Software Development 3 (F27SG)

Lecture 13

Implementing Binary Search Trees

Rob Stewart

Summary

- We have
  - Introduced Trees and Binary Search Trees (BST)
  - Looked at how to **traverse** BSTs
- Today we will look at Binary Search Tree
  - operations:
    - add
    - remove
    - search
  - implementation

Binary Tree Class

- Same approach as with the Linked List Class
  - A **wrapper class**
    - that exposes the **ADT** to other parts of the program
  - An **inner class**
    - represents **Nodes** and has all the recursive elements

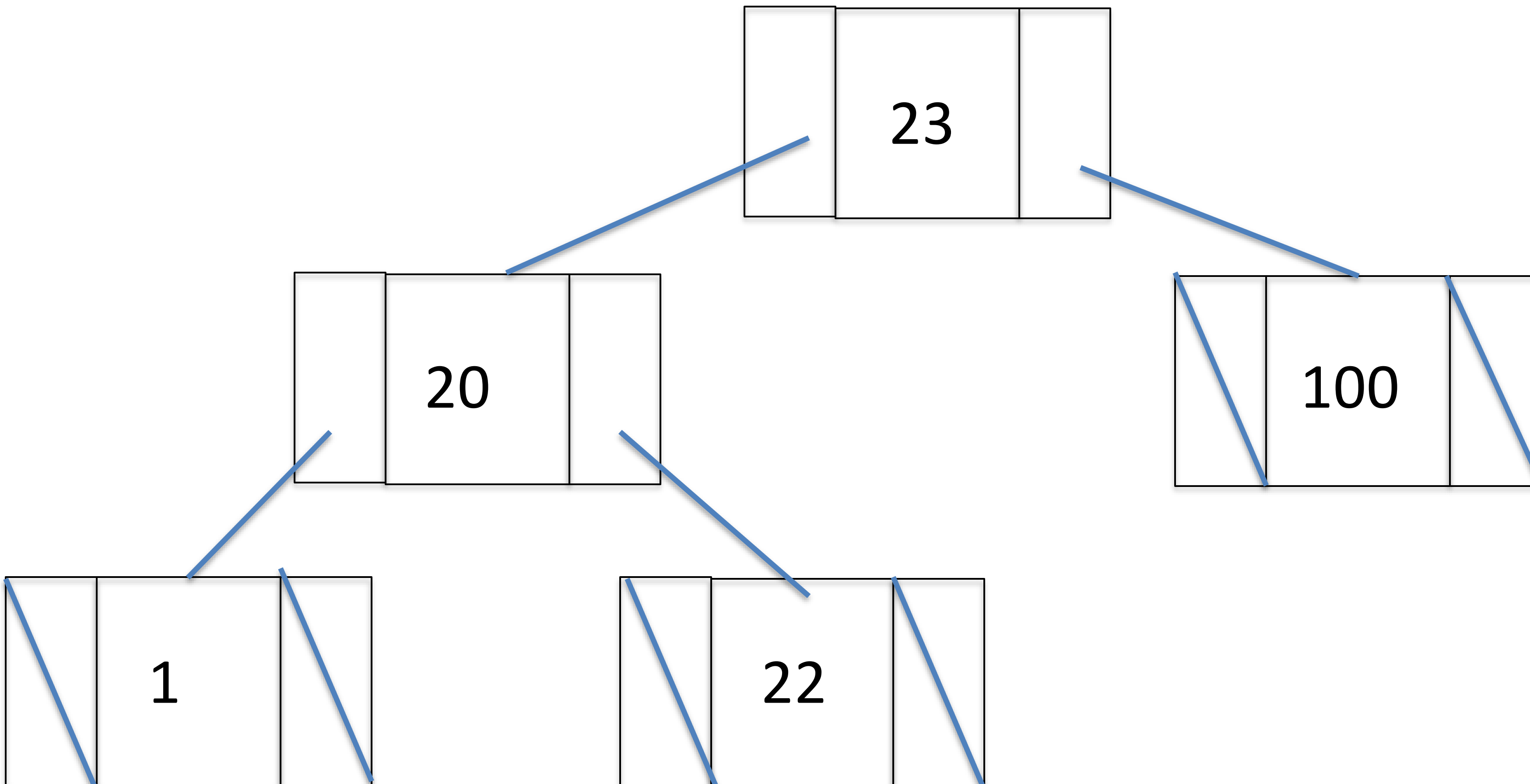
Inserting Into BSTs

- Two things to remember
  - 1. You cannot have duplicates
  - 2. **Insert** into "empty tree"
    - If (rootNode == null)
    - Then make a new node and set as root

- Basic **Insert** Algorithm
  - From root node, traverse left or right down the tree
    - Compare value to see if it should be left or right
    - Smaller to left, larger to the right
  - When you reach a null reference, insert the new node.

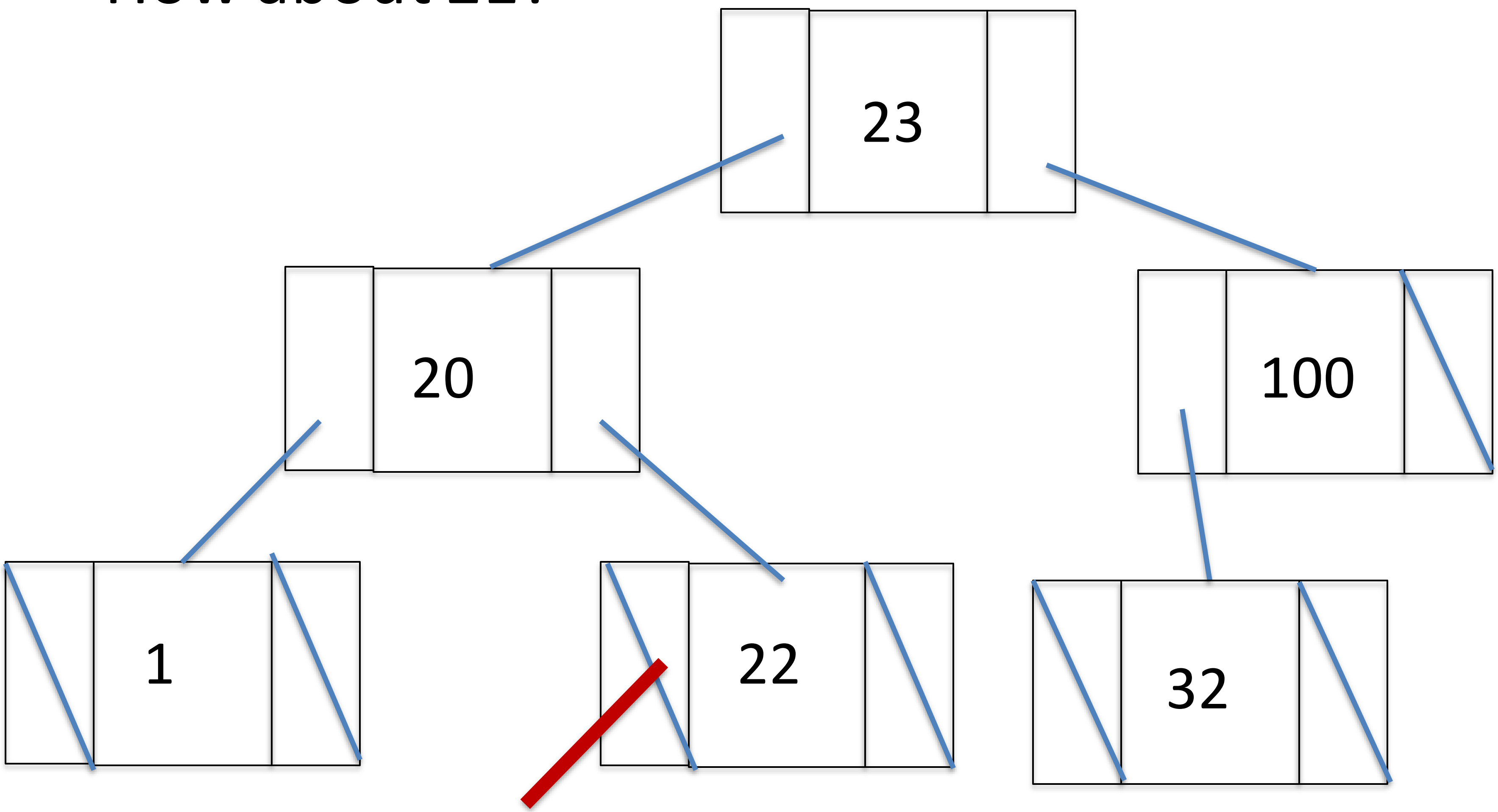
Inserting into BSTs

- Insert 32 into this tree…



Inserting Into a BST

- How about 21?



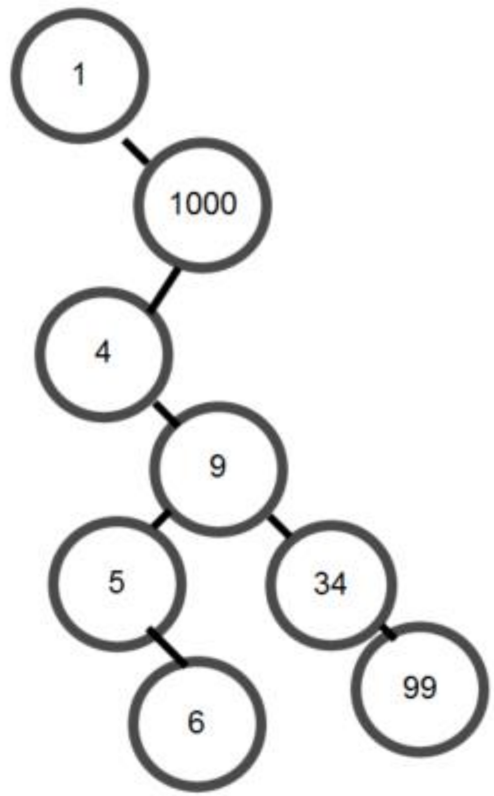
Exercise

Assuming the following numbers are sequentially added to a Binary Search Tree:

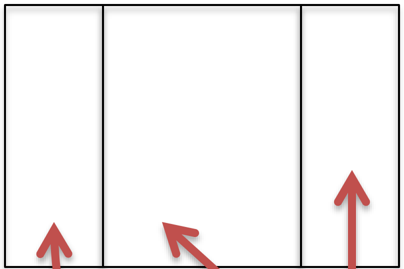
1, 1000, 4, 9, 5, 6, 34, 99, 1000

Draw the resulting Tree

Solution

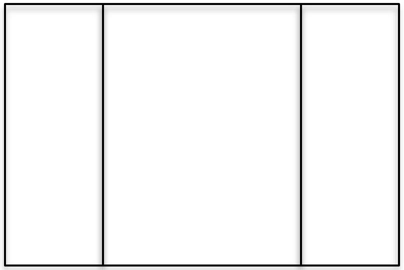


Binary Tree Class



```java
public class BinarySearchTree {

    private class BSTNode{

        private int value;
        private BSTNode leftChild;
        private BSTNode rightChild;

        public BSTNode(int v){
                value = v;
                leftChild = null;
                rightChild = null;
        }

        public BSTNode getLeftChild(){
            return leftChild;
        }
        public BSTNode getRightChild(){
            return rightChild;
        }
        public void setLeftChild(BSTNode n){
            leftChild = n;
        }
        public void setRightChild(BSTNode n){
                rightChild = n;
        }
        public int getValue(){
            return value;
        }

        //recursive implementations of operations
        void insertNode(BSTNode n){ ... }
        boolean containsNodeWithValue(int v){ ... }
    }

    private BSTNode rootNode = null;

    public void insertIntoTree(int v){
        if (rootNode == null){
            rootNode = new BSTNode(v);
        }else{
            rootNode.insertNode(new BSTNode(v));
        }
    }

    //Other operations go here

}
```

Binary Tree Class



```
public class BinarySearchTree {


    private class BSTNode{

        private int value;
        private BSTNode leftChild;
        private BSTNode rightChild;

        public BSTNode(int v){
                value = v;
                leftChild = null;
                rightChild = null;
        }

        public BSTNode getLeftChild(){
            return leftChild;
        }
        public BSTNode getRightChild(){
            return rightChild;
        }
        public void setLeftChild(BSTNode n){
            leftChild = n;
        }
public void setRightChild(BSTNode n){
            rightChild = n;
        }
        public int getValue(){
            return value;
        }

        //recursive implementations of operations
        void insertNode(BSTNode n){ ... }
        boolean containsNodeWithValue(int v){ ... }
    }

    private BSTNode rootNode = null;

    public void insertIntoTree(int v){
        if (rootNode == null){
            rootNode = new BSTNode(v);
        }else{
            rootNode.insertNode(new BSTNode(v));
        }
    }

    //Other operations go here

}
```

Eclipse Demo - insertion

Insert BST in Code

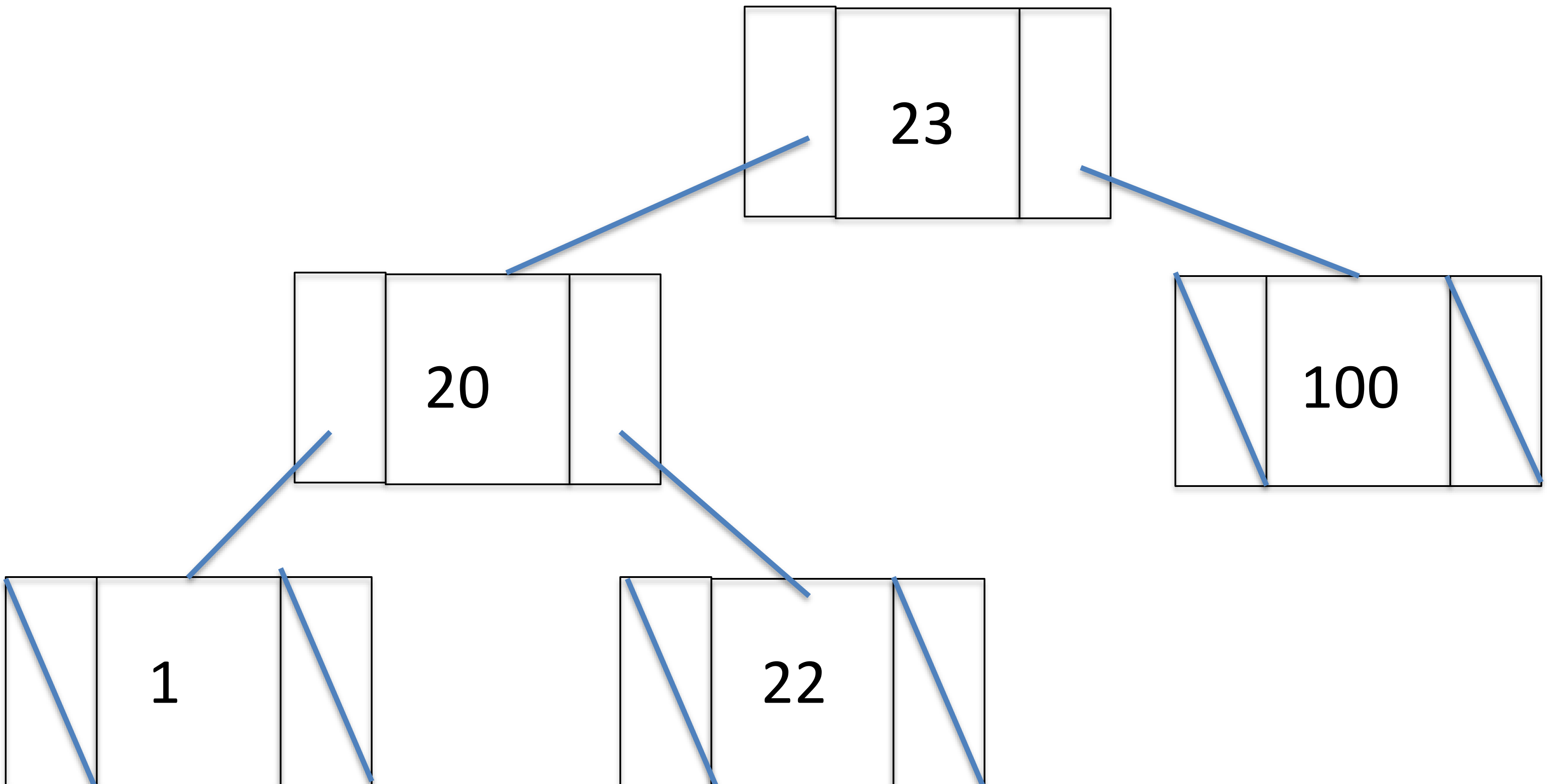
```
void insertNode(BSTNode n){
	if(n.value < this.value){
		if(this.getLeftChild() == null){
			this.setLeftChild(n);
		}else{
			this.getLeftChild().insertNode(n);
		}
	}else if(n.value > this.value){
		if (this.getRightChild() == null){
			this.setRightChild(n);
		}else{
			this.getRightChild().insertNode(n);
		}
	}else{
		return; //At this point n.value == this.value
				//could handle this in multiple ways
				//silently failing probably isn't the best
	}
}
```

Find in BST

- Same procedure as insert
  - Start at the root node
  - Work left and/or right until
    - You find the node with the correct value
    - You reach a leaf node (failure to find)

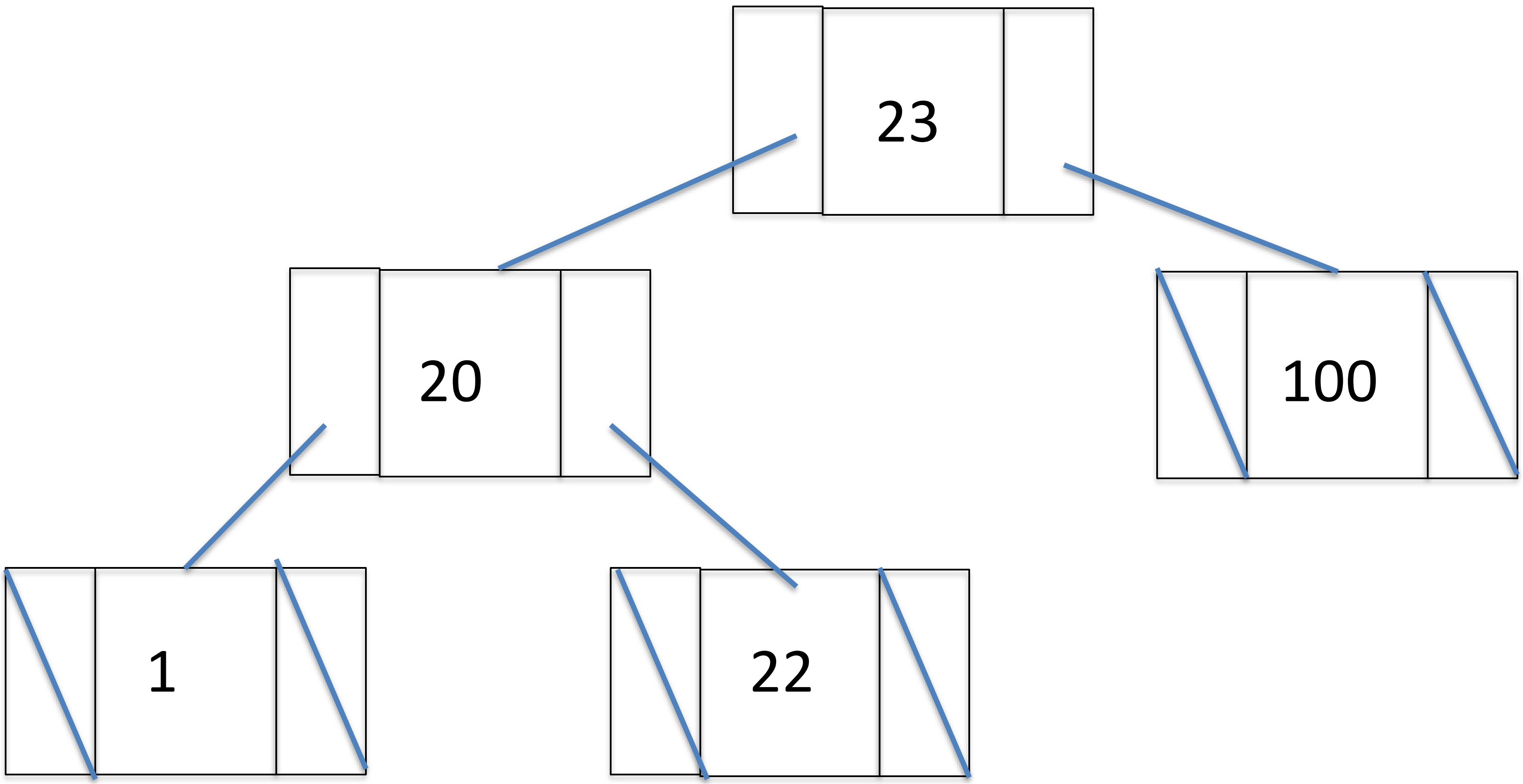
Find in BST

- Does this BST contain 22?



Find in BST

- Does this BST contain 9?



Eclipse Demo - searching

Find Value in BST Code

```
boolean containsNodeWithValue(int v){
        if(v < this.value){
            if(this.getLeftChild() == null){
                return false;
            }else{
                this.getLeftChild().containsNodeWithValue(v);
            }
        }else if(v > this.value){
            if (this.getRightChild() == null){
                return false;
            }else{
                this.getRightChild().containsNodeWithValue(v);
            }
        }else{
            return true;
        }
    }
```

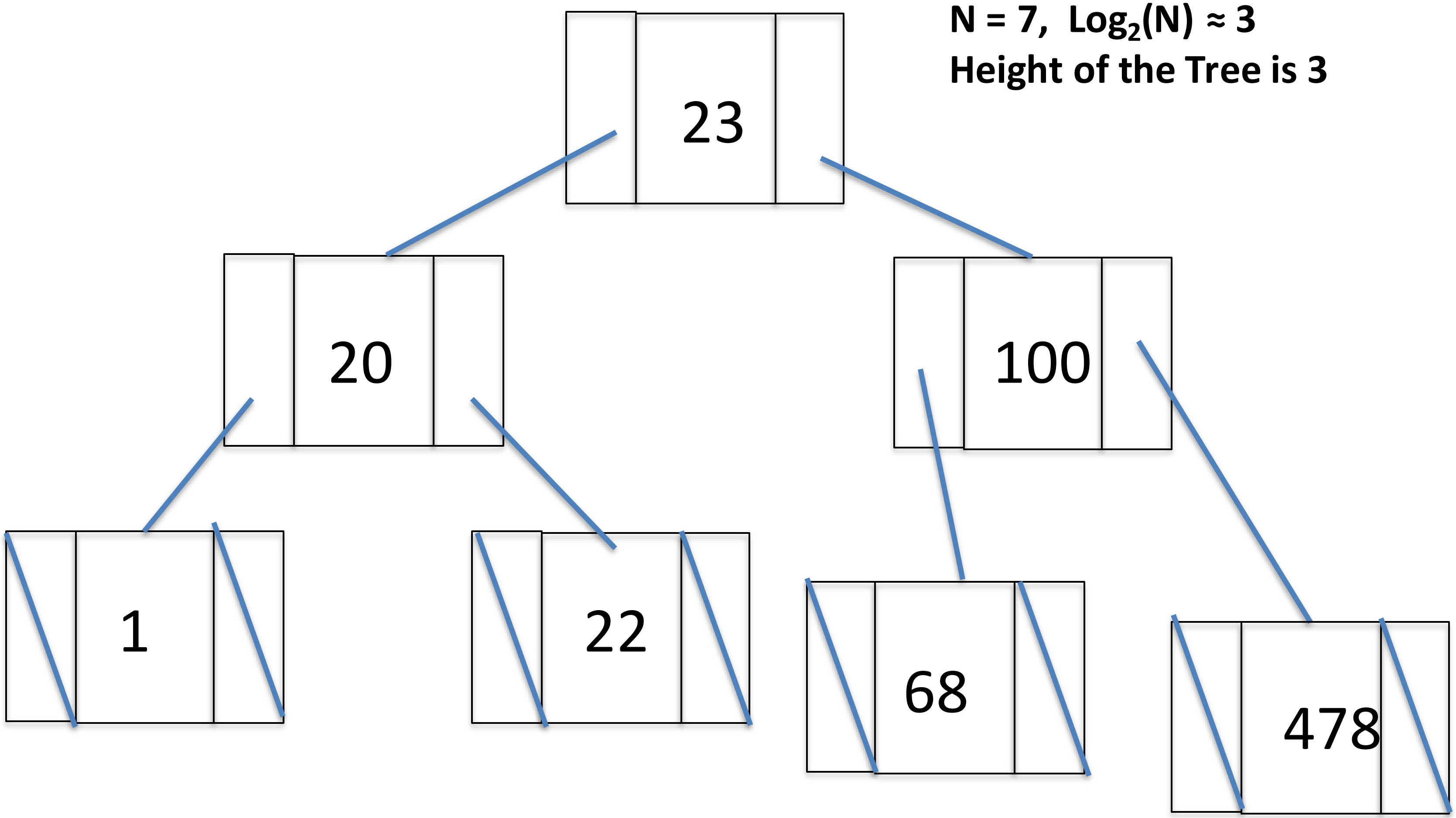
- Does this remind you of something we have seen before?

BST Adding and Finding

- Binary Search Trees are great for Searching
  - Its in the name
  - Generally efficient
    - No explicit sorting cost. **The tree is always ordered**
    - Average time complexity O(log n)
    - Traversals are O(n)
  - But you need to be careful!

Balanced Tree

$N = 7$, $Log_2(N) \approx 3$
Height of the Tree is 3



What about this tree

- Lets try manually building the tree
- Once like:
  - 23,20,1,100,22,478,68
- And once like:
  - 1, 20, 22, 23, 68, 100, 478

- What do you notice?

Pathological Case

- The order you insert values matters
  - Still have a valid tree.
  - Trees can **look different** but **be equivalent**
- Time complexity depends on its height.
  - Can vary wildly
- We want trees to be as balanced as possible
- Next year you will see examples of enforcing balanced trees (AVL trees)

Deletion

- Three Cases
  - 2 are easy
    - Delete a leaf node
    - Delete node with 1 subtree
  - 1 is hard
    - Delete node with 2 subtrees

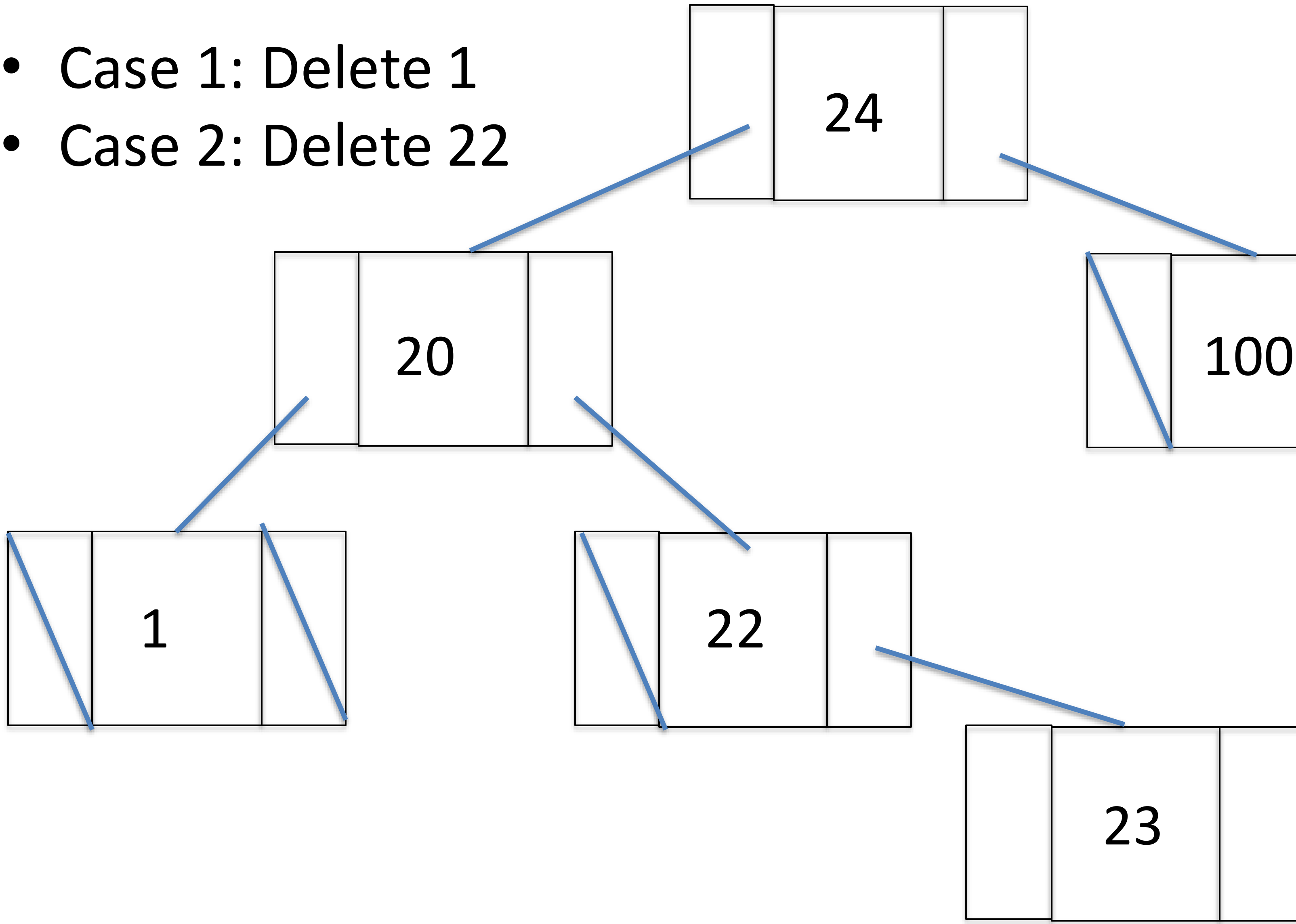
Deletion: easy cases

1. Leaf Node:
   – Find Node
   – Set appropriate left/right child of parent to be null
2. Node with 1 subtree:
   – Find Node
   – Set parent’s left/right subtree to be deleted node’s subtree

- We can combine this in the same code

Deletion

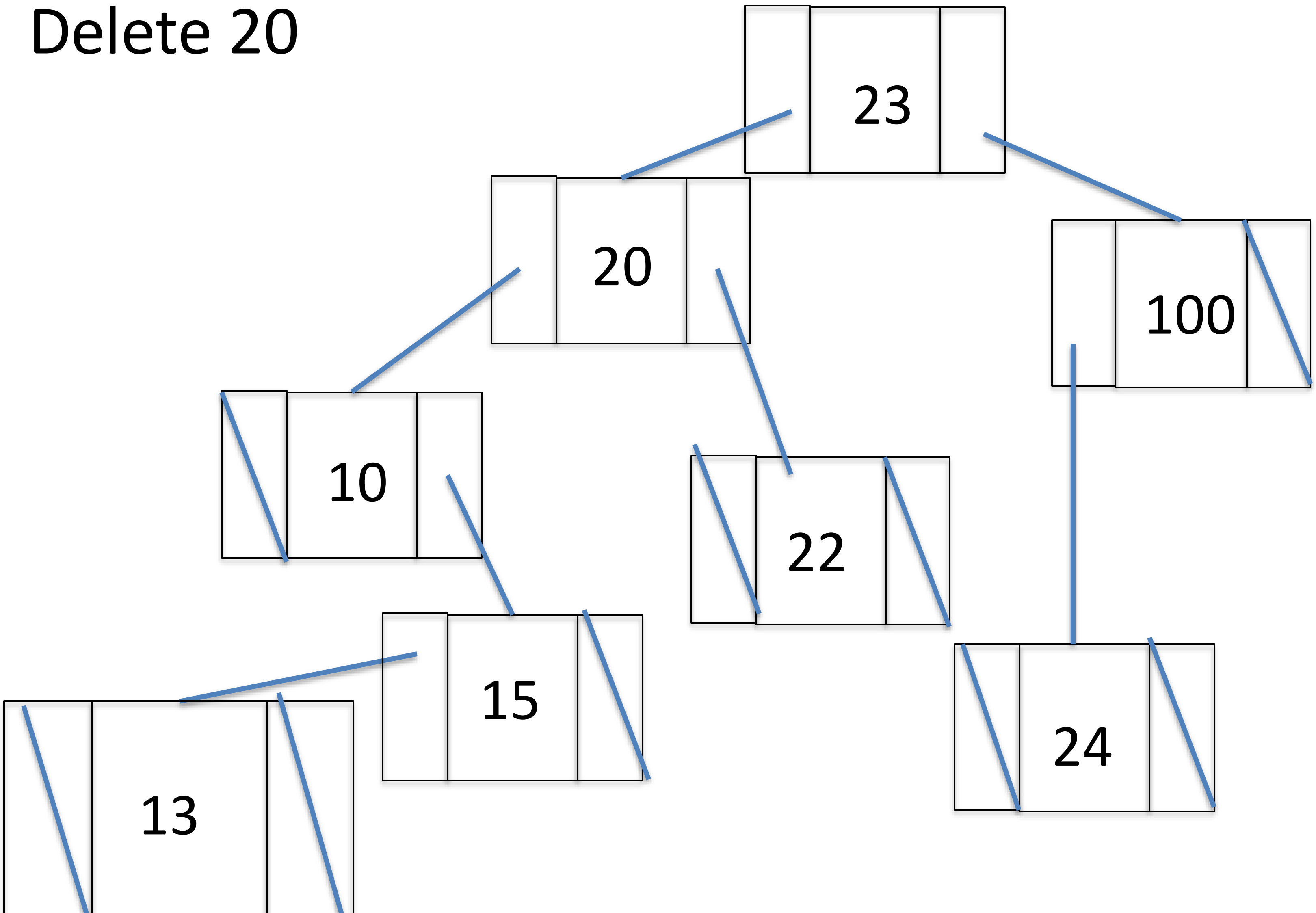
- Case 1: Delete 1
- Case 2: Delete 22



Eclipse Demo - deletion

Deletion Case: difficult case

- Delete 20



Deletion Case 3

- For the node you want to delete
  1. Find either the **largest** value in the **left** subtree
  2. Replace **value** of node you want to remove with **largest left** node value
     - We don’t replace the Node, we just change its value
  3. Travel back down the subtree and delete the **largest left** node

Eclipse Demo - deletion

Deletion Code

```
Public BSTNode deleteNodeWithValue(int v){
        if(v == this.value){ //this is the node we want to remove
                //does it have two children?
                if (this.getLeftChild() != null && this.getRightChild() != null){
                        //Yikes!
                }else if(this.getLeftChild() != null){
                                return this.getLeftChild();
                }else if(this.getRightChild() != null){
                                return this.getRightChild();
                }else{ // no children. this is a leaf node
                                return null;
                }
        }else{ //this isn't the node we want to remove
                if(v > this.value && this.getRightChild()!= null){
                   this.rightChild = this.getRightChild().deleteNodeWithValue(v);
                   return this;
                }else if(v < this.value && this.getLeftChild()!=null){
                   this.leftChild = this.getLeftChild().deleteNodeWithValue(v);
                   return this;
                }else{ //We are trying to remove a non existent node
                        return this;
                }
        }
    }
```

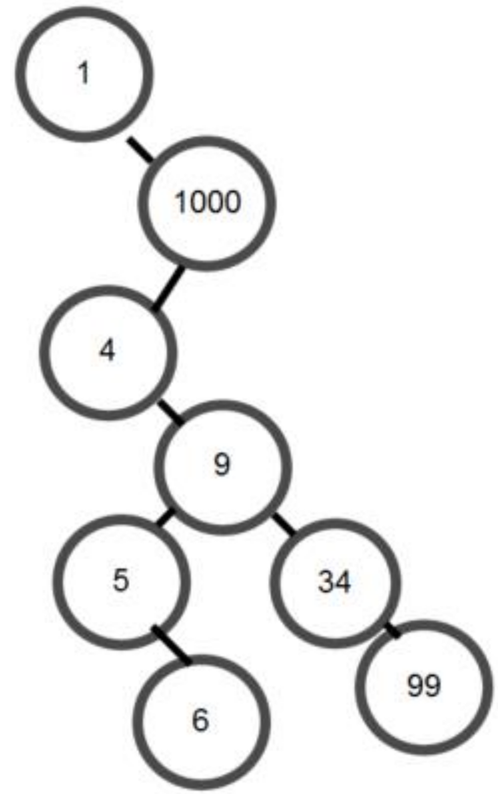
Deletion Case: difficult case

```
//The if statement from the last code snippet completed
if (this.getLeftChild() != null && this.getRightChild() != null){
                BSTNode largestLeft = this.getLeftChild().getLargestValueNode();
                this.value = largestLeft.value; //note we don't swap nodes.
                this.getLeftChild().deleteNodeWithValue(largestLeft.value);
                return this;
            }

//Also needs this implemented in BSTNode
//Trival to right the get smallest node version as well

public BSTNode getLargestValueNode(){
        if (this.getRightChild() == null){
                return this;
        }else{
            return this.getRightChild().getLargestValueNode();
        }
    }
```

Tree traversal exercise

Write down the order in which nodes would be visited in a

1. pre-order traversal
2. post-order traversal
3. in-order traversal



Binary Search Tree Space Race

Conclusions

- We have covered Binary Search Tree implementations
  - Representation in Java
  - Basic operations: search, add, remove
  - Shown you the pitfalls and how to do this in code
- Attendance sheet
- Next lecture: **priority queues**