CS 2100: Data Structures & Algorithms 1

Trees

~ Binary Search Trees ~

Dr. Nada Basit // basit@virginia.edu

Spring 2022

Friendly Reminders

- Masks are **required** at all times during class (University Policy)
- If you forget your mask (or mask is lost/broken), I have a few available
  - Just come up to me at the start of class and ask!
- No eating or drinking in the classroom, please
- Our lectures will be **recorded** (see Collab) – please allow 24-48 hrs to post
- If you feel **unwell**, or think you are, please stay home
  - *We will work with you!*
  - At home: eye mask instead! Get some rest ☺

Announcements / Reminders

- **Reminder of Homework Late Policy:** [Announcement sent 02/14/2022]

  - "Homework 1 (coding)" for each module:
    - Official due date: **Wednesday** by 11:59pm ET
    - Late period (with 10% penalty): 1 week; until the following Wednesday by 11:59pm ET

  - "Homework 2 (analysis)" for each module *[if applicable]*:
    - Official due date: **Friday** by 11:59pm ET
    - Late period (with 10% penalty): 3 days; until following Monday by 11:59pm ET

  - Manage your time wisely, seek help (TAs or Profs) when needed, ***use grace period as your extension*** if need be.

Any Questions about: Preoder, Inorder, Postorder

- In Preorder, the root is visited **before** (pre) the subtrees traversals
- In Inorder, the root is visited **in-between** left and right subtree traversal
- In Postorder, the root is visited **after** (post) the subtrees traversals

Preorder Traversal:
1. Visit the **root**
2. Traverse **left** subtree
3. Traverse **right** subtree

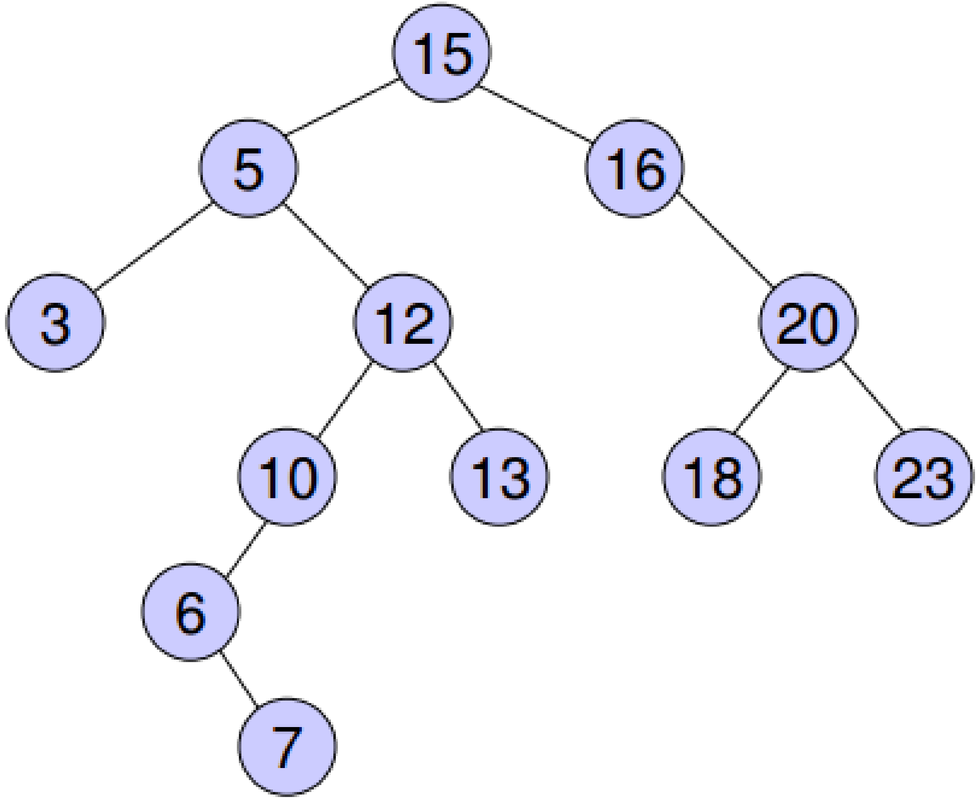
Inorder Traversal:
1. Traverse **left** subtree
2. Visit the **root**
3. Traverse **right** subtree

Postorder Traversal:
1. Traverse **left** subtree
2. Traverse **right** subtree
3. Visit the **root**

Any Questions about:
Tree Traversal Example [*3 methods*]

Let's do an example first…

(Notice: this is a Binary Search Tree!)



- **pre-order:** (root, left, right)

  15, 5, 3, 12, 10, 6, 7,

  13, 16, 20, 18, 23
- **in-order:** (left, root, right)

  3, 5, 6, 7, 10, 12, 13,

  15, 16, 18, 20, 23
- **post-order:** (left, right, root)

  3, 7, 6, 10, 13, 12, 5,

  18, 23, 20, 16, 15

Binary Search Trees: Motivation

- It would be nice to **find/search** for items quickly
  - Want a fast look up time
  - Want to handle inserts and deletes into list
  - Idea: store items in sorted order
- Lists, like `ArrayList` or `LinkedList` aren't ideal
  - If not sorted: O(n) lookup (*Linear search*)
  - If can make use of *Binary Search*: O(log n) lookup
    - Must pay **O(n log n)** to sort beforehand
    - If we insert or remove items, **sort** may become invalid!
- Is there a way to combine what we have been talking about to get the best of both worlds?

Yes…!

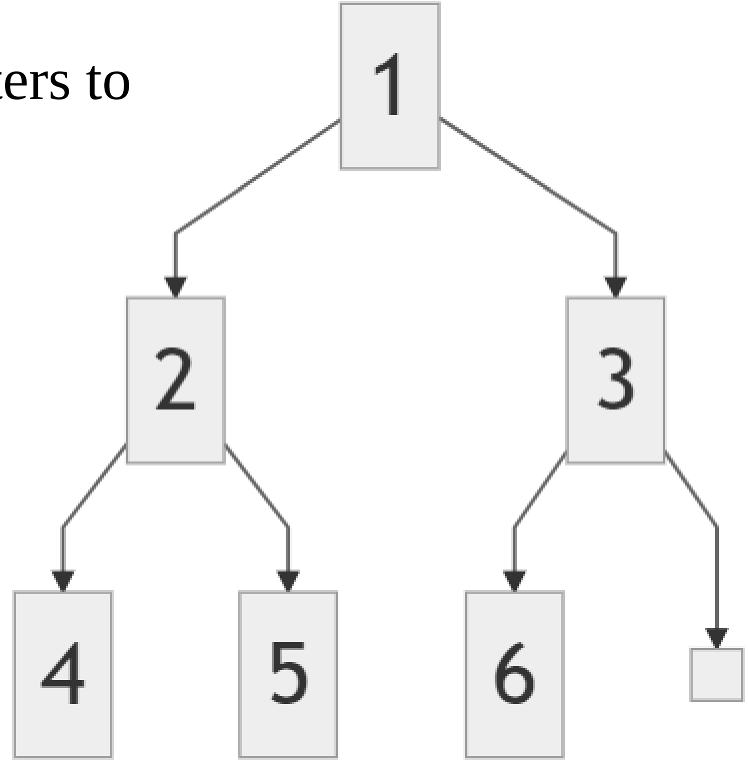
Binary Search Trees

The utility is in the name… ***Facilitating fast SEARCH!***

A Binary Search Tree (BST) is a kind of Binary Tree

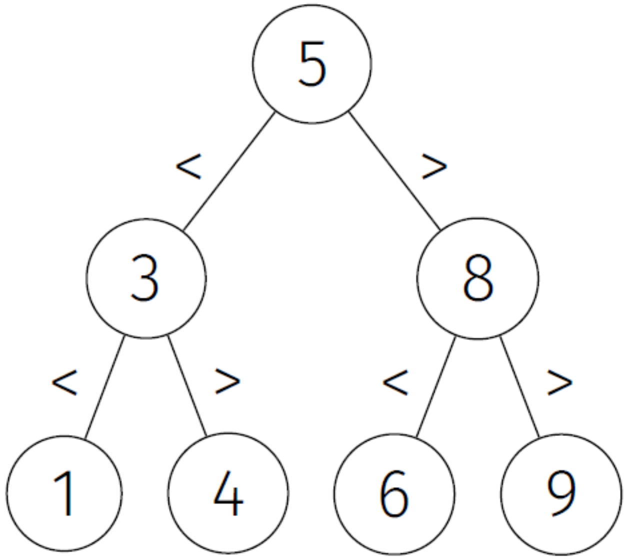
- A Binary tree
  - Maximum 2 children per node
  - Each **node** has a **data** item, e.g. value (or key), and pointers to it's **left** and **right** child **nodes**:

```
public class BinaryNode{
   int value;
   BinaryNode left;
   BinaryNode right;
}
```



  - In reality, any arrow/edge not shown is a **null pointer**.

Binary Search Trees (BSTs)



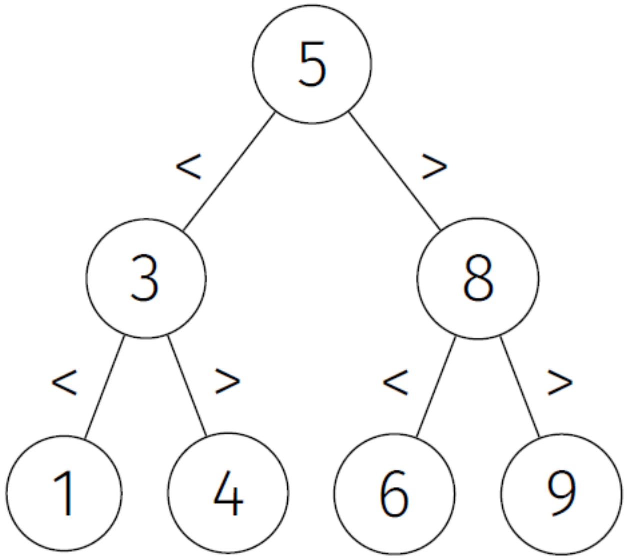
- Each node has a *key* value that can be **compared**

- **Binary Search Tree property:**
  - For a given node, which we will call the **root**...
  - Every node in **left** subtree has a key whose value is **less than** the **root's** key value, AND
  - Every node in **right** subtree has a key whose value is **greater than** the **root's** key value

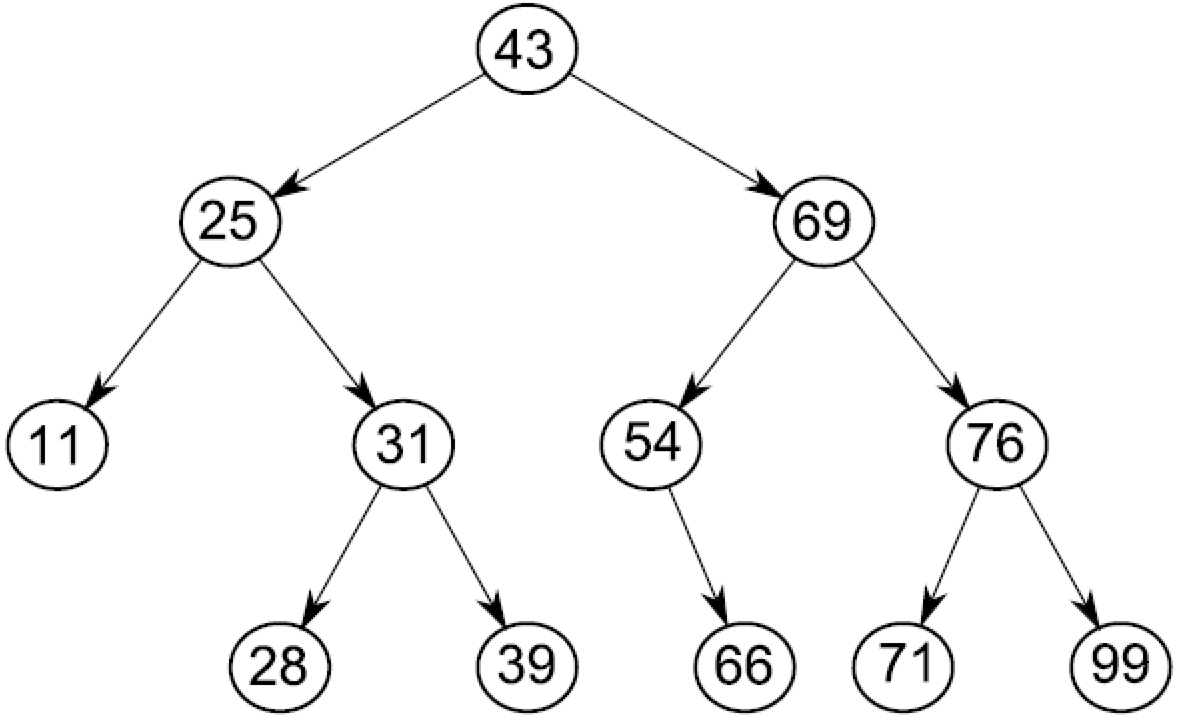
- We assume that duplicate values are not allowed

Binary Search Trees: Cool Property

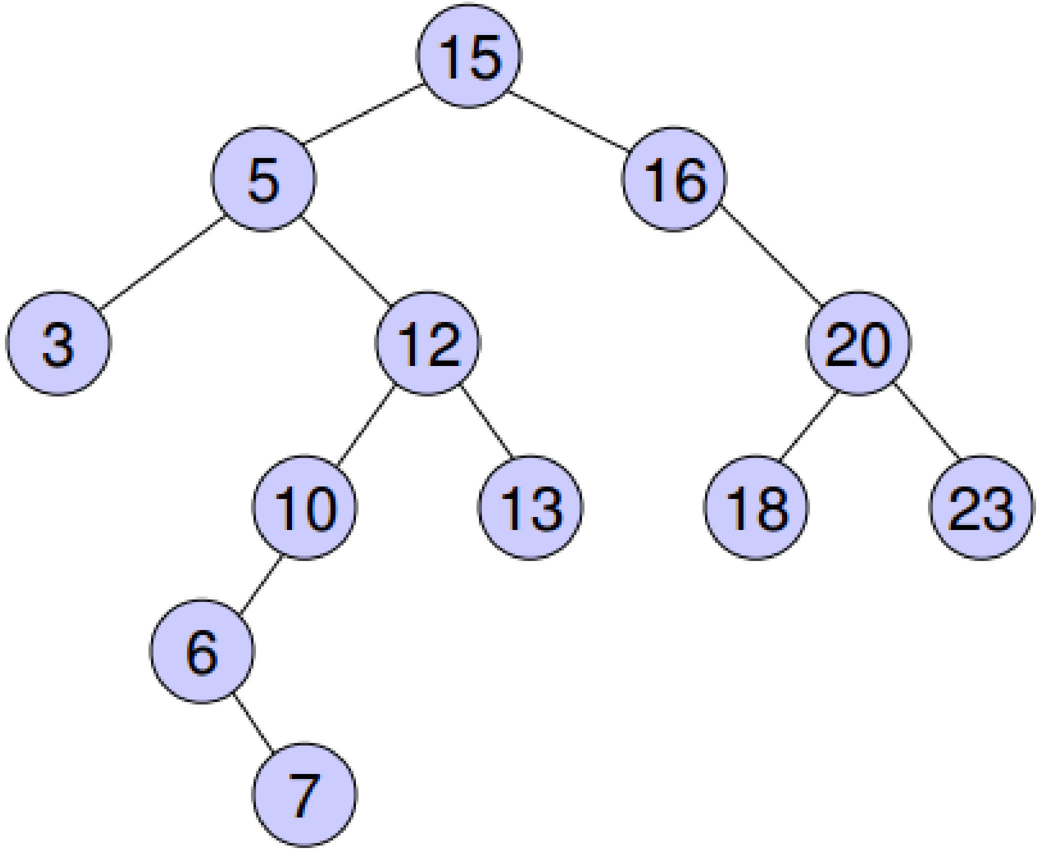
- How could we traverse a BST so that the nodes are visited in **sorted** order?
  - *In-order* traversal: left tree, node, right tree
- It's a very useful property about Binary Search Trees.
- Note: If you perform in-order traversal on a regular Binary Tree (not a BST) then the nodes are **NOT** visited in sorted order!



Example of a Binary Search Tree

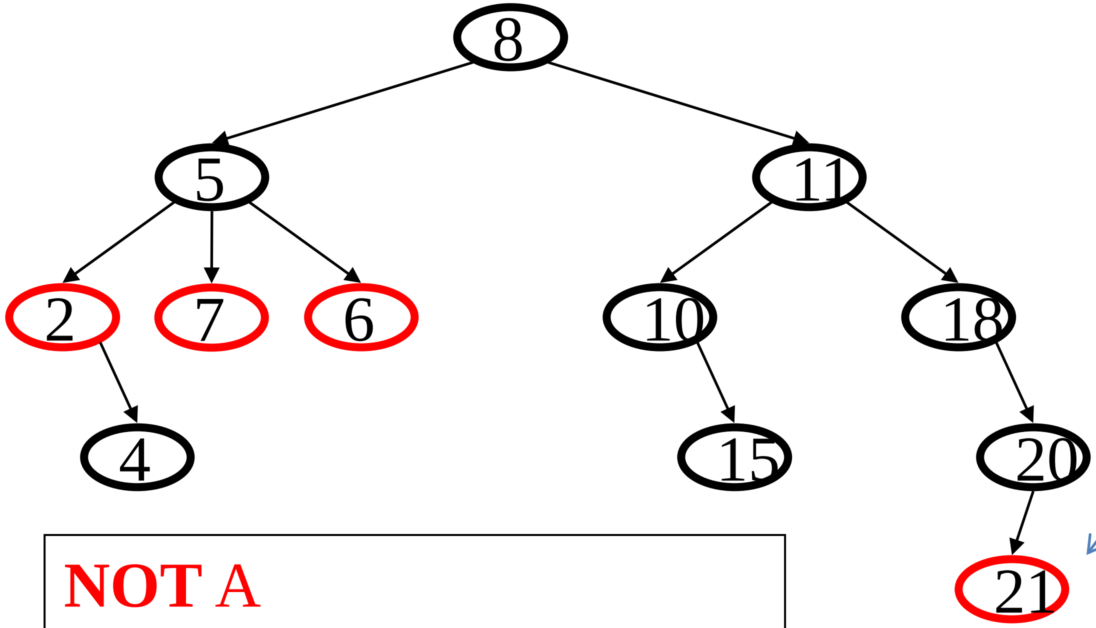


Another example of a Binary Search Tree



Counter-Example (**not** a BST)

Three child nodes

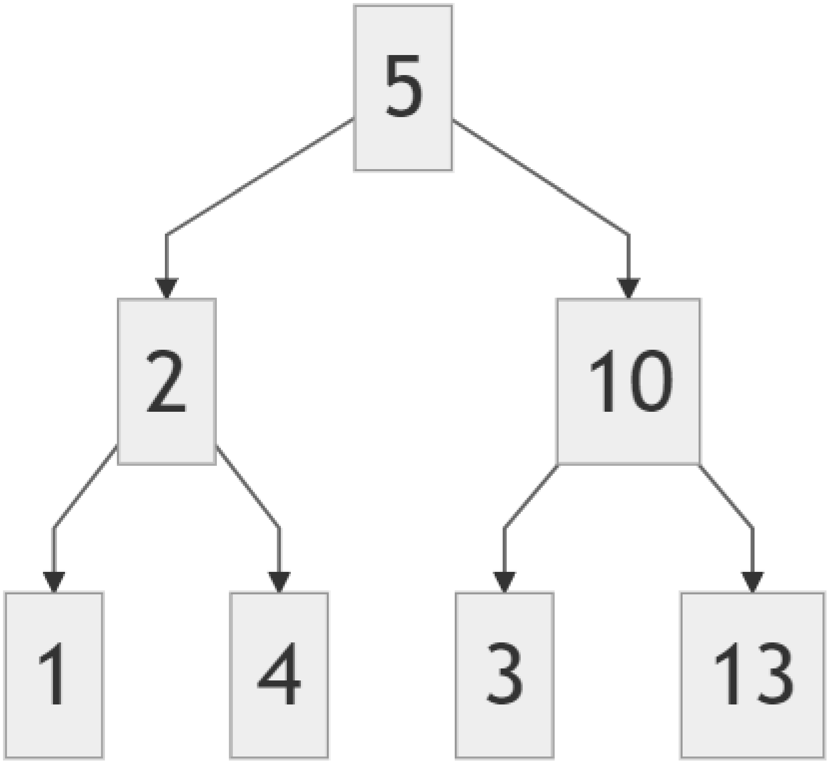


Left node not less than parent node (20)

NOT A
BINARY SEARCH TREE
(in fact, not a binary tree!)

Counter-Example (**not** a BST)

This is a Binary Tree.



Why is this **not** a Binary Search Tree?

The Difference Between Binary Trees and BSTs

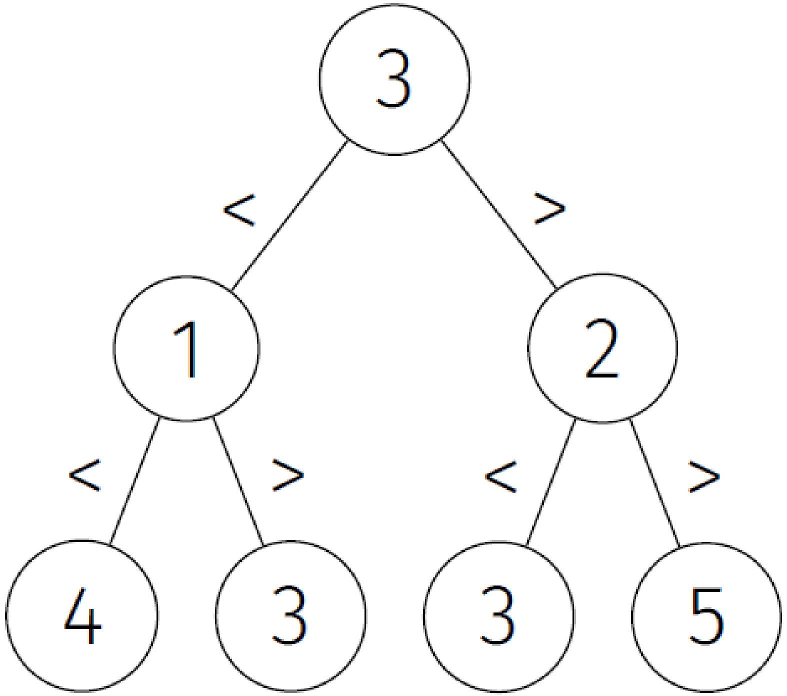
- Both **binary trees** and **binary search trees** have zero, one, or two children per node
- But a binary search tree is ***sorted***
- However, most people, when they say "binary tree", really mean a "**binary search tree**"
- Note that we assume that we can NOT have duplicate elements in a BST

Let's Practice – Can You Identify BSTs?

Are the following trees Binary Search Trees (BSTs) or not?

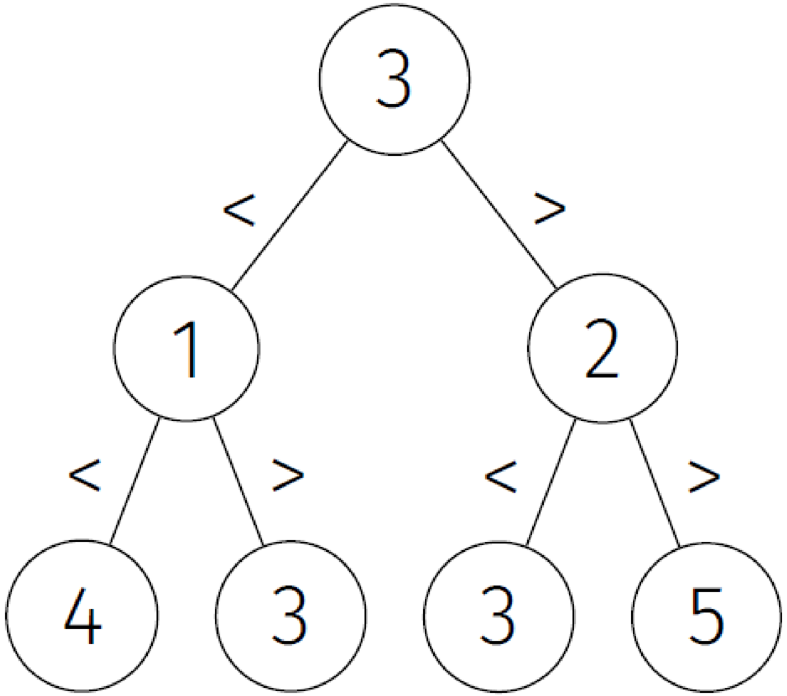
Question!

- Is this a binary search tree?



Question!

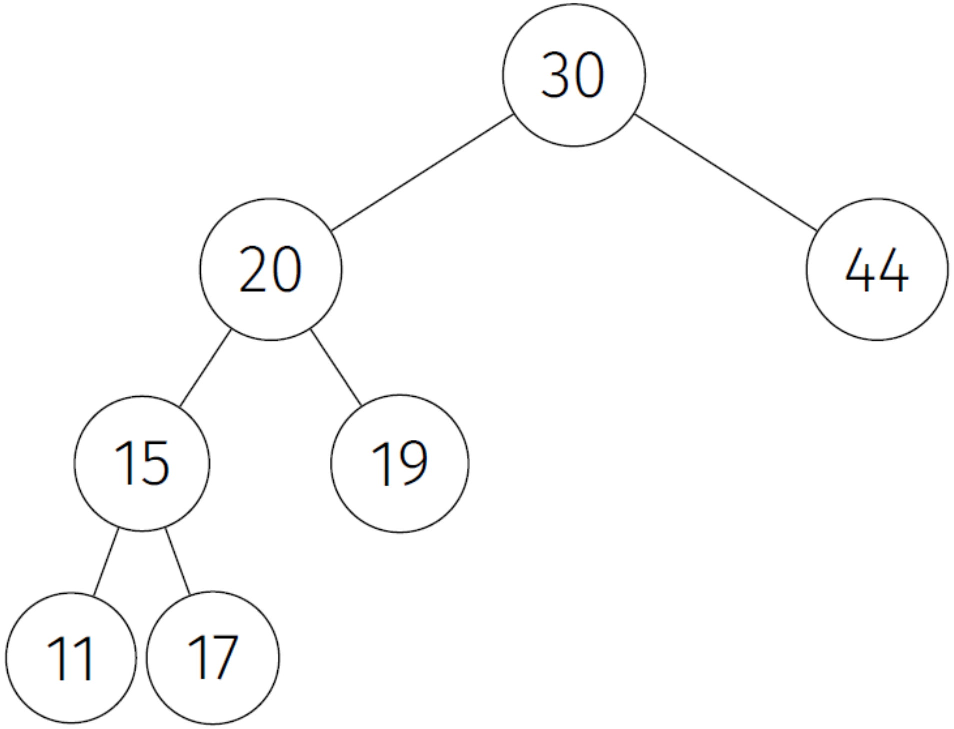
- Is this a binary search tree?



- ***No!*** Binary search tree property not preserved

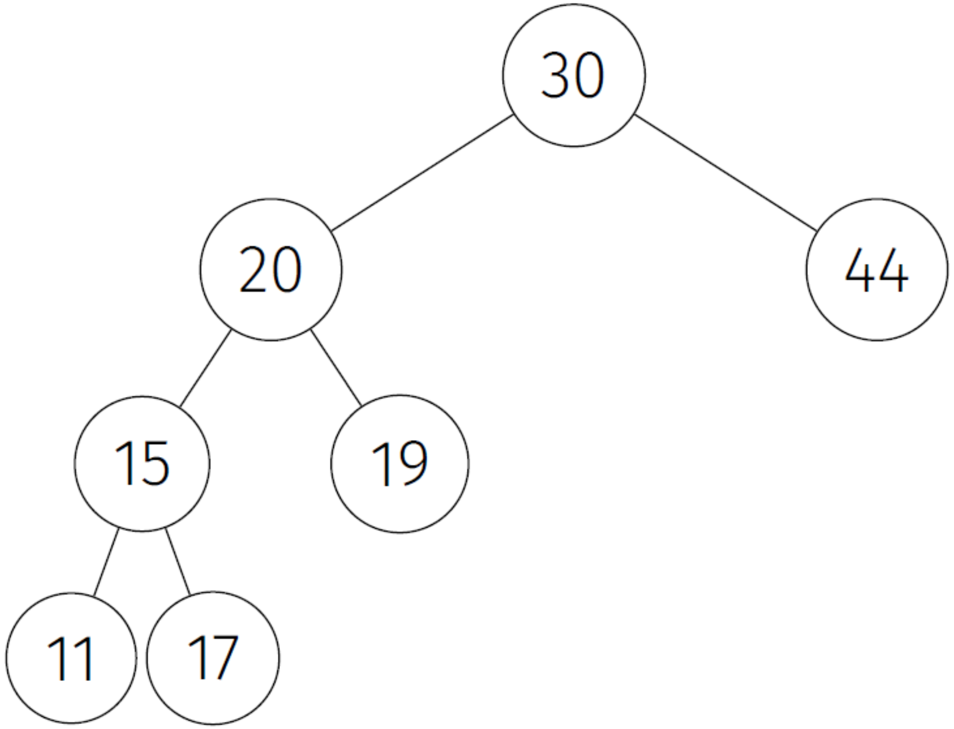
Question!

- Is this a binary search tree?



Question!

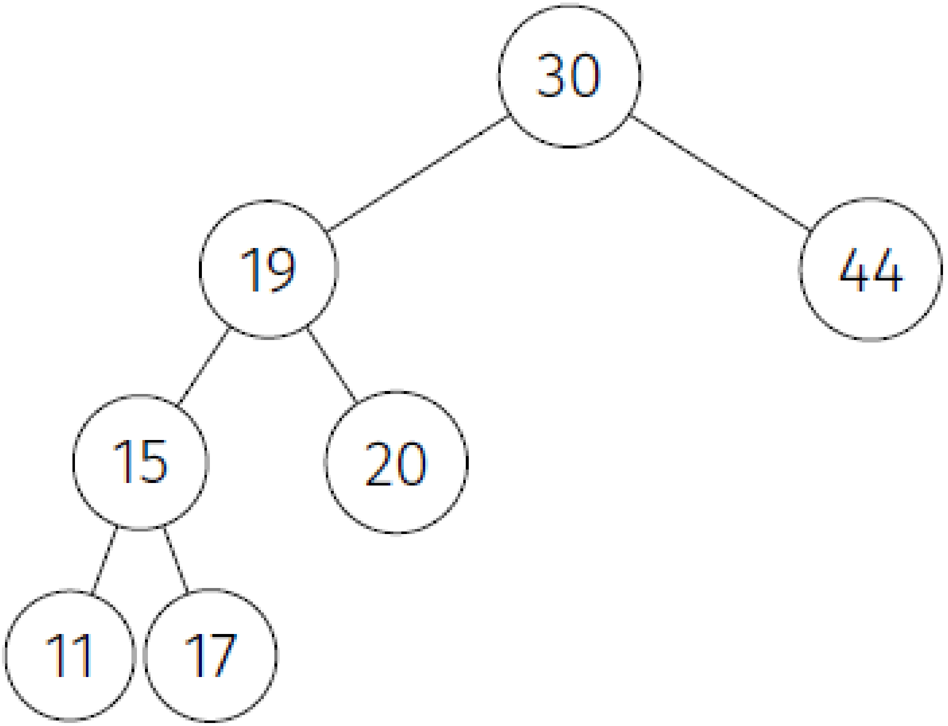
- Is this a binary search tree?



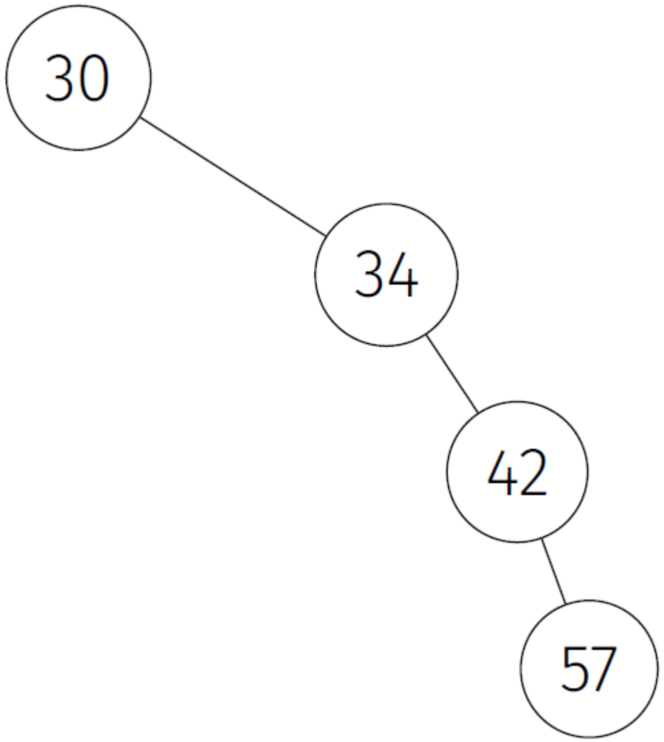
- ***No!*** Binary search tree property not preserved

Question!

- Are these binary search trees?



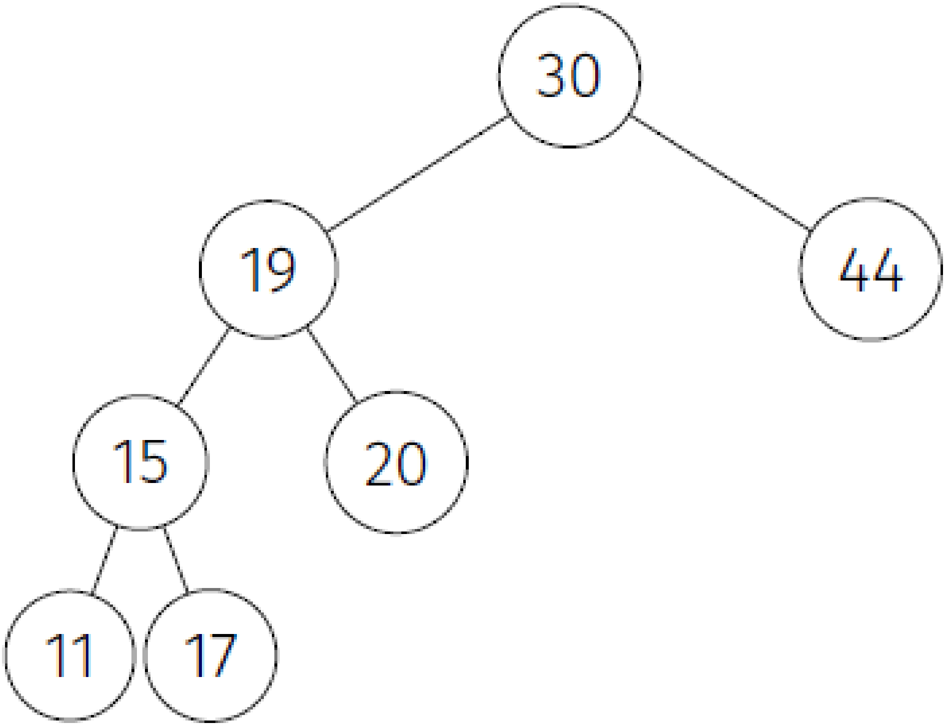
(a)



(b)

Question!

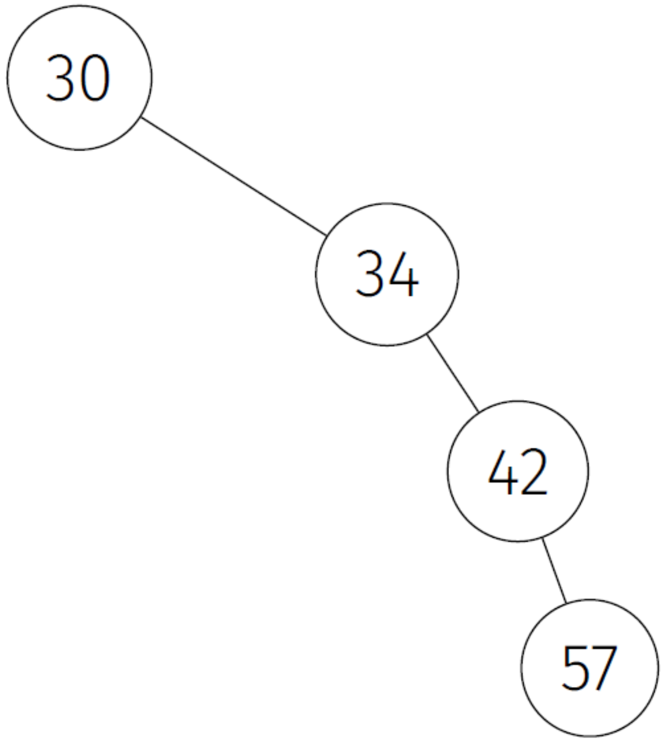
- Are these binary search trees? ***Yes!*** Binary search tree properties are preserved



(a)

Question!

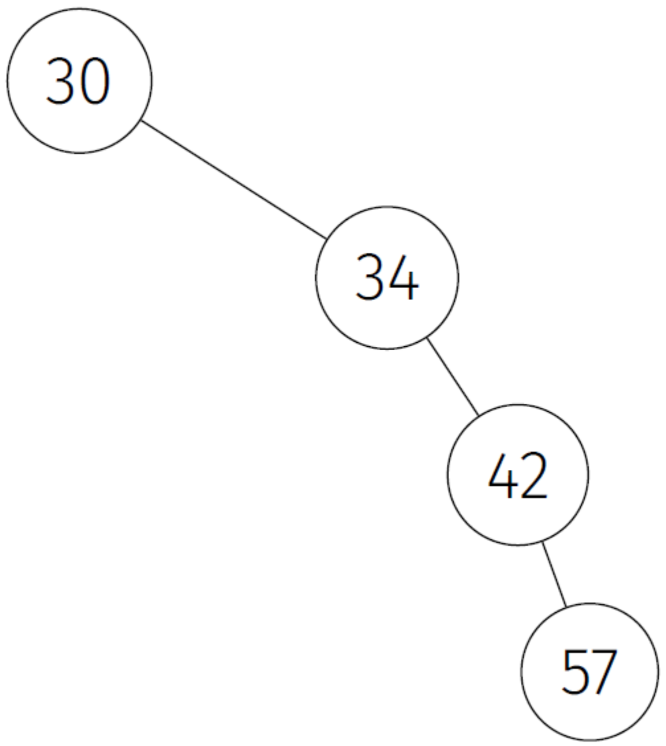
- Are these binary search trees?
  Yes! Binary search tree properties are preserved



(b)

Question!

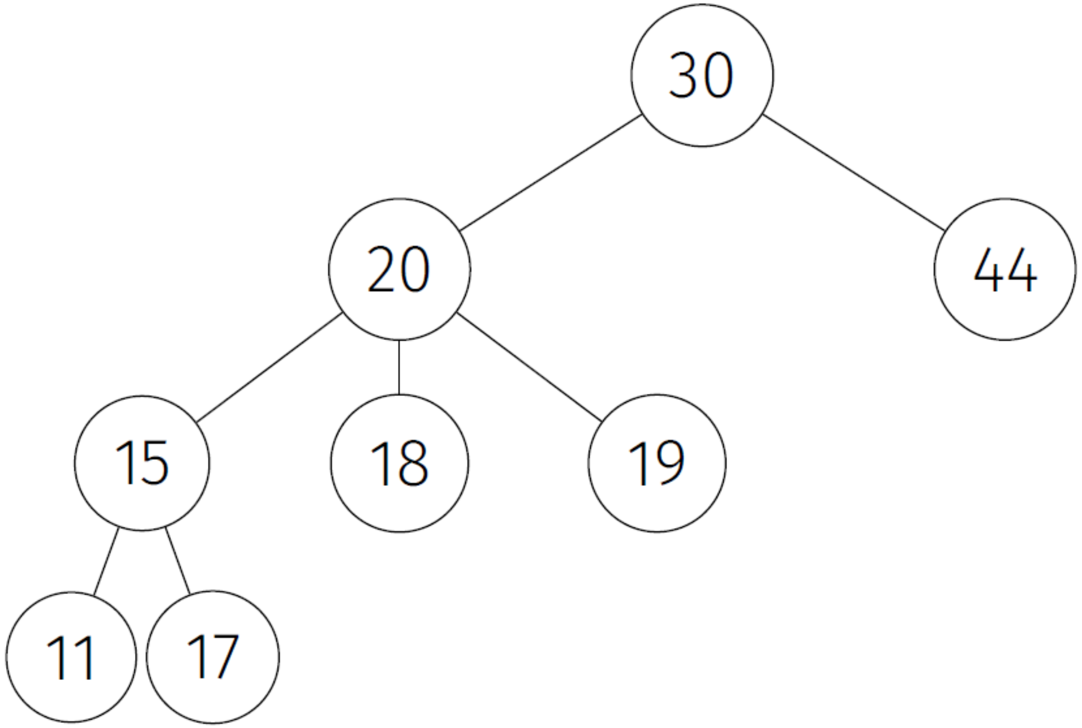
- Are these binary search trees? ***Yes!***
- However, this tree is **unbalanced**!
  - **O(n)** to find 57!
    - essentially *linear!* ☹
  - This is an ordered **list**



- A **balanced** binary tree
  - Guarantees height of child subtrees differ by no more than 1
  - Is better! Produces **O(log n)** runtimes

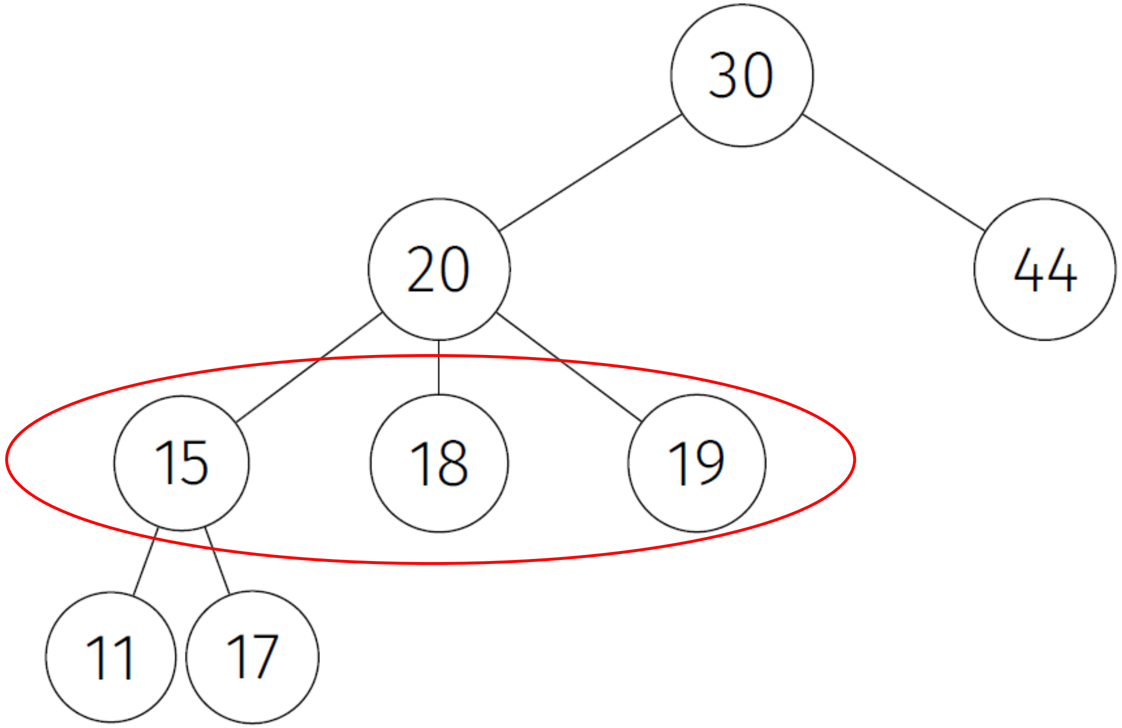
Question!

- Is this a binary search tree?



Question!

- Is this a binary search tree?



- ***No!*** It is not even a binary tree!

BST Operations

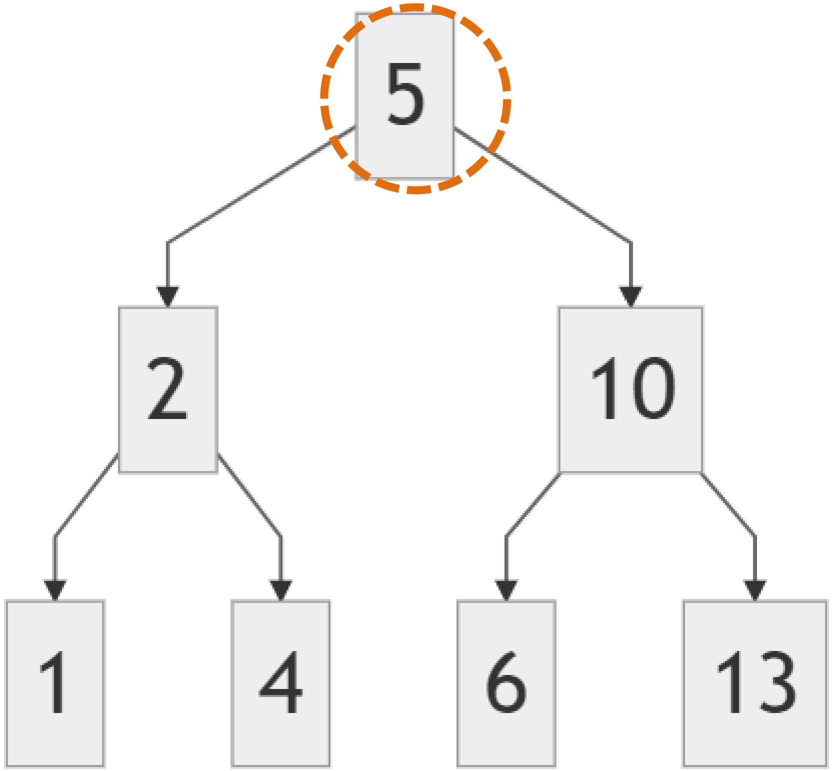
Find and Insert

BST: Find

- Compare **value** to be found to **key** of the root of the tree
  - If they are equal, then **done**
  - If not equal, **recurse** depending on which half of tree the value should be in if it is in the tree
  - If you hit a NULL pointer, then you have "run off" the bottom of the tree, and **the value is not in the tree**

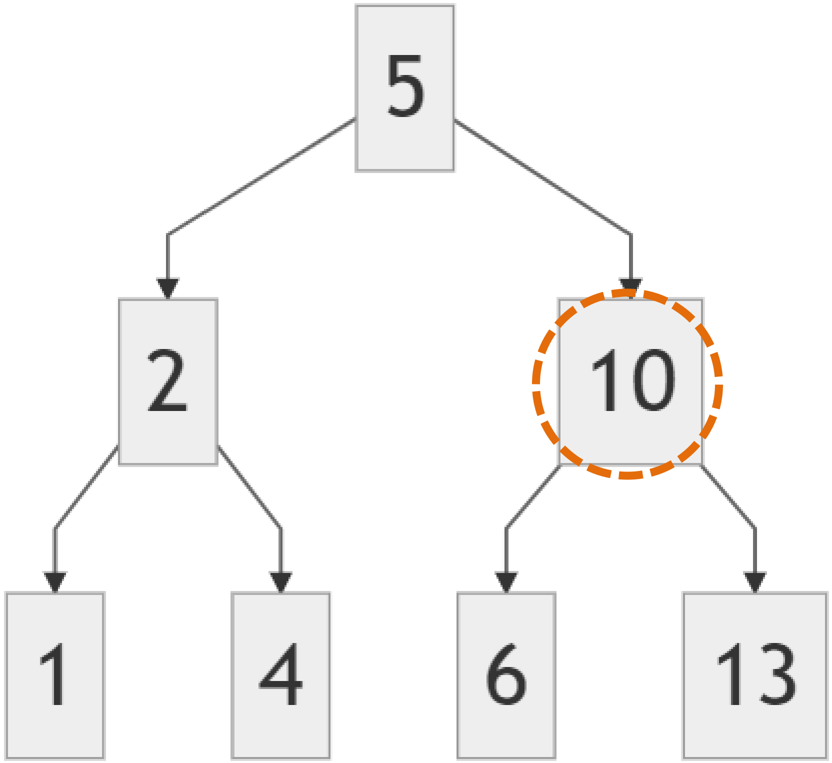
BST: Find Example

- Try to `find(6)`
- Always start at the **root** of the tree!
- 6 is GREATER than 5, go **RIGHT**



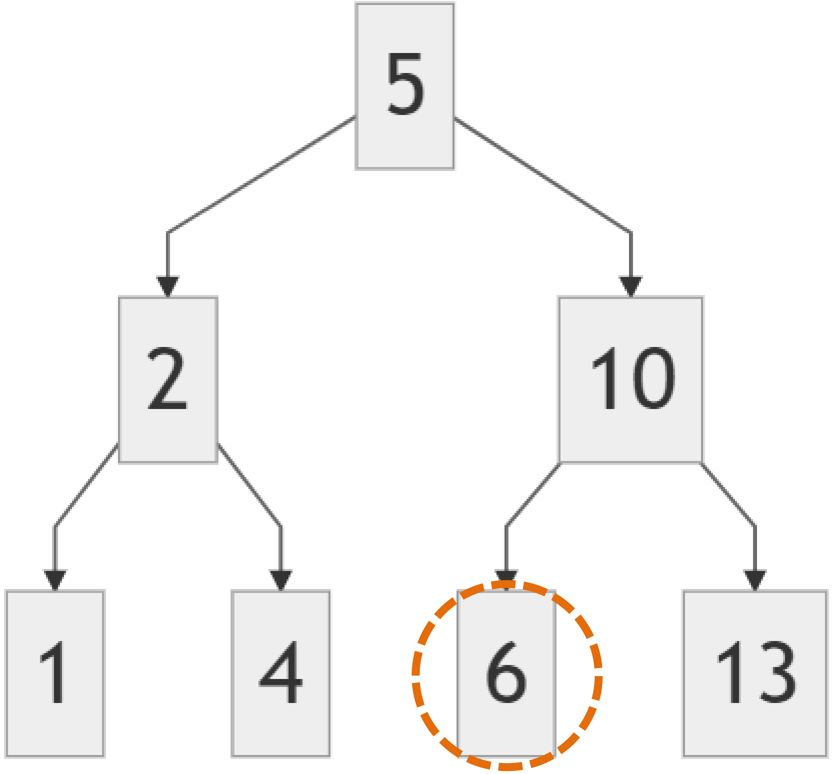
BST: Find Example

- Try to **`find(6)`**
- 6 is LESS than 10, go **LEFT**



BST: Find Example

- Try to **`find(6)`**
- Found it!
- The **value** to be found (6) matches the **key** of the root of the tree (where we are, which is 6)



BST: Find Java Code

- Here is how we might write the **`find()`** method for a Binary Search Tree where the data value is an **int** (very easy to compare)
- It looks fine, but we can do better / make it more general/useful

```
boolean find(int x, BSTNode curNode){
  if(curNode == null) return false; //off end of tree

  else if(x < curNode.value)
    return find(x, curNode.left);

  else if(x > curNode.value)
    return find(x, curNode.right);

  else return true; //found it!
}
```

BST: Find Java Code

- What do we do if you are storing **Objects** in Java? (Complex types; your own Objects…)
- Solution: Use the `compareTo()` method

```
private boolean find(T data, BSTNode< T > curNode) {
  if(curNode == null) return false;

  else if (data.compareTo(curNode.data) < 0)
    return find(data, curNode.left);

  else if (data.compareTo(curNode.data) > 0)
    return find(data, curNode.right);

  else
    return true;
}
```

BST: Find (Final Java Code Solution)

- Programmers using your tree doesn't know what ***curNode*** is…
- **Helper method** hides this (form of abstraction).

```
public boolean find(T data){
  return find(data, rootNode); //start at root of tree
}

private boolean find(T data, BSTNode< T > curNode) {
  if(curNode == null) return false;
  else if (data.compareTo(curNode.data) < 0)
    return find(data, curNode.left);
  else if (data.compareTo(curNode.data) > 0)
    eturn find(data, curNode.right);
  return true;
}
```

compareTo() and the Comparable Interface

Needed Detour!!

Comparable Interface

- **Collections Framework** provides a `Comparable` interface
  - Defines the *natural ordering* of objects of a class

> *"This interface imposes a total ordering on the objects of each class that implements it. This ordering is referred to as the class's **natural ordering**, and the class's **compareTo method** is referred to as its natural comparison method."* – Comparable API

Implementing Comparable

Using *Generics*!

- The `Comparable` **interface** requires only **one** method:
  - `.compareTo(T o)` – compare **this** object to "**o**"
- We must implement the interface and define `T`:

```
public class PhoneBookEntry implements Comparable<PhoneBookEntry> {
    ...
    @Override
    public int compareTo(PhoneBookEntry o) {...}
}
```

Fill in *actual type*!

- Comparable interface is generic, where you must include the type of the class
- The type inside the <> defines **T**

Implementing Comparable ~ fulfilling the contract

- Implement .compareTo(T o) to fulfill the contract

```
public int compareTo(T o) { … }
```

  - Format: `string1.compareTo(string2) //returns an int`

  - Programming convention: **Return value as follows**:
    - **zero** if the same ~ sameness should be same as .equals()
    - **negative value** if first item strictly **less** than second
    - **positive value** if first item strictly **greater** than second

  - We don't care about the actual value

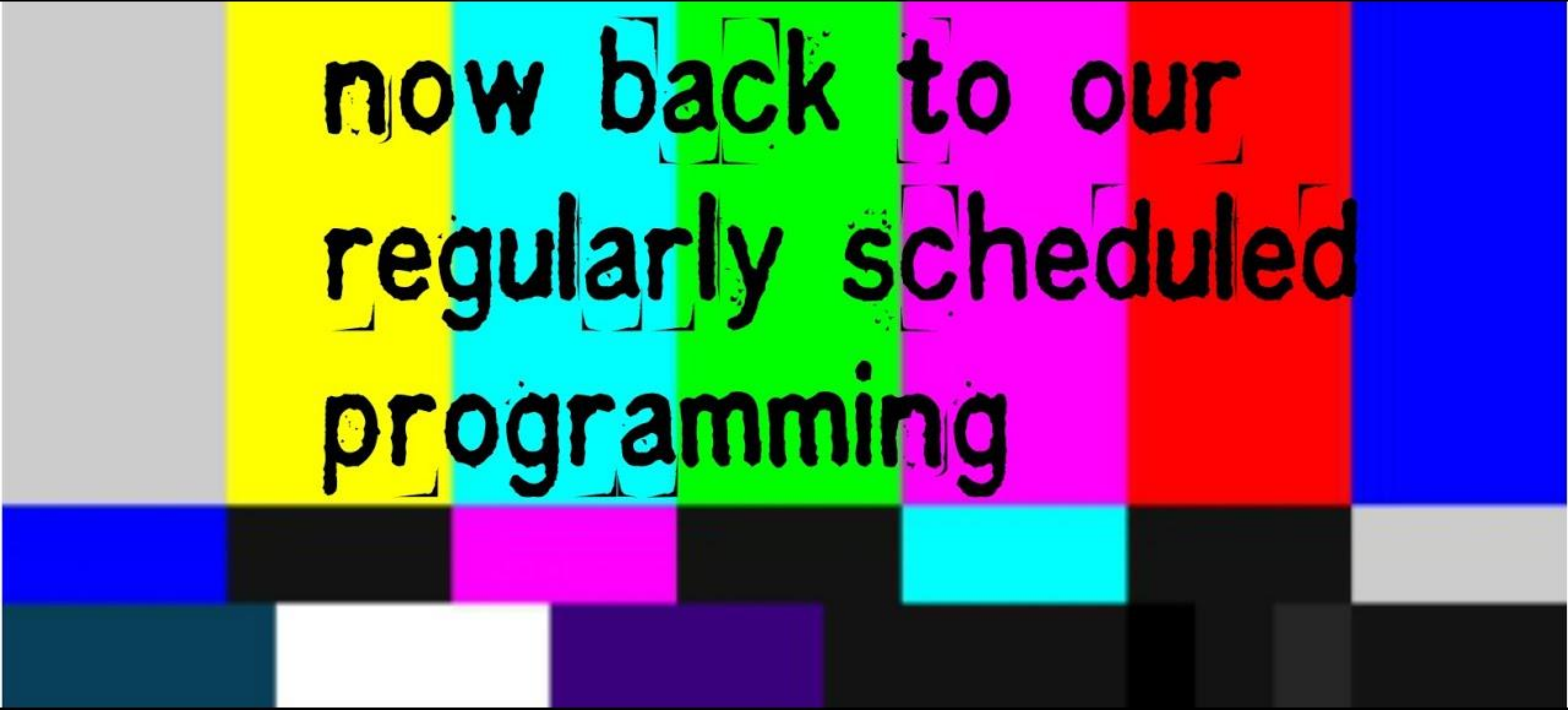
In Order To Store YOUR Items Into A BST…

- If you ever want to put your own objects in **a Binary Search Tree (BST)**, you must:
  1. Make your class implement the Comparable interface
  2. Implement (write) the compareTo() method in your class
- How to write **compareTo()**?
  - Think about state-variables that determine natural order
  - Compare them and return proper-value
  - *What makes one of your objects less-than or greater-than the other?*

Example: To Be Able to Add Students to a BST...

- Student class "*implements*" the Comparable interface: `Comparable<Student>`
- Must fulfil contract: override the compareTo() method stub
- St1.compareTo(St2);
- Body: define the *natural ordering* of the class
- Now that we can say one student is > or < another, we can create a BST of type Student *(otherwise we can't!)*

```java
public class Student implements Comparable<Student> {
    protected String name;
    protected int score;

    public Student (String name, int score) {
        this.name = name;
        this.score = score;
    }

    public String toString() {
        return name + " - " + score;
    }

    @Override
    public int compareTo(Student o) {
        // TODO Auto-generated method stub
        return 0;
    }
```


now back to our
regularly scheduled
programming

BST: Insert (very similar to Find!)

- **Find** an element in the tree
  - Compare with root, if less traverse left, else traverse right; repeat
  - Stops when found or at a leaf
  - Sounds like **binary search**!
  - Time complexity: **O(log n)**, worst case **height** of the tree
- **Insert** a new element into the tree
  - Easy! Do a **find** operation. At the leaf node, add it!
  - Remember: add it to the correct side (left or right)

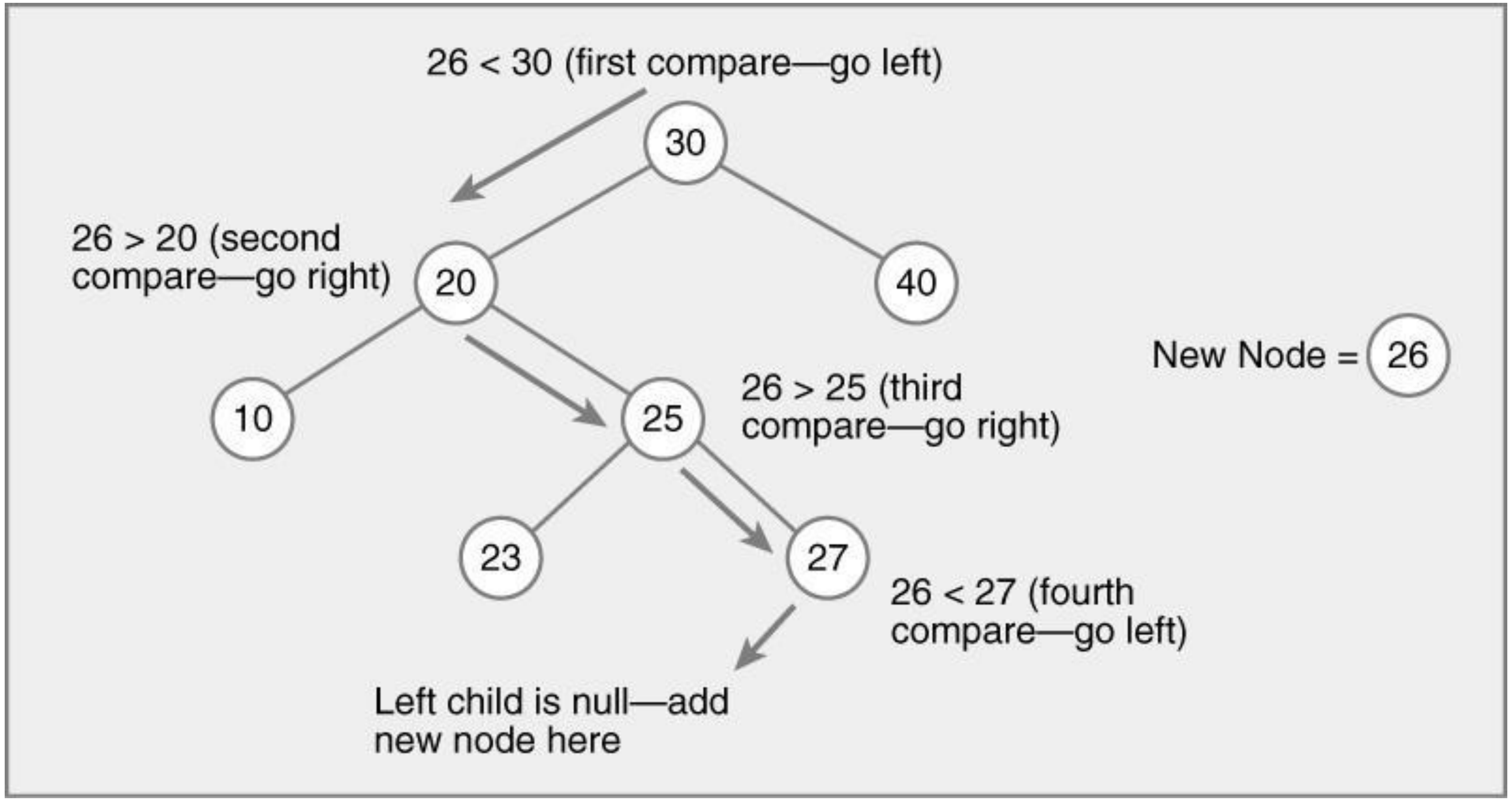
BST: Insert (Final Java Code Solution)

- Idea: Move down the tree like in the find() method to discover location
  - Make and put the new node when you encounter a null subtree

```
public void insert(T data) {
  this.root = insert(data, root);
}

private BSTNode< T > insert(T data, BSTNode< T > curNode) {
  if(curNode == null) return new BSTNode< T >(data);

  else if (data.compareTo(curNode.data) < 0)
    curNode.left = insert(data, curNode.left);

  else if (data.compareTo(curNode.data) > 0)
    curNode.right = insert(data, curNode.right);

  else ; //duplicate, ignoring the insert
  return curNode; //curNode still the root of this subtree
}
```

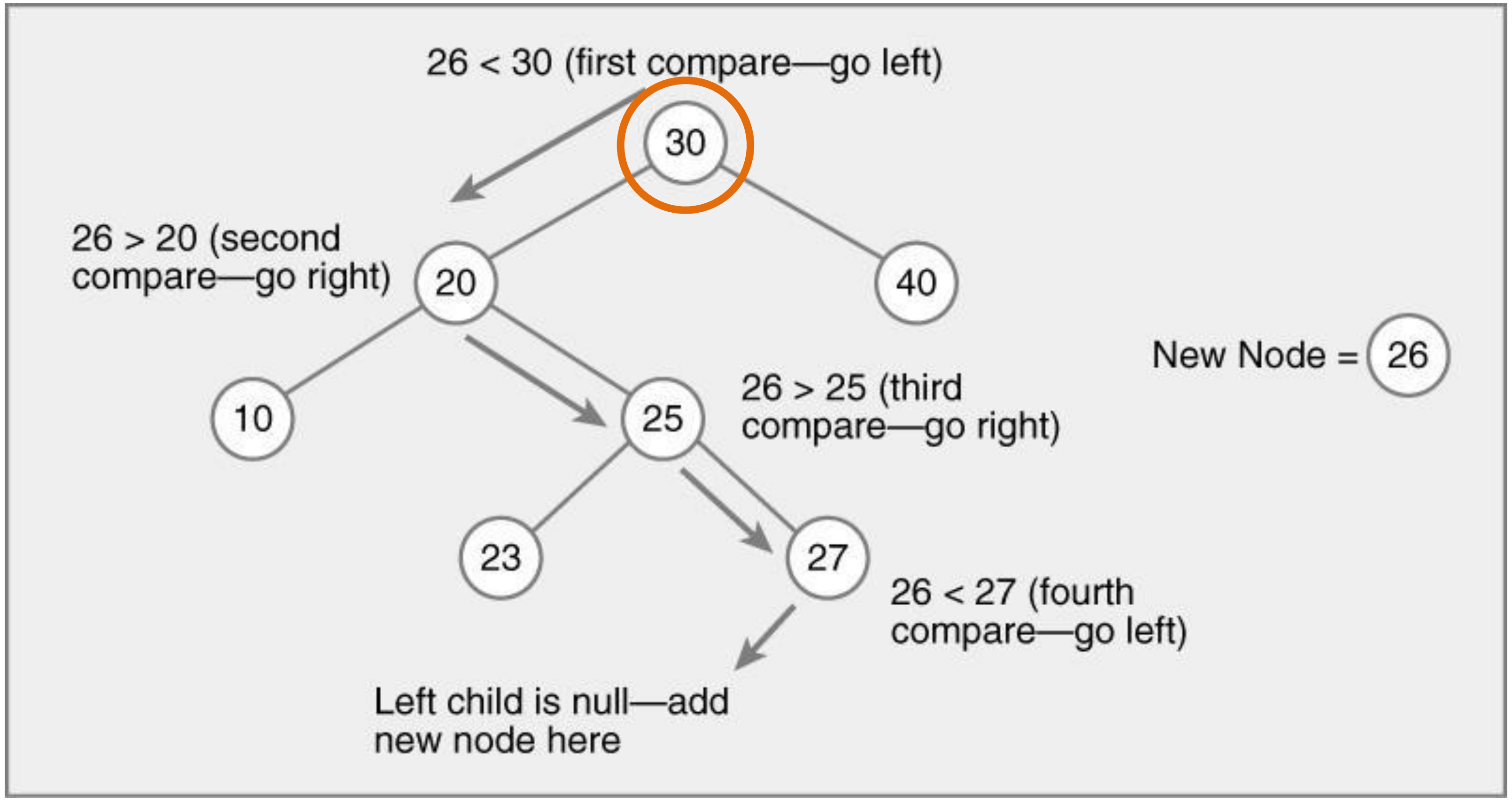
Find and Insert in BST

- **Find**: look for where it should be
- If not there, that's where you **insert**



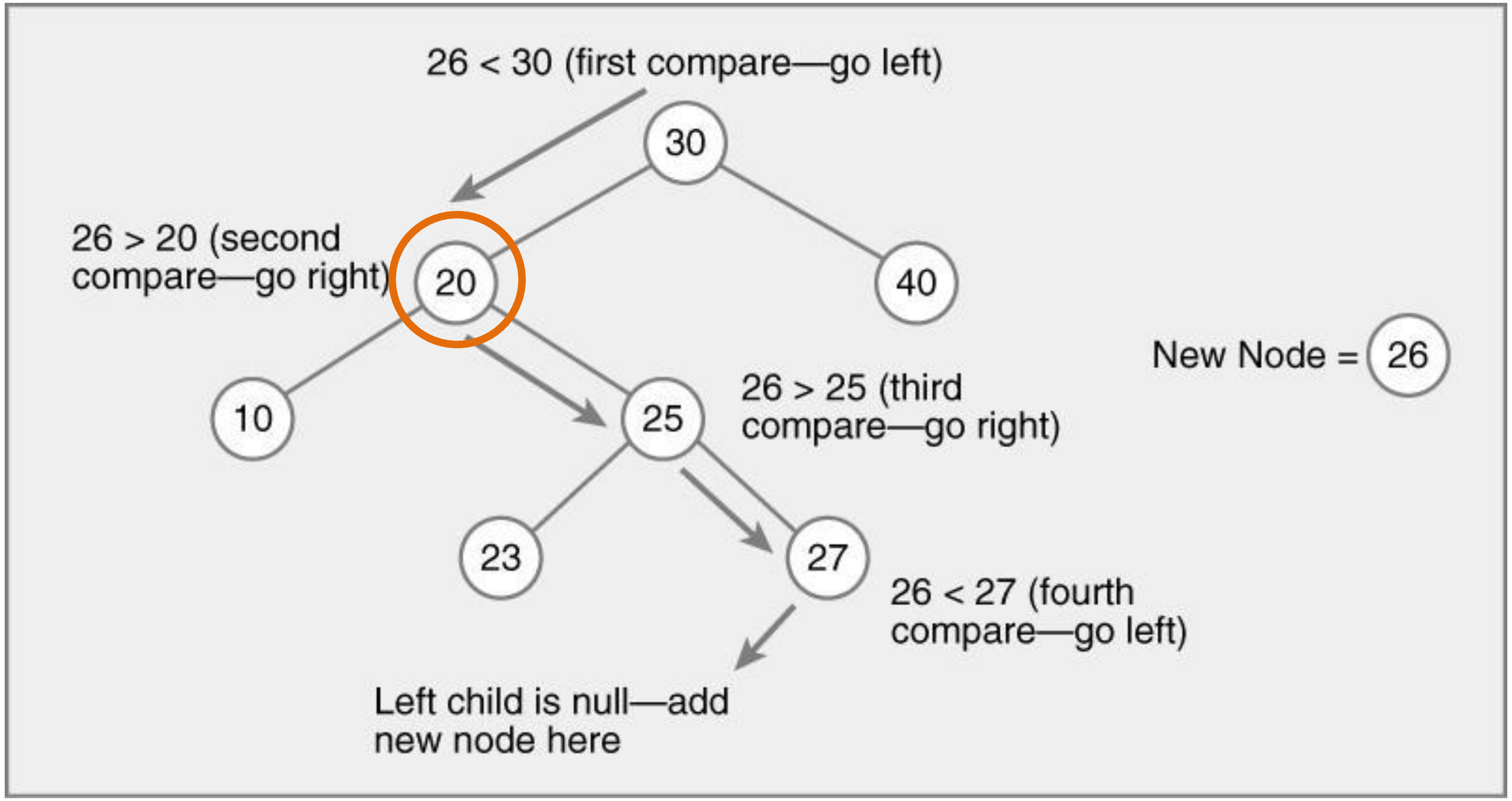
Find 23 in BST

- **Always start at the root of the tree!**
- **23 is LESS than 30, so go LEFT**



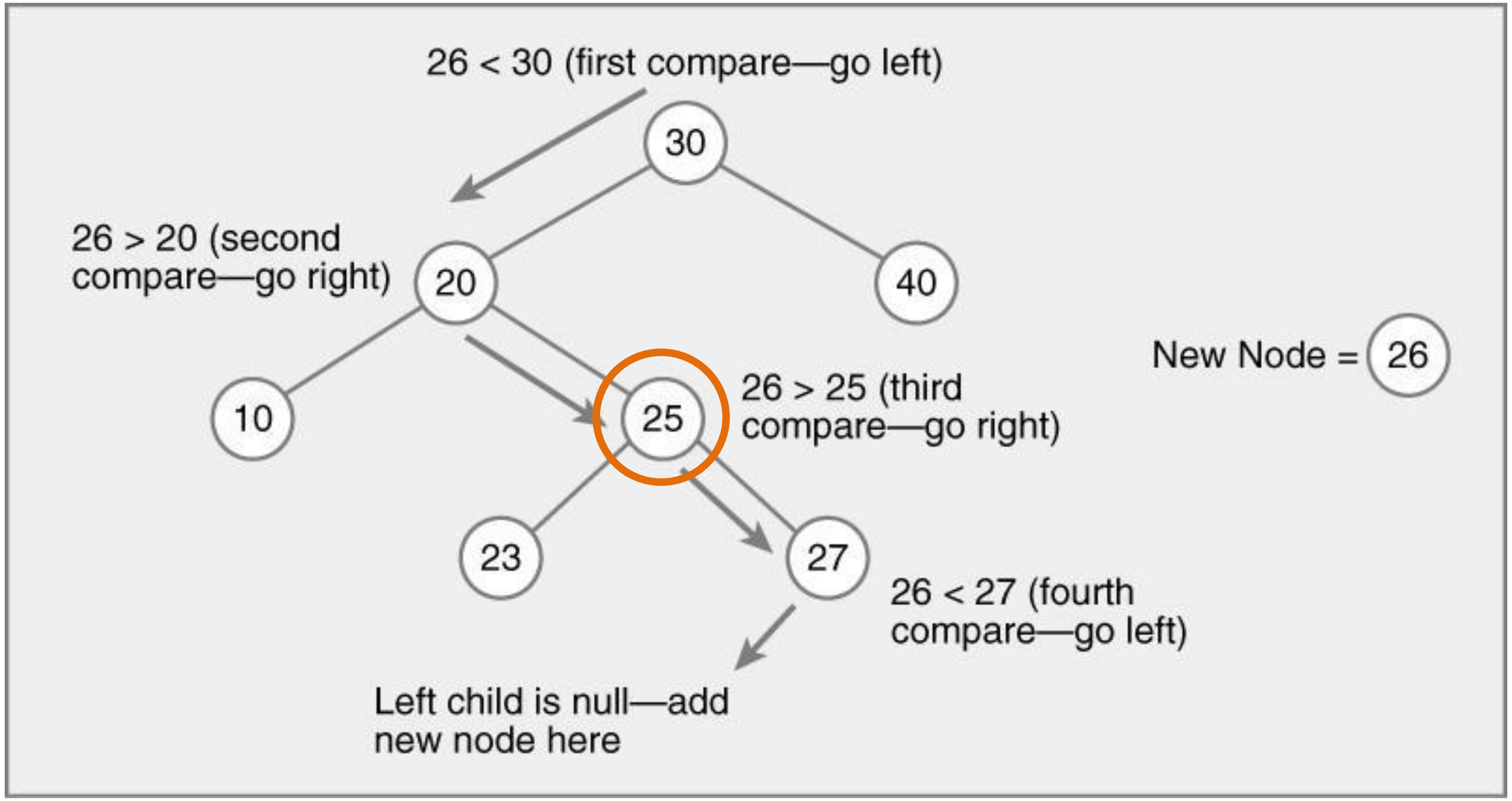
Find 23 in BST

- **Always start at the root of the tree!**
- **23 is GREATER than 20, so go RIGHT**



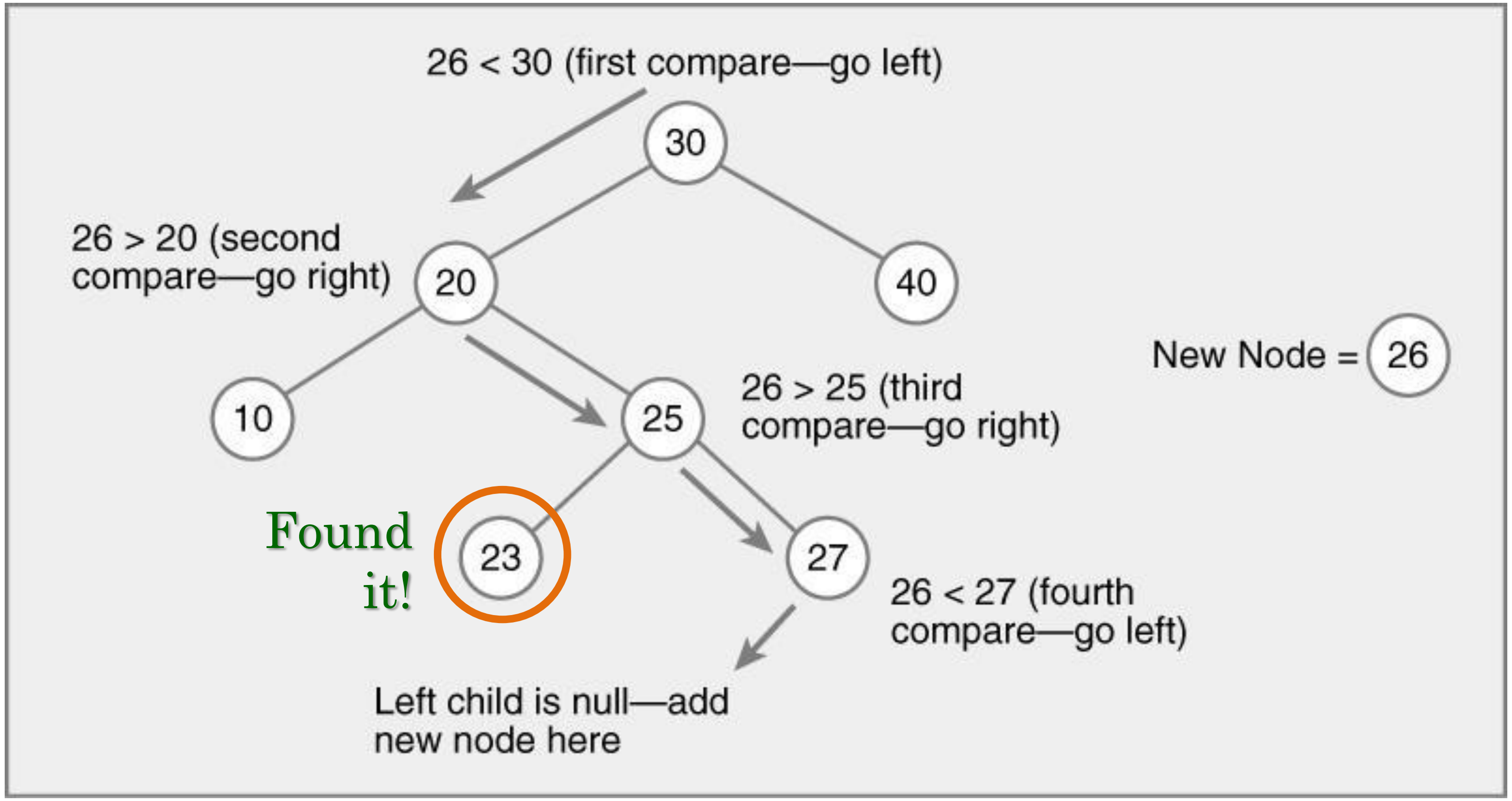
Find 23 in BST

- **Always start at the root of the tree!**
- **23 is LESS than 25, so go LEFT**



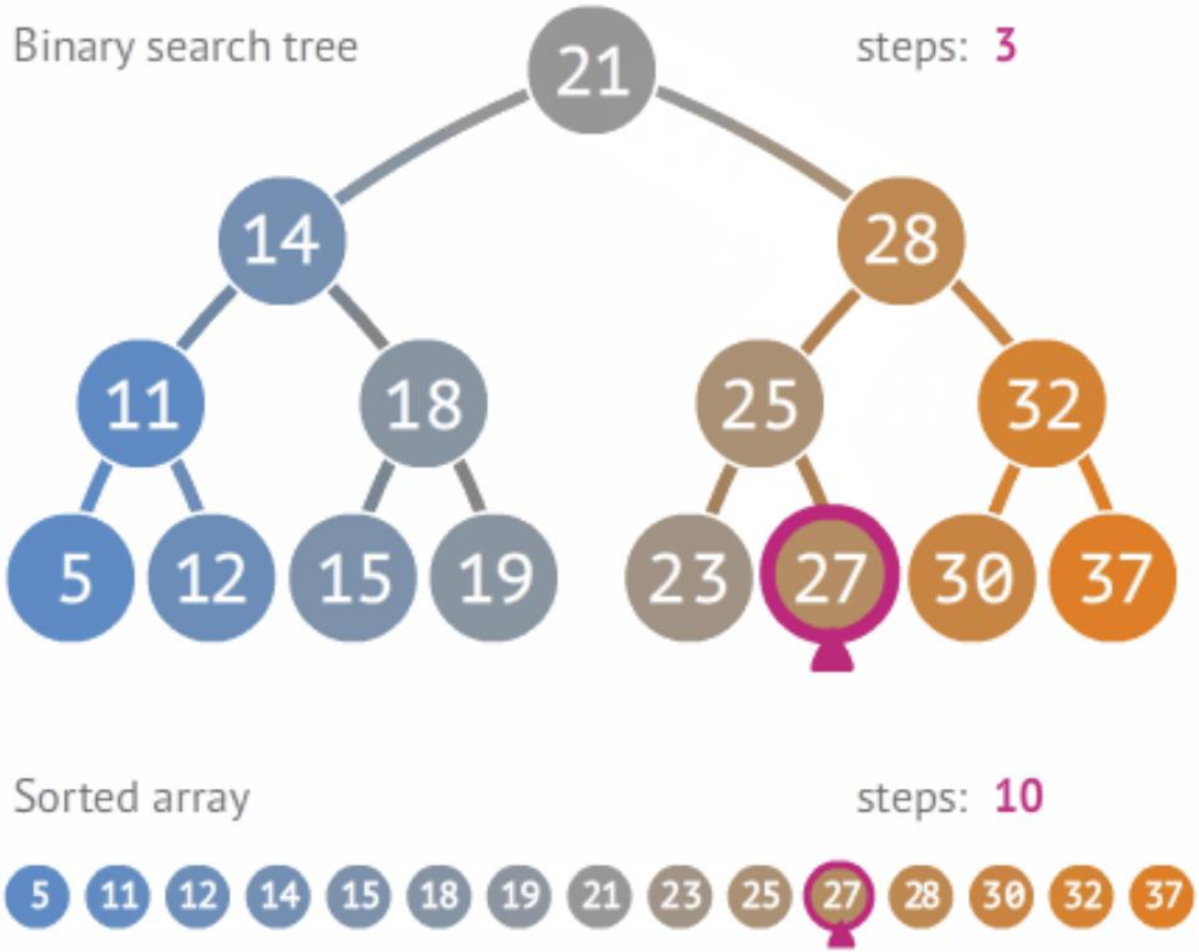
Find 23 in BST

- **Always start at the root of the tree!**
- **We found it!** *If not, 23 would be in this sub-tree*



Binary Search Tree vs Array

- Can find an element much quicker using a BST



Source:
penjee.com

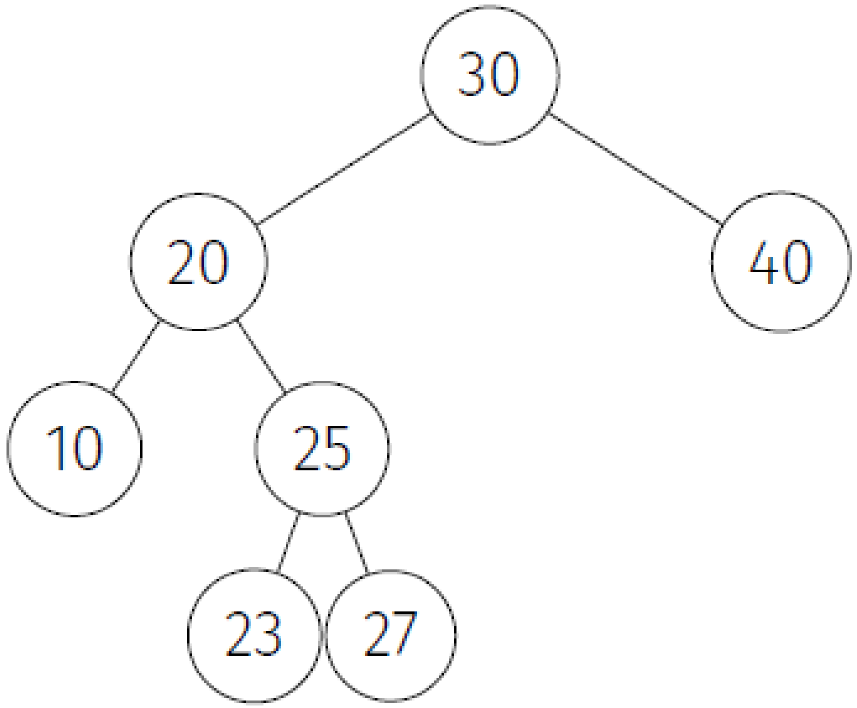
BST: Insert

- Where do we insert a new element?

  - Run `find()` method to determine **where** the element *should have been*
  - **Add** the new node at that position

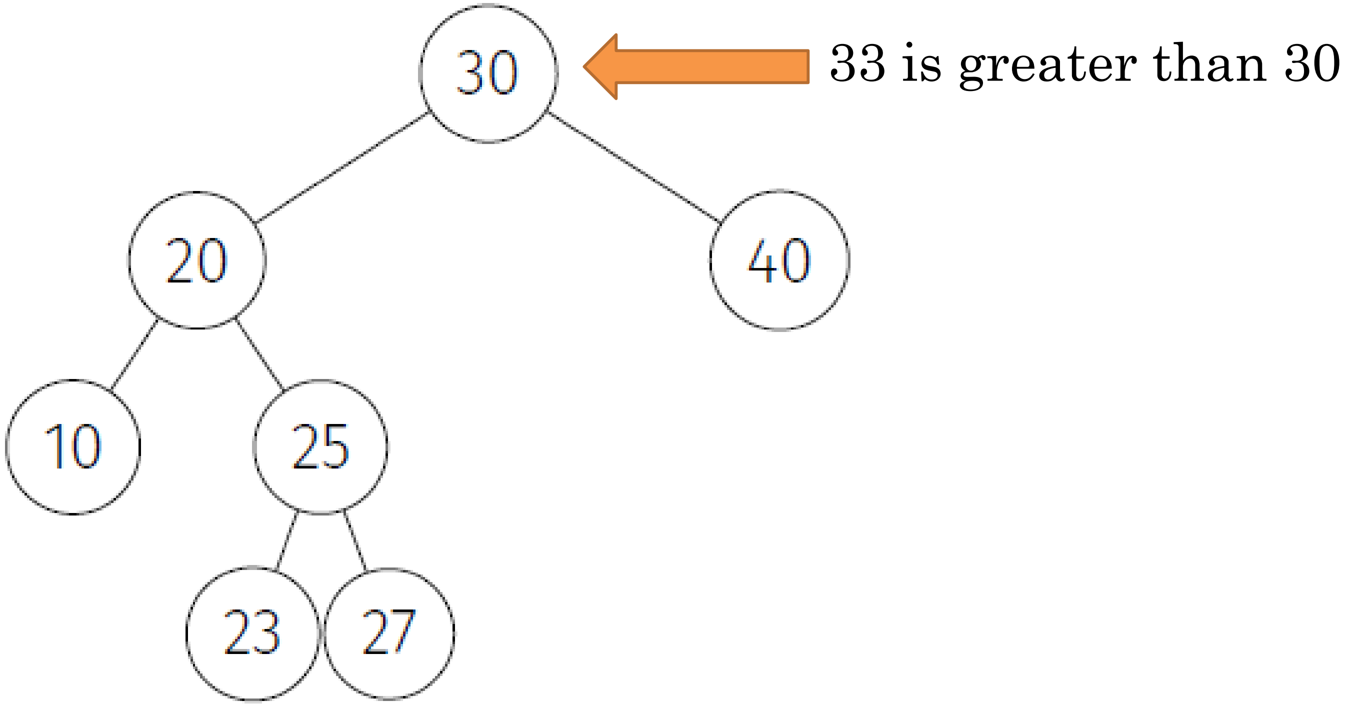
BST: Insert Example

- Insert 33 into the following binary search tree



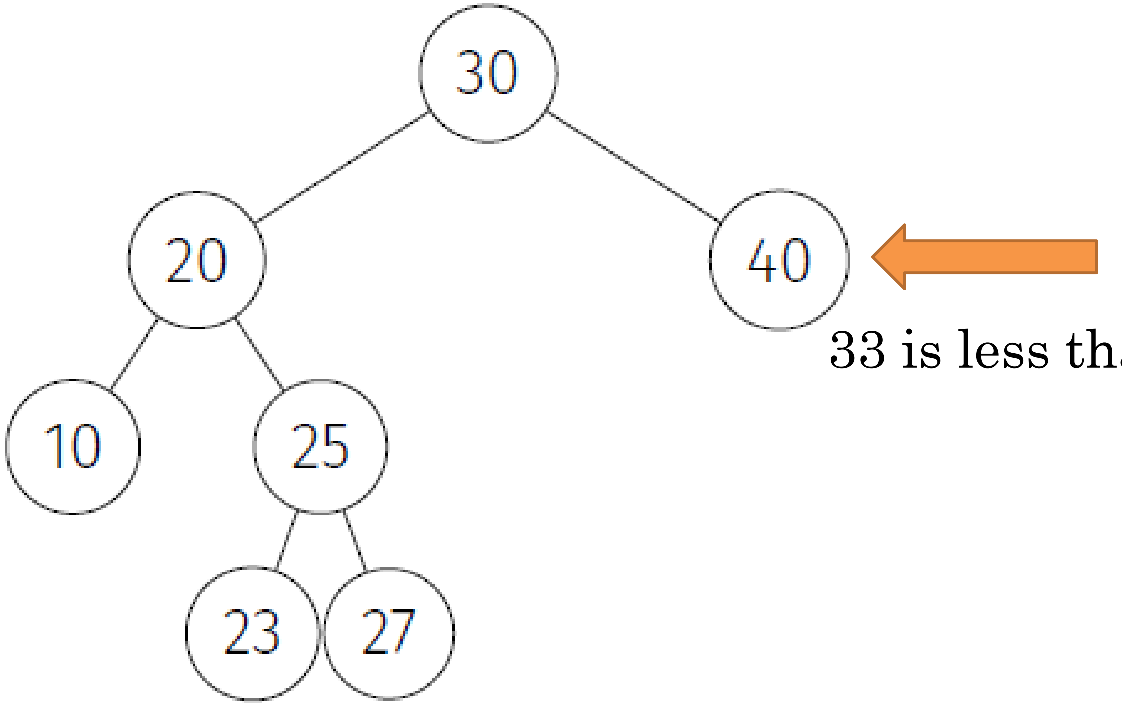
BST: Insert Example

- Insert 33 into the following binary search tree



BST: Insert Example

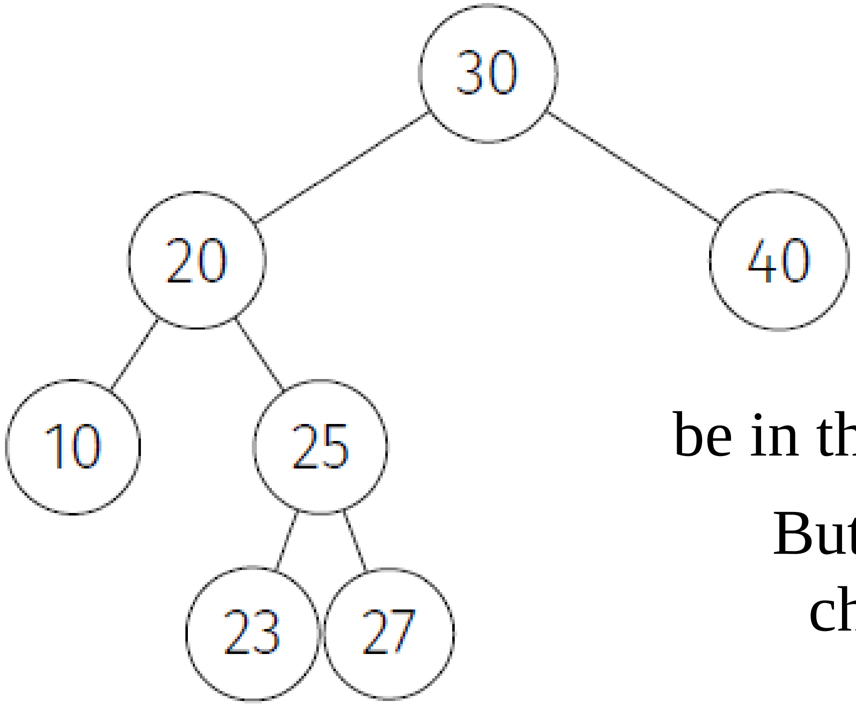
- Insert 33 into the following binary search tree



33 is less than 40

BST: Insert Example

- Insert 33 into the following binary search tree

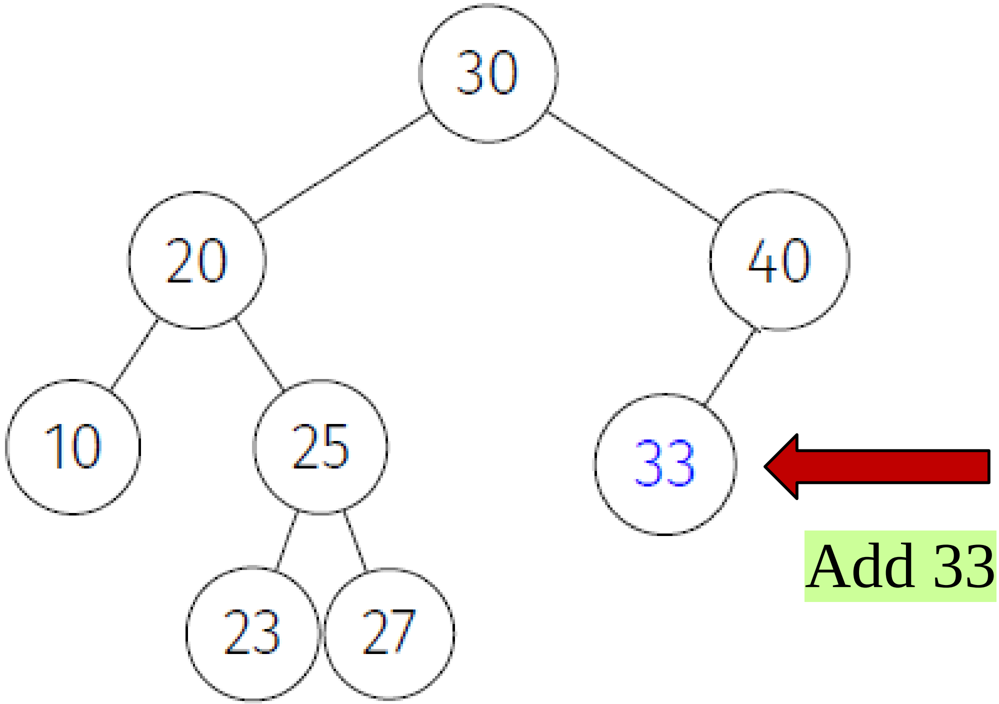


If 33 existed, it would be in the the **LEFT subtree** of 40.

But 40 does not have a left child: ***33 should go here!***

BST: Insert Example

- Insert 33 into the following binary search tree



Add 33 as left child of 40