CSE 332 Winter 2024
Lecture 7: Dictionaries, BSTs

MAP

Nathan Brunelle

http://www.cs.uw.edu/332

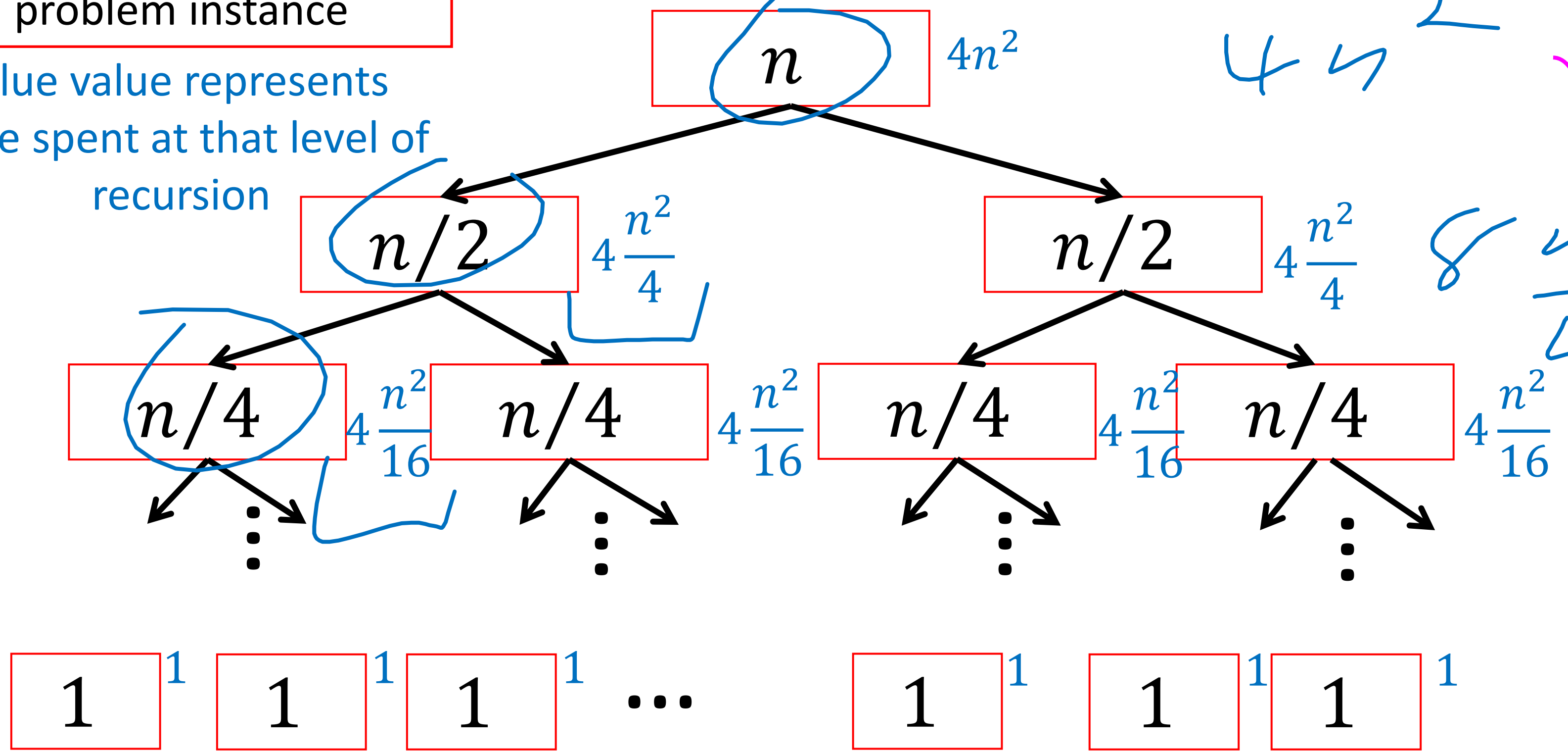
Warm Up: Tree Method

$T(1) = 1$

$$T(n) = 2T\left(\frac{n}{2}\right) + 4n^2$$

Red box represents a problem instance

Blue value represents time spent at that level of recursion



$4n^2$

$8\frac{n^2}{4}$

$\Rightarrow \boxed{4\frac{n^2}{4^i}}$ work at level i

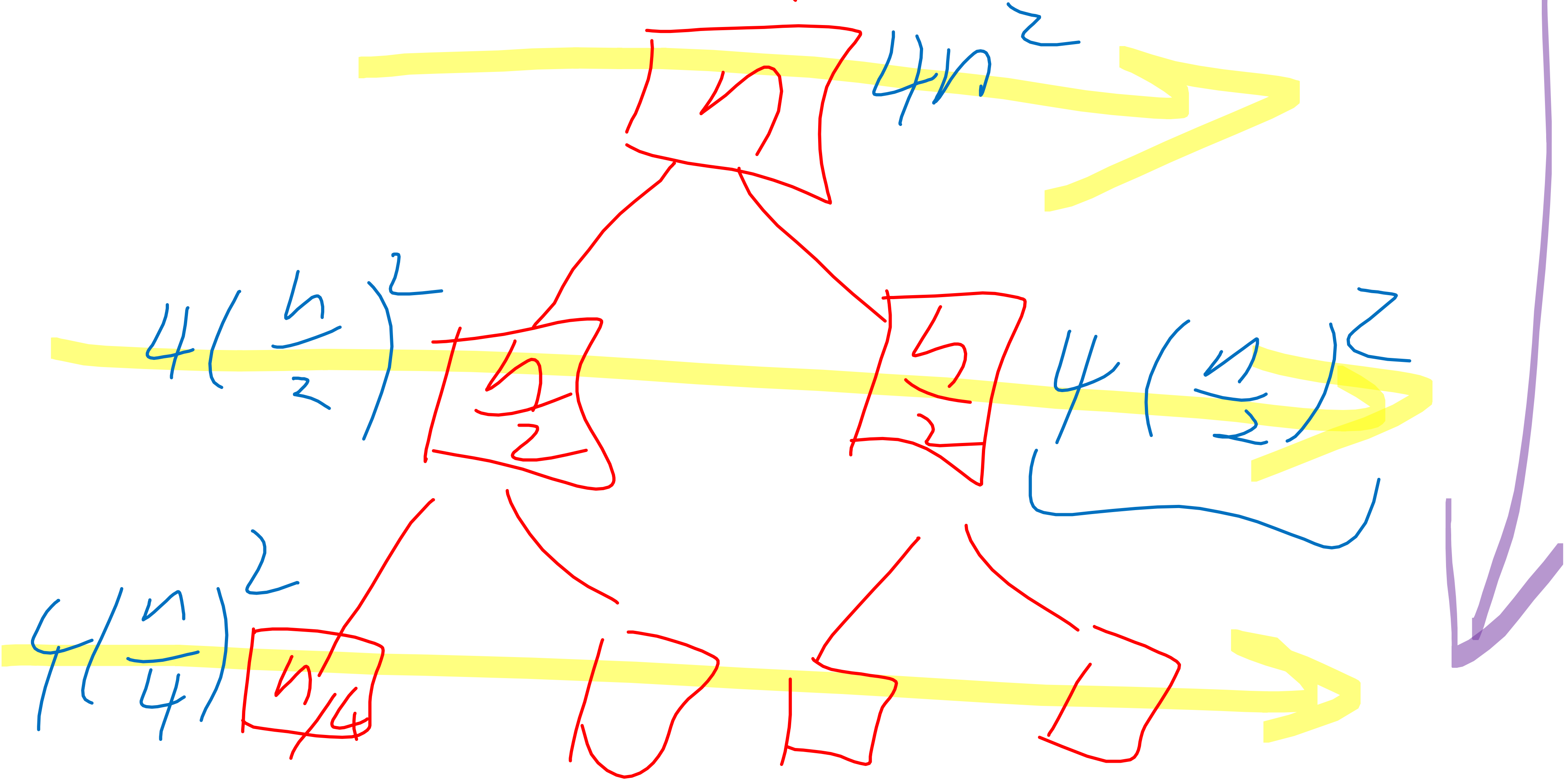
$4^i \frac{n^2}{4^i}$

$\log_2 n$ levels of recursion

16

$$T(n) = \sum_{i=1}^{\log_2 n} \boxed{4\frac{n^2}{4^i}} = 4n^2 \sum_{i=1}^{\log_2 n} \frac{1}{4^i} = \Theta(n^2)$$

$$T(n) = 2T\left(\frac{n}{2}\right) + 4n^2$$



$n \xrightarrow{1} \frac{n}{2} \xrightarrow{2} \frac{n}{4} \xrightarrow{3} \frac{n}{8} \ldots$

$$\frac{n}{2^1} \qquad \frac{n}{2^2} \qquad \frac{n}{2^3}$$

$i \longrightarrow \frac{n}{2^i}$ $\qquad 2^i = n$

Warm Up: Which is better?

Floyd's

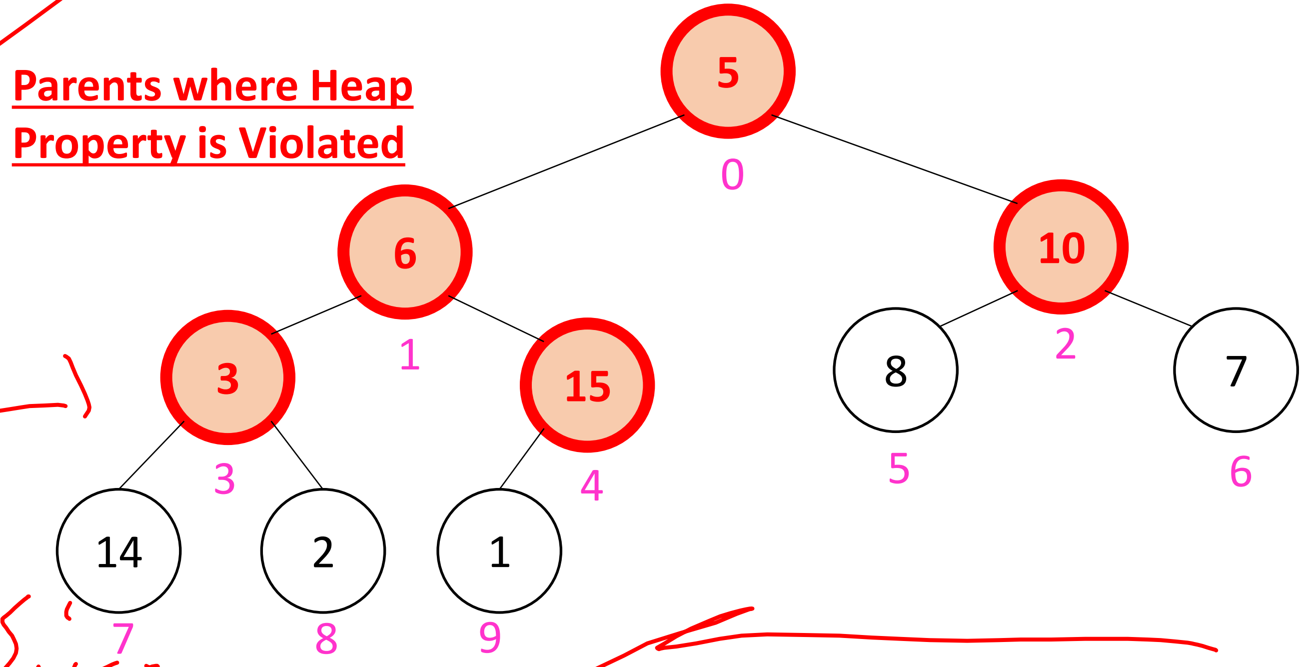
Both of the following build a binary heap within an unordered array.
Which is better?

Θ(n)

```
buildHeapDown(arr){
   for(int i = arr.length; i>0; i--){
      percolateDown(arr, i);
   }
}
```

leaves are expensive

```
buildHeapUp(arr){
   for(int i = 0; i<arr.length; i++){
      percolateUp(arr, i);
   }
}
```



5	6	10	3	15	8	7	14	2	1
0	1	2	3	4	5	6	7	8	9

Dictionary (Map) ADT

word → def

key

value

- Contents:
  - Sets of key+value pairs
  - Keys must be comparable
- Operations:
  - insert(key, value)
    - Adds the (key,value) pair into the dictionary
    - If the key already has a value, overwrite the old value
      - Consequence: Keys cannot be repeated
  - find(key)
    - Returns the value associated with the given key
  - delete(key)
    - Remove the key (and its associated value)

Trie

Naïve attempts

Data Structure	Time to insert	Time to find	Time to delete
Unsorted Array	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Unsorted Linked List	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Sorted Array	$\boxed{\Theta(n)}$	$\boxed{\Theta(\log n)}$	$\boxed{\Theta(n)}$
Sorted Linked List	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$

Less Naïve attempts

- Binary Search Trees (today)
- Tries (Section week 1, Project 1)
- AVL Trees (next week)
- B-Trees (next week)
- HashTables (week after)
- Red-Black Trees (not included in this course)
- Splay Trees (not included in this course)

Naïve attempts

Data Structure	Time to insert	Time to find	Time to delete
Unsorted Array	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Unsorted Linked List	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Sorted Array	$\Theta(n)$	$\Theta(\log n)$	$\Theta(n)$
Sorted Linked List	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Binary Search Tree (W.C.)	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Binary Search Tree (average)	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$

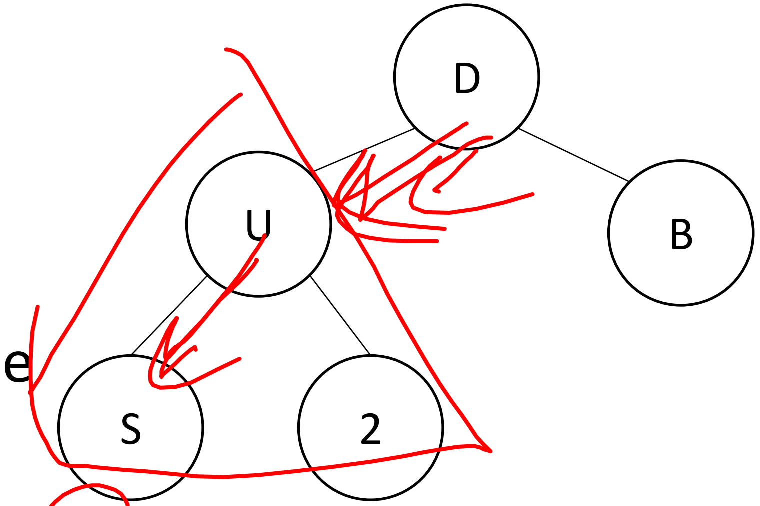
Tree Height

Counts edges

?

```
treeHeight(root){
        height = 0;
        if (root.left != Null){
                height = max(height, treeHeight(root.left));
        }
        if (root.right != Null){
                height = max(height, treeHeight(root.right));
        }
        return height;
}
```

More Tree "Vocab"



- Traversal:
  - An algorithm for "visiting/processing" every node in a tree
- Pre-Order Traversal:
  - Root, Left Subtree, Right Subtree

DUS2B

- In-Order Traversal:
  - Left Subtree, Root, Right Subtree

SU2DB

- Post-Order Traversal
  - Left Subtree, Right Subtree, Root

S2UBD

Name that Traversal!

```
AorderTraversal(root){
        if (root.left != Null){
                process(root.left);
        }
        if (root.right != Null){
                process(root.right);
        }
        process(root);
}
```

Post

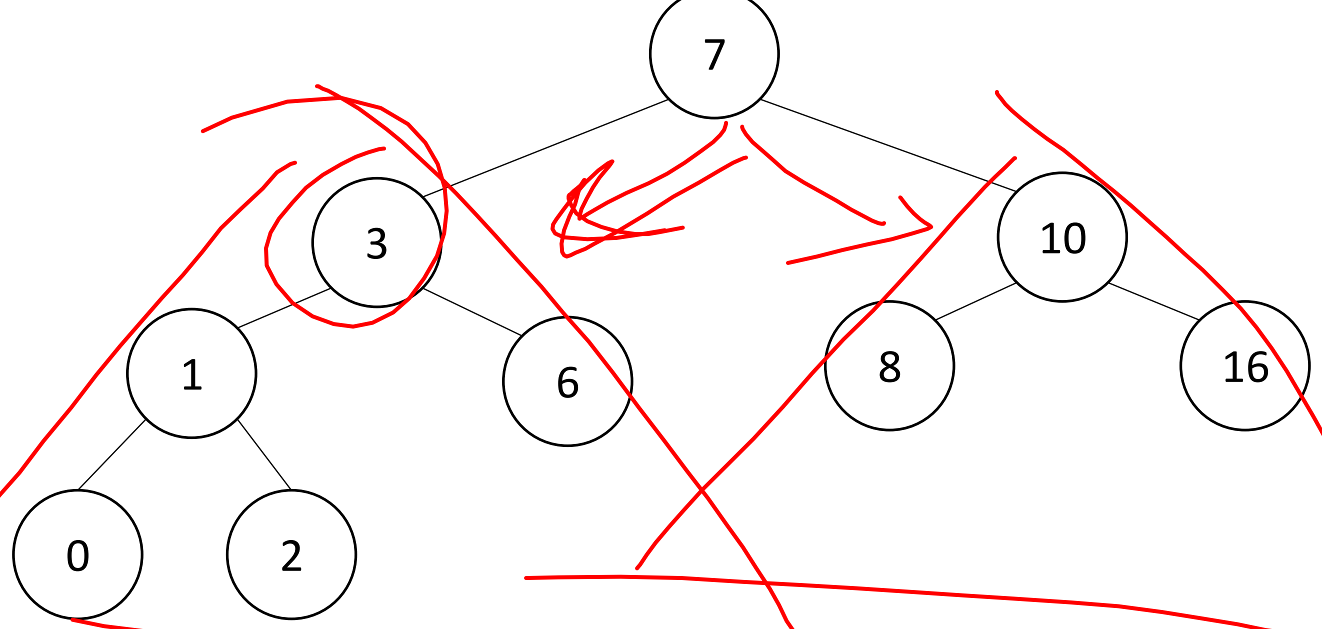
```
BorderTraversal(root){
        process(root);
        if (root.left != Null){
                process(root.left);
        }
        if (root.right != Null){
                process(root.right);
        }
}
```

Pre

```
CorderTraversal(root){
        if (root.left != Null){
                process(root.left);
        }
        process(root)
        if (root.right != Null){
                process(root.right);
        }
}
```

In

Binary Search Tree



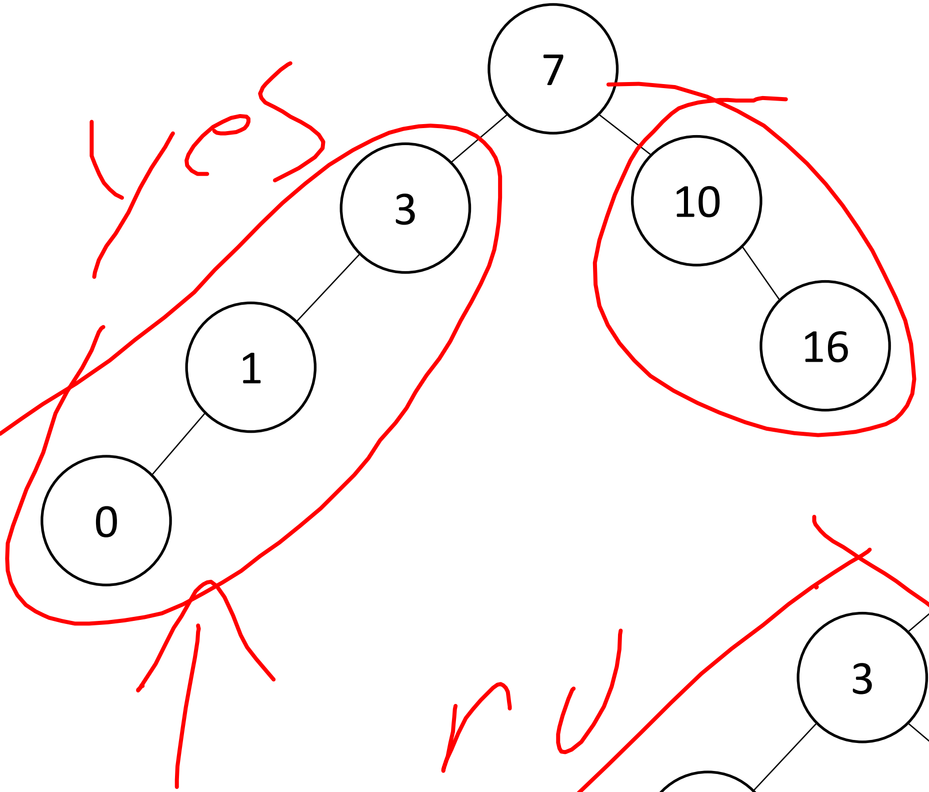
- Binary Tree
  - Definition:
- Order Property
  - All keys in the left subtree are smaller than the root
  - All keys in the right subtree are larger than the root

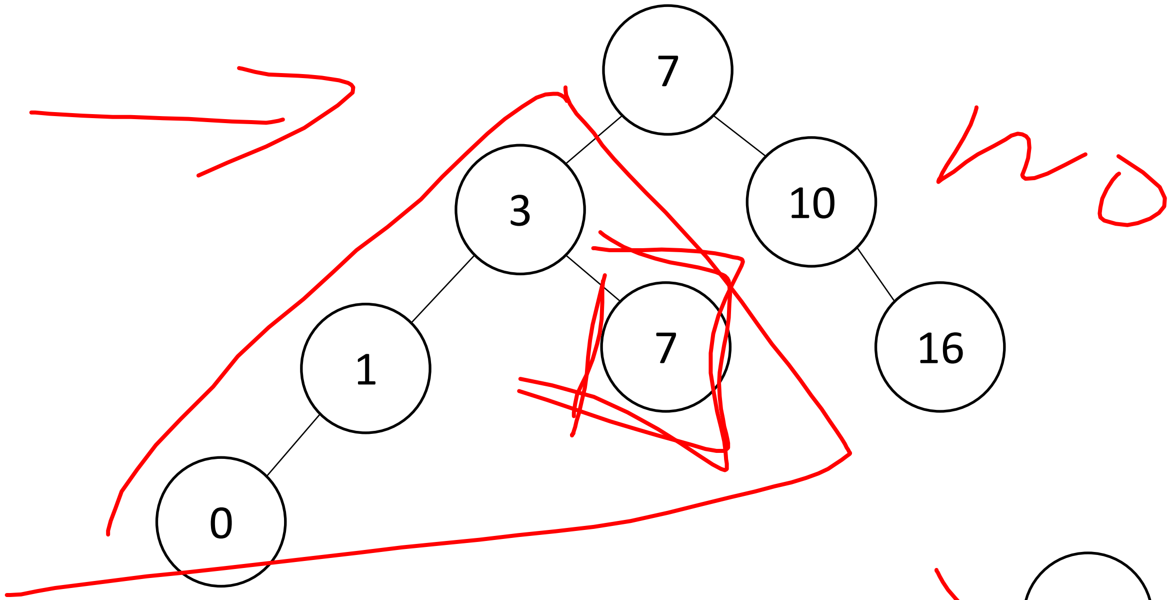
4

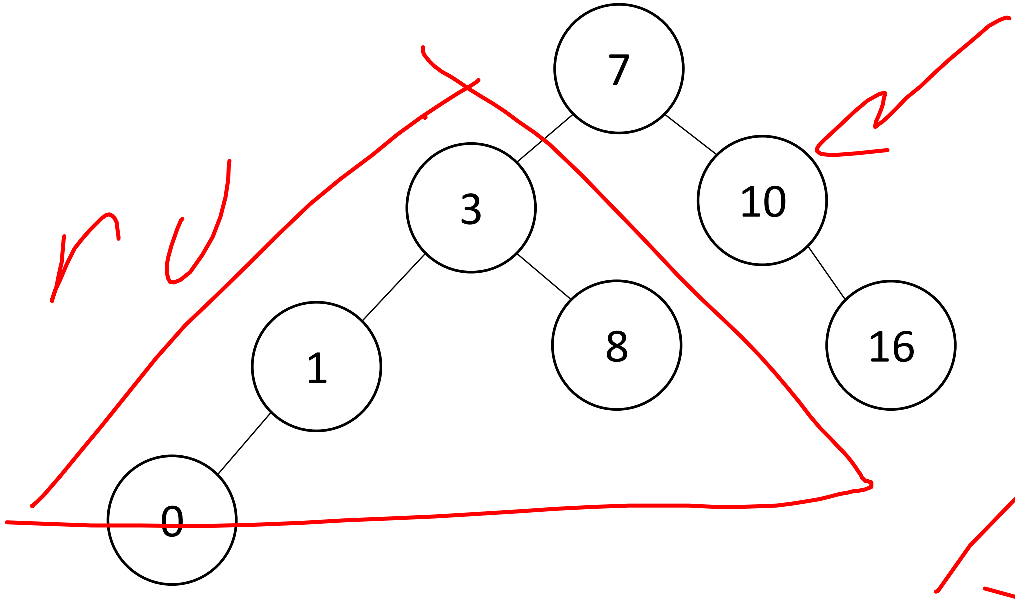
- Why?

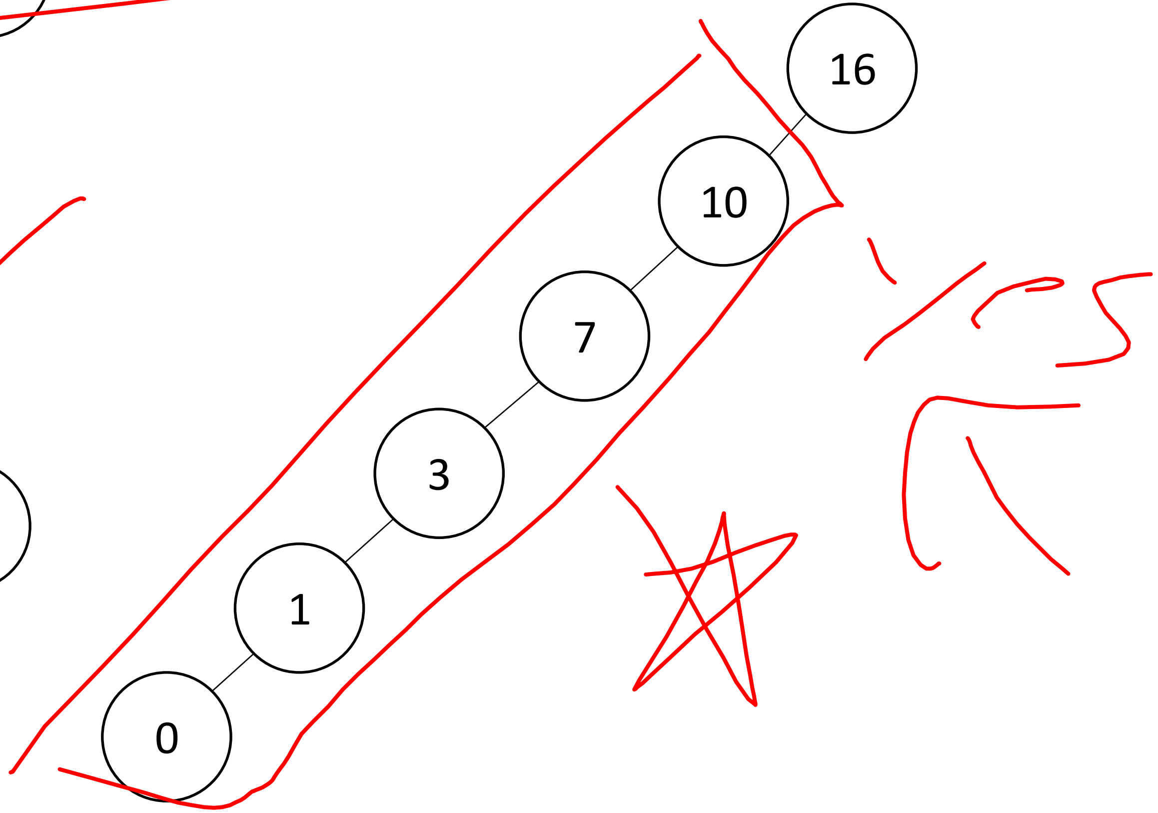
→ makes find faster

Are these BSTs?



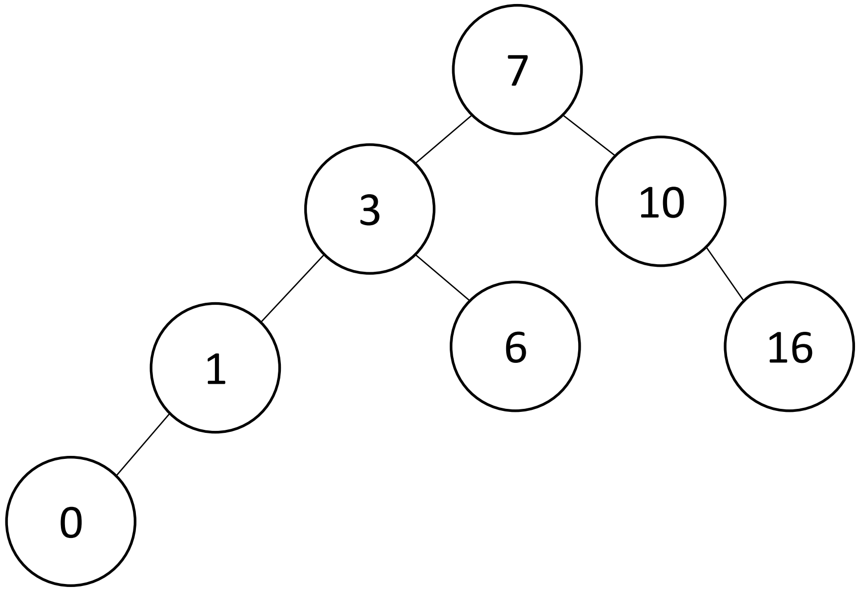






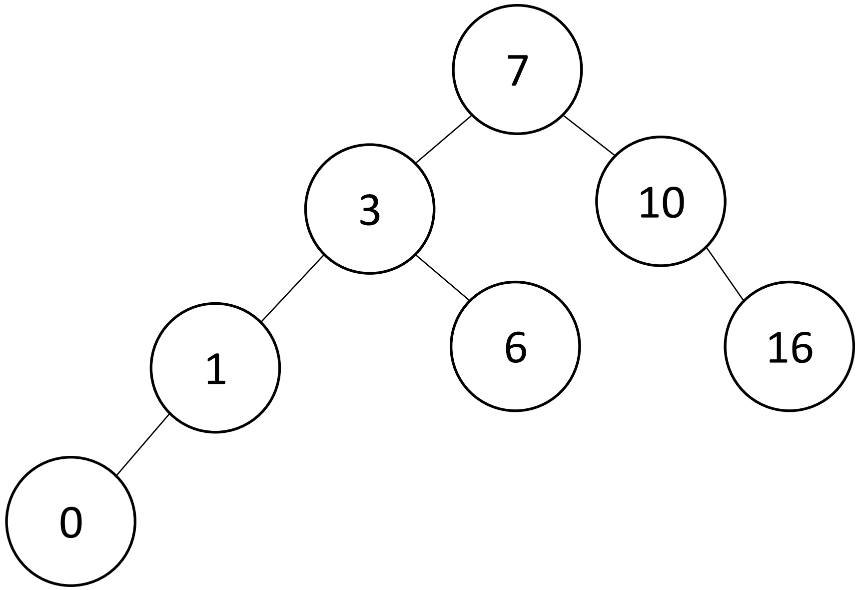
Find Operation (recursive)

```
find(key, root){
        if (root == Null){
                return Null;
        {
        if (key == root.key){
                return root.value;
        }
        if (key < root.key){
                return find(key, root.left);
        }
        if (key > root.key){
                return find(key, root.right);
        }
        return Null;
}
```



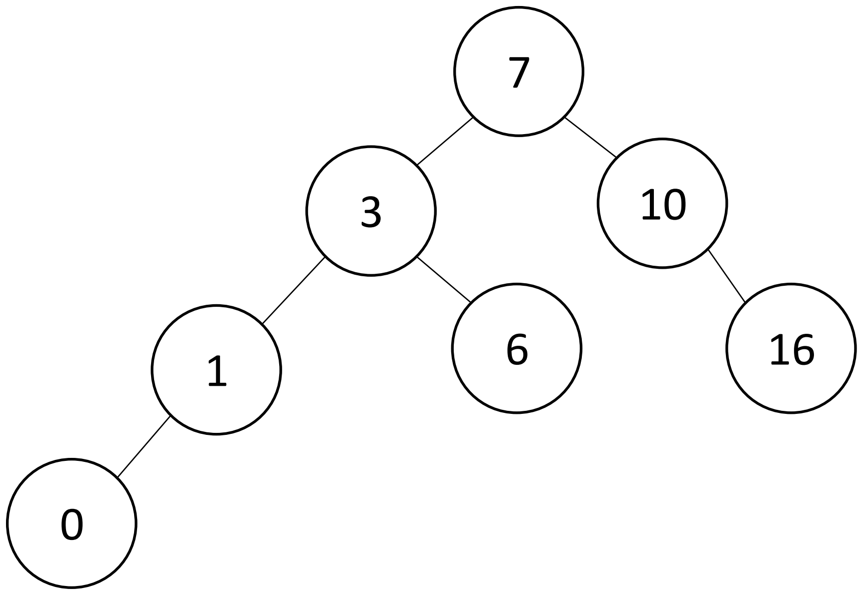
Find Operation (iterative)

```
find(key, root){
	while (root != Null && key != root.key){
		if (key < root.key){
			root = root.left;
		}
		else if (key > root.key){
			root = root.right;
		}
	}
	if (root == Null){
		return Null;
	}
	return root.value;
}
```



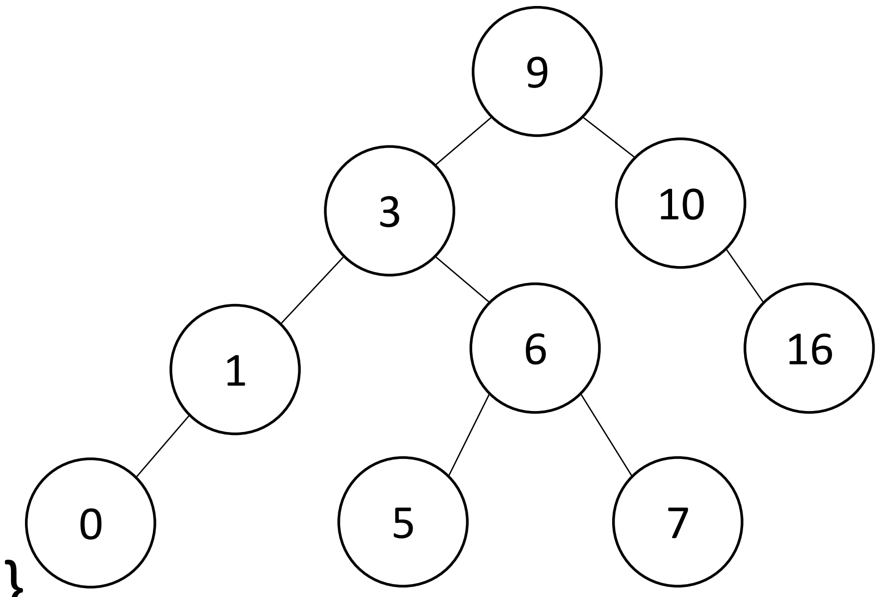
Insert Operation (iterative)

```
insert(key, value, root){
        if (root == Null){ this.root = new Node(key, value); }
        parent = Null;
        while (root != Null && key != root.key){
                parent = root;
                if (key < root.key){ root = root.left; }
                else if (key > root.key){ root = root.right; }
        }
        if (root != Null){ root.value = value; }
        else if (key < parent.key){ parent.left = new Node(key, value); }
        else{ parent.right = new Node (key, value); }
}
```



Note: Insert happens only at the leaves!

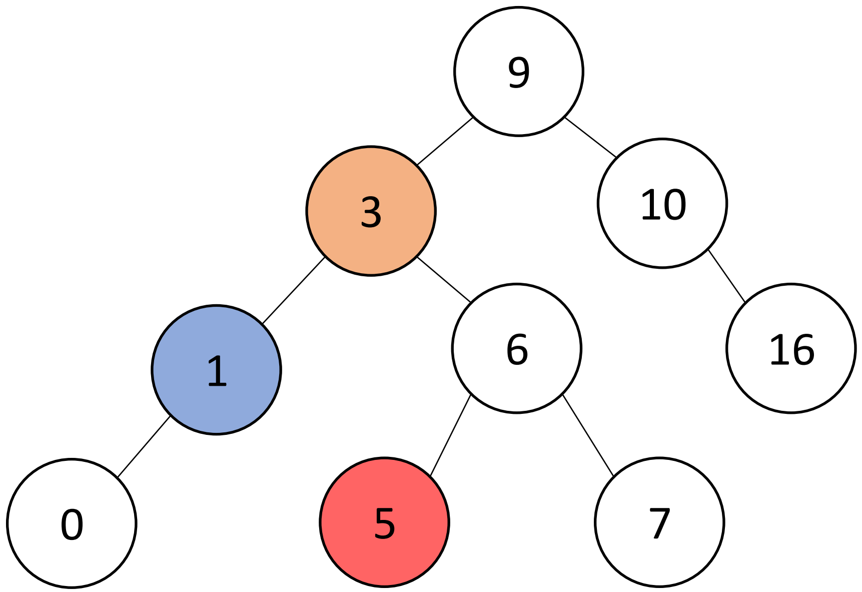
Delete Operation (iterative)



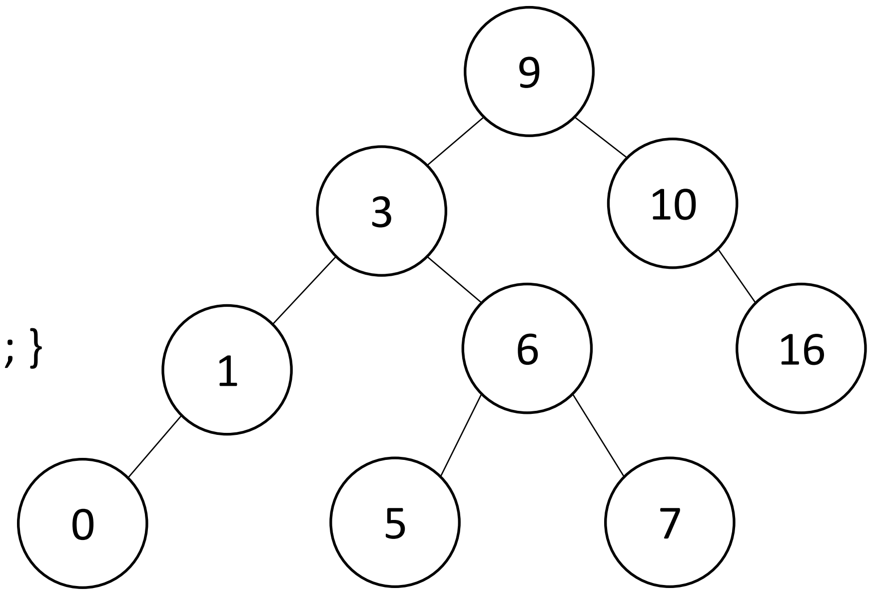

```
delete(key, root){
    while (root != Null && key != root.key){
        if (key < root.key){ root = root.left;      }
        else if (key > root.key){ root = root.right; }
    }
    if (root == Null){ return; }
    // Now root is the node to delete, what happens next?
}
```

Delete – 3 Cases

- 0 Children (i.e. it's a leaf)
- 1 Child
- 2 Children



Finding the Max and Min



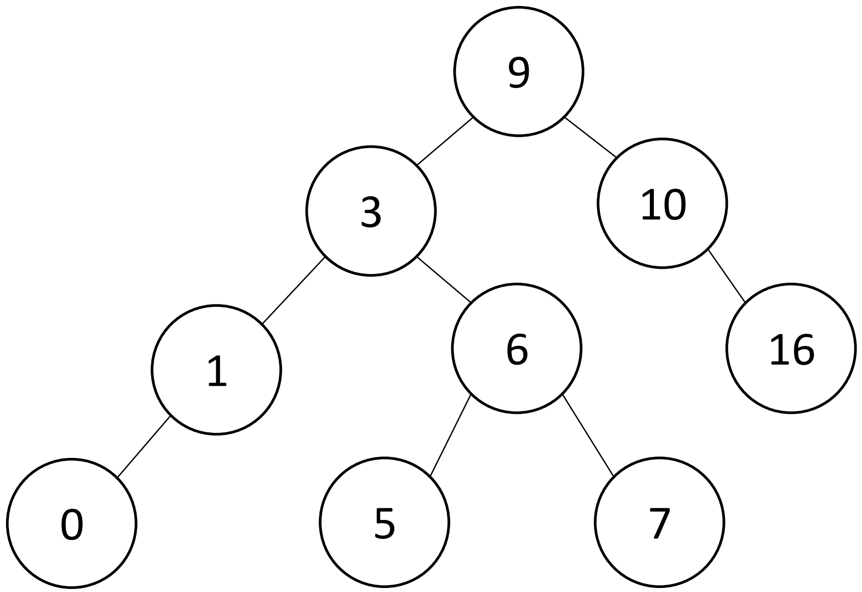
- Max of a BST:
  - Right-most Thing

```
maxNode(root){
        if (root == Null){ return Null; }
        while (root.right != Null){
                root = root.right;
        }
        return root;
}
```

- Min of a BST:
  - Left-most Thing

```
minNode(root){
        if (root == Null){ return Null; }
        while (root.left != Null){
                root = root.left;
        }
        return root;
}
```

Delete Operation (iterative)




```
delete(key, root){
        while (root != Null && key != root.key){
                if (key < root.key){ root = root.left;   }
                else if (key > root.key){ root = root.right; }
        }
        if (root == Null){ return; }
        if (root has no children){
                make parent point to Null Instead;
        }
        if (root has one child){
                make parent point to that child instead;
        }
        if (root has two children){
                make parent point to either the max from the left or min from the right
        }
}
```

Worst Case Analysis

- For each of Find, insert, Delete:
  - Worst case running time matches height of the tree
- What is the maximum height of a BST with n nodes?

Improving the worst case

- How can we get a better worst case running time?

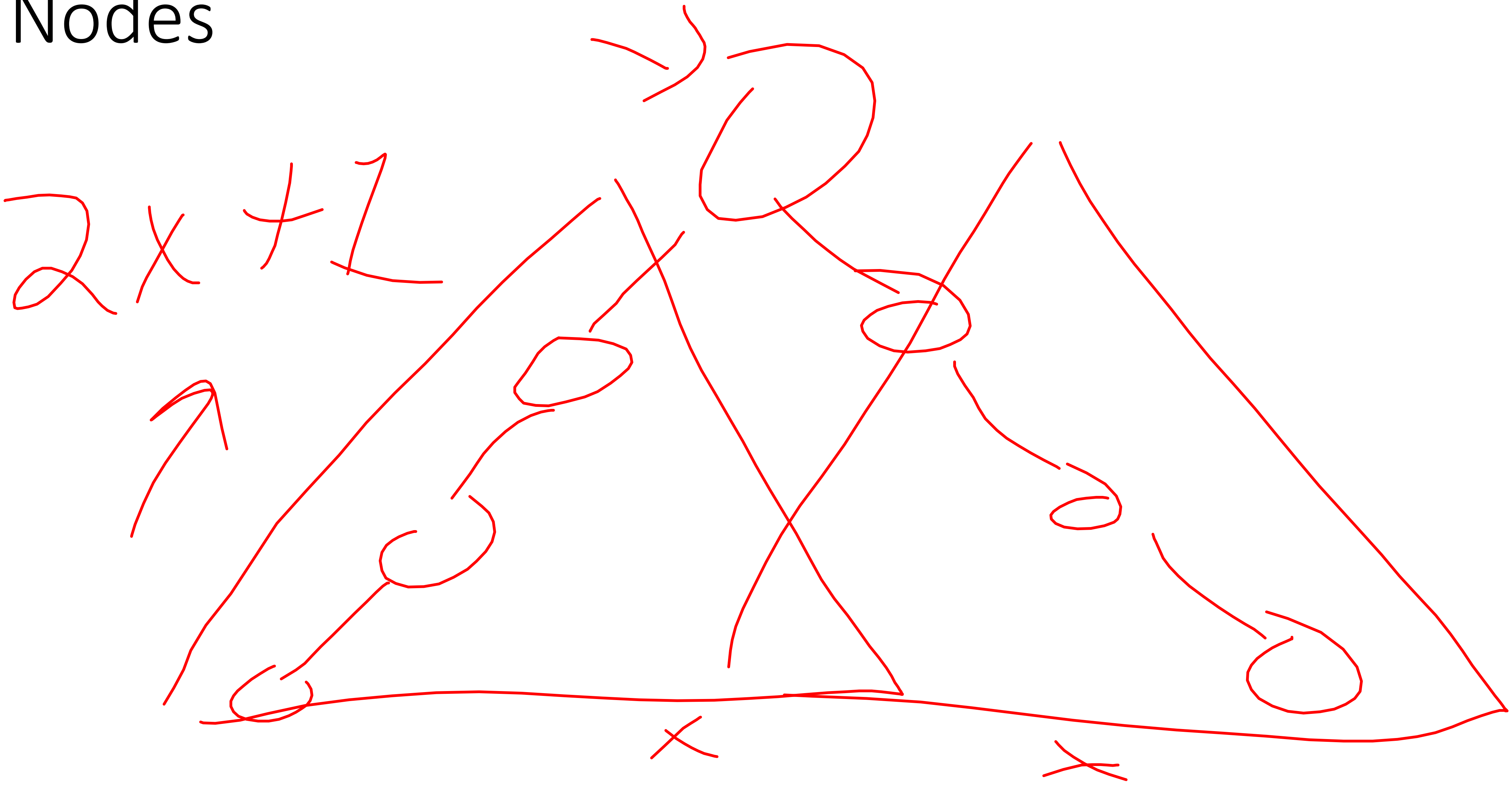
make our trees
more "full"

add shape rules
to binary tree

“Balanced” Binary Search Trees

- We get better running times by having “shorter” trees
- Trees get tall due to them being “sparse” (many one-child nodes)
- Idea: modify how we insert/delete to keep the tree more “full”

Idea 1: Both Subtrees of Root have same # Nodes



Idea 2: Both Subtrees of Root have same height

Idea 3: Both Subtrees of every Node have same # Nodes

Idea 4: Both Subtrees of every Node have same height

Teaser: AVL Tree

- A Binary Search tree that maintains that the left and right subtrees of every node have heights that differ by at most one.
  - Not too weak (ensures trees are short)
  - Not too strong (works for any number of nodes)