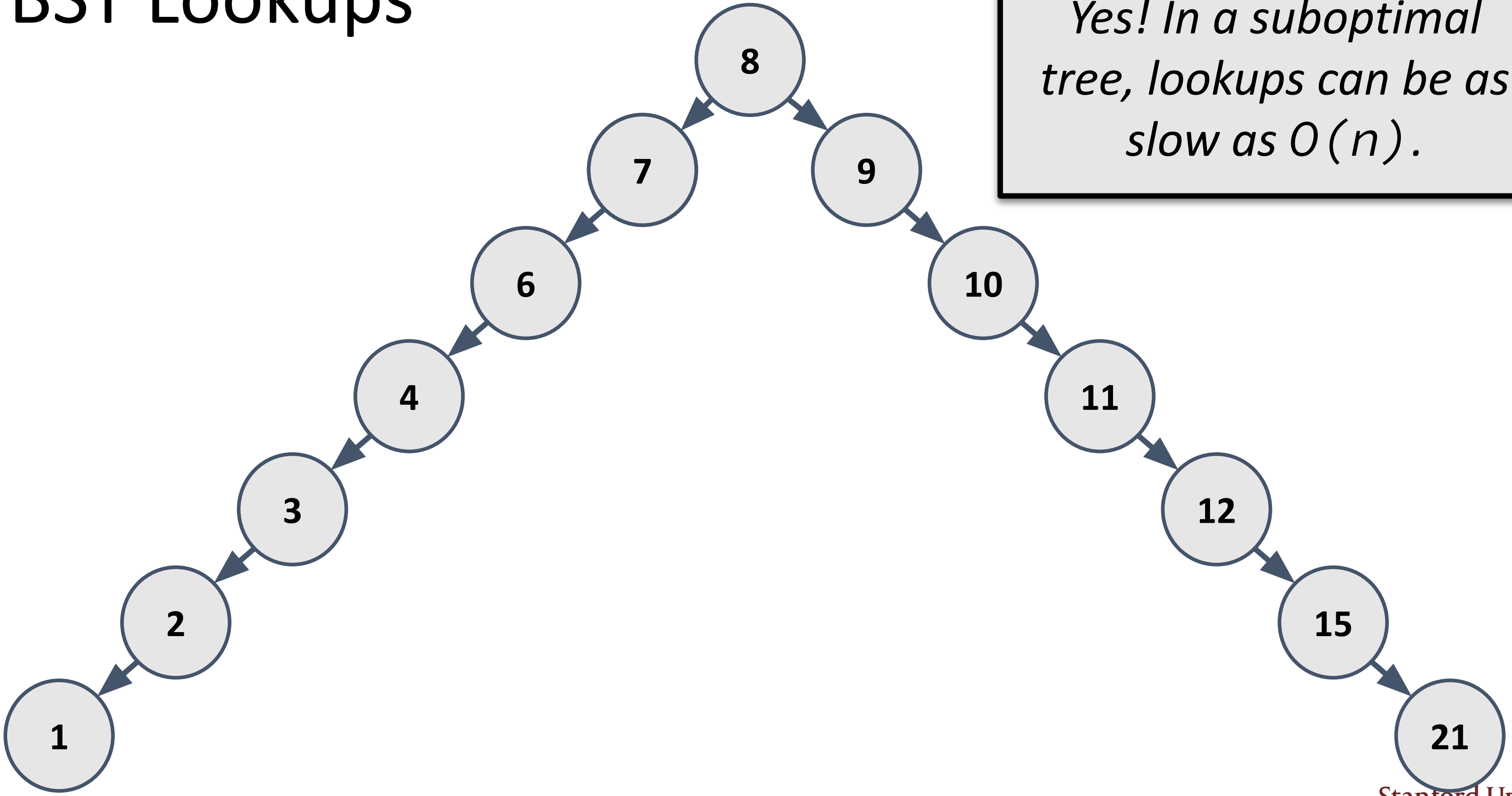
BST Lookups

Is this a valid BST?



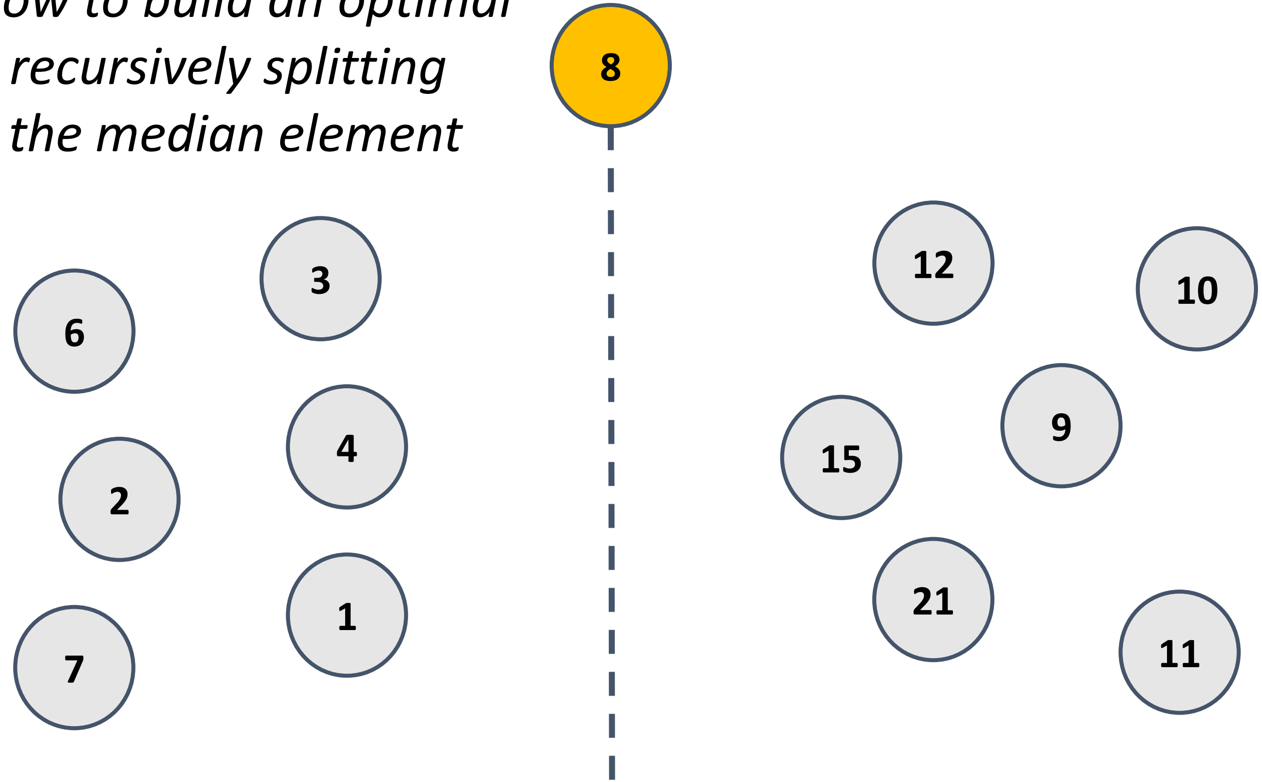
BST Lookups

Yes! In a suboptimal tree, lookups can be as slow as $O(n)$.



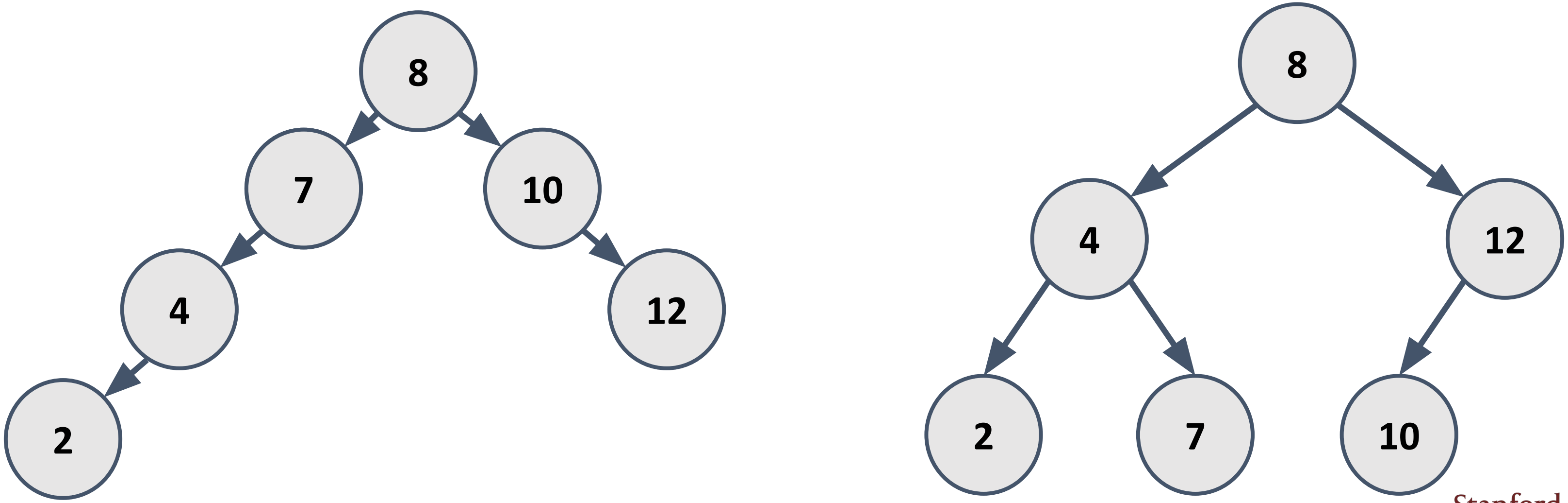
Building an Optimal BST

We saw how to build an optimal BST by recursively splitting around the median element



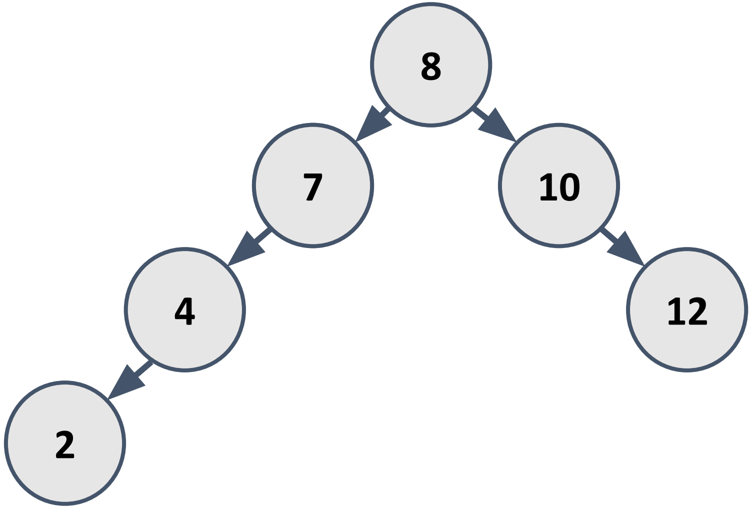
Takeaways

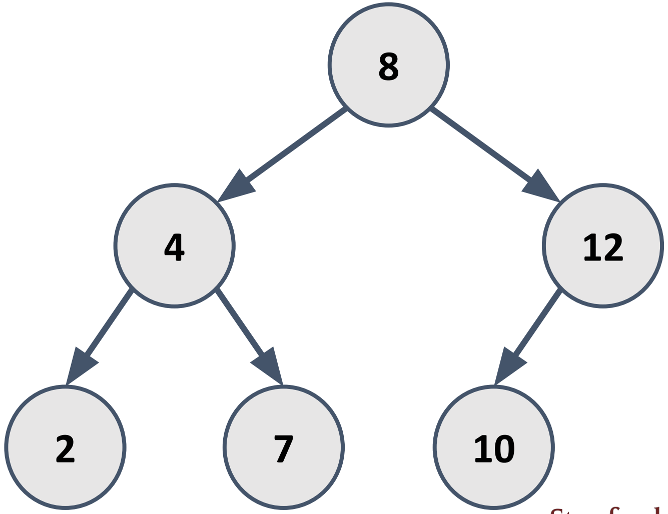
- There can be multiple valid BSTs for the same set of data
- How you construct the tree matters!



Balanced BSTs

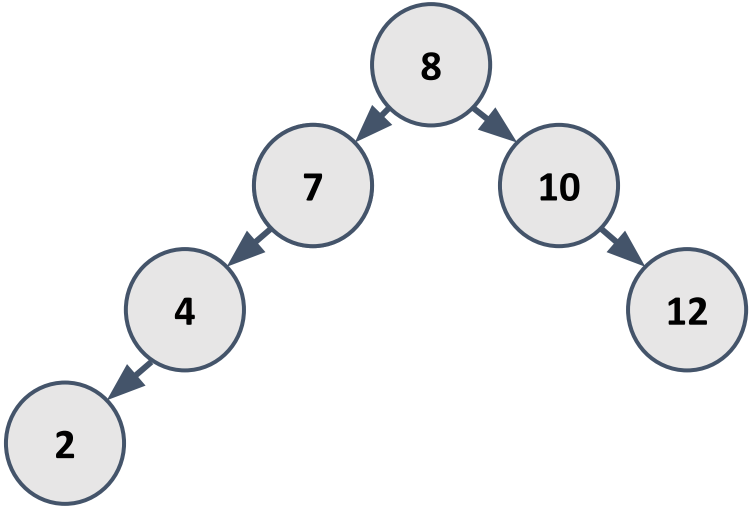
- A BST is **balanced** if its height is `O(log n)`, where `n` is the number of nodes in the tree
    - This means left/right subtrees don't differ in height by more than 1

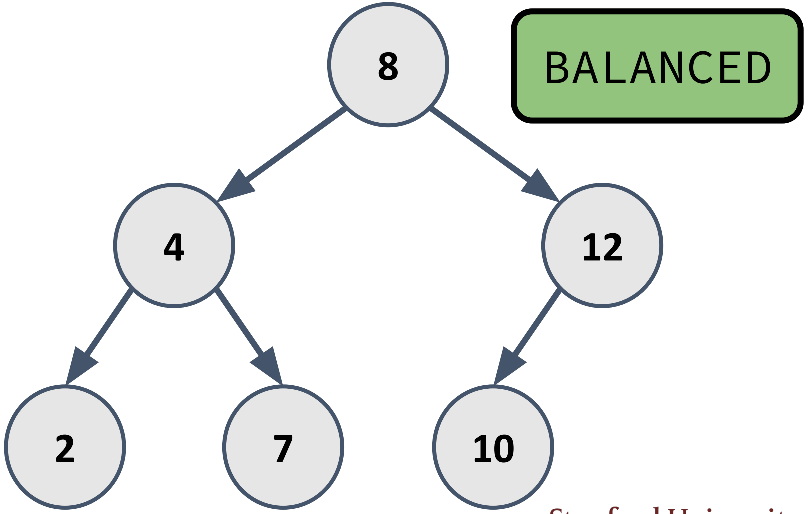




Balanced BSTs

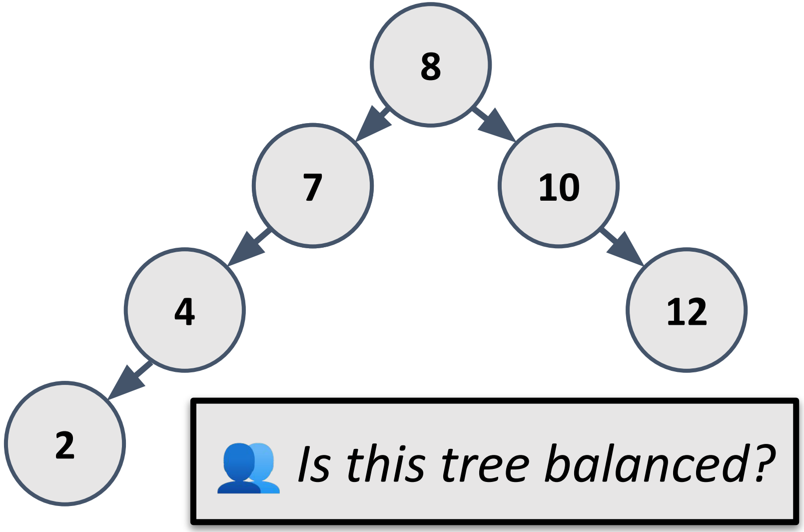
- A BST is **balanced** if its height is `O(log n)`, where `n` is the number of nodes in the tree
  - This means left/right subtrees don't differ in height by more than 1

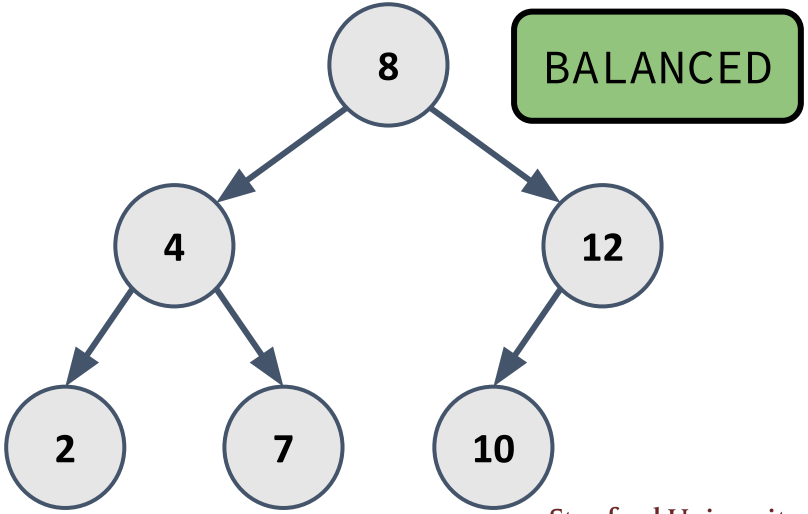




Balanced BSTs

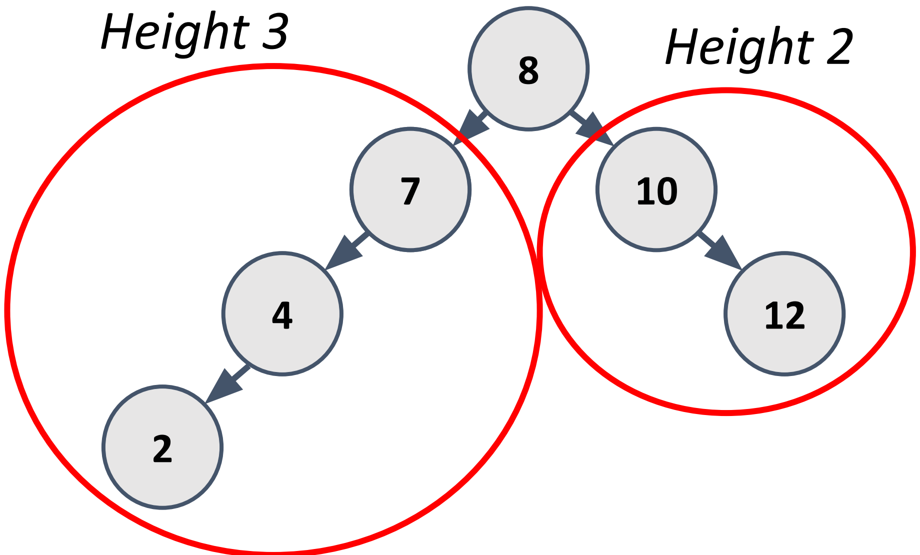
- A BST is **balanced** if its height is `O(log n)`, where `n` is the number of nodes in the tree
  - This means left/right subtrees don't differ in height by more than 1

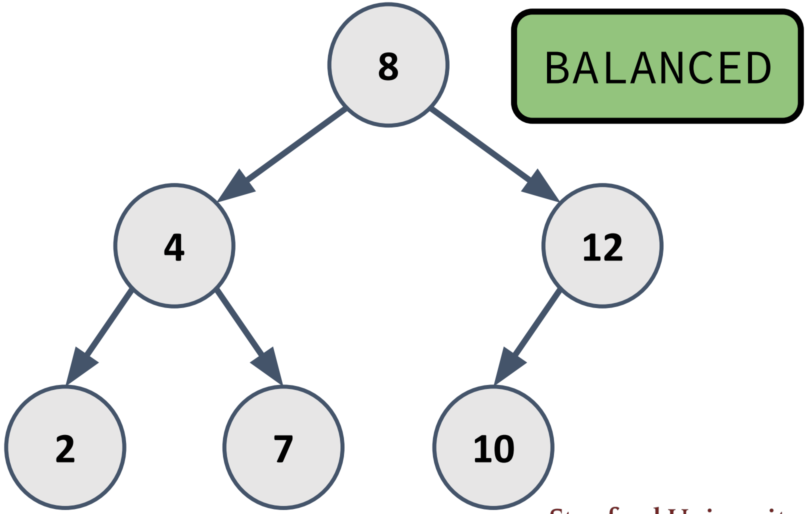




Balanced BSTs

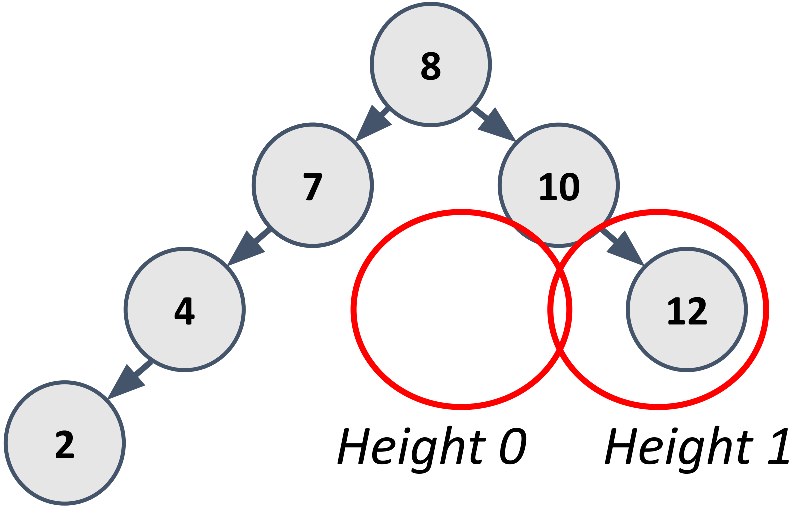
- A BST is **balanced** if its height is `O(log n)`, where `n` is the number of nodes in the tree
    - This means left/right subtrees **don't differ in height by more than 1**

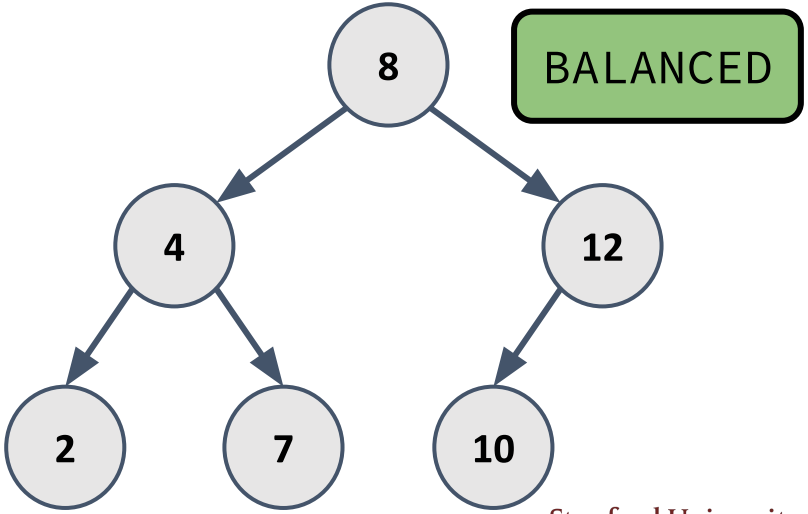




Balanced BSTs

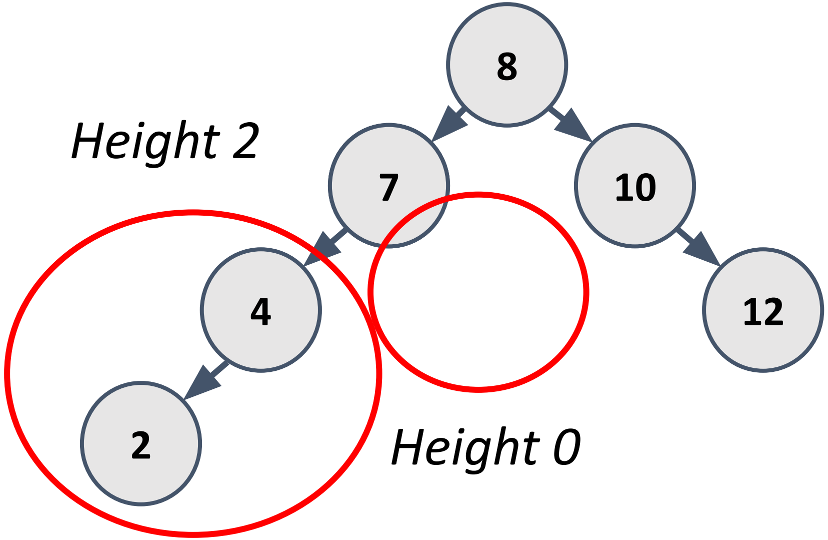
- A BST is **balanced** if its height is `O(log n)`, where `n` is the number of nodes in the tree
  - This means left/right subtrees **don’t differ in height by more than 1**

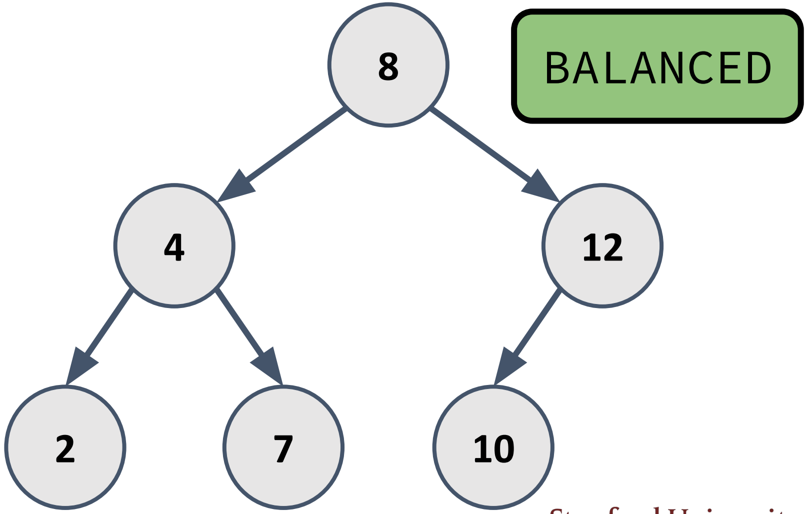




Balanced BSTs

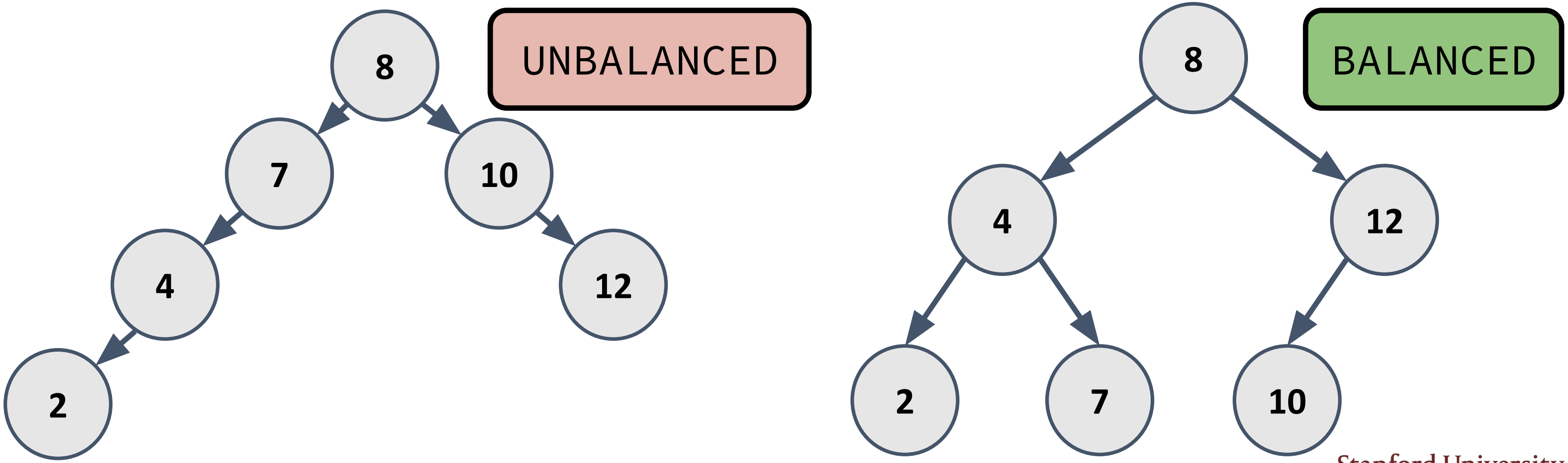
- A BST is **balanced** if its height is `O(log n)`, where `n` is the number of nodes in the tree
  - This means left/right subtrees **don't differ in height by more than 1**





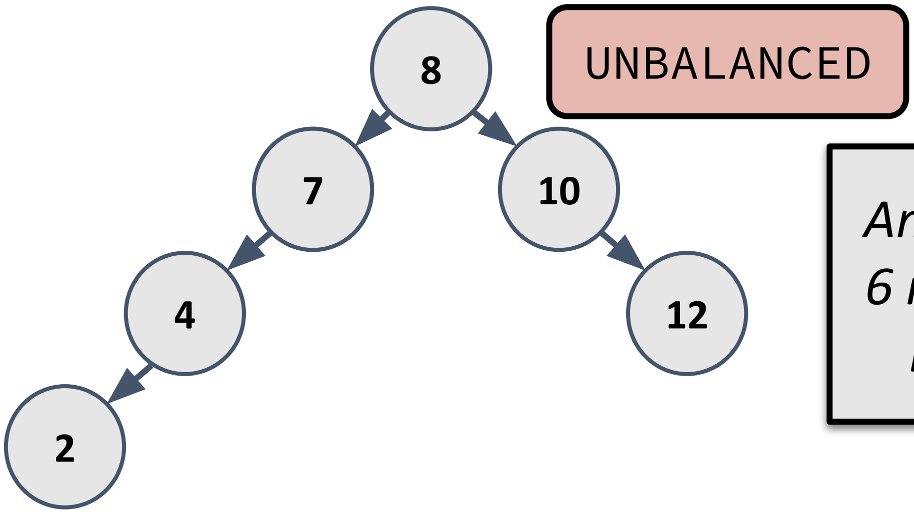
Balanced BSTs

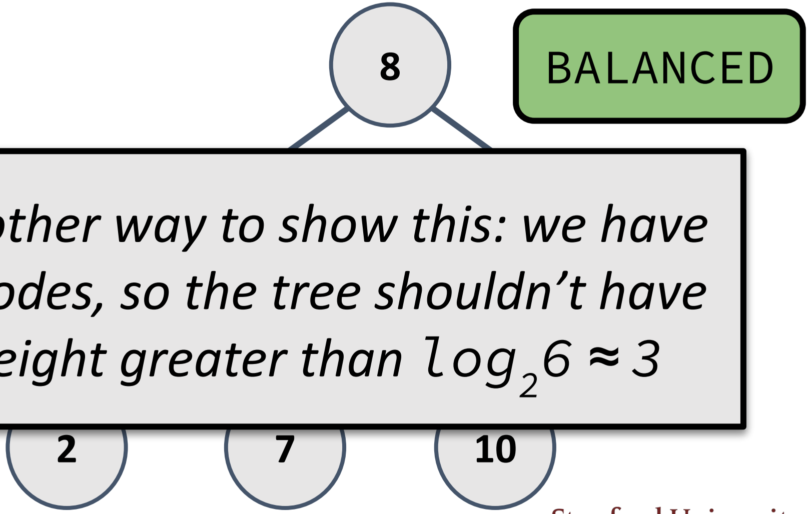
- A BST is **balanced** if its height is `O(log n)`, where `n` is the number of nodes in the tree
  - This means left/right subtrees **don't differ in height by more than 1**



Balanced BSTs

- A BST is **balanced** if its **height is `O(log n)`**, where n is the number of nodes in the tree
  - This means left/right subtrees don't differ in height by more than 1





Another way to show this: we have 6 nodes, so the tree shouldn't have height greater than $\log_2 6 \approx 3$

Balanced BSTs

- A BST is **balanced** if its height is `O(log n)`, where `n` is the number of nodes in the tree
- Theorem: If you start with an empty tree and add in random values, then with high probability the tree is balanced
    - Take CS161 to find out why!

Balanced BSTs

- A BST is **balanced** if its height is `O(log n)`, where `n` is the number of nodes in the tree
- Theorem: If you start with an empty tree and add in random values, then with high probability the tree is balanced
  - Take CS161 to find out why!
- A self-balancing BST reshapes itself on insertions and deletions to stay balanced (how to do this is beyond the scope of this class)
  - AVL trees
  - Red-black trees

Big-O of ADT Operations

Vectors

- .size() - O(1)
- .add() - O(1)
- v[i] - O(1)
- .insert() - O(n)
- .rem
- .sub
- trav

Grids

- .num
- .numCols() - O(1)
- grid[i][j] - O(1)
- .inBounds() - O(1)
- traversal - $O(n^2)$

Queues

- .size() - O(1)
- .peek() - O(1)
- .enqueue() - O(1)

- .peek() - O(1)
- .push() - O(1)
- .pop() - O(1)
- .isEmpty() - O(1)
- traversal - O(n)

Sets

- .size() - O(1)
- .isEmpty() - O(1)
- .add() - O(log n)
- .remove() - O(log n)
- **.contains() - O(log n)**
- traversal - O(n)

Maps

- .size() - O(1)
- .isEmpty() - O(1)
- m[key] - O(log n)
- **.contains() - O(log n)**
- traversal - O(n)

Why do Sets and Maps have O(log n) lookups? They use BSTs behind the scenes to store data!

Big-O of ADT Operations

Vectors

- .size() – O(1)
- .add() – O(1)
- v[i] – O(1)
- .insert() – O(n)
- .remove() – O(n)
- .sub
- trav

Grids

- .num
- .numCols() – O(1)
- grid[i][j] – O(1)
- .inBounds() – O(1)
- traversal – $O(n^2)$

Queues

- .size() – O(1)
- .peek() – O(1)
- .enqueue() – O(1)
- .dequeue() – O(1)

- .peek() – O(1)
- .push() – O(1)
- .pop() – O(1)
- .isEmpty() – O(1)
- traversal – O(n)

Sets

- .size() – O(1)
- .isEmpty() – O(1)
- **.add() – O(log n)**
- **.remove() – O(log n)**
- **.contains() – O(log n)**
- traversal – O(n)

Maps

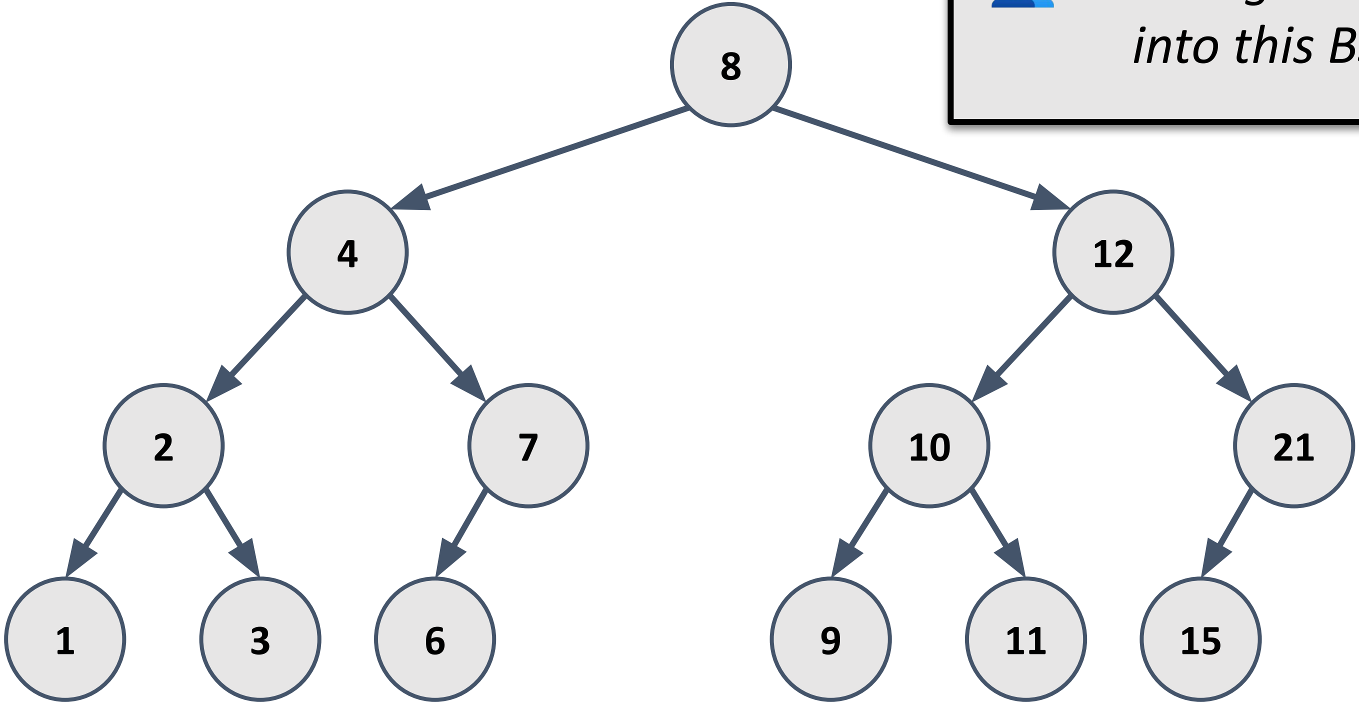
- .size() – O(1)
- .isEmpty() – O(1)
- **m[key] – O(log n)**
- **.contains() – O(log n)**
- traversal – O(n)

Let's investigate how BSTs can have O(log n) insertion and deletion.

BST Insertion

BST Insertion

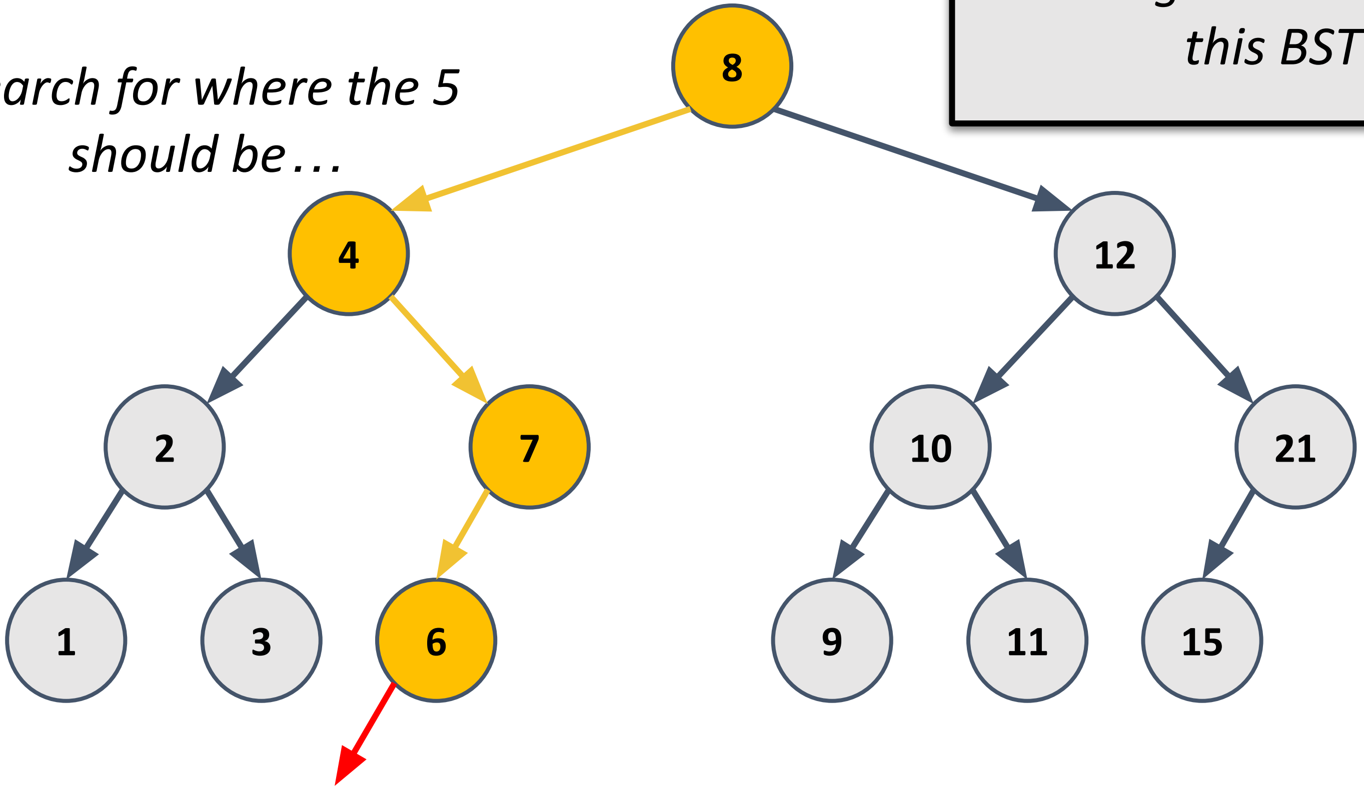
👥 *How might we insert 5 into this BST?*



BST Insertion

Search for where the 5 should be…

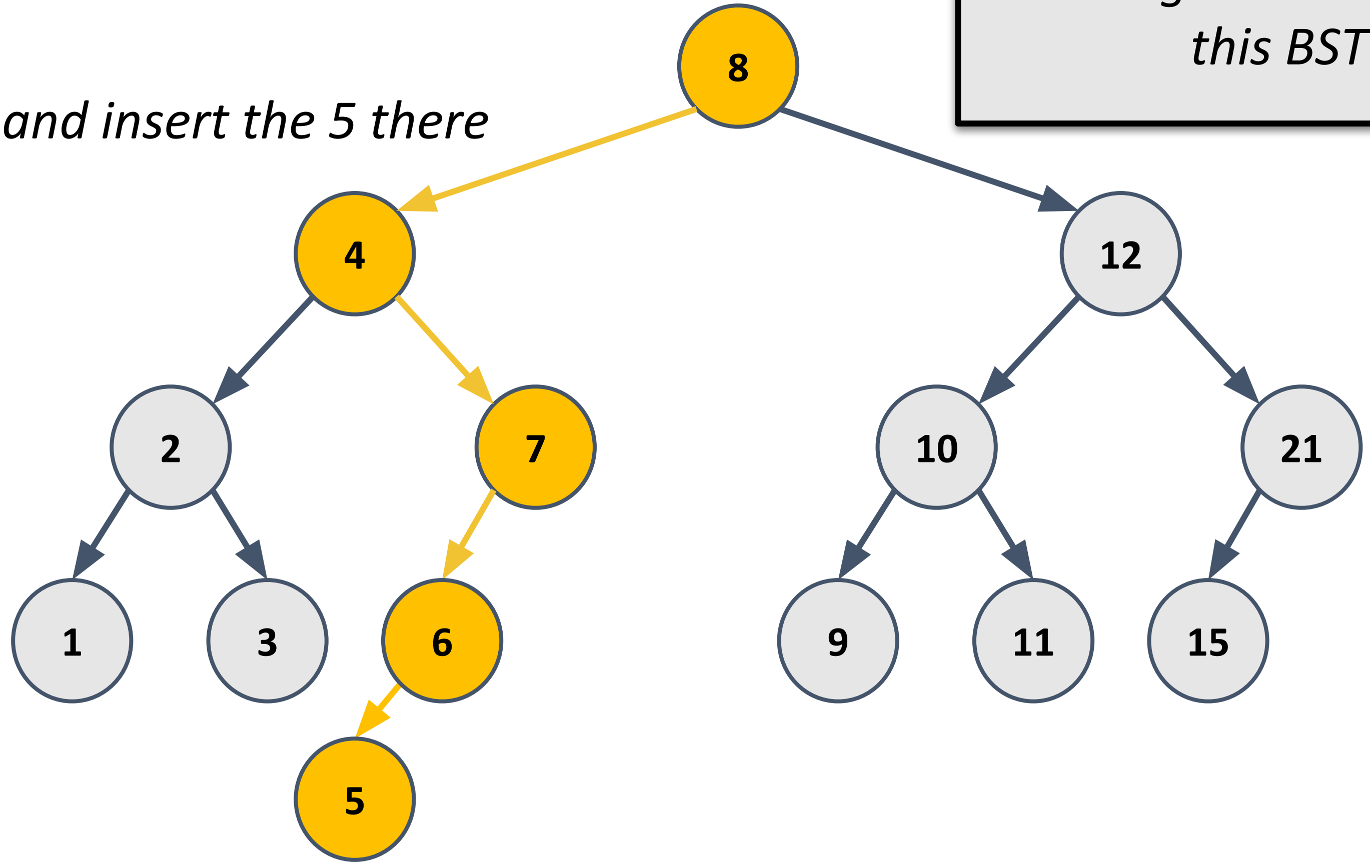
How might we insert 5 into this BST?



BST Insertion

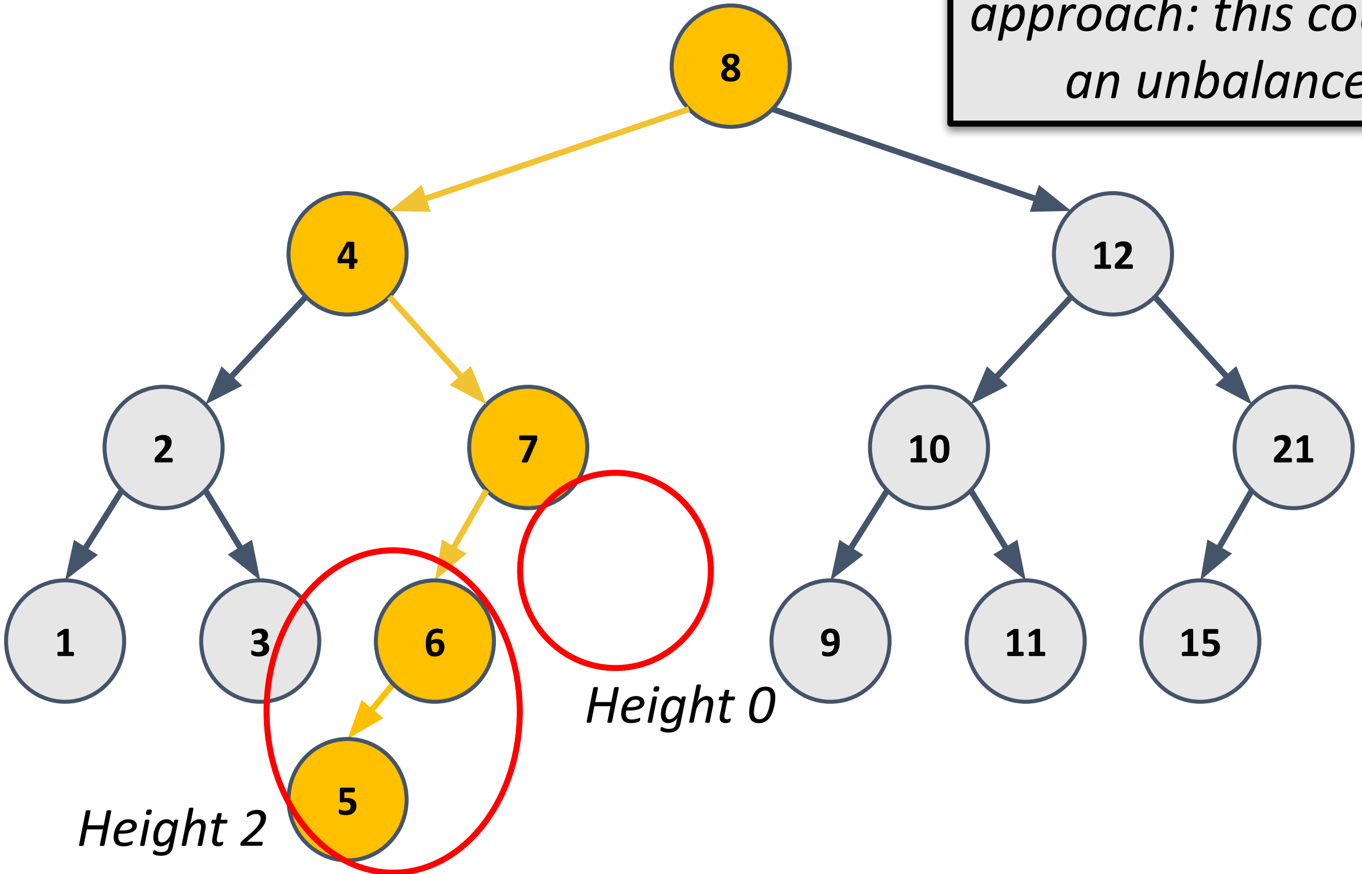
… and insert the 5 there

How might we insert 5 into this BST?



BST Insertion

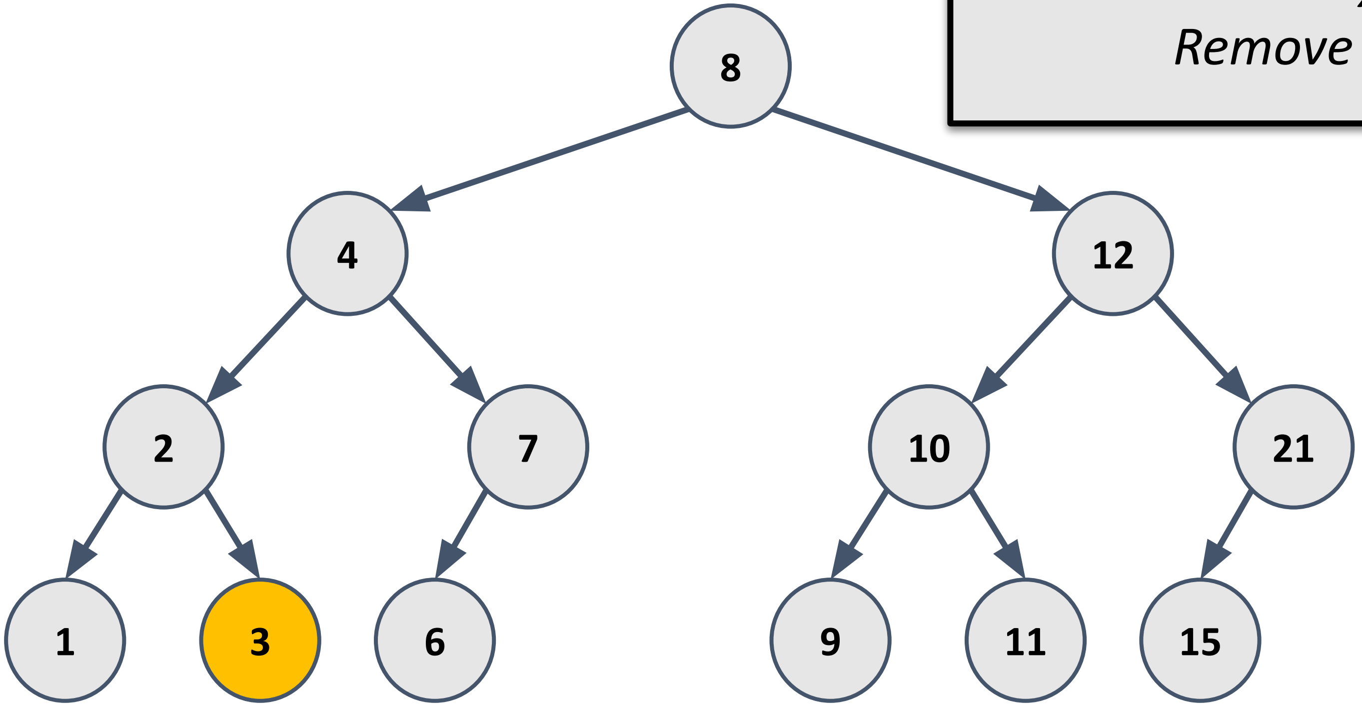
We're using a naïve approach: this could lead to an unbalanced tree



BST Deletion

BST Deletion

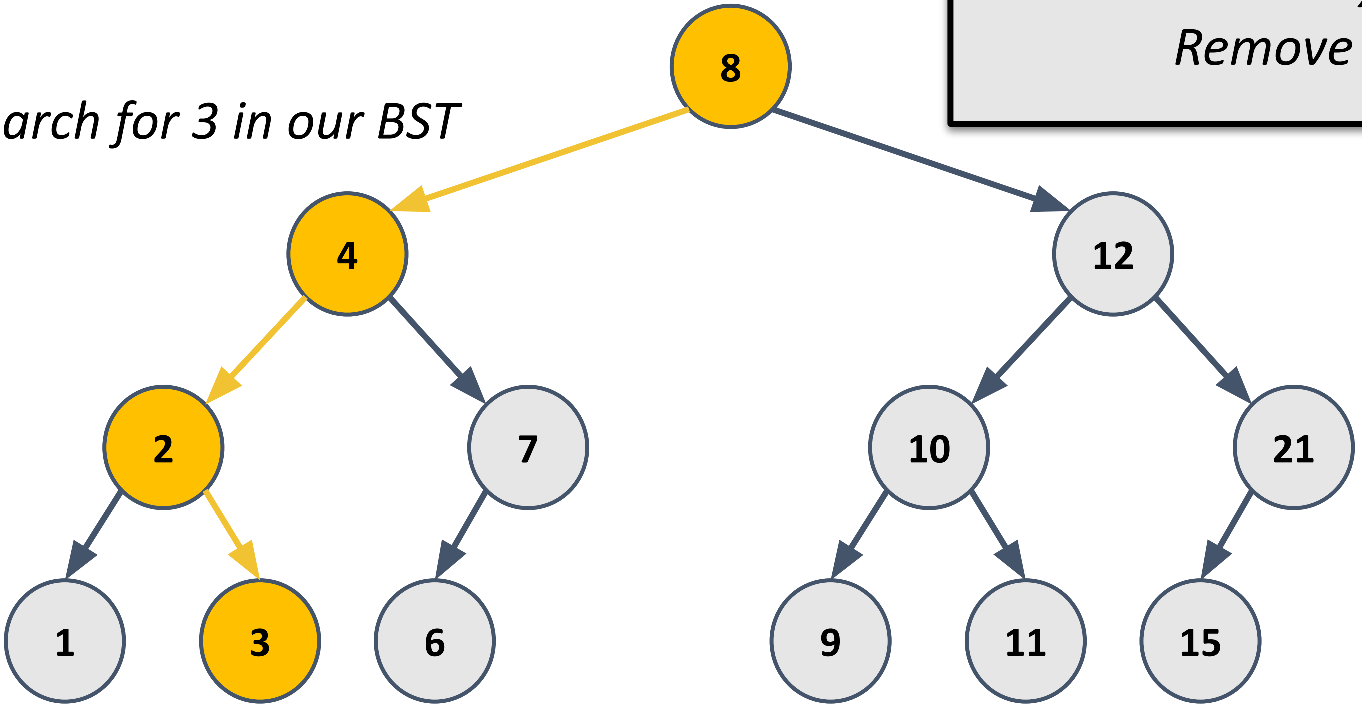
Here's an easy case:
Remove 3



BST Deletion

Here's an easy case: Remove 3

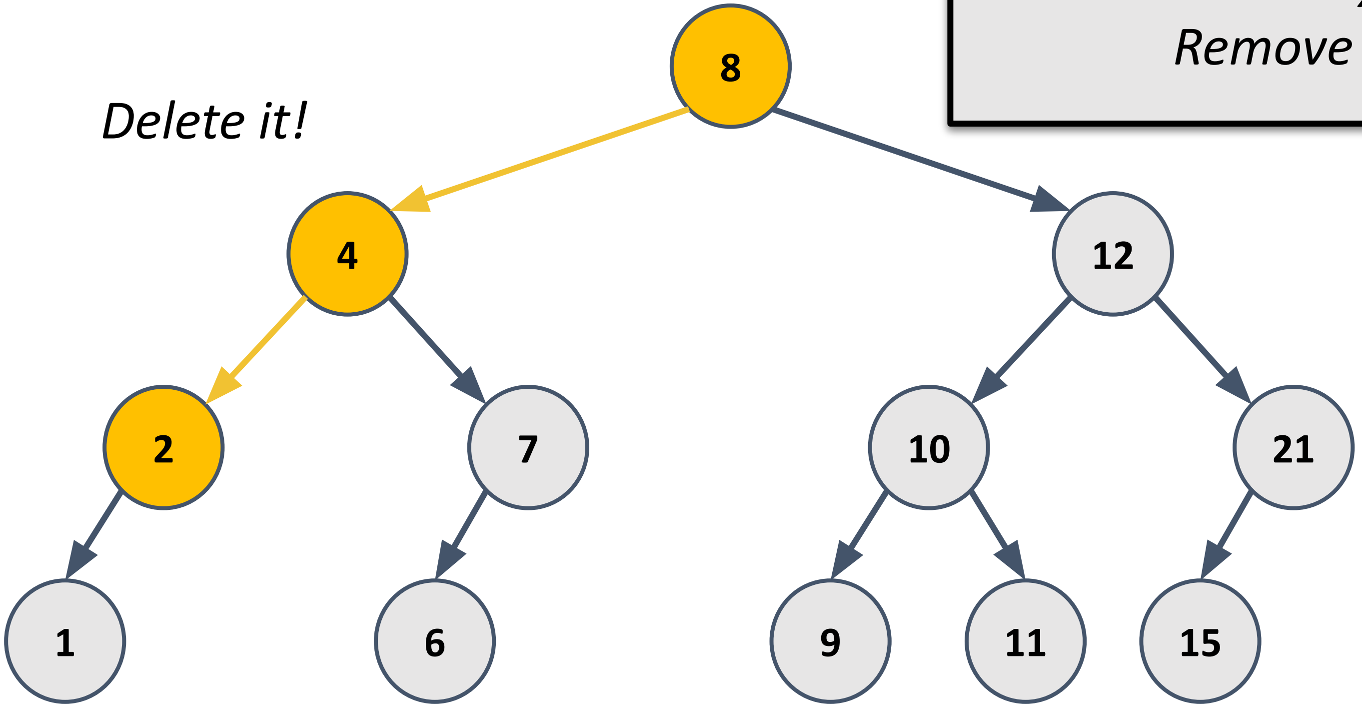
Search for 3 in our BST



BST Deletion

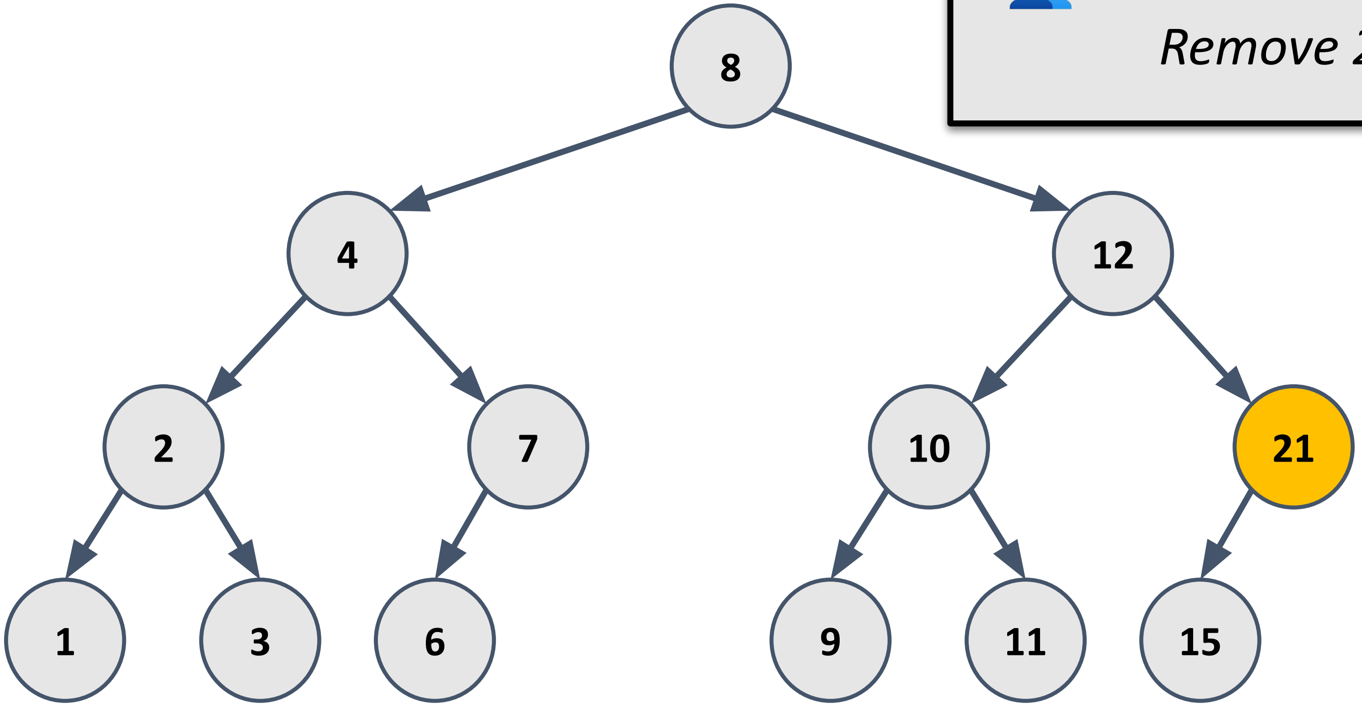
Here's an easy case:
Remove 3

Delete it!



BST Deletion

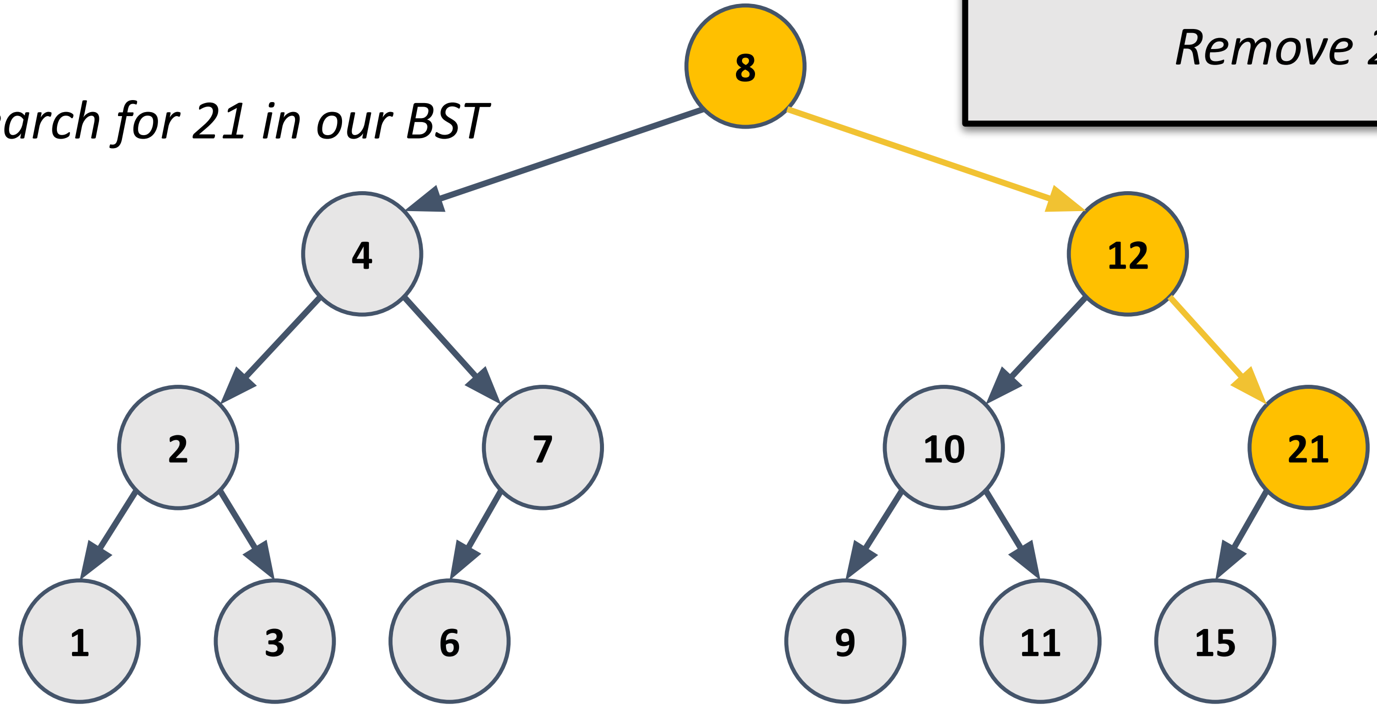
Here's a harder case: Remove 21



BST Deletion

Here's a harder case: Remove 21

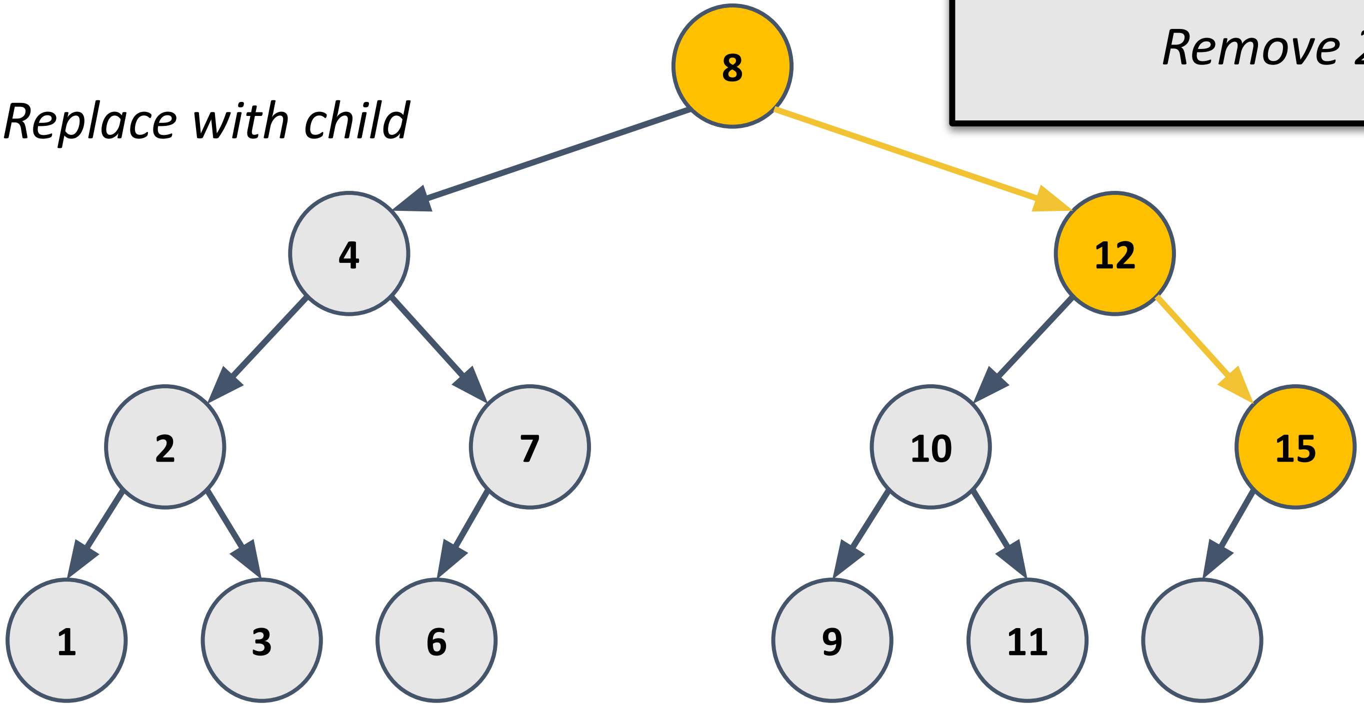
Search for 21 in our BST



BST Deletion

Here's a harder case:
Remove 21

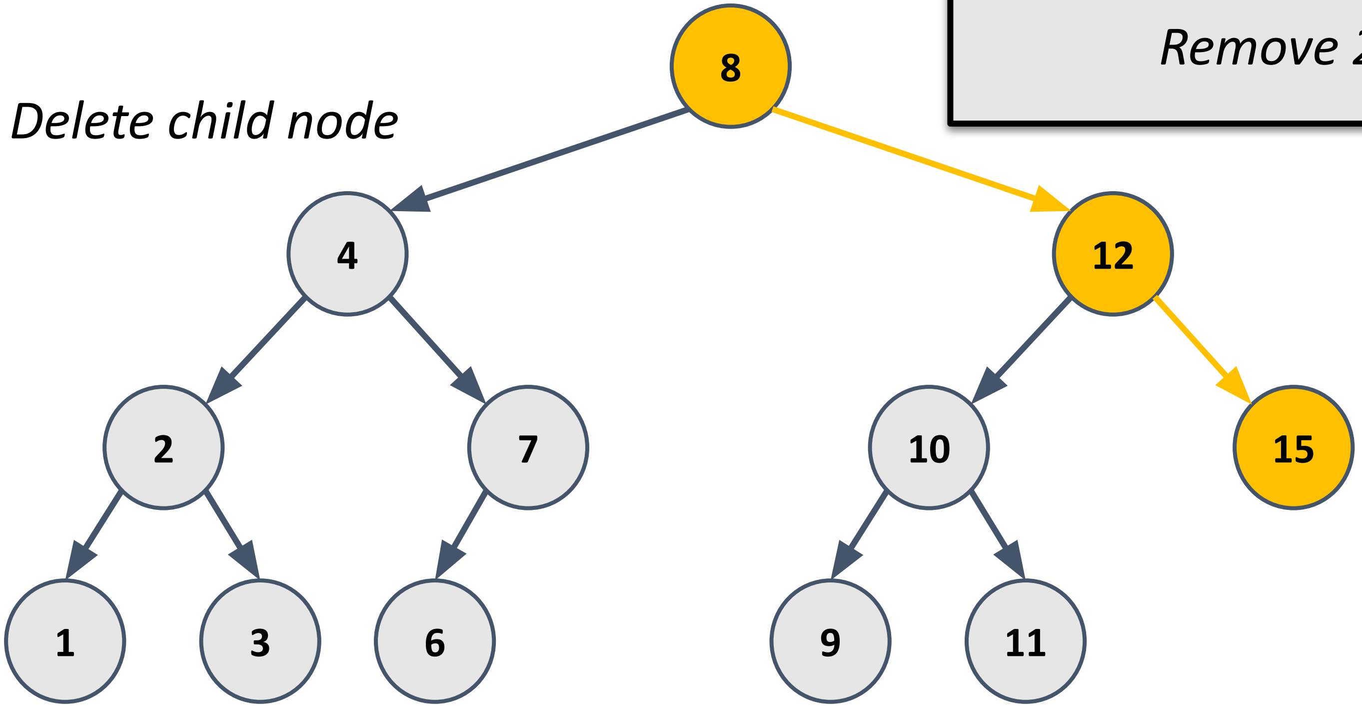
Replace with child



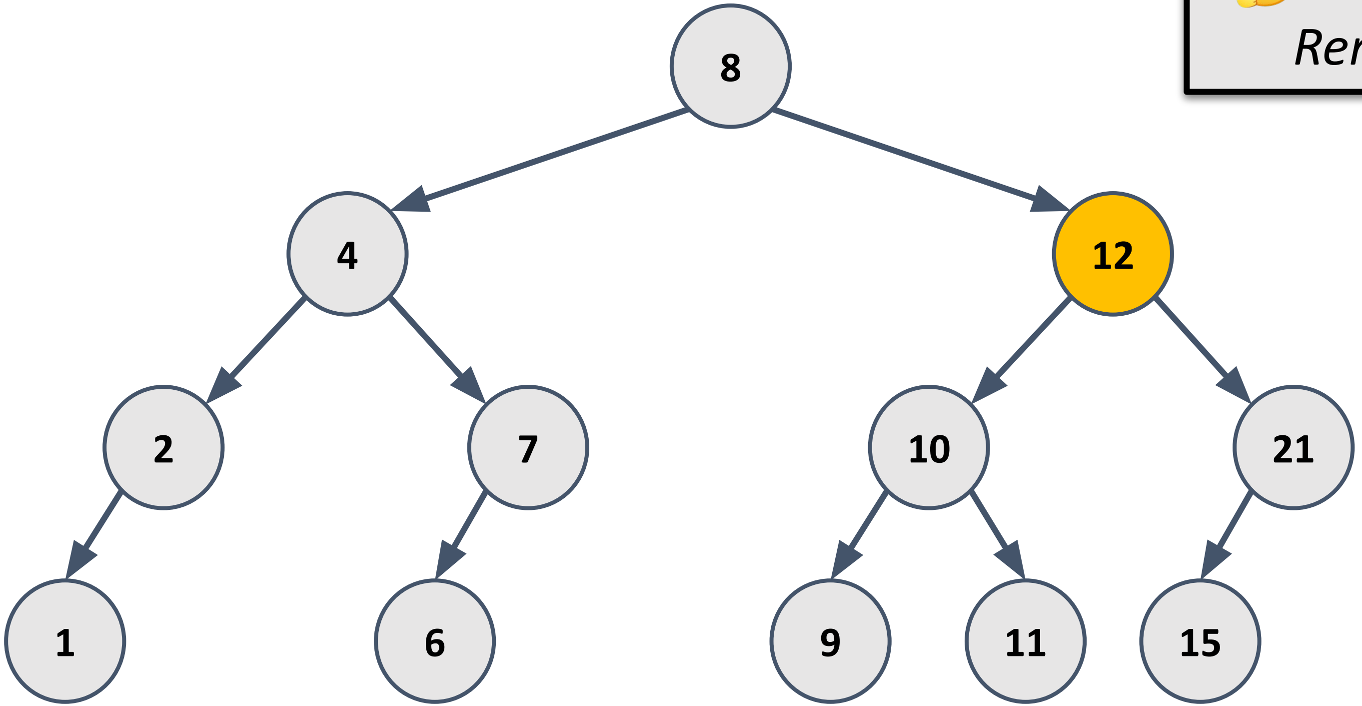
BST Deletion

Here's a harder case:
Remove 21

Delete child node



BST Deletion

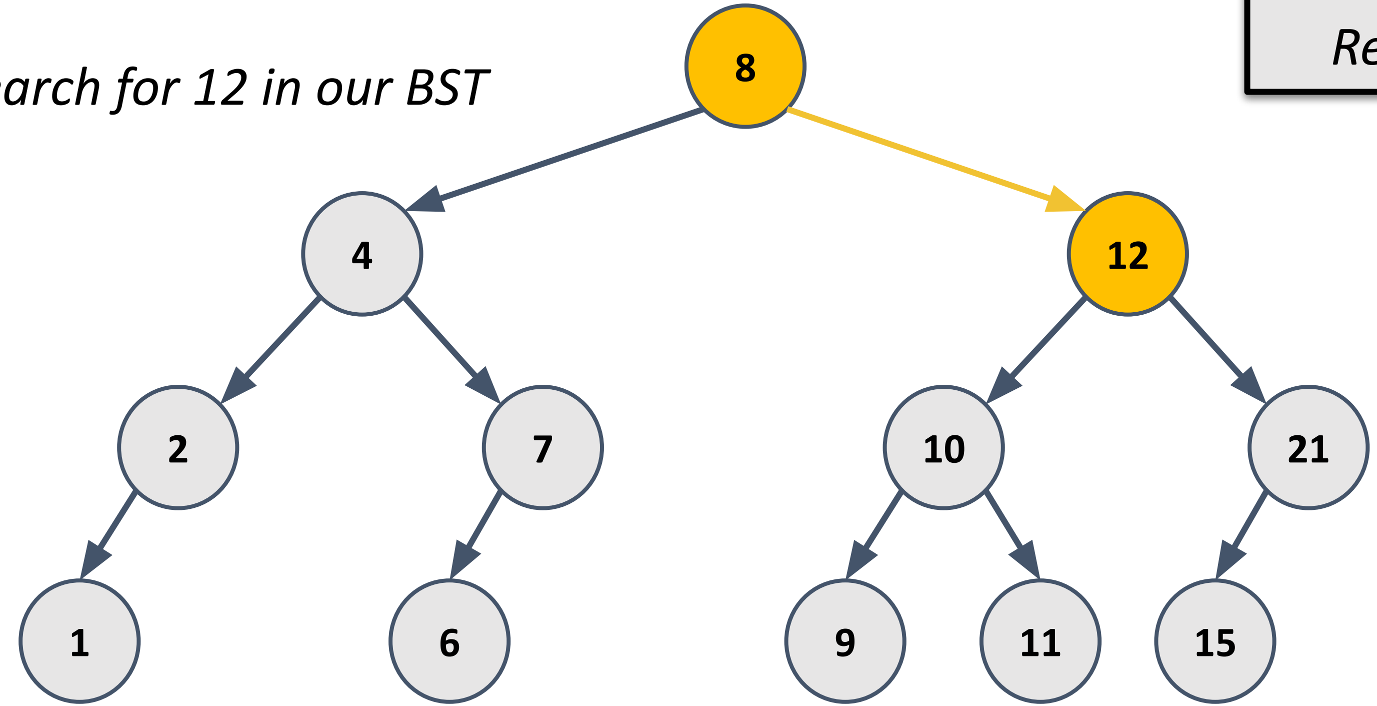


🤔 *Even trickier: Remove 12*

BST Deletion

Search for 12 in our BST

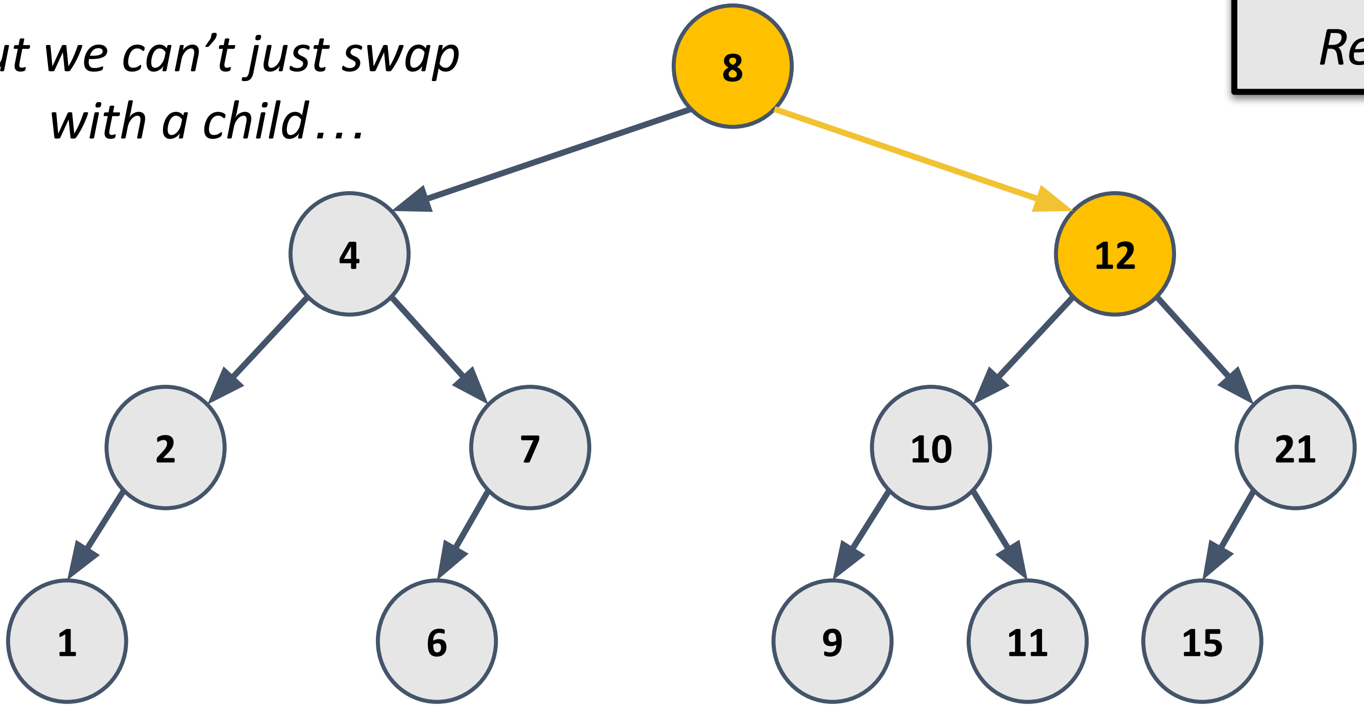
Even trickier:
Remove 12



BST Deletion

But we can't just swap with a child…

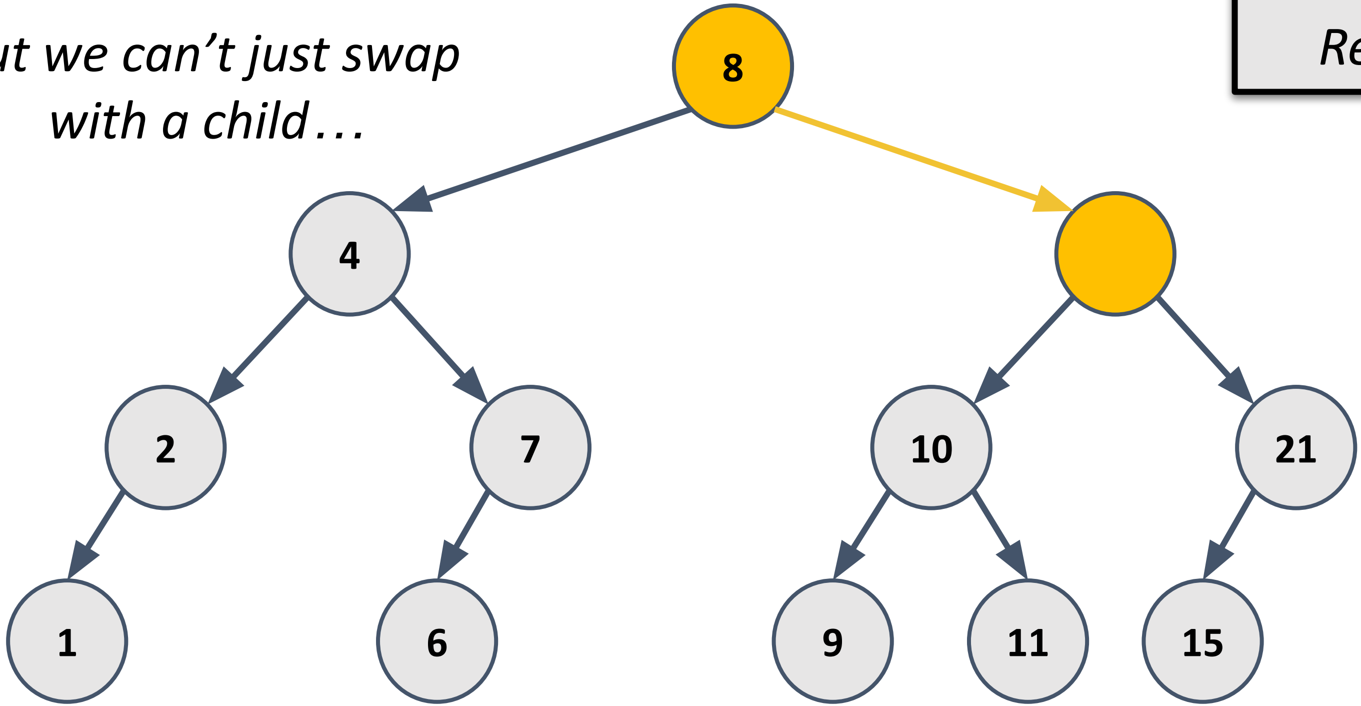
Even trickier: Remove 12



BST Deletion

But we can't just swap with a child…

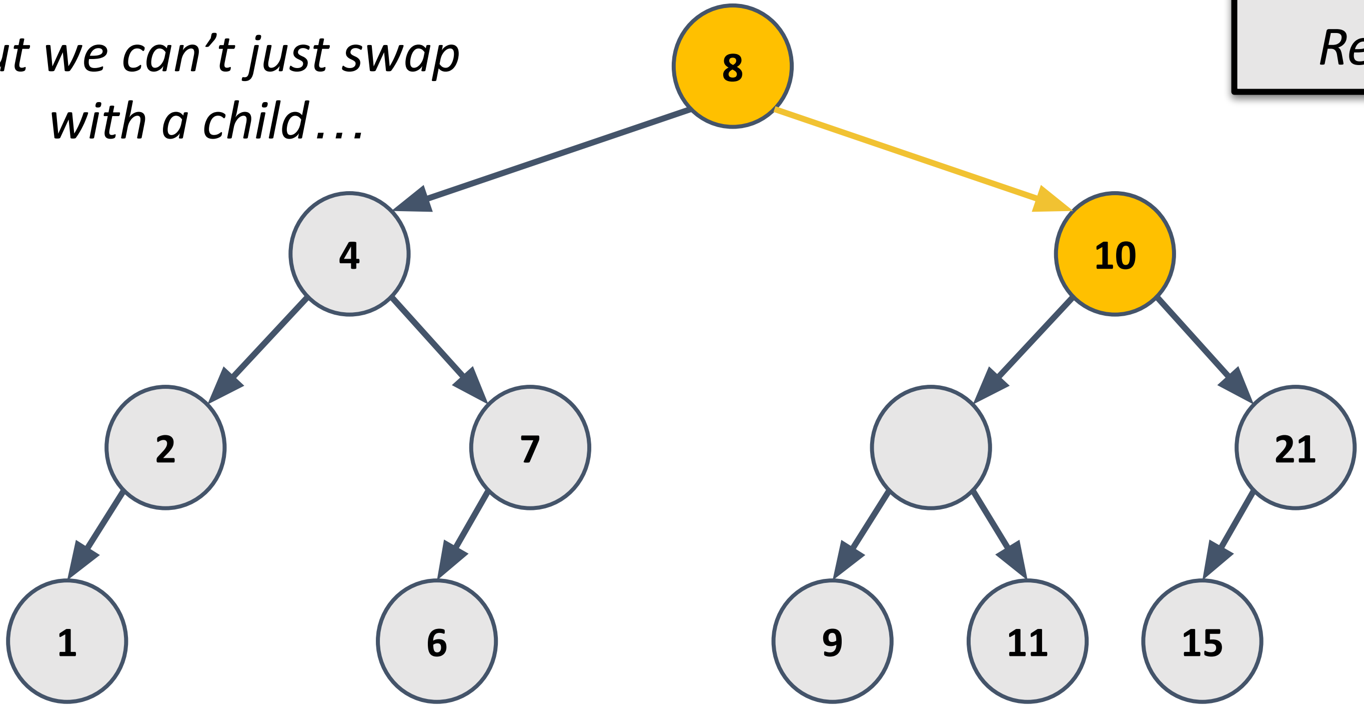
Even trickier: Remove 12



BST Deletion

But we can't just swap with a child…

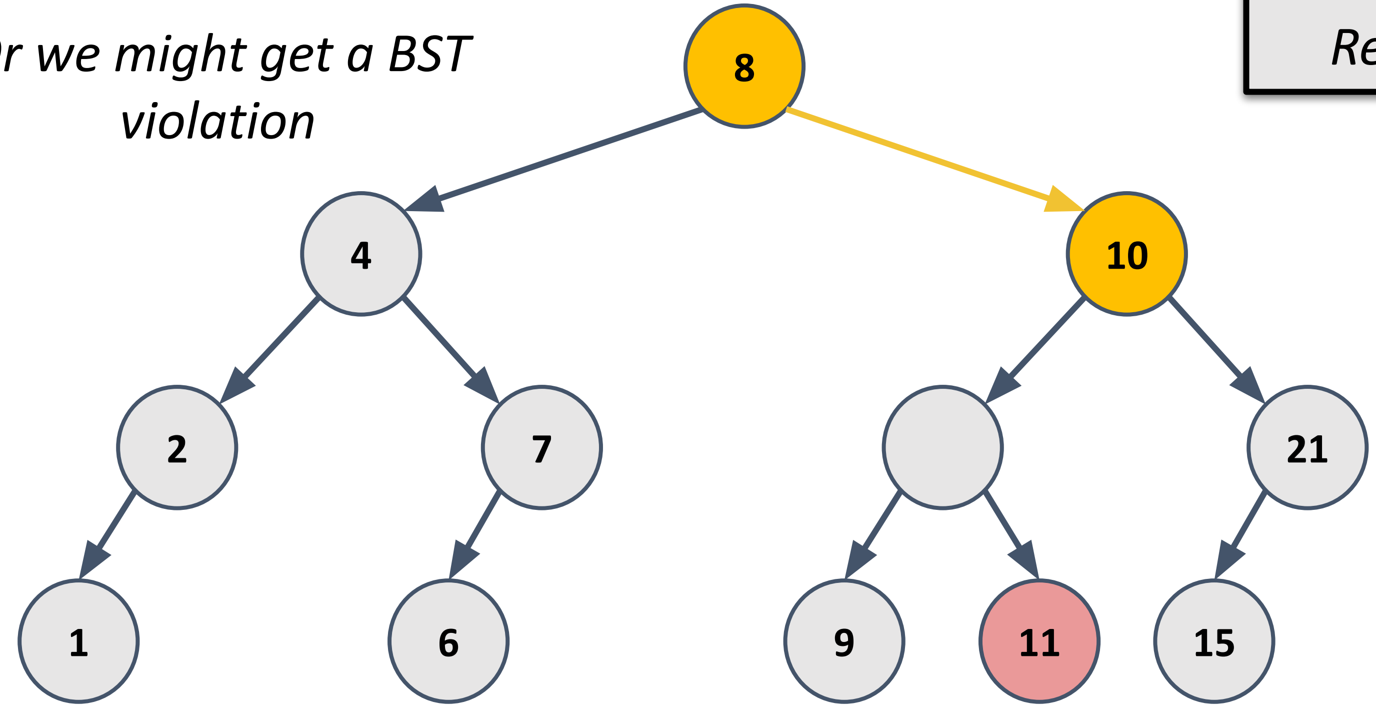
Even trickier: Remove 12



BST Deletion

Or we might get a BST violation

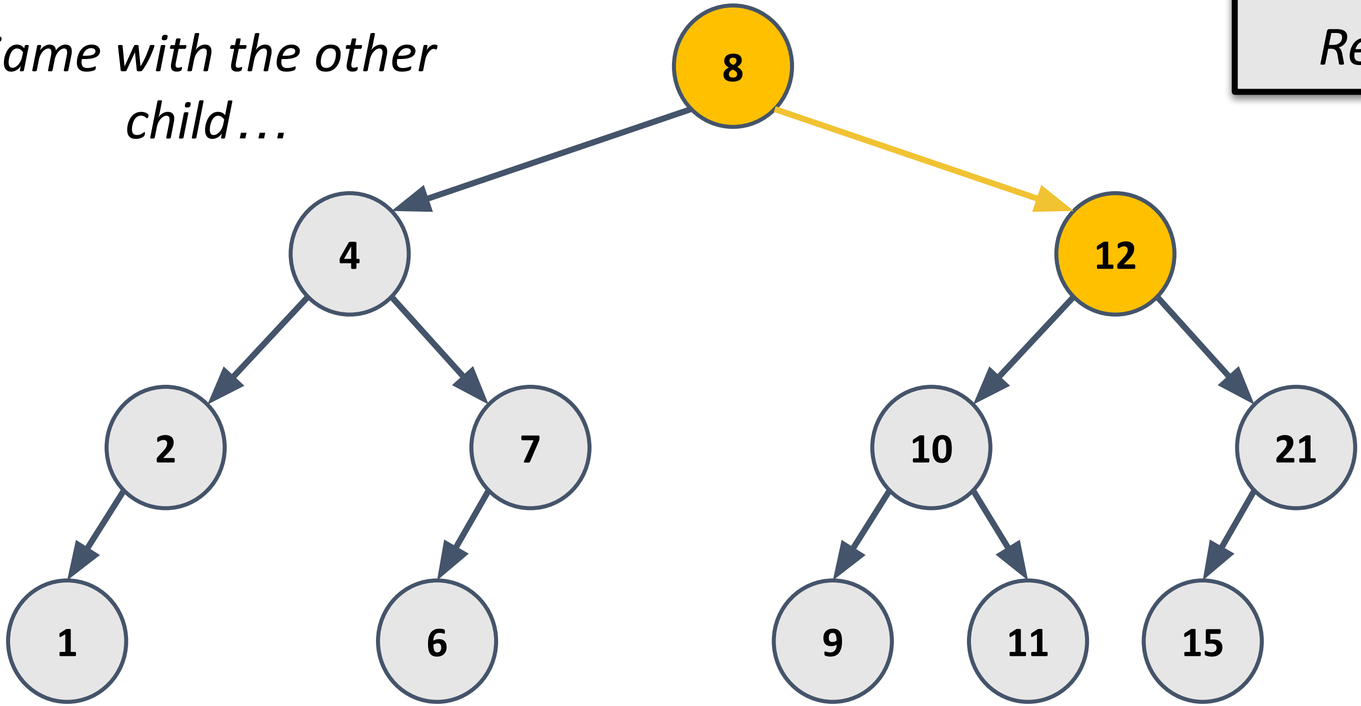
Even trickier: Remove 12



BST Deletion

Same with the other child…

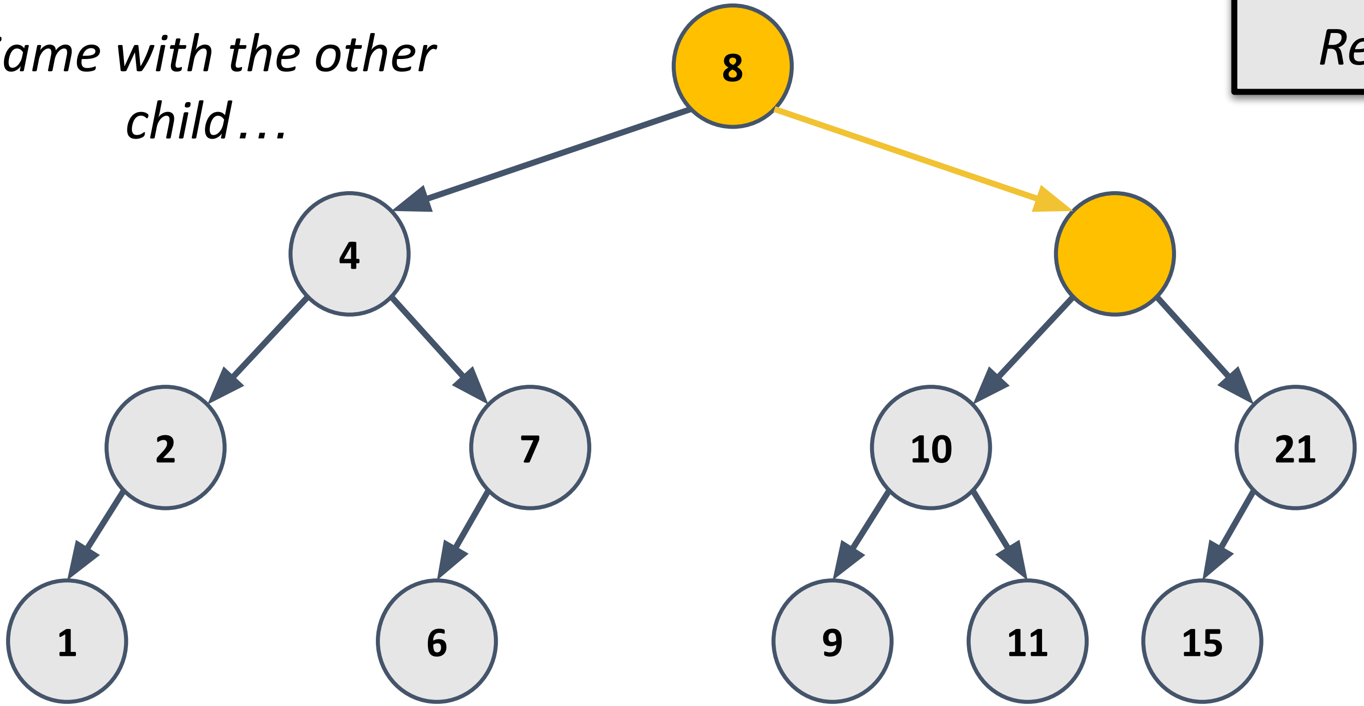
Even trickier: Remove 12



BST Deletion

Same with the other child…

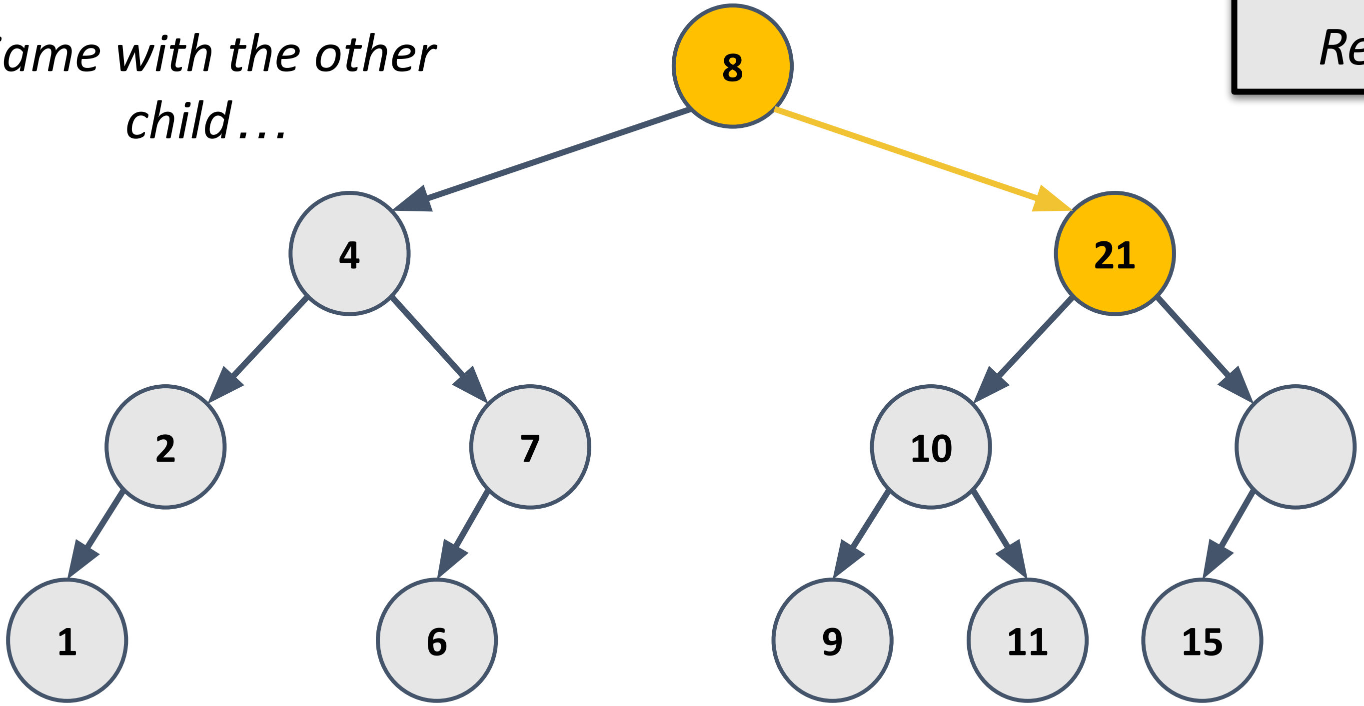
Even trickier: Remove 12



BST Deletion

Same with the other child…

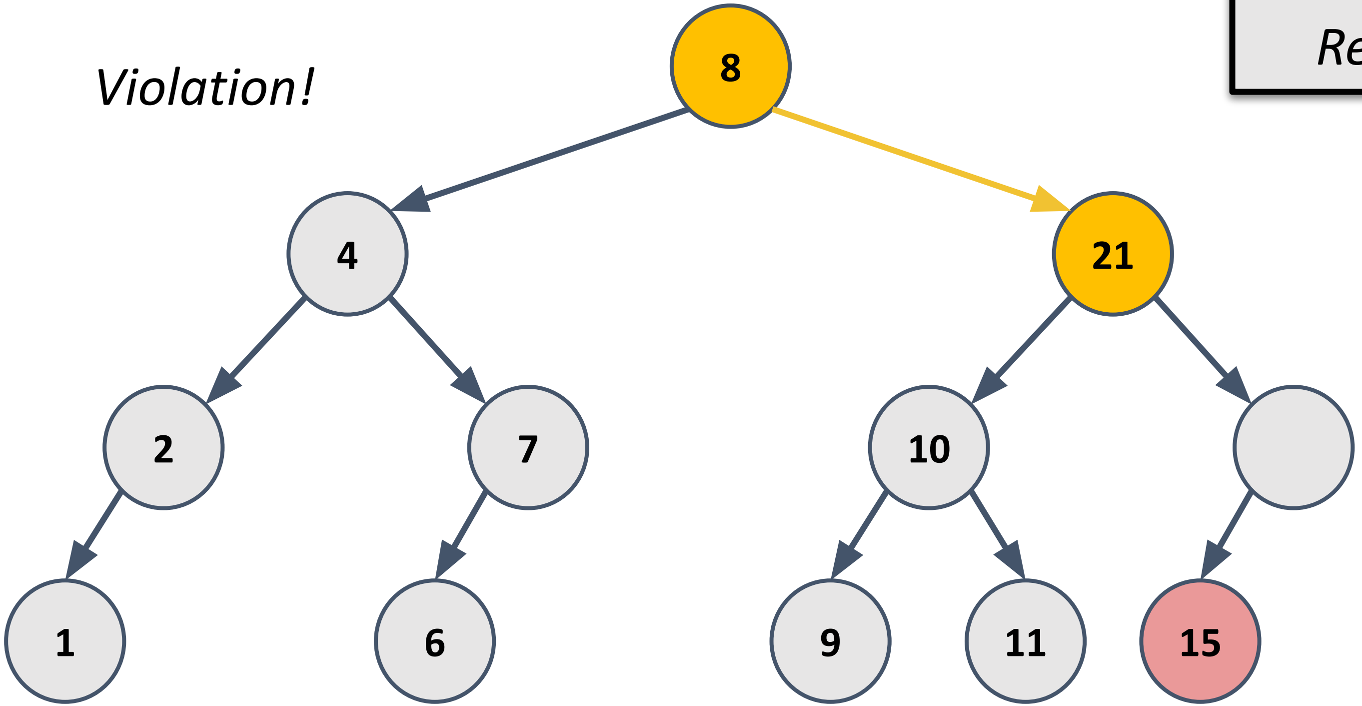
Even trickier: Remove 12



BST Deletion

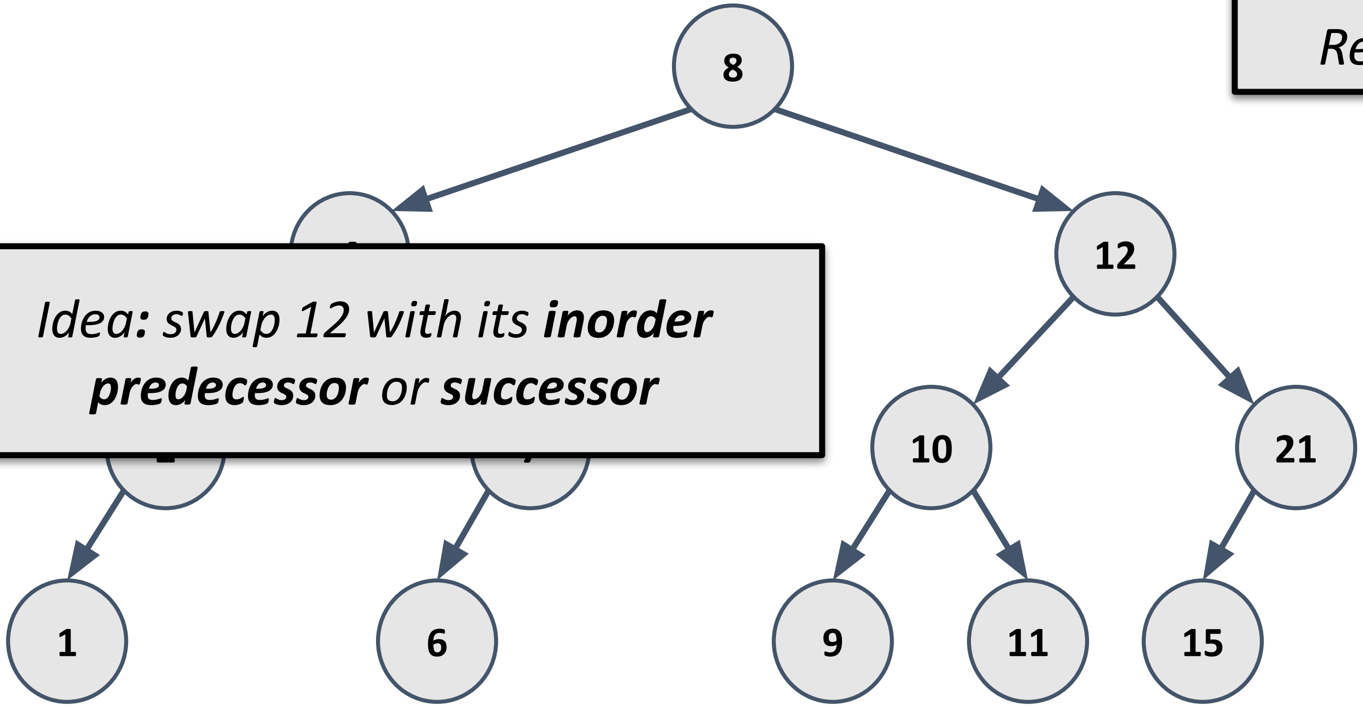
Even trickier: Remove 12

Violation!



BST Deletion

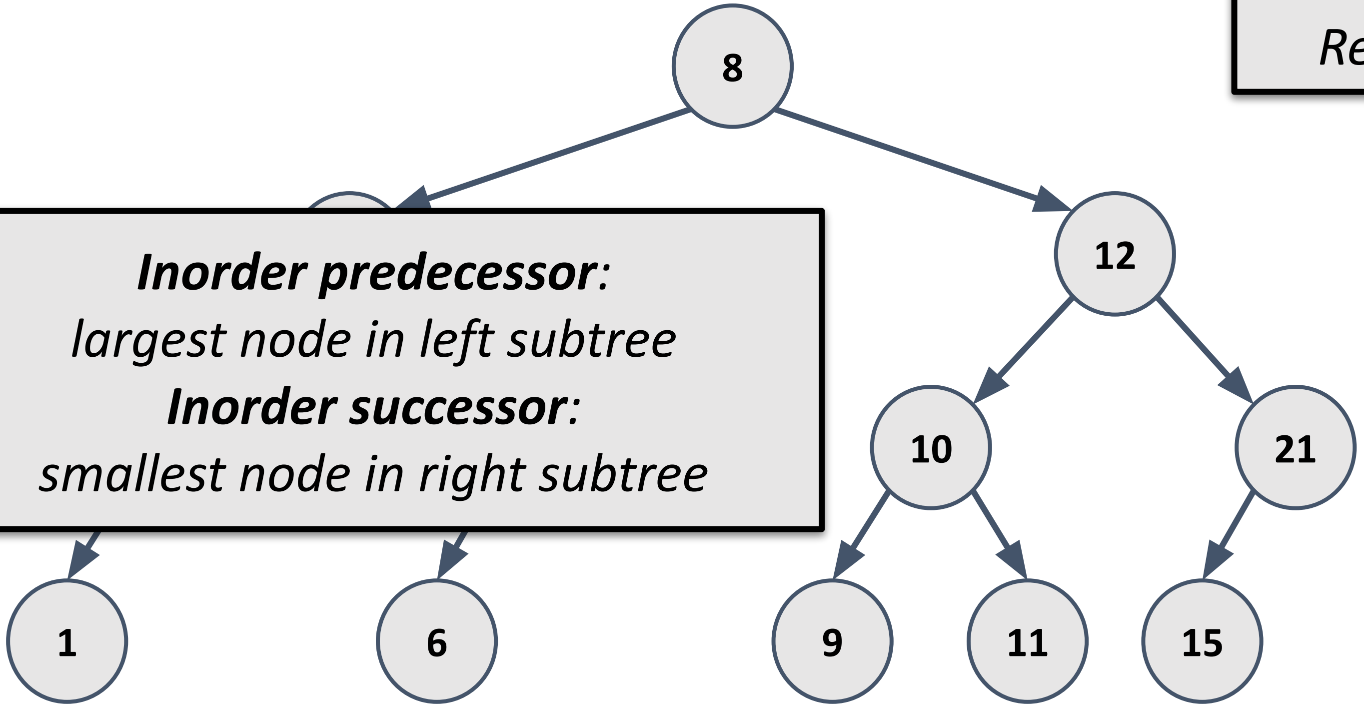
Even trickier: Remove 12



Idea: swap 12 with its ***inorder predecessor*** *or* ***successor***

BST Deletion

Even trickier: Remove 12



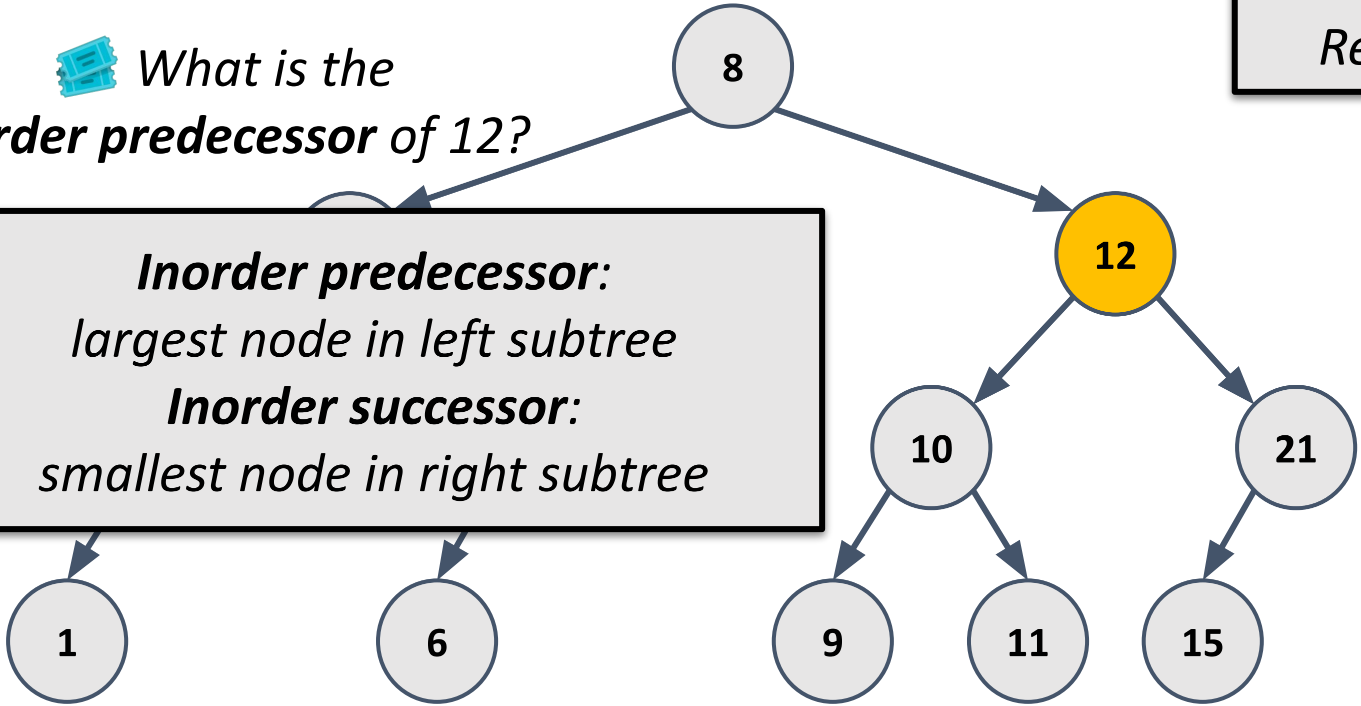
***Inorder predecessor*:**
largest node in left subtree
***Inorder successor*:**
smallest node in right subtree

BST Deletion

Even trickier: Remove 12

What is the ***inorder predecessor*** *of 12?*

Inorder predecessor*:*
largest node in left subtree
Inorder successor*:*
smallest node in right subtree

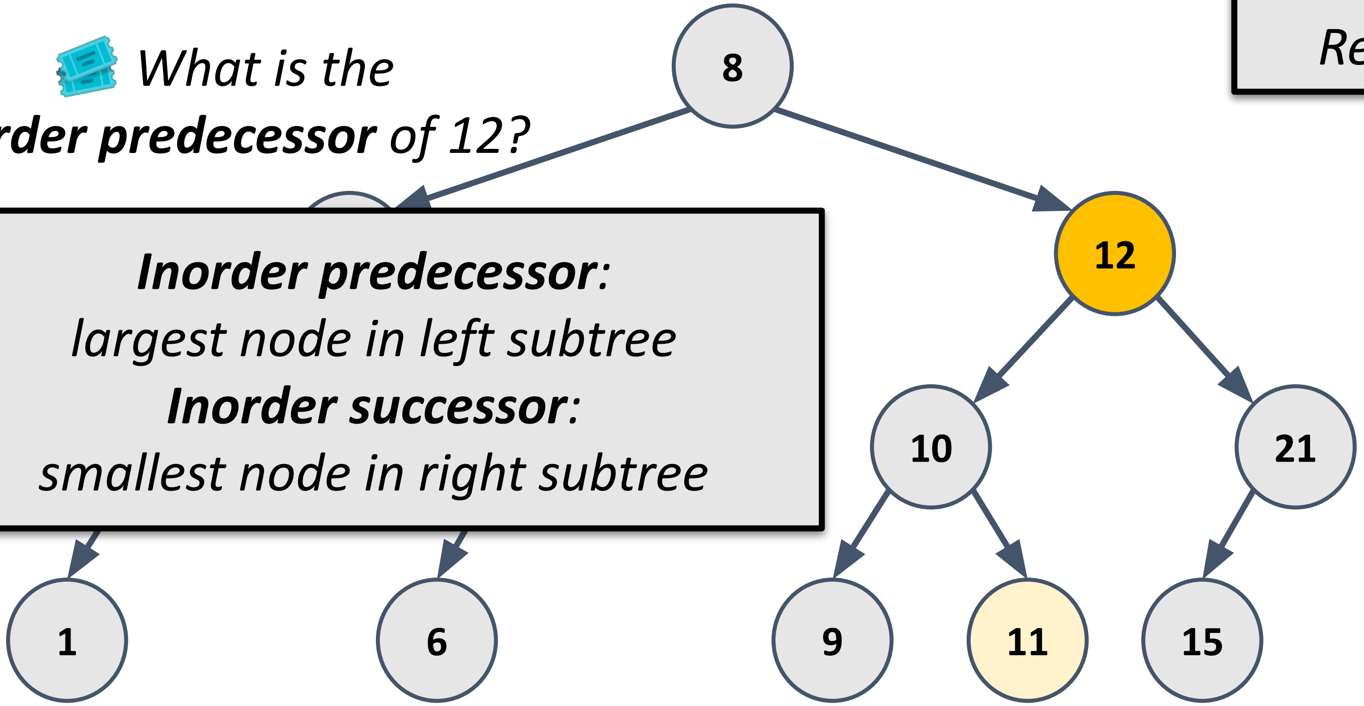


BST Deletion

What is the ***inorder predecessor*** *of 12?*

Even trickier: Remove 12

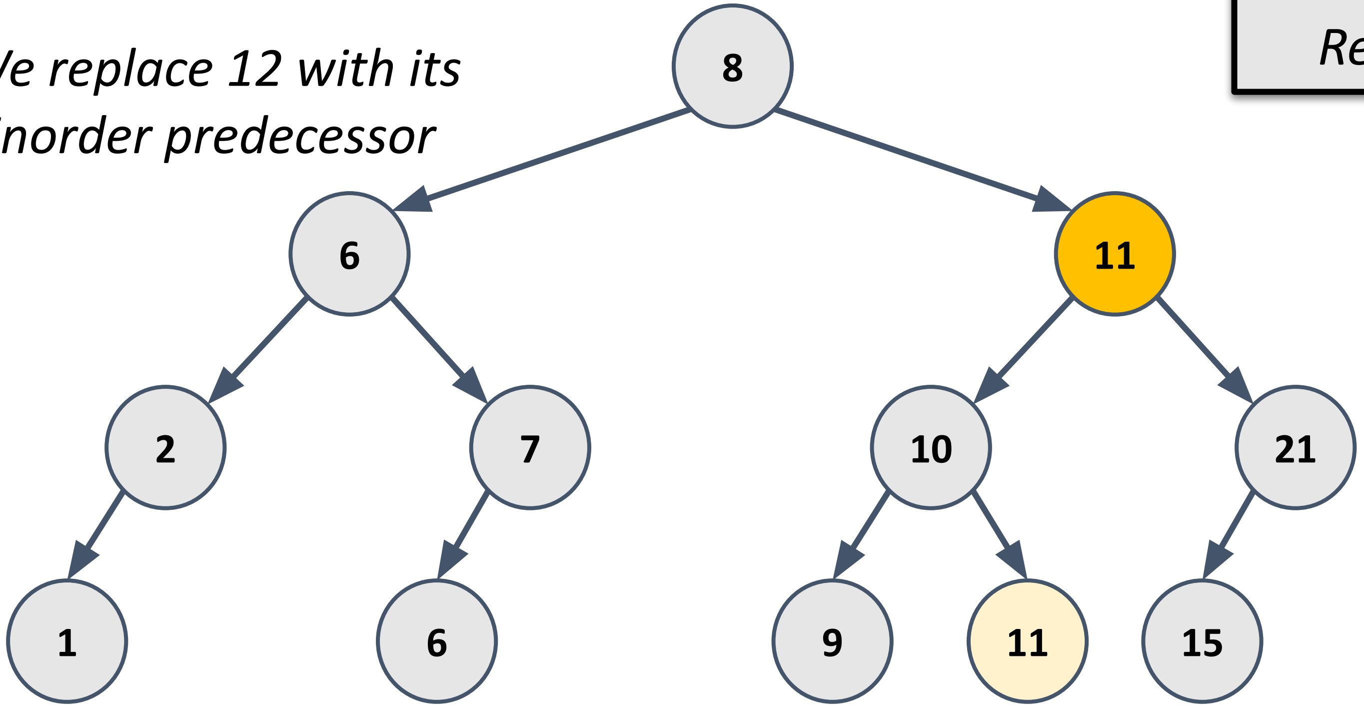
Inorder predecessor: *largest node in left subtree*
Inorder successor: *smallest node in right subtree*



BST Deletion

We replace 12 with its inorder predecessor

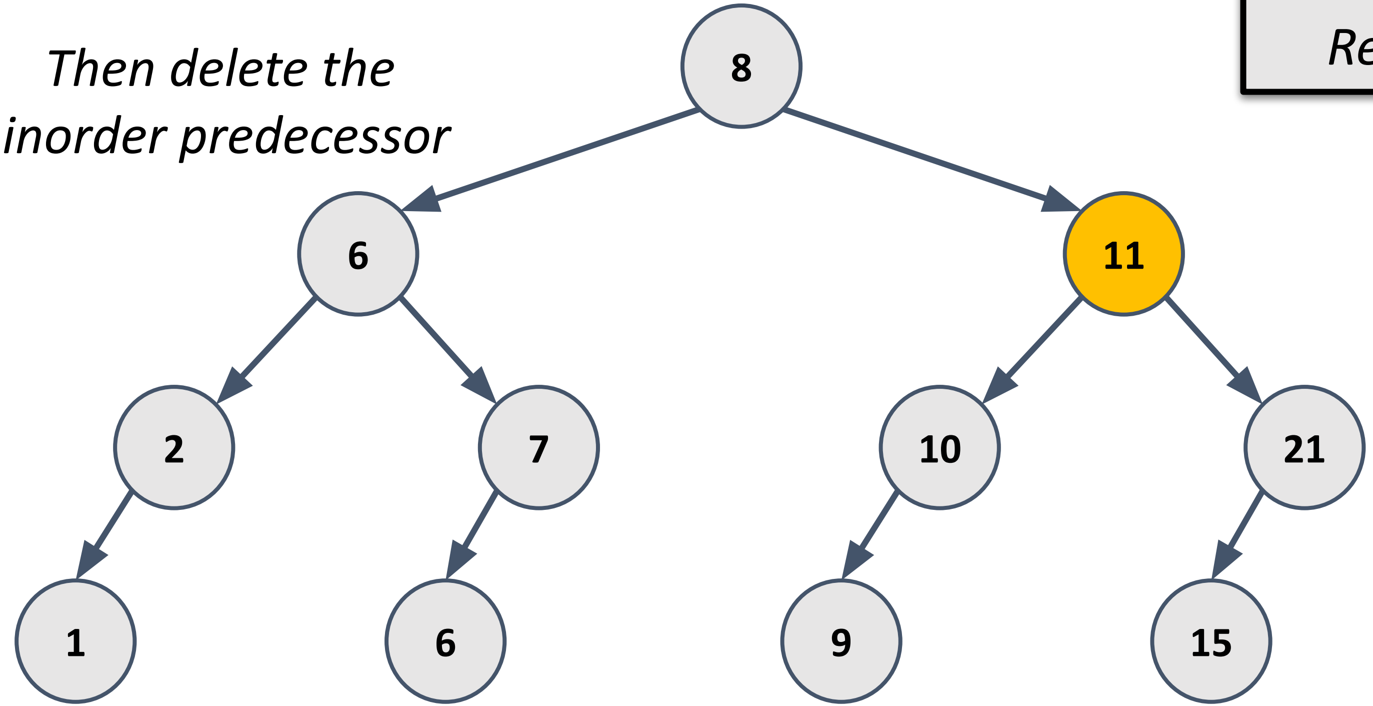
Even trickier: Remove 12



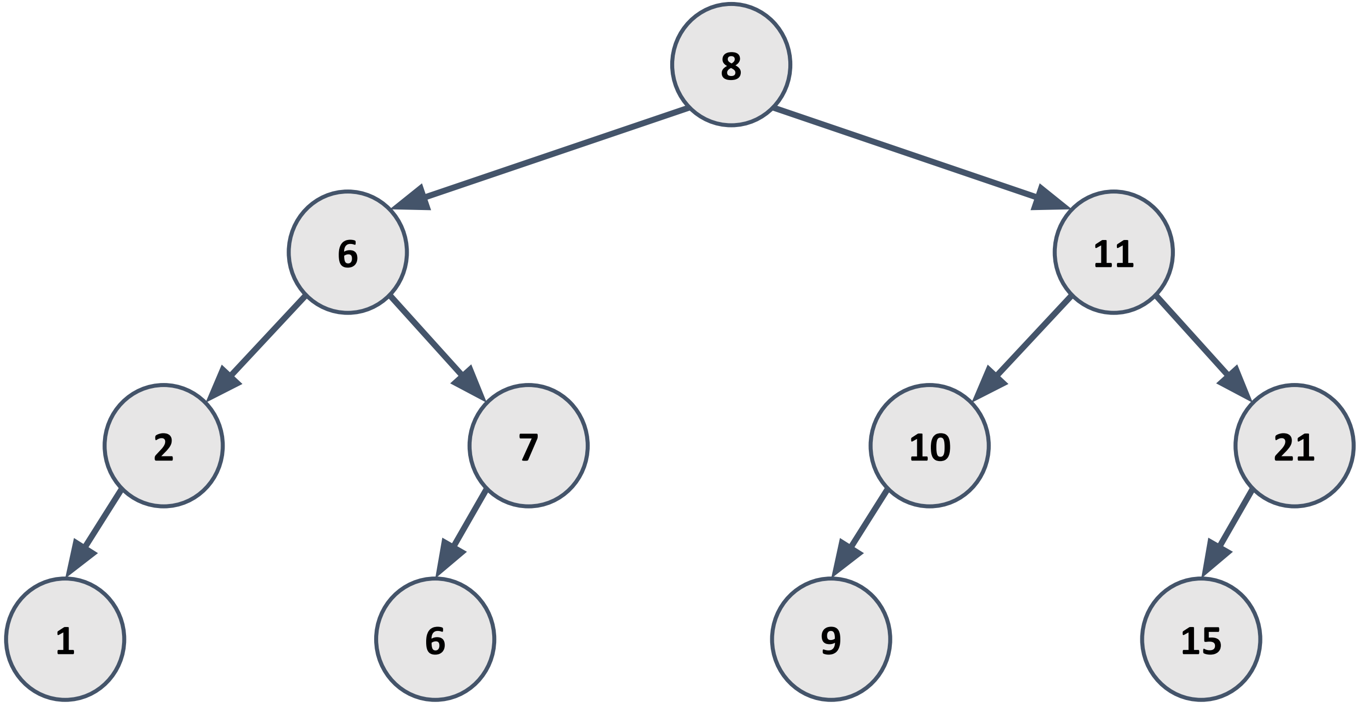
BST Deletion

Then delete the inorder predecessor

Even trickier: Remove 12

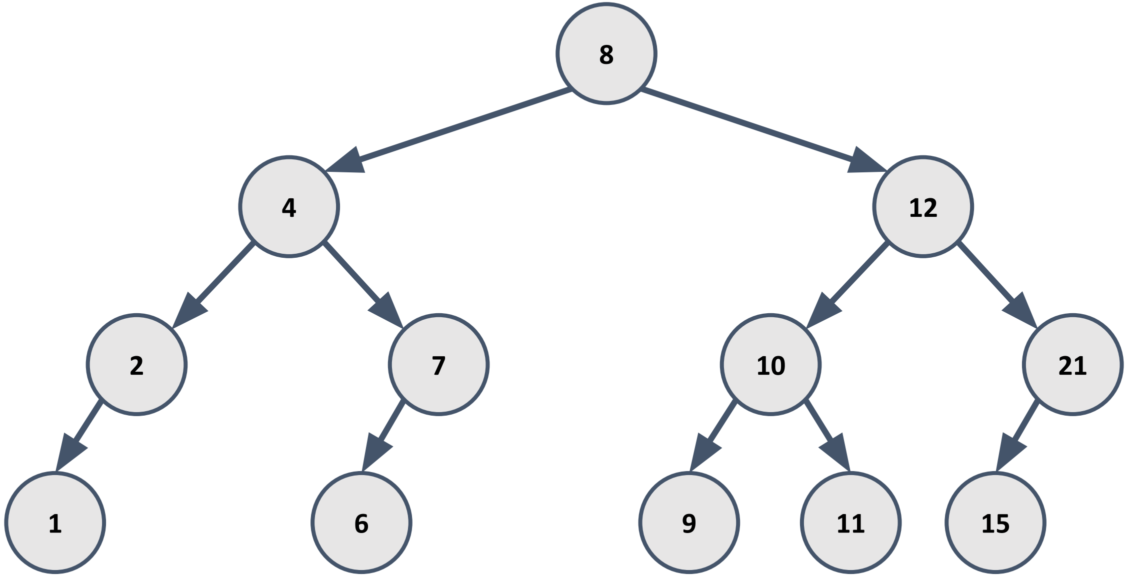


BST Deletion



Takeaways

- To insert/delete nodes, we have to look them up in our BST
  - This is why insertions/deletions are `O(log n)`, just like lookups



Demo: OurSet

Let’s implement a Set using a BST

Implementing OurSet

- We’re going to use a BST to implement a Set
- We’ll create a header file, then implement a few core functions

Implementing OurSet

- We're going to use a BST to implement a Set
- We'll create a header file, then implement a few core functions

```
OurSet set;
set.add(8);
set.add(9);
set.add(4);
```

Implementing OurSet

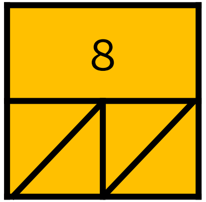
- We're going to use a BST to implement a Set
- We'll create a header file, then implement a few core functions

```
OurSet set;
set.add(8);
set.add(9);
set.add(4);
```

Implementing OurSet

- We're going to use a BST to implement a Set
- We'll create a header file, then implement a few core functions

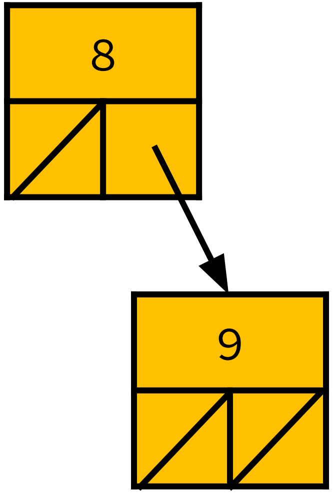
```
OurSet set;
set.add(8);
set.add(9);
set.add(4);
```



Implementing OurSet

- We're going to use a BST to implement a Set
- We'll create a header file, then implement a few core functions

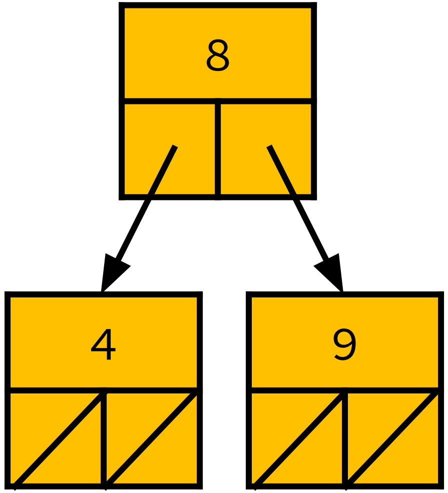
```
OurSet set;
set.add(8);
set.add(9);
set.add(4);
```



Implementing OurSet

- We're going to use a BST to implement a Set
- We'll create a header file, then implement a few core functions

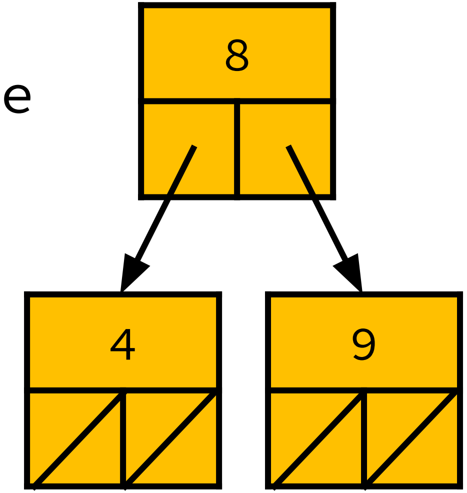
```
OurSet set;
set.add(8);
set.add(9);
set.add(4);
```



Implementing OurSet

- We're going to use a BST to implement a Set
- We'll create a header file, then implement a few core functions

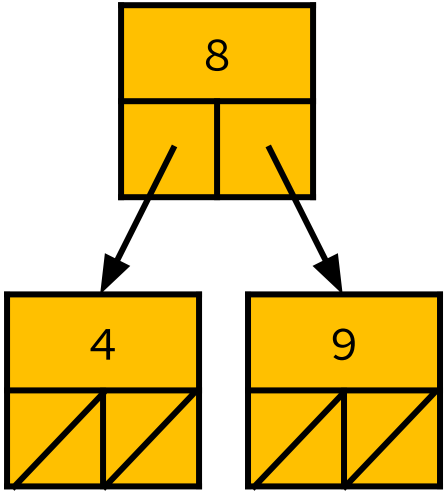
```
set.contains(5); // false
set.contains(4); // true
```



Implementing OurSet

- We're going to use a BST to implement a Set
- We'll create a header file, then implement a few core functions

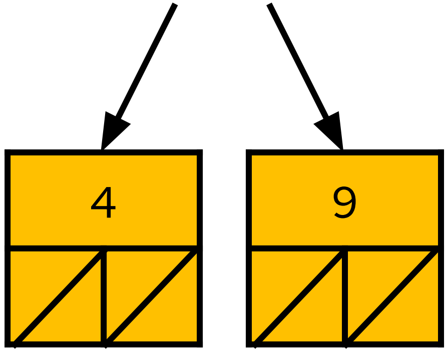
```
set.remove(8);
set.remove(9);
```



Implementing OurSet

- We're going to use a BST to implement a Set
- We'll create a header file, then implement a few core functions

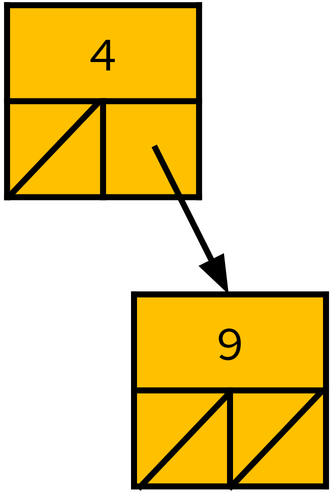
```
set.remove(8);
set.remove(9);
```



Implementing OurSet

- We're going to use a BST to implement a Set
- We'll create a header file, then implement a few core functions

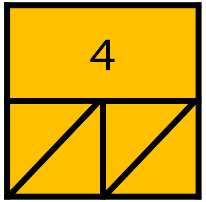
```
set.remove(8);
set.remove(9);
```



Implementing OurSet

- We're going to use a BST to implement a Set
- We'll create a header file, then implement a few core functions

```
set.remove(8);
set.remove(9);
```



The Power of Abstraction

- The client doesn't need to know we're using a BST behind the scenes, they just need to be able to store their data
  - After all, you've used a Set all quarter without needing to know this!

```
OurSet set;
set.add(8);
set.add(9);
set.add(4);
set.contains(5); // false
set.contains(4); // true
set.remove(8);
set.remove(9);
```

OurSet Header

Find solutions in starter code after class

```
class OurSet {
public:
    OurSet(); // constructor
    ~OurSet(); // destructor
    bool contains(int value);
    void add(int value);
    void remove(int value);
    void clear();
    int size();
    bool isEmpty();
    void printSetContents();
private:
    /* To be defined soon! */
};
```

Let’s code it up!

Implement OurSet with a BST

Thank you! 🌳