CSE 332 Winter 2026 Lecture 7: Recurrences, Dictionaries, BSTs

Nathan Brunelle

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

Analysis of Recursive Algorithms

- Overall structure of recursion:
  - Do some non-recursive "work"
  - Do one or more recursive calls on some portion of your input
  - Do some more non-recursive "work"
  - Repeat until you reach a base case
- Running time: $T(n) = T(p_1) + T(p_2) + \cdots + T(p_x) + f(n)$
  - The time it takes to run the algorithm on an input of size n is:
  - The sum of how long it takes to run the same algorithm on each smaller input
  - Plus the total amount of non-recursive work done in that stack frame
- Usually:
  - $T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$
    - Called "divide and conquer"
  - $T(n) = T(n - c) + f(n)$
    - Called "chip and conquer"

Recursive List Summation

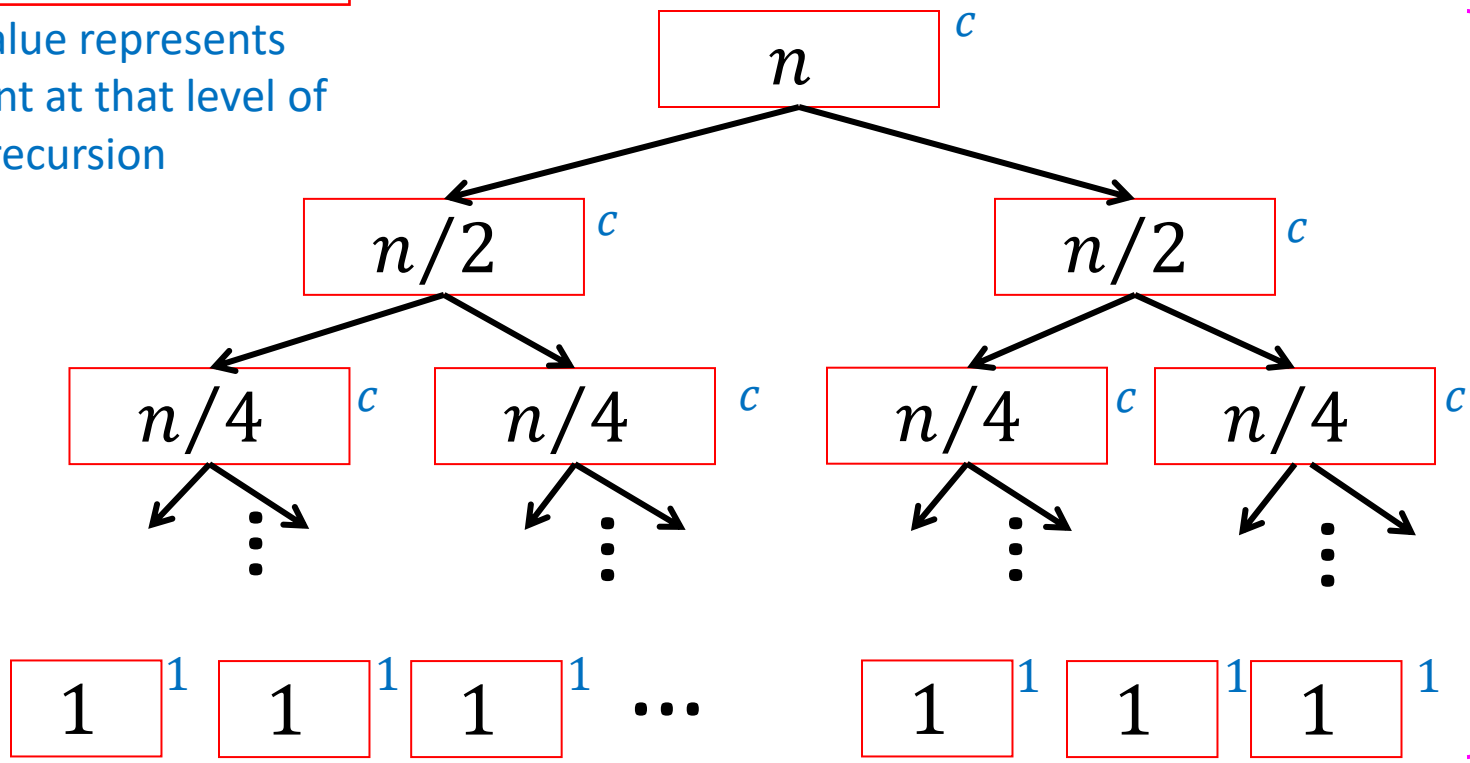
```
public int sum(int[] list){
    return sum_helper(list, 0, list.size);
}
private int sum_helper(int[] list, int low, int high){
    if (low == high){ return 0; }
    if (low == high-1){ return list[low]; }
    int middle = (high+low)/2;
    return sum_helper(list, low, middle) + sum_helper(list, middle, high);
}
```

Tree Method

Red box represents a problem instance

Blue value represents time spent at that level of recursion

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



$\Rightarrow 2^i \cdot c$ work per level

$\log_2 n$ levels of recursion

$$T(n) = \sum_{i=1}^{\log_2 n} 2^i \cdot c$$

Tree Method Summary: Chip and Conquer

- Recurrence looks like $T(n) = aT(n-b) + f(n)$
- Use the recurrence to draw a tree
  - a is the branching factor of the tree (e.g. if $a = 2$ then it's a binary tree)
  - Subtract b from the parent's input size to get children's input size
  - Work done per node is given by applying $f(n)$ to that node's input size
  - Height of the tree is $\frac{n}{b}$
    - Because that is the number of times we must subtract b until reaching a base case
    - Answer to the question "how many times must we subtract b until we reach 0?"
      - Any base case is a constant, so to reach a larger value would just be a constant change
- Use the tree to express running time as a series
  - Adding work done for each node level-by-level
  - Identify a pattern to express work done at level i as a function of i
  - Write a series using $i = 0$ up to $\frac{n}{b}$
- Solve the series

Tree Method Summary: Divide and Conquer

- Recurrence looks like $T(n) = aT\left(\frac{n}{b}\right) + f(n)$
- Use the recurrence to draw a tree
  - a is the branching factor of the tree (e.g. if $a = 2$ then it's a binary tree)
  - Divide the parent's input size by b to get children's input size
  - Work done per node is given by applying $f(n)$ to that node's input size
  - Height of the tree is $\log_b n$
    - Because that is the number of times we must divide by b until reaching a base case
    - Answer to the question "how many times must we divide by b until we reach 1?"
      - Any base case is a constant, so to reach a larger value would just be a constant change
- Use the tree to express running time as a series
  - Adding work done for each node level-by-level
  - Identify a pattern to express work done at level i as a function of i
  - Write a series using $i = 0$ up to $\log_b n$
- Solve the series

Let’s do some more!

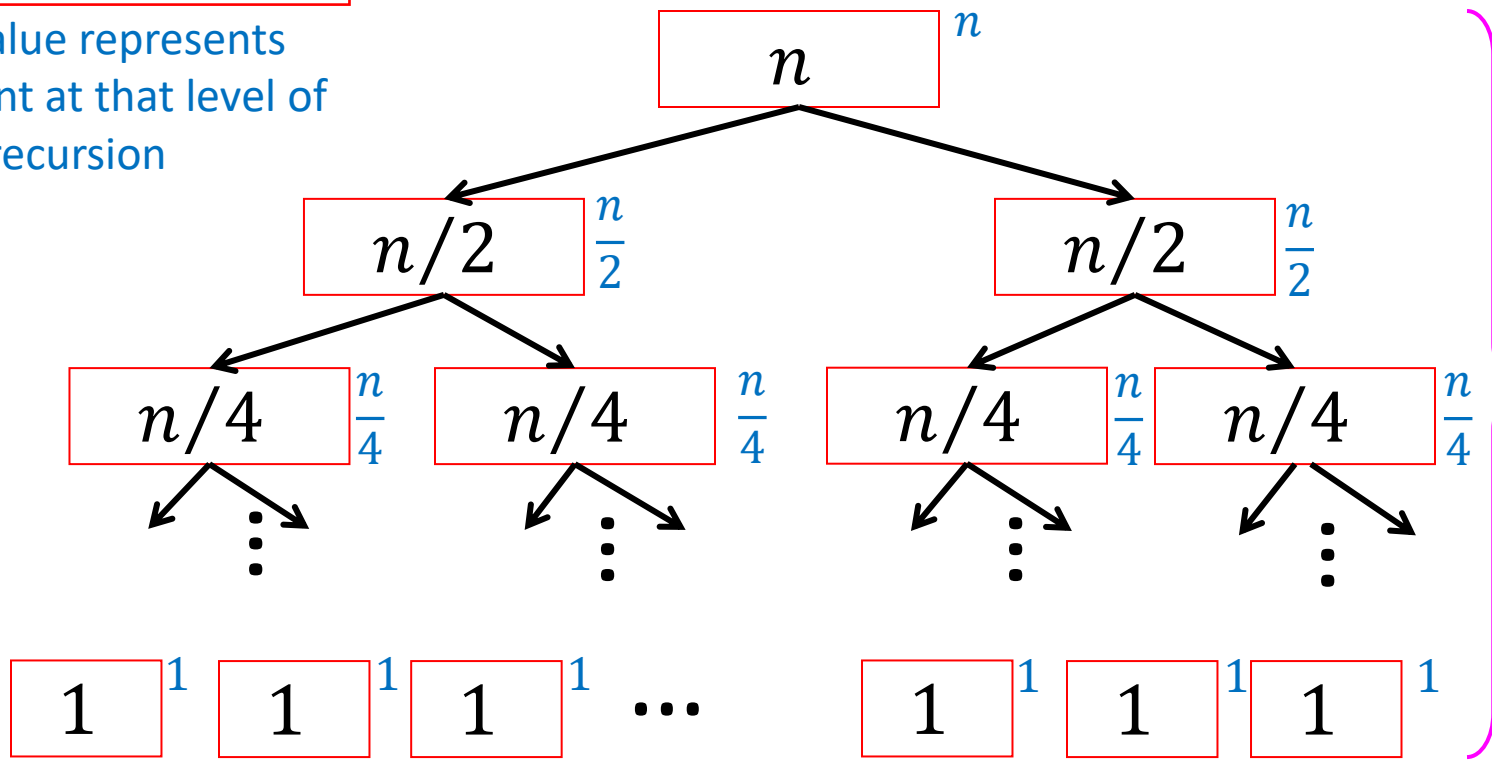
- For each, assume the base case is $n = 1$ and $T(1) = 1$
- $T(n) = 2T\left(\frac{n}{2}\right) + n$
- $T(n) = 2T\left(\frac{n}{2}\right) + n^2$
- $T(n) = 2T\left(\frac{n}{8}\right) + 1$

Tree Method

Red box represents a problem instance

Blue value represents time spent at that level of recursion

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



$\Rightarrow n$ work per level

$\log_2 n$ levels of recursion

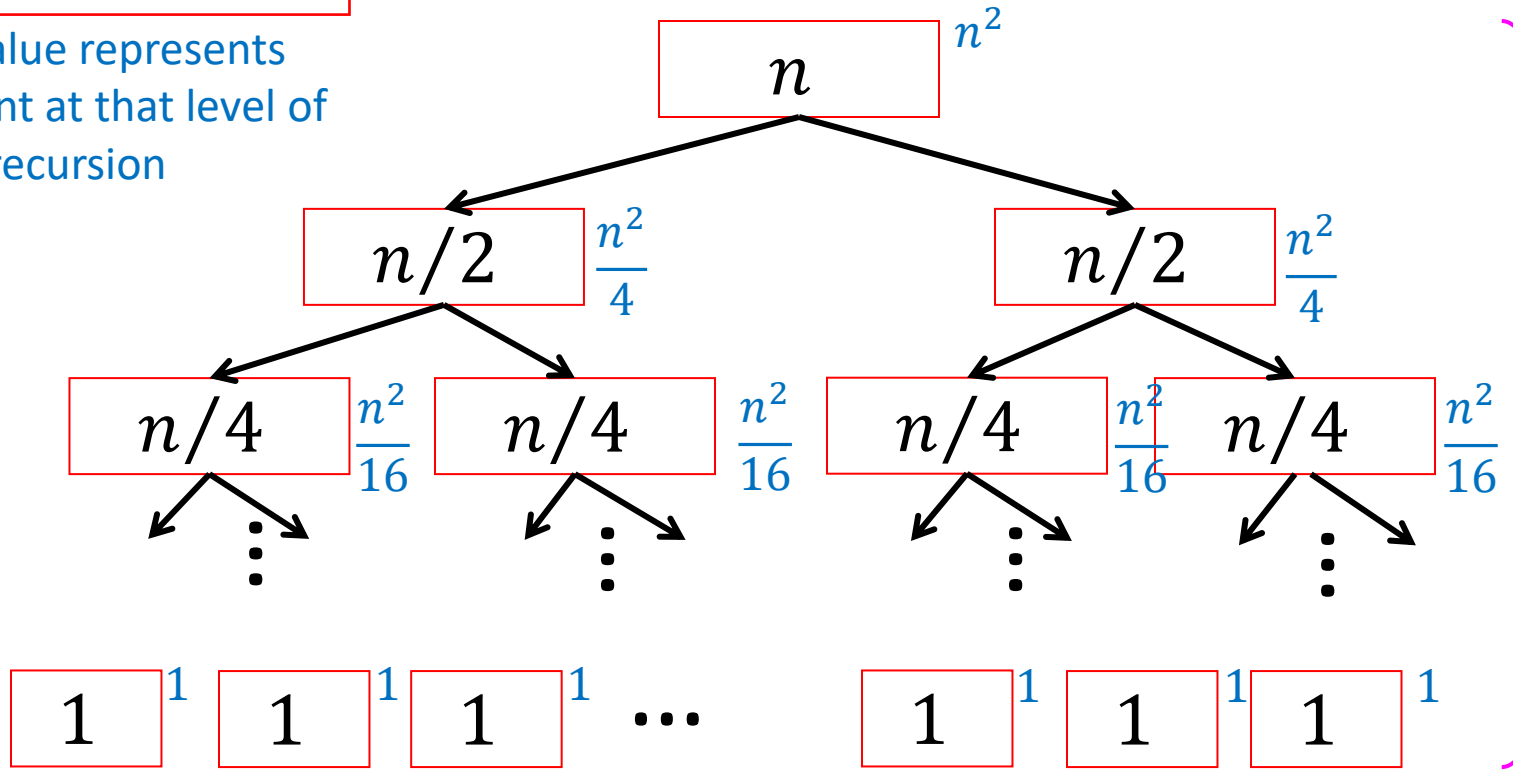
$$T(n) = \sum_{i=1}^{\log_2 n} n$$

Tree Method

Red box represents a problem instance

Blue value represents time spent at that level of recursion

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



$\Rightarrow$?? work per level

$\log_2 n$ levels of recursion

$$T(n) = \sum_{i=1}^{\log_2 n} ??$$

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

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

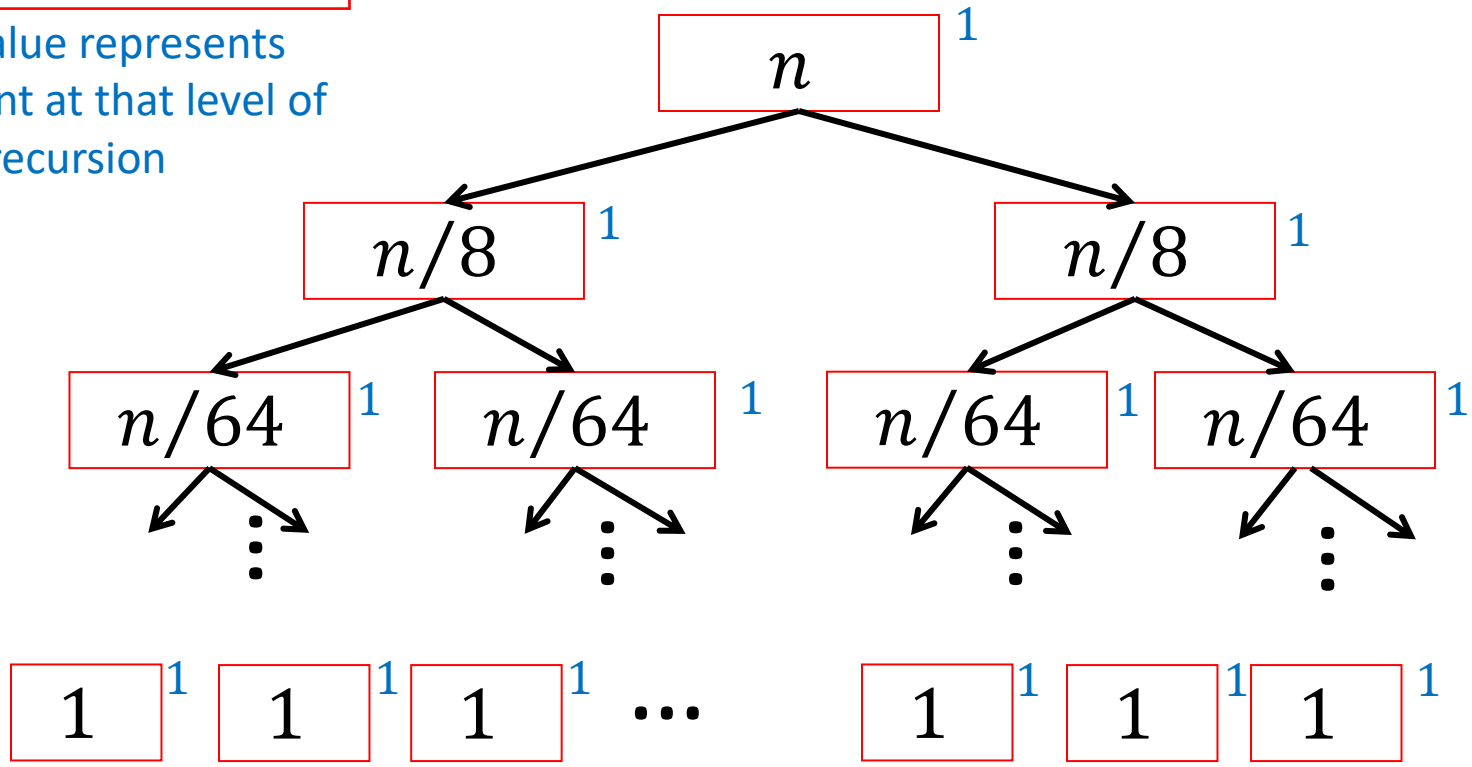
$$= n^2 \cdot \sum_{i=1}^{\log_2 n} \left(\frac{1}{2}\right)^i$$

Tree Method

Red box represents a problem instance

Blue value represents time spent at that level of recursion

$$T(n) = 2T\left(\frac{n}{8}\right) + 1$$



$\Rightarrow 2^i$ work per level

$\log_8 n$ levels of recursion

$$T(n) = \sum_{i=1}^{\log_8 n} 2^i$$

Solving $T(n) = 2T\left(\frac{n}{8}\right) + 1$

$$T(n) = \sum_{i=1}^{\log_8 n} 2^i$$

$$= \left(\frac{1 - 2^{\log_8 n}}{1 - 2}\right)$$

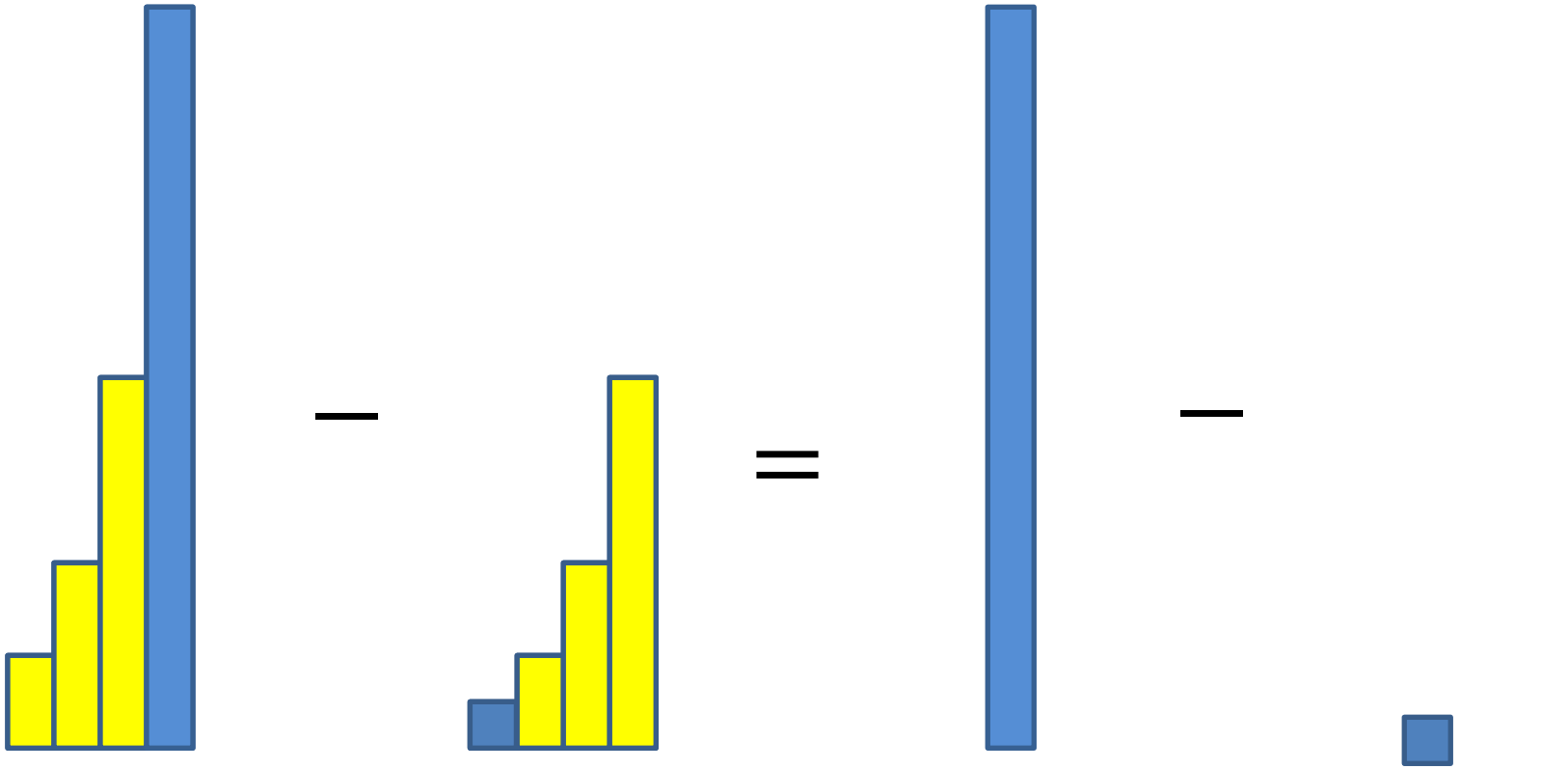
$$= 2^{\log_8 n} - 1$$

$$= n^{\log_8 2} = n^{\frac{1}{3}}$$

$$\sum_{i=0}^{L} a^i$$

Finite Geometric Series

If $a > 1$



The series multiplied by a

$(1 + a + a^2 + \cdots + a^L)a$

The series

$(1 + a + a^2 + \cdots + a^L)1$

The next term in the series

a^{L+1}

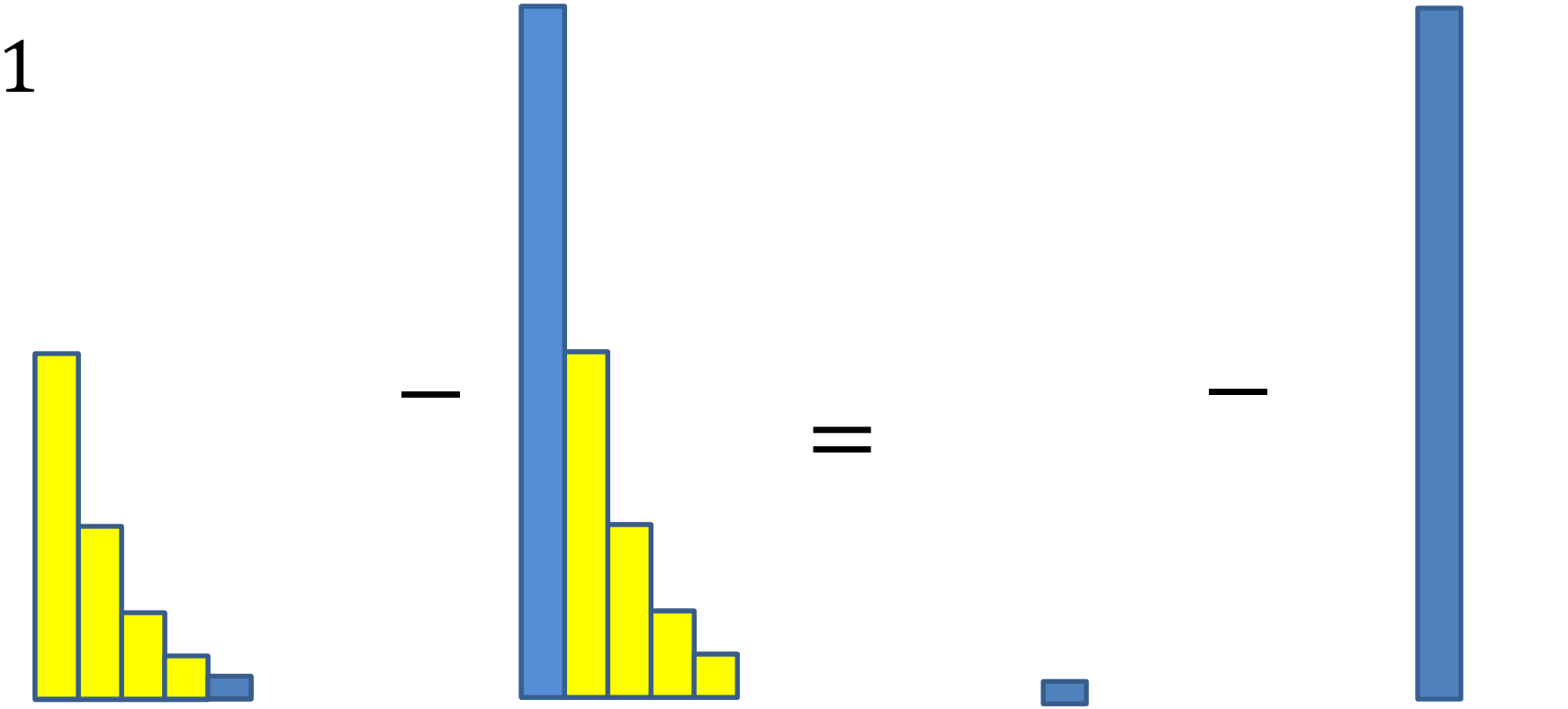
The first term

1

$$\sum_{i=0}^{L} a^i$$

Finite Geometric Series

If $a < 1$



The series multiplied by a

$(1 + a + a^2 + \cdots + a^L)a$

The series

$(1 + a + a^2 + \cdots + a^L)1$

The next term in the series

a^{L+1}

The first term

1

Solve for the series

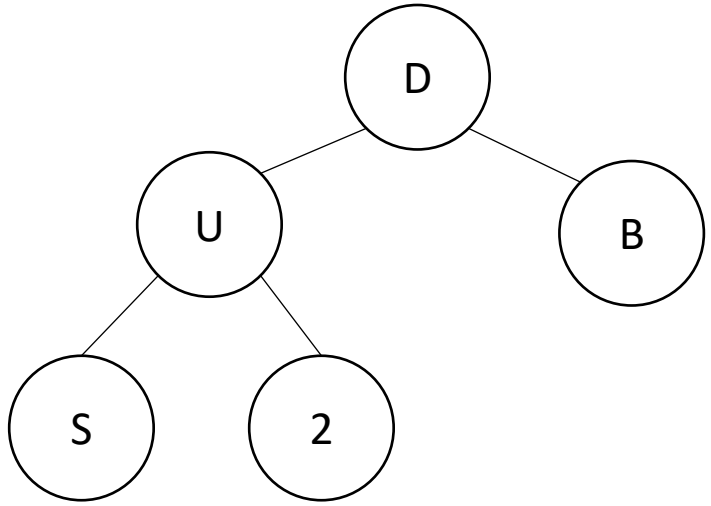
Dictionary (Map) ADT

- 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	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
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)	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$

More 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

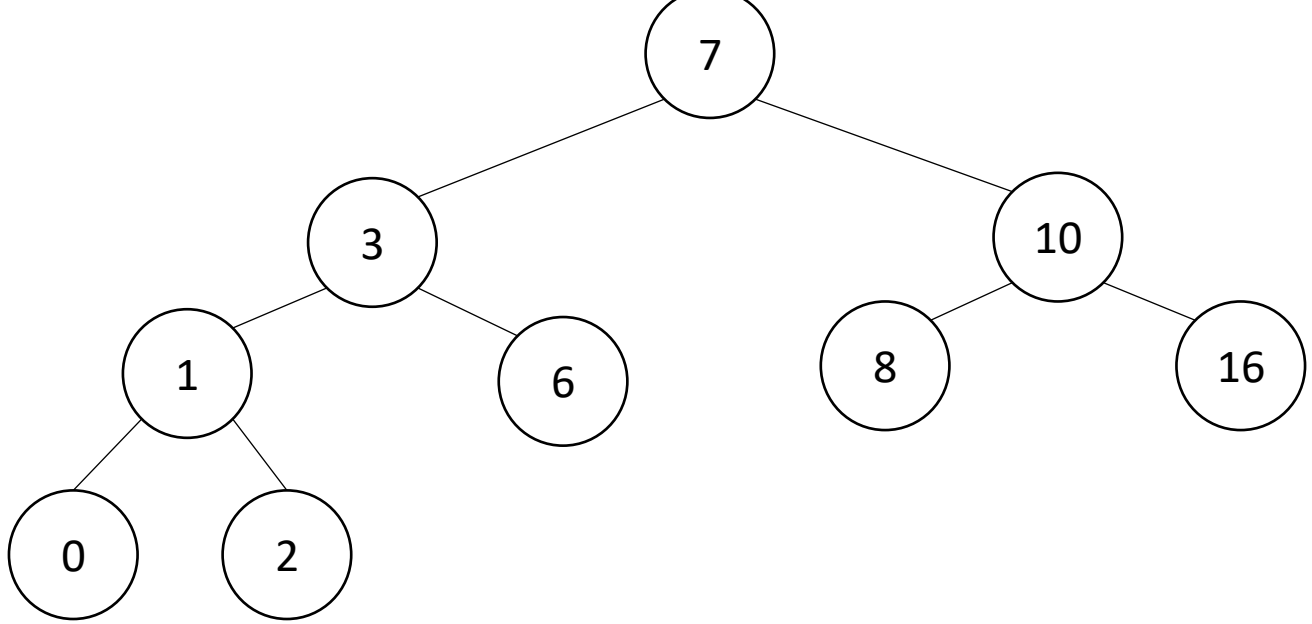
Name that Traversal!

```
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);
        }
}
```

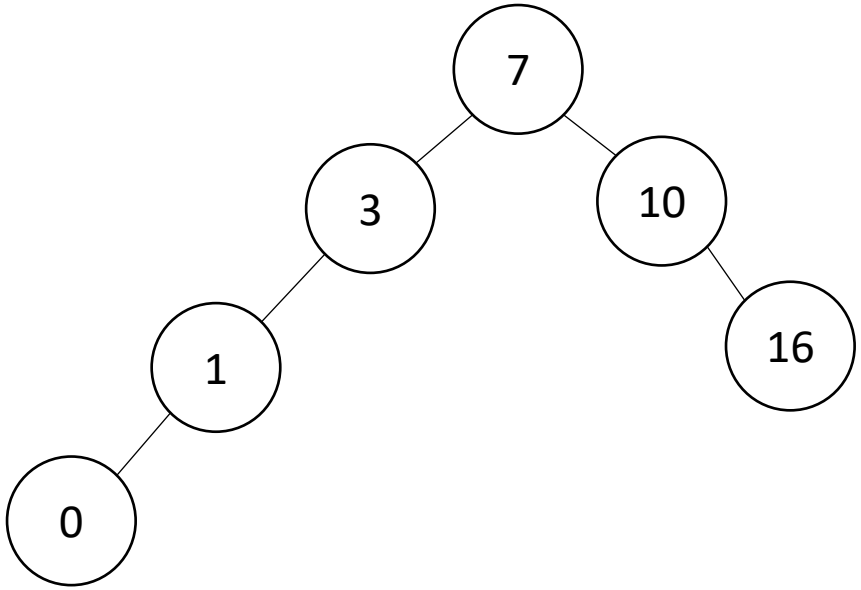
```
CorderTraversal(root){
        if (root.left != Null){
                process(root.left);
        }
        process(root)
        if (root.right != Null){
                process(root.right);
        }
}
```

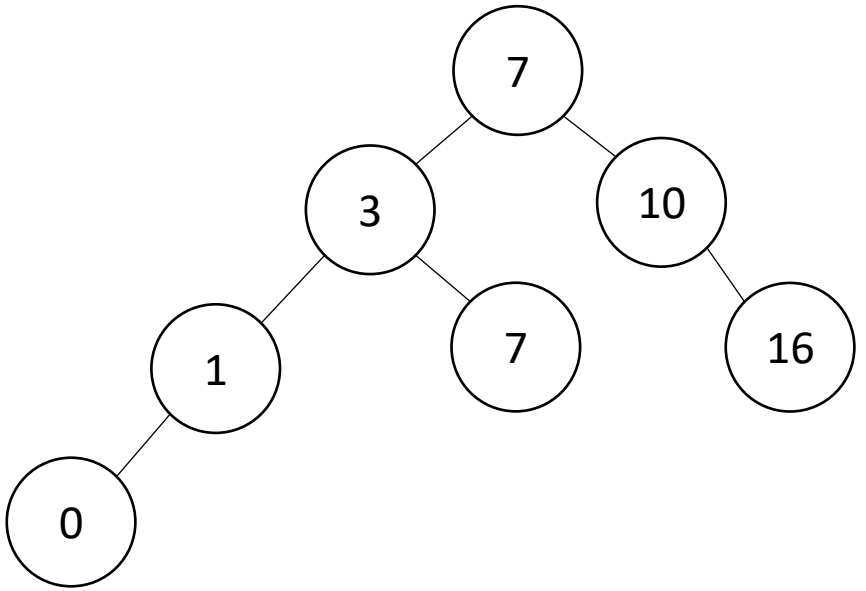
Binary Search Tree

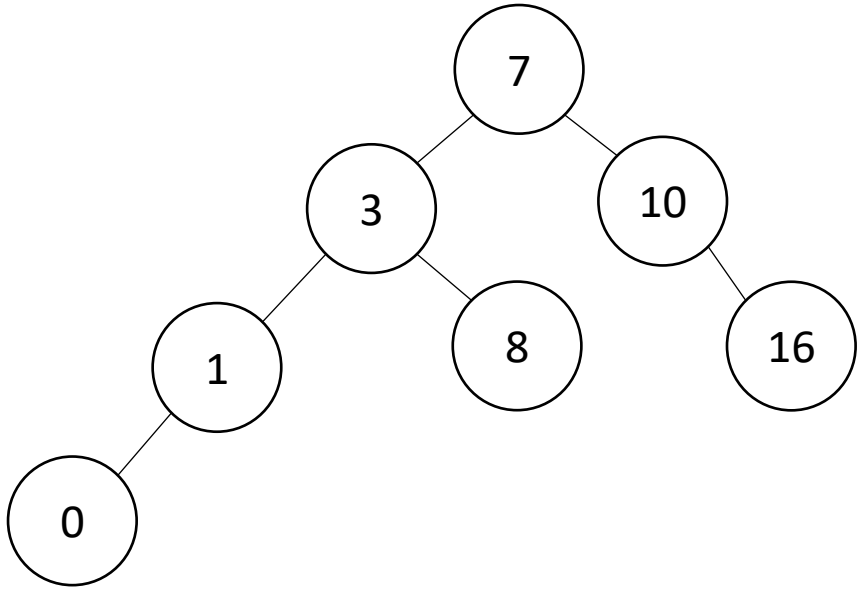


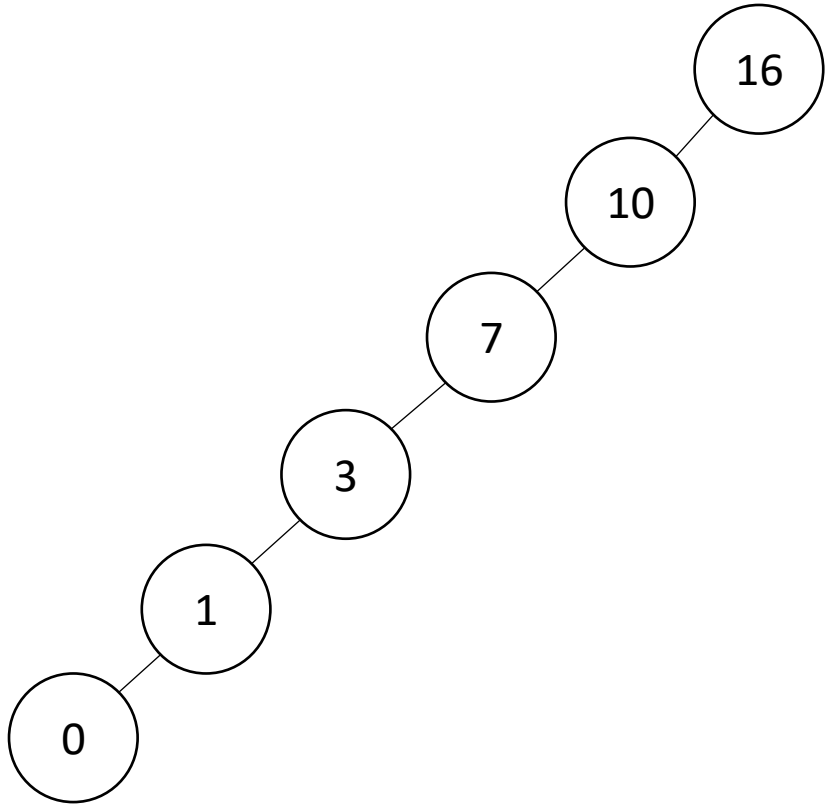
- Binary Tree
  - Definition:
    - Tree where each node has at most 2 children
- Order Property
  - All keys in the left subtree are smaller than the root
  - All keys in the right subtree are larger than the root
  - Consequence: cannot have repeated values

Are these BSTs?







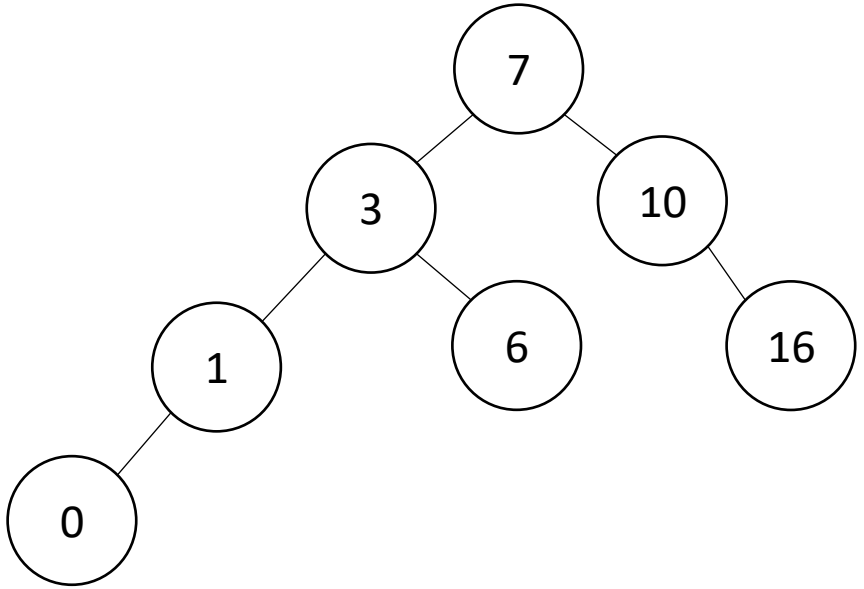


Aside: Why not use an array?

- 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

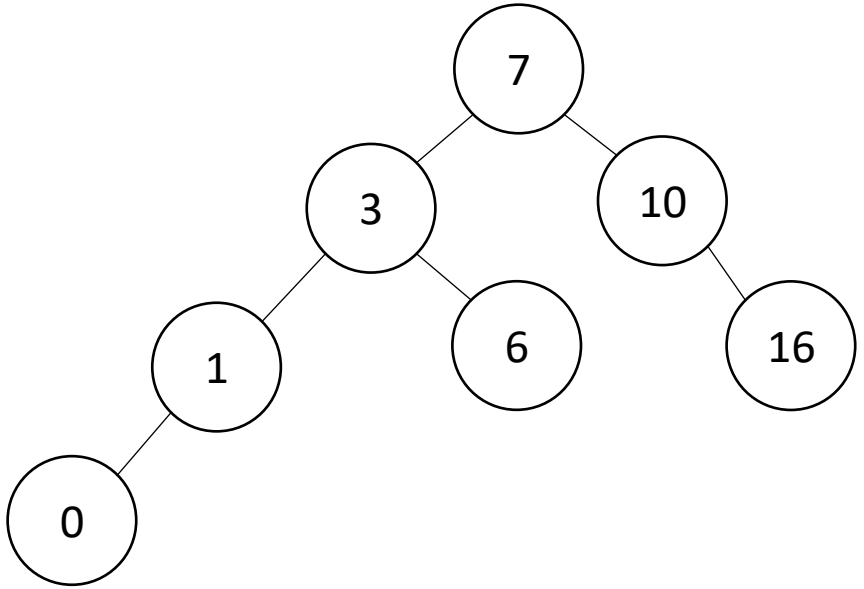
Find Operation (recursive)

```
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;
}
```



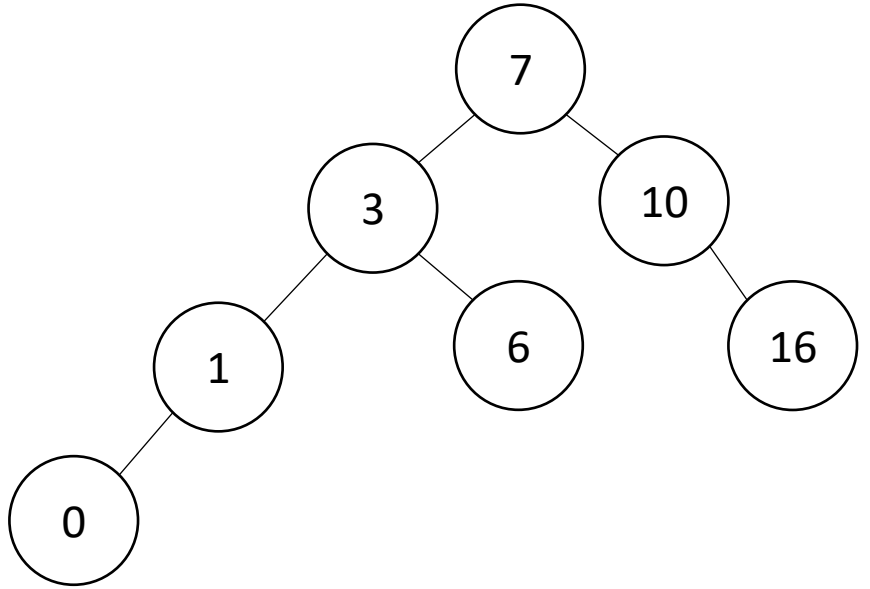
Find Operation (iterative)

```
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)

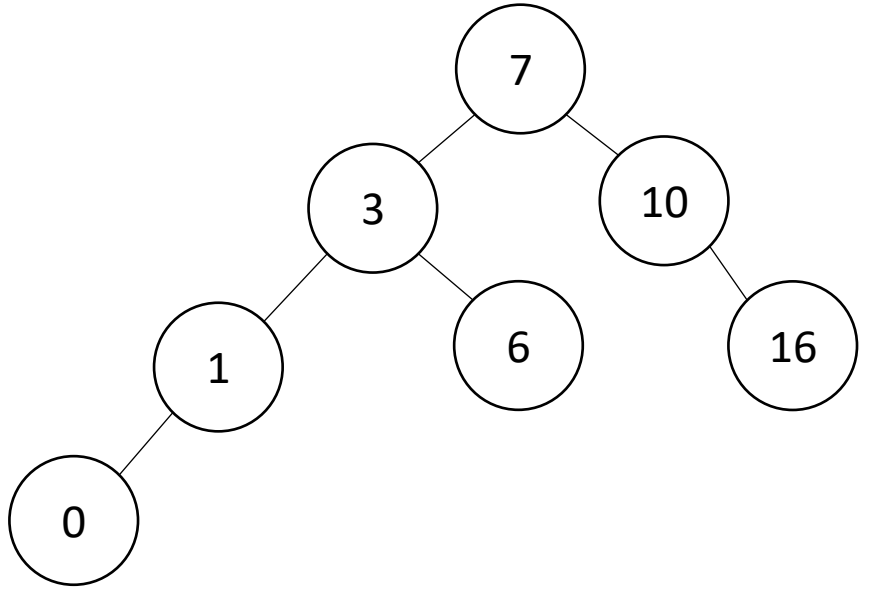
```
insert(key, value, root){
        root = insertHelper(key, value, root);
}
insertHelper(key, value, root){
        if(root == null)
                return new Node(key, value);
        if (root.key < key)
                root.right = insertHelper(key, value, root.right);
        else
           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



Note: Insert happens only at the leaves!

Insert Operation (iterative)

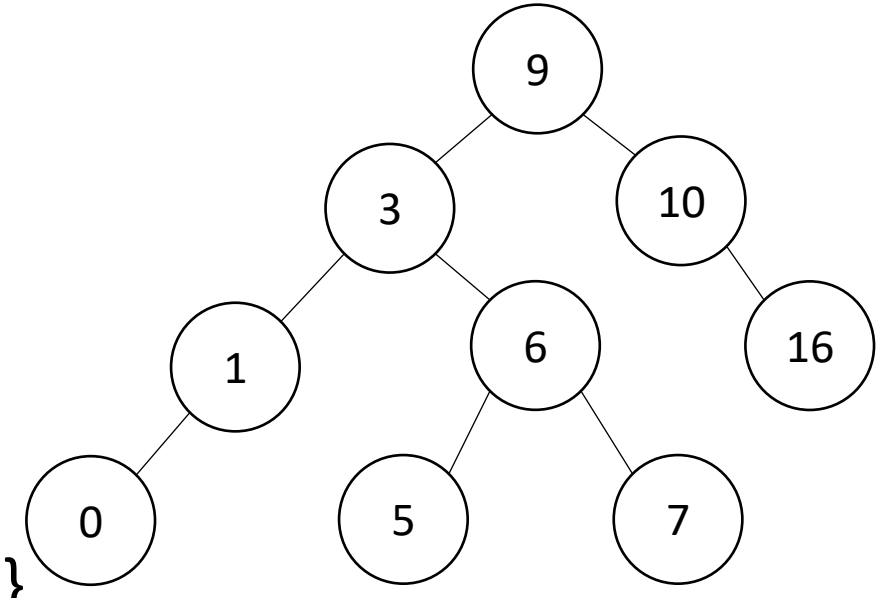
```
insert(key, value, root){
        if (root == Null){ this.root = new Node(key, value); }
        parent = Null;
        while (root != Null && key != root.key){
                parent = root;
                if (key < root.key){ root = root.left; }
                else if (key > root.key){ root =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



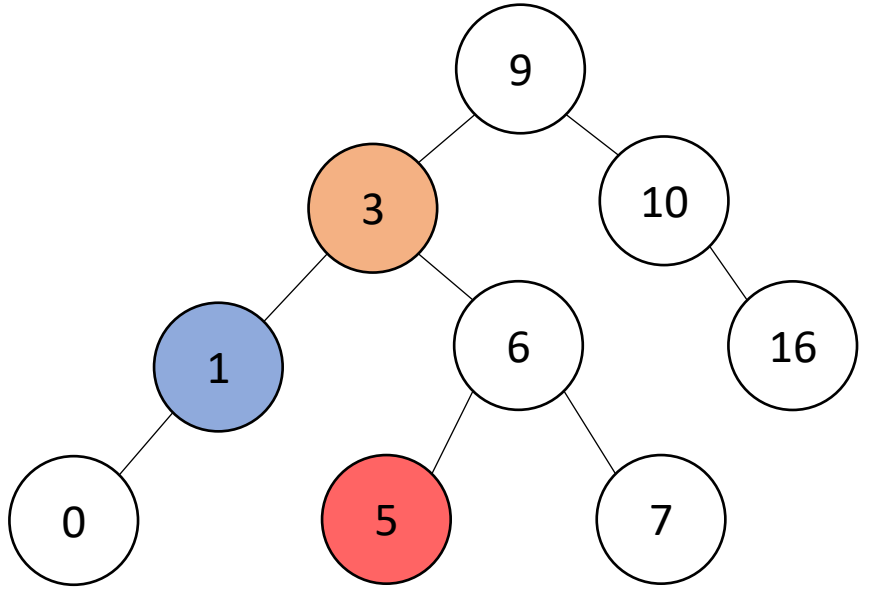
Note: Insert happens only at the leaves!

Delete Operation (iterative)

```
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

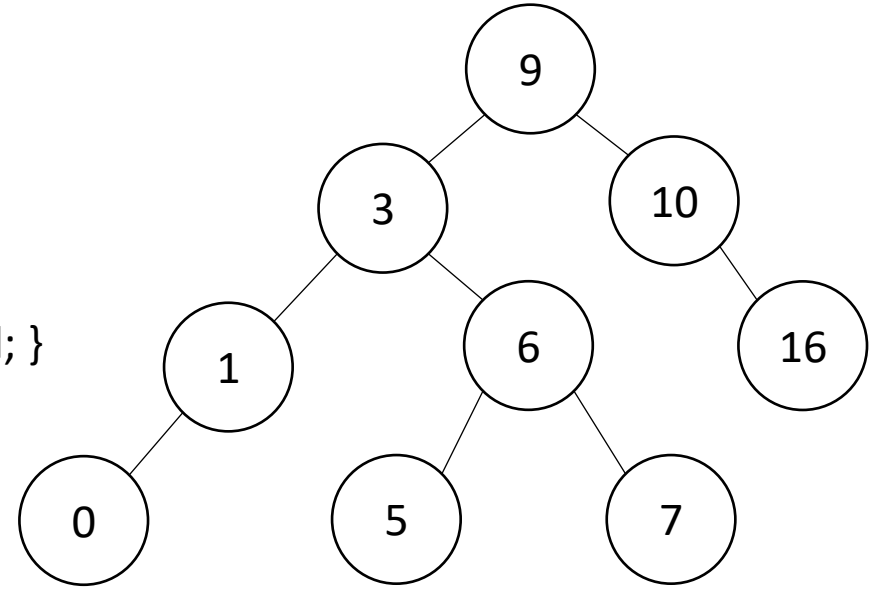


- 0 Children (i.e. it's a leaf)
- 1 Child
  - Replace the deleted node with its child
- 2 Children
  - Replace the deleted with the largest node to its left or else the smallest node to its right

Finding the Max and Min

- Max of a BST:
  - Right-most Thing

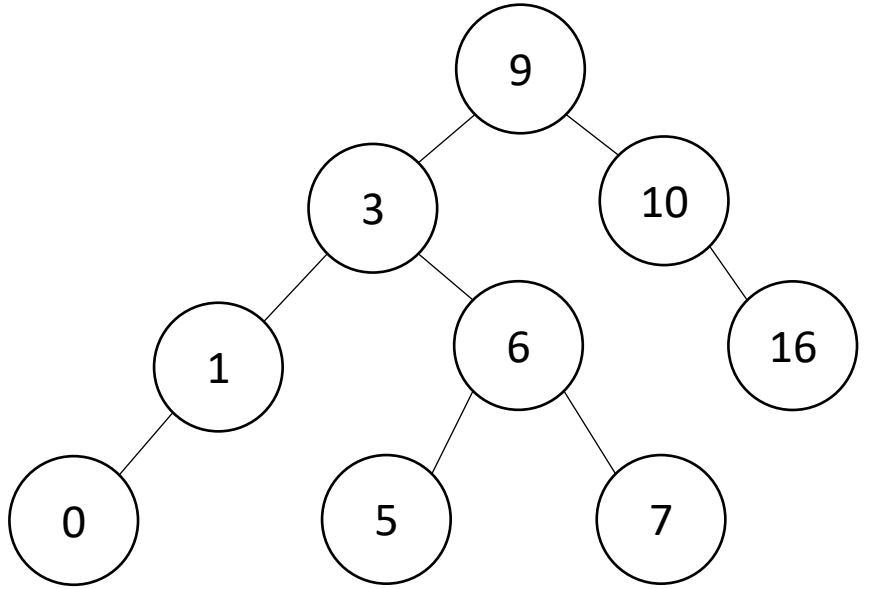
```
maxNode(root){
        if (root == Null){ return Null; }
        while (root.right != Null){
                root = root.right;
        }
        return root;
}
```



- Min of a BST:
  - Left-most Thing

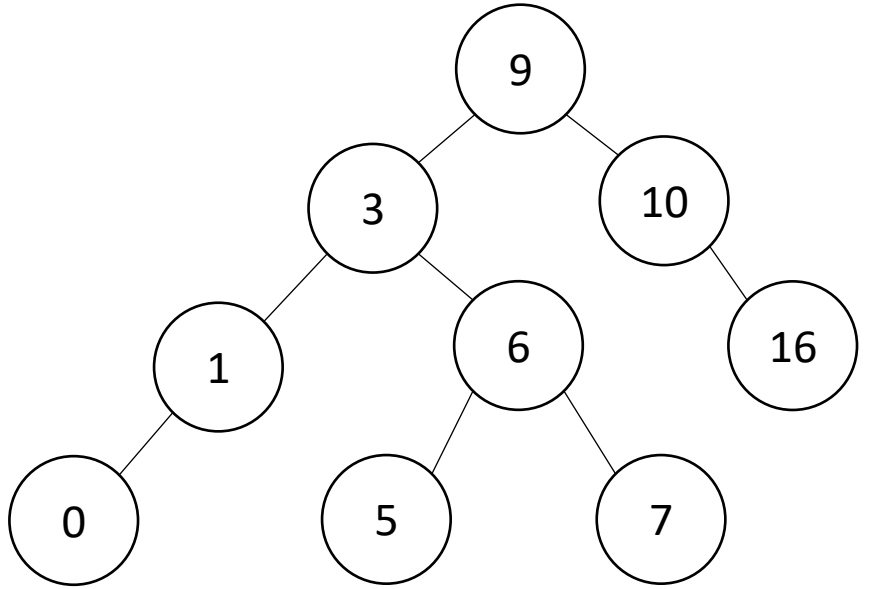
```
minNode(root){
        if (root == Null){ return Null; }
        while (root.left != Null){
                root = root.left;
        }
        return root;
}
```

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
        }
}
```

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); }
}
```