Recursive Data Structures: Trees

-- Binary Search Trees --

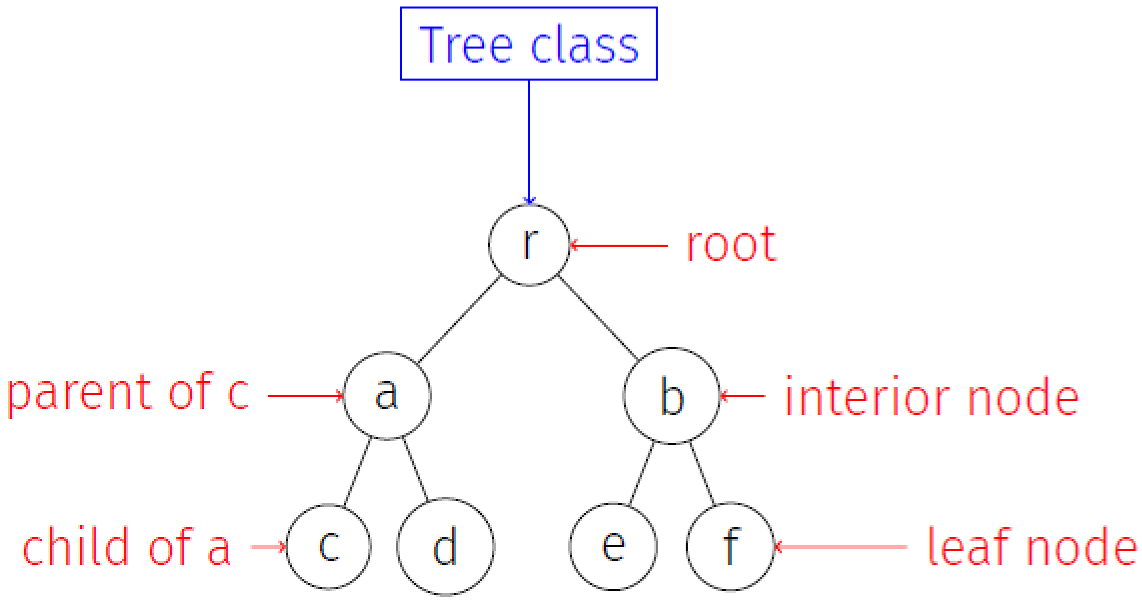
University of Virginia
CS 2110
Prof. N. Basit

Announcements

- Exam 2 regrades
  - Accepted until Tuesday, April 21 (attend OH)
- Homework 6 – Binary Search Trees
  - Released Monday (today, 4/20) due April 28, 2020
  - Auto-graded on Web-CAT; Submit JUnit tests
- Weekly Quiz out this Friday
  - Due by 11:30pm Sunday night as usual
- Schedule changes (slight):
  - Binary Search Trees (Monday – today)
  - Tree Traversals on Wednesday
  - Binary Heaps on Friday

Trees

- Reminder…

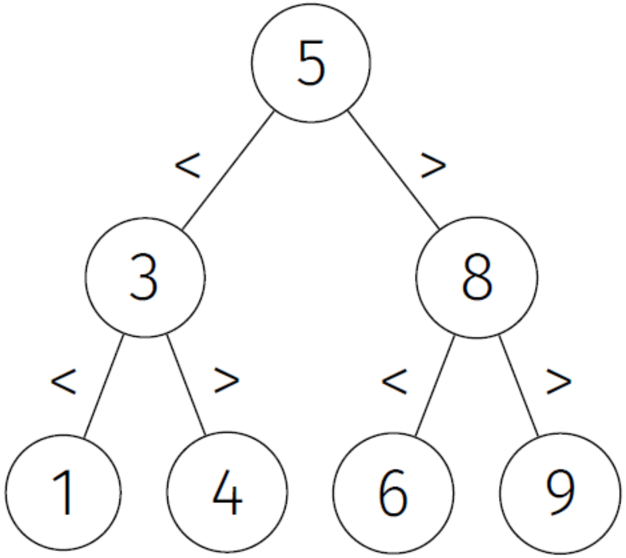


Binary Search Trees: Motivation

- 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` 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’ve been talking about to get the best of both worlds?

Binary Search Trees

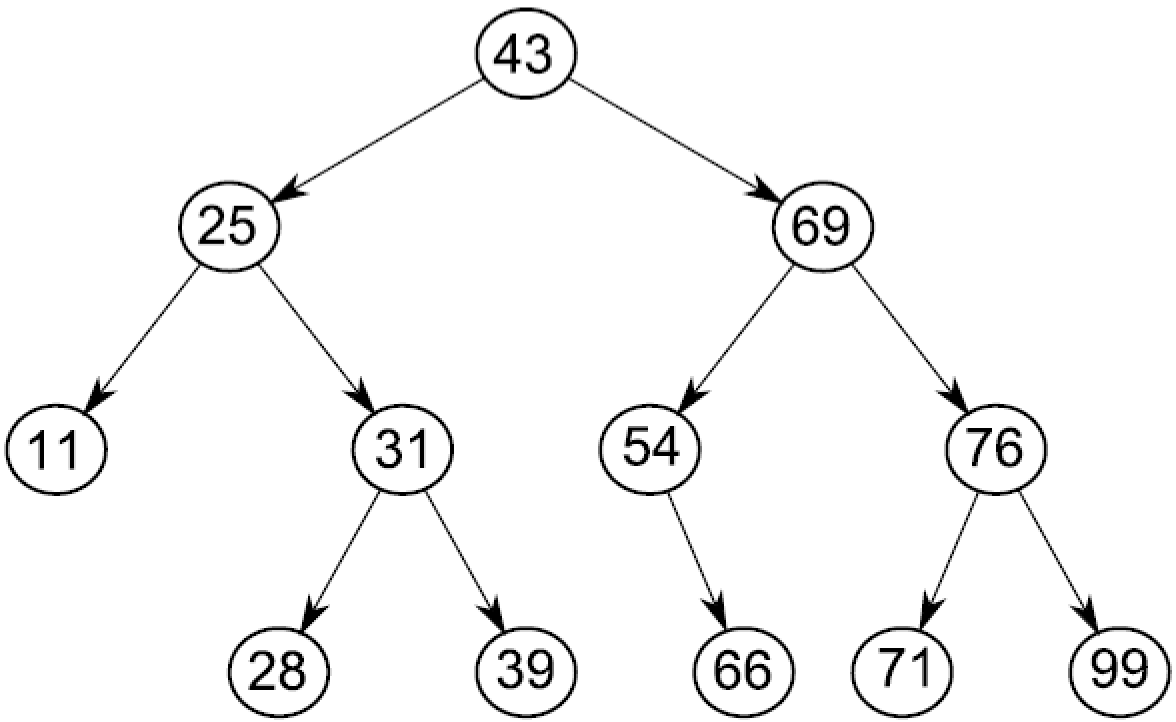


- Binary tree with **comparable** data values
- **Binary search tree (BST) property:**
  - The data values of all *descendants* to the **left** *subtree* are **less** than the data value stored in the parent node, and
  - The data values of all *descendants* to the **right** *subtree* are **greater** than the data value stored in the parent node
- BST requirement:
  - The data variable should be a **Comparable** type (not Object) in order for the data comparisons to work

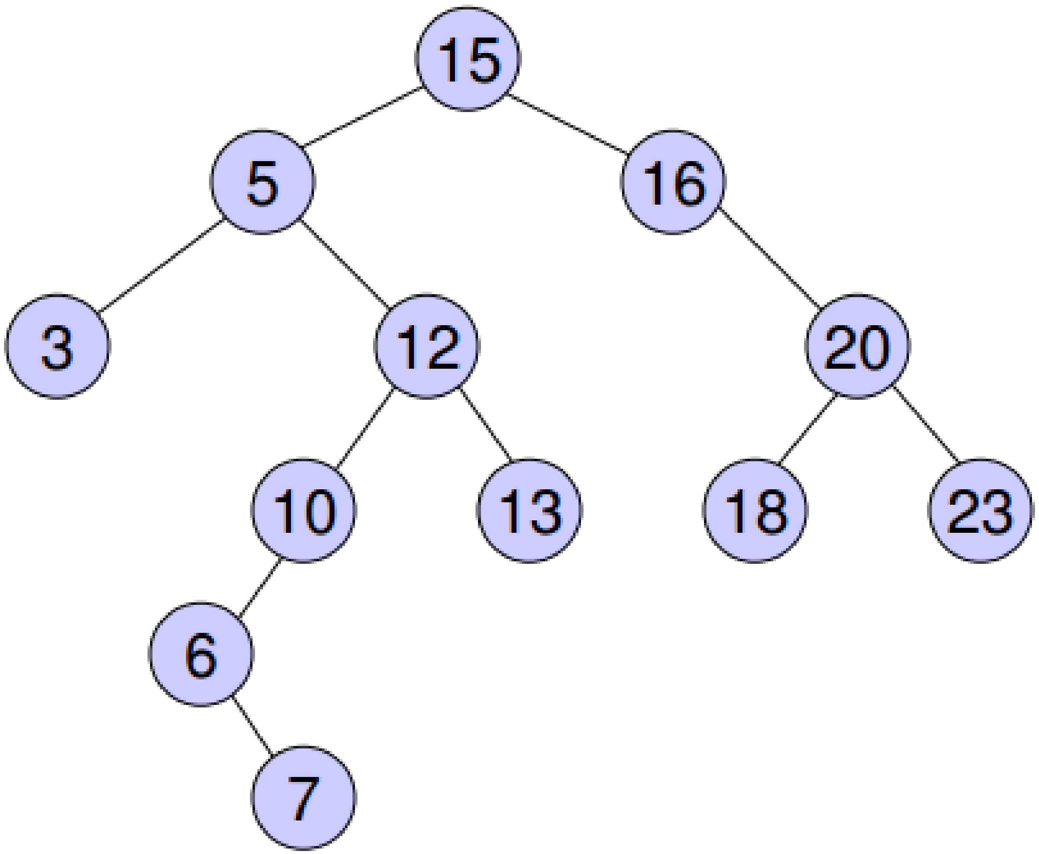
Binary Search Trees: Cool Property

- How could we traverse a BST so that the nodes are visited in **sorted** order?
  - *In-order* traversal: left tree, node, right tree
  - [We'll see tree traversals in the next topic!]
- It's a very useful property about BSTs
- Consider Java's **TreeSet** and **TreeMap**
  - Built using search trees (not a BST, but one of its better "cousins")
    - In CS2150: AVL trees, Red-Black trees
  - Guarantee: search times are **O(lg n)**

Example of a Binary Search Tree

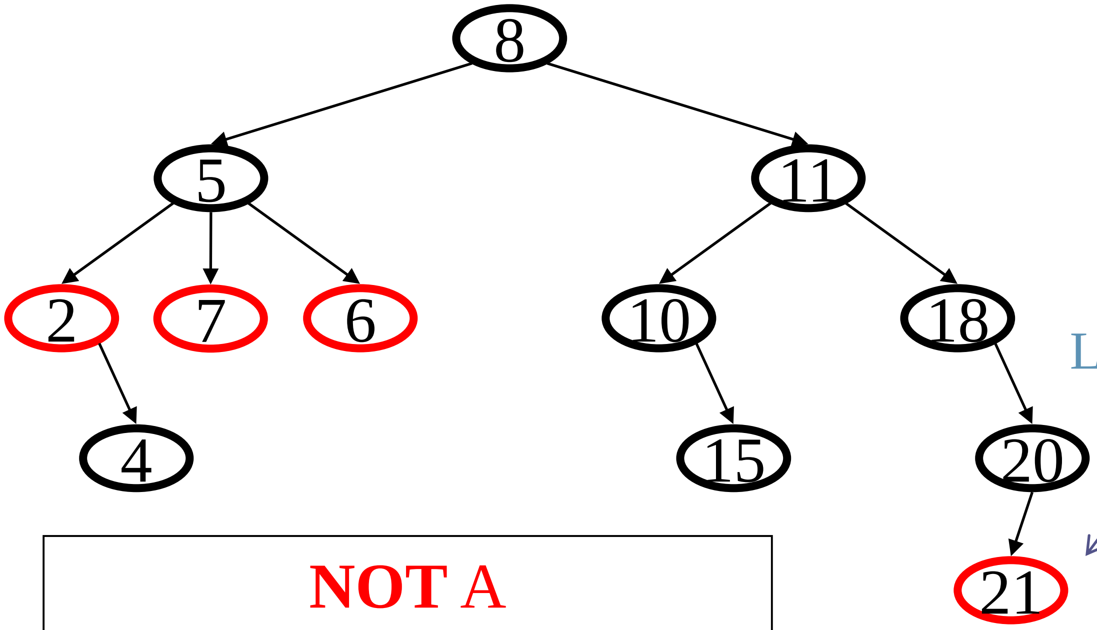


Another example of a Binary Search Tree



Counterexample (**not** a BST)

Three child nodes

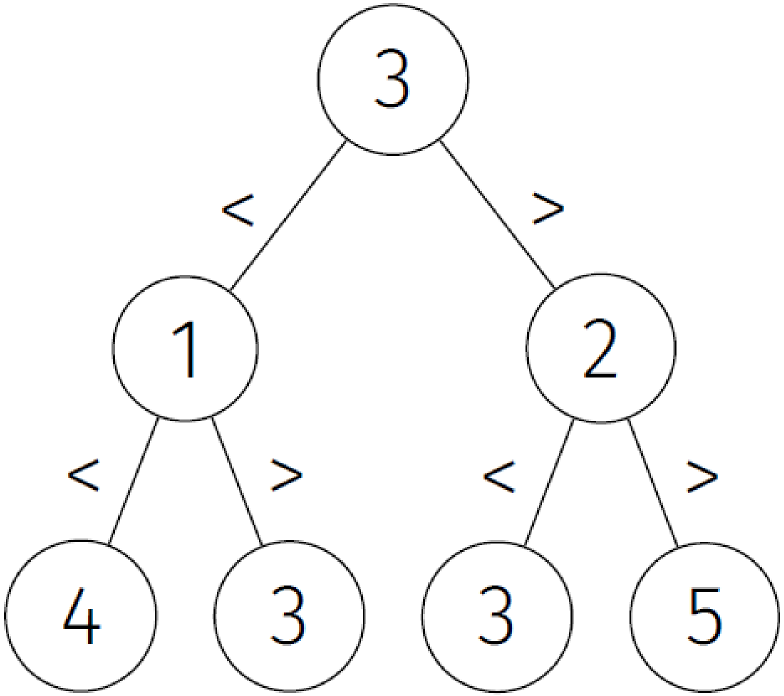


Left node not less than parent node (20)

NOT A
BINARY SEARCH TREE

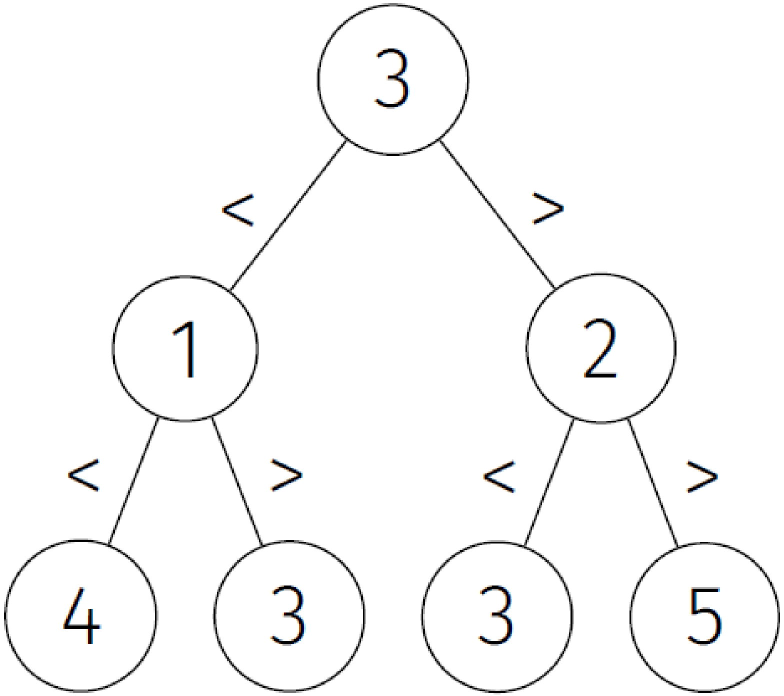
Question!

- Is this a binary search tree?



Question!

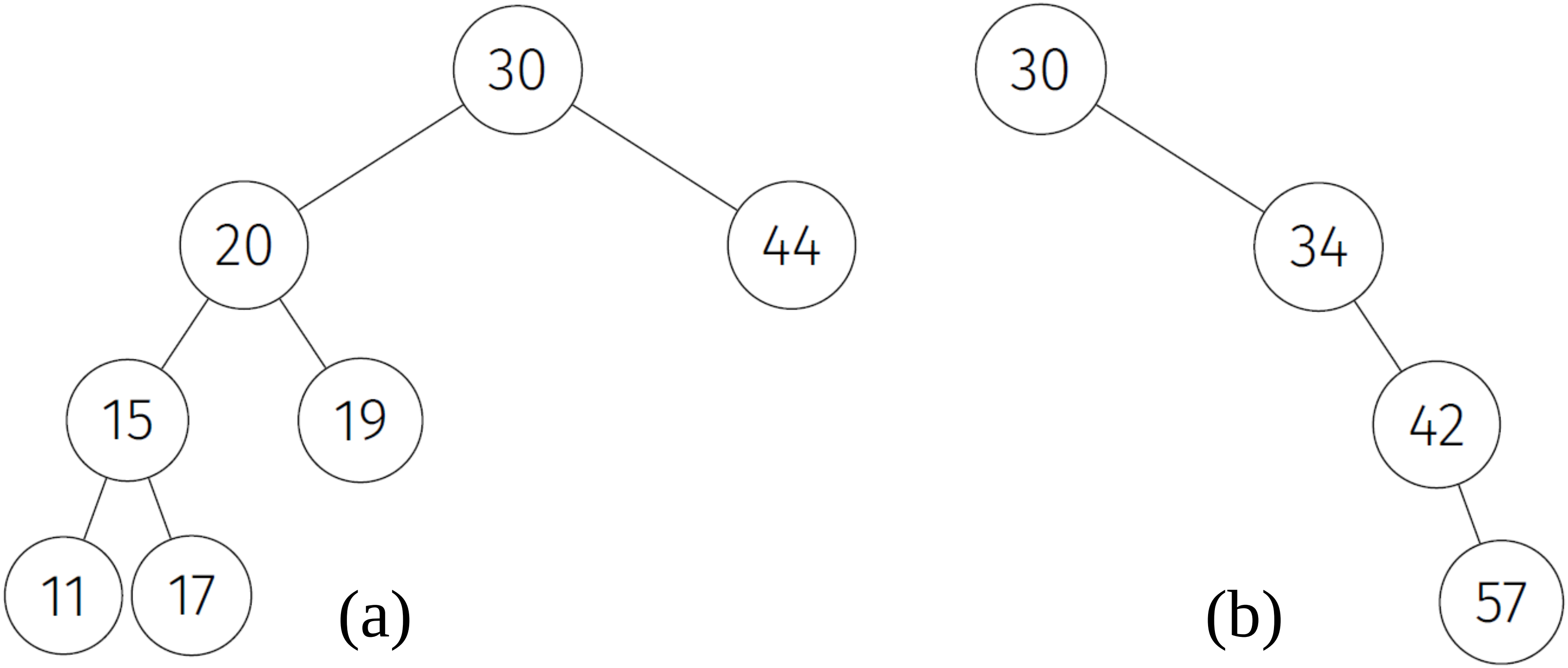
- Is this a binary search tree?



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

Question!

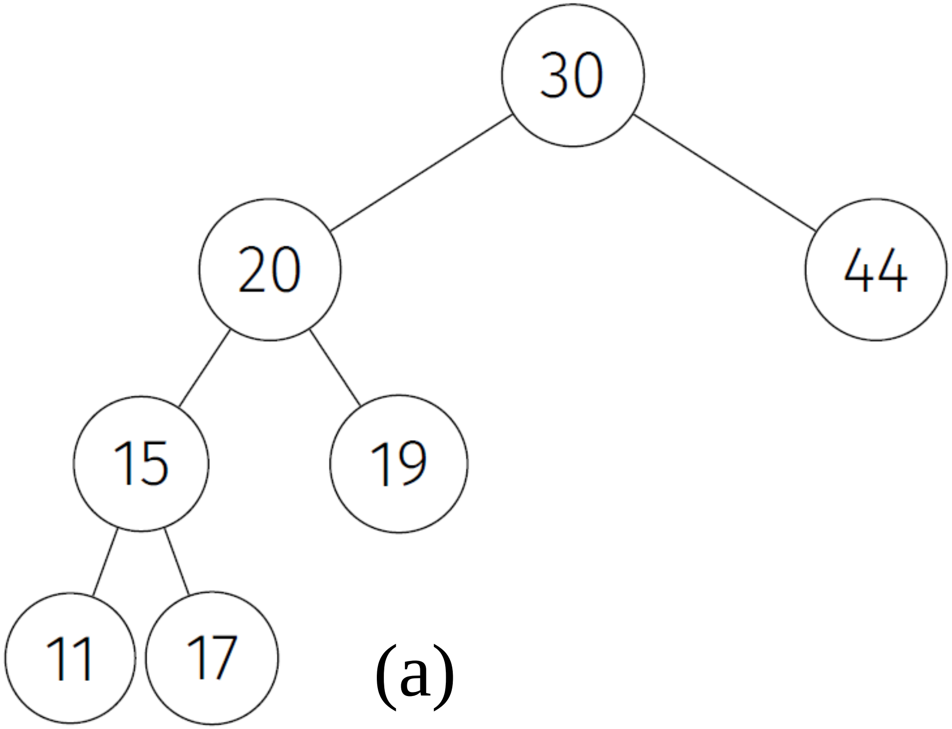
- Are these binary search trees?



(a) (b)

Question!

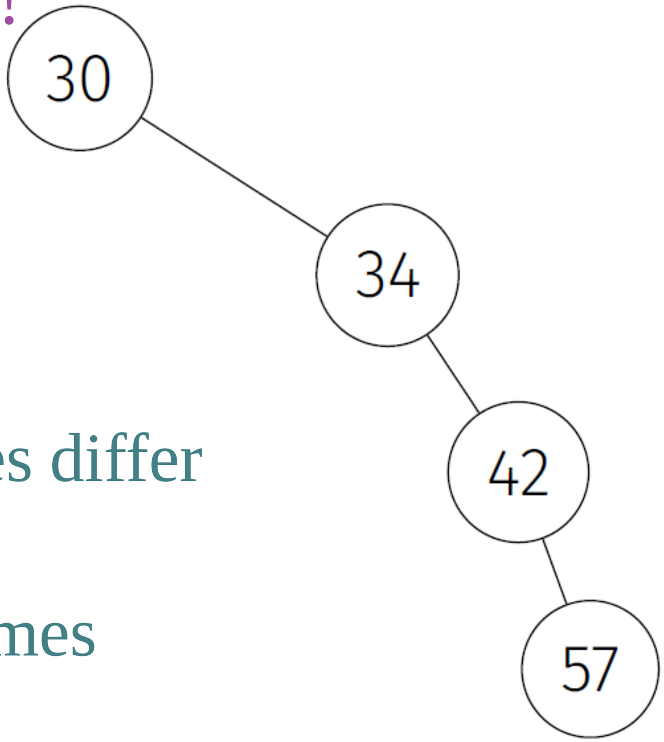
- Are these binary search trees? ***No!*** Binary search tree property not preserved



(a)

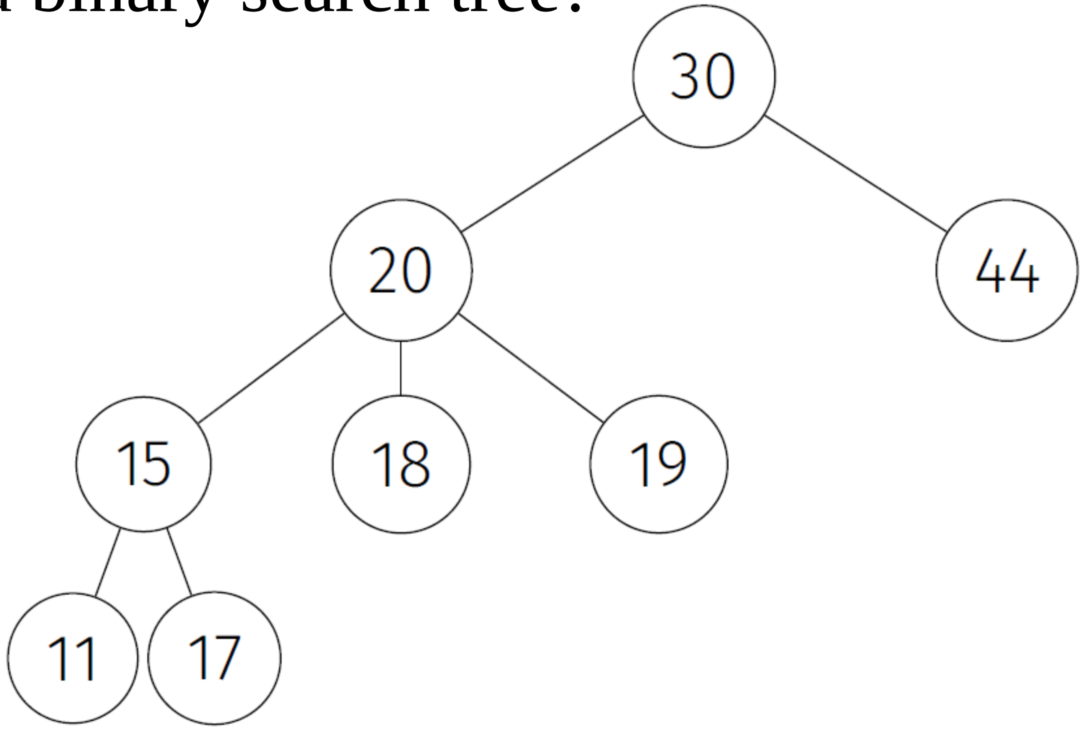
Question!

- 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



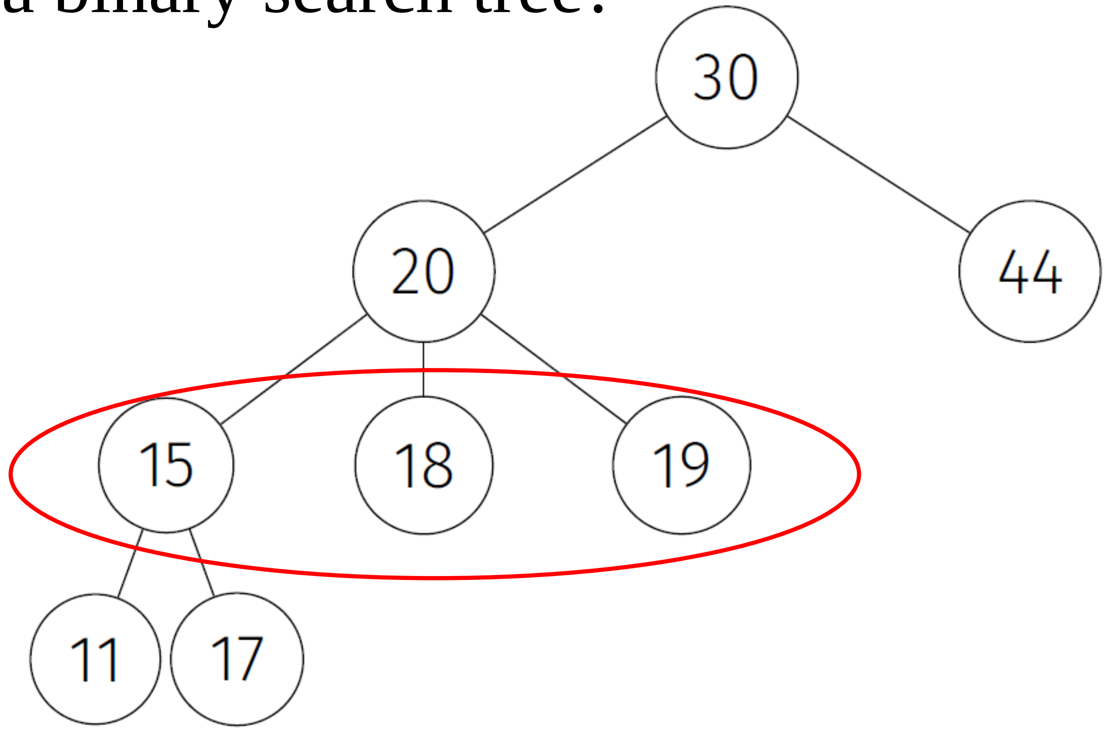
Question!

- Is this a binary search tree?



Question!

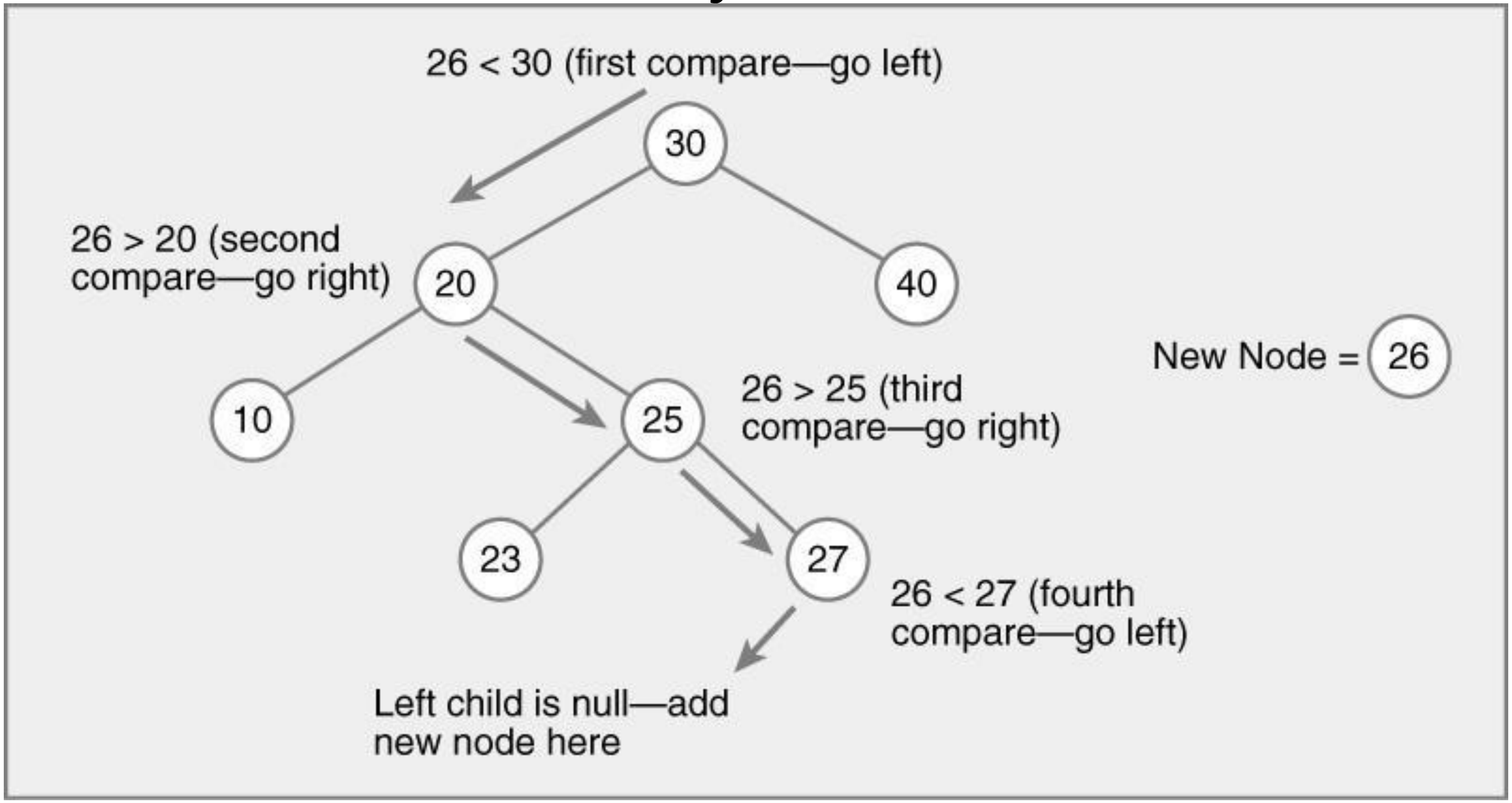
- Is this a binary search tree?



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

Find and **Insert** in BST

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

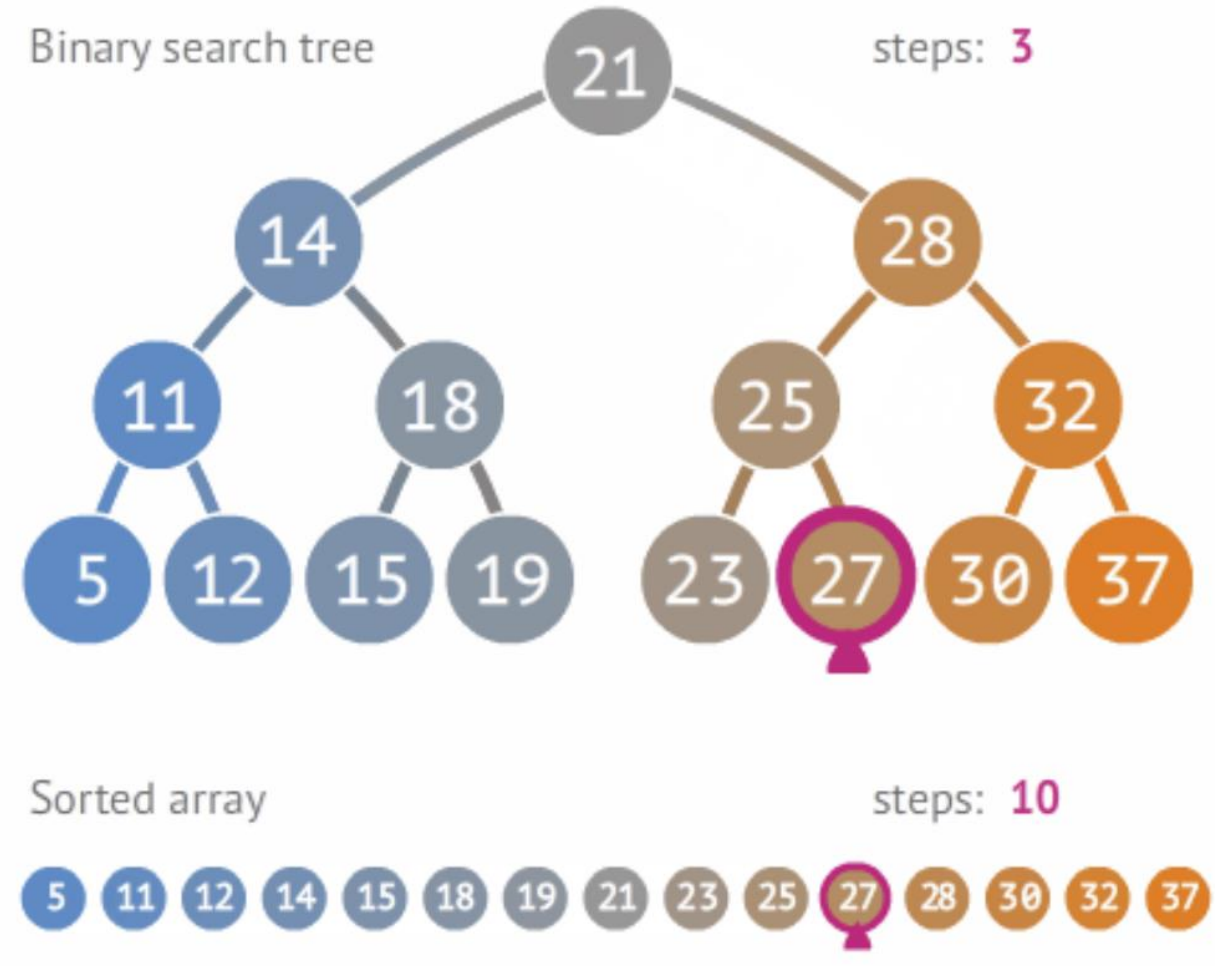


BST Find and Insert

- **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)

Binary Search Tree vs Array

- Can find an element much quicker using a BST



Source: penjee.com

Recursion and Tree Operations

- Recursive code for tree operations is simple, natural, elegant
- Example: **pseudo-code** in TreeNode

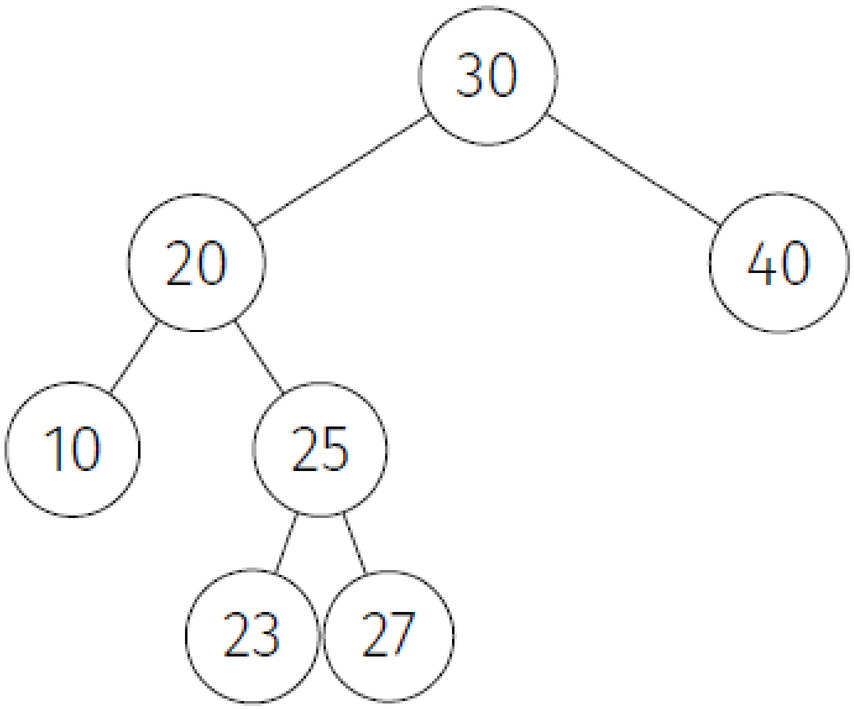
```
boolean find(Comparable target) { //find target
      Node next = null;
      if (this.data matches target) //found it!
            return true
      else if (target’s data < this.data)
            next = this.leftChild //look left
      else
            next = this.rightChild //look right
      // ‘next’ points to left or right subtree
      if (next == null ) return false // no subtree
      return next.find(target) // search on
  }
```

Find and 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

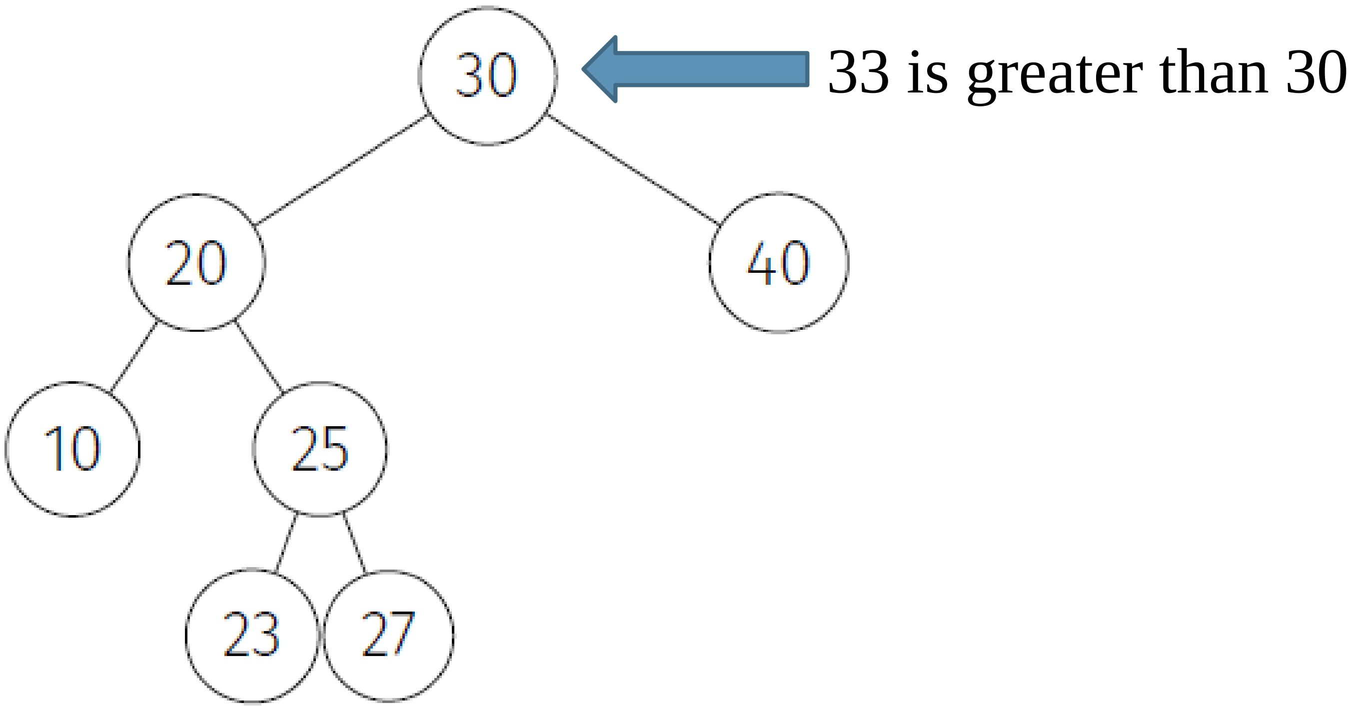
Insert Example

- Insert 33 into the following binary search tree



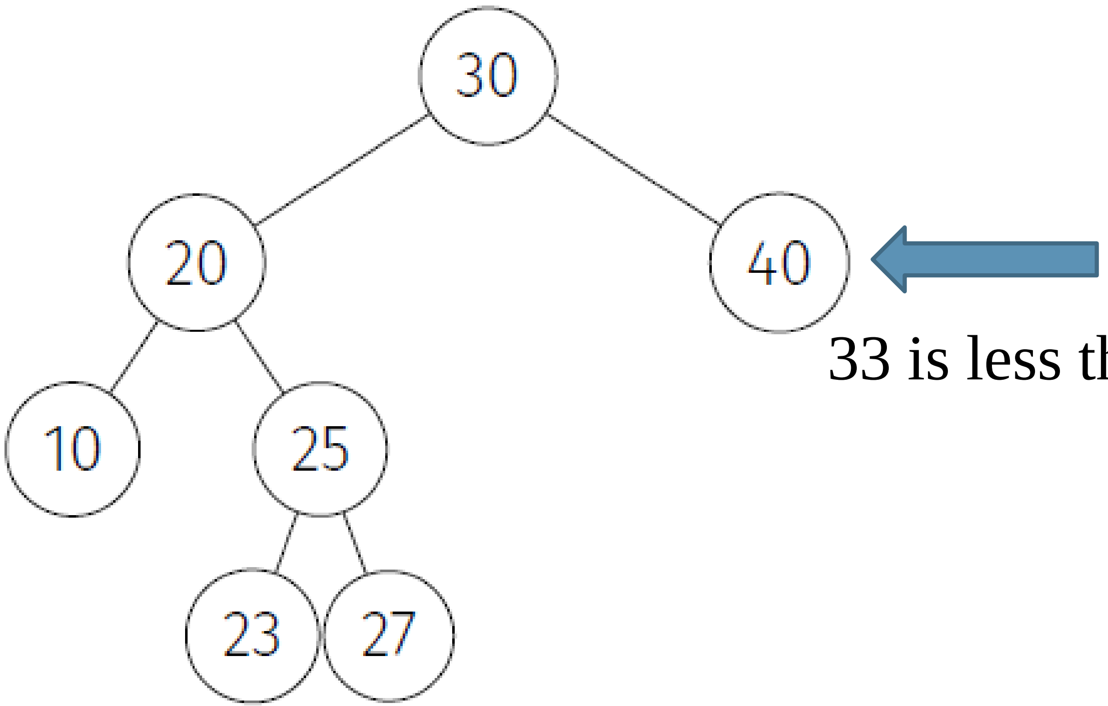
Insert Example

- Insert 33 into the following binary search tree



Insert Example

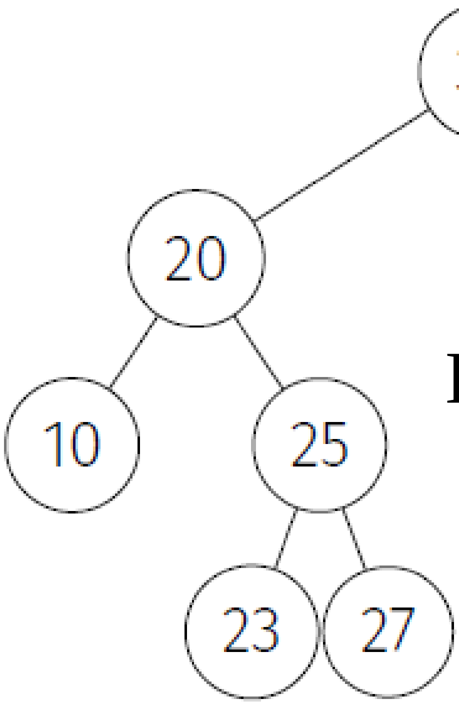
- Insert 33 into the following binary search tree



33 is less than 40

Insert Example

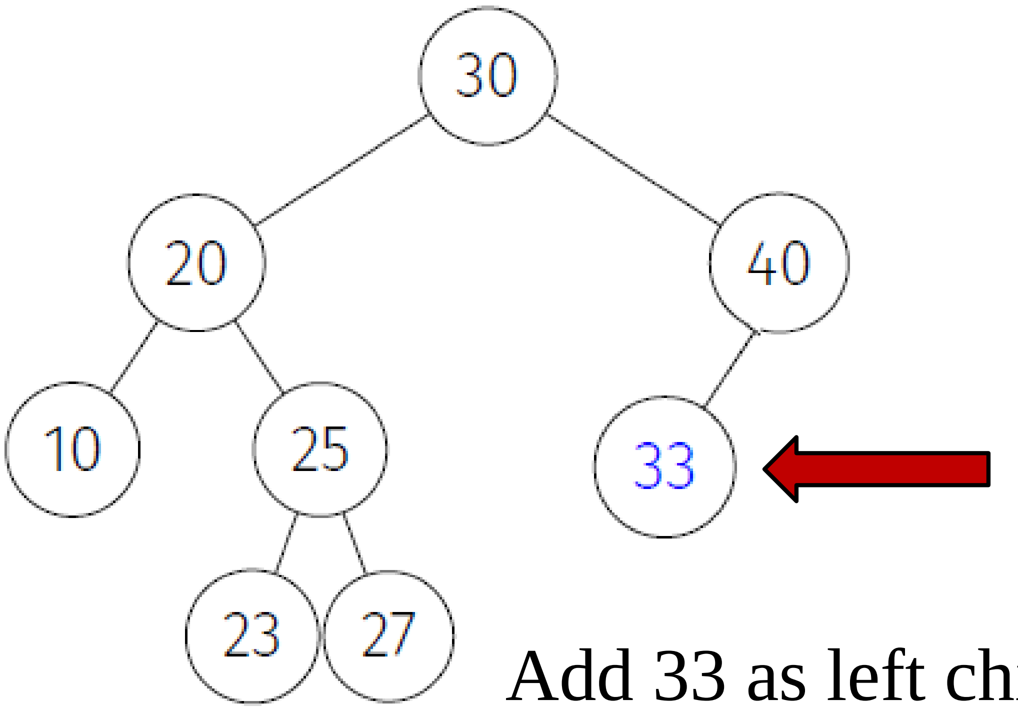
- Insert 33 into the following binary search tree



If 33 existed, it would be in the **left subtree** of 40. But 40 does not have a left child: *33 should go here!*

Insert Example

- Insert 33 into the following binary search tree



Add 33 as left child of 40

Deleting from a BST

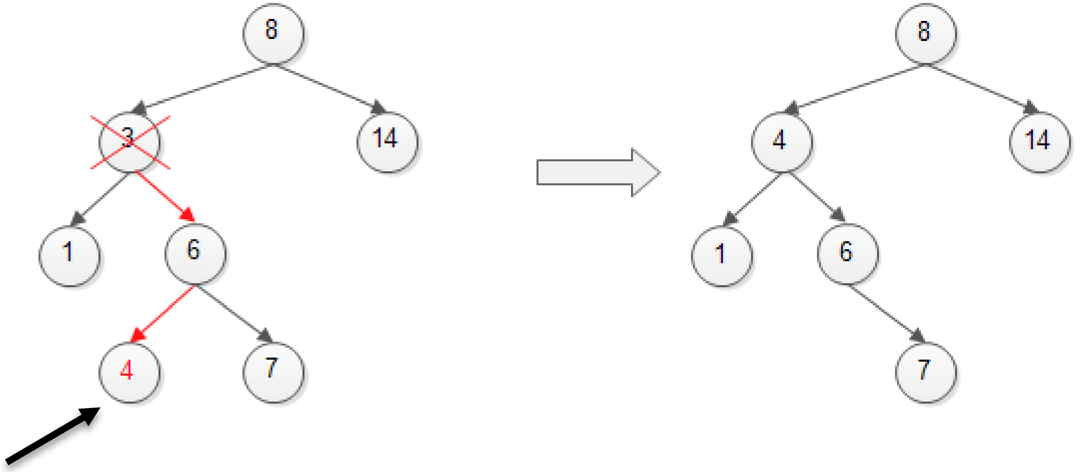
- **Delete** a node from the tree
  - More complicated – we need to select a node as replacement!
- **Removing** a node requires
  - Moving its left and right subtrees
  - If 0 children: delete node
  - If 1 child: replace node with its only child
  - If 2 children: find next largest (or smallest) to fill in
- Answer: not too tough, we'll go over the idea of how to remove nodes next

Deleting from a BST *(finding a successor)*

- After removing an element from a BST, you have to find a node with which to replace it (it's "Successor")
- Where to find the successor? Well, there are 2 options:
  - The next "largest" element
  - The next "smallest" element
- Where would these exist in the BST?
  - Next largest: in right sub-tree – ***but where in the sub-tree?***
  - Next smallest: in left sub-tree – ***but where in the sub-tree?***

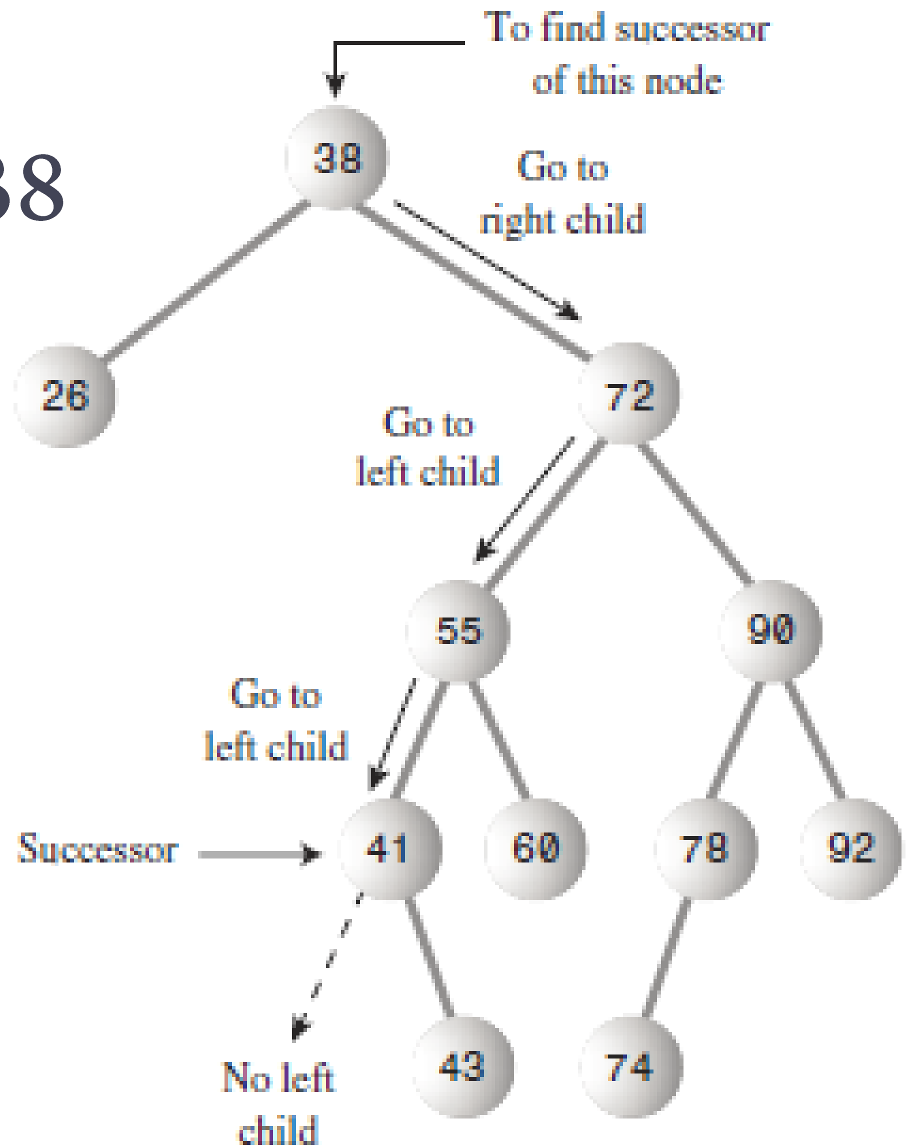
Deleting Quick Overview

- Find successor (of 3) in it's **right** subtree (i.e. node **4**) – finding the **minimum** (*leftmost node*) of right subtree



Find Successor of 38

- *Minimum* of **right** subtree (*leftmost node*)
- Which is the <u>next largest</u> number

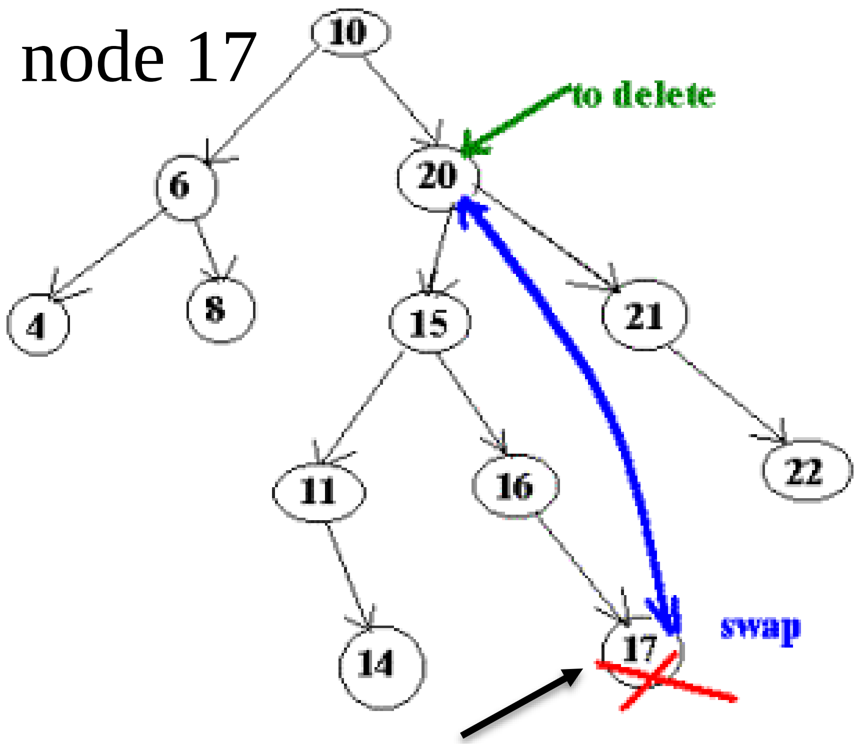


Finding the successor.

Another Example – successor of 20 [1]

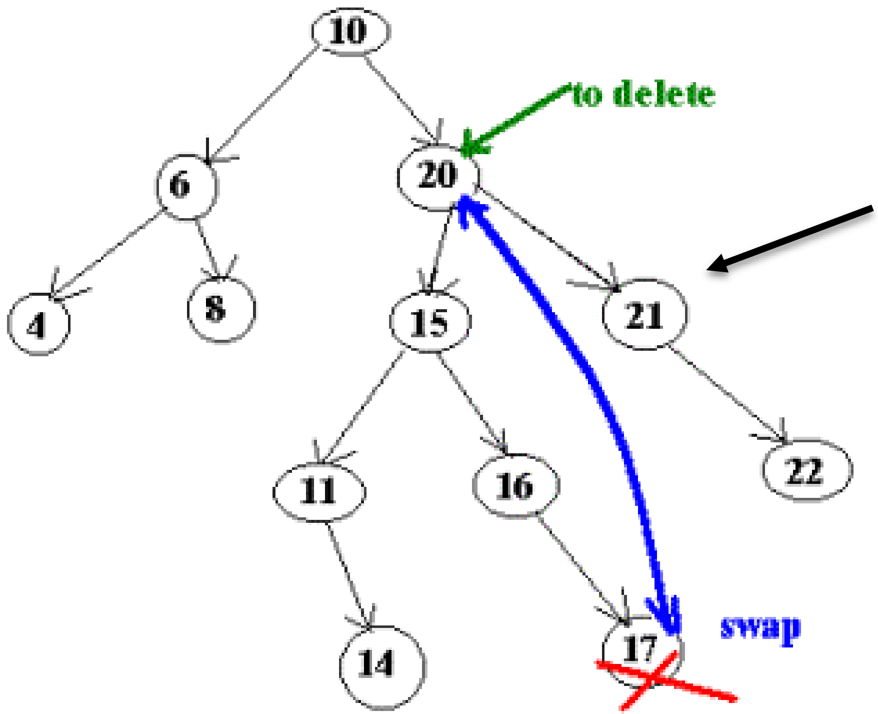
- Largest number –*rightmost*– of **left** subtree (largest number out of left subtree that contains values *smaller* than that node) ➔ node 17

- Which is the next smallest number



Another Example – successor of 20 [2]

- Next largest number (minimum number –*leftmost*– of **right** subtree) ➔ node 21



(Node "21" has no left child, therefore it becomes the leftmost node in the **right** subtree.)

In-Class Activity – Binary Search Trees

1. Download `BinaryTree.java` and `BinaryTreeNode.java`
2. Given the following sequence of integers, your task is to build a Binary Search Tree (BST). Each integer will be a data value in a node. **Input sequence:** **{6, 4, 3, 5, 8, 9, 1, 2}**
3. In the main method of `BinaryTreeNode.java` create these nodes
   - Use the Integer data type:
     ```
     BinaryTreeNode<Integer> n1 = new BinaryTreeNode<Integer>(6);
     ```
4. Create the connections using `setLeft()` & `setRight()`
5. When finished, take the root node (e.g. n1 – with data value 6) and call `toString()` to print out the result. If done correctly the **output should be:** **2, 1, 3, 5, 4, 9, 8, 6**
6. **SUBMIT**: your `BinaryTreeNode.java` file on Collab
   - Work with a partner, but submit individually