

Lecture 8

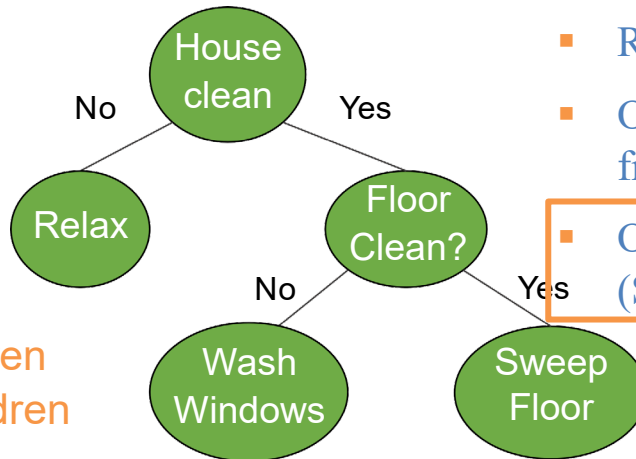
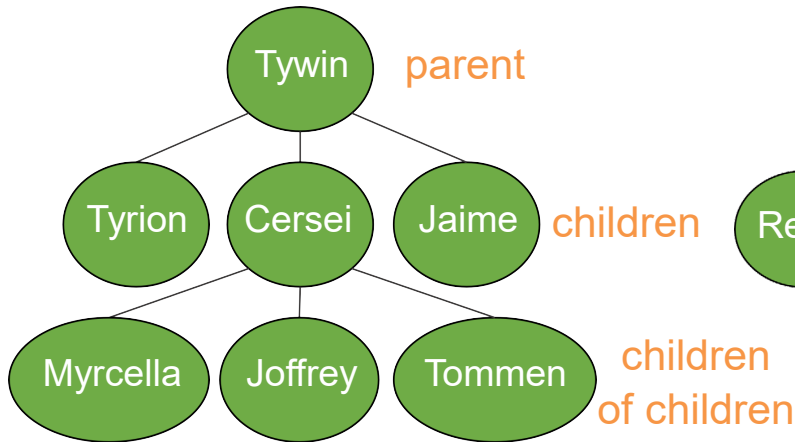
Binary Search Tree and Trie

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
- Compare the **performance** of linked lists and BSTs
- Explain what a **trie** data structure is

Different Trees in Computer Science



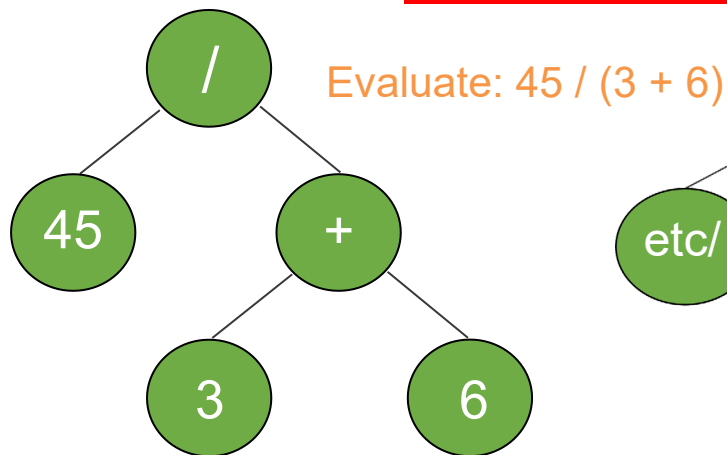
- Root is most important (Heap)
- Organized by character frequency (Huffman Tree)
- Organized by node ordering (Search Trees)
- Etc...

Family Trees

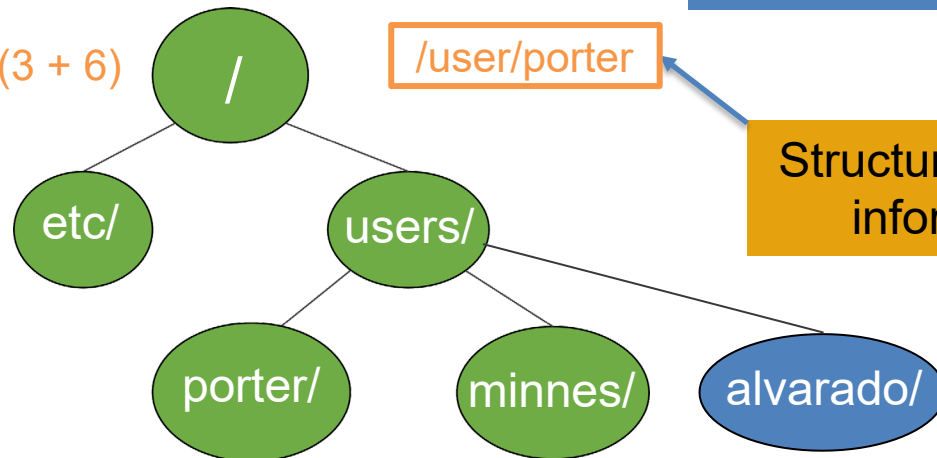
Why trees?

Decision Trees

Different Organizations
→ Different Trees



Expression Trees



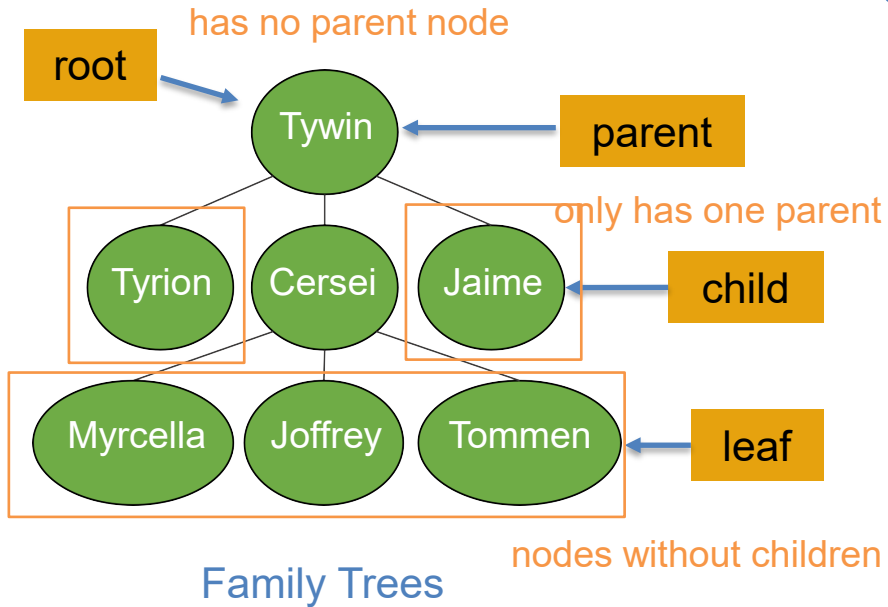
File System

/user/porter

Structure conveys
information

Dynamic Data Structure

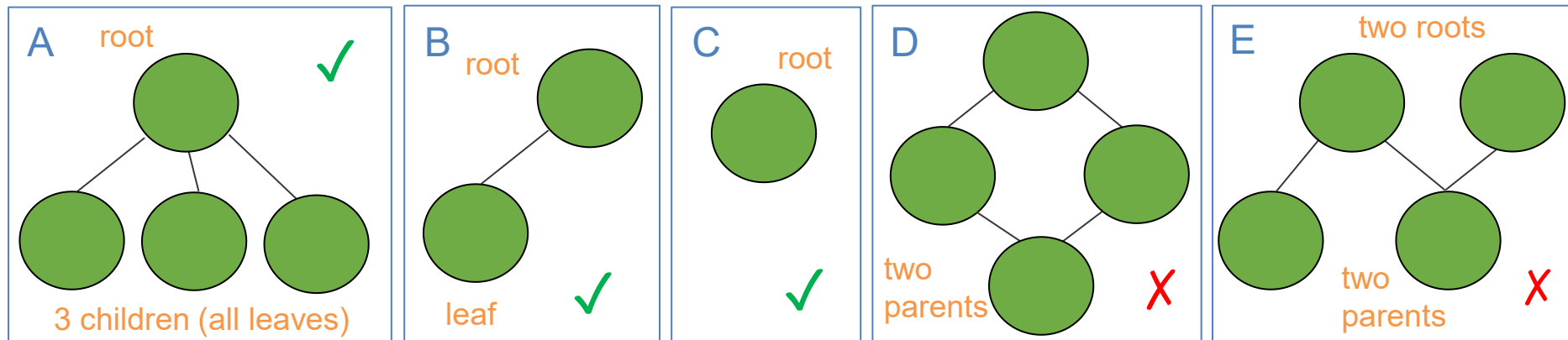
Defining Trees



What defines a tree?

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

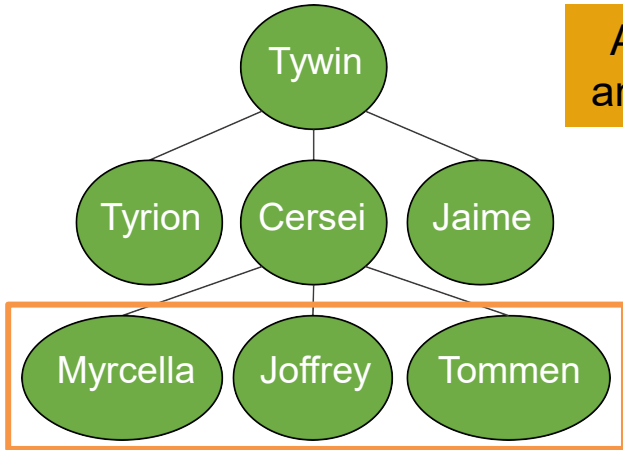
Which are trees?



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

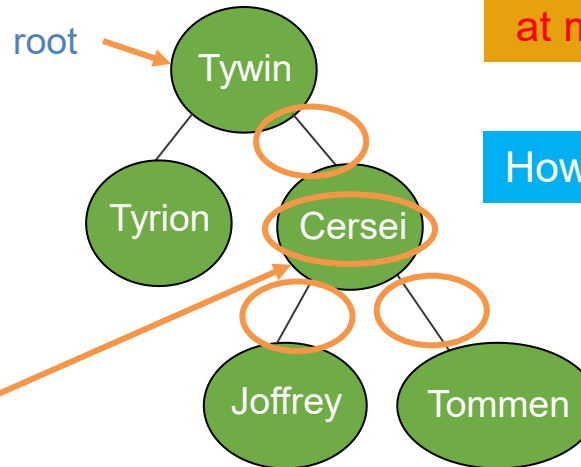
A tree just needs a root node

like the head and tail for linked list

Each node needs:

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

Binary Tree



Any Parent can have **at most** two children

How do we construct a tree?

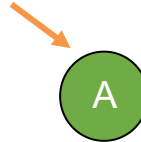
Like Linked Lists, Trees have a "Linked Structure"

nodes are connected by references

Write Code for Binary Tree

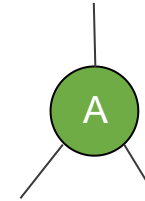
```
public class BinaryTree<E> {
    TreeNode<E> root;
    // more methods
}
```

root



```
public class TreeNode<E> {
    private E value;
    private TreeNode<E> parent;
    private TreeNode<E> left;
    private TreeNode<E> right;
    public TreeNode(E val, TreeNode<E> par) {
        this.value = val;
        this.parent = par;
        this.left = null;
        this.right = null;
    }
    public TreeNode<E> addLeftChild(E val) {
        this.left = new TreeNode<E>(val, this);
        return this.left;
    }
}
```

For root: `TreeNode(val, null)`

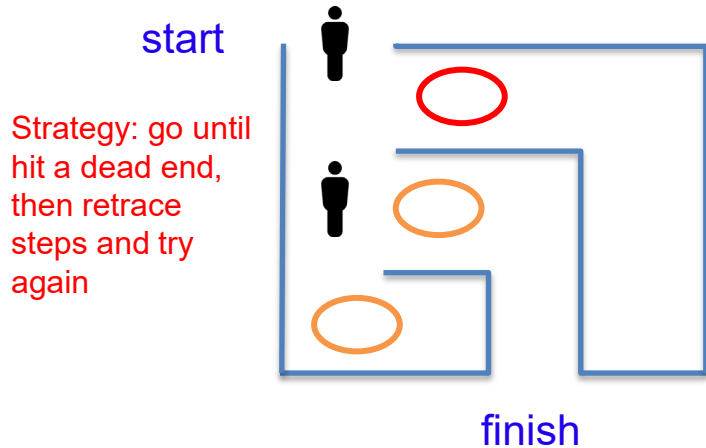


Let's write a constructor together

Next Step is to able to set/get children

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



Maze Traversal

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.

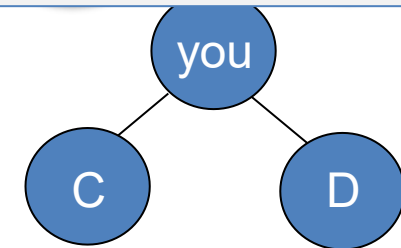
This problem benefits from "Breadth First Traversals"

Imagine this is a hedge maze

What's my next step?

Mazes benefit from "Depth First Traversals"

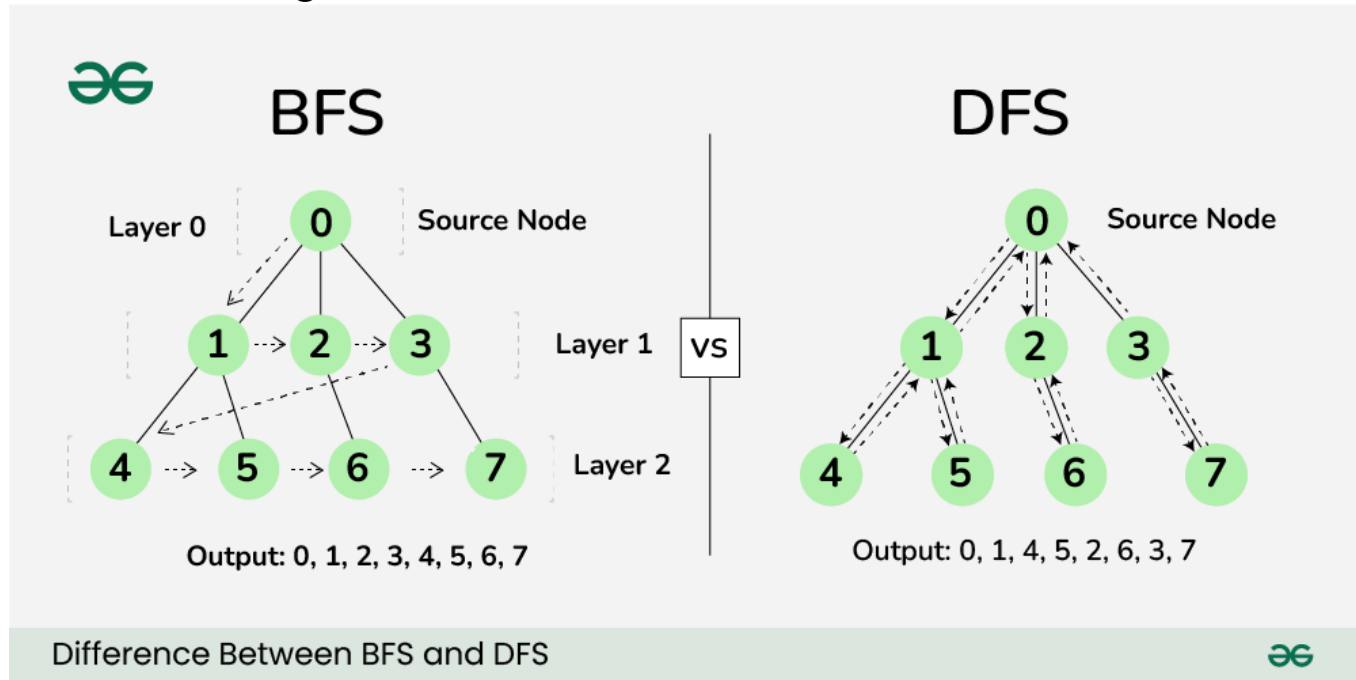
Bottom line: Order we visit matters and we'll make choices based on our needs



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

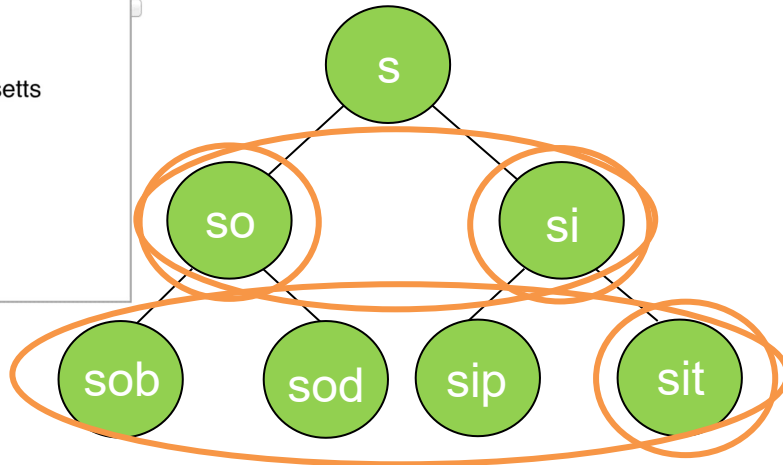
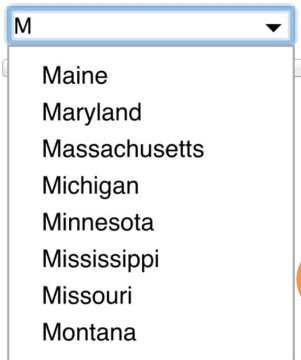
Tree Traversal Algos // Michael Sambol

<https://www.youtube.com/watch?v=iaBEKo5sM7w>

Traversal Order for Binary Trees

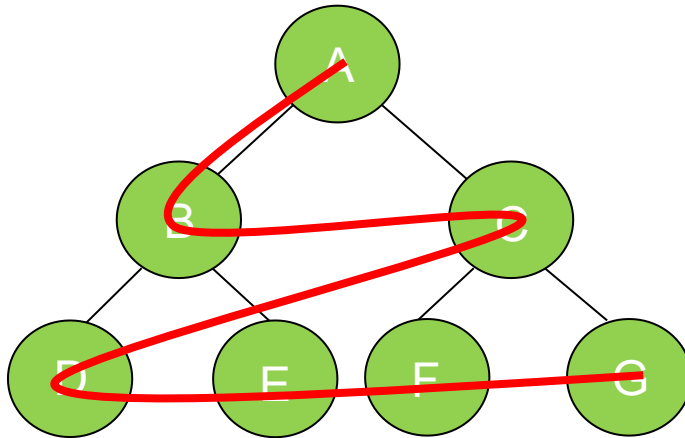
- Breadth First Traversal with BFS
 - Level Order Traversal
- Depth First Traversals with DFS
 - Pre-order Traversal (Root-Left-Right)
 - In-order Traversal (Left-Root-Right)
 - Post-order Traversal (Left-Right-Root)

Graph traversal with BFS: Level-order Traversal



- You've typed "s" What words should we suggest?
- Most frequent?
- How about "closest"?

"Breadth First Traversal"



Visit: Level-order

A B C D E F G

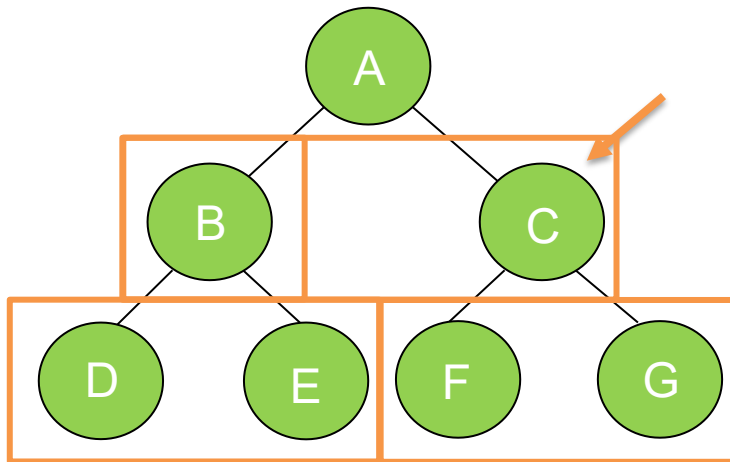
Level-order is
"Breadth First Traversal"

Pre/In/Post Order are:
"Depth First Traversals"

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

Level-order Traversal Implementation

LinkedList implements both
list and queue interfaces

```
public class BinaryTree<E> {  
    TreeNode<E> root;  
    public void levelOrder() {  
        Queue<TreeNode<E>> q = new LinkedList<TreeNode<E>>();  
        q.add(root);  
        while(!q.isEmpty()) {  
            TreeNode<E> curr = q.remove();  
            if(curr != null) {  
                curr.visit();  
                q.add(curr.getLeftChild());  
                q.add(curr.getRightChild());  
            }  
        }  
    }  
}
```

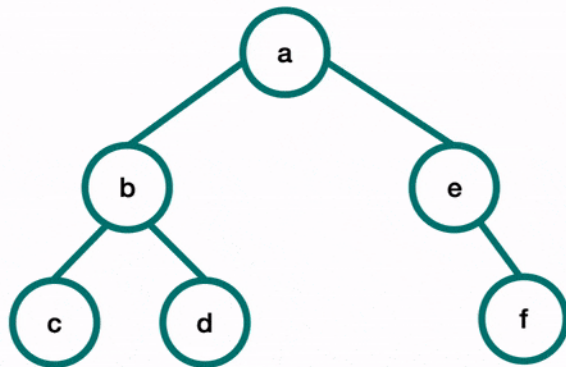


Could also check for null
children before adding

Graph traversal with DFS: pre-order, in-order, post-order

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

Pre-Order Traversal

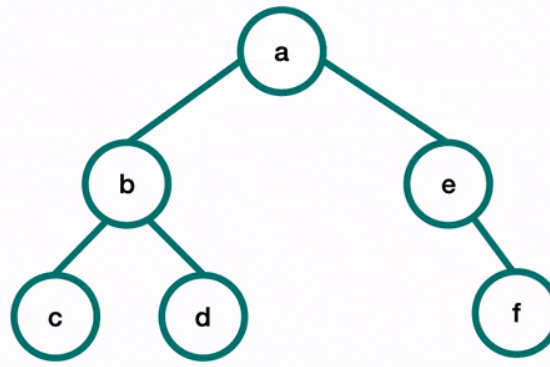


Print ""

abcdef

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

In-Order Traversal

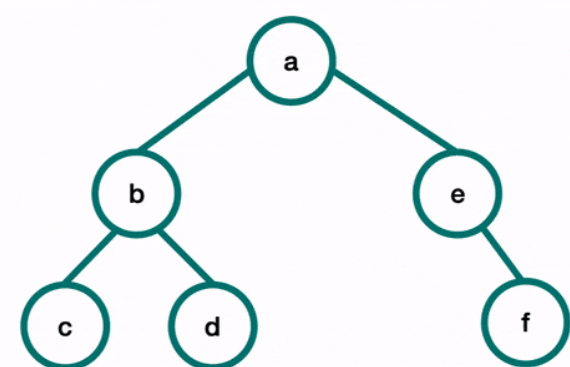


Print ""

cbdaef

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

Post-Order Traversal



Print ""

cdbfea

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

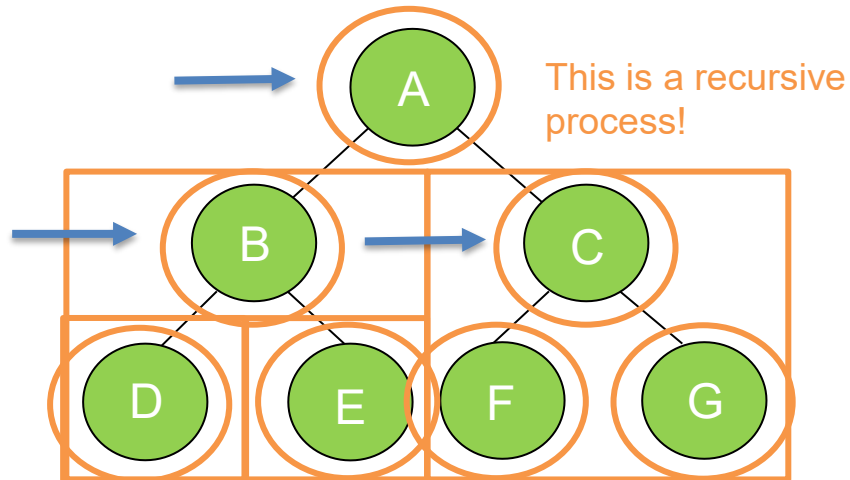
Inorder Traversal in Binary Tree Animations
<https://www.youtube.com/watch?v=ne5oOmYdWGw>

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

Graph traversal with DFS: pre-order, in-order, post-order

- Pre-order Traversal Algorithm | Tree Traversal | Visualization, Code, Example
 - <https://www.youtube.com/watch?v=8xue-ZBlTKQ>
- In-order Traversal Algorithm | Tree Traversal | Visualization, Code, Example
 - https://www.youtube.com/watch?v=4_UDUj1j1KQ
- Post-order Traversal Algorithm | Tree Traversal | Visualization, Code, Example
 - <https://www.youtube.com/watch?v=4Xo-GtBiQN0>

Pre-order Traversal (Recursively)



Idea:

- Visit yourself
- Then visit all your left subtree
- Then visit all your right subtree

Visited:

A B D E C F G

What's the order in which you think the nodes will be visited?

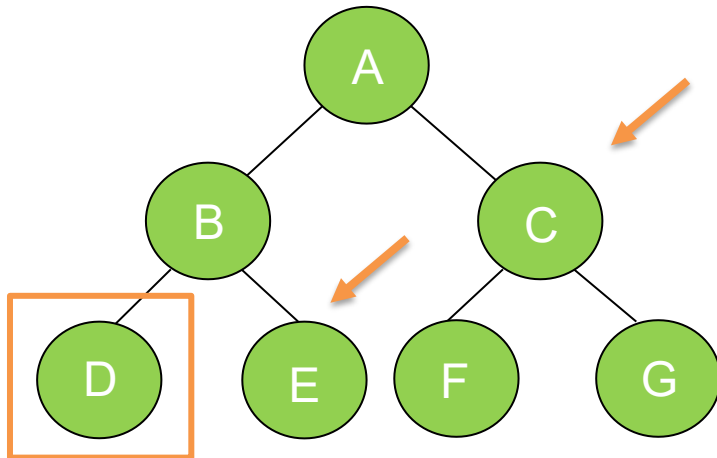
Recursion will help us do this!

This can be done iteratively

```
public class BinaryTree<E> {
    TreeNode<E> root;
    private void preOrder(TreeNode<E> node) {
        if (node != null) {
            node.visit();
            preOrder(node.getLeftChild());
            preOrder(node.getRightChild());
        }
    }
    public void preOrder() {
        this.preOrder(root);
    }
}
```

Iterative
algorithm not
covered in
exam

Pre-order Traversal (Iteratively)



Challenging: When we finish D, how do we go to E and C next?

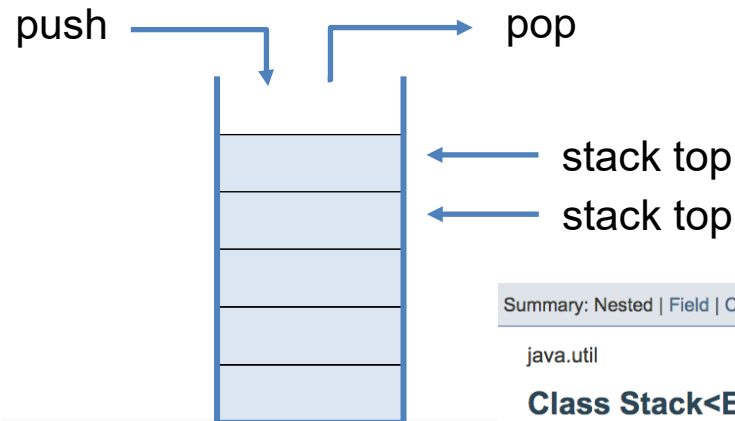
Idea: Keep a list and keep adding to it and removing from end.

Visit: A B D E C F G

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

We used this list like a "Stack"

- Add to the top
- Remove from the top
- Last-In, First-Out (LIFO)



Summary: Nested | Field | Constr | Method

java.util

Class Stack<E>

Methods

Modifier and Type	Method and Description
boolean	empty() Tests if this stack is empty.
E	peek() Looks at the object at the top of this stack v
E	pop() Removes the object at the top of this stack
E	push(E item) Pushes an item onto the top of this stack.

PREORDER TRAVERSAL USING A STACK

<https://www.youtube.com/watch?v=zvleLiQn- I>

Pre-order Traversal (Iteratively)

```

public class BinaryTree<E> {
    TreeNode<E> root;

    void iterativePreorder(TreeNode<E> par) {
        if (par == null) { return; }
        Stack<TreeNode<E>> nodeStack = new Stack<TreeNode<E>>();
        nodeStack.push(par);

        while (nodeStack.empty() == false) {
            TreeNode<E> node = nodeStack.peek();
            node.visit();
            nodeStack.pop();
            if (node.right != null) {
                nodeStack.push(node.right);
            }
            if (node.left != null) {
                nodeStack.push(node.left);
            }
        }
    }

    void iterativePreorder() {
        iterativePreorder(root);
    }
}

```

- 1) Create an empty stack *nodeStack* and push root node to stack.
- 2) Do following while *nodeStack* is not empty.
 -a) Pop an item from stack and print it.
 -b) Push right child of popped item to stack
 -c) Push left child of popped item to stack
 Right child is pushed before left child to make sure that left subtree is processed first.

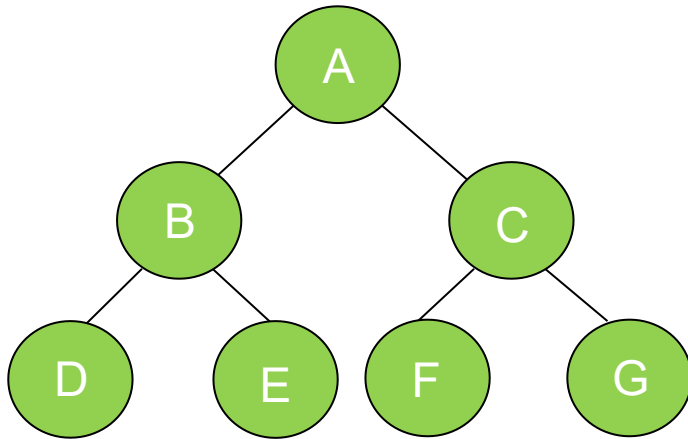
In-order Traversal (Recursively and Iteratively)

Iterative
algorithm not
covered in
exam

```
public class BinaryTree<E> {  
    TreeNode<E> root;  
  
    public void Inorder(TreeNode<E> node) {  
        if (node == null)  
            return;  
  
        Inorder(node.left);  
        node.visit();  
        Inorder(node.right);  
    }  
    void Inorder() { Inorder(root); }  
}
```

Recursive

- 1) Create an empty stack S.
- 2) Initialize current node as root
- 3) If current is not NULL, push the current node to S and set current = current->left. Repeat until current is NULL
- 4) If current is NULL and stack is not empty then
....a) Pop the top item from stack.
....b) Print the popped item, set current = popped_item->right
....c) Go to step 3.
- 5) If current is NULL and stack is empty then we are done.



Visit: D B E A F C G

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

```
public class BinaryTree<E> {  
    TreeNode<E> root;
```

Iterative

```
    public void iterativeInorder() {  
        if (root == null)  
            return;
```

```
        Stack<TreeNode<E>> s = new Stack<TreeNode<E>>();  
        TreeNode<E> curr = root;
```

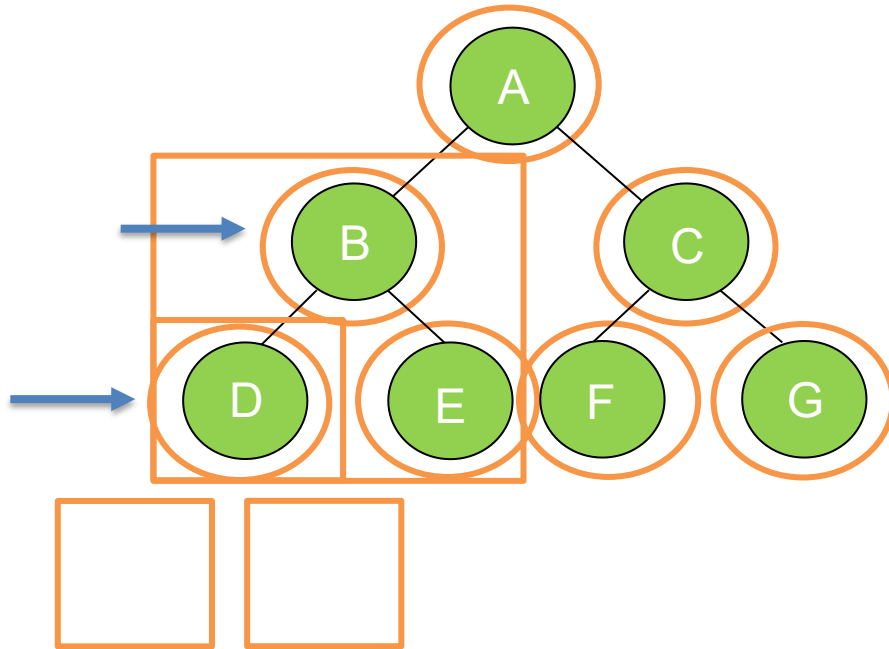
```
        while (curr != null || s.empty() == false) {
```

```
            while (curr != null) {  
                s.push(curr);  
                curr = curr.left;            }
```

```
            curr = s.pop();  
            curr.visit();  
            curr = curr.right;
```

```
        }  
    }
```

Post-order and In-order Traversal



Visit: In-order

What does this do?

- Visit all your left subtree
- Visit yourself
- Visit all your right subtree

Visit: Post-order

D E B F G C A

REARRANGE:

- ? Visit yourself
- ? Visit all your left subtree
- ? Visit all your right subtree

- Visit all your left subtree
- Visit all your right subtree
- Visit yourself

Fill in the Blank:

A. A B C D E F G

B. D B E A F C G

C. D B A E F C G

Recursion will
help us do these!

They can also be
done iteratively
with Stack.

Iterative
algorithm not
covered in
exam

Post-order Traversal (Recursively and Iteratively)

```
public class BinaryTree<E> {
    TreeNode<E> root;

    public void Postorder(TreeNode<E> node) {
        if (node == null)
            return;

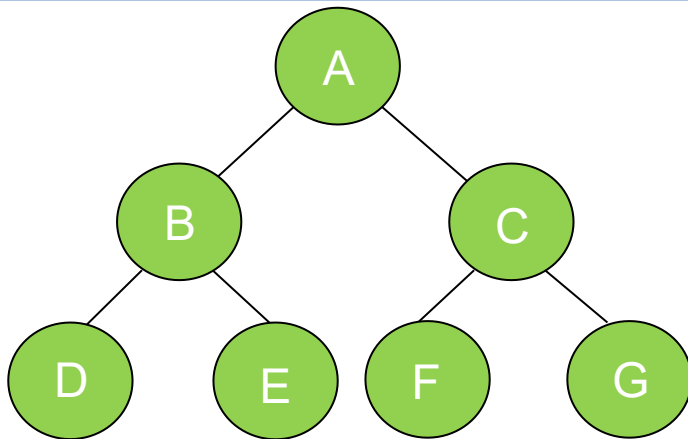
        Postorder(node.left);
        Postorder(node.right);
        node.visit();
    }

    void Posterorder() {Postorder(root); }
}
```

Recursive

For iterative version, the idea is to push reverse postorder traversal to a stack. Then, we can just pop all items one by one from the stack and visit them. To get reversed postorder elements in a stack – the second stack is used for this purpose. We can observe that this sequence is very similar to the preorder traversal. The only difference is that the right child is visited before left child.

1. Push root to first stack.
2. Loop while first stack is not empty
 -2.1 Pop a node from first stack and push it to second stack
 -2.2 Push left and right children of the popped node to first stack
3. Visit contents of second stack



Visit: D E B F G C A

Stack 2: ~~A~~ ~~C~~ ~~E~~ ~~F~~ ~~B~~ ~~E~~ ~~D~~

Stack 1: ~~A~~ ~~B~~ ~~C~~ ~~F~~ ~~G~~ ~~D~~ ~~E~~

```
public class BinaryTree<E> {
    TreeNode<E> root;

    public void iterativePostorder() {
        Stack<TreeNode<E>> s1 = new Stack<TreeNode<E>>();
        Stack<TreeNode<E>> s2 = new Stack<TreeNode<E>>();

        if (root == null)
            return;
        s1.push(root);
        while (!s1.isEmpty()) {
            TreeNode<E> temp = s1.pop();
            s2.push(temp);

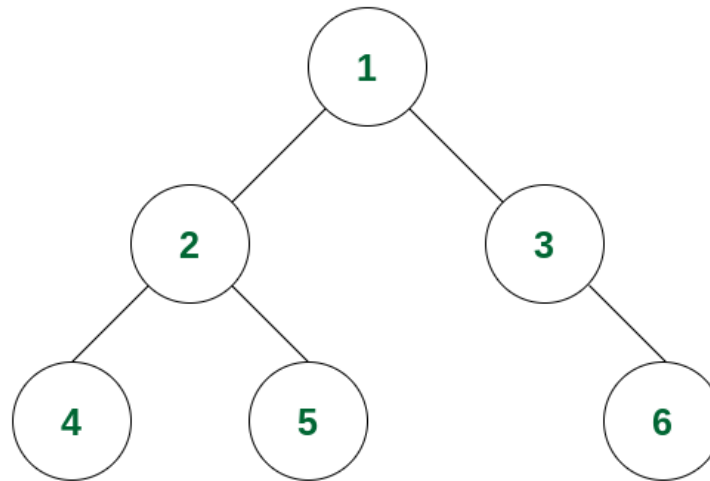
            if (temp.left != null)
                s1.push(temp.left);
            if (temp.right != null)
                s1.push(temp.right);
        }
        while (!s2.isEmpty()) {
            TreeNode<E> temp = s2.pop();
            temp.visit();
        }
    }
}
```

Iterative

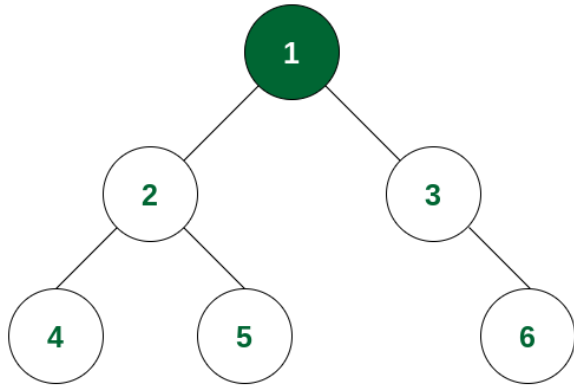
visit all elements of second stack

Geeks for Geeks Tutorials

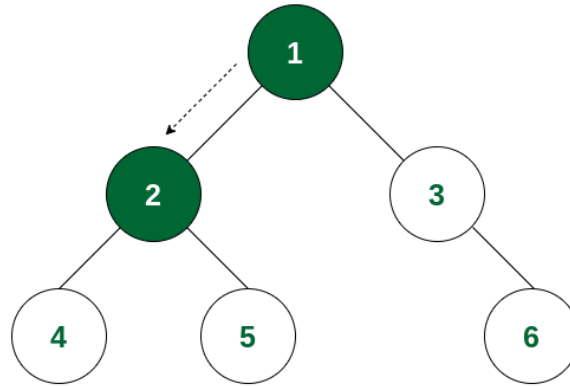
- <https://www.geeksforgeeks.org/preorder-traversal-of-binary-tree/>
- <https://www.geeksforgeeks.org/inorder-traversal-of-binary-tree/>
- <https://www.geeksforgeeks.org/postorder-traversal-of-binary-tree/>
- Running Example



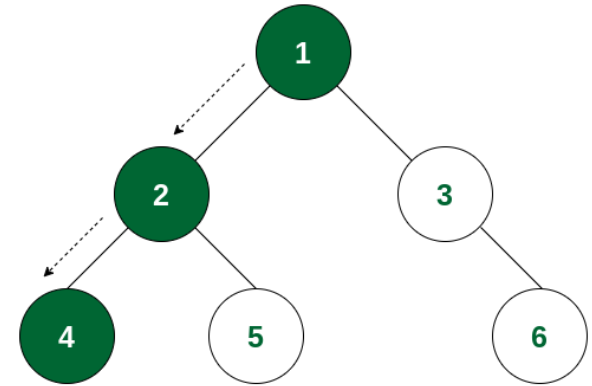
Pre-order traversal of nodes is 1 -> 2 -> 4 -> 5 -> 3 -> 6



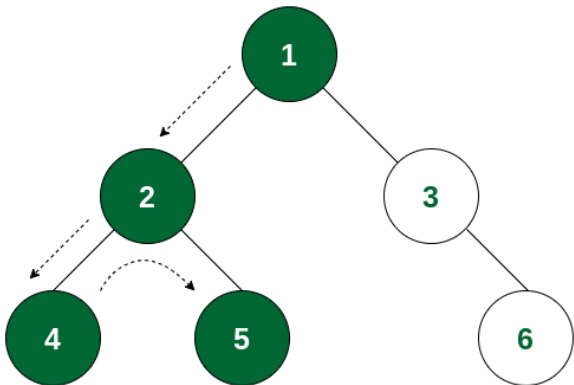
Root of the tree (i.e., 1) is visited



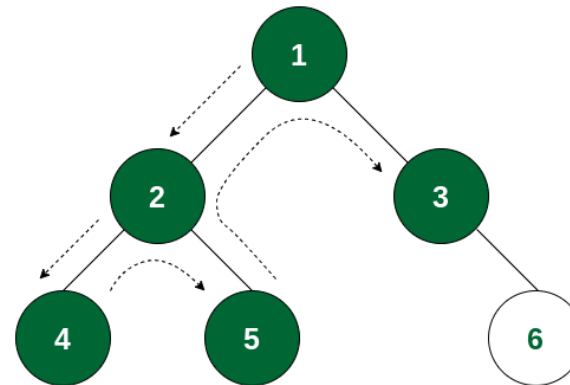
Root of left subtree of 1 (i.e., 2) is visited



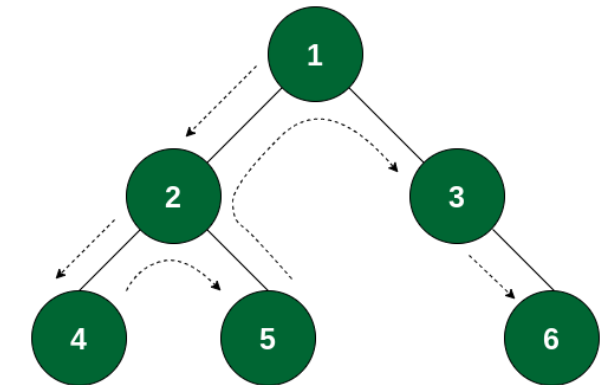
Left child of 2 (i.e., 4) is visited



Right child of 2 (i.e., 5) is visited

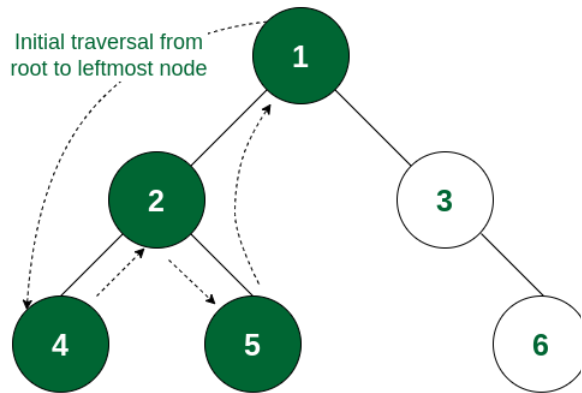
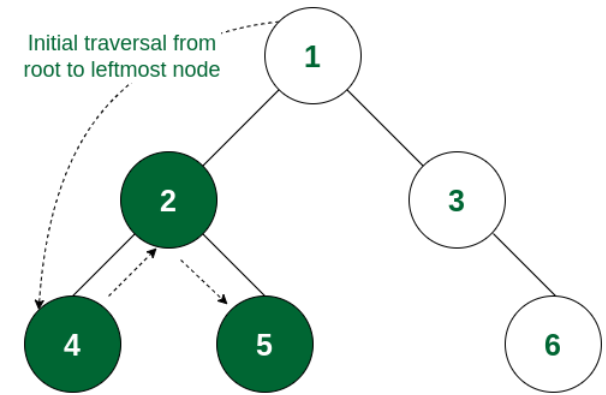
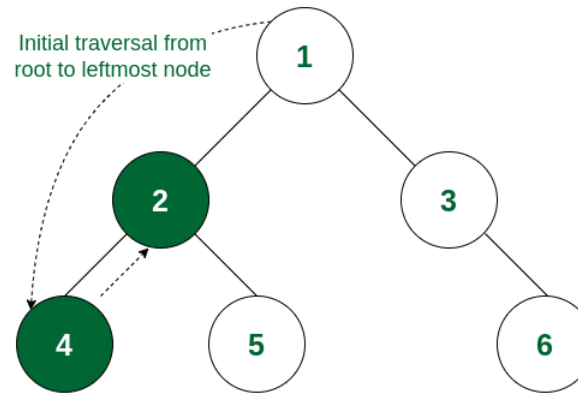
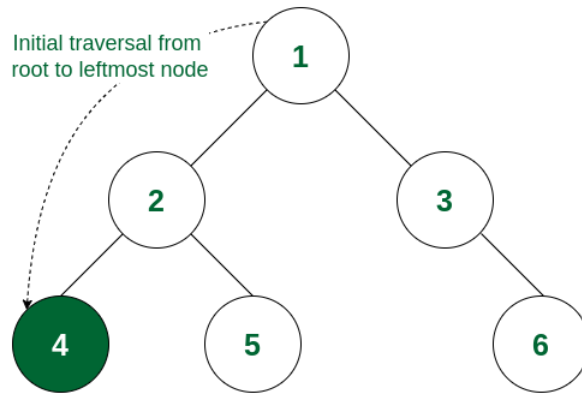


Root of right subtree of 1 (i.e., 3) is visited

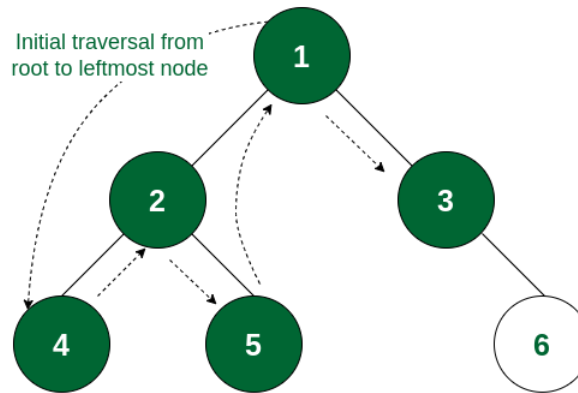


3 has no left subtree. So right subtree is visited

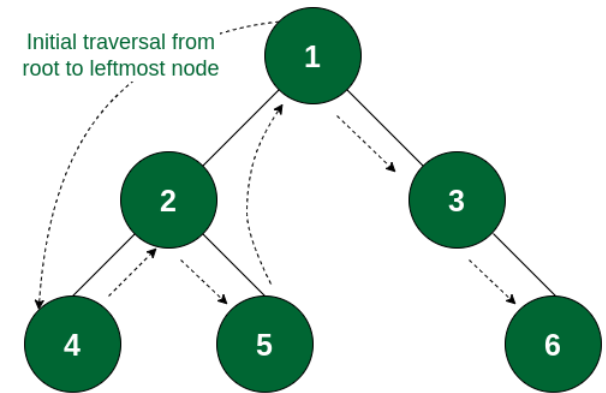
In-order traversal of nodes is 4 -> 2 -> 5 -> 1
-> 3 -> 6.



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

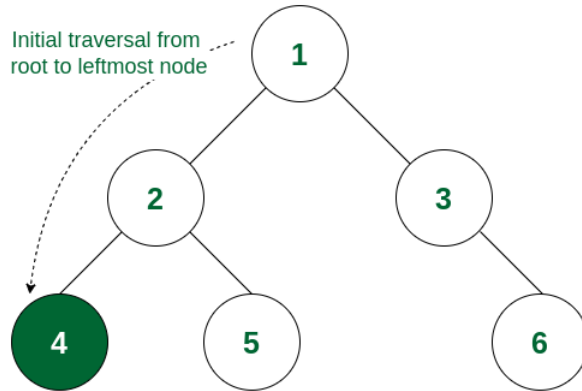


3 has no left subtree, so it is visited

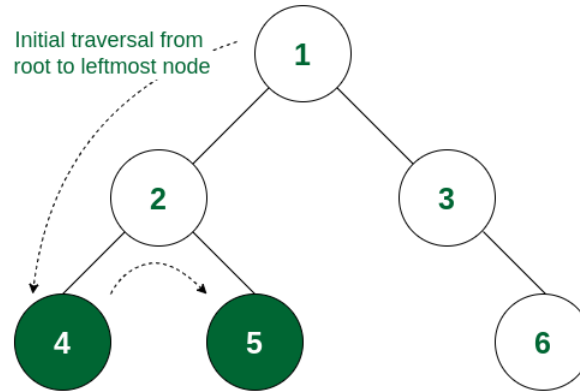


Right Child of 3 is visited

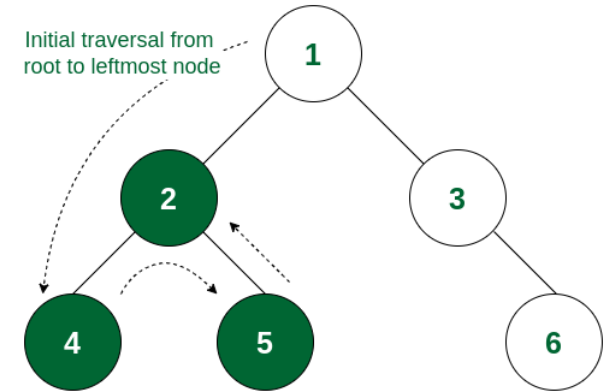
Post-order traversal of nodes is 4 -> 5 -> 2 -> 6 -> 3 -> 1



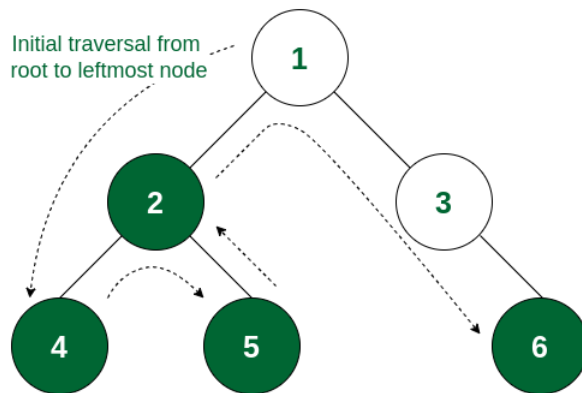
The leftmost leaf node (i.e., 4) is visited first



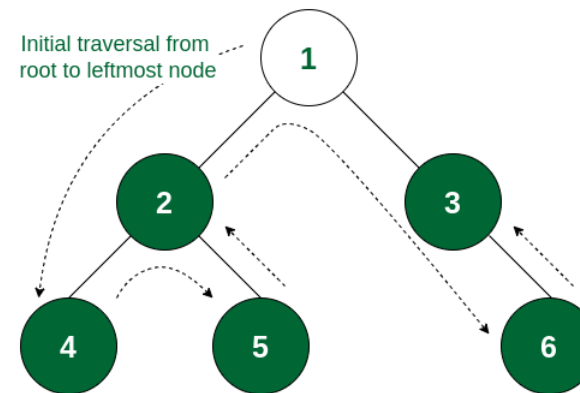
Left subtree of 2 is traversed. So 5 is visited next



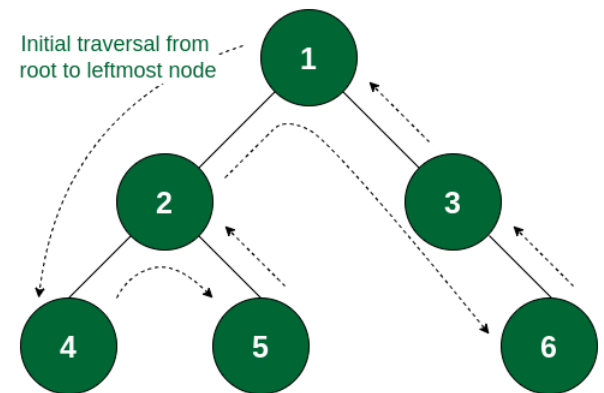
All subtrees of 2 are visited. So 2 is visited next



6 has no subtrees. So it is visited



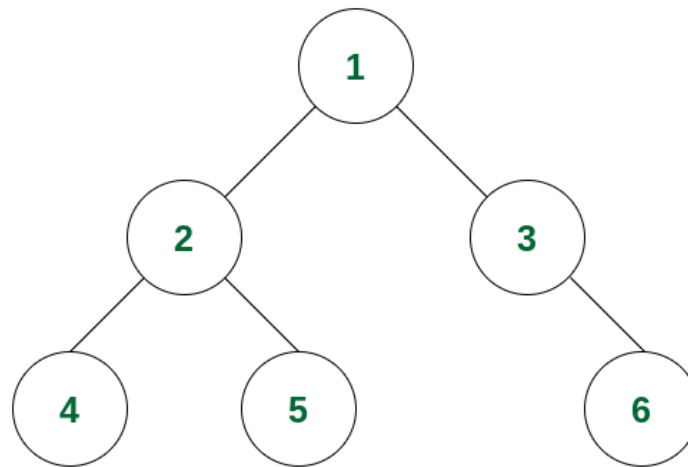
3 is visited after all its subtrees are traversed



The root of the tree (i.e., 1) is visited

Quiz

- Given: Pre-order traversal of nodes is 1 -> 2 -> 4 -> 5 -> 3 -> 6; In-order traversal of nodes is 4 -> 2 -> 5 -> 1 -> 3 -> 6. What is the post-order traversal of nodes?
- ANS: we know 1 is the tree root from pre-order traversal, so we know the left subtree has nodes 4,2,5, and right subtree has nodes 3,6, from in-order traversal 4 -> 2 -> 5 -> 1 -> 3 -> 6. We can draw the tree now and derive the post order traversal 4 -> 5 -> 2 -> 6 -> 3 -> 1



Motivation for Binary Search Tree

Agra	Beijing	Chicago	Essen	Lagos	Montreal	Quito
------	---------	---------	-------	-------	----------	-------



Binary Search - $O(\log n)$ search:
get rid of half each time

toFind

Chicago

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

root

Essen

Beijing

Montreal

Agra

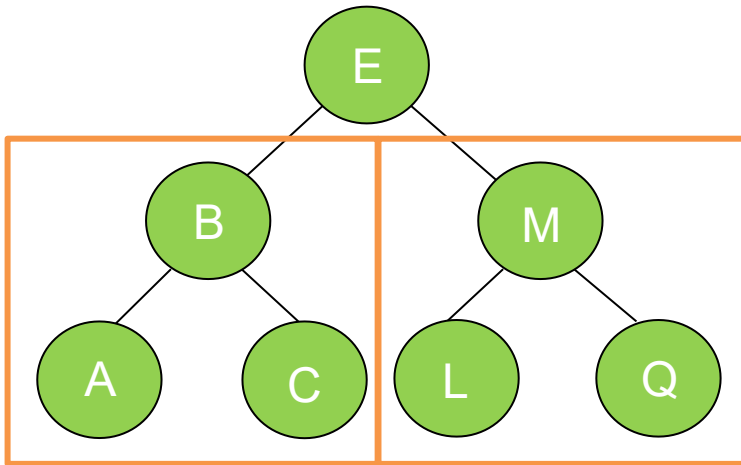
Chicago

Lagos

Quito

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 Trees

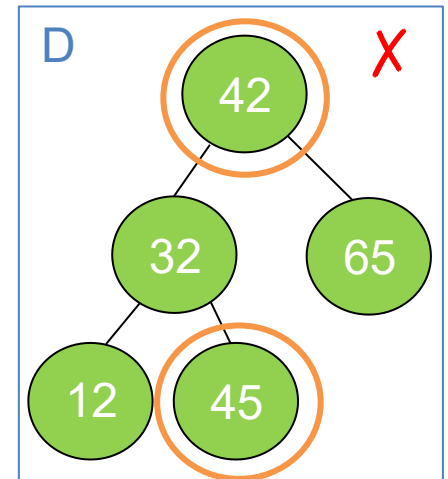
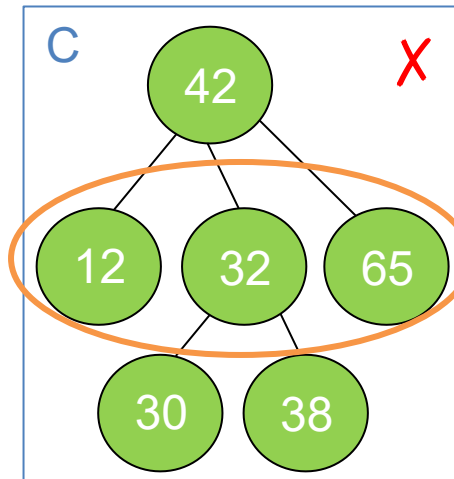
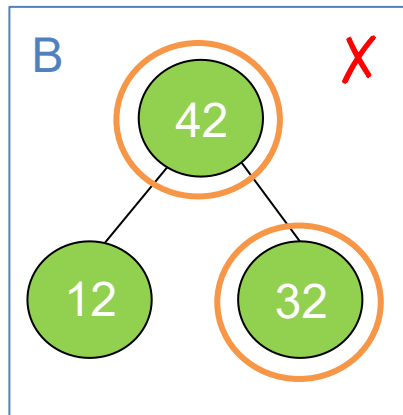
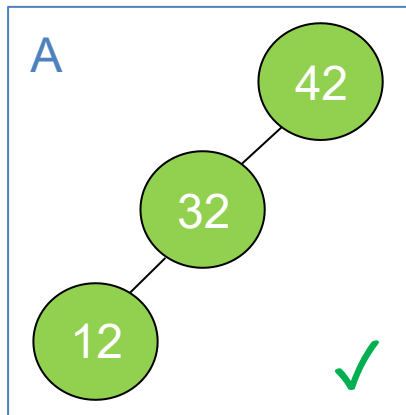


Left subtree's values
must be lesser

Right subtree's values
must be greater

- Ordered, or sorted, binary trees.
- Each node can have 2 subtrees.
- Items to the left of a given node are smaller.
- Items to the right of a given node are larger.

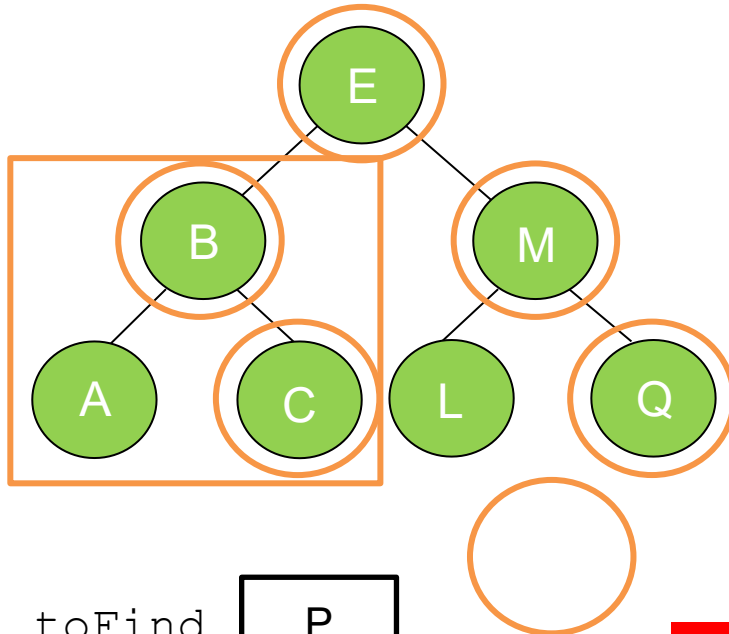
Which of these are binary search trees?



BST Video Tutorials

- 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>

Searching a BST



toFind

P

Compare: E and P

Compare: M and P

Compare: Q and P

Node is null

Not Found!

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!

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!

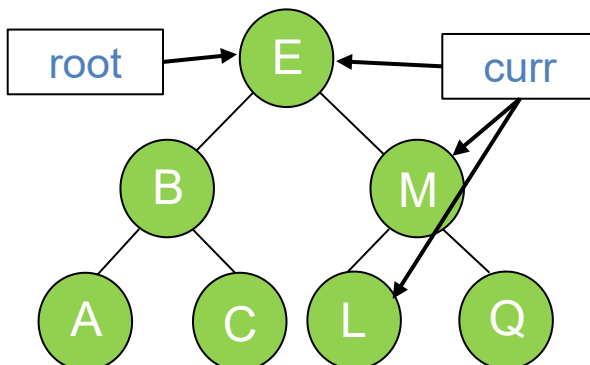
We need to do this over and over if not found

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

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

if calling object is equal to parameter, compareTo returns 0

Are we done?



t.search('L')

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;
```

Root of the tree we look at

```
private boolean search(TreeNode<E> p, E toSearch) {
```

```
    if (p == null)
```

```
        return false;
```

Tree is empty

```
    int comp = toSearch.compareTo(p.getValue());
```

```
    if (comp == 0)
```

```
        return true;
```

Found it!

```
    else if (comp < 0)
```

```
        return search(p.left, toSearch);
```

look left

```
    else // comp > 0
```

```
        return search(p.right, toSearch);
```

look right

```
}
```

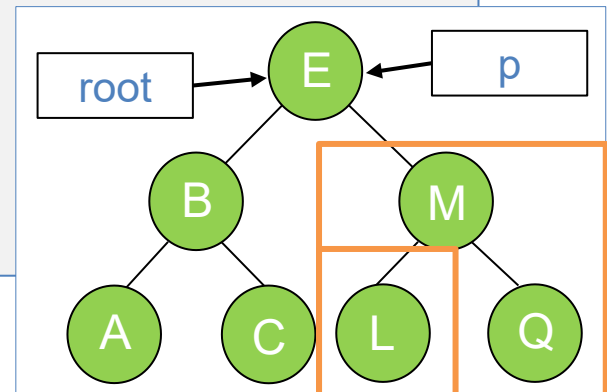
```
public boolean search(E toSearch) {
```

```
    return search(root, toSearch);
```

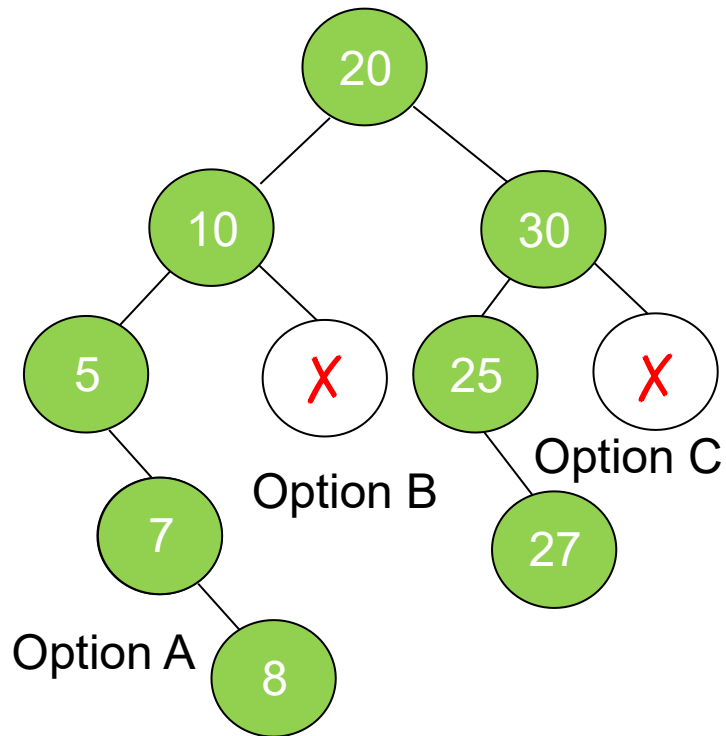
```
}
```

```
}
```

```
t.search('L')
```



Inserting into a BST



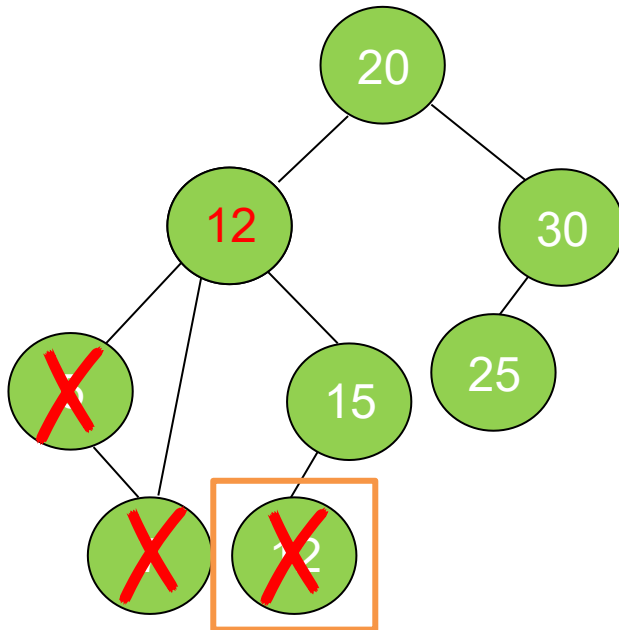
Where should we insert 7?

Insert 27?

Insert 8?

X Option D: Either Option A or Option B are fine.

Deleting from a BST



For the smallest value in a node's right subtree, its left child is null.

For the largest value in a node's left subtree, its right child is null.

Delete 7

If leaf node: Delete parent's link 7

Delete 5

If only one child, hoist child

Delete 10

When a deleted node has two children, this gets tricky.

Find smallest value in right subtree

Replace deleted element with it

Then delete right subtree duplicate (12)

Find largest value in left subtree

Replace deleted element with it

Then delete left subtree duplicate (7)

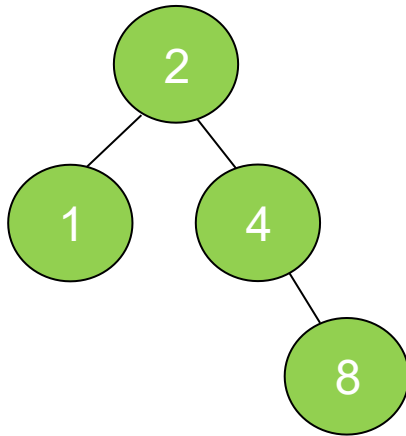
OR

Binary Search Tree Shape

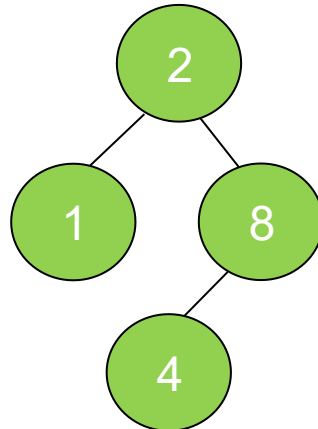
Which of the following Binary Search Trees could be the result of adding elements: 1, 2, 4, and 8 in some order.

These are all valid binary search trees!

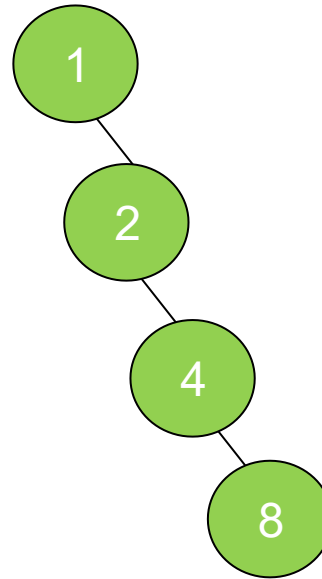
A



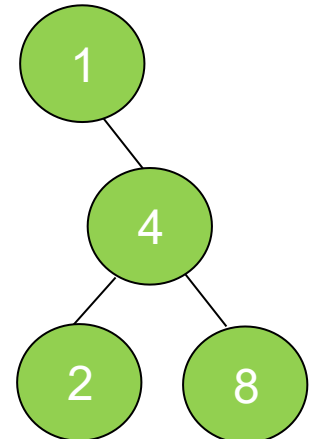
B



C



D



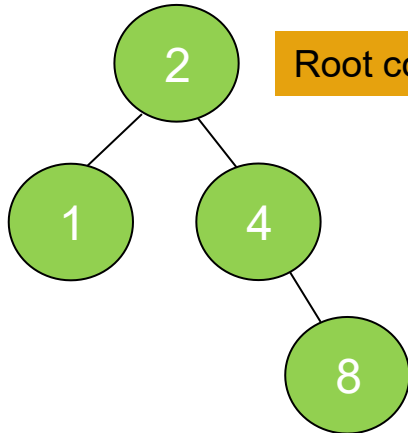
Binary Search Tree Shape (Contd.)

A



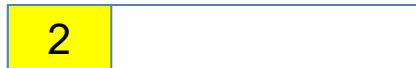
Insert nodes as leaves

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



Root comes first

8 needs to be inserted AFTER 4



2

2

4

1

8

2

1

4

8

2

4

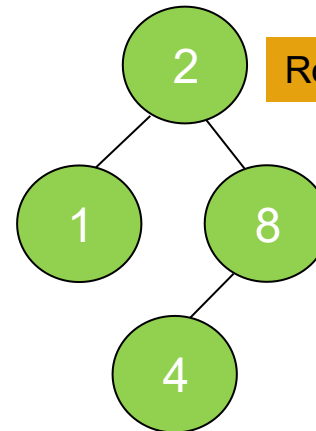
8

1

B

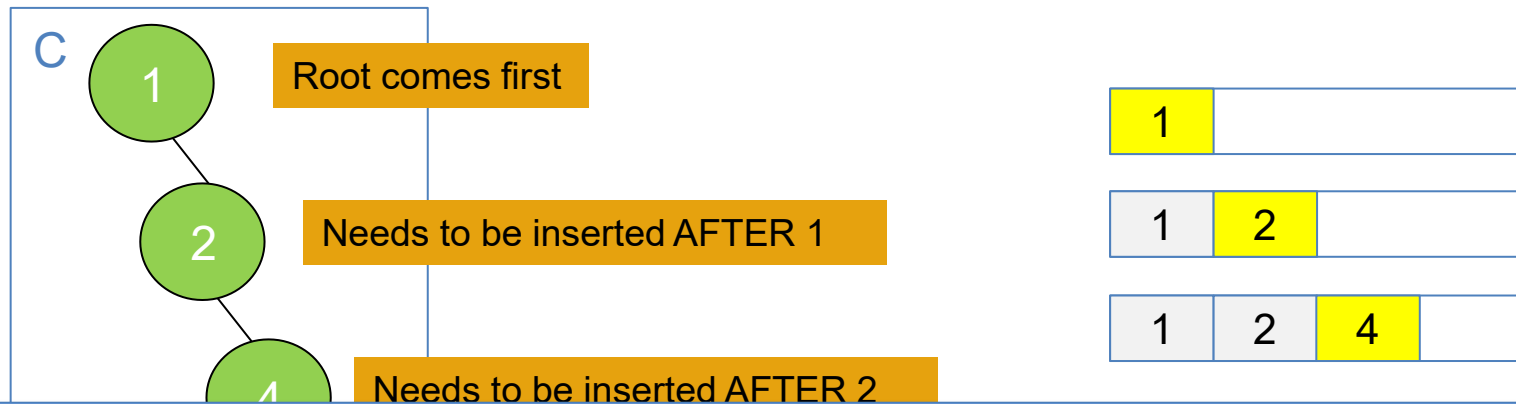


Root comes first

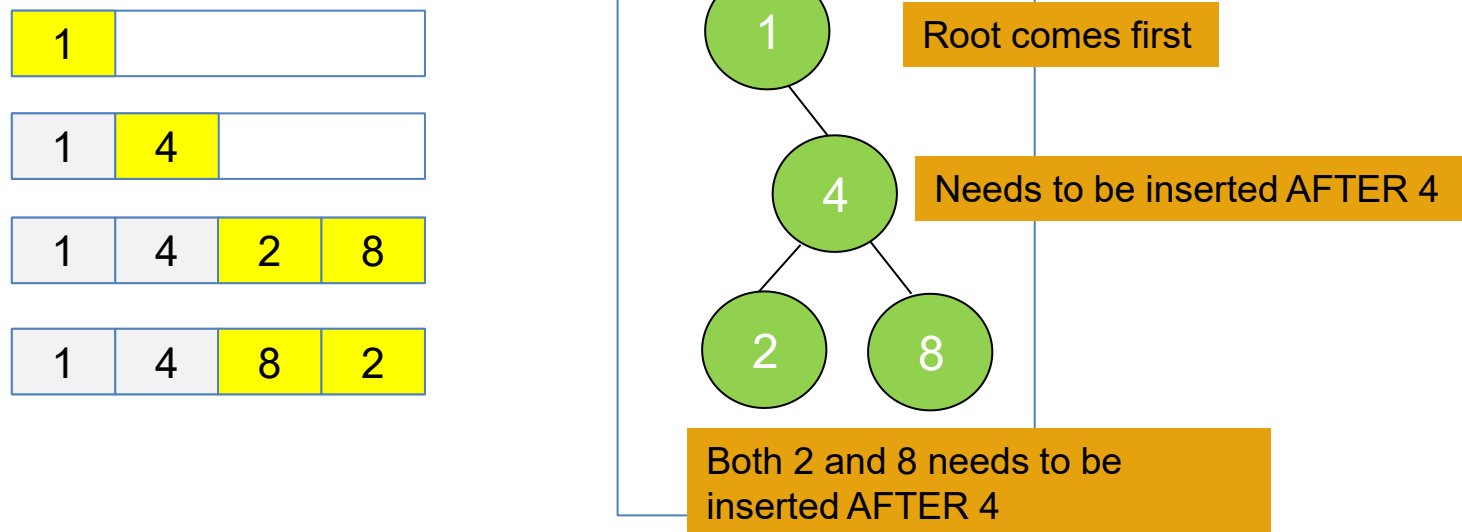


4 needs to be inserted AFTER 8

Binary Search Tree Shape (Contd.)



The order in which we put elements into a BST impacts the shape, and what you'll see is that the shape of BST will have a huge impact on the performance of operations.



Height of a BST

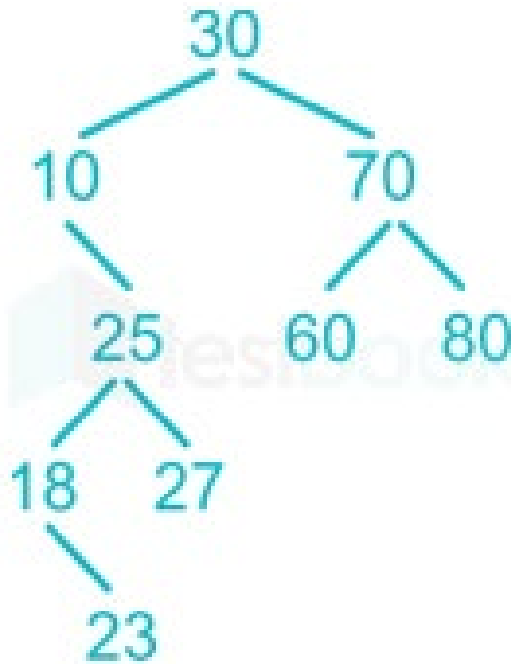
- The height of a tree is defined as the number of edges in the longest path from the root node to a leaf node.
- For a tree with only a root node, the height is 0.
- Mathematically, for a binary tree with n nodes, the height h is bounded by: $\lceil \log_2(n+1) \rceil - 1 \leq h \leq n - 1$
 - $\lceil \cdot \rceil$ is the ceiling operator. The lower bound represents a perfectly balanced tree, and the upper bound represents a degenerate tree (essentially a linked list).
 - The height of a tree is equivalent to the maximum depth of any node in the tree.
 - For a binary search tree, the minimum height with n nodes is $\lceil \log_2(n+1) \rceil - 1$, which occurs in the most balanced configuration.
 - The maximum height of a binary tree with n nodes is $n-1$, which occurs in the case of a skewed tree (essentially a linked list)

Video Tutorial

- Binary Search Trees (BST) Explained in Animated Demo
 - <https://www.youtube.com/watch?v=mtvbVLK5xDQ>

In-Order Traversal on a BST

- 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

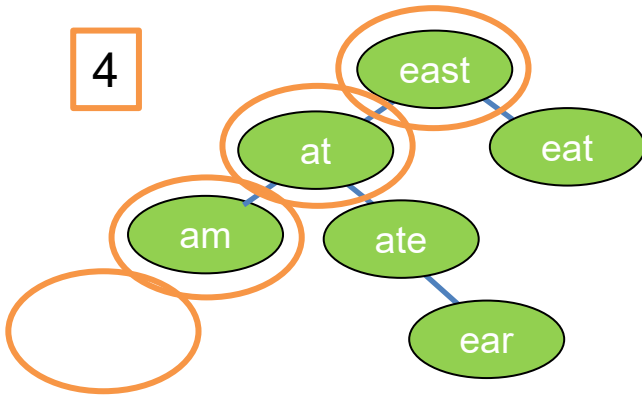
Performance Analysis of BST

Storing a dictionary as a BST

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

Structure of a BST depends on the order of insertion

4



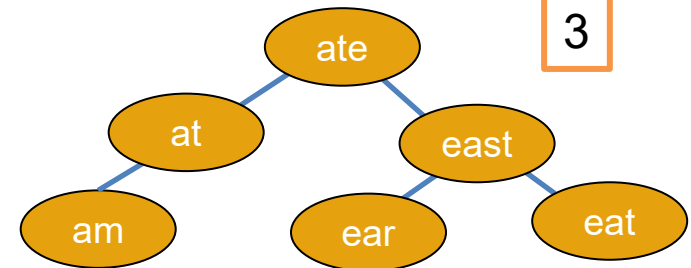
`isWord(east)`

Best case: $O(1)$

`isWord(a)`

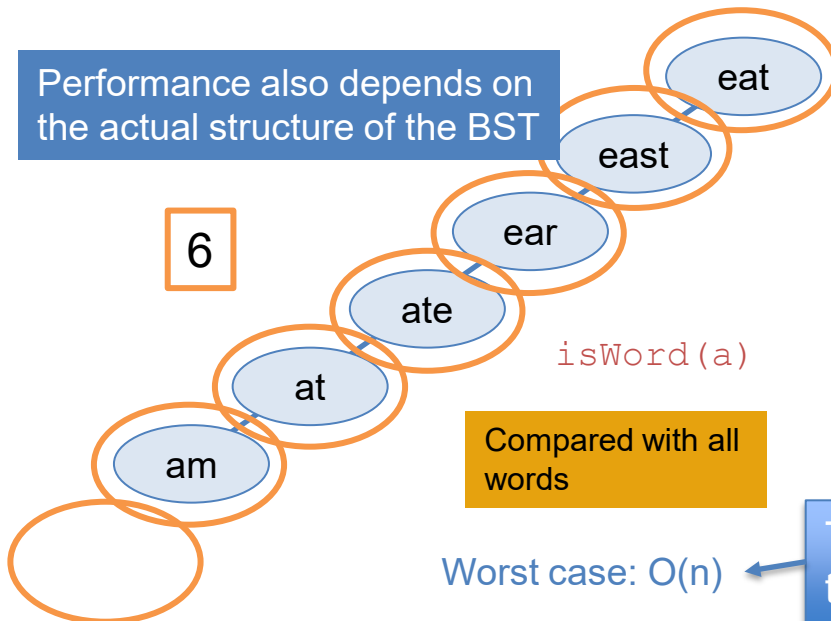
Compared with 3 out of 7 words

3



Performance also depends on the actual structure of the BST

6



`isWord(a)`

Compared with all words

Worst case: $O(n)$

How does the performance of `isWord` relate to input size n ?

`isWord(String wordToFind)`

1. Start at root
2. Compare word to current node
 1. If current node is null, return false
 2. If `wordToFind` is less than word at current node, continue searching in left subtree
 3. If `wordToFind` is greater than word at current node, continue searching in right subtree
 4. If `wordToFind` is equal to word at current node, return true

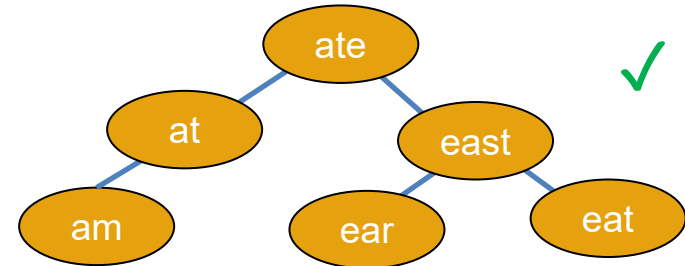
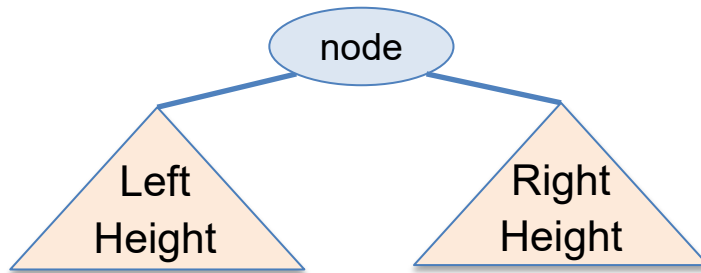
To optimize the worst case, we can modify the tree to control the max distance until leaf

height

Balanced BST

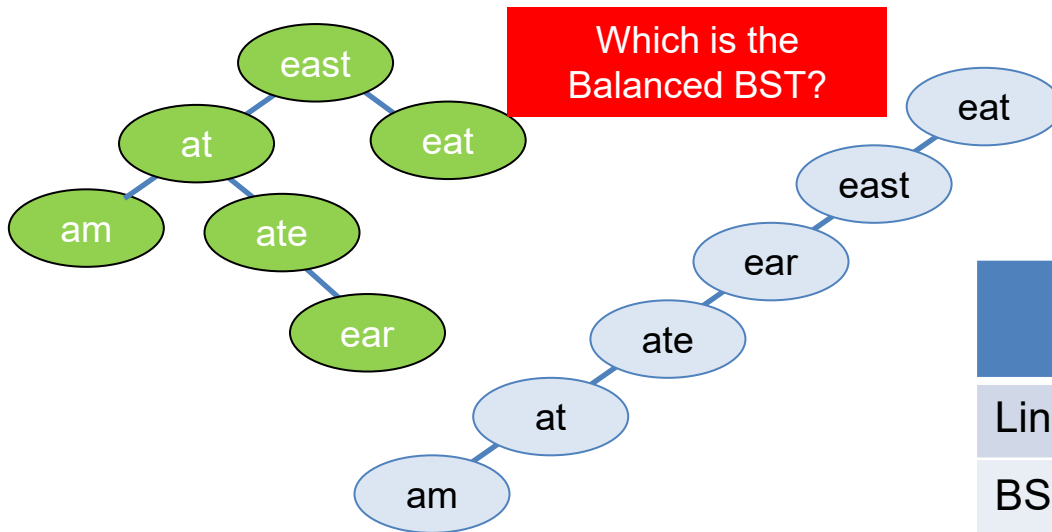
We want to keep the height down as much as we can while still maintaining the same number of nodes.

$$| \text{LeftHeight} - \text{RightHeight} | \leq 1$$



$$\text{height} \approx \log(n)$$

Especially if insert to BST in order!



	Best case	Average case	Worst case
Linked List	O(1)	O(n)	O(n)
BST	O(1)	O(log n)	O(n)
Balanced BST	O(1)	O(log n)	O(log n)

How to keep balanced? TreeSet and TreeMap in Java API

`isWord(String wordToFind)`

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 and finding closest elements.
 - Hash Tables do not inherently maintain order, making these operations more difficult.

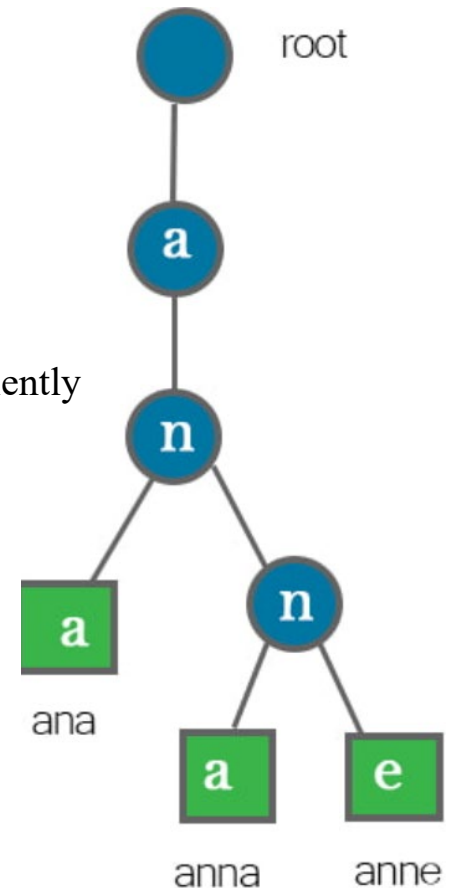
Tree vs. Trie

■ Structure and Purpose

- Trees:
 - General-purpose data structure for representing hierarchical relationships
 - Each node can contain any type of data
 - Nodes typically have a value and references to child nodes
- Tries:
 - Specialized tree structure for storing and retrieving strings efficiently
 - Also known as a prefix tree
 - Optimized for operations on strings or sequences

■ Node Content

- Trees:
 - Each node stores a value directly
- Tries:
 - Nodes typically do not store complete strings
 - The path from the root to a node represents a string or prefix
 - Characters are stored along the edges between nodes

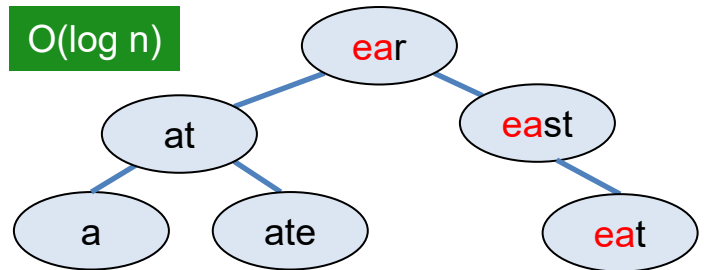


Trie Data Structure

re(TRIE)ve

Storing a dictionary as a (balanced) BST

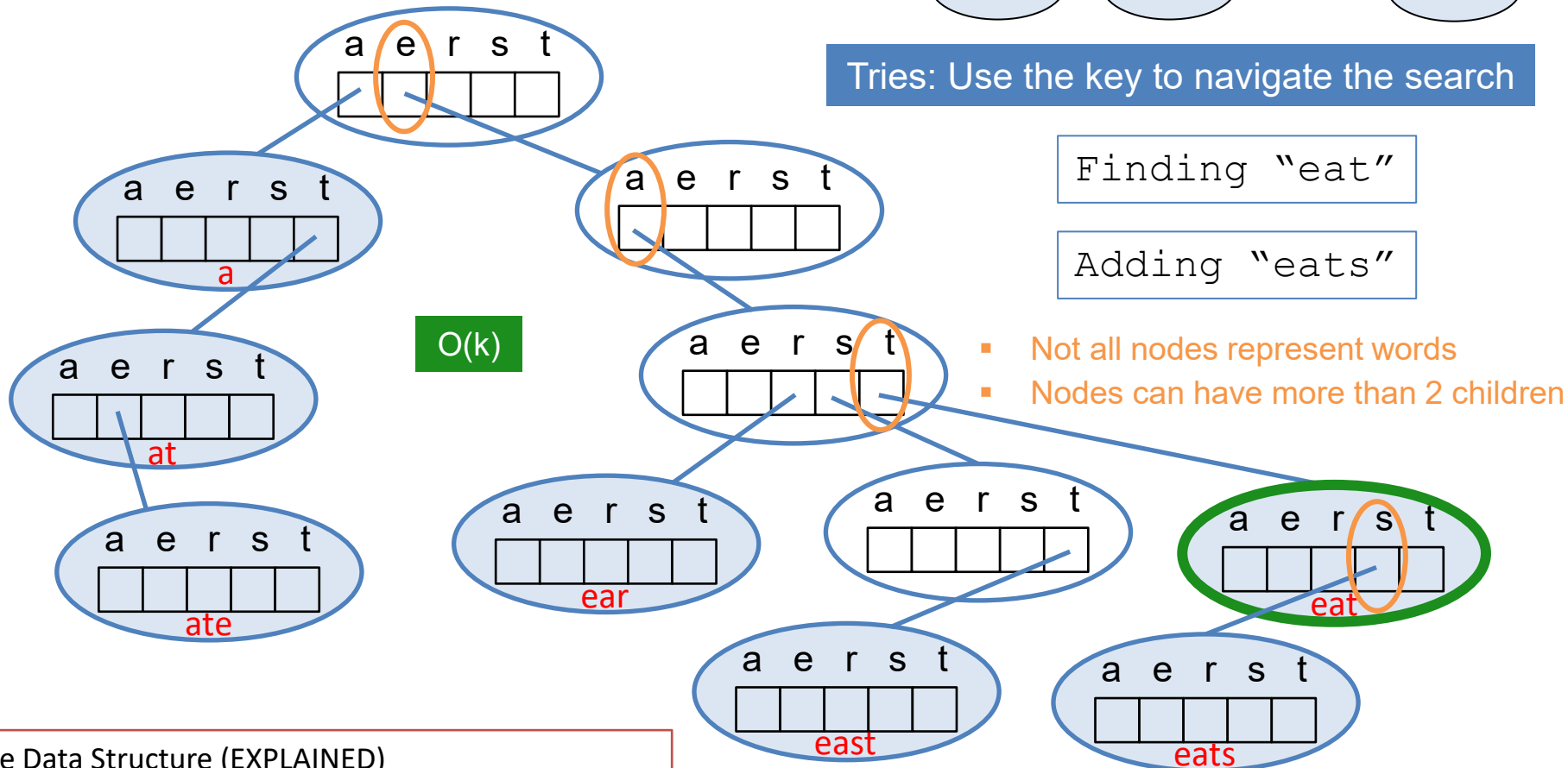
BSTs don't take advantage of shared structure



Tries: Use the key to navigate the search

Finding "eat"

Adding "eats"



Trie Data Structure (EXPLAINED)

<https://www.youtube.com/watch?v=-urNrIAQnNo>

$\log_2(250000) \approx 18$

Additional Resources

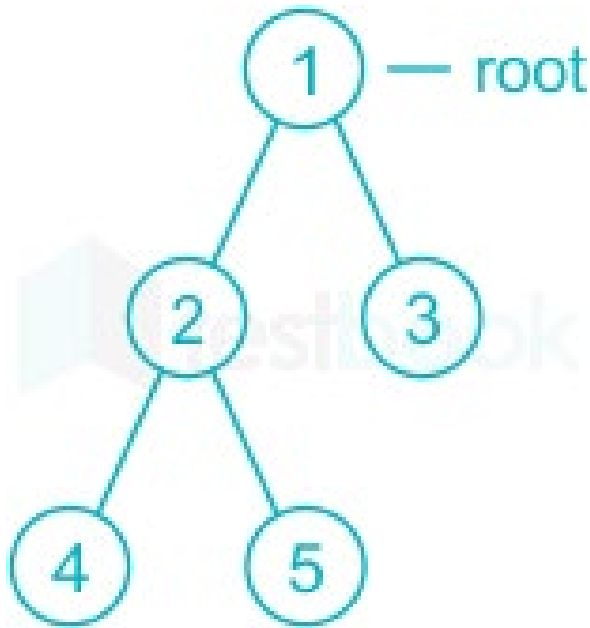
■ Trees and Binary Search Trees

- <https://www.geeksforgeeks.org/bfs-vs-dfs-binary-tree/> BFS vs DFS for Binary Tree
- <http://www.openbookproject.net/thinkcs/archive/java/english/chap17.htm> -- explains trees, how to build and traverse it
- <http://algs4.cs.princeton.edu/32bst/> -- about binary search trees
- Data structures: Binary Search Tree
 - https://www.youtube.com/watch?v=pYT9F8_LFTM

■ Tries

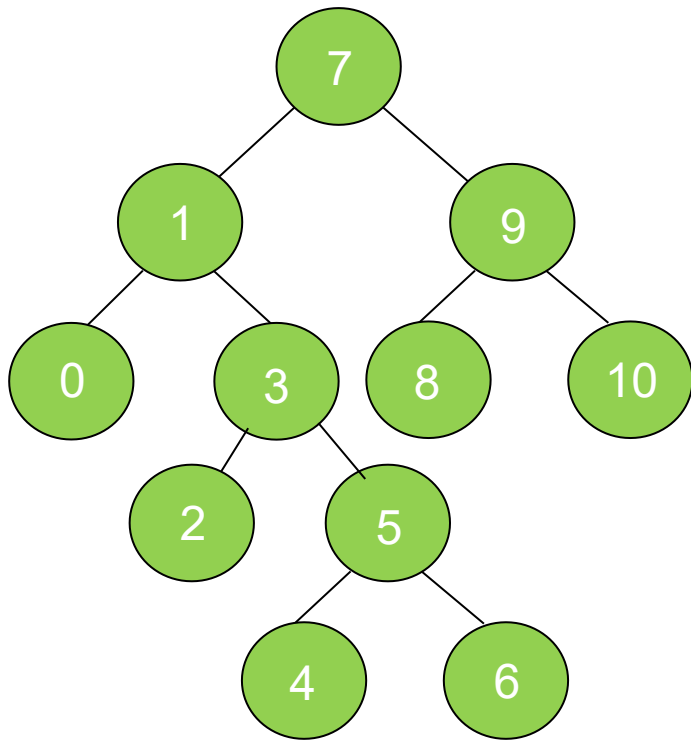
- <https://www.toptal.com/java/the-trie-a-neglected-data-structure> -- explains with solid example
- <https://www.topcoder.com/community/data-science/data-science-tutorials/using-tries/> -- explains as well as providing code

Quiz



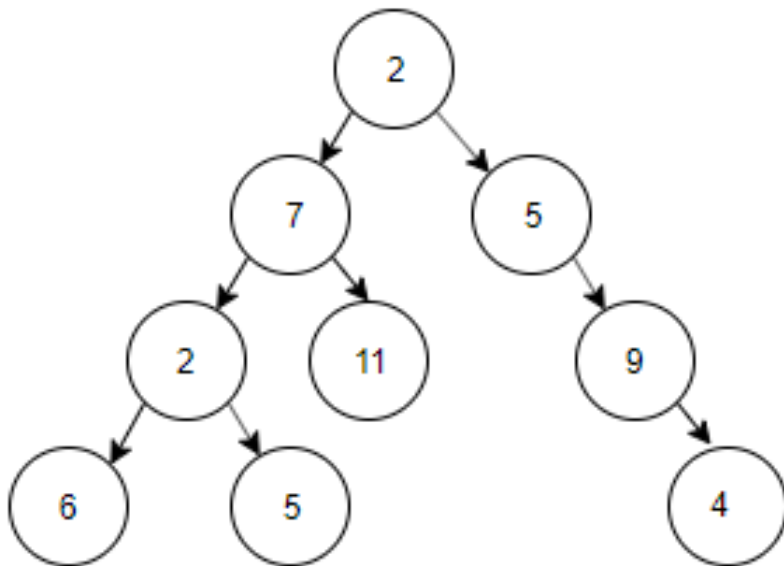
- **Pre-order traversal:**
- Traversal sequence: 12453
- **In-order traversal:**
- Traversal Sequence: 42513
- **Post-order traversal:**
- Traversal sequence: 45231

Quiz



- **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 with the root (**7**)
- Traversal sequence: 0, 2, 4, 6, 5, 3, 1, 8, 10, 9, 7

Quiz



- **Pre-order traversal:**
- Traversal sequence: 2, 7, 2, 6, 5, 11, 5, 9, 4
- **In-order traversal:**
- Traversal Sequence: 6, 2, 5, 7, 11, 2, 5, 9, 4
- **Post-order traversal:**
- Traversal sequence: 6, 5, 2, 11, 7, 4, 9, 5, 2

Quiz

- 1. ****What is the sequence of nodes visited in a Preorder traversal?****
- - a) Left, Root, Right
- - b) Root, Left, Right
- - c) Left, Right, Root
- - d) Right, Root, Left

ANS: b

- 2. ****Which traversal method is used to get nodes of a Binary Search Tree in ascending order?****
- - a) Preorder
- - b) Inorder
- - c) Postorder
- - d) Level order

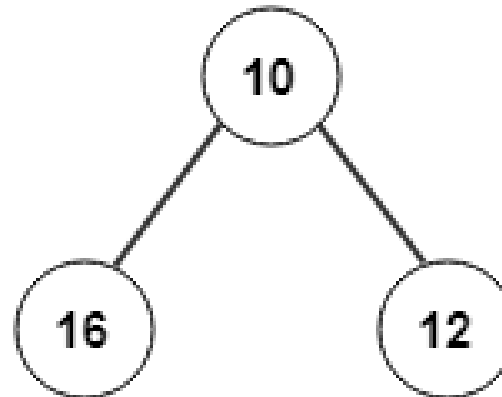
ANS: c

- 3. ****In a Postorder traversal, when is the root node visited?****
- - a) First
- - b) After visiting the left subtree
- - c) After visiting both left and right subtrees
- - d) Before visiting any subtree

ANS: c

Quiz

- The following given tree is an example for?
- a) Binary tree
- b) Binary search tree
- c) Fibonacci tree
- d) AVL tree



ANS: a

Explanation: The given tree is an example for binary tree since has got two children and the left and right children do not satisfy binary search tree's property, Fibonacci and AVL tree.

Quiz

- How many common operations are performed in a binary tree?
- a) 1
- b) 2
- c) 3
- d) 4

ANS: c

Explanation: Three common operations are performed in a binary tree- they are insertion, deletion and traversal.

Quiz

- When searching for the key value 60 in a binary search tree, nodes containing the key values 10, 20, 40, 50, 70 80, 90 are traversed, not necessarily in the order given. How many different orders are possible in which these key values can occur on the search path from the root to the node containing the value 60?