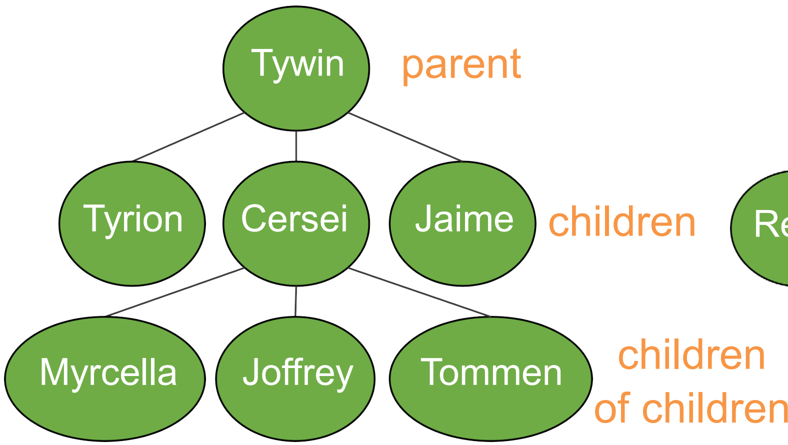
Lecture 8
Binary Search Tree

Department of Computer Science
Hofstra University

Lecture Goals

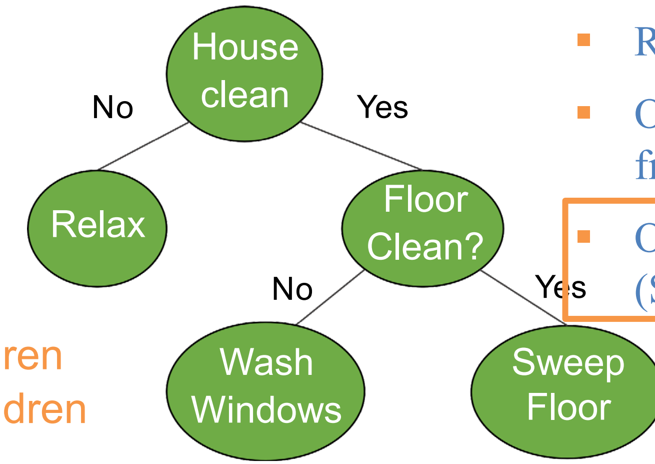
- Describe the value of trees and their data structure
- Explain the need to visit data in different orderings
- Perform pre-order, in-order, post-order and level-order traversals
- Define a Binary Search Tree
- Perform search, insert, delete in a Binary Search Tree
- Explain the running time performance to find an item in a BST

Different Trees in Computer Science



Family Trees

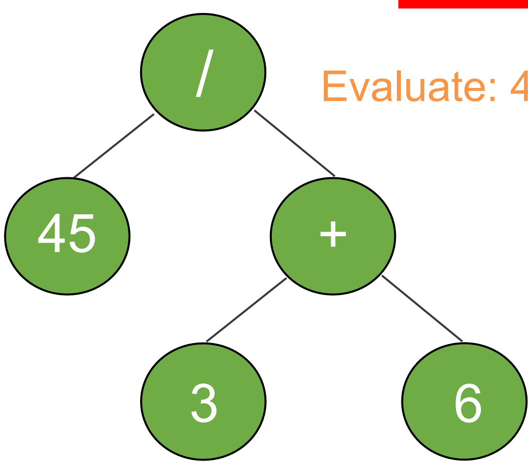
Why trees?



Decision Trees

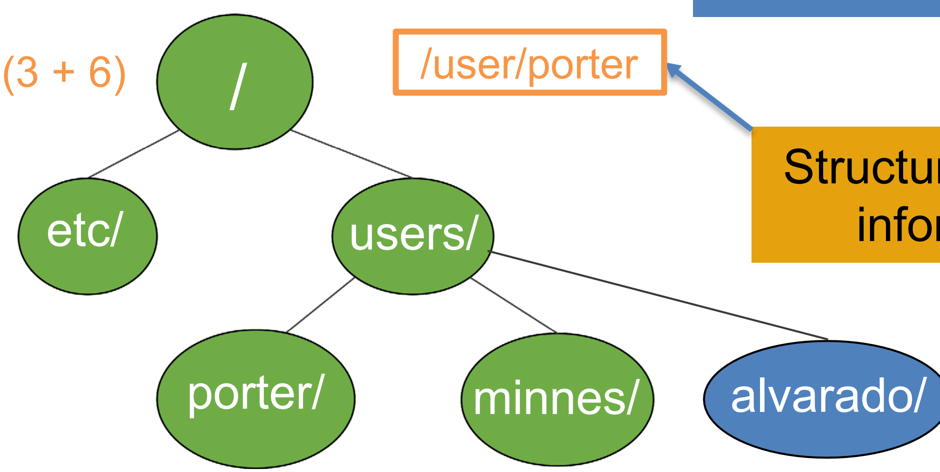
- Root is most important (Heap)
- Organized by character frequency (Huffman Tree)
- Organized by node ordering (Search Trees)
- Etc…

Different Organizations
→ Different Trees



Evaluate: 45 / (3 + 6)

Expression Trees

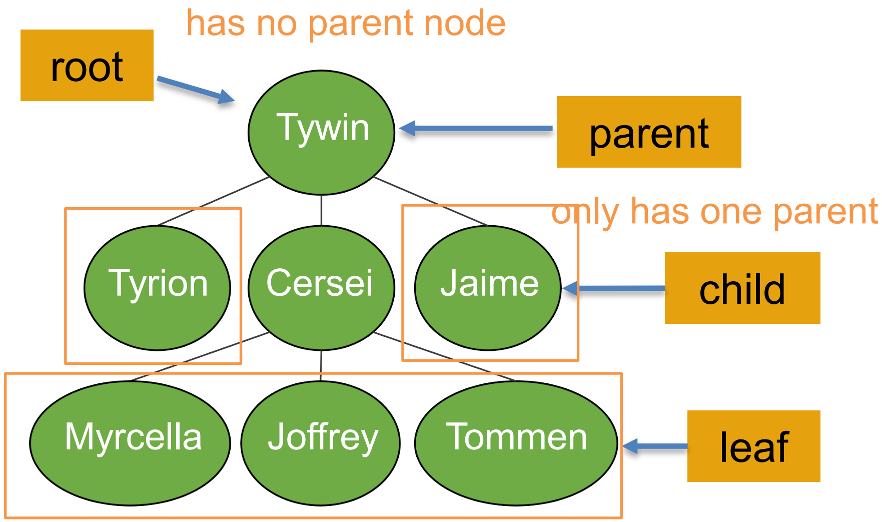


Structure conveys information

File System

Dynamic Data Structure

Defining Trees

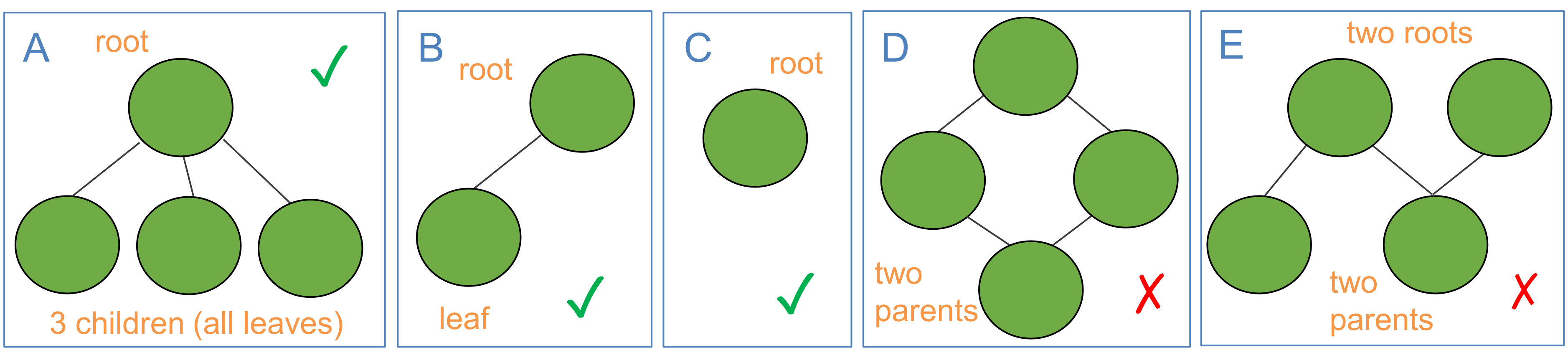


Family Trees

What defines a tree?

- Single root
- Each node can have only one parent (except for root)
- No cycles in a tree

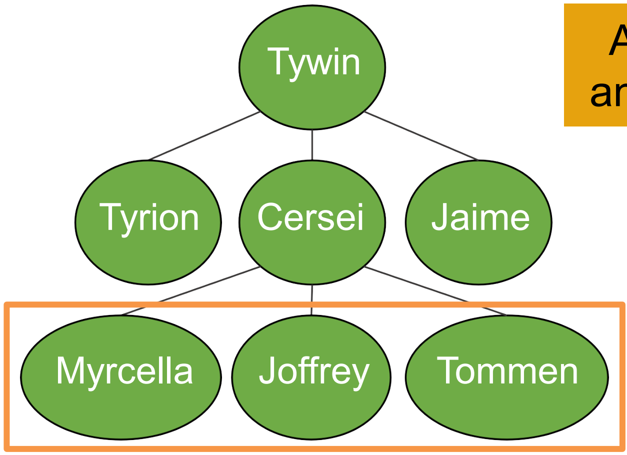
Which are trees?



Cycle: two different paths between a pair of nodes

Binary Trees

Generic Tree

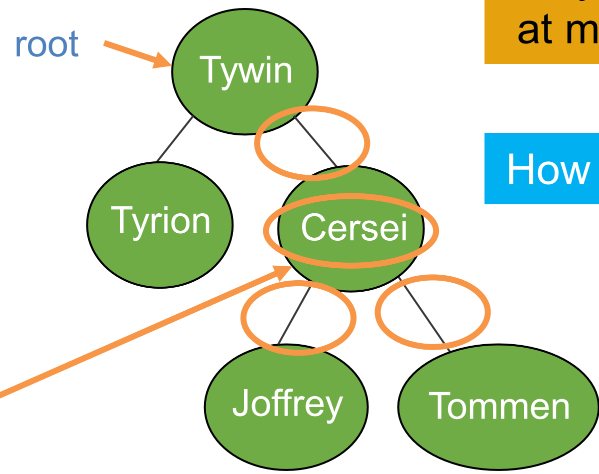


Any Parent can have any number of children

How would a general tree node differ?

A general tree would just have a list for children

Binary Tree



Any node can have at most two children

A tree just needs a root node

like the head and tail for linked list

How do we construct a tree?

Each node needs:

1. A value
2. A parent
3. A left child
4. A right child

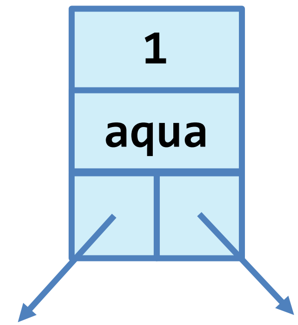
Like Linked Lists, Trees have a "Linked Structure"

nodes are connected by references

Tree Node

- Each node represents a key/value pair.

```
public class Node<K, V> {
    K key;
    V value;
    Node<K, V> left;
    Node<K, V> right;
}
```

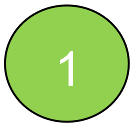


A node with key 1 and value “aqua”

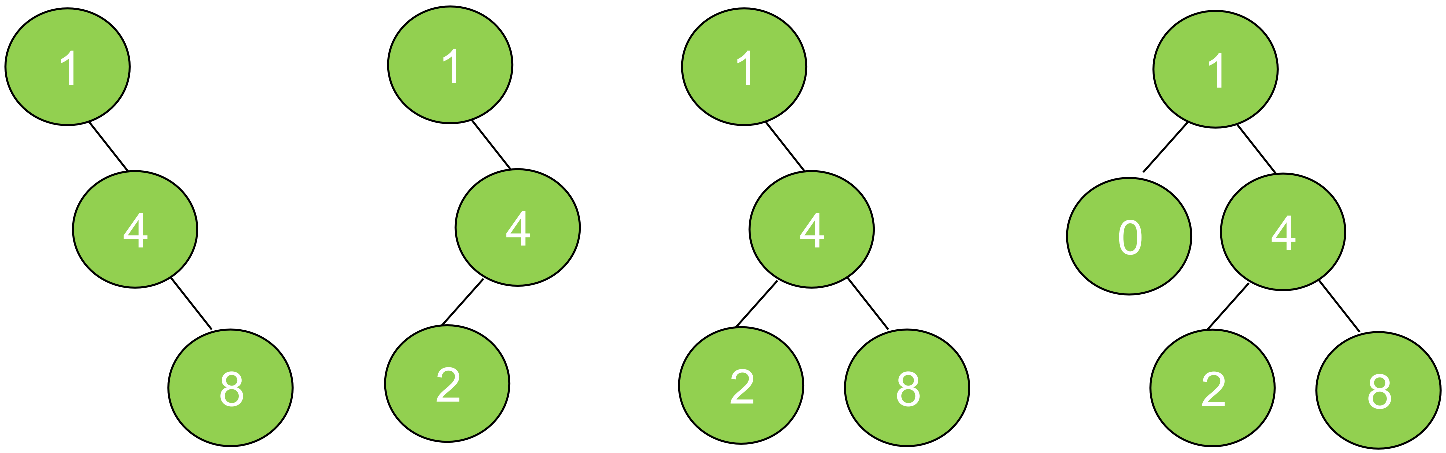
- For simplicity, we focus on keys and omit the values in the discussions
  - Keys determine where the nodes go

Definitions

- Root node: the single node with no parent at the top of the tree. Leaf node: a node with no children
- Subtree: a node and all it descendants
- Height of a tree: defined as the number of edges in the longest path from the root node to a leaf node.
  - A tree with only a root node has height of 0.

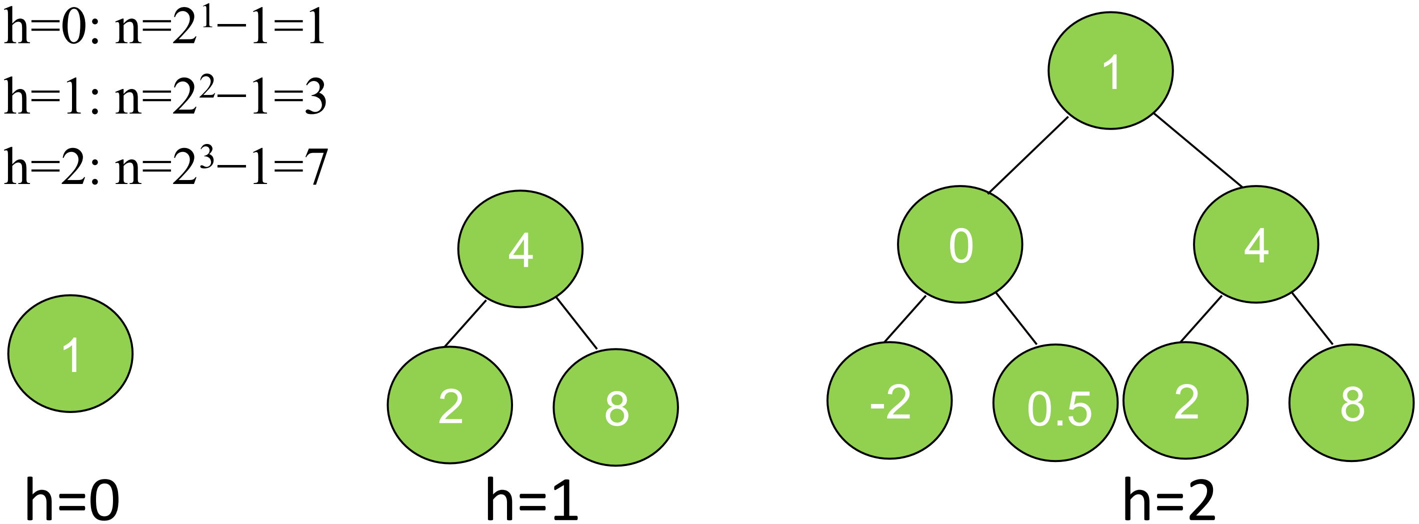


  - The trees below all have height of 2.



Full Binary Tree

- A full binary tree with height h has total number of leaves 2^h, and total number of nodes: $n = 2^{h+1}-1$
- In a full binary tree, each level is completely filled. The number of nodes at each level l is 2^l. Therefore, the total number of nodes is the sum of nodes at all levels from 0 to h, which is a geometric series: $n=1+2+4+\ldots+2^h=2^{h+1}-1$
- This means that for a full binary tree, the total number of nodes grows exponentially with the height of the tree
  - $h=0$: $n=2^1-1=1$
  - $h=1$: $n=2^2-1=3$
  - $h=2$: $n=2^3-1=7$



Height of a Binary Tree

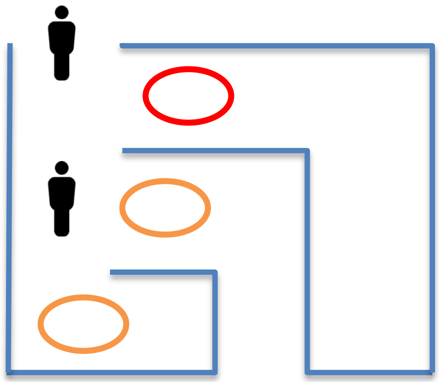
- For a binary tree with n nodes, the height h is bounded by: $\lceil \log_2(n+1) \rceil - 1 \leq h \leq n - 1$
  - The lower bound represents a perfectly balanced tree, and the upper bound represents a degenerate tree (essentially a linked list).
  - The minimum height of a binary tree with n nodes is $\lceil \log_2(n+1) \rceil - 1$, which occurs in the most balanced configuration, where $\lceil\rceil$ is the ceiling operator, e.g., $\lceil 1.0 \rceil = 1$, $\lceil 1.3 \rceil = 2$.
  - The maximum height of a binary tree with n nodes is n-1, which occurs in the case of a skewed tree (a linear chain or linked list).

Tree Traversal - Motivation

Warning: These first examples are really graphs. We'll visit graphs in detail in the next course. Here they are used as motivating examples

start

Strategy: go until hit a dead end, then retrace steps and try again



finish

Maze Traversal

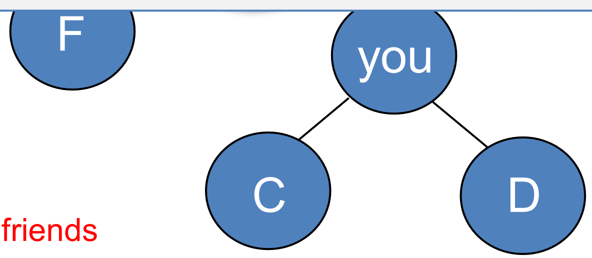
Imagine this is a hedge maze

What's my next step?

Mazes benefit from "Depth First Traversals"

Bottom line: Order we visit matters and we'll make choices based on our needs

Suppose you have a list of your friends and each of your friends have lists



How closely are you connected with D?

What's my next step?

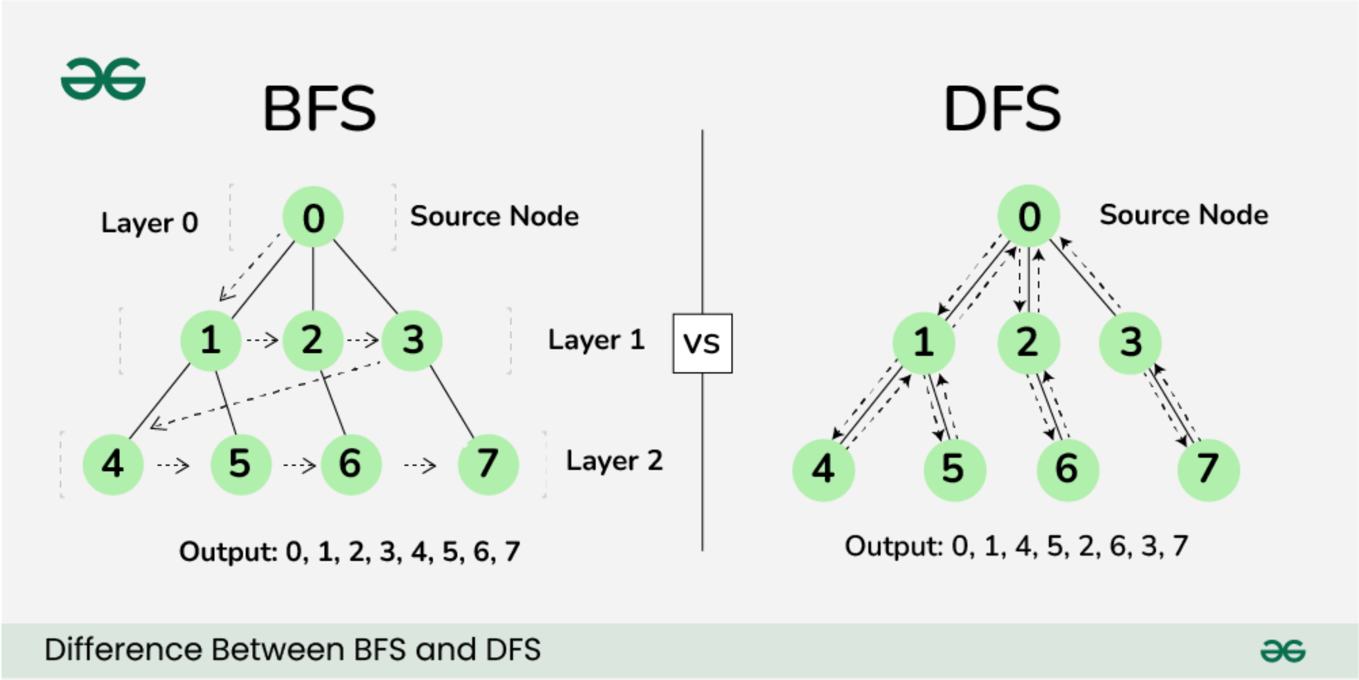
Strategy: look at all of your friends first, and then branch out.

This problem benefits from "Breadth First Traversals"

Social Network

BFS vs. DFS

- Breadth-First Search (BFS) and Depth-First Search (DFS) are two fundamental algorithms used for traversing or searching graphs and trees
  - BFS traversal explores all the neighboring nodes at the present depth prior to moving on to the nodes at the next depth level.
  - DFS uses backtracking. The deepest node is visited and then backtracks to its parent node if no sibling of that node exists

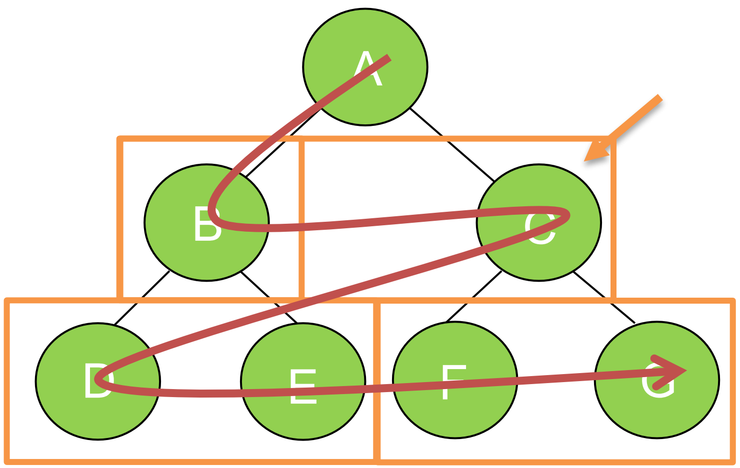


Breadth First Search (BFS) Animations
https://www.youtube.com/watch?v=QUfEOCOEKkc
Depth First Search (DFS) Animations
https://www.youtube.com/watch?v=3_NMDJkmvLo

Traversal Order for Binary Trees

- Breadth First Traversal with BFS
  - Level Order Traversal
- Depth First Traversals with DFS
  - Pre-order Traversal (Root-Left-Right)
  - In-order Traversal (Left-Root-Right)
  - Post-order Traversal (Left-Right-Root)

Graph Traversal with BFS: Level-order Traversal (Contd.)



Visit:

A B C D E F G

Challenging: When we finish B, how do we go to C next?

Idea: Keep a list and keep adding to it and removing from start.

Visit: A B C D E F G

List: ~~A~~ ~~B~~ ~~C~~ ~~D~~ ~~E~~ ~~F~~ ~~G~~

We used this list like a "Queue"

- Add to the end
- Remove from the front
- First-In, First-Out (FIFO)

Summary: Nested | Field | Constr | Method   Detail: Field | Constr | Method

java.util

Interface Queue<E>

	Throws exception
Insert	`add(e)`
Remove	`remove()`
Examine	`element()`

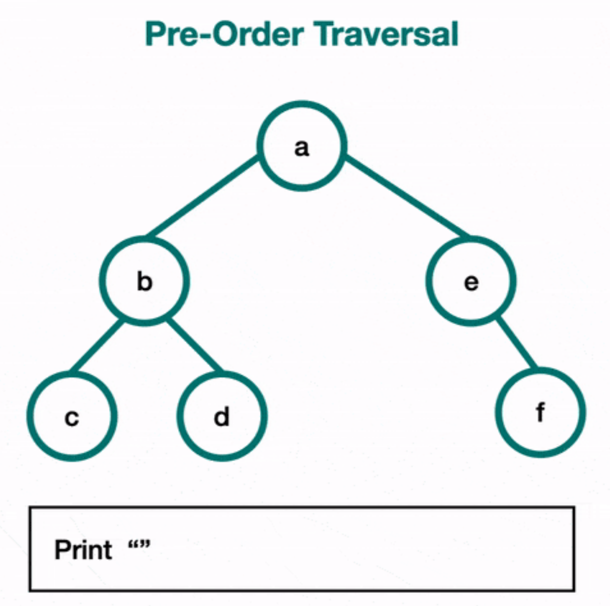
look at the first element

Tree traversals with DFS: pre-order, in-order, post-order

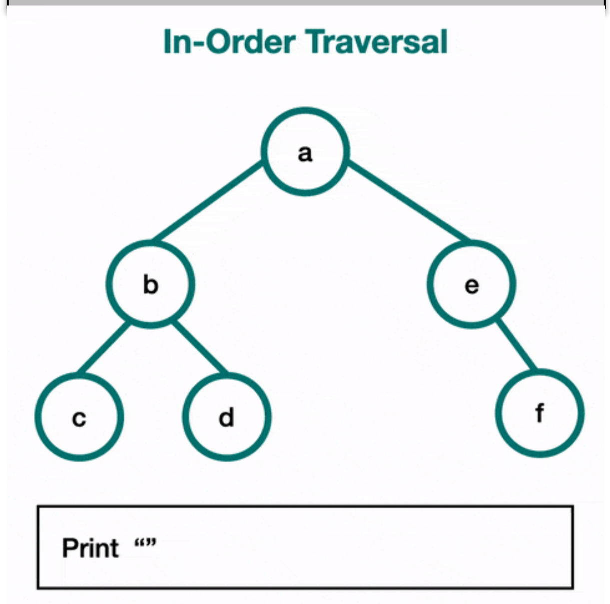
```
function preOrderTraversal(node) {
  if (node !== null) {
    visitNode(node);
    preOrderTraversal(node.left);
    preOrderTraversal(node.right);
  }
}
```

```
function inOrderTraversal(node) {
  if (node !== null) {
    inOrderTraversal(node.left);
    visitNode(node);
    inOrderTraversal(node.right);
  }
}
```

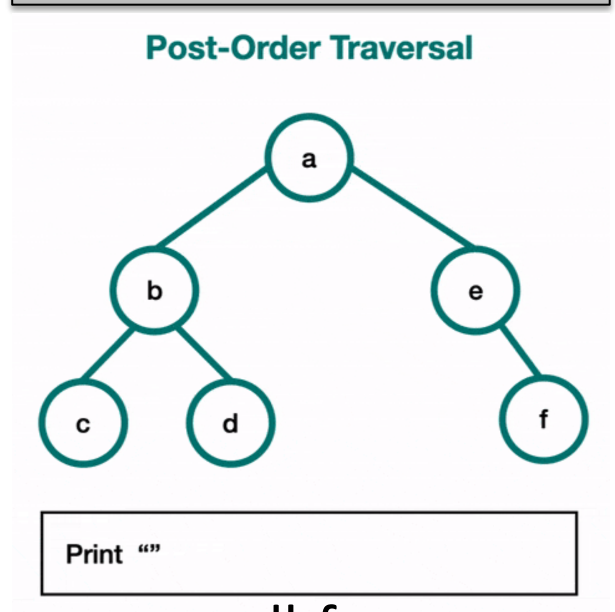
```
function postOrderTraversal(node) {
  if (node !== null) {
    postOrderTraversal(node.left);
    postOrderTraversal(node.right);
    visitNode(node);
  }
}
```



abcdef



cbdaef



cdbfea

Preorder Traversal in Binary Tree Animations
https://www.youtube.com/watch?v=gLx7Px7IEzg

Inorder Traversal in Binary Tree Animations
https://www.youtube.com/watch?v=ne5oOmYdWGw

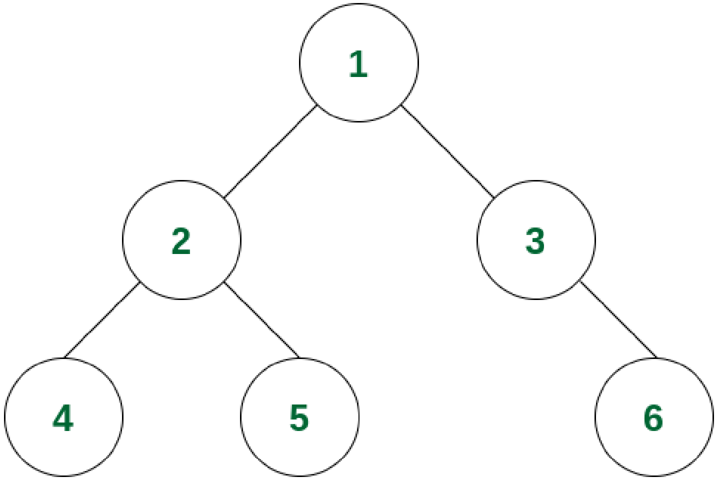
Postorder Traversal in Binary Tree Animations
https://www.youtube.com/watch?v=a8kmbuNm8Uo

Summary of Tree Traversals with DFS

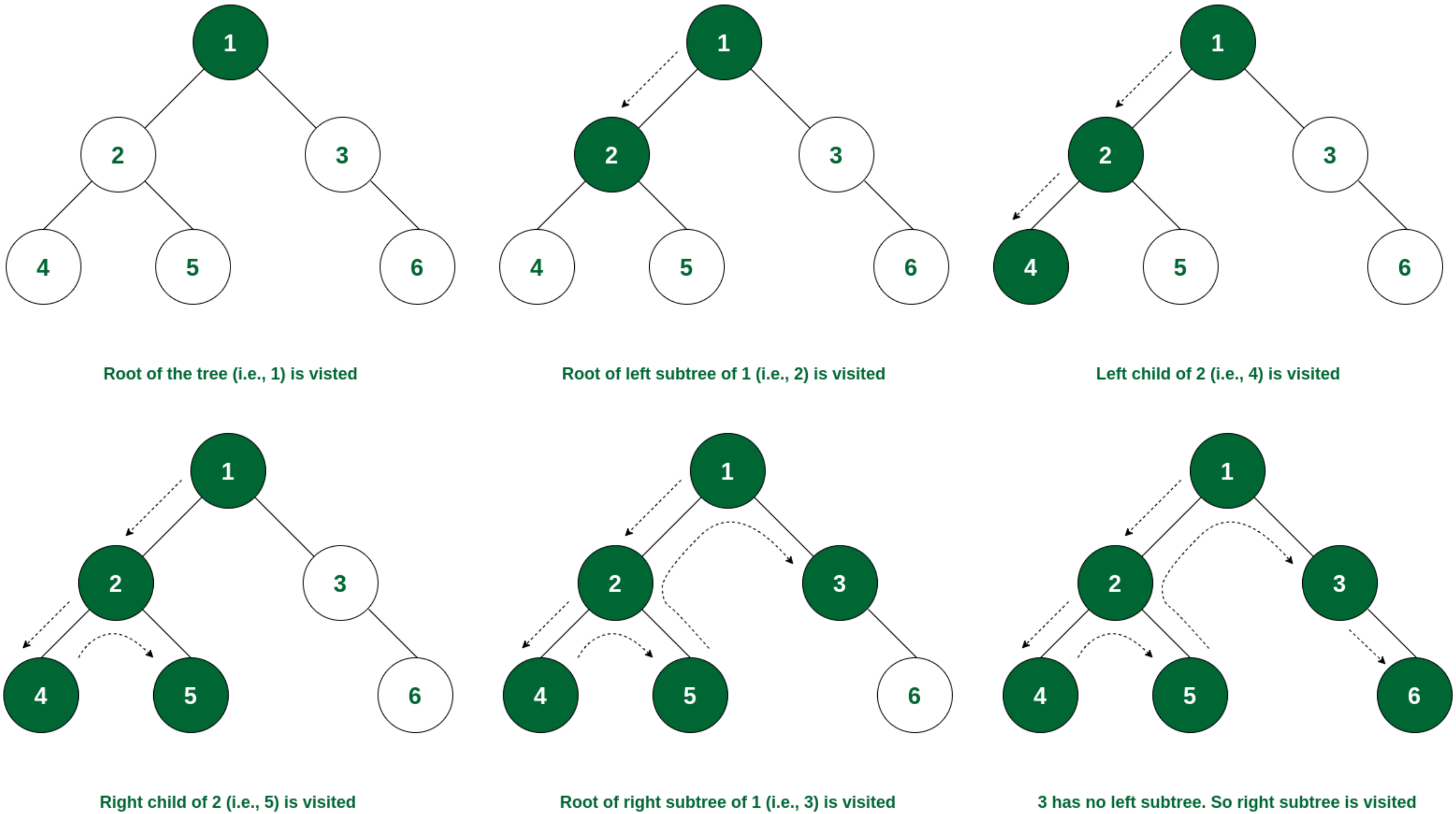
- **Pre-order traversal**:
  1) Visit the node itself.
  2) Traverse the left subtree.
  3) Traverse the right subtree.
  - Begins at the root, ends at the right-most node.
- **In-order traversal**:
  1) Traverse the left subtree.
  2) Visit the node itself.
  3) Traverse the right subtree.
  - Begins at the left-most node, ends at the rightmost node.
- **Post-order traversal**:
  1) Traverse the left subtree.
  2) Traverse the right subtree.
  3) Visit the node itself.
  - Begins with the left-most node, ends with the root.

Geeks for Geeks Tutorials

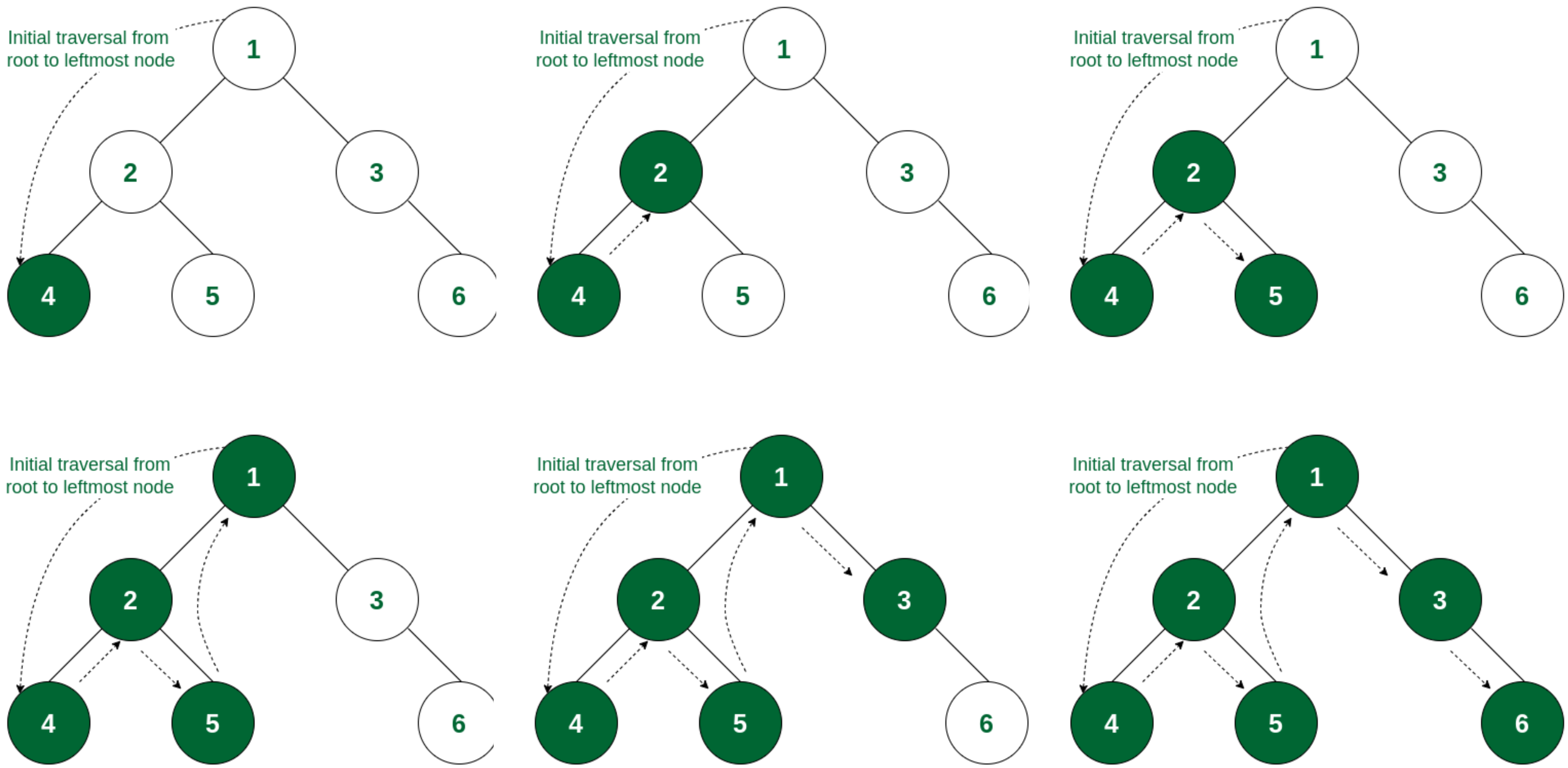
- https://www.geeksforgeeks.org/bfs-vs-dfs-binary-tree/
- https://www.geeksforgeeks.org/preorder-traversal-of-binary-tree/
- https://www.geeksforgeeks.org/inorder-traversal-of-binary-tree/
- https://www.geeksforgeeks.org/postorder-traversal-of-binary-tree/
- Running Example



Pre-order traversal of nodes is 1 -> 2 -> 4 -> 5 -> 3 -> 6



In-order traversal of nodes is 4 -> 2 -> 5 -> 1 -> 3 -> 6.

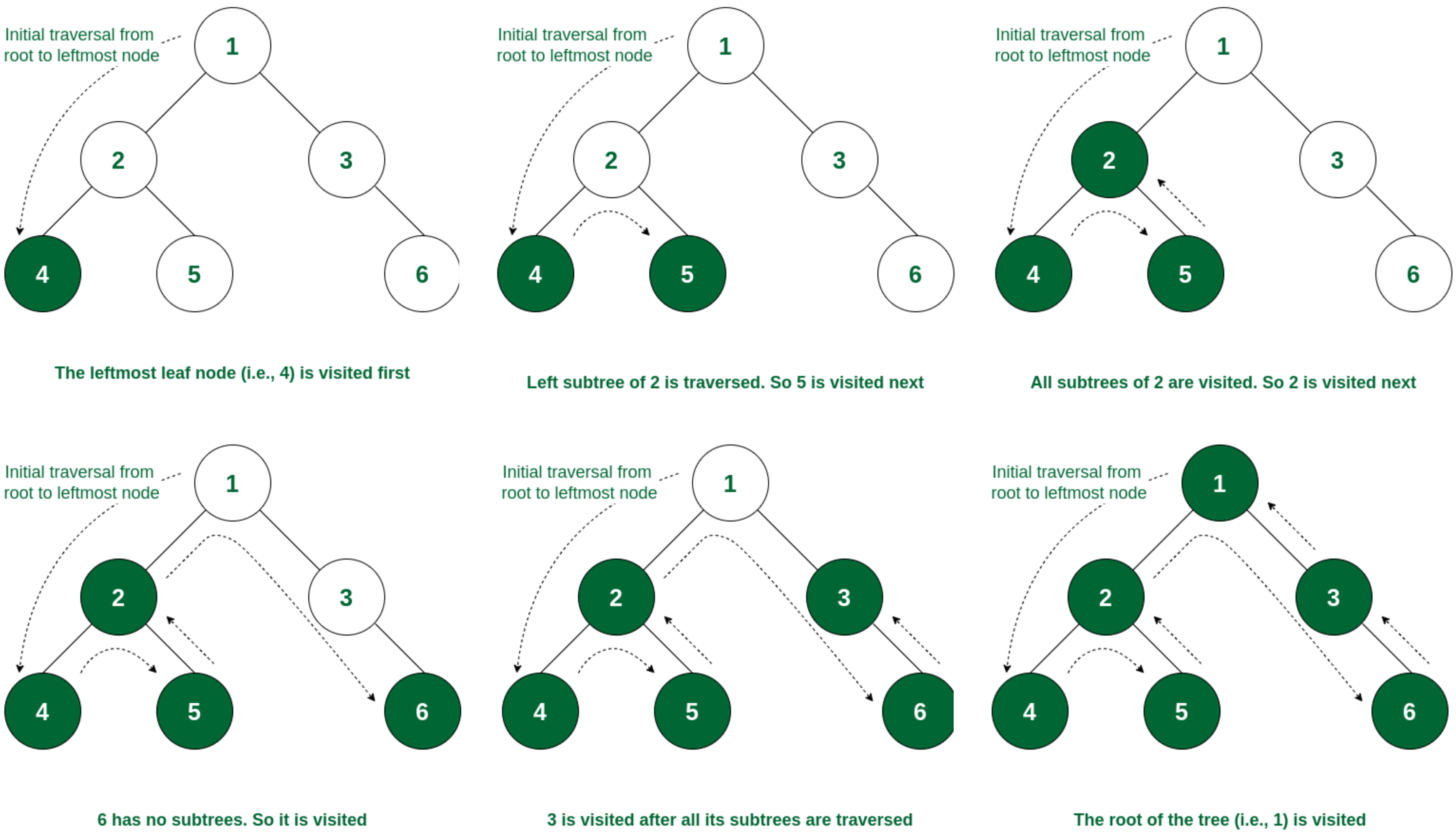


Left subtree of 1 is fully traversed. So 1 is visited next

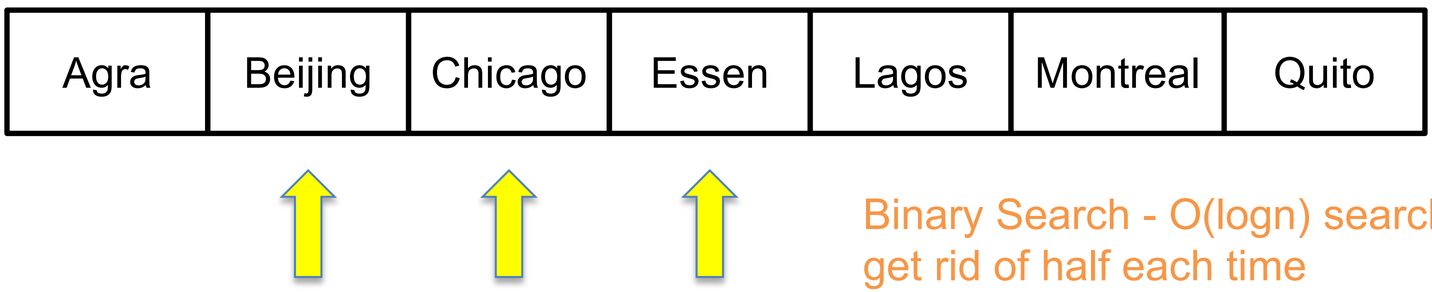
3 has no left subtree, so it is visited

Right Child of 3 is visited

Post-order traversal of nodes is 4 -> 5 -> 2 -> 6 -> 3 -> 1



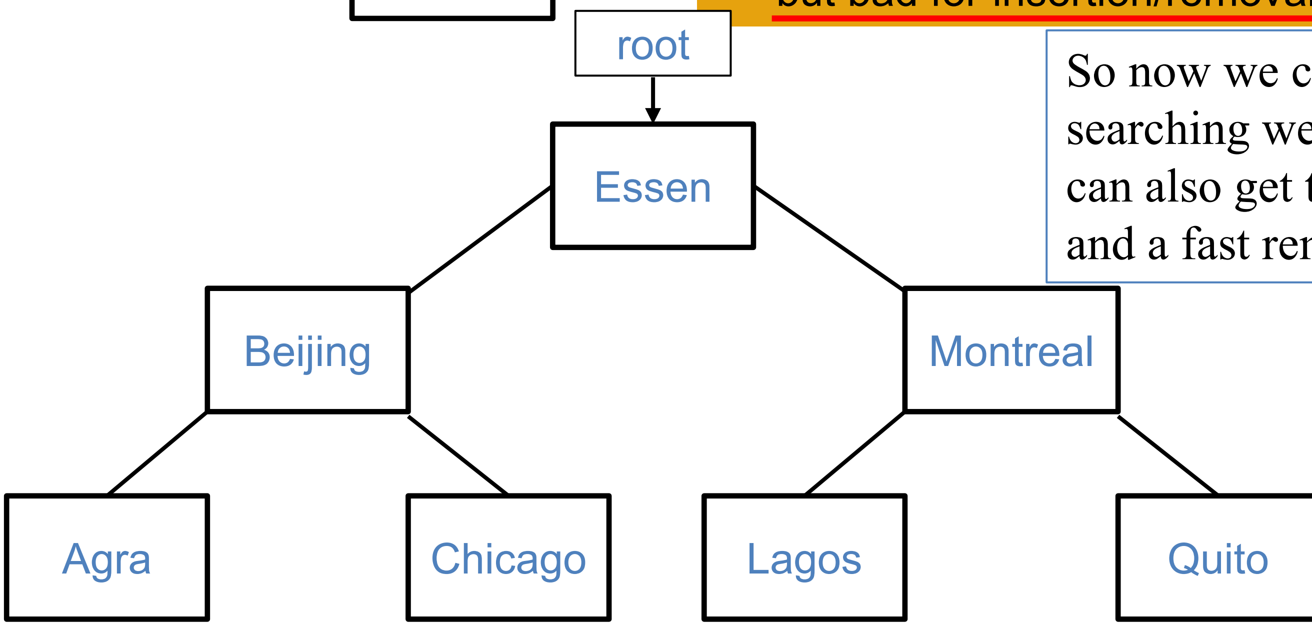
Motivation for Binary Search Tree



Binary Search - O(logn) search:
get rid of half each time

`toFind` Chicago

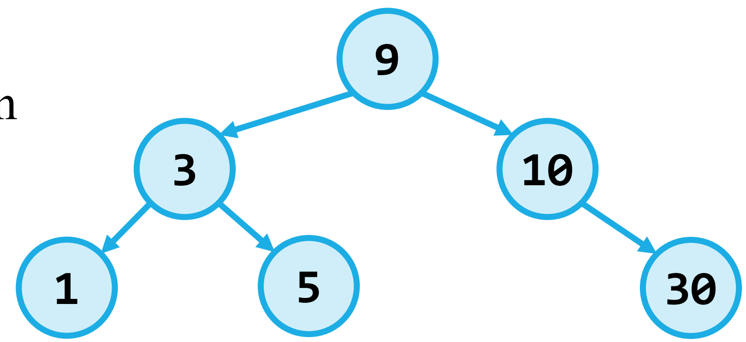
Sorted arrays are good for search,
but bad for insertion/removal

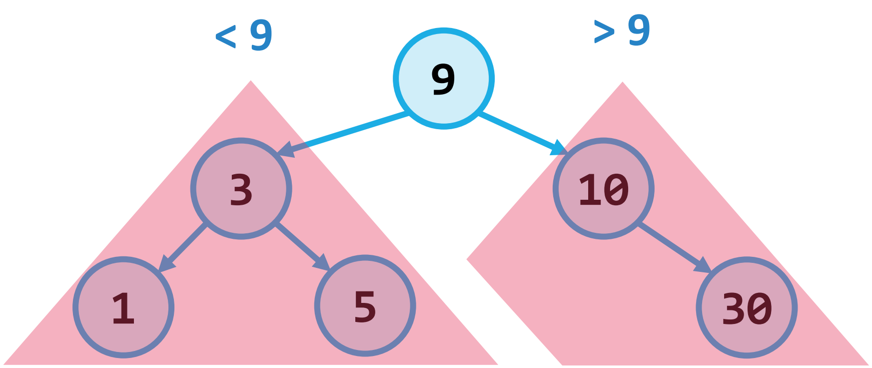


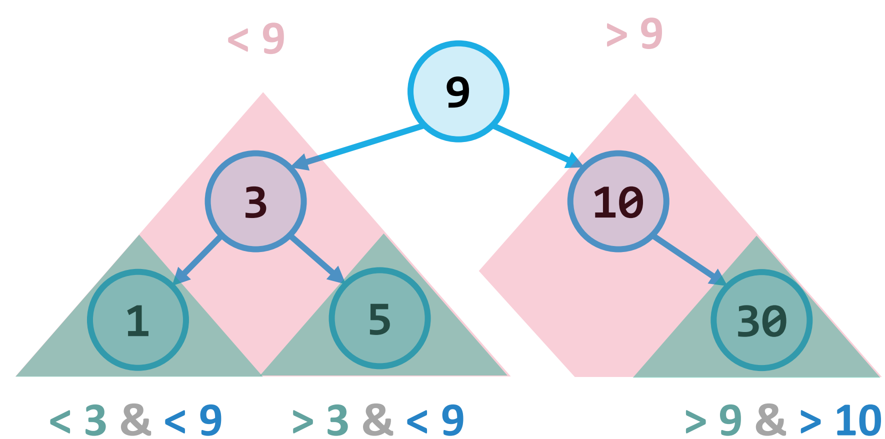
So now we can do the same kind of fast searching we did within an array, but we can also get the benefit of a fast insert and a fast removal that a tree provides.

Binary Search Tree (BST)

- A BST is an ordered, or sorted, binary tree, with the following invariants:
- For every node with key k:
  - The left subtree has only keys smaller than k
  - The right subtree has only keys greater than k
  - This invariant applies recursively throughout tree







Binary Search Tree Animations | Data Structure | Visual How
https://www.youtube.com/watch?v=ymGjUOiR8Jg

Searching for a Key: Binary Tree vs. Binary Search Tree

```
public boolean containsKeyBT(node,
key) {
    if (node == null) {
        return false;
    } else if (node.key == key) {
        return true;
    } else {
        return
          containsKeyBT(node.left)||
          containsKeyBT(node.right);
    }
}
```

* explores left, if not found then explores right

```
public boolean containsKeyBST(node, key) {
    if (node == null) {
        return false;
    } else if (node.key == key) {
        return true;
    } else {
        if (key <= node.key) {
            return containsKeyBST(node.left);
        } else {
            return containsKeyBST(node.right);
        }
    }
}
```

* explores either left or right at each level

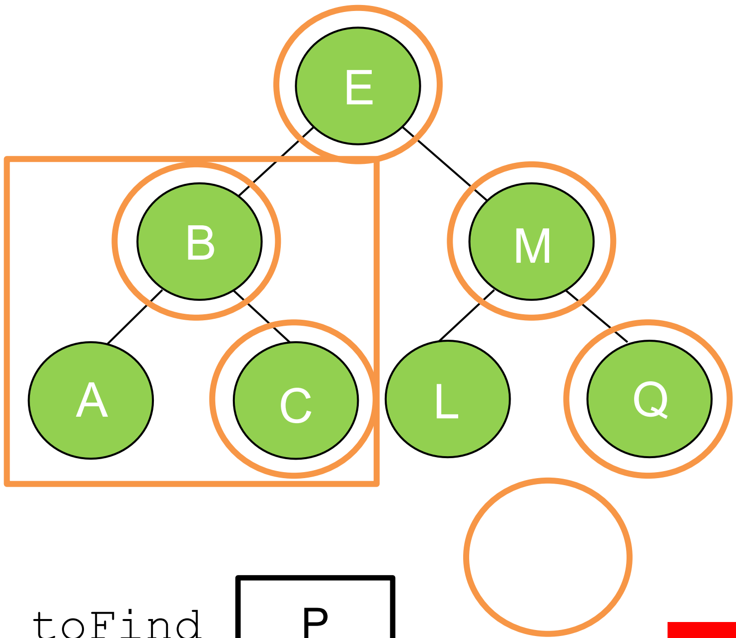
Best Case:
- finds value at overallRoot (random value)

Worst Case:
- doesn't find value, has to check every node

Best Case:
- finds value at overallRoot (middle value)

Worst Case:
- doesn't find value, has to check one path

Searching a BST



Same fundamental idea as binary search of an array

toFind **C**

```
Compare: E and C
Compare: B and C
Compare: C and C
```

Found it!

toFind **P**

```
Compare: E and P
Compare: M and P
Compare: Q and P
Node is null
```

Not Found!

How to implement this?

You could solve this with recursion.

You could also solve it with iteration by keeping track of your current node.

Searching a BST Iteratively

```
public class BinaryTree<E> {        <E extends Comparable<? super E>> {
    TreeNode<E> root;
    public boolean search(E toSearch) {
        TreeNode<E> curr = root;
        while (curr != null) {
            int comp = toSearch.compareTo(curr.getValue());
                if (comp < 0)
                    curr = curr.getLeftChild();
                else if (comp > 0)
                    curr = curr.getRightChild();
                else // comp = 0
                    return true;
        }
        return false;
    }
}
```

It means that either the class E itself or one of its super classes implements Comparable

Doesn't work with objects

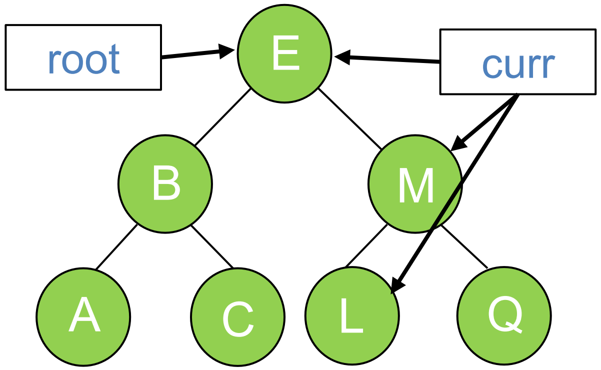
Do NOT change root pointer!

We need to do this over and over if not found

if calling object is less than parameter, compareTo returns a value < 0

if calling object is greater than parameter, compareTo returns a value > 0

Are we done?



`t.search('L')`

if calling object is equal to parameter, compareTo returns 0

Traverse down tree until:

a) end is reached
b) element is found

Searching a BST Recursively

```
public class BinaryTree<E extends Comparable<? super E>> {
   TreeNode<E> root;

      private boolean search(TreeNode<E> p, E toSearch) {
            if (p == null)
                  return false;
            int comp = toSearch.compareTo(p.getValue());
            if (comp == 0)
                  return true;
            else if (comp < 0)
                  return search(p.left, toSearch);
            else // comp > 0
                  return search(p.right, toSearch);
      }
      public boolean search(E toSearch) {
            return search(root, toSearch);
      }
}
```

Root of the tree we look at

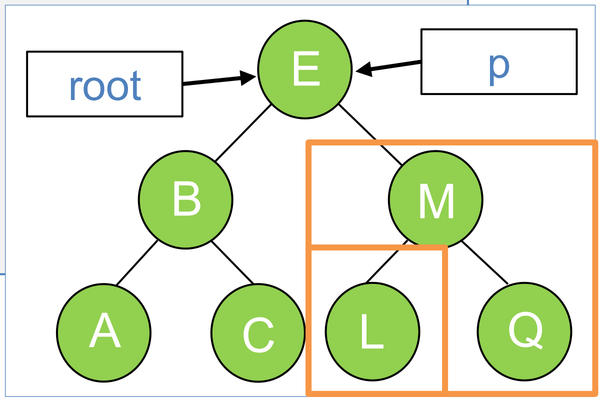
Tree is empty

Found it!

look left

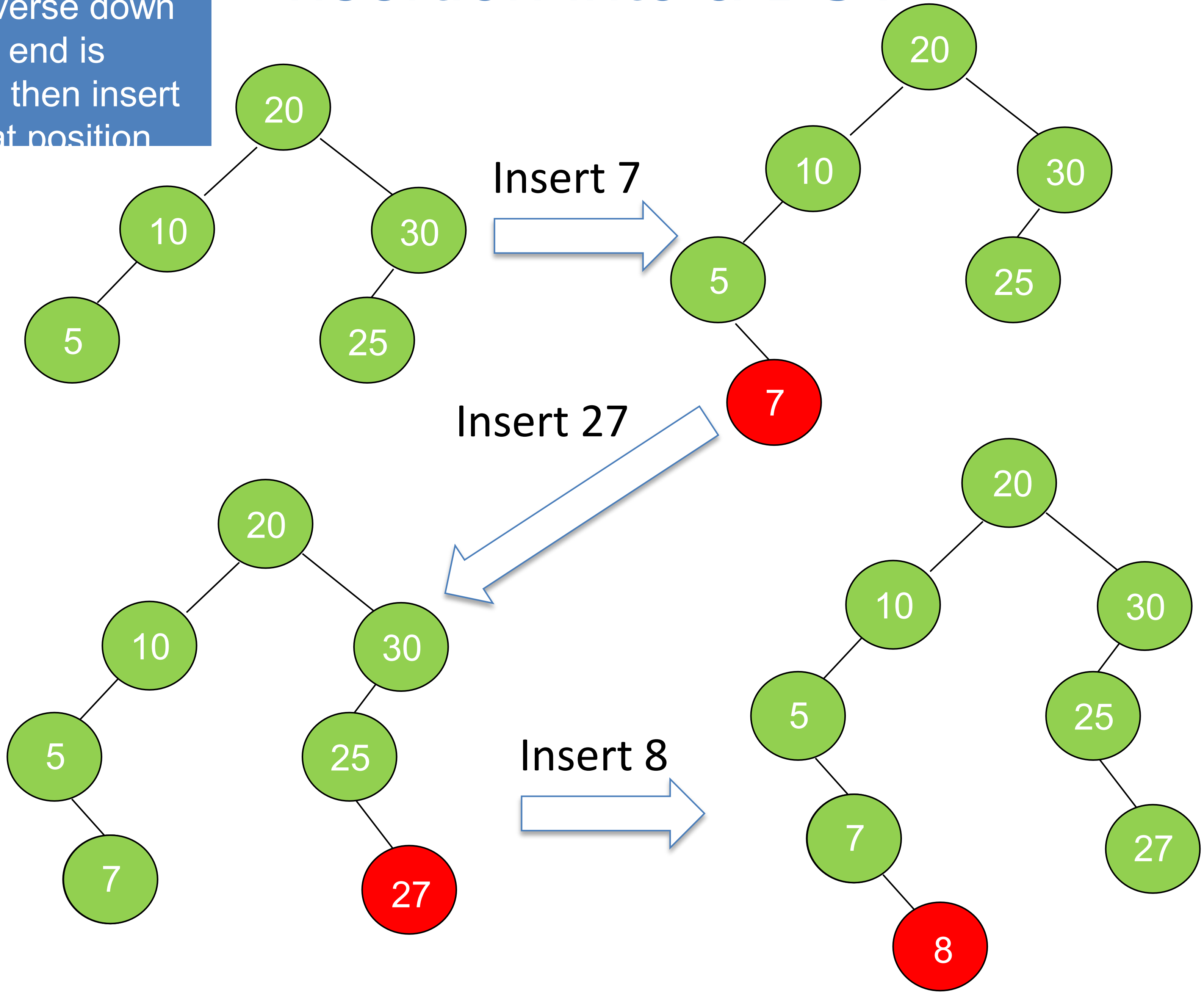
look right

`t.search('L')`

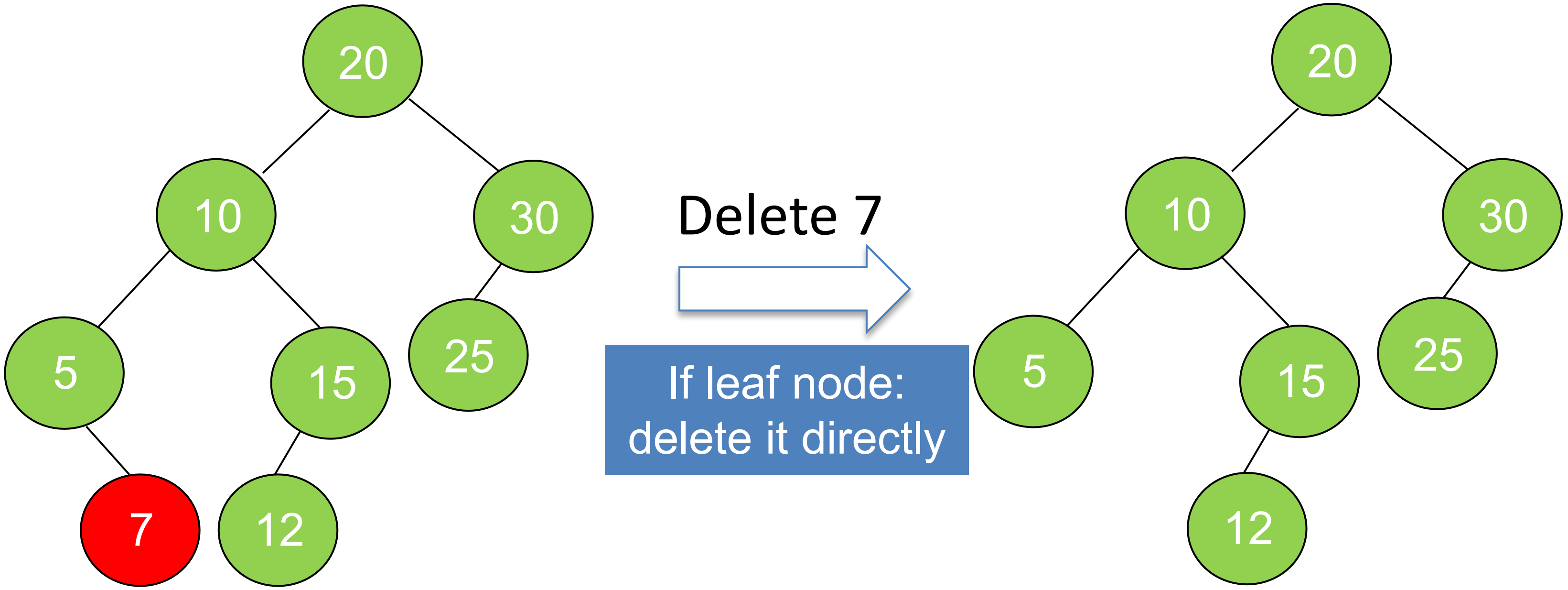


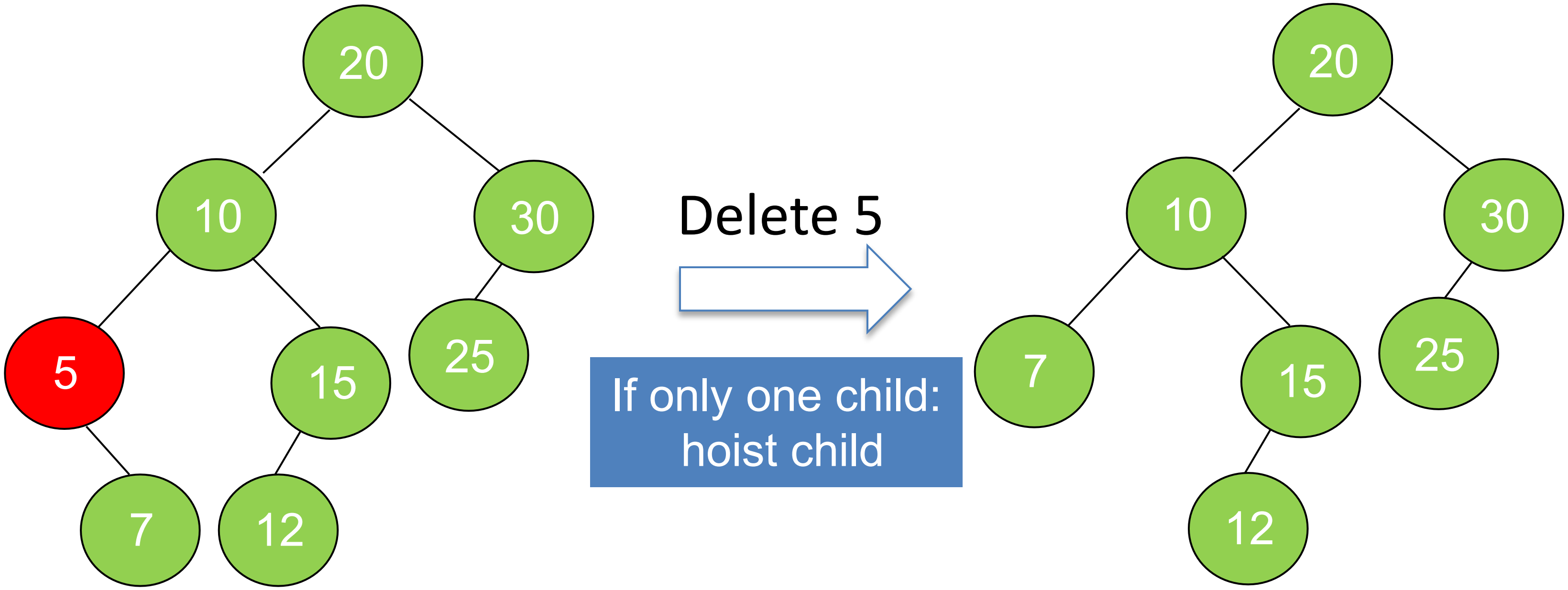
Insertion into a BST

Run BST search algo, traverse down tree until end is reached, then insert it into that position



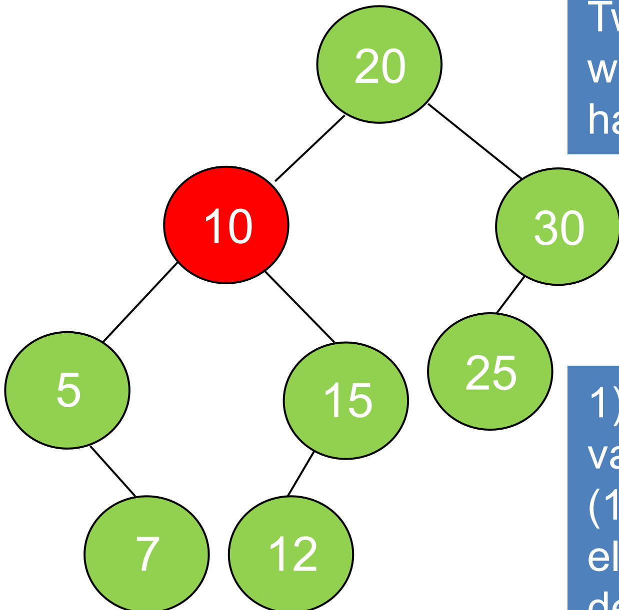
Deletion from a BST



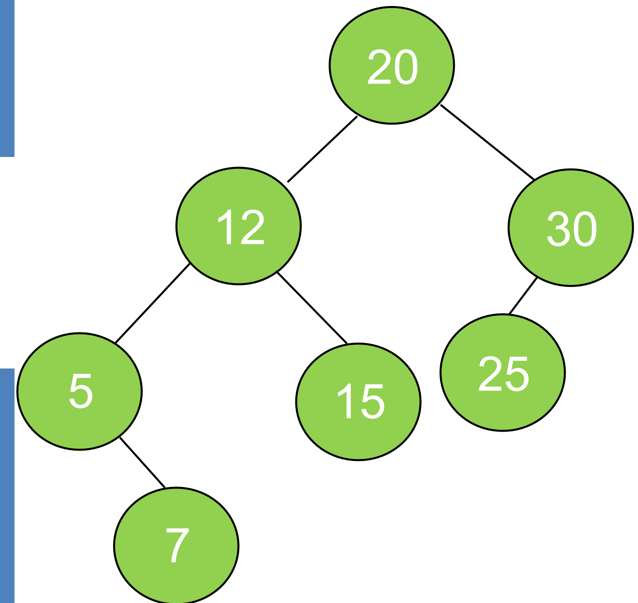


Deletion from a BST

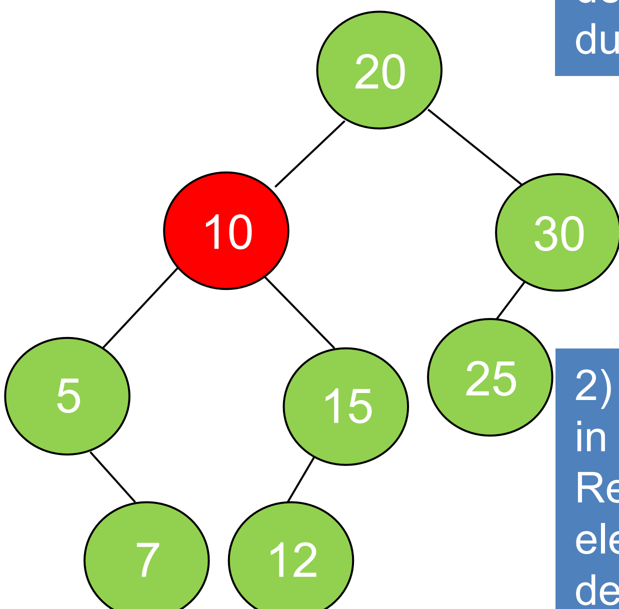
Two alternatives when a deleted node has two children.



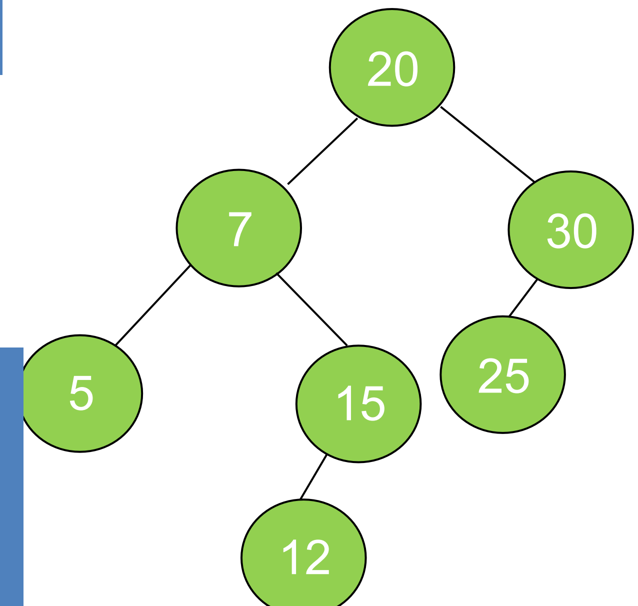
Delete 10



1) Find smallest value in right subtree (12). Replace deleted element with it, then delete right subtree duplicate.



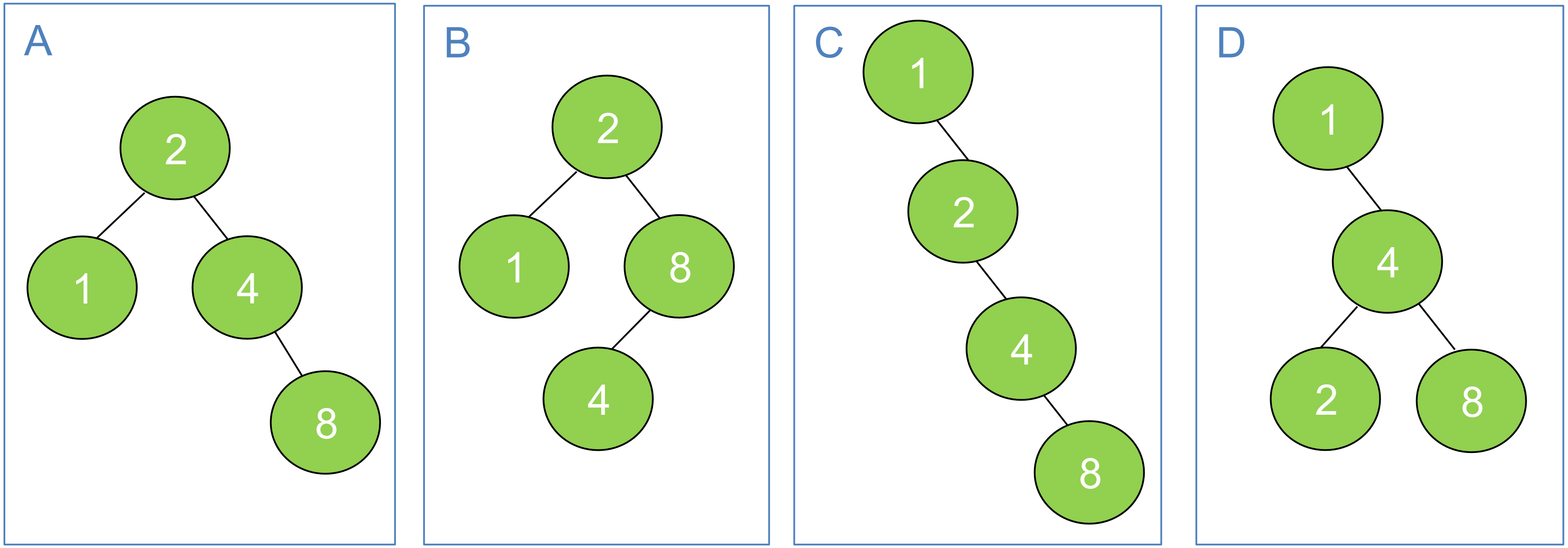
Delete 10



2) Find largest value in left subtree (7). Replace deleted element with it, then delete left subtree duplicate.

Binary Search Tree Shape

The following are all valid BSTs resulting from adding elements: 1, 2, 4, and 8 in some order.
The order in which we put elements into a BST impacts the shape, and the shape of a BST has a huge impact on the performance of operations.

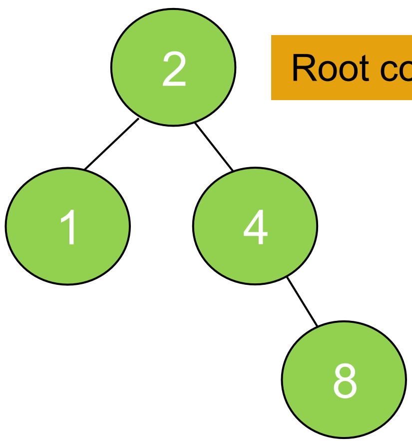


Binary Search Tree Shape (Contd.)

Inserting a node means making it a child of an existing node

Insert nodes as leaves

A ✓



Root comes first

8 needs to be inserted AFTER 4

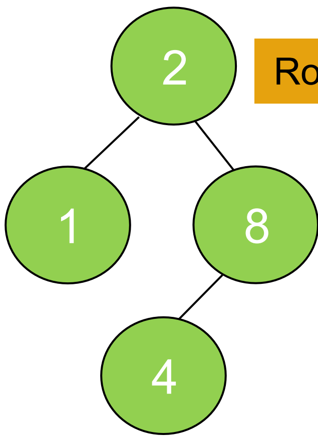
2			

2	8	1	4
2	1	8	4
2	8	4	1

2			

2	4	1	8
2	1	4	8
2	4	8	1

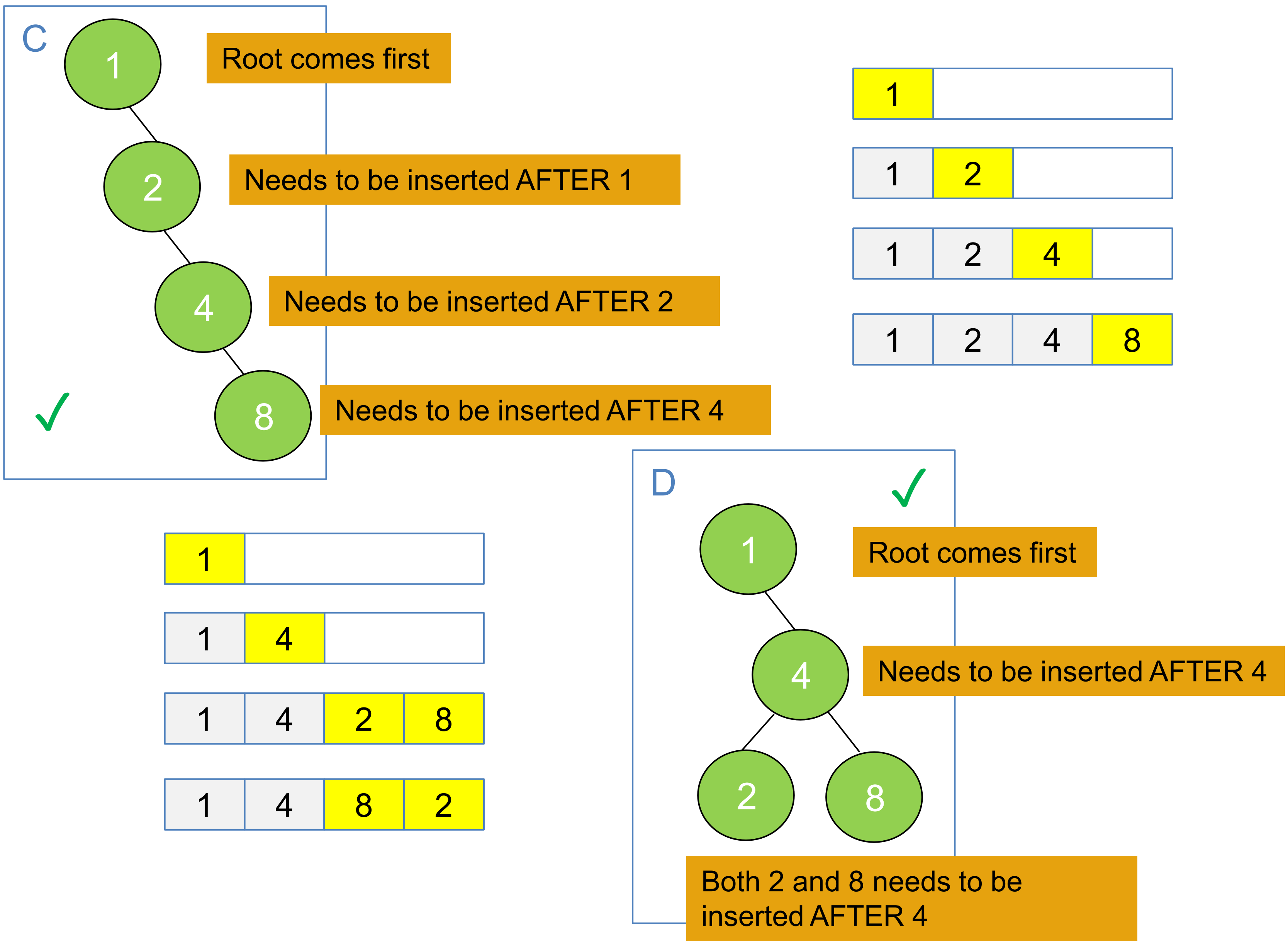
B ✓



Root comes first

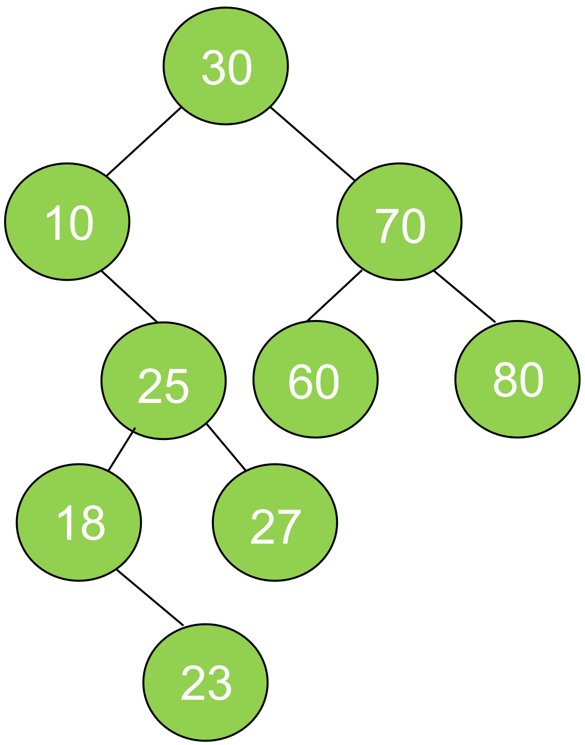
4 needs to be inserted AFTER 8

Binary Search Tree Shape (Contd.)



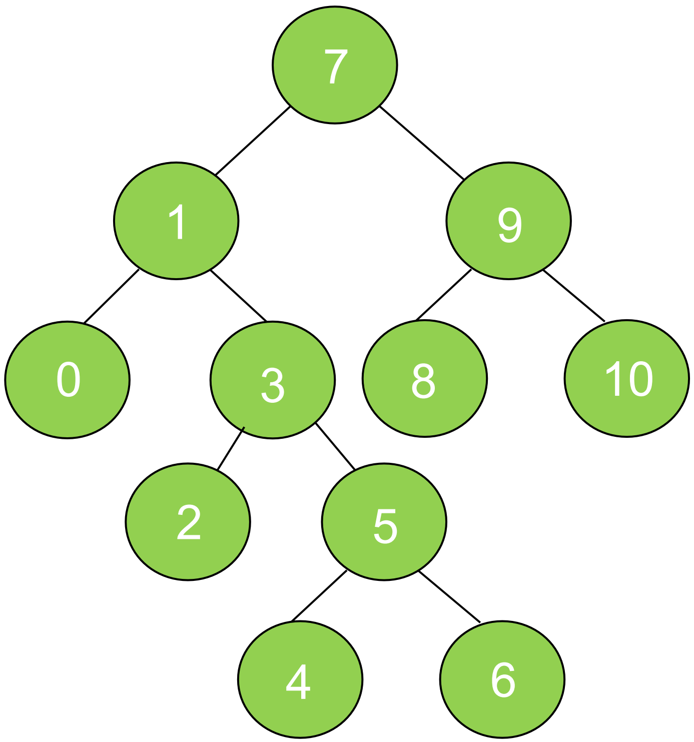
Traversal of a BST: Example I

- When we perform in-order traversal on a binary search tree, we get the ascending order array.



- **Pre-order traversal**:
- Traversal sequence: 30, 10, 25, 18, 23, 27, 70, 60, 80
- **In-order traversal**:
- Traversal Sequence: 10, 18, 23, 25, 27, 30, 60, 70, 80
- **Post-order traversal**:
- Traversal sequence: 23, 18, 27, 25, 10, 60, 80, 70, 30

Traversal of a BST: Example II



- **Pre-order traversal**:
- Begins at the root (**7**), ends at the right-most node (**10**)
- Traversal sequence: 7, 1, 0, 3, 2, 5, 4, 6, 9, 8, 10
- **In-order traversal**:
- Begins at the left-most node (**0**), ends at the rightmost node (**10**)
- Traversal Sequence: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
- **Post-order traversal**:
- Begins with the left-most node (**0**), ends at the root (**7**)
- Traversal sequence: 0, 2, 4, 6, 5, 3, 1, 8, 10, 9, 7

In-Order Traversal of a BST

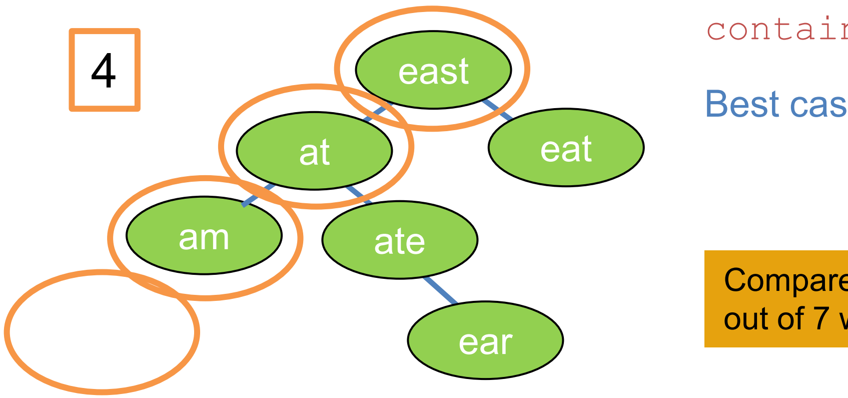
- In-order traversal of a BST visits the nodes in ascending order of their values, i.e., from smallest to largest.
  - BST Property: In a BST, for any given node:
    - Values in the left subtree are less than the value of the node.
    - Values in the right subtree are greater than the value of the node.
  - In-order Traversal:
    1) Traverse the left subtree.
    2) Visit the node itself.
    3) Traverse the right subtree.
  - Resulting Order: By first visiting all nodes in the left subtree (which are smaller), then the root, and finally all nodes in the right subtree (which are larger), in-order traversal naturally outputs the nodes in non-decreasing order.
- This property makes in-order traversal particularly useful for retrieving data from a BST in sorted order.

Performance Analysis of BST

Storing a dictionary as a BST

{ am, at, ate, ear, eat, east }

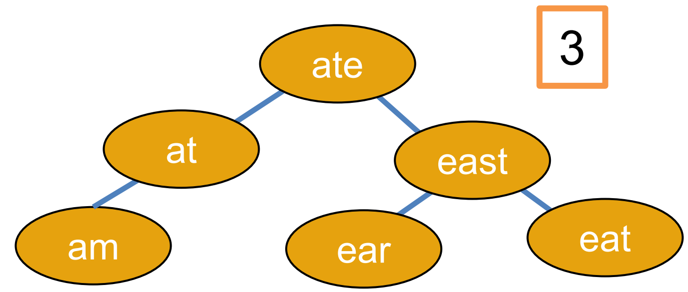
Structure of a BST depends on the order of insertion



`containsKey(root, east)`

Best case: O(1)

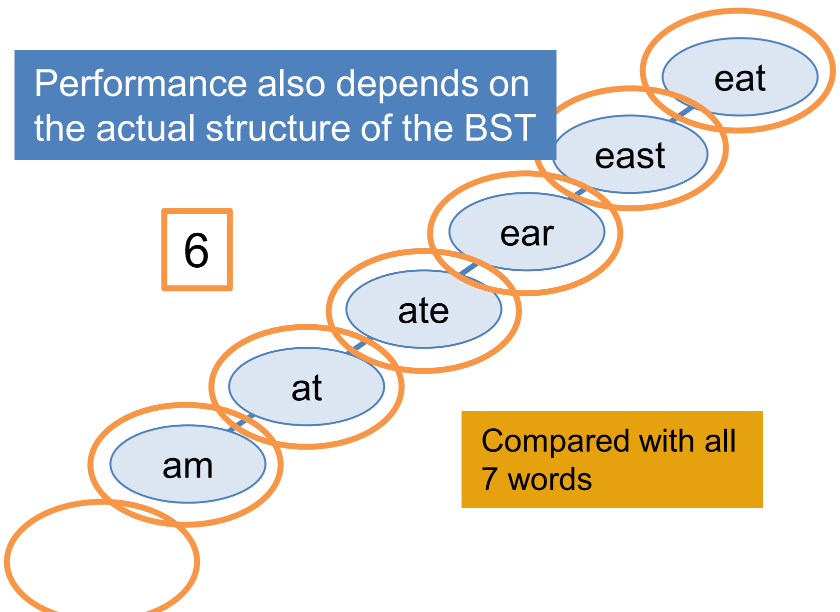
Compared with 3 out of 7 words



How does the performance scale with input size n?

```
public boolean containsKey(node, key) {
    if (node == null) {
        return false;
    } else if (node.key == key) {
        return true;
    } else {
        if (key <= node.key) {
            return containsKey(node.left);
        } else {
            return containsKey(node.right);
        }
    }
}
```

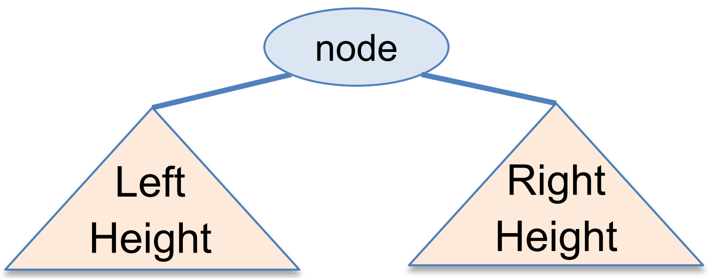
Performance also depends on the actual structure of the BST

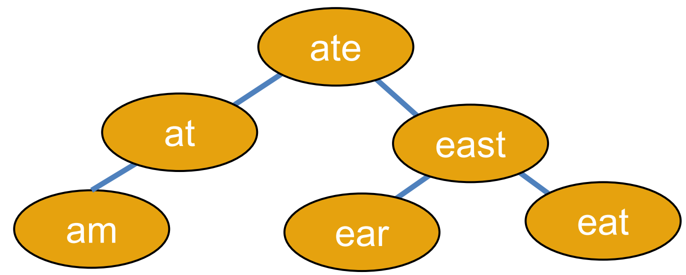


Compared with all 7 words

AVL Tree

AVL Tree: A balanced BST that maintains the invariant: |LeftHeight – RightHeight | <= 1 for all nodes in the tree. It minimizes the BST height. (discussed in next lecture.)





height ≈ log(n)

Inserting elements into BST in order results in a linked list!

	Best case	**Average case**	**Worst case**
Linked List	O(1)	O(n)	O(n)
BST	O(1)	O(log n)	O(n)
AVL Tree	O(1)	O(log n)	O(log n)

containsKey(root, key)

BST vs. Hash Table

- Time Complexity
  - Average case:
    - Hash Tables generally offer O(1) average time complexity for insertion, deletion, and search operations.
    - BSTs provide O(log n) time complexity for these operations, assuming the tree is balanced.
  - Worst case
    - Hash Tables can degrade to O(n) performance in cases of poor hash function design or many collisions.
    - BSTs maintain O(log n) performance even in the worst-case for self-balancing BST.
- Ordered Operations
  - BSTs excel at operations requiring ordered data
    - In-order traversal yields sorted elements.
    - Efficient range searches (e.g., finding all keys within a range)
  - Hash Tables do not inherently maintain order, making these operations more difficult.

Video Tutorials

- Tree Traversal Algos // Michael Sambol
  - https://www.youtube.com/playlist?list=PL9xmBV_5YoZO1JC2RgEi04nLy6D-rKk6b
- Binary Search Tree : Overview
  - https://www.youtube.com/watch?v=6I3evyt9ApA
- Binary Search Tree : Insert Overview
  - https://www.youtube.com/watch?v=KkEnuK-2Ymc
- Binary Search Tree: Deletion Overview
  - https://www.youtube.com/watch?v=DkOswl0k7s4
- Binary Search Tree Removal
  - https://www.youtube.com/watch?v=8K7EO7s_iFE
- Binary Search Trees (BST) Explained in Animated Demo
  - https://www.youtube.com/watch?v=mtvbVLK5xDQ