

# CSE 332 Autumn 2026

## Lecture 7: Dictionaries, BSTs

Ruth Anderson

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

# Announcements

- “Meet the staff” activity – Come meet us in office hours!
- Participate in Section, earn a late day (see Ed Board post)
- Exercise 1 Running Time: Closes TONIGHT, Closes Wed Oct 14
  - Entirely in Gradescope. Save your answers outside of Gradescope and confirm that you have answered/uploaded everything that is needed!
- Exercise 2 Heaps: due Fri Oct 16
- Exercise 3 Recurrences: due Fri Oct 23
- Midterm: 5-6pm Monday November 9
- Week 3 Survey in canvas: Due Tuesday Oct 20

# Dictionary (Map) ADT

- Contents:
  - Sets of key+value pairs
  - Keys must be:
    - Unique (always) and Comparable (for many dictionary implementations)
- 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)

# A Modest Few Uses for Dictionaries

Any time you want to store information according to some key and be able to retrieve it efficiently – a **dictionary** is the ADT to use!

- Lots of programs do that! (see even EX02...)

- |                      |                                          |
|----------------------|------------------------------------------|
| • Networks:          | router tables                            |
| • Operating systems: | page tables                              |
| • Compilers:         | symbol tables                            |
| • Databases:         | dictionaries with other nice properties  |
| • Search:            | inverted indexes, phone directories, ... |
| • Biology:           | genome maps                              |
| • ...                |                                          |

# Naïve attempts

| Data Structure       | Time to insert | Time to find | Time to delete |
|----------------------|----------------|--------------|----------------|
| Unsorted Array       |                |              |                |
| Unsorted Linked List |                |              |                |
| Sorted Array         |                |              |                |
| Sorted Linked List   |                |              |                |
| Binary Min Heap?     |                |              |                |

# Naïve attempts (Answers)

| 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 Min Heap?              | $\Theta(n)$      | $\Theta(n)$      | $\Theta(n)$      |
| Binary Search Tree (worst)    | $\Theta(n)$      | $\Theta(n)$      | $\Theta(n)$      |
| Binary Search Tree (expected) | $\Theta(\log n)$ | $\Theta(\log n)$ | $\Theta(\log n)$ |

# Lazy Deletion (e.g. in a sorted array)

|    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|
| 10 | 12 | 24 | 30 | 41 | 42 | 44 | 45 | 50 |
| ✓  | ✗  | ✓  | ✓  | ✓  | ✓  | ✗  | ✓  | ✓  |

A *general technique* for making **delete** as fast as find:

- Instead of actually removing the item just mark it deleted
- No need to shift values, etc.

Plusses:

- Simpler
- Can do removals later in batches
- If re-added soon thereafter, just unmark the deletion

Minuses:

- Extra *space* for the “is-it-deleted” flag
- Data structure full of deleted nodes wastes *space*
- **find**  $O(\log m)$  *time* where  $m$  is data-structure size ( $m \geq n$ )
- May complicate other operations

# Better Dictionary data structures

Will spend the next several lectures looking at dictionaries with two different data structures

## 1.AVL trees

- Binary search trees with *guaranteed balancing*

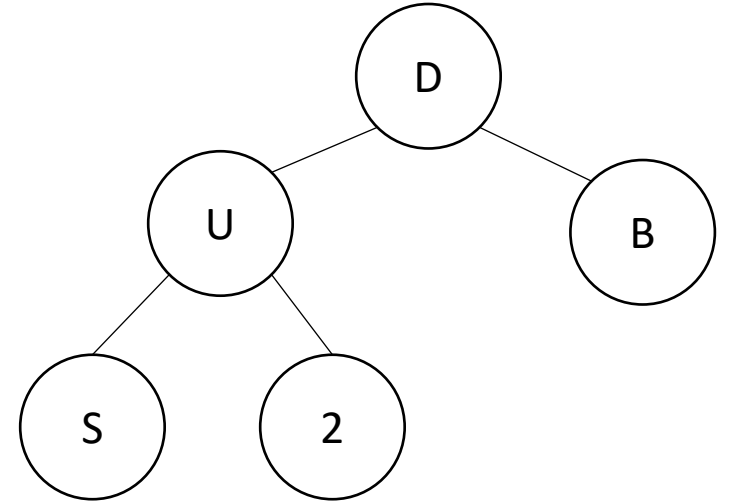
## 2.Hash Tables

- Not tree-like at all

Skipping: Other balanced trees (B-trees, red-black, splay)

# Tree Vocabulary

- 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



# 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);  
    }  
}
```

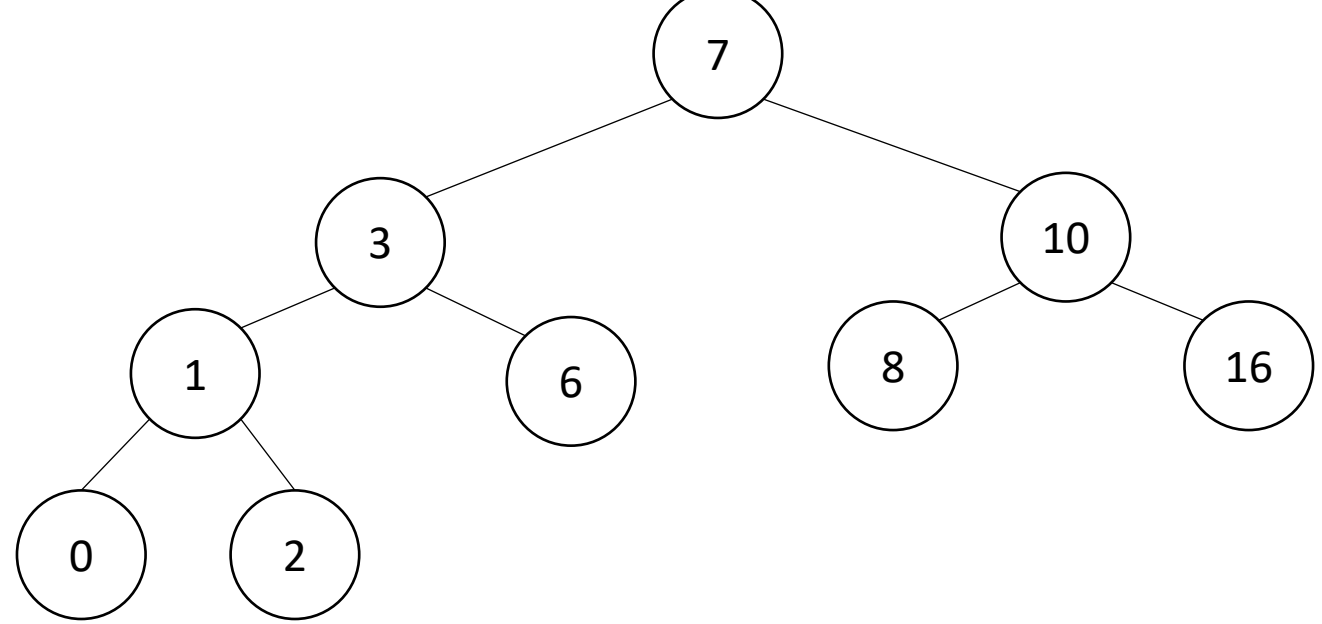
# Name that Traversal! (Answers)

```
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);  
    }  
}
```

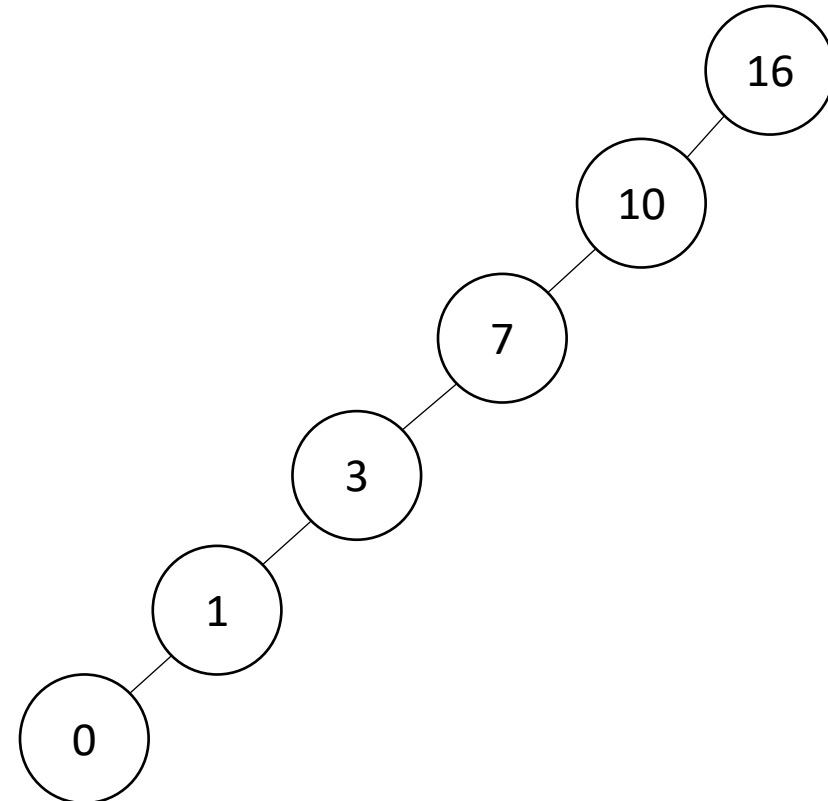
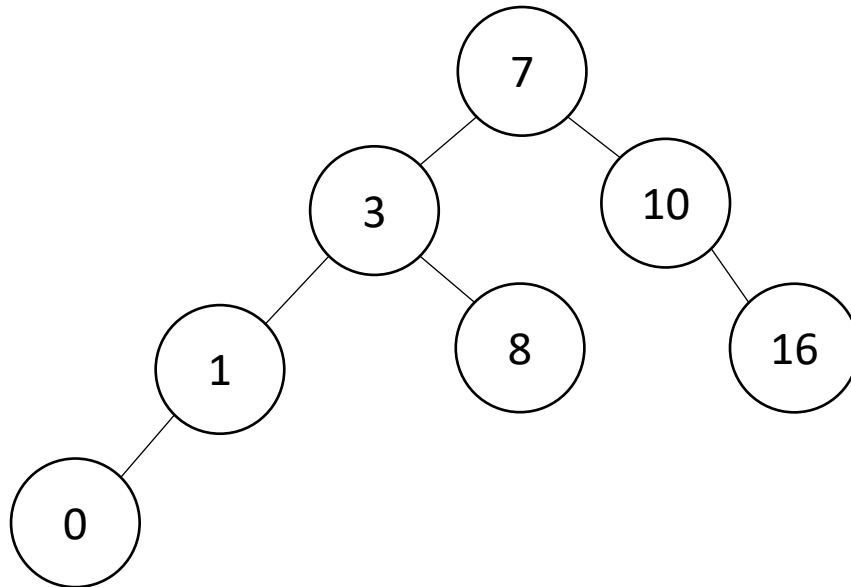
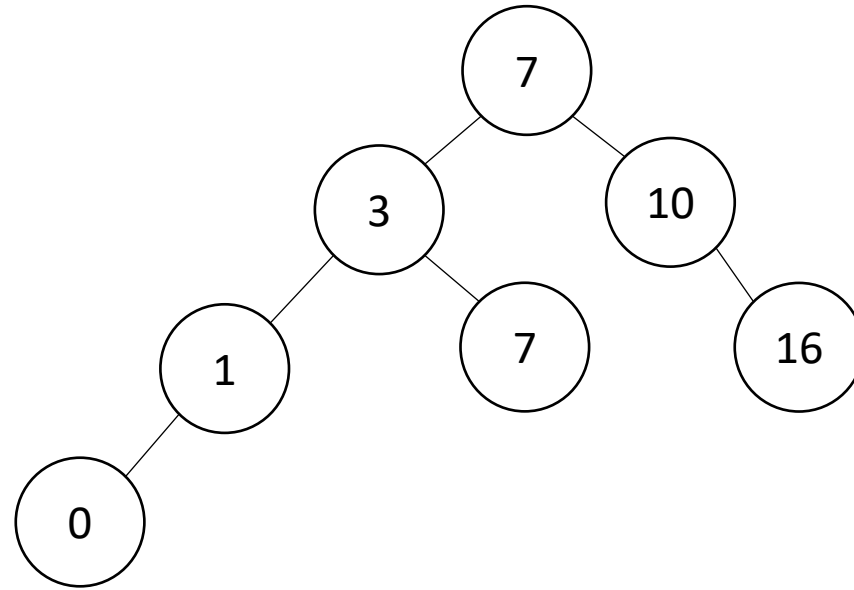
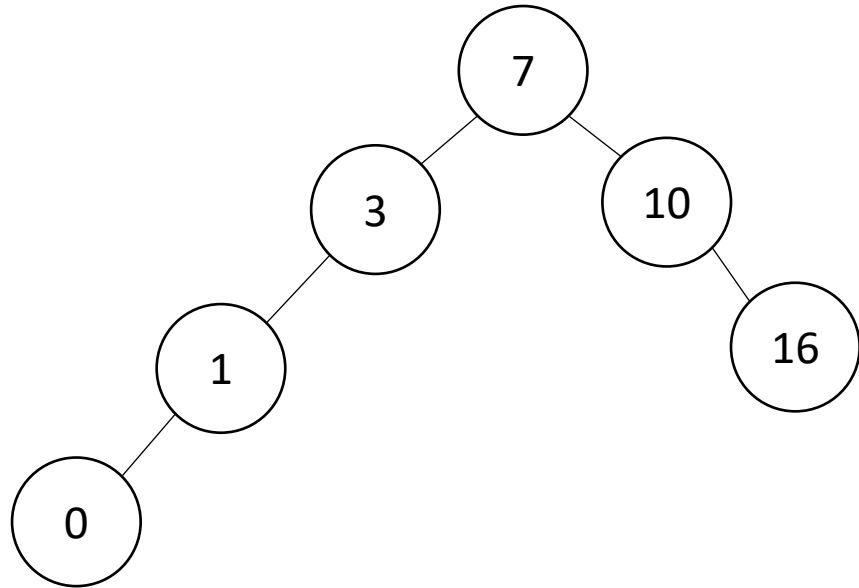
```
inOrder(root) {  
    if (root.left != Null) {  
        inOrder(root.left);  
    }  
    process(root)  
    if (root.right != Null) {  
        inOrder(root.right);  
    }  
}
```

# Binary Search Tree

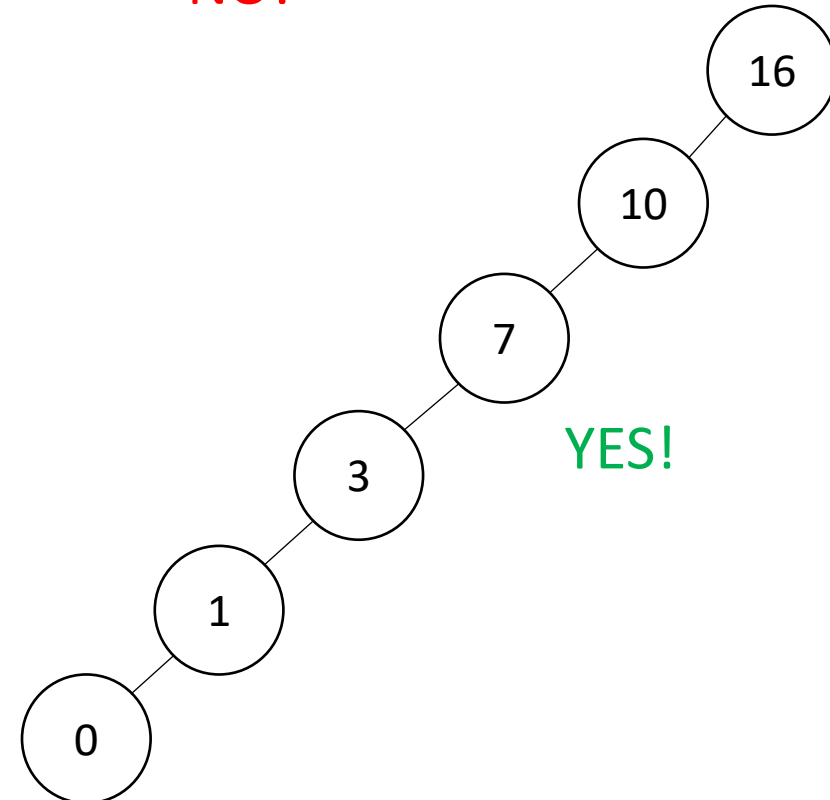
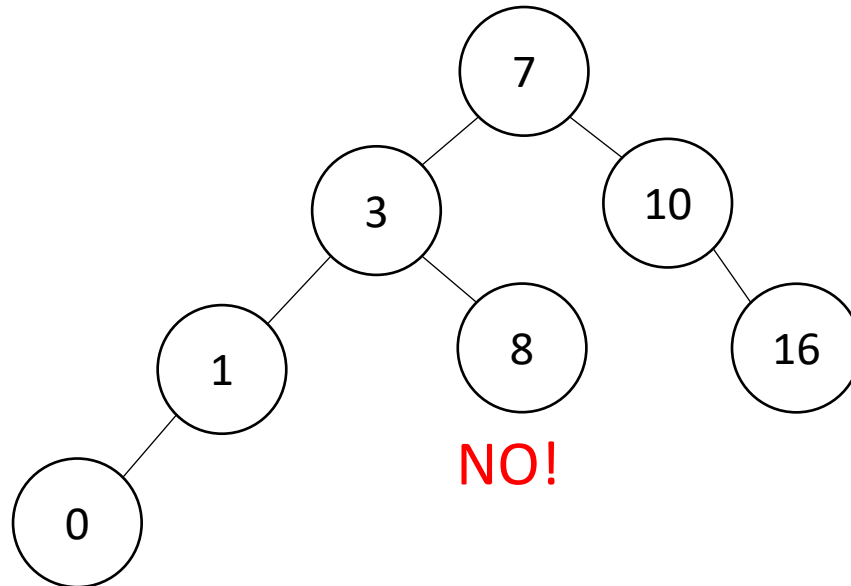
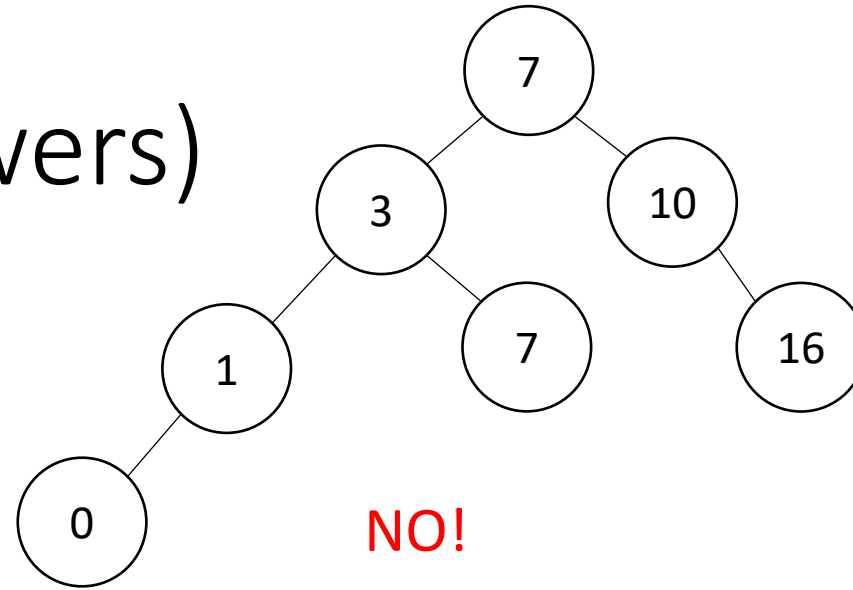
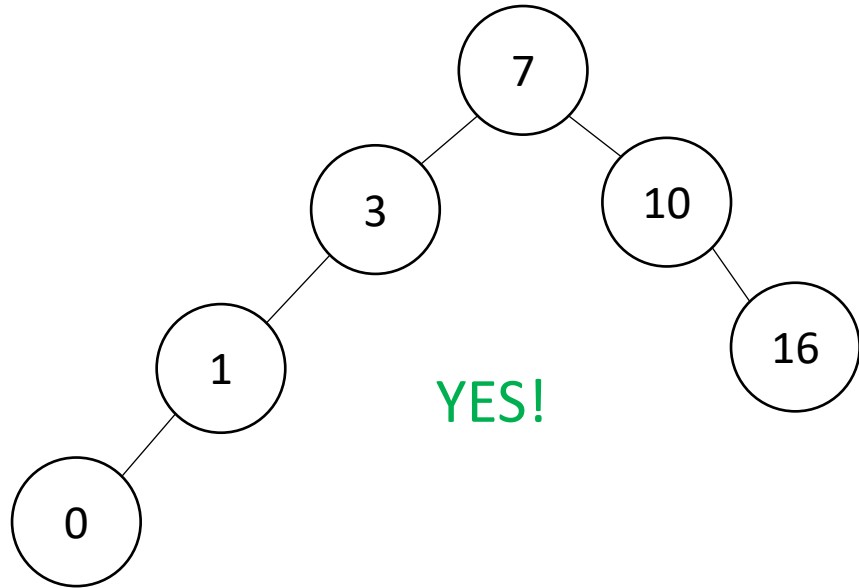


- Binary Tree
  - Definition:
    - Tree where each node has at most 2 children
- Order Property
  - All keys in the left subtree are smaller than the root
  - All keys in the right subtree are larger than the root
  - Consequence: cannot have repeated values

# Are these BSTs?



# Are these BSTs? (Answers)



# Binary Trees: Some Numbers

Recall: height of a tree = longest path from root to leaf (count edges)

For binary tree of height  $h$ :

- max # of leaves:
- max # of nodes:
- min # of leaves:
- min # of nodes:

# Binary Trees: Some Numbers (Answers)

Recall: height of a tree = longest path from root to leaf (count edges)

For binary tree of height  $h$ :

- max # of leaves:  $2^h$
- max # of nodes:  $2^{(h+1)} - 1$
- min # of leaves:  $1$
- min # of nodes:  $h + 1$

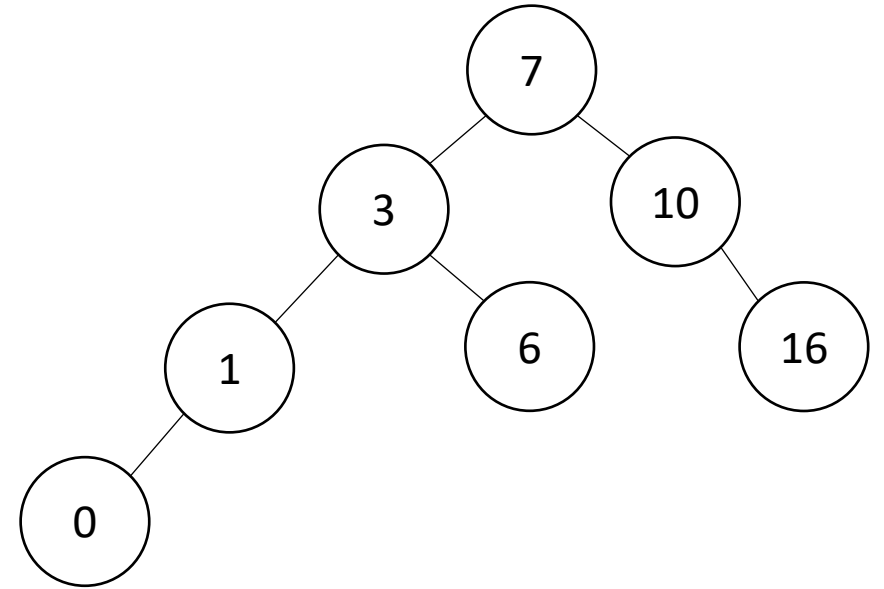
*For  $n$  nodes, we cannot do better than  $O(\log n)$  height,  
and we want to avoid  $O(n)$  height*

## 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!
    - $2^n$

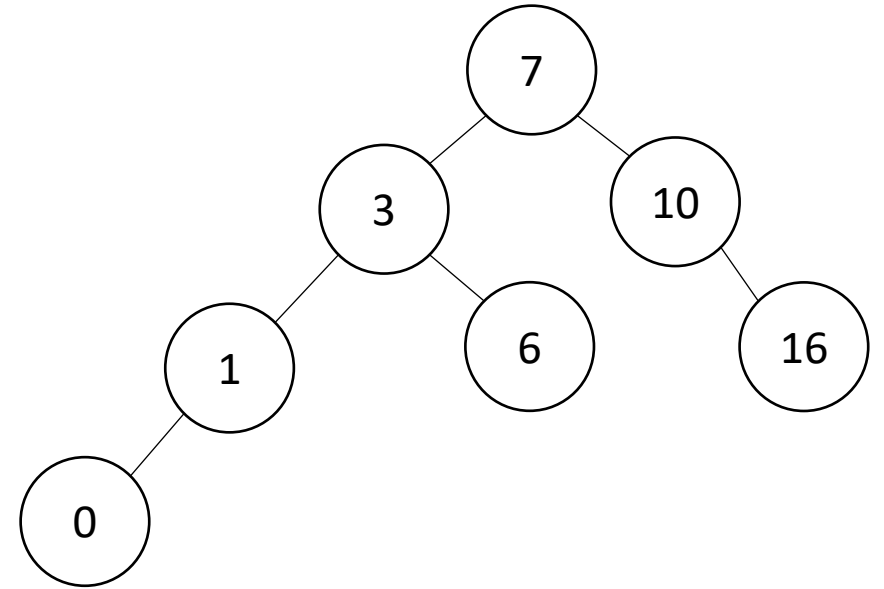
# Find Operation (recursive)

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



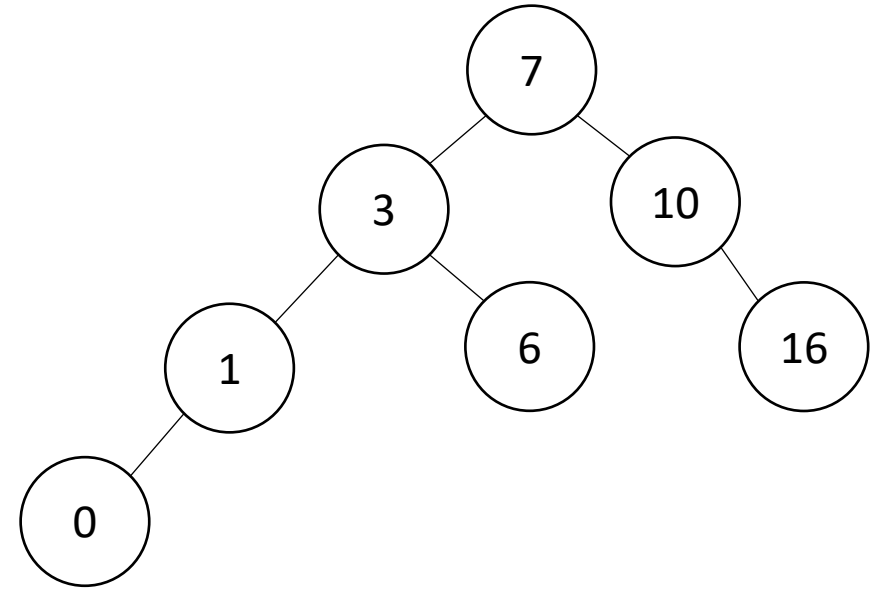
# Find Operation (iterative)

```
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)

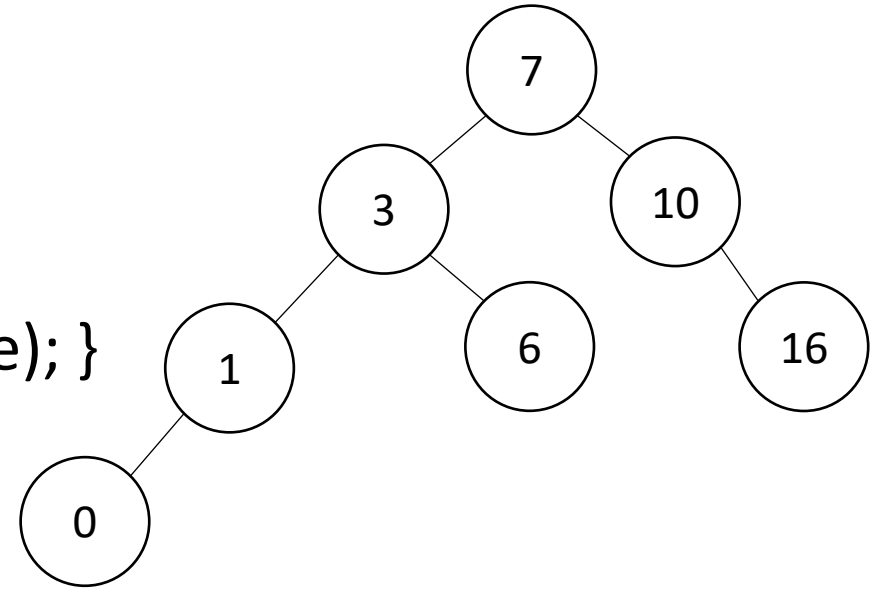
```
insert(key, value, root) {  
    root = insertHelper(key, value, root);  
}  
  
insertHelper(key, value, root) {  
    if (root == null)  
        return new Node(key, value);  
    if (root.key < key)  
        root.right = insertHelper(key, value, root.right);  
    else  
        root.left = insertHelper(key, value, root.left);  
    return root;  
}
```



**Note: Insert happens only at the leaves!**

# Insert Operation (iterative)

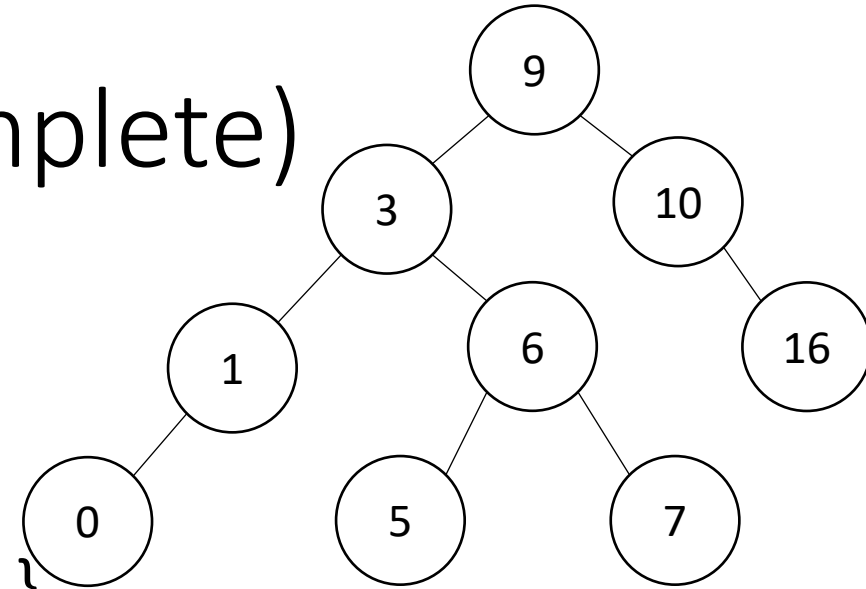
```
insert(key, value, root) {  
    if (root == Null) { this.root = new Node(key, value); }  
    parent = Null;  
    while (root != Null && key != root.key) {  
        parent = root;  
        if (key < root.key) { root = root.left; }  
        else if (key > root.key) { root = root.right; }  
 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

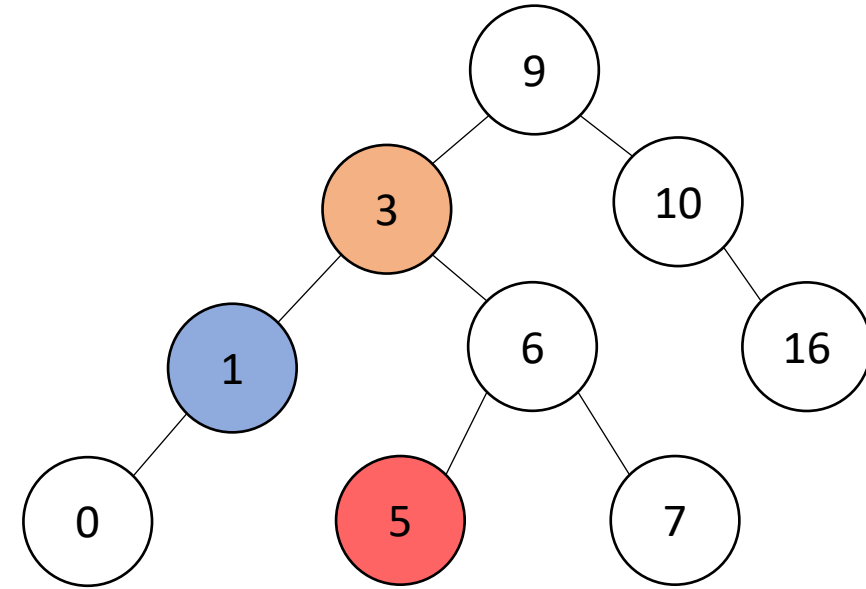
# Delete Operation (iterative, incomplete)

```
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) (e.g. delete 5)
- 1 Child (e.g. delete 1)
  - Replace the deleted node with its child
- 2 Children (e.g. delete 3)
  - Replace the deleted with the largest node to its left or else the smallest node to its right

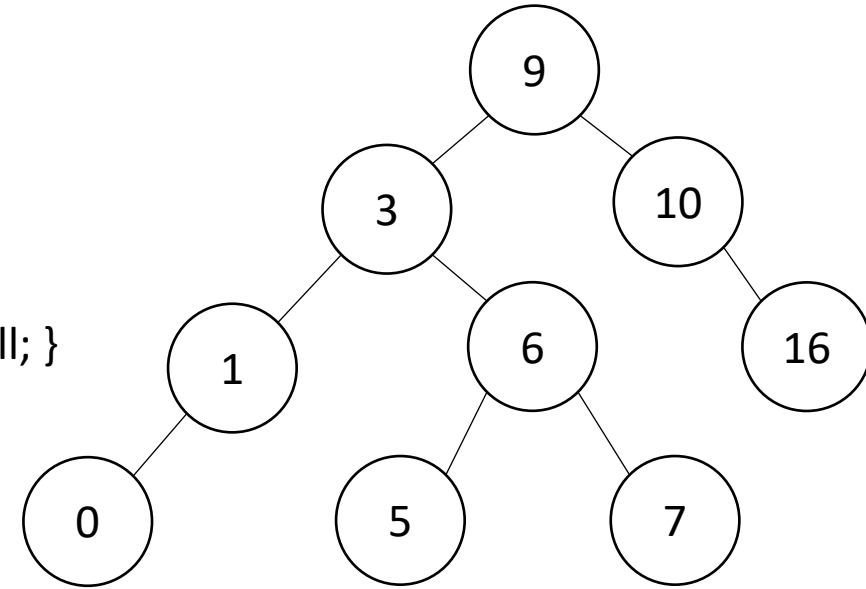


# Finding the Max and Min

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

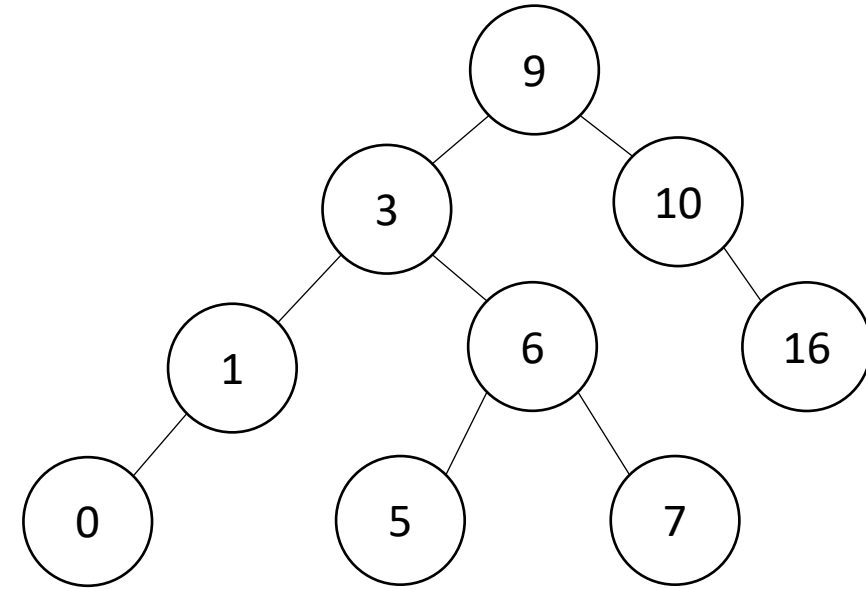
```
maxNode(root) {  
    if (root == Null) { return Null; }  
    while (root.right != Null) {  
        root = root.right;  
    }  
    return root;  
}
```

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



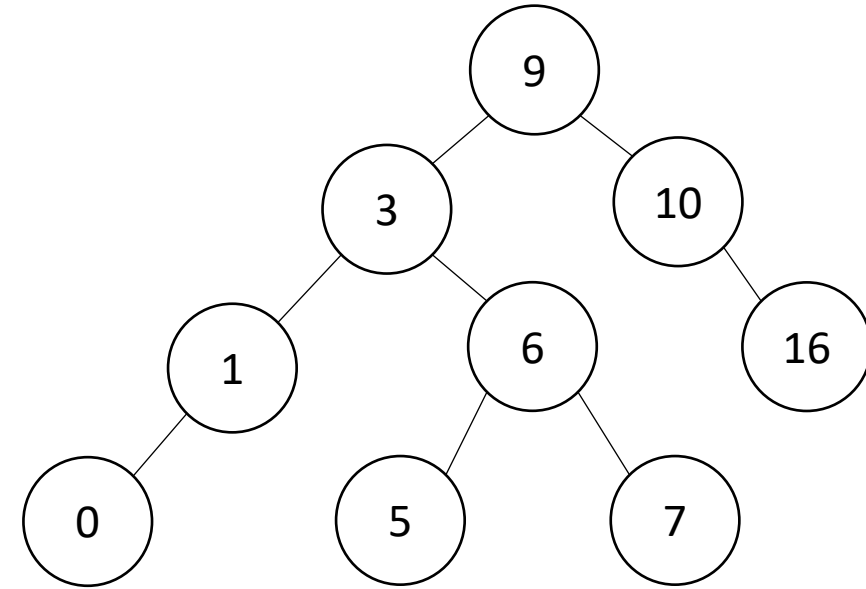
# Delete Operation (iterative)

```
delete(key, root) {  
    while (root != Null && 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  
    }  
}
```



# Delete Operation (recursive)

```
delete(key, root) {  
    if (root == Null) { 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) { 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!