CSE 332 Summer 2024
Lecture 6: Priority Queues & Dictionaries

Nathan Brunelle

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

Warm Up

- Describe an algorithm for finding the maximum value in a min heap.
  - The last $\frac{n}{2}$ nodes in the heap are leaves, the max must be a leaf, we just check the last $\frac{n}{2}$ items
  - What is its running time?

ADT: Priority Queue

- What is it?
  - A collection of items and their “priorities”
  - Allows quick access/removal to the “top priority” thing
    - Usually a smaller priority value means the item is “more important”
- What Operations do we need?
  - insert(item, priority)
    - Add a new item to the PQ with indicated priority
  - extract
    - Remove and return the “top priority” item from the queue
      - Usually the item with the smallest priority value
  - IsEmpty
    - Indicate whether or not there are items still on the queue
- Note: the “priority” value can be any type/class so long as it’s comparable (i.e. you can use “<“ or “compareTo” with it)

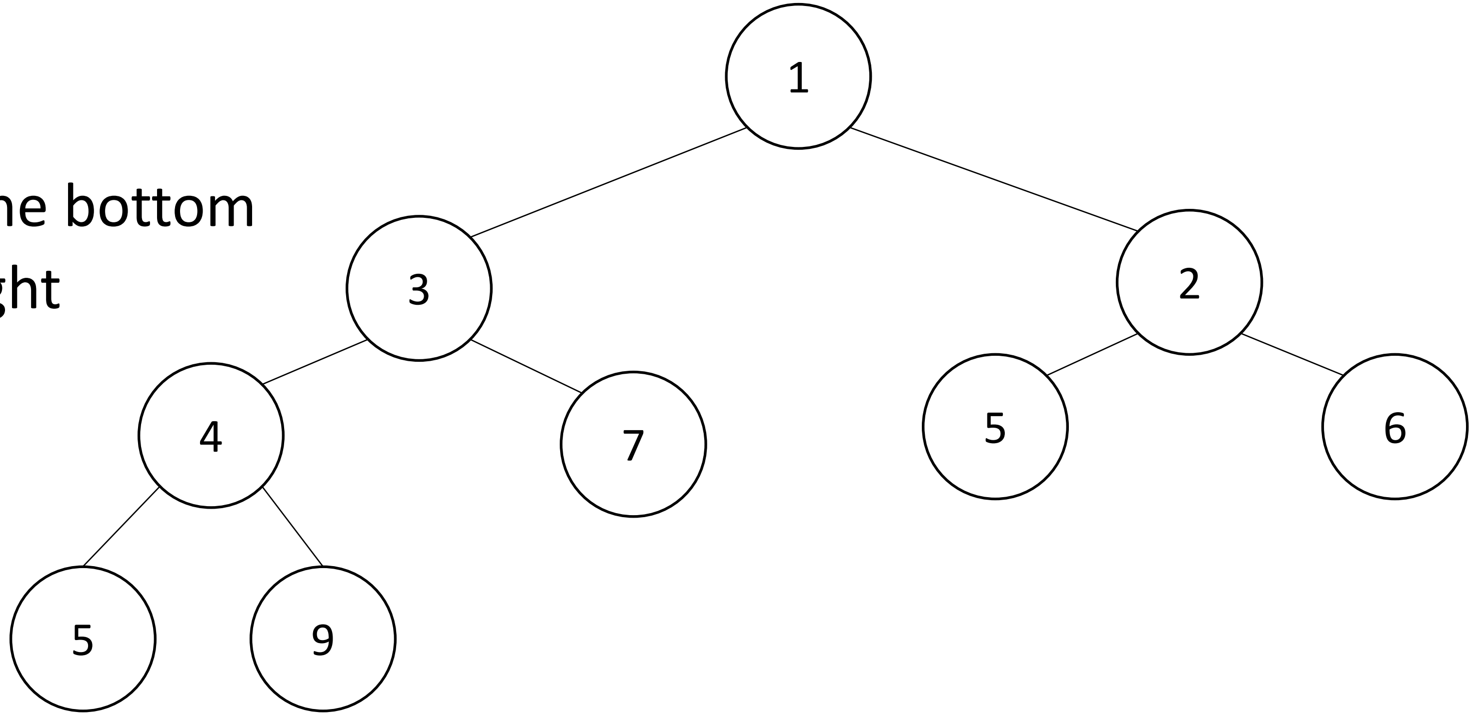
Thinking through implementations

Data Structure	Worst case time to insert	Worst case time to extract
Unsorted Array	$\Theta(1)$	$\Theta(n)$
Unsorted Linked List	$\Theta(1)$	$\Theta(n)$
Sorted Array	$\Theta(n)$	$\Theta(1)$
Sorted Linked List	$\Theta(n)$	$\Theta(1)$
Binary Search Tree	$\Theta(n)$	$\Theta(n)$
Binary Heap	$\Theta(\log n)$	$\Theta(\log n)$

For simplicity, Assume we know the maximum size of the PQ in advance
(otherwise we'd do an amortized analysis, but get the same answers…)

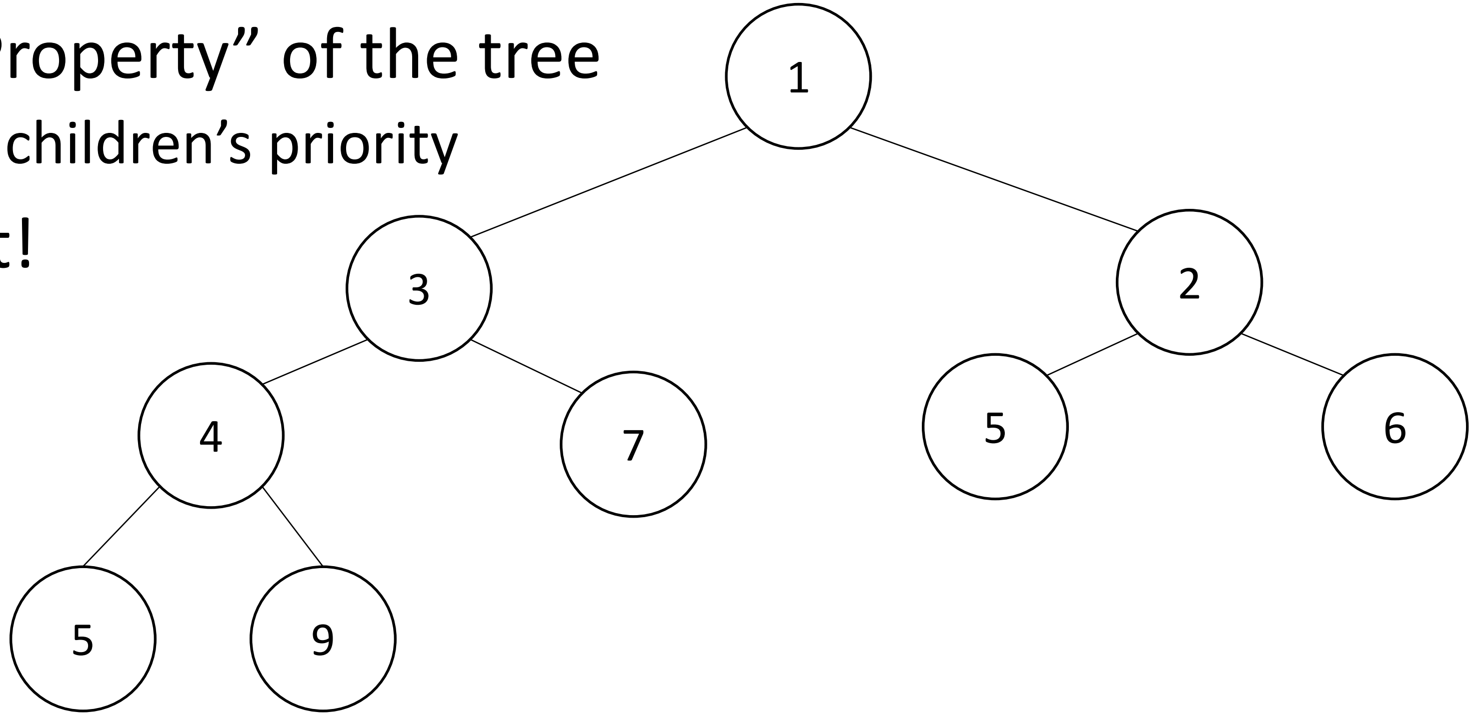
Trees for Heaps

- Binary Trees:
  - The branching factor is 2
  - Every node has $\leq$ 2 children
- Complete Tree:
  - All “layers” are full, except the bottom
  - Bottom layer filled left-to-right



(Min) Heap Data Structure

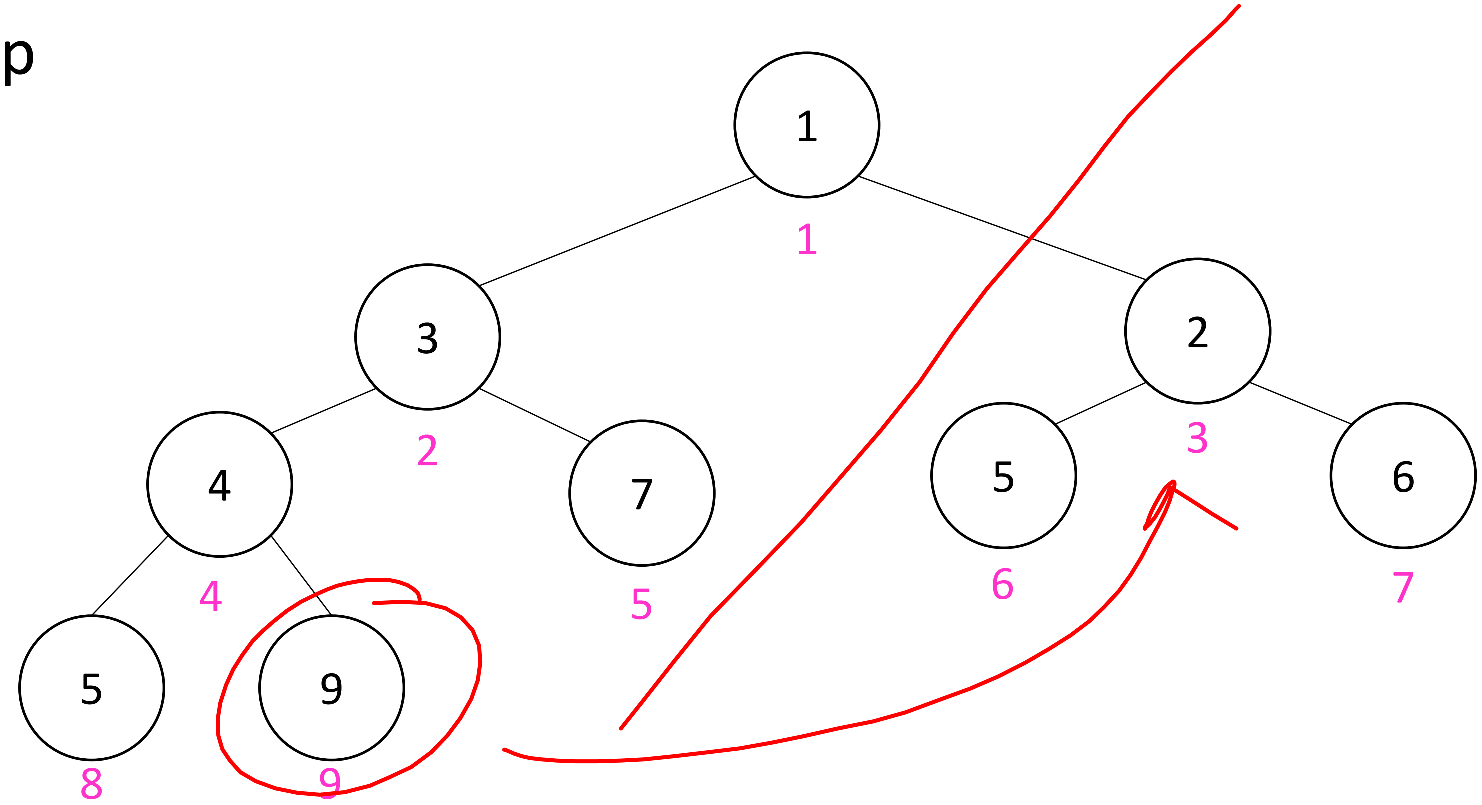
- Keep items in a complete binary tree
  - All “layers” are full, except the bottom
  - Bottom layer filled left-to-right
- Maintain the “(Min) Heap Property” of the tree
  - Every node’s priority is $\leq$ its children’s priority
- Minimum is always the root!



Representing a Heap

	1	3	2	4	7	5	6	5	9
0	1	2	3	4	5	6	7	8	9

- Every complete binary tree with the same number of nodes uses the same positions and edges
- Use an array to represent the heap
- Index of root: 1
- Parent of node i: $\left\lfloor \frac{i}{2} \right\rfloor$
- Left child of node i: $2i$
- Right child of node i: $2i + 1$
- Location of the leaves:
  - Last $\left\lceil \frac{n}{2} \right\rceil$ indices



Percolate Up and Down (for a Min Heap)

- Goal: restore the “Heap Property”
- Percolate Up:
  - Take a node that may be smaller than a parent, repeatedly swap with a parent until it is larger than its parent
- Percolate Down:
  - Take a node that may be larger than one of its children, repeatedly swap with smallest child until both children are larger
- Worst case running time of each:
  - $\Theta(\log n)$

Percolate Up

```
percolateUp(int i){
    int parent = i/2;  \\ index of parent
    Item val = arr[i];  \\ value at current location
    while(i > 1 && arr[i] < arr[parent]){  \\ until location is root or heap property holds
        arr[i] = arr[parent];  \\ move parent value to this location
        arr[parent] = val; \\ put current value into parent's location
        i = parent;  \\ make current location the parent
        parent = i/2;  \\ update new parent
    }
}
```

Percolate Down

```
percolateDown(int i){
    int left = i*2;  \\ index of left child
    int right = i*2+1;  \\ index of right child
    Item val = arr[i];  \\ value at location
    while(left <= size){  \\ until location is leaf
        int toSwap = right;
        if(right > size || arr[left] < arr[right]){  \\ if there is no right child or if left child is smaller
            toSwap = left;  \\ swap with left
        } \\ now toSwap has the smaller of left/right, or left if right does not exist
        if (arr[toSwap] < val){  \\ if the smaller child is less than the current value
            arr[i] = arr[toSwap];
            arr[toSwap] = val; \\ swap parent with smaller child
            i = toSwap; \\ update current node to be smaller child
            left = i*2;
            right = i*2+1;
        }
        else{ return;} \\ if we don't swap, then heap property holds
    }
}
```

Operations

- Insert
  - Make the new item last in the array, percolate up
- Extract
  - Move the last item to the root, percolate down
- Increase Key
  - Given the index of an item in the PQ, make its priority value larger
    - Min Heap: Then percolate Down
    - Max Heap: Then percolate Up
- Decrease Key
  - Given the index of an item in the PQ, make its priority value smaller
    - Min Heap: Then percolate Up
    - Max Heap: Then percolate Down
- Remove
  - Return the item at the given index from the PQ, remove that item from the PQ
    - Incorrect algorithm: consider the subtree rooted at the given index, perform an extract on that subtree

Operations

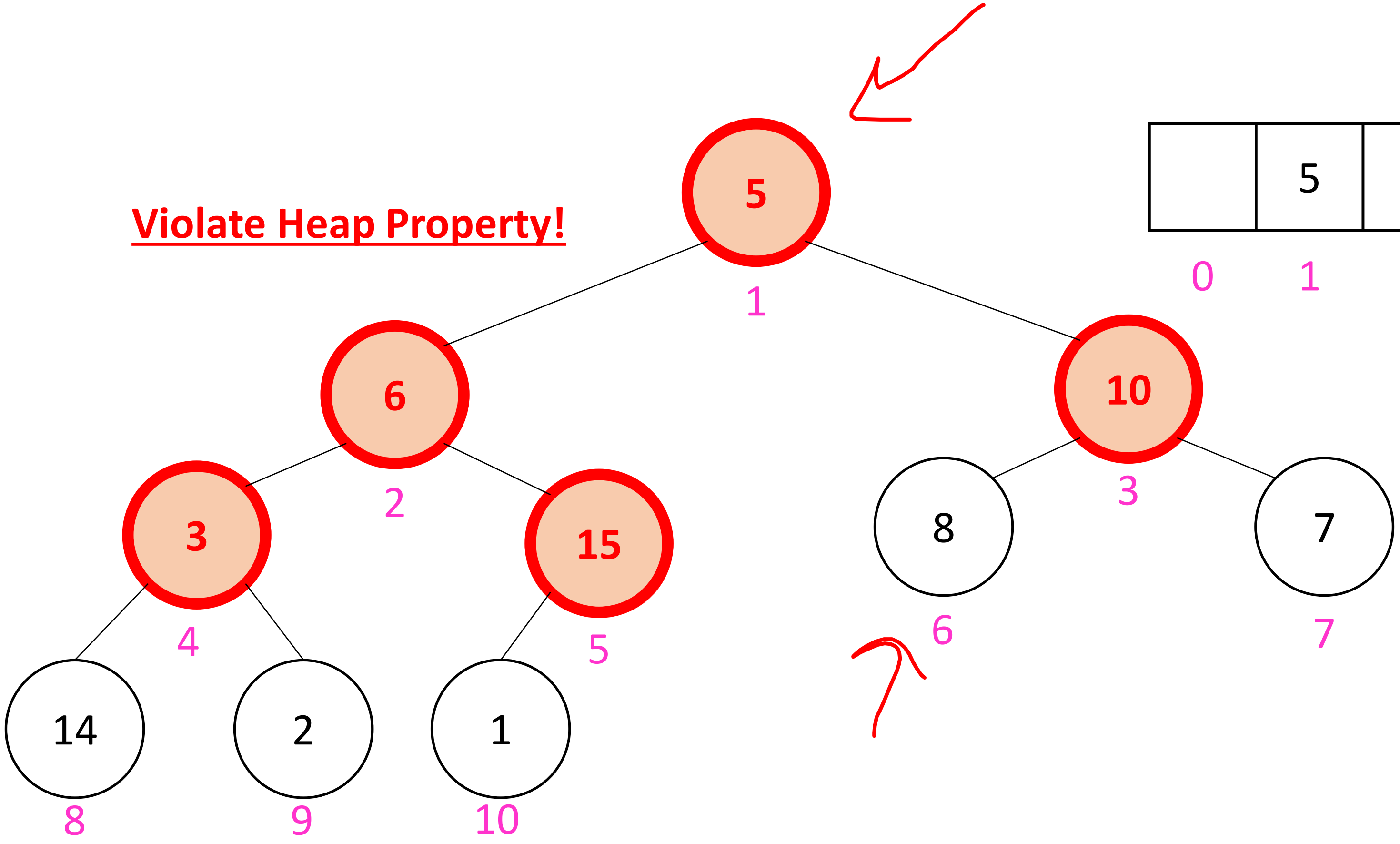
- Insert
  - Place new item last in the array, percolate it up
- Extract
  - Save root’s value, move last item to root, percolate it down
- Increase Key
  - Given the index of an item in the PQ, make its priority value larger
    - Min Heap: Then percolate Down
    - Max Heap: Then percolate Up
- Decrease Key
  - Given the index of an item in the PQ, make its priority value smaller
    - Min Heap: Then percolate Up
    - Max Heap: Then percolate Down
- Remove
  - Return the item at the given index from the PQ, remove that item from the PQ
    - Save value at the given index, move last item to that index, percolated it down

Aside: Expected Running time of Insert

- Assume I have a heap with n items in it. I insert a “random” item.
- Probability that the item is a leaf of the heap
  - Roughly half the nodes in a heap are leaves
  - There is a 50% chance of needing to do 1 comparison
- The new node is the parent of a leaf
  - We do 2 comparisons
  - $\frac{n}{4}$
- $\frac{1}{2} + 2 \cdot \frac{1}{4} + 3 \cdot \frac{1}{8} + 4 \cdot \frac{1}{16} + \cdots$
- $\Theta(1)$

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

- Working towards the root, one row at a time, percolate down

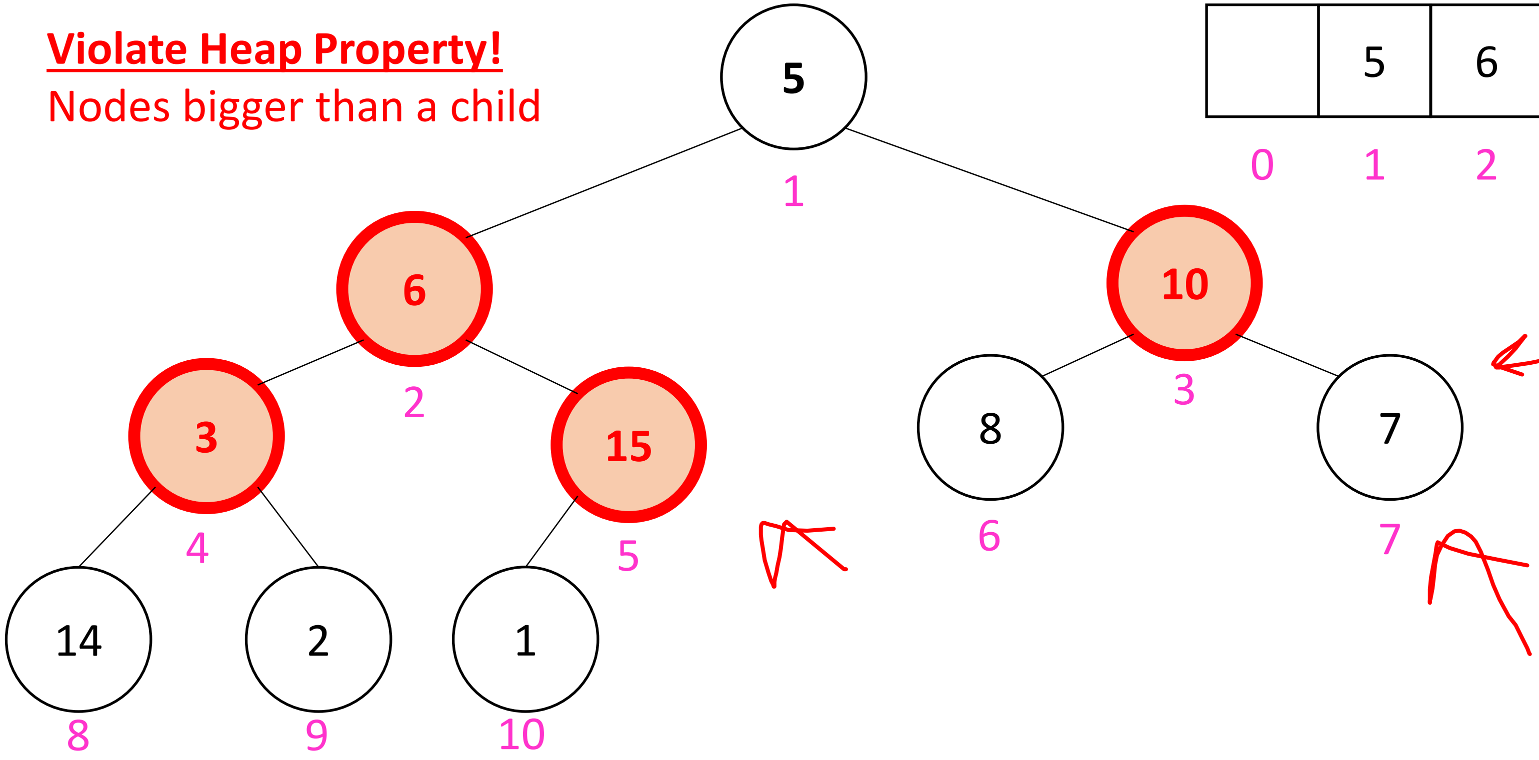
```
buildHeap(){
    for(int i = size; i>0; i--){
        percolateDown(i);
    }
}
```

Floyd's buildHeap method

$\frac{n}{2} \cdot 1 + \frac{n}{4} \cdot 2 + \frac{n}{8} \cdot 3$

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

Violate Heap Property!
Nodes bigger than a child



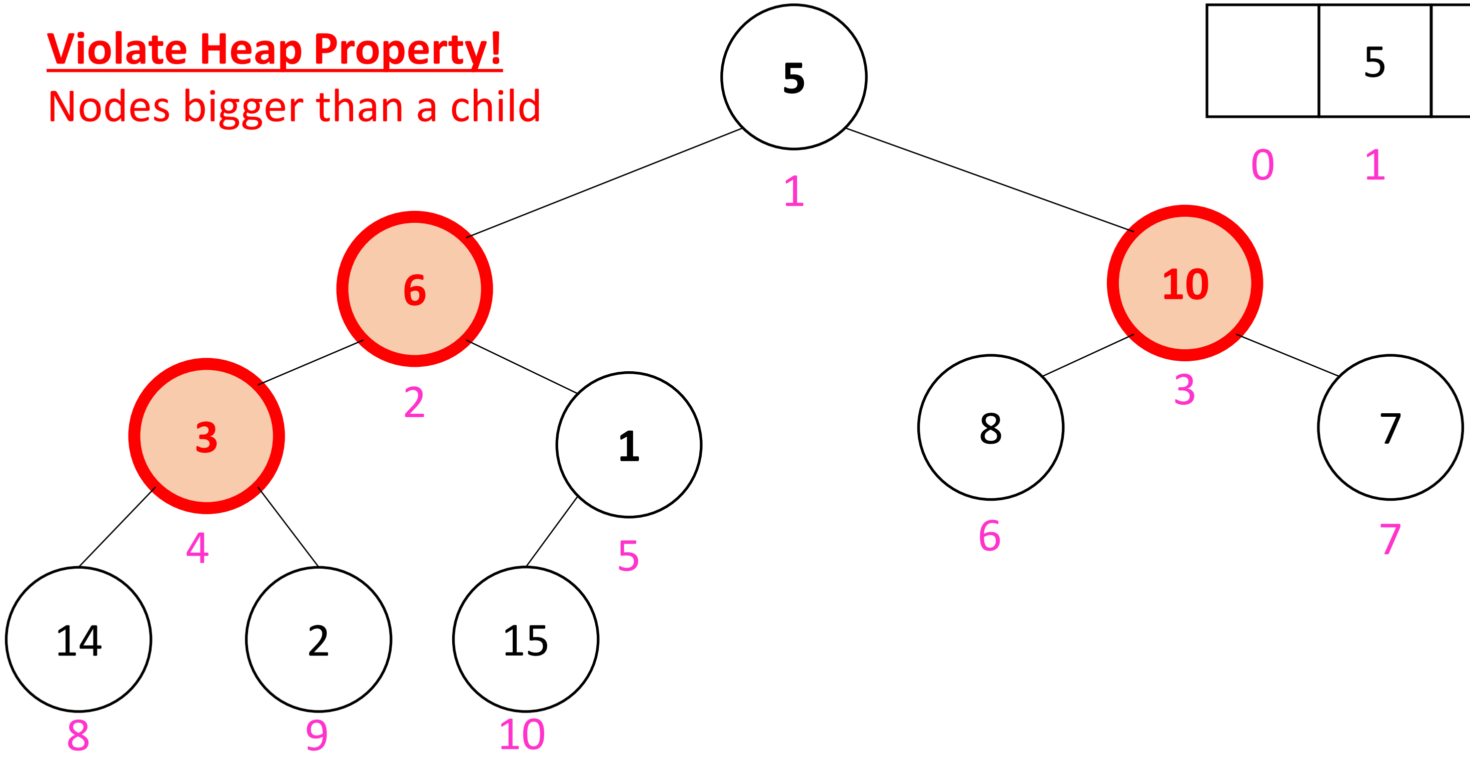
	5	6	10	3	15	8	7	14	2	1
0	1	2	3	4	5	6	7	8	9	10

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

Floyd's buildHeap method

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

Violate Heap Property!
Nodes bigger than a child

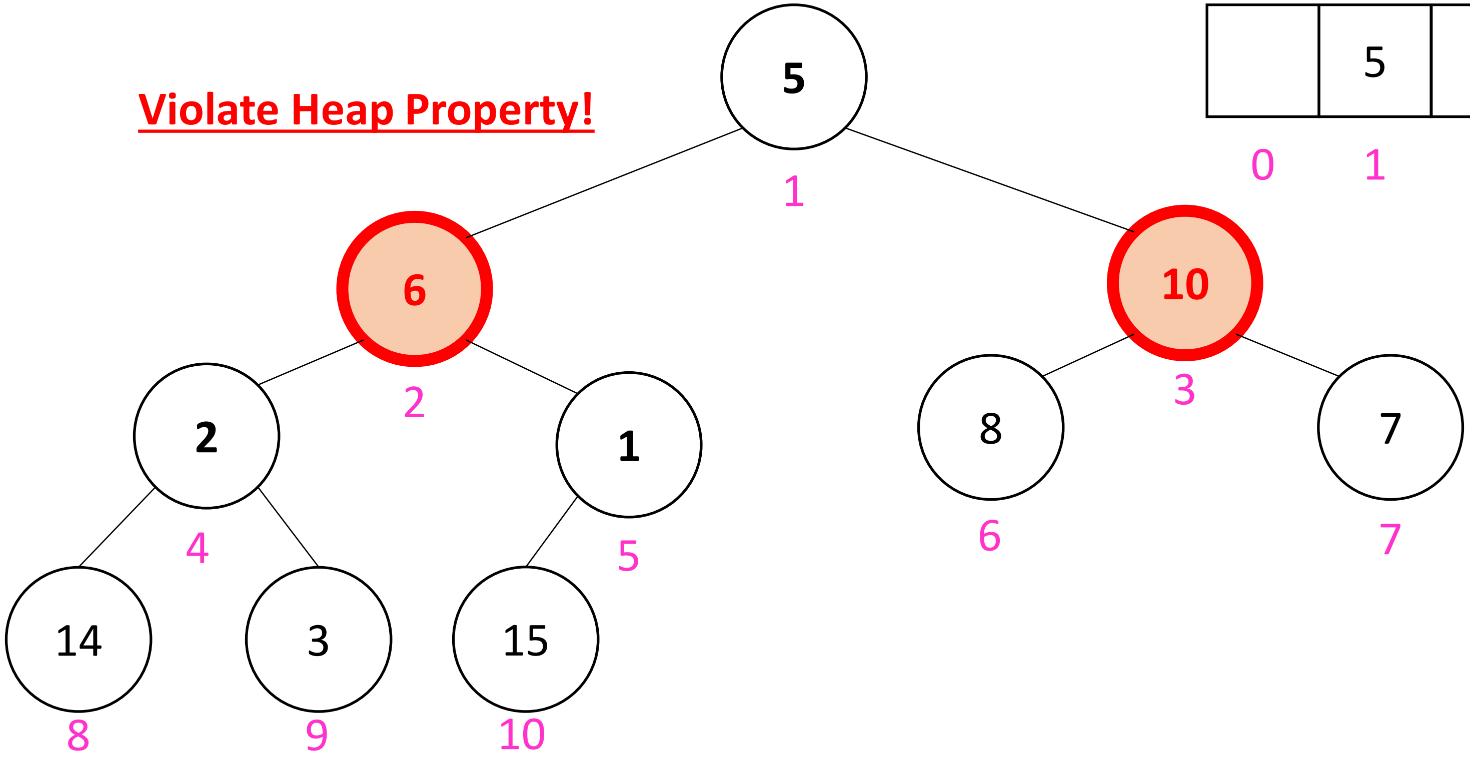


	5	6	10	3	15	8	7	14	2	1
0	1	2	3	4	5	6	7	8	9	10

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

Floyd’s buildHeap method

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

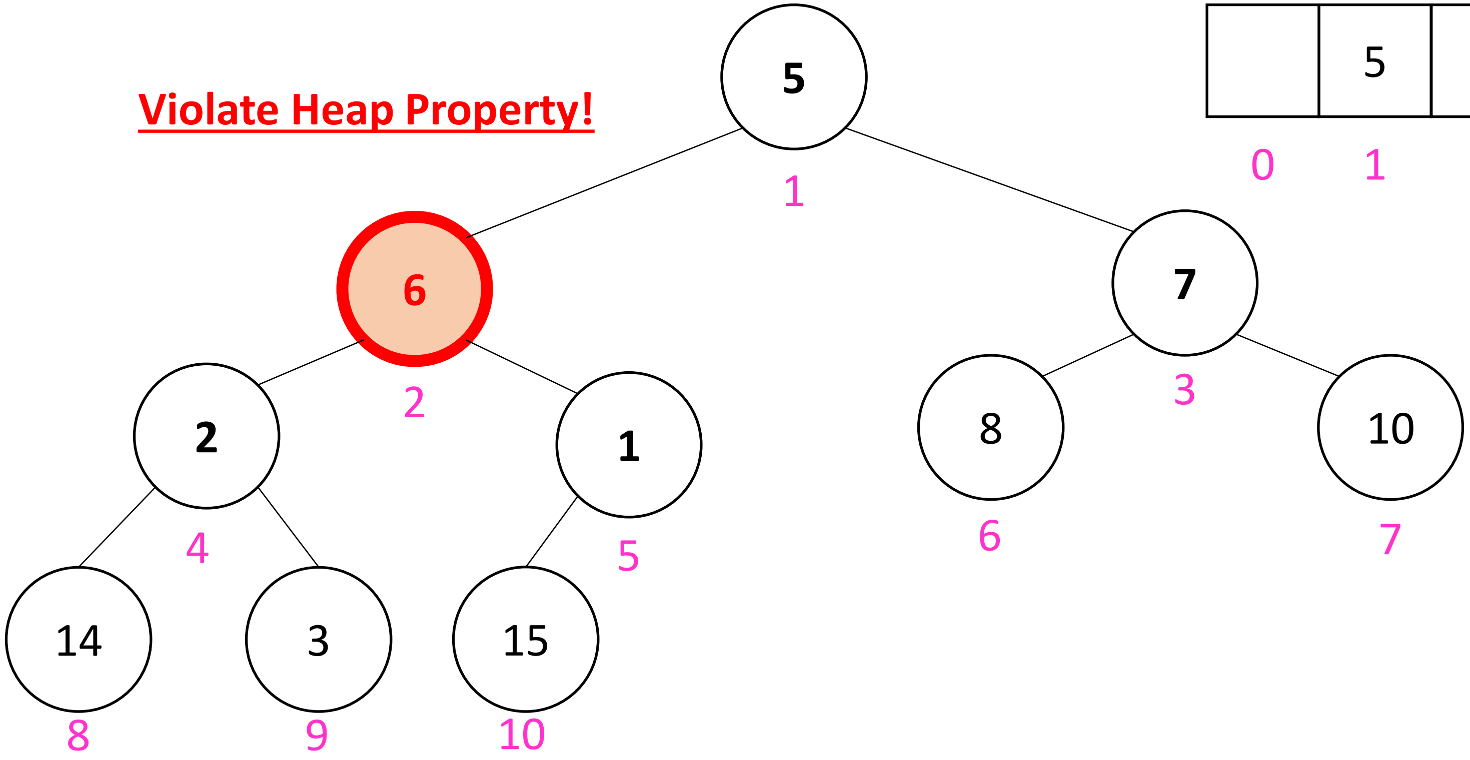


	5	6	10	3	15	8	7	14	2	1
0	1	2	3	4	5	6	7	8	9	10

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

Floyd's buildHeap method

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

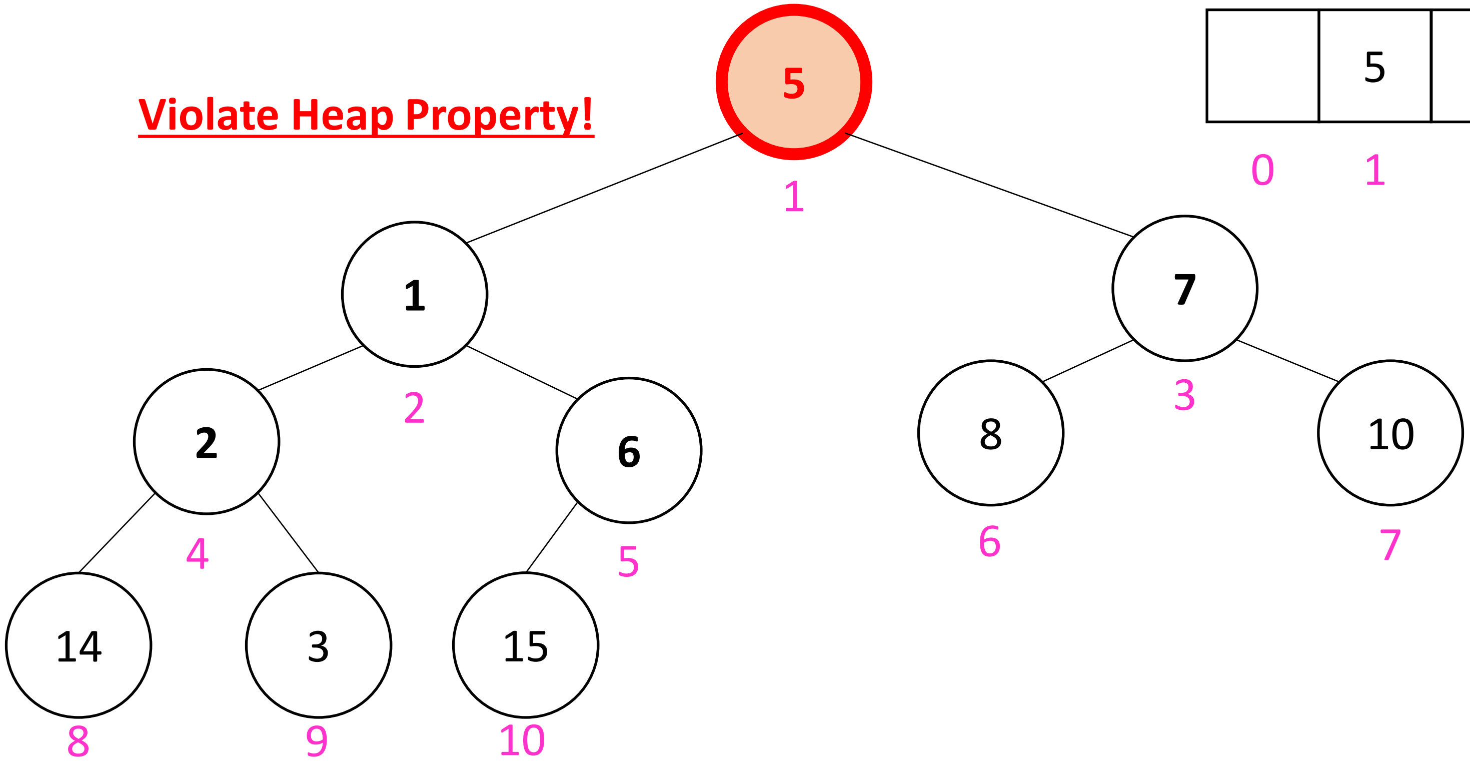


0	1	2	3	4	5	6	7	8	9	10
	5	6	10	3	15	8	7	14	2	1

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

Floyd’s buildHeap method

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



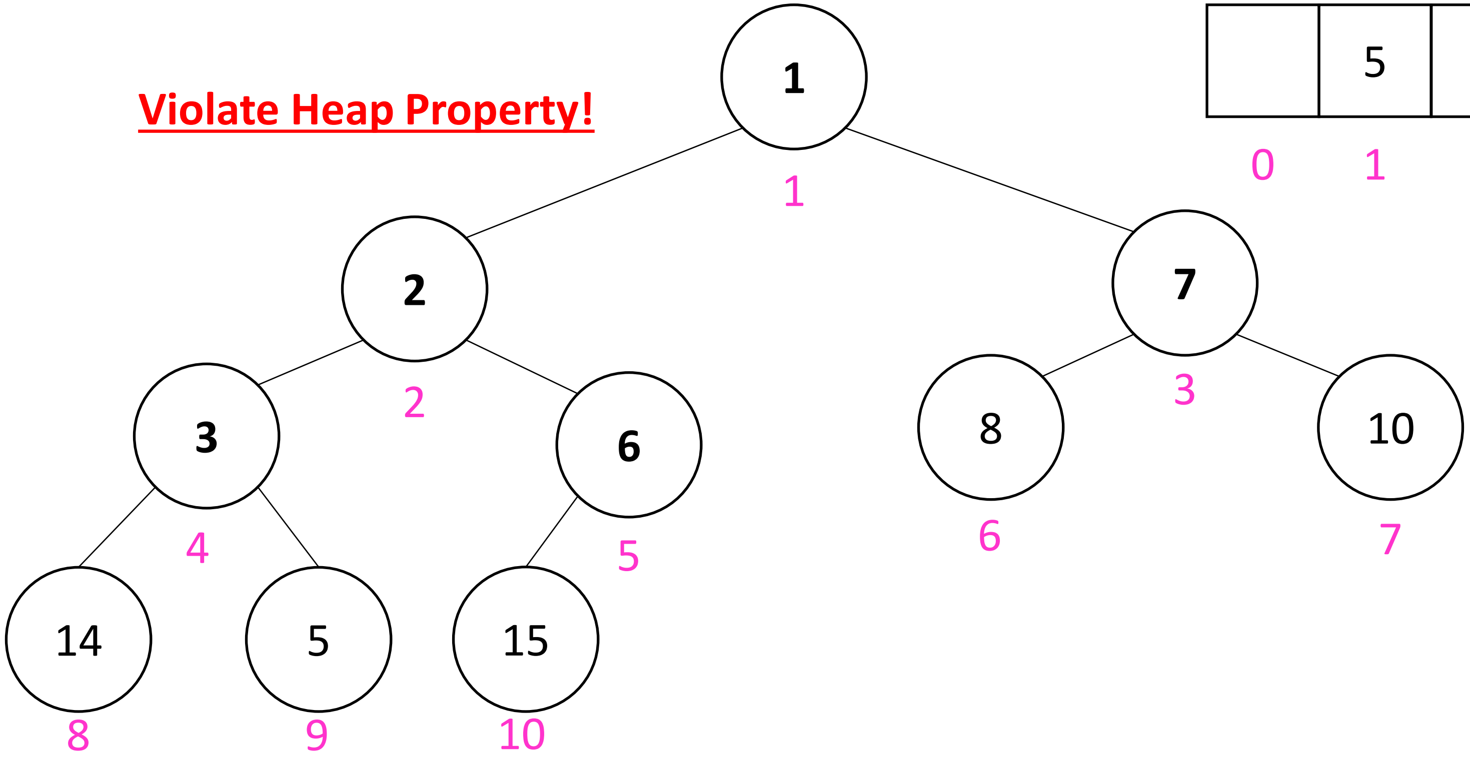
	5	6	10	3	15	8	7	14	2	1
0	1	2	3	4	5	6	7	8	9	10

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

Floyd's buildHeap method

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

Violate Heap Property!



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

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

How long did this take?

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

- Worst case running time of buildHeap:
- No node can percolate down more than the height of its subtree
  - When i is a leaf: 1
  - When i is second-from-last level: 2
  - When i is third-from-last level: 3
- Overall Running time:
  - $\frac{n}{2} + 2 \cdot \frac{n}{4} + 3 \cdot \frac{n}{8} + \cdots$
  - $\Theta(n)$

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	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$

Less Naïve attempts

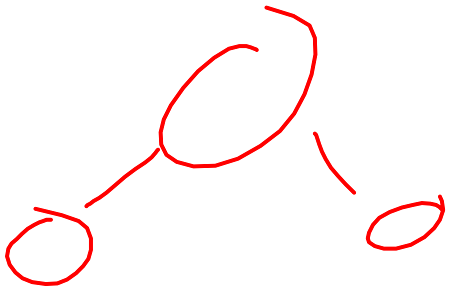
- Binary Search Trees (today)
- AVL Trees (Wednesday/Friday)
- Hash Tables (Next week)
- Red-Black Trees (not included in this course)
- Splay Trees (not included in this course)
- Tries (not included in this course)
- B-Trees (included in 331)

Naïve attempts

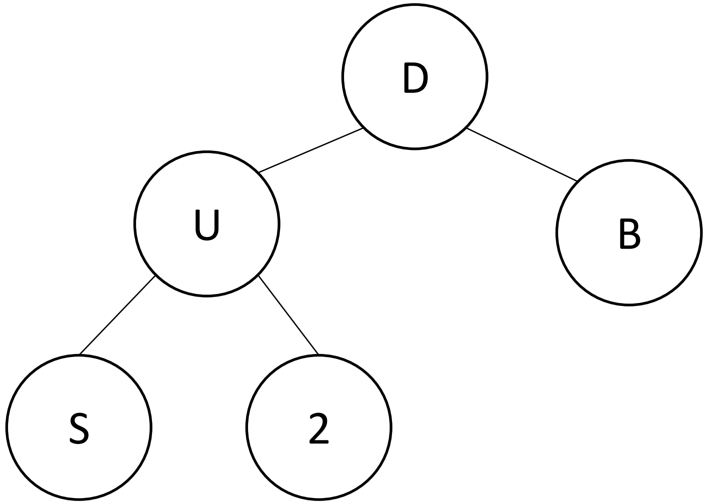
Data Structure	Time to insert	Time to find	Time to delete
Unsorted Array	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
Unsorted Linked List	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
Sorted Array	$\Theta(n)$	$\Theta(\log n)$	$\Theta(n)$
Sorted Linked List	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Heap	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$
Binary Search Tree (worst)	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Binary Search Tree (expected)	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$

Tree Height

```
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 1 + height;
}
```



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

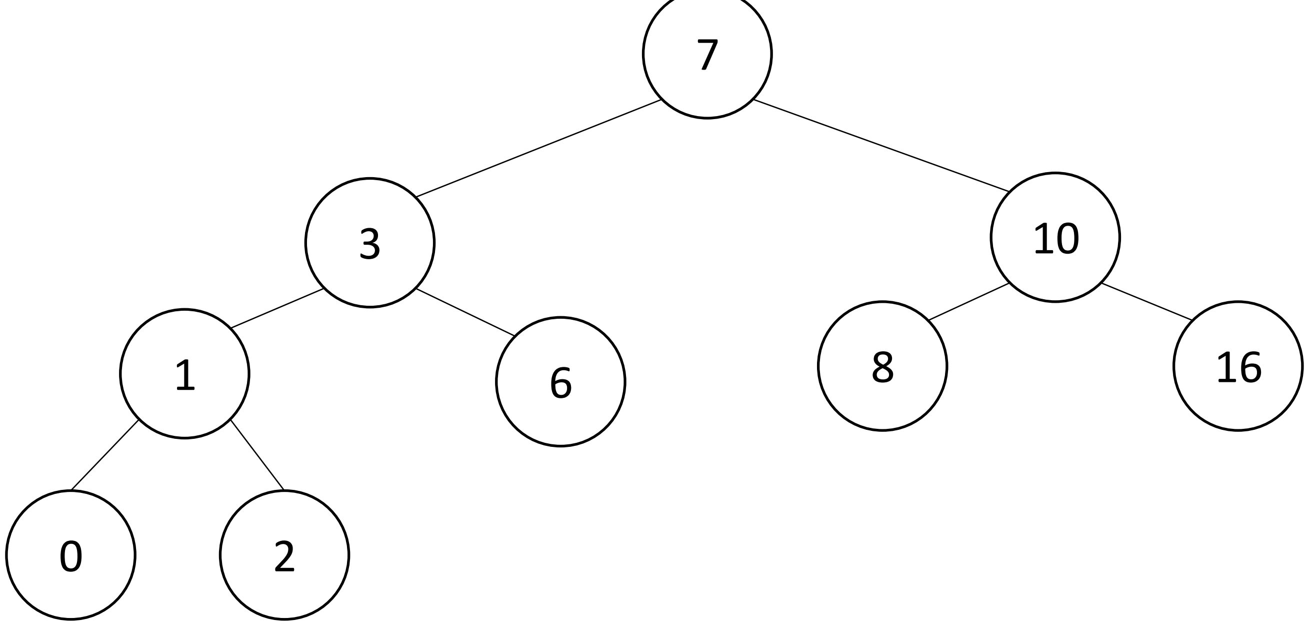
Name that Traversal!

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

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

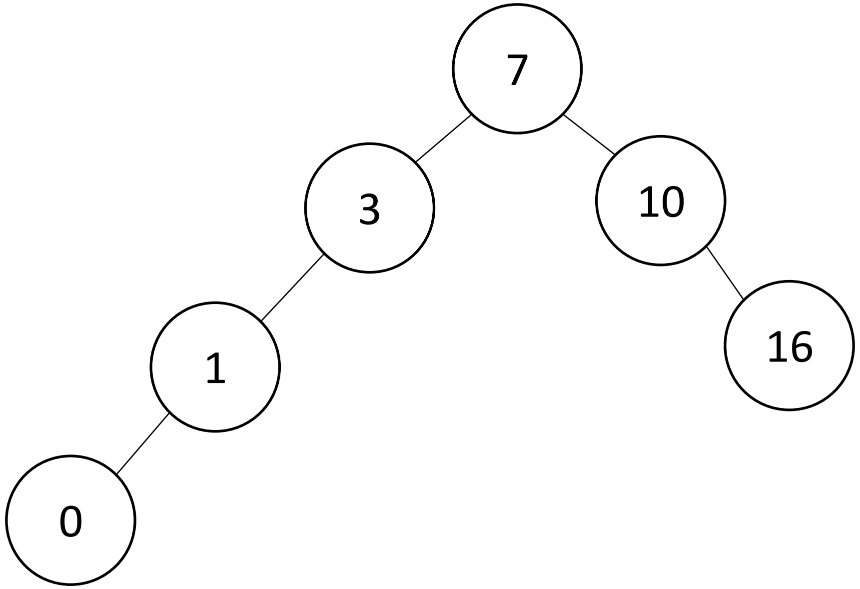
```
CorderTraversal(root){
        if (root.left != Null){
                process(root.left);
        }
        process(root)
        if (root.right != Null){
                process(root.right);
        }
}
```

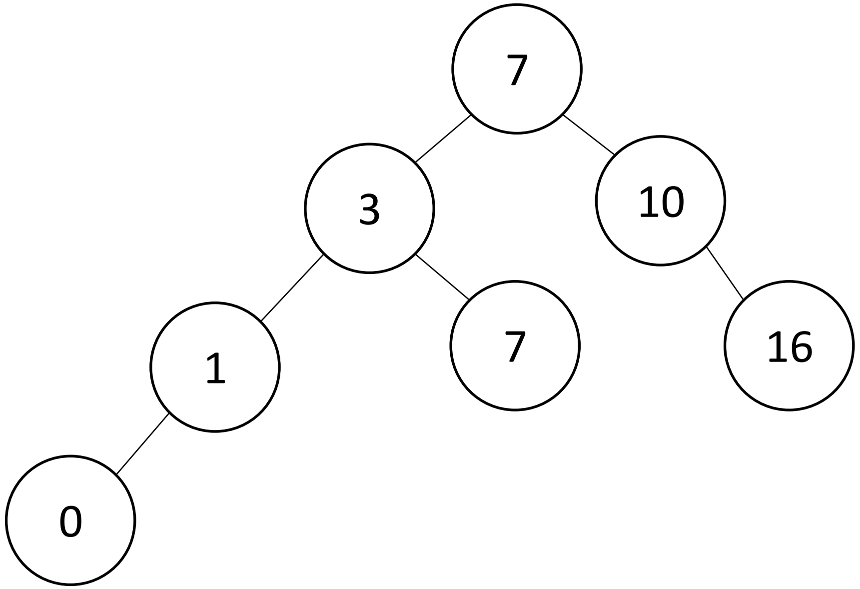
Binary Search Tree

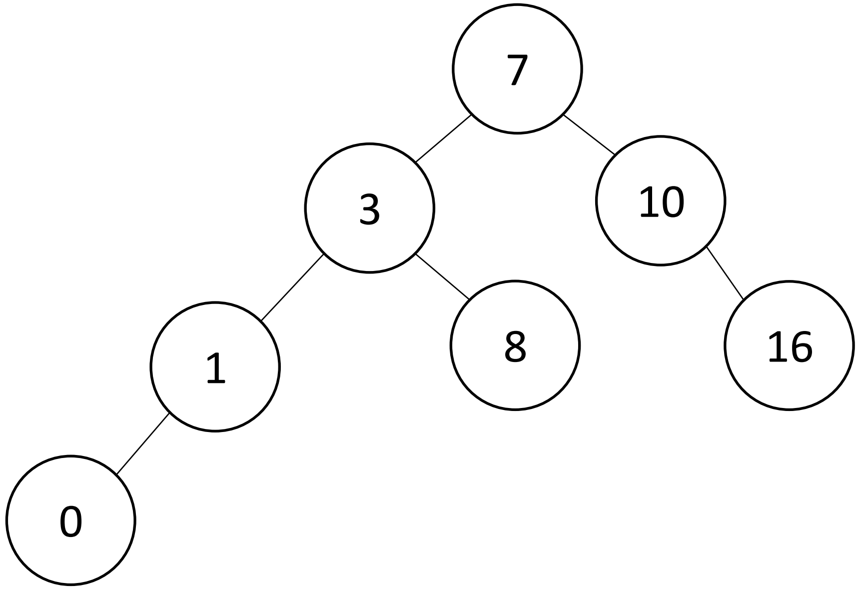


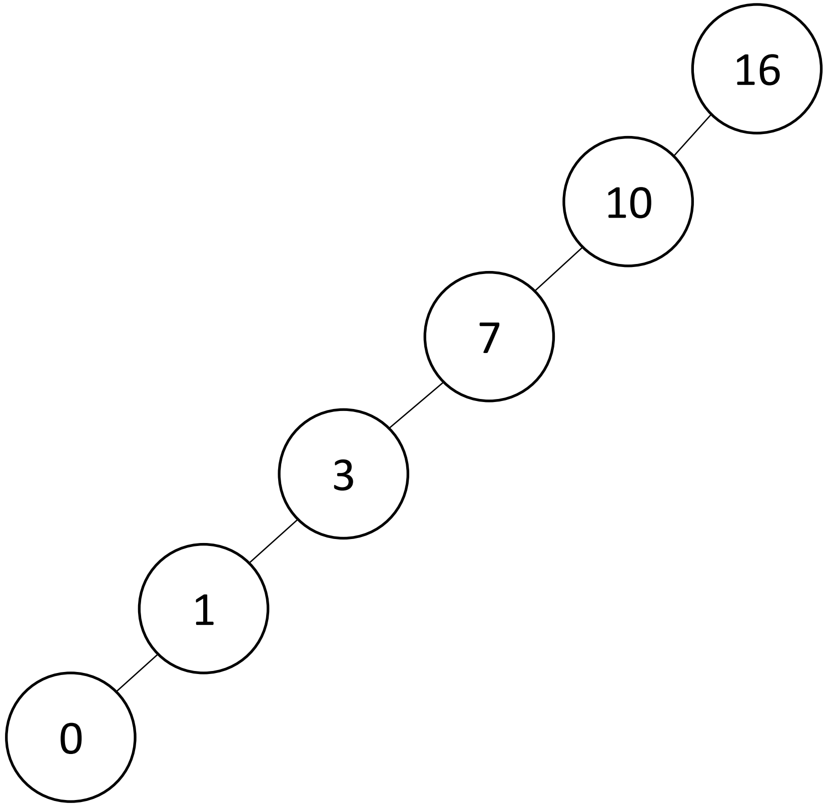
- Binary Tree
  - Definition:
- Order Property
  - All keys in the left subtree are smaller than the root
  - All keys in the right subtree are larger than the root
- Why?

Are these BSTs?







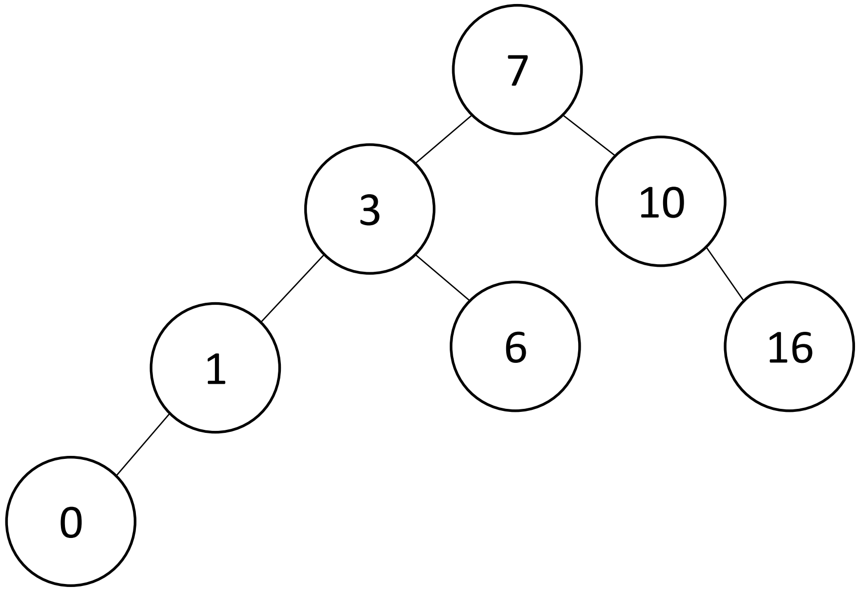


Aside: Why not use an array?

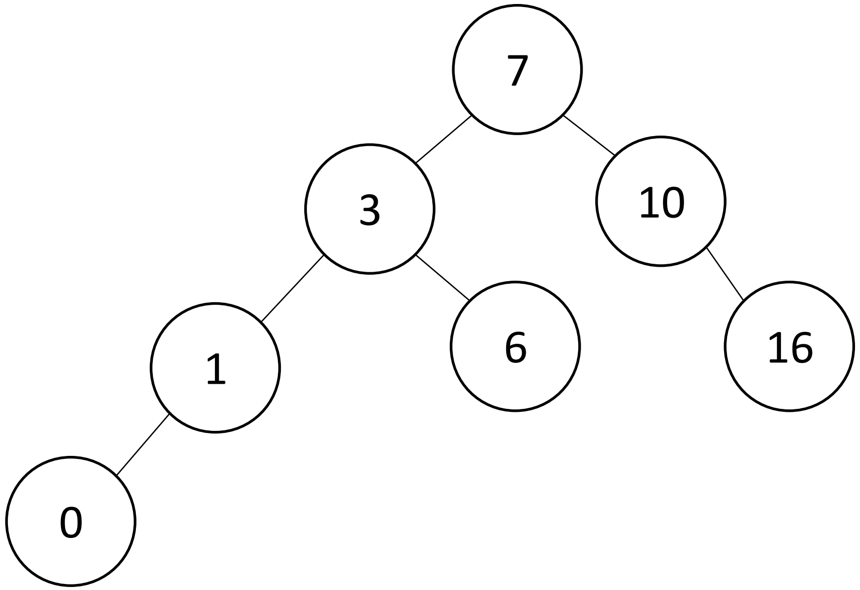
- We represented a heap using an array, finding children/parents by index
- We will represent BSTs with nodes and references. Why?

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;
}
```

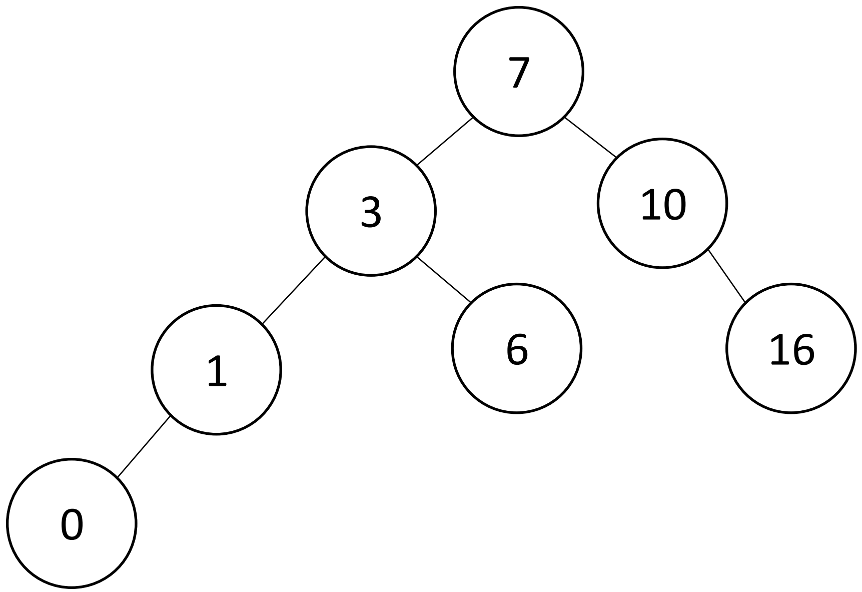


Find Operation (iterative)




```
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;
}
```

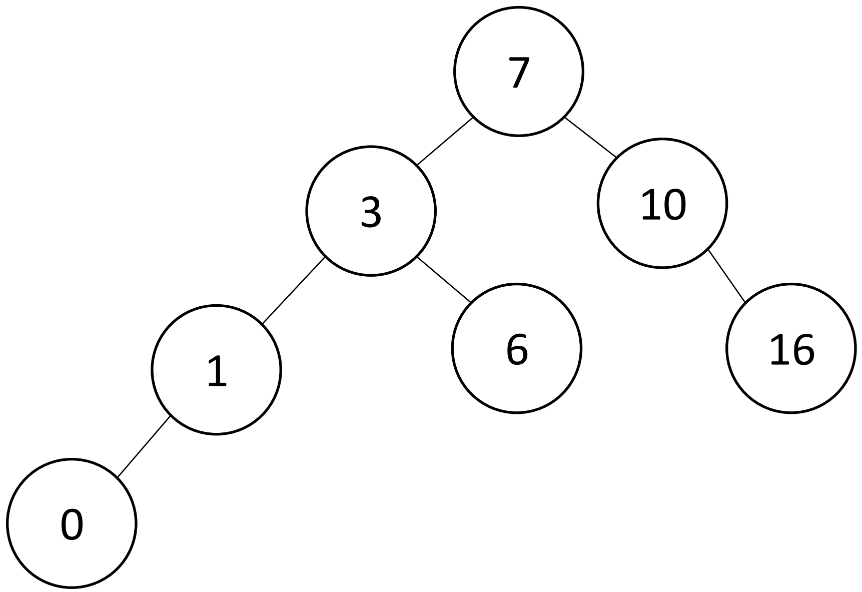
Insert Operation (recursive)




```
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!

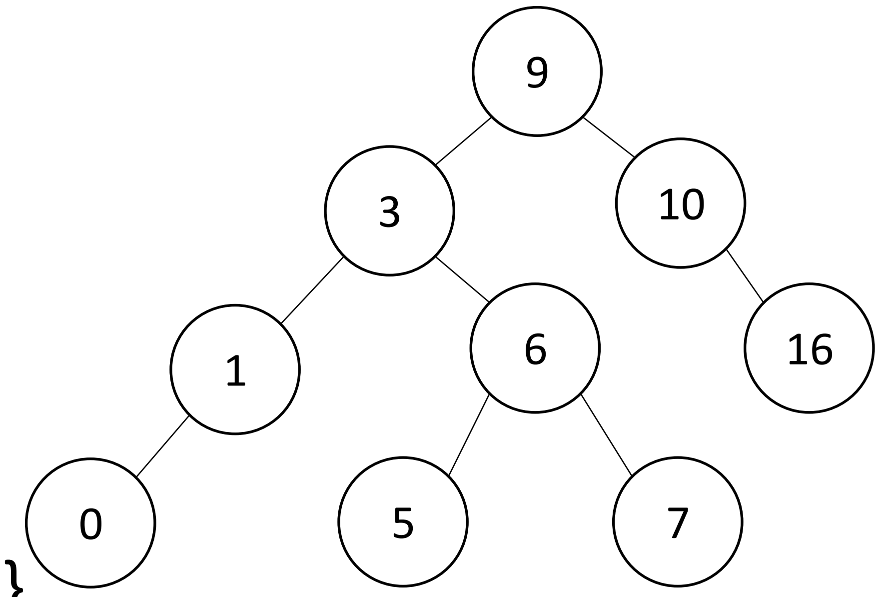
Insert Operation (iterative)




```
insert(key, value, root){
        if (root == Null){ 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!

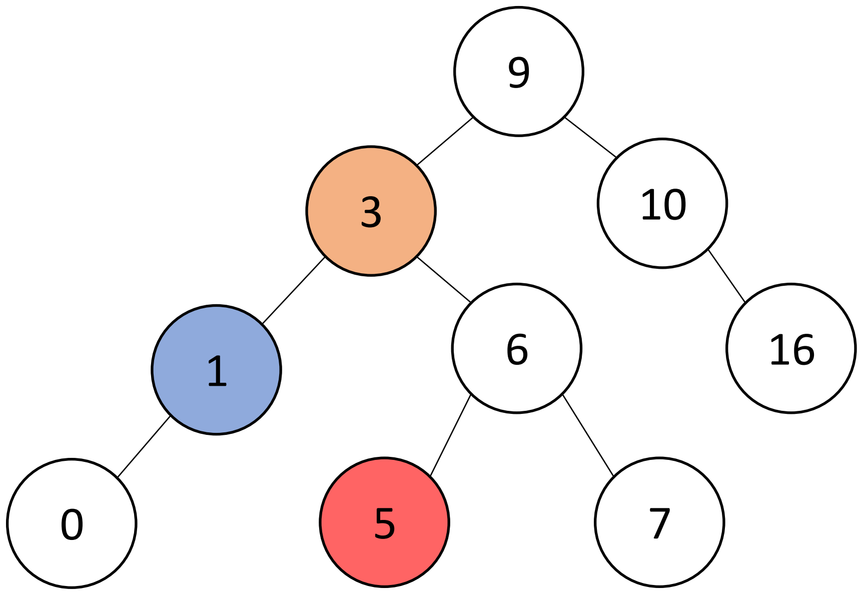
Delete Operation (iterative)



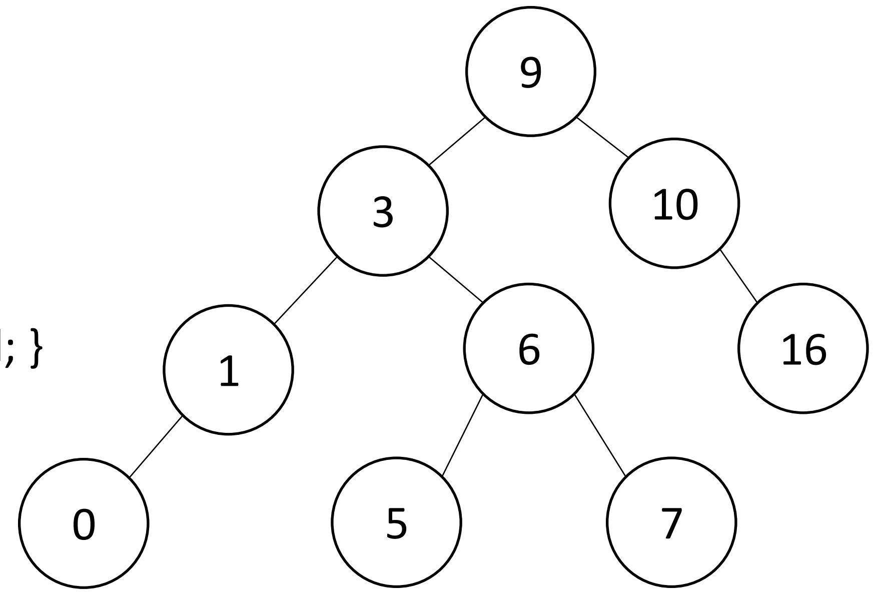

```
delete(key, root){
    while (root != Null && key != root.key){
        if (key < root.key){ root = root.left; }
        else if (key > root.key){ root = root.right; }
    }
    if (root == Null){ return; }
    // Now root is the node to delete, what happens next?
}
```

Delete – 3 Cases

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



Finding the Max and Min



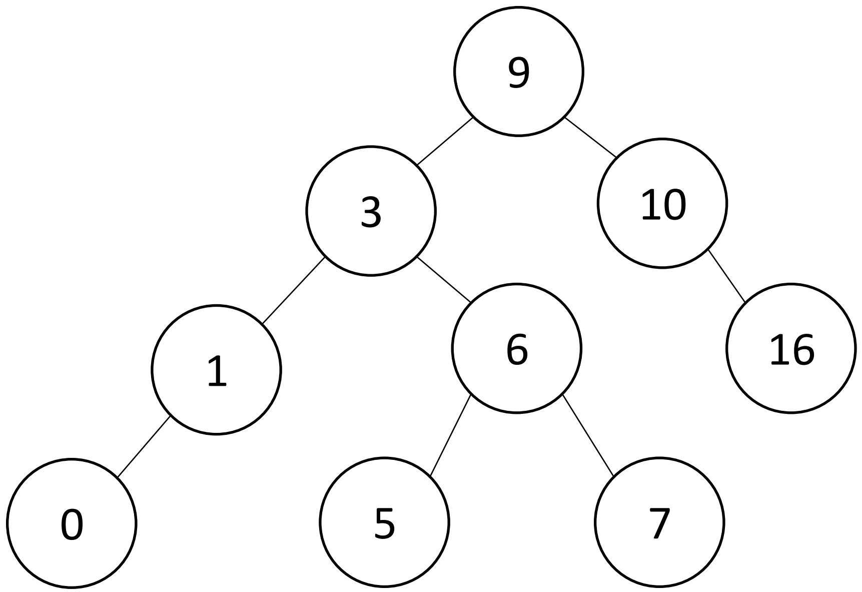
- Max of a BST:
  - Right-most Thing

```
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?