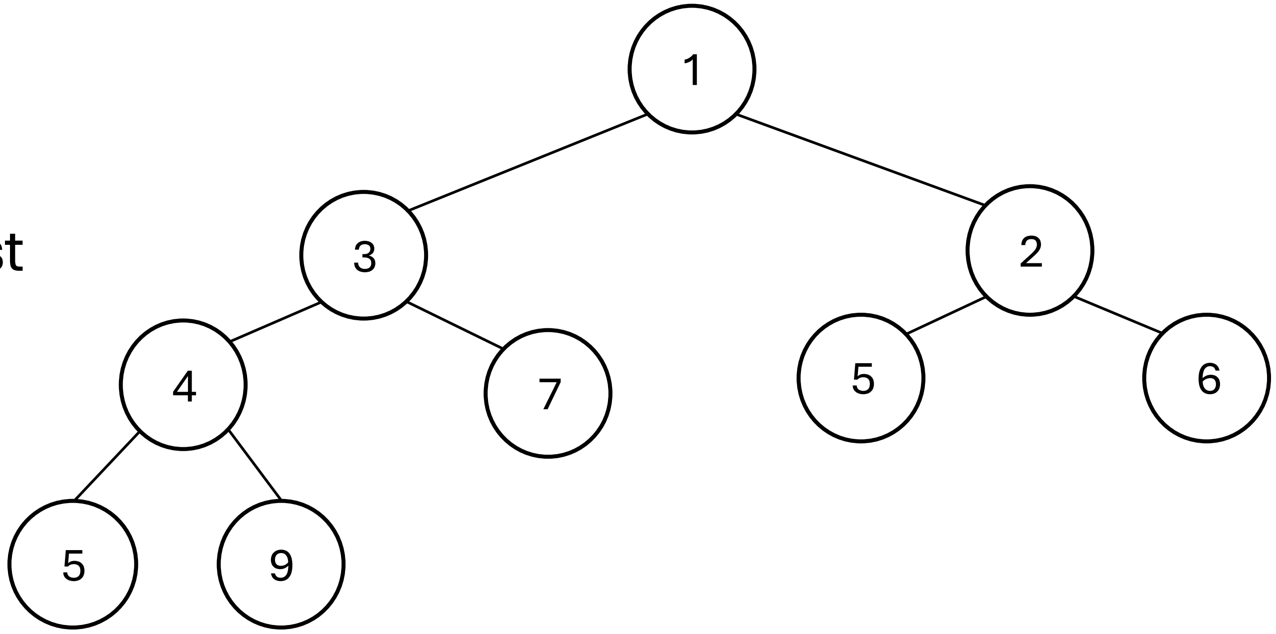
CSE 332 Summer 2026 Lecture 6: Heap Wrapup, Dictionaries

Michael Whitmeyer

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

Reminder: Heap Data Structure (for priority queue ADT)

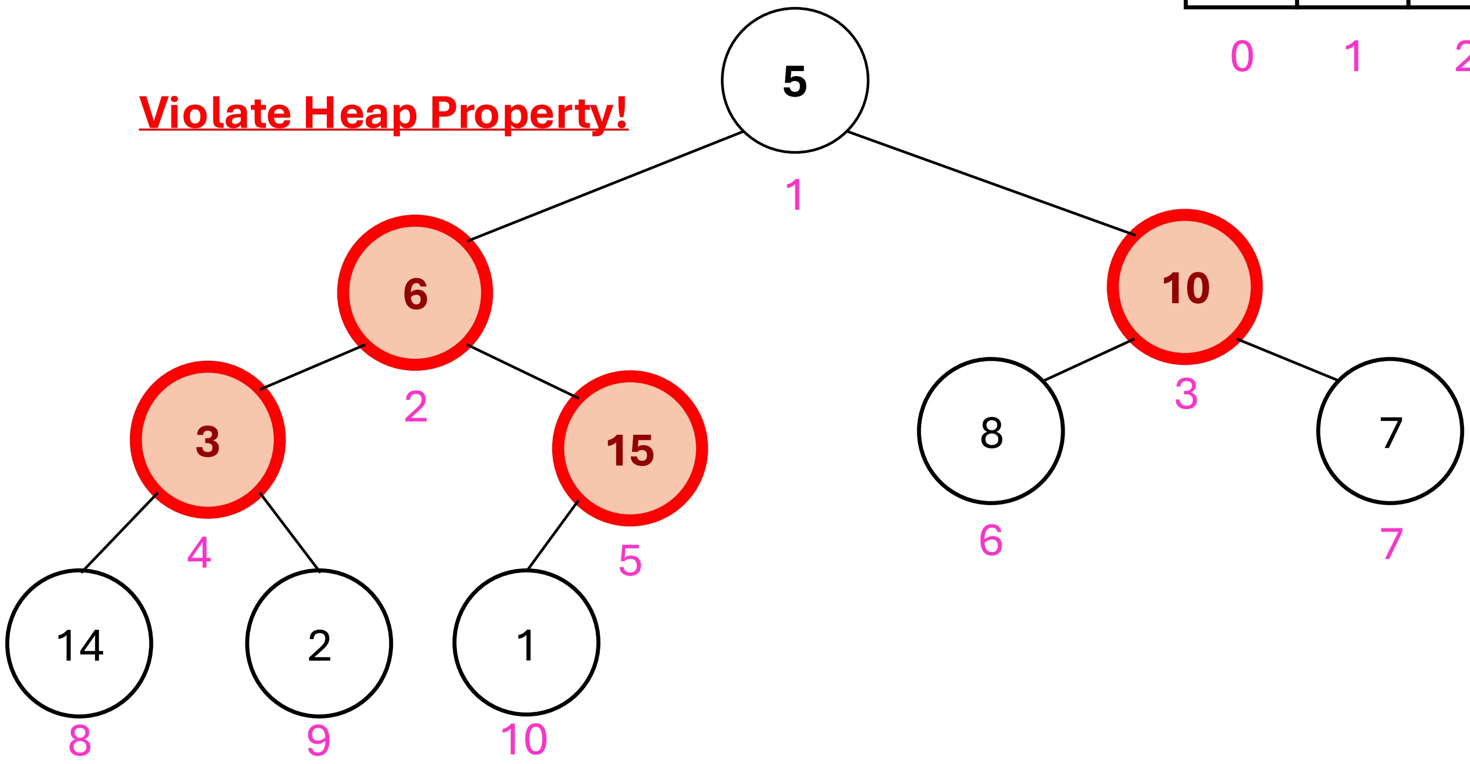
- **Invariant 1**: tree will be "complete"
  - All layers full except possibly last one, filled left to right
- **Invariant 2**: priority(parent) $\leq$ priority(children)
  - "Heap Property"
- Stored with an Array



Building a Heap From "Scratch"

- Suppose we had n items and wanted to "heapify" them

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



Two ways for "fix" the heap:
1) Percolate Up
2) Percolate Down

Floyd’s buildHeap method

- **Starting from the leaves**, one row at a time, percolate down

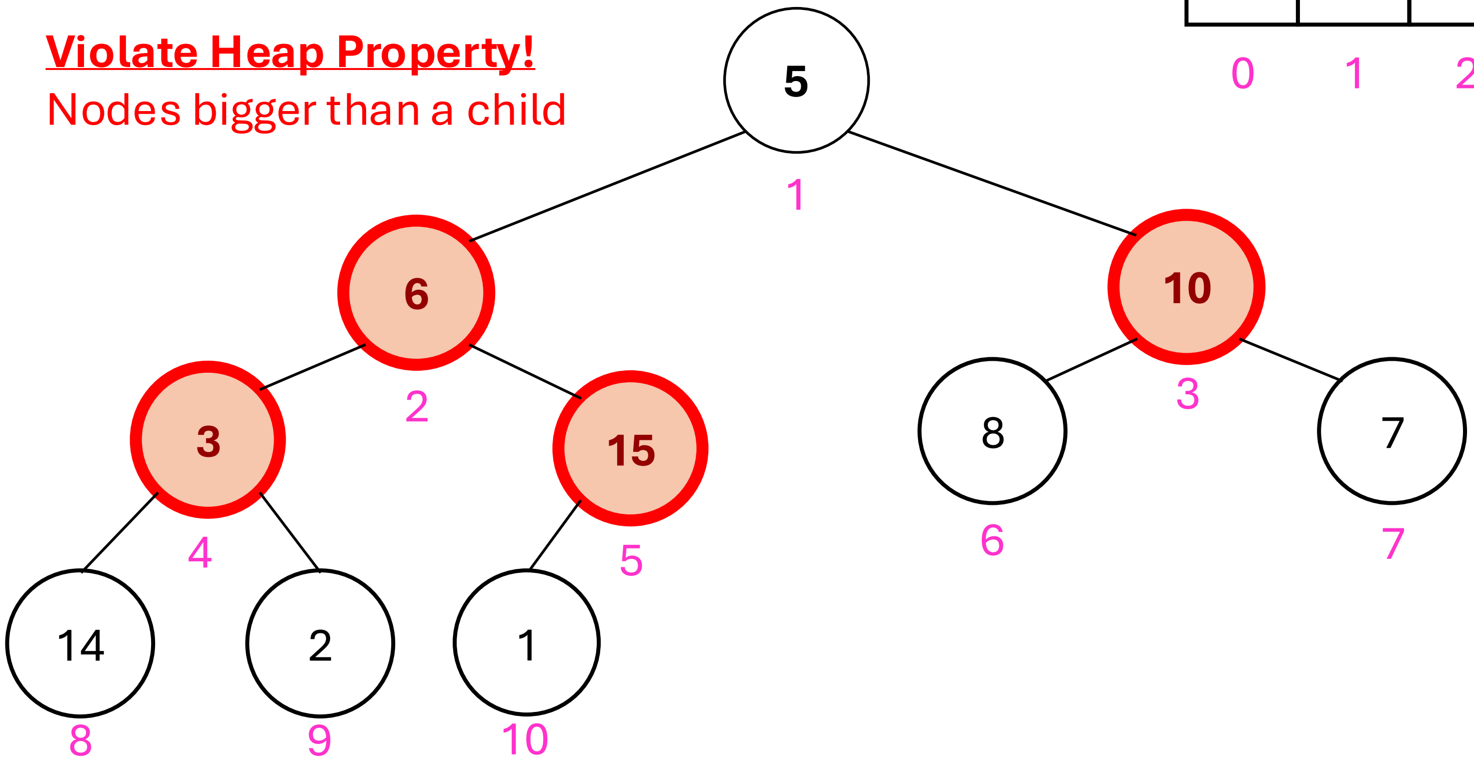
```
buildHeap(){
   for(int i = size; i>0; i--){
      percolateDown(i);
   }
}
```

Floyd’s buildHeap method (Percolate 15)

- Suppose we had n items and wanted to “heapify” them

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

<u>Violate Heap Property!</u>
Nodes bigger than a child

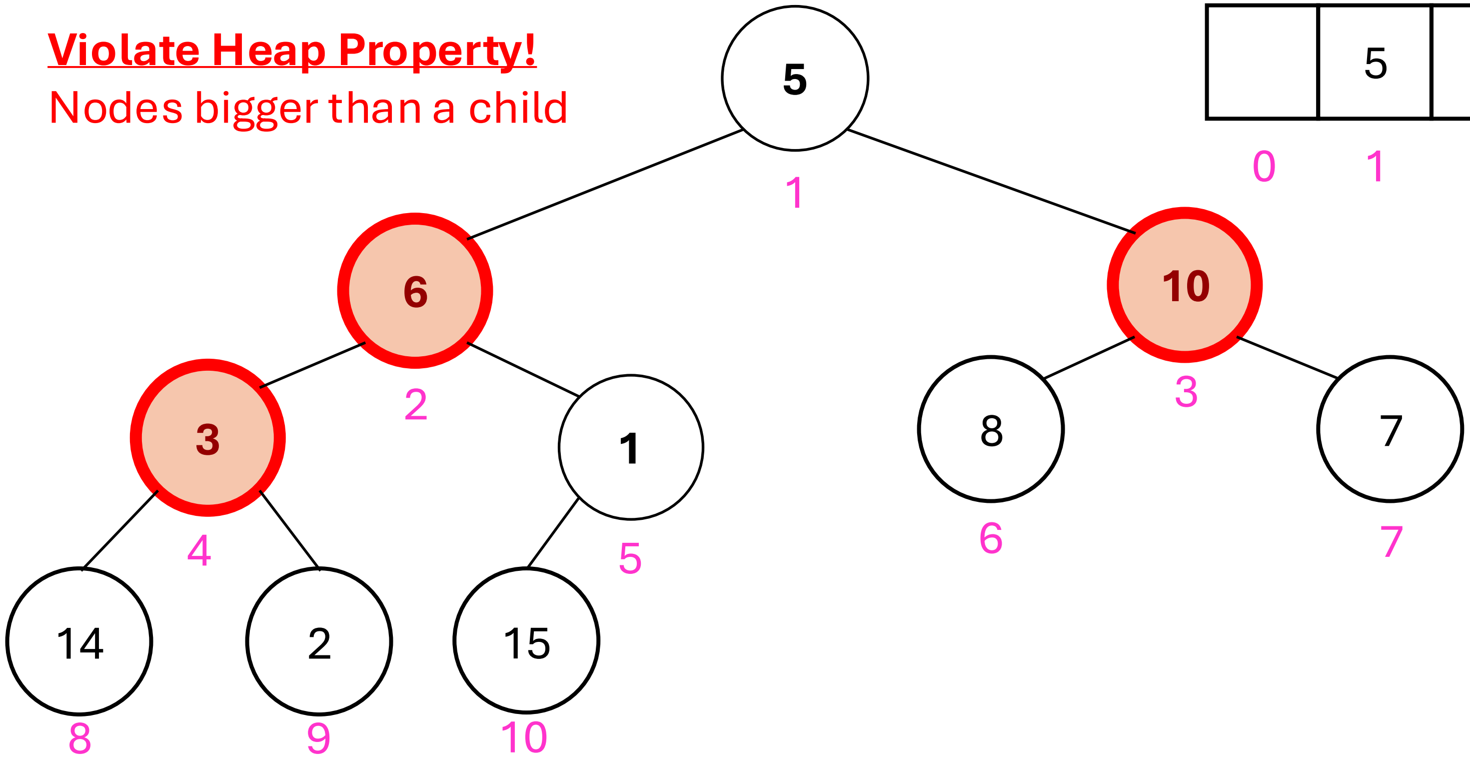



```
buildHeap(){
   for(int i = size; i>0; i--){
      percolateDown(i);
   }
}
```

Floyd's buildHeap method (Percolate 3)

- Suppose we had n items and wanted to "heapify" them

Violate Heap Property!
Nodes bigger than a child

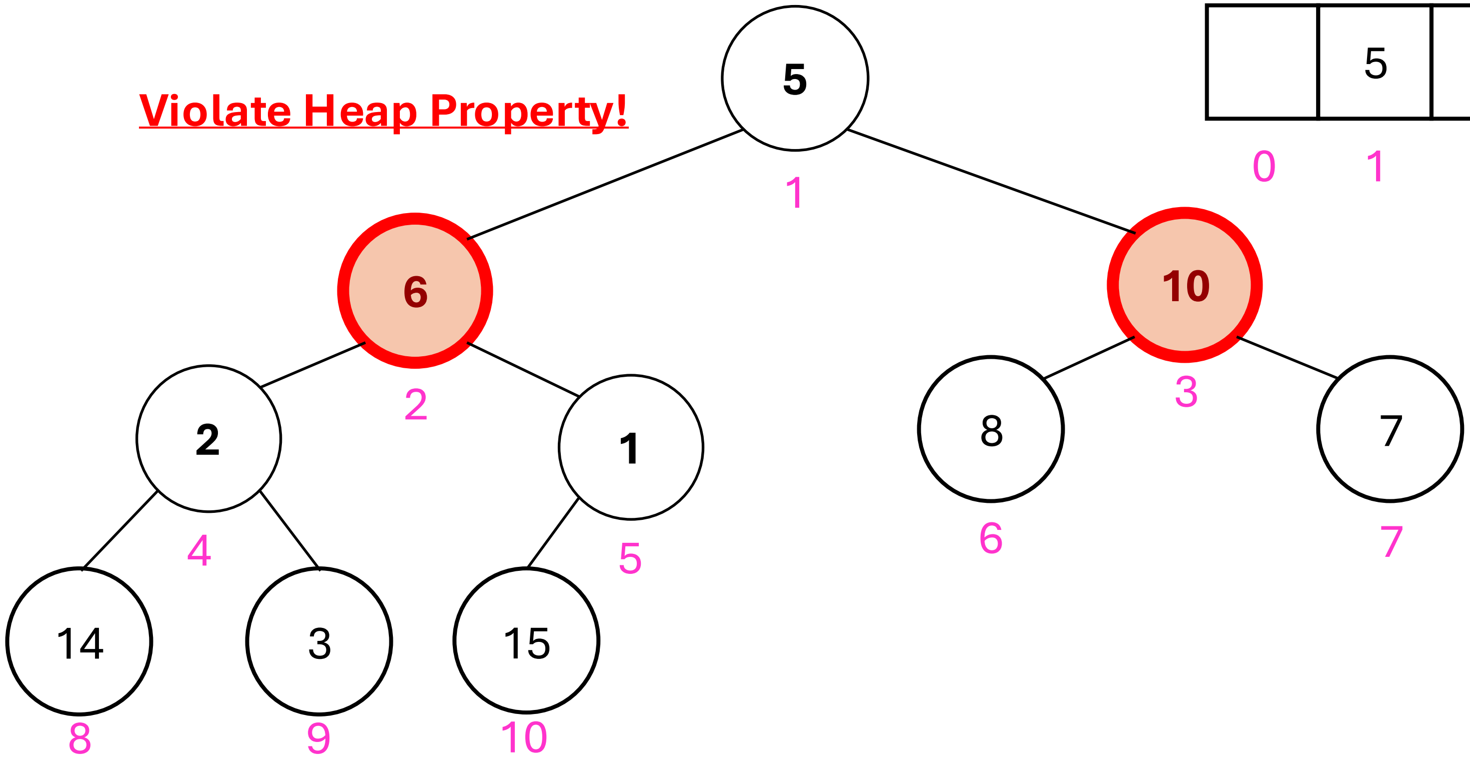


	5	6	10	3	1	8	7	14	2	15
0	1	2	3	4	5	6	7	8	9	10

```
buildHeap(){
   for(int i = size; i>0; i--){
      percolateDown(i);
   }
}
```

Floyd’s buildHeap method (Percolate 10)

- Suppose we had n items and wanted to “heapify” them

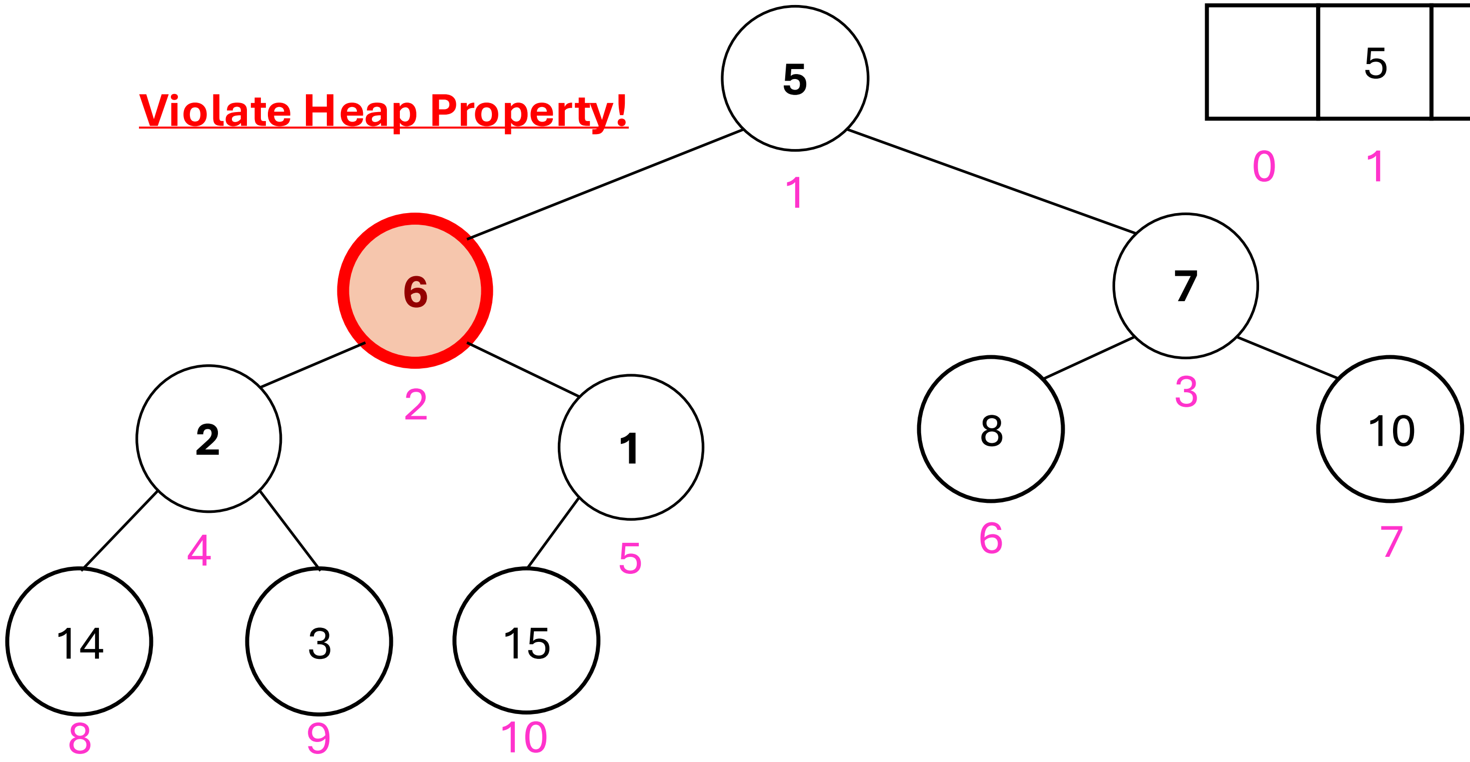


	5	6	10	2	1	8	7	14	3	15
0	1	2	3	4	5	6	7	8	9	10

```
buildHeap(){
   for(int i = size; i>0; i--){
      percolateDown(i);
   }
}
```

Floyd's buildHeap method (Percolate 6)

- Suppose we had n items and wanted to "heapify" them

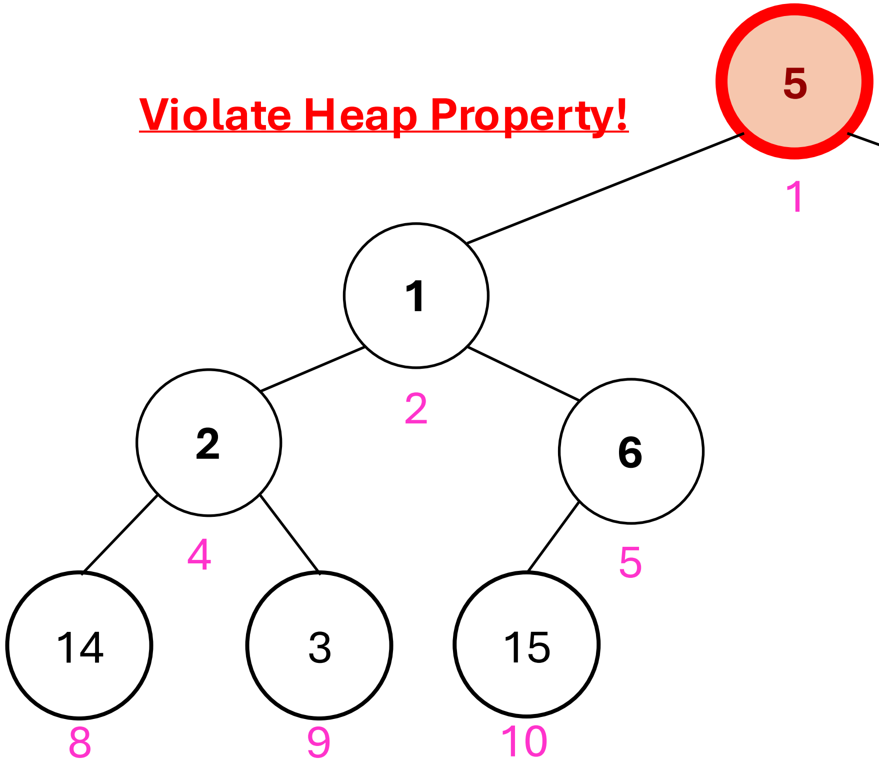


0	1	2	3	4	5	6	7	8	9	10
	5	6	7	2	1	8	10	14	3	15

```
buildHeap(){
   for(int i = size; i>0; i--){
      percolateDown(i);
   }
}
```

Floyd’s buildHeap method (Percolate 5)

- Suppose we had n items and wanted to “heapify” them



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

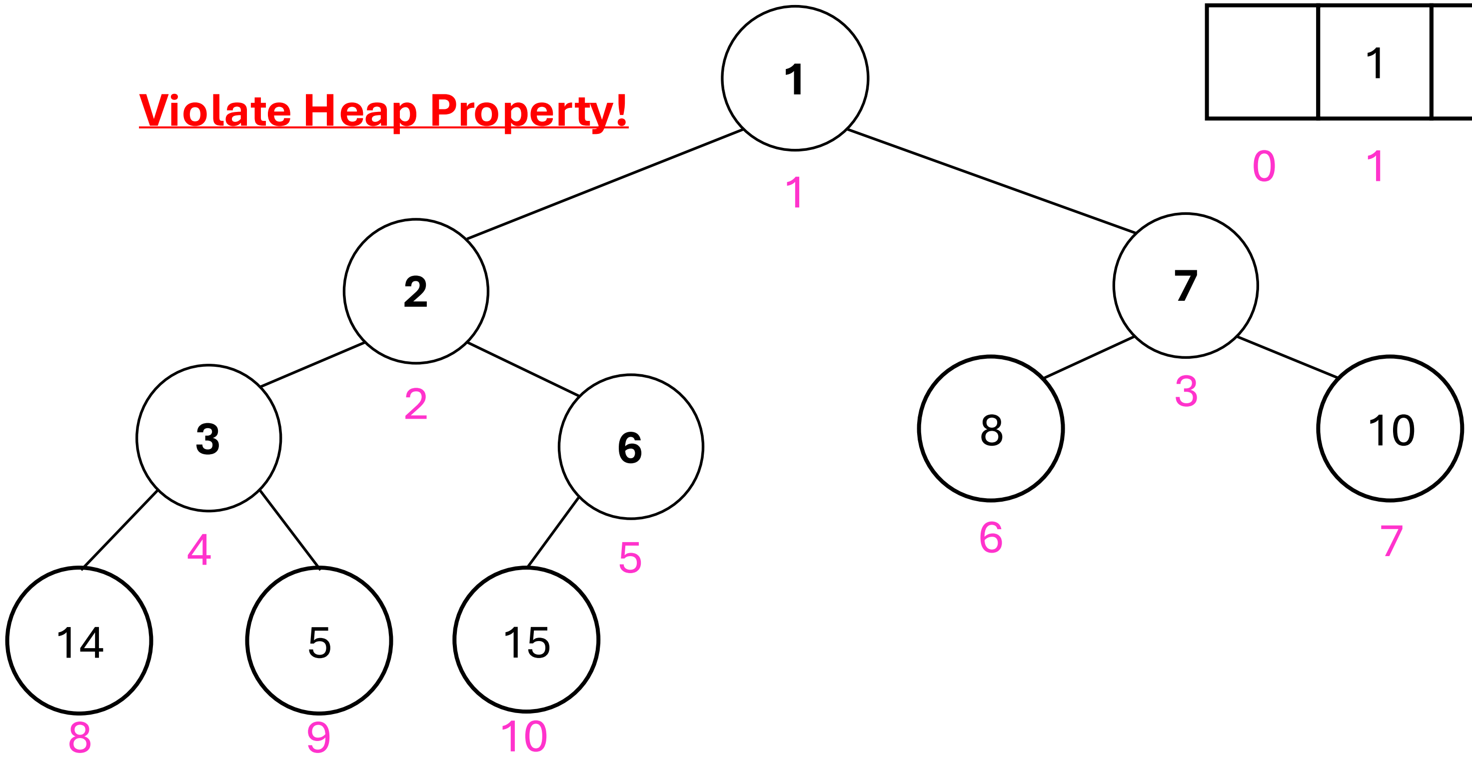
```
buildHeap(){
   for(int i = size; i>0; i--){
      percolateDown(i);
   }
}
```

Floyd's buildHeap method (Done)

- Suppose we had n items and wanted to "heapify" them

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

Violate Heap Property!




```
buildHeap(){
   for(int i = size; i>0; i--){
      percolateDown(i);
   }
}
```

How long did this take?

- Worst case running time of buildHeap:
- No node can percolate down more than the height of its subtree
  - When i is a leaf:
  - When i is second-from-last level:
  - When i is third-from-last level:
- Overall Running time:

```
buildHeap(){
   for(int i = size; i>0; i--){
      percolateDown(i);
   }
}
```

How long did this take?

- Worst case running time of buildHeap:
- No node can percolate down more than the height of its subtree
  - When i is a leaf ($n/2$ of these): 0 comparisons
  - When i is second-from-last level ($n/4$ of these): ≤ 2 comparisons
  - When i is third-from-last level ($n/8$ of these): ≤ 4 comparisons
  - …
- Overall Running time:

```
buildHeap(){
  for(int i = size; i>0; i--){
    percolateDown(i);
  }
}
```

How long did this take?

- Worst case running time of buildHeap: $n \sum_{i=1}^{\log_2(n)} \frac{i}{2^i}$
- No node can percolate down more than the height of its subtree
  - When i is a leaf ($n/2$ of these): 0 comparisons
  - When i is second-from-last level ($n/4$ of these): ≤ 2 comparisons
  - When i is third-from-last level ($n/8$ of these): ≤ 4 comparisons
  - …
- Overall Running time:

```
buildHeap(){
   for(int i = size; i>0; i--){
      percolateDown(i);
   }
}
```

A Serious Series Calculation

- **Lemma:** $E \coloneqq \; n \sum_{i=1}^{\log_2(n)} \frac{i}{2^i} \in \Theta(n)$
- **Proof:** First term is $n/2$**,** so $E \; \in \Omega(n)$ is immediate.

A Serious Series Calculation

- **Lemma:** $E \coloneqq \; n \sum_{i=1}^{\log_2(n)} \frac{i}{2^i} \in \Theta(n)$
- **Proof:** First term is $n/2$**,** so $E \; \in \Omega(n)$ is immediate.

A Serious Series Calculation

- **Lemma:** $E \coloneqq \; n \sum_{i=1}^{\log_2(n)} \frac{i}{2^i} \in \Theta(n)$
- **Proof:** First term is $n/2$**,** so $E \; \in \Omega(n)$ is immediate.
- Next, observe $\sum_{i=1}^{\log_2(n)} \frac{i}{2^i} \leq \sum_{i=1}^{\infty} \frac{i}{2^i}$

A Serious Series Calculation

- **Lemma:** $E \coloneqq\ n\sum_{i=1}^{\log_2(n)} \frac{i}{2^i} \in \Theta(n)$
- **Proof:** First term is $n/2$**,** so $E\ \in \Omega(n)$ is immediate.
- Next, observe $\sum_{i=1}^{\log_2(n)} \frac{i}{2^i} \leq \sum_{i=1}^{\infty} \frac{i}{2^i}$
- Recall for $|x| < 1$ we have $\sum_{i=0}^{\infty} x^i\ =\ \frac{1}{1-x}$

A Serious Series Calculation

- **Lemma:** $E \coloneqq \ n \sum_{i=1}^{\log_2(n)} \frac{i}{2^i} \in \Theta(n)$
- **Proof:** First term is $n/2$**,** so $E\ \in \Omega(n)$ is immediate.
- Next, observe $\sum_{i=1}^{\log_2(n)} \frac{i}{2^i} \leq \sum_{i=1}^{\infty} \frac{i}{2^i}$
- Recall for $|x| < 1$ we have $\sum_{i=0}^{\infty} x^i = \ \frac{1}{1-x}$
- Differentiate: $\sum_{i=1}^{\infty} i x^{i-1} = \ \frac{1}{(1-x)^2}$

A Serious Series Calculation

- **Lemma:** $E \coloneqq \; n \sum_{i=1}^{\log_2(n)} \frac{i}{2^i} \in \Theta(n)$
- **Proof:** First term is $n/2$**,** so $E \, \in \Omega(n)$ is immediate.
- Next, observe $\sum_{i=1}^{\log_2(n)} \frac{i}{2^i} \leq \sum_{i=1}^{\infty} \frac{i}{2^i}$
- Recall for $|x| < 1$ we have $\sum_{i=0}^{\infty} x^i = \frac{1}{1-x}$
- Differentiate: $\sum_{i=1}^{\infty} i x^{i-1} = \frac{1}{(1-x)^2}$
- Multiply both sides by x: $\sum_{i=1}^{\infty} i x^i = \frac{x}{(1-x)^2}$

A Serious Series Calculation

- **Lemma:** $E \coloneqq \; n \sum_{i=1}^{\log_2(n)} \frac{i}{2^i} \in \Theta(n)$
- **Proof:** First term is $n/2$**,** so $E \in \Omega(n)$ is immediate.
- Next, observe $\sum_{i=1}^{\log_2(n)} \frac{i}{2^i} \leq \sum_{i=1}^{\infty} \frac{i}{2^i}$
- Recall for $|x| < 1$ we have $\sum_{i=0}^{\infty} x^i = \frac{1}{1-x}$
- Differentiate: $\sum_{i=1}^{\infty} i x^{i-1} = \frac{1}{(1-x)^2}$
- Multiply both sides by x: $\sum_{i=1}^{\infty} i x^i = \frac{x}{(1-x)^2}$
- Plug in $x = 1/2$: $\sum_{i=1}^{\infty} i/2^i = \frac{1/2}{(1/2)^2} = 2.$

Next up: Dictionaries!

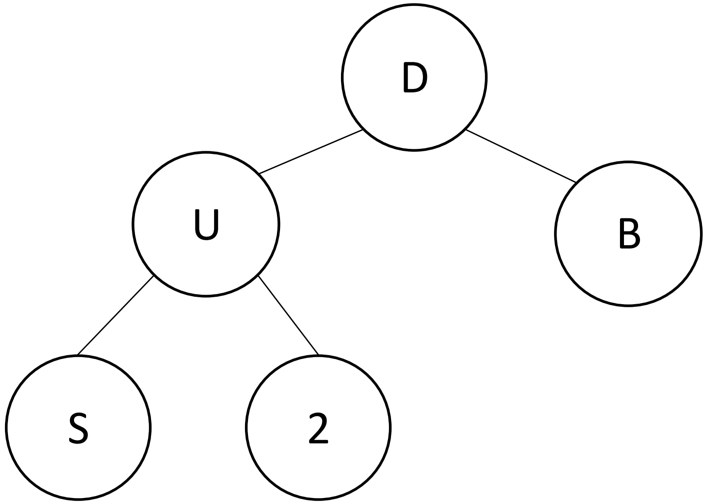
Dictionary (Map) ADT

- 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)

Naïve attempts

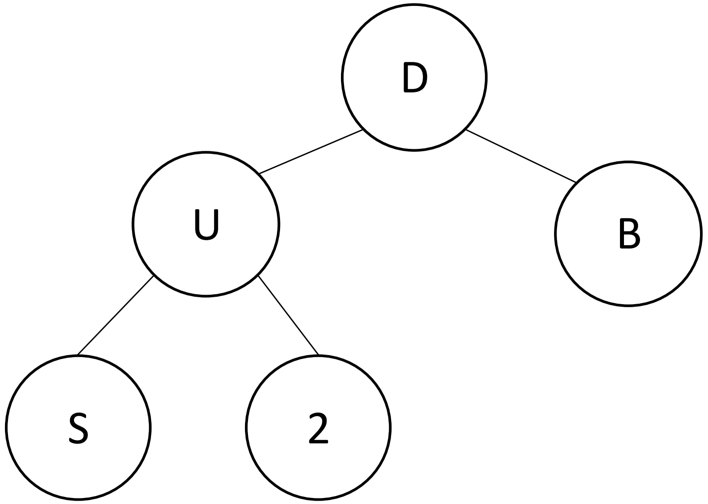
Data Structure	Time to insert	Time to find	Time to delete
Unsorted Array	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Unsorted Linked List	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Sorted Array	$\Theta(n)$	$\Theta(\log n)$	$\Theta(n)$
Sorted Linked List	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Heap	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Binary Search Tree (worst)	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Binary Search Tree (expected)	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$

Tree "Vocab"



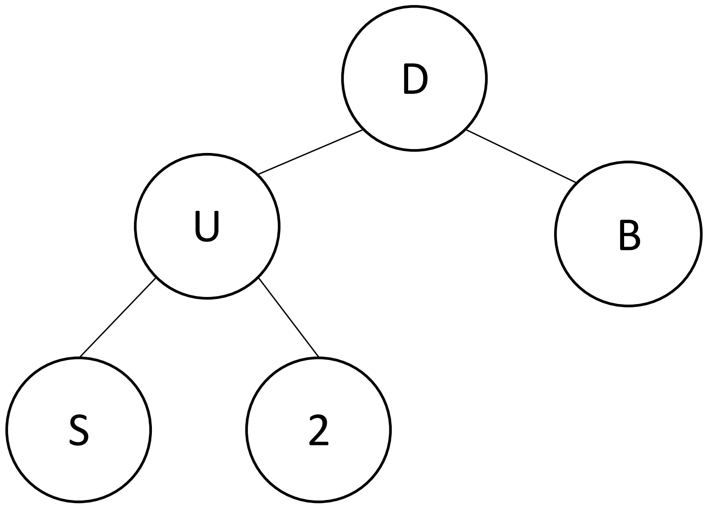
- Traversal:
  - An algorithm for "visiting/processing" every node in a tree
- Pre-Order Traversal:
  - Root, Left Subtree, Right Subtree
- In-Order Traversal:
  - Left Subtree, Root, Right Subtree
- Post-Order Traversal
  - Left Subtree, Right Subtree, Root

Tree “Vocab”



- Traversal:
  - An algorithm for “visiting/processing” every node in a tree
- Pre-Order Traversal:
  - Root, Left Subtree, Right Subtree
  - D U S 2 B
- In-Order Traversal:
  - Left Subtree, Root, Right Subtree
  - S U 2 D B
- Post-Order Traversal
  - Left Subtree, Right Subtree, Root
  - S 2 U B D

Pre/In/Post Order Trick



Name that Traversal!

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

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

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

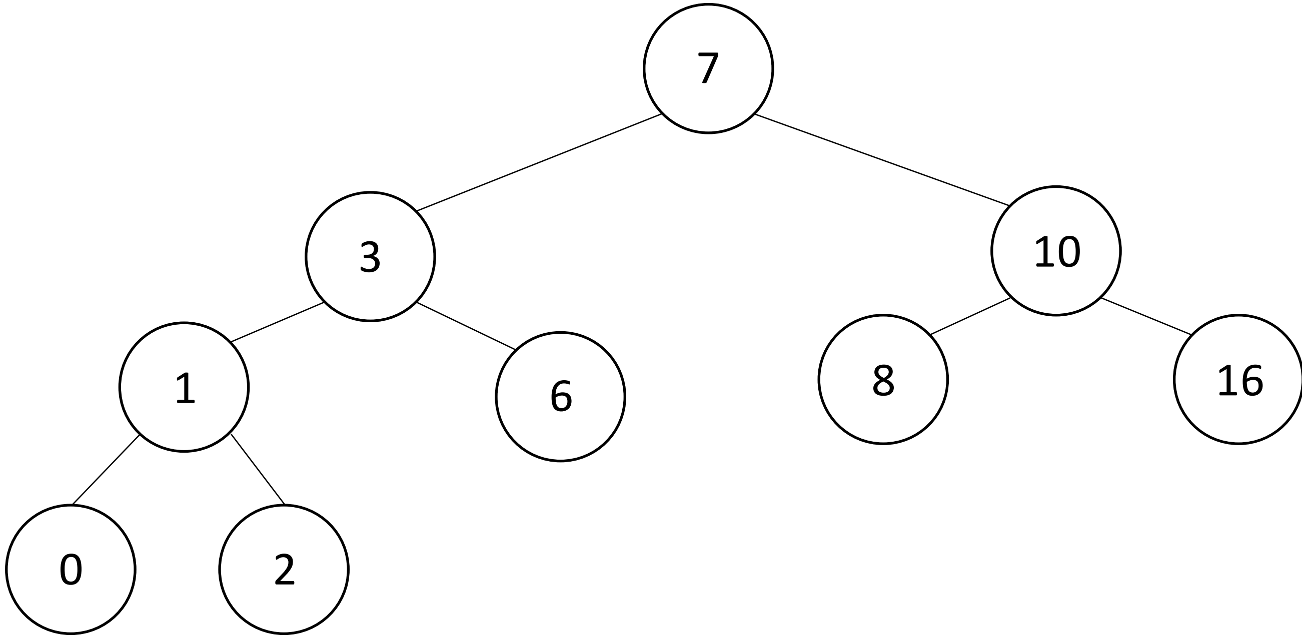
Name that Traversal! (Answers)

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

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

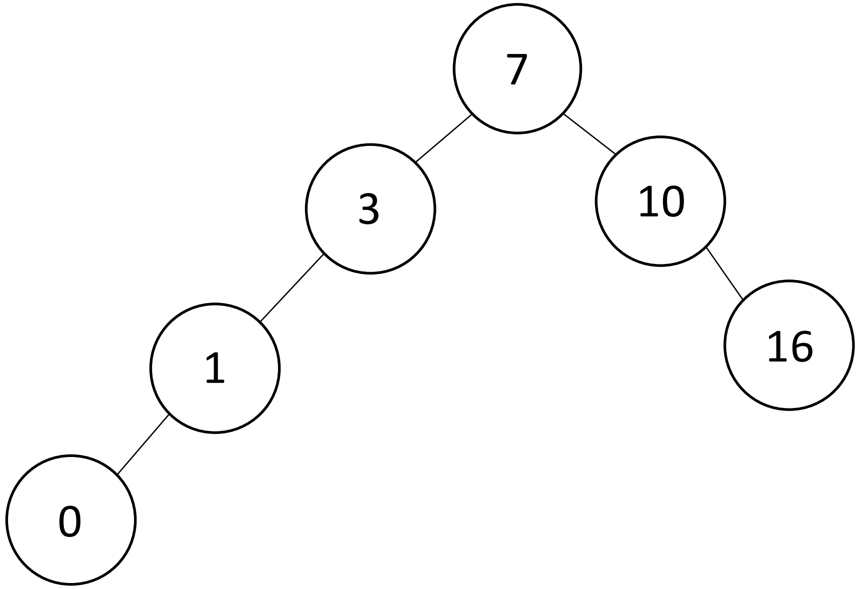
```
inOrder(root){
	if (root.left != Null){
		inOrder(root.left);
	}
	process(root)
	if (root.right != Null){
		inOrder(root.right);
	}
}
```

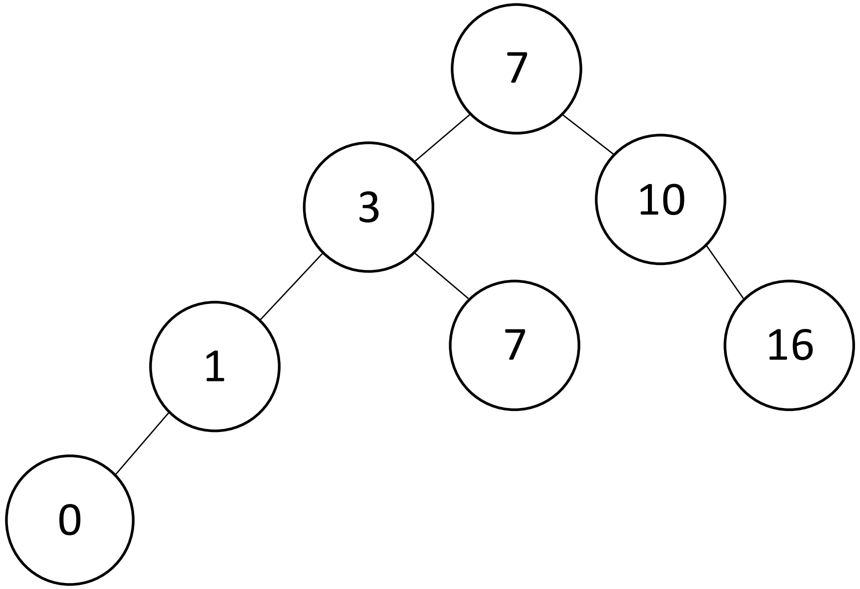
Binary Search Tree

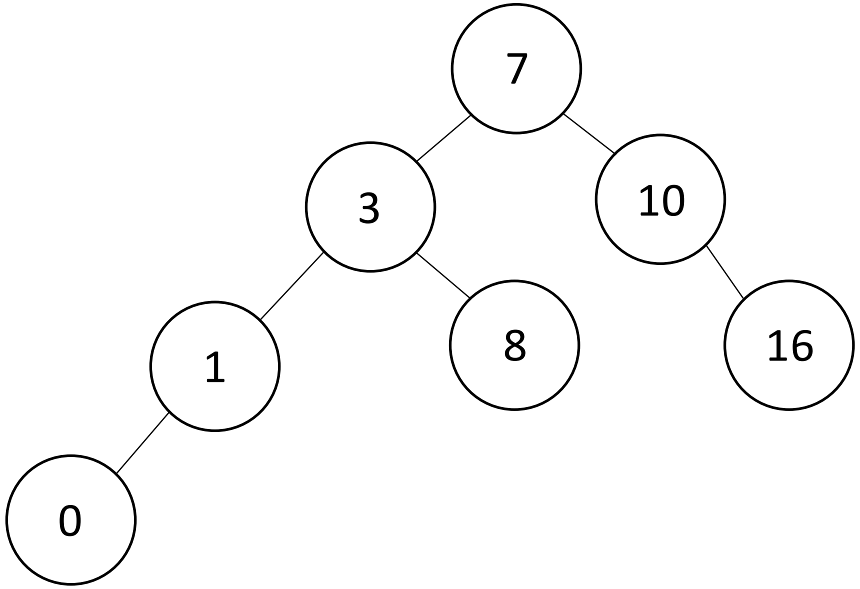


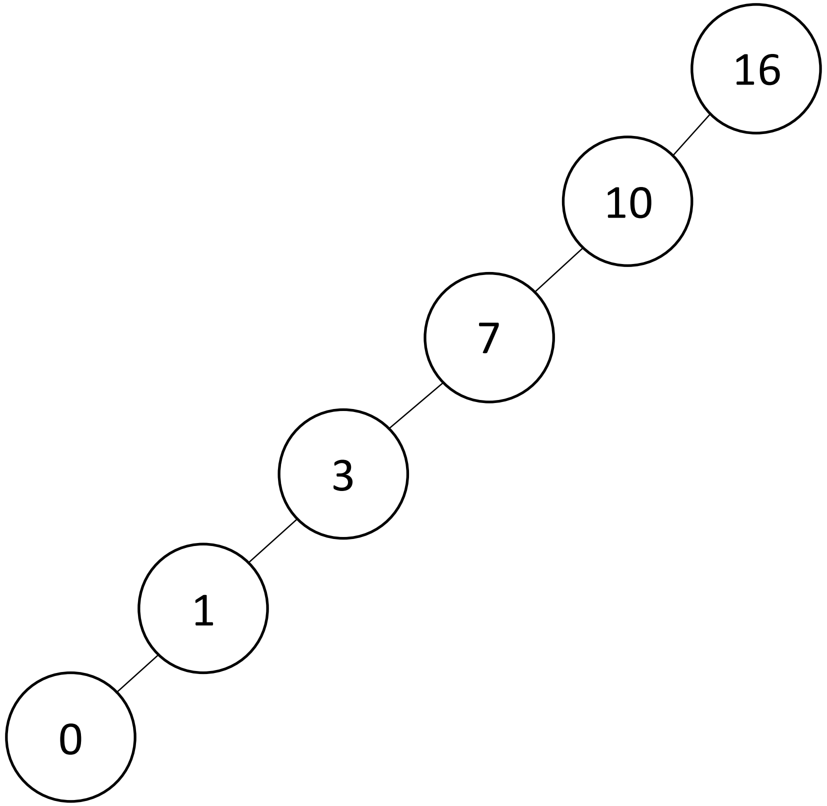
- Binary Tree
  - Definition:
    - Tree where each node has at most 2 children
- **Order Property**
  - All keys in the left subtree are smaller than the parent
  - All keys in the right subtree are larger than the parent
  - **Consequence**: cannot have repeated values

Are these BSTs?

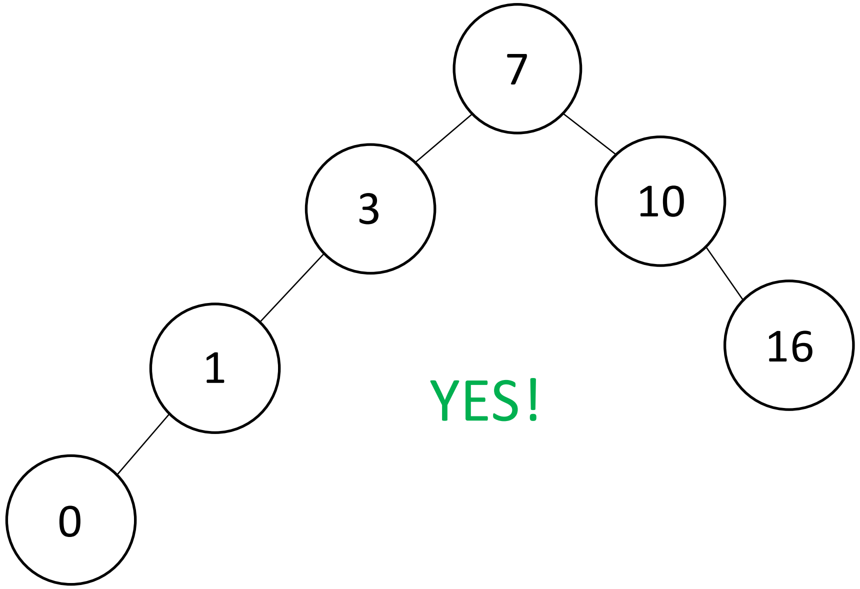




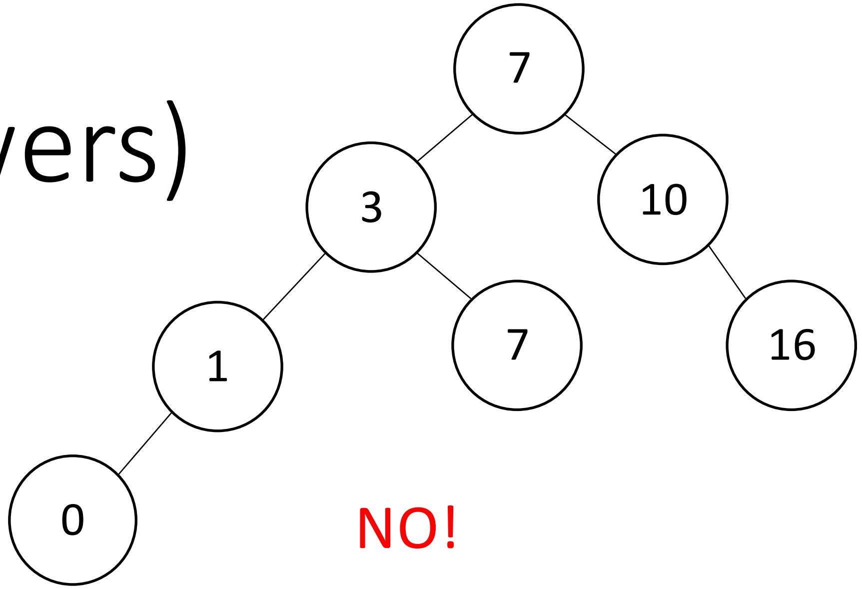




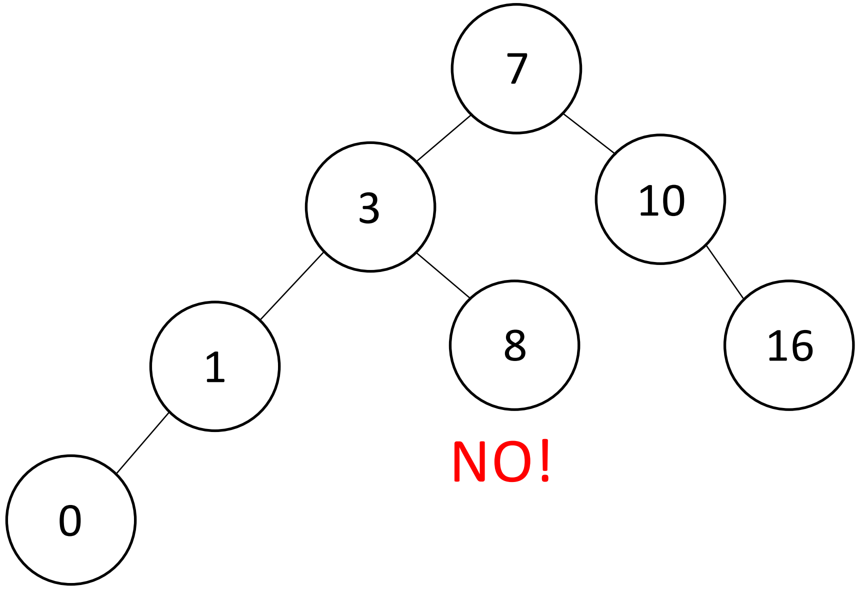
Are these BSTs? (Answers)



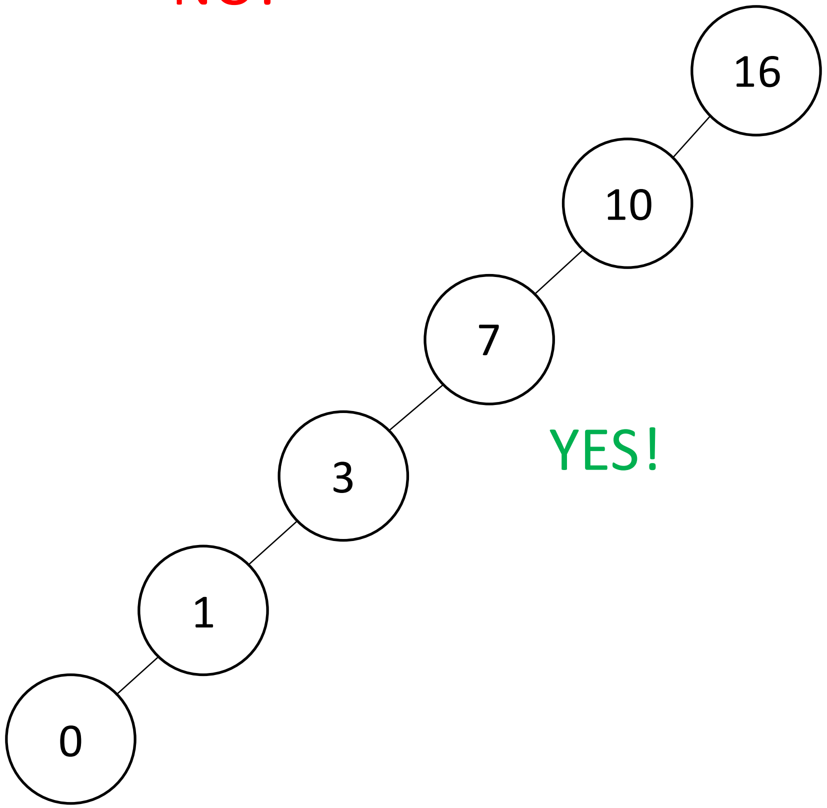
YES!



NO!



NO!

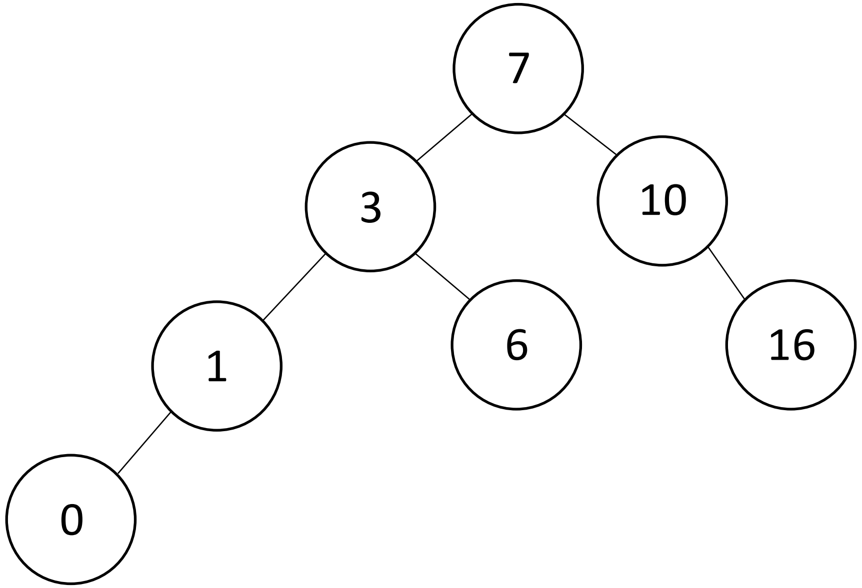


YES!

Aside: Why not use an array?

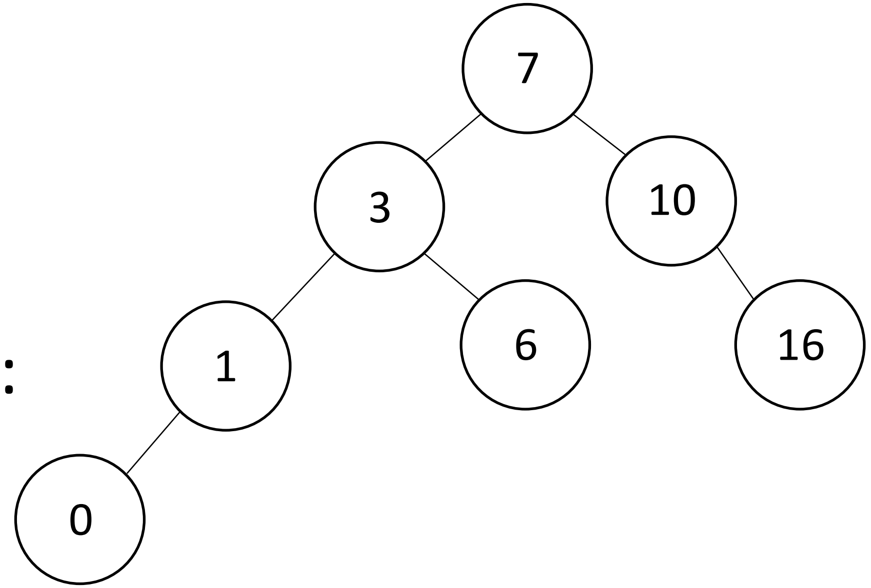
- We represented a heap using an array, finding children/parents by index
- We will represent BSTs with nodes and references. Why?
  - We might have “gaps” in our tree
  - Memory!
    - 2^n

Find Operation (recursive)



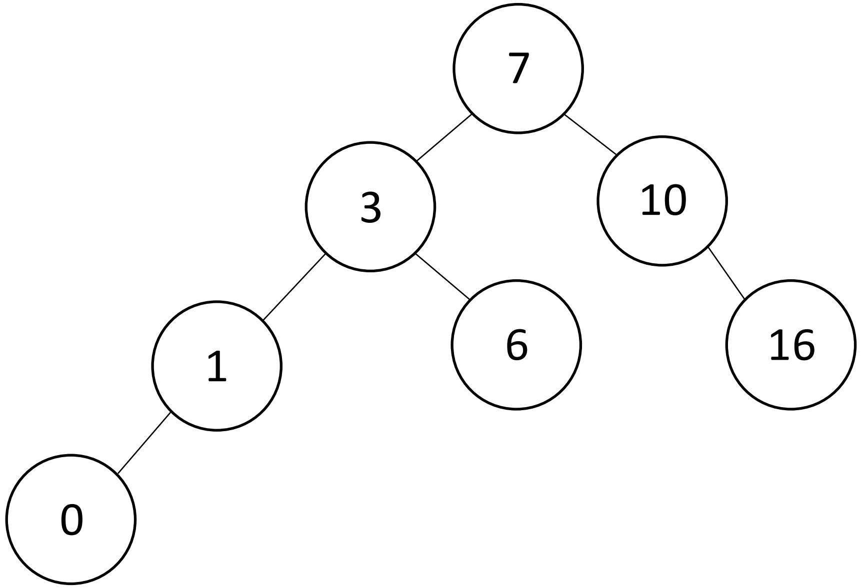

```
find(key, root):
        IF (root == Null):
                RETURN Null
        IF (key == root.key):
                RETURN root.value
        ELSE IF (key < root.key):
                RETURN find(key, root.left)
        ELSE IF (key > root.key):
                RETURN find(key, root.right);
        RETURN Null
```

Find Operation (iterative)




```
find(key, root):
    WHILE (root ≠ Null AND 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 (recursive)



insert(key, value, root):

    root = insertHelper(key, value, root)

insertHelper(key, value, root){

    IF(root == null):

        RETURN new Node(key, value)

    IF (root.key < key):

        root.right = **insertHelper**(key, value, root.right)

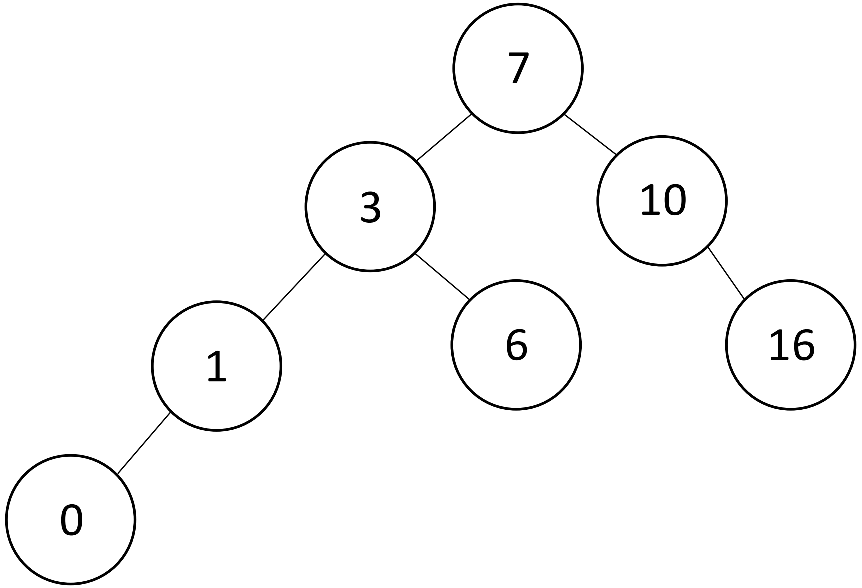
    ELSE:

        root.left = **insertHelper**(key, value, root.left)

    RETURN root

Note: Insert happens only at the leaves!

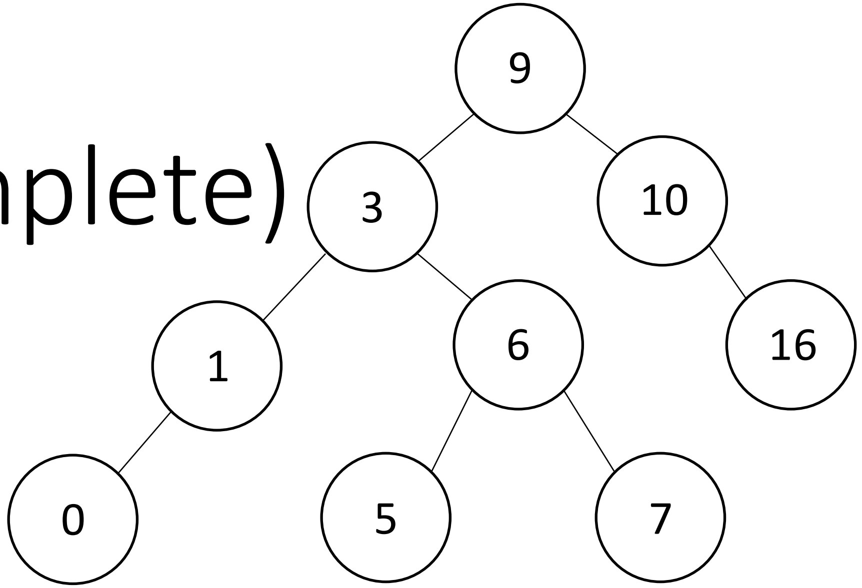
Insert Operation (iterative)




```
insert(key, value, root):
    IF (root == Null):
        root = new Node(key, value)
    parent = Null
    WHILE (root ≠ Null AND key ≠ root.key):
        parent = root
        IF (key < root.key):
            root = root.left
        ELSE IF (key > root.key):
            root = root.right
    IF (key == root.key) THEN root.value = value
    ELSE IF (key < parent.key) THEN parent.left = new Node(key, value)
    ELSE: parent.right = new Node (key, value)
```

Note: Insert happens only at the leaves!

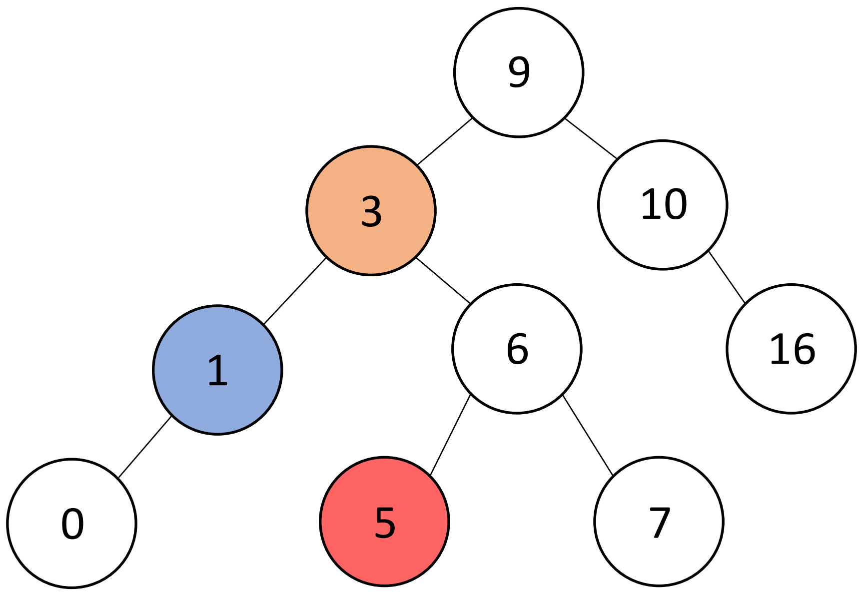
Delete Operation (iterative, incomplete)




```
delete(key, root):
    WHILE (root ≠ Null AND key ≠ root.key):
        IF (key < root.key) THEN root = root.left
        ELSE IF (key > root.key)THEN root = root.right
    IF (root == Null) THEN RETURN
    // Now we have found the node to delete, what happens next?
}
```

Delete – 3 Cases

- 0 Children (i.e. it’s a leaf)
- 1 Child
  - Replace the deleted node with its child
- 2 Children
  - Replace the deleted with the largest node to its left or alternatively the smallest node to its right

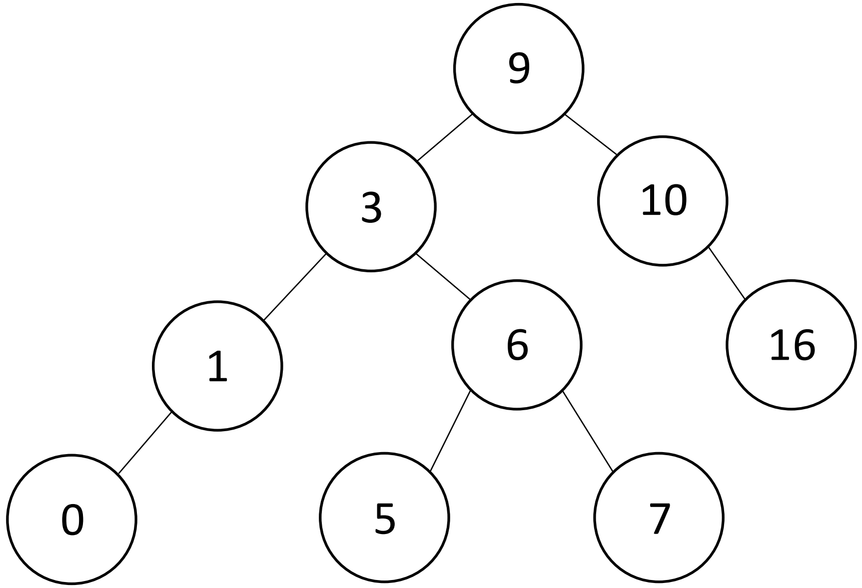


Finding the Max and Min

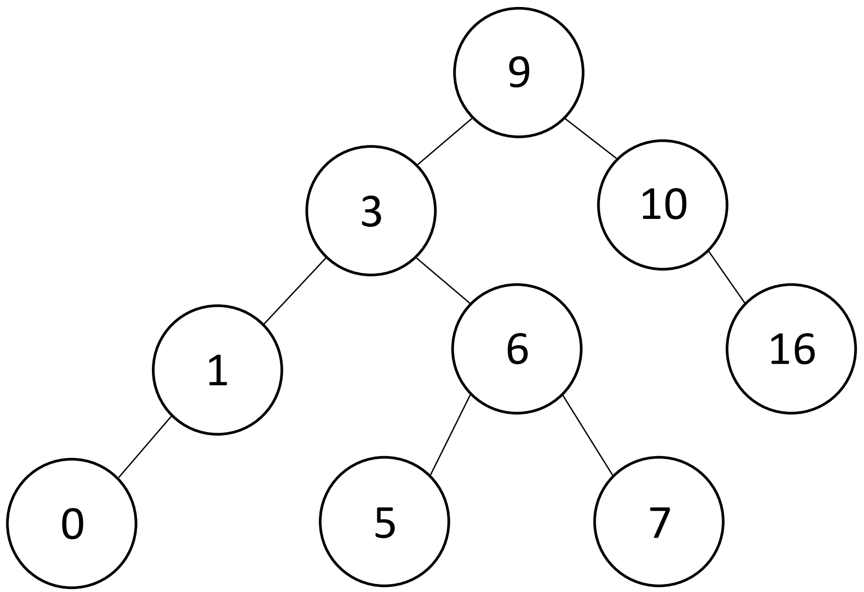
- Max of a BST:
  - Right-most Thing
- Min of a BST:
  - Left-most Thing

```
maxNode(root):
        IF (root == Null):
                RETURN Null
        WHILE (root.right ≠ Null):
                root = root.right
        RETURN root
```

```
minNode(root){
        IF (root == Null)
                return Null
        WHILE (root.left ≠ Null):
                root = root.left
        RETURN root
```

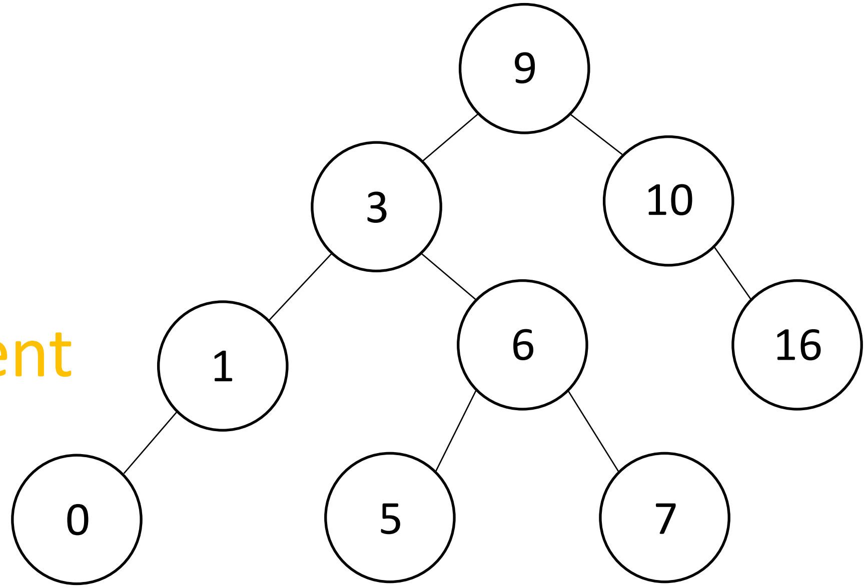


Delete Operation (iterative)




```
delete(key, root):
    WHILE (root ≠ Null AND key ≠ root.key):
        IF (key < root.key):
            root = root.left
        ELSE IF (key > root.key):
            root = root.right
    IF (root == Null) THEN 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
```

Delete Operation (recursive)




```
delete(key, root){
    IF (root == Null) THEN  RETURN // key not present
    IF (root.key == key):
        IF (root has no children):
            RETURN Null
        IF (root has one child):
            RETURN that child
        IF (root has two children)
            RETURN removeMax(root.left)
    IF (root.key < key): THEN root.right = delete(key, root.right)
    ELSE: root.left = delete(key, root.left)
```

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?
  - $\Theta(n)$

Improving the worst case

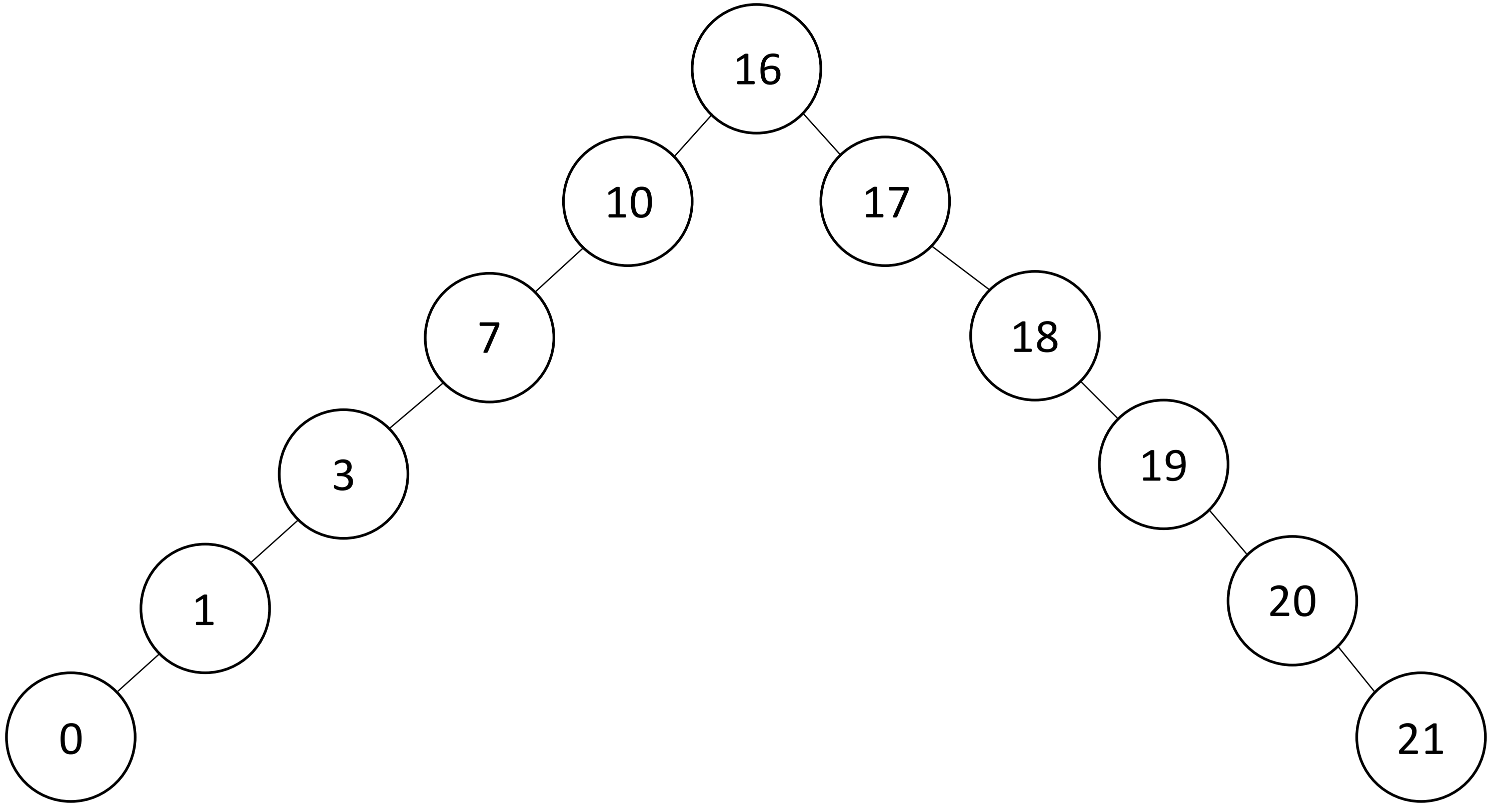
- How can we get a better worst case running time?
  - Add rules about the shape of our BST
- AVL Tree
  - A BST with some shape rules
    - Algorithms need to change to accommodate those

“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”
  - Encourage Branches!

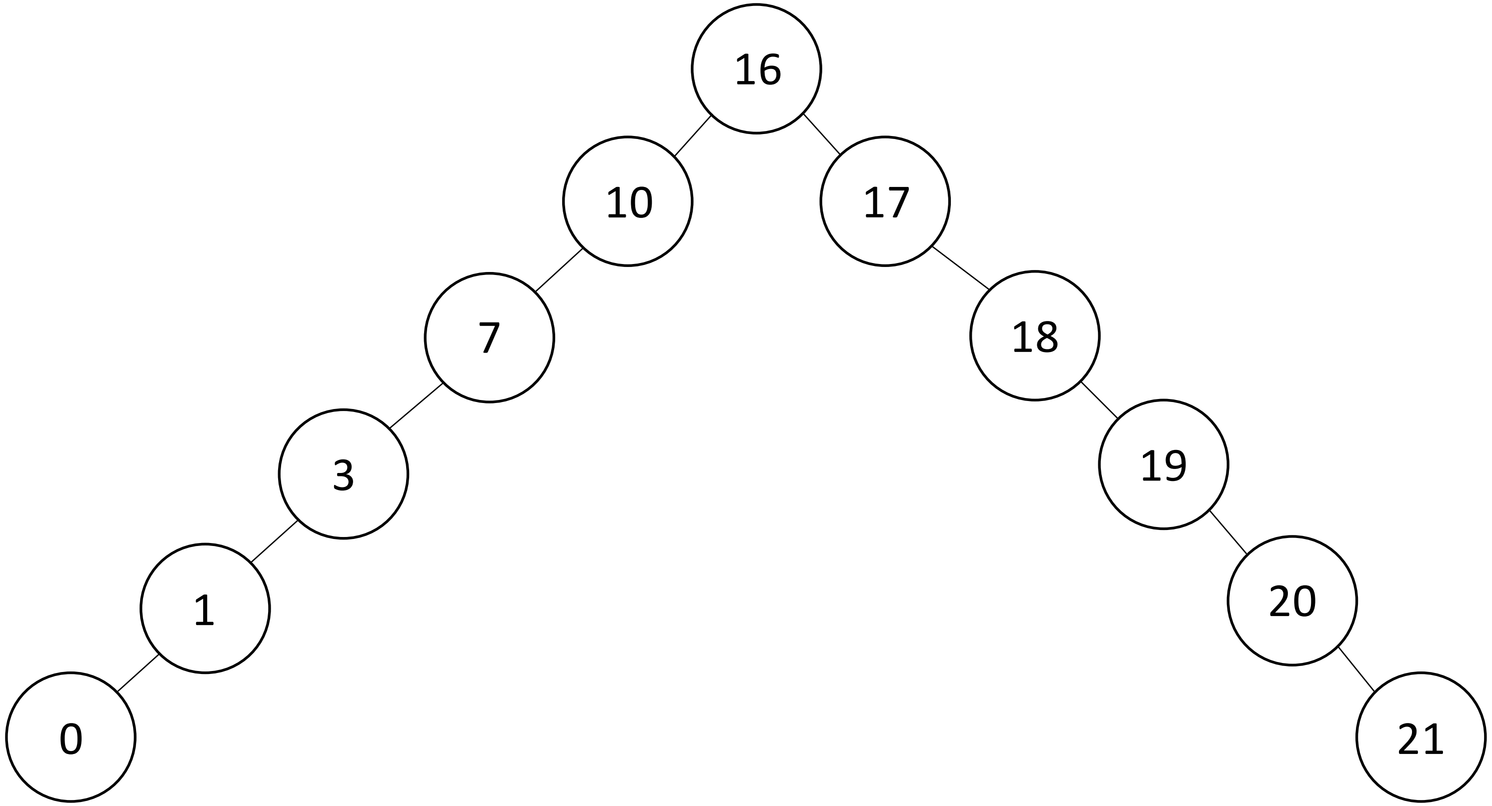
Idea 1: Both Subtrees of Root have same # Nodes

Idea 1: Both Subtrees of Root have same # Nodes - Issue



Idea 2: Both Subtrees of Root have same height

Idea 2: Both Subtrees of Root have same height - Issue



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

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

Not all tree sizes are possible!

For example, cannot have a tree of size 2.

Idea 4: Both Subtrees of every Node have same height

Idea 4: Both Subtrees of every Node have same height - Issue

Not all tree sizes are possible!

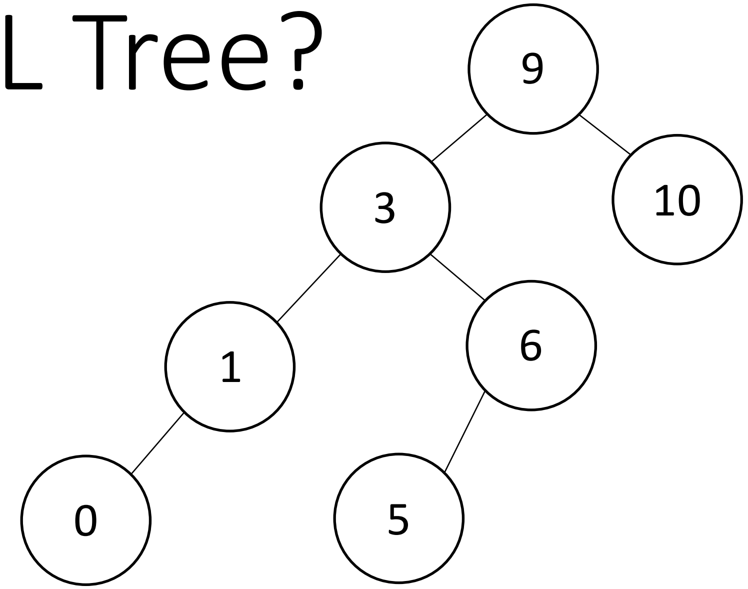
For example, cannot have a tree of size 2.

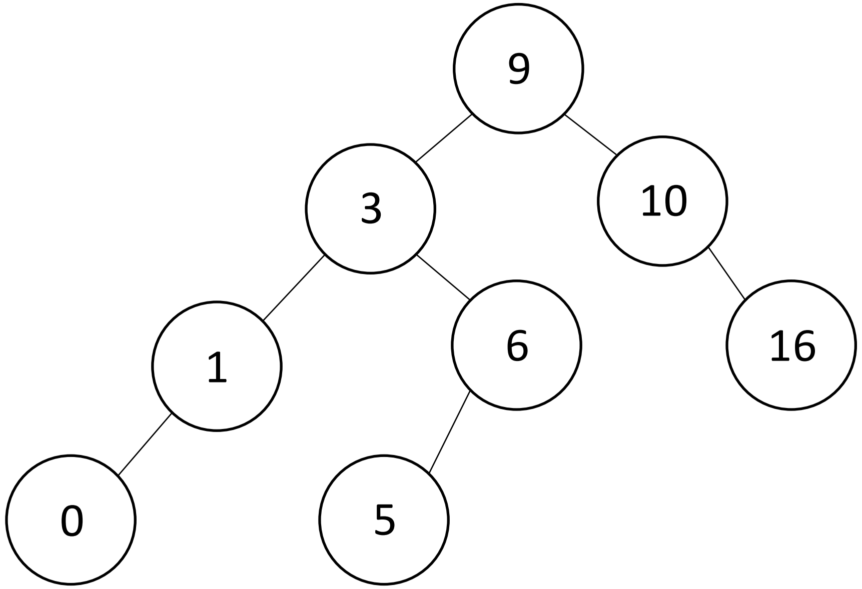
AVL Tree

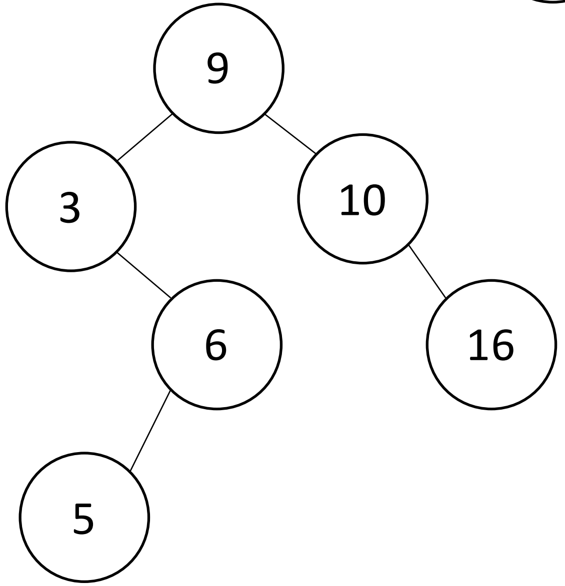
- A Binary Search tree that maintains that the left and right subtrees of every node have heights that differ by at most one.
  - height of left subtree and height of right subtree off by at most 1
  - Not too weak (ensures trees are short)
  - Not too strong (works for any number of nodes)

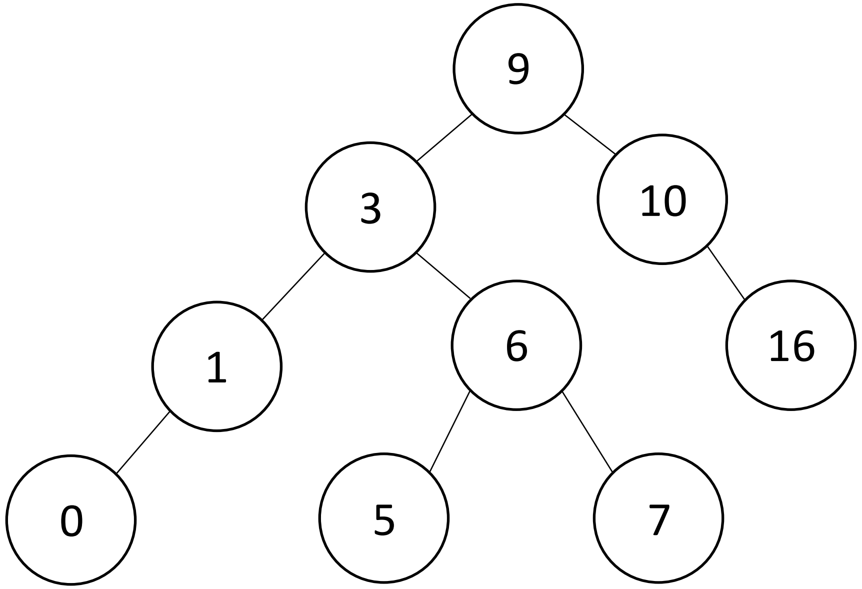
- Idea of AVL Tree:
  - When you insert/delete nodes, if tree is “out of balance” then modify the tree
  - Modification = “rotation”

Is it an AVL Tree?



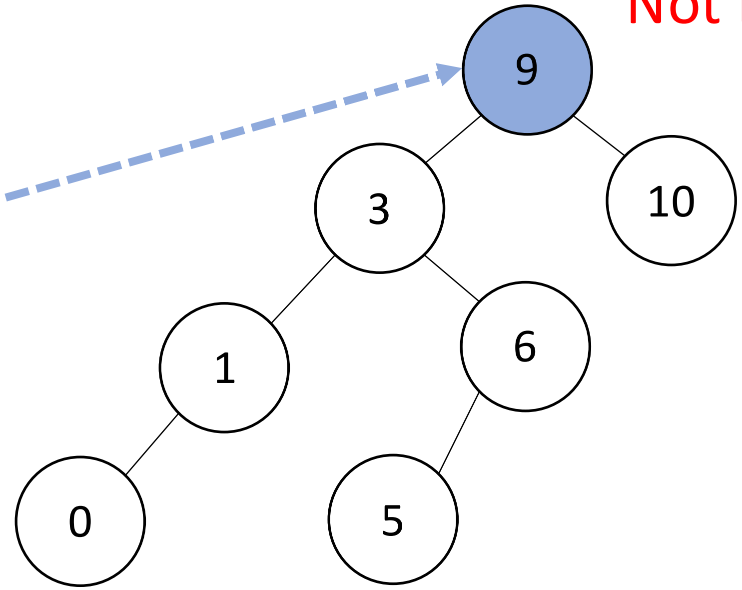




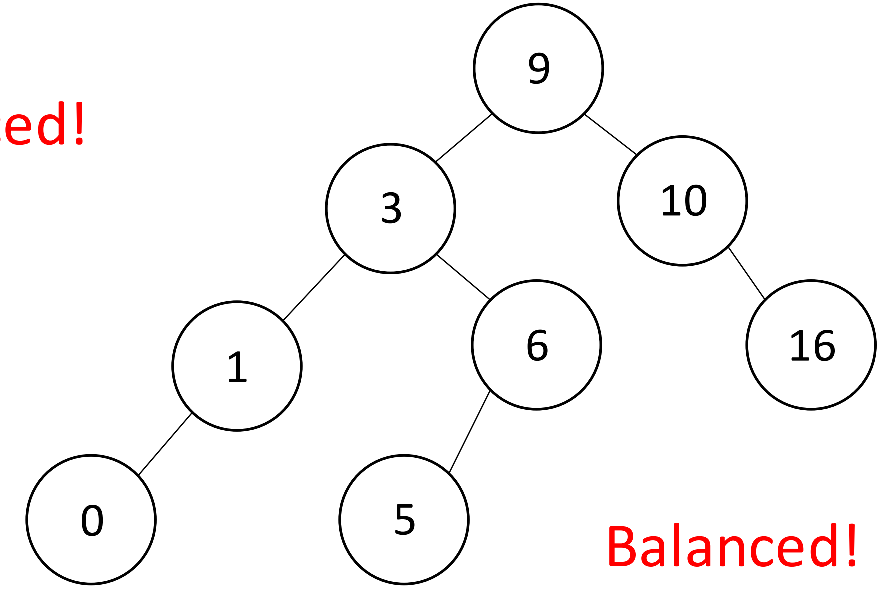


Is it an AVL Tree? (Answers)

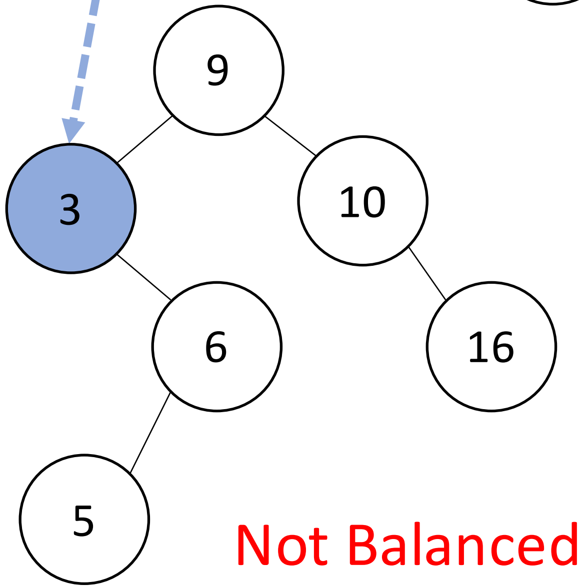
“Problem” Node
Its children’s heights differ by more than 1



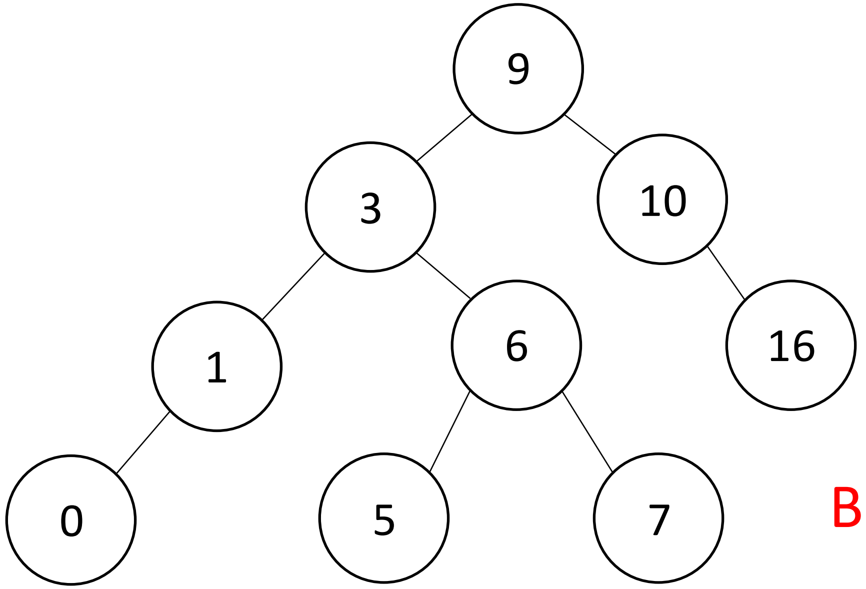
Not Balanced!



Balanced!



Not Balanced!



Balanced!