Data Structures

Arthur Hoskey, Ph.D.
Farmingdale State College
Computer Systems Department

Today's Lecture

- Binary Search Trees

Goals

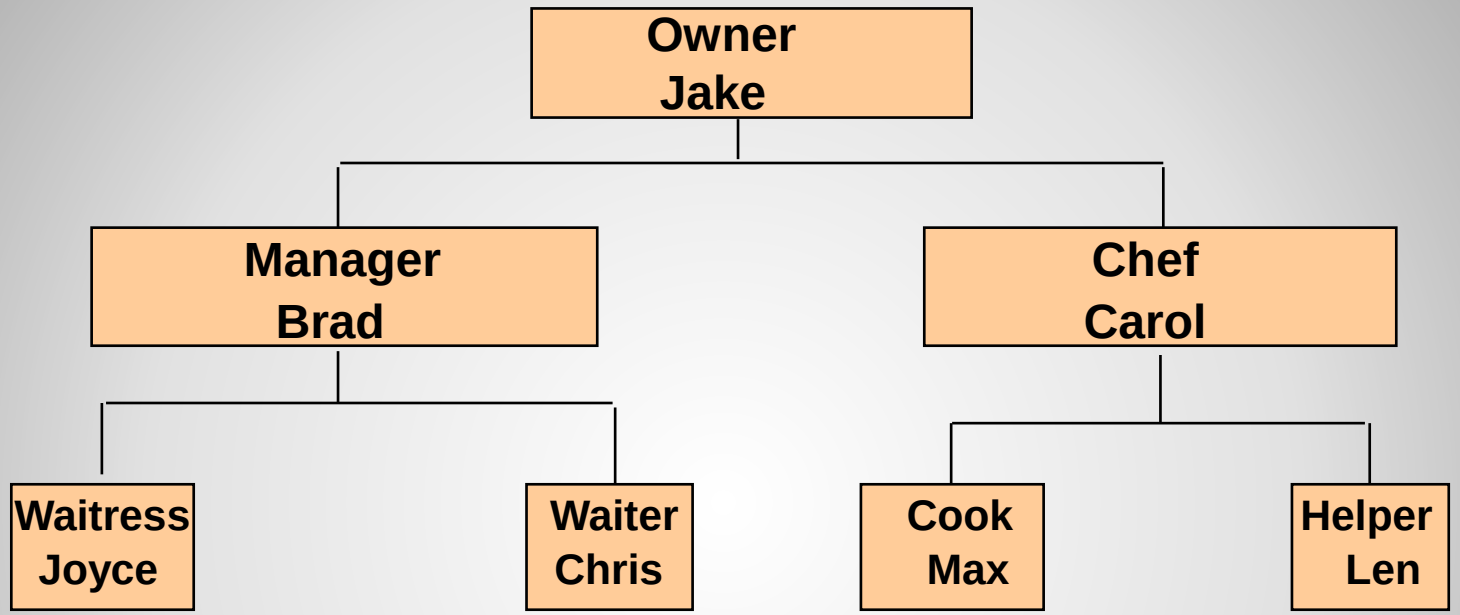
- Define and use the following terminology:
  binary tree root descendant
  subtree binary search tree
  parent level ancestor
  child height
- Define a binary search tree at the logical level
- Show what a binary search tree would look like after a series of insertions and deletions

Goals

- Implement the following binary search tree algorithms in C++
  - Inserting an element
  - Deleting an element
  - Retrieving an element
  - Modifying an element
  - Copying a tree
  - Traversing a tree in preorder, inorder, postorder

Goals

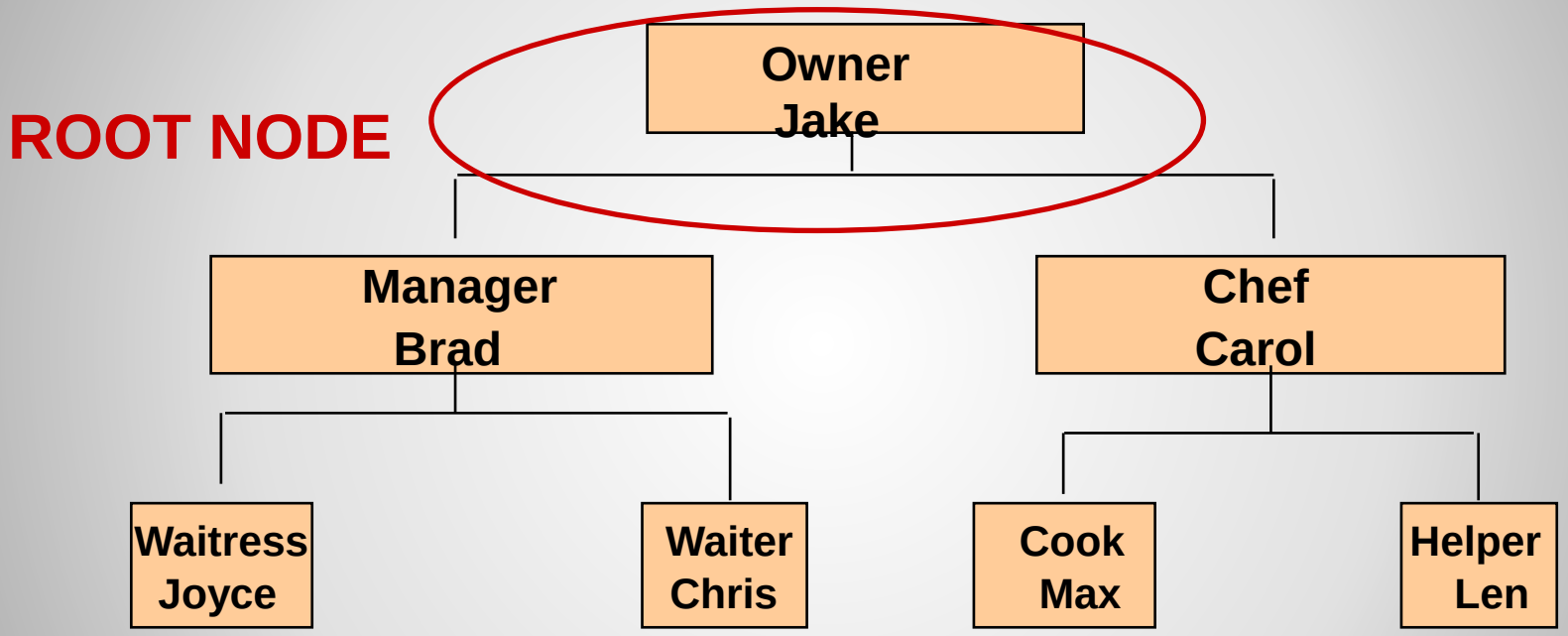
- Discuss the Big-O efficiency of a given binary search tree operation
- Describe an algorithm for balancing a binary search tree
- Show how a binary tree can be represented in an array, with implicit positional links between the elements
- Define the terms full binary tree and complete binary tree

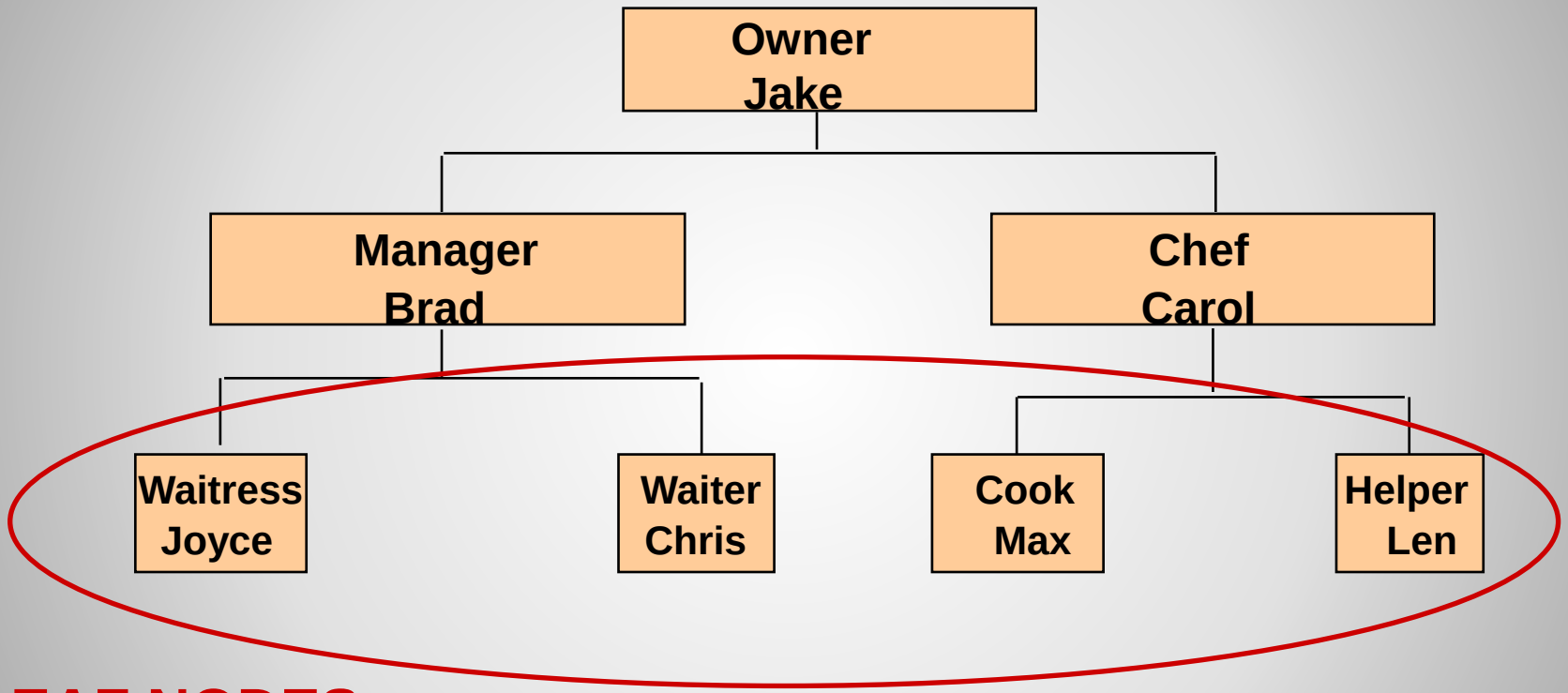


Jake's Pizza Shop

Trees

Trees

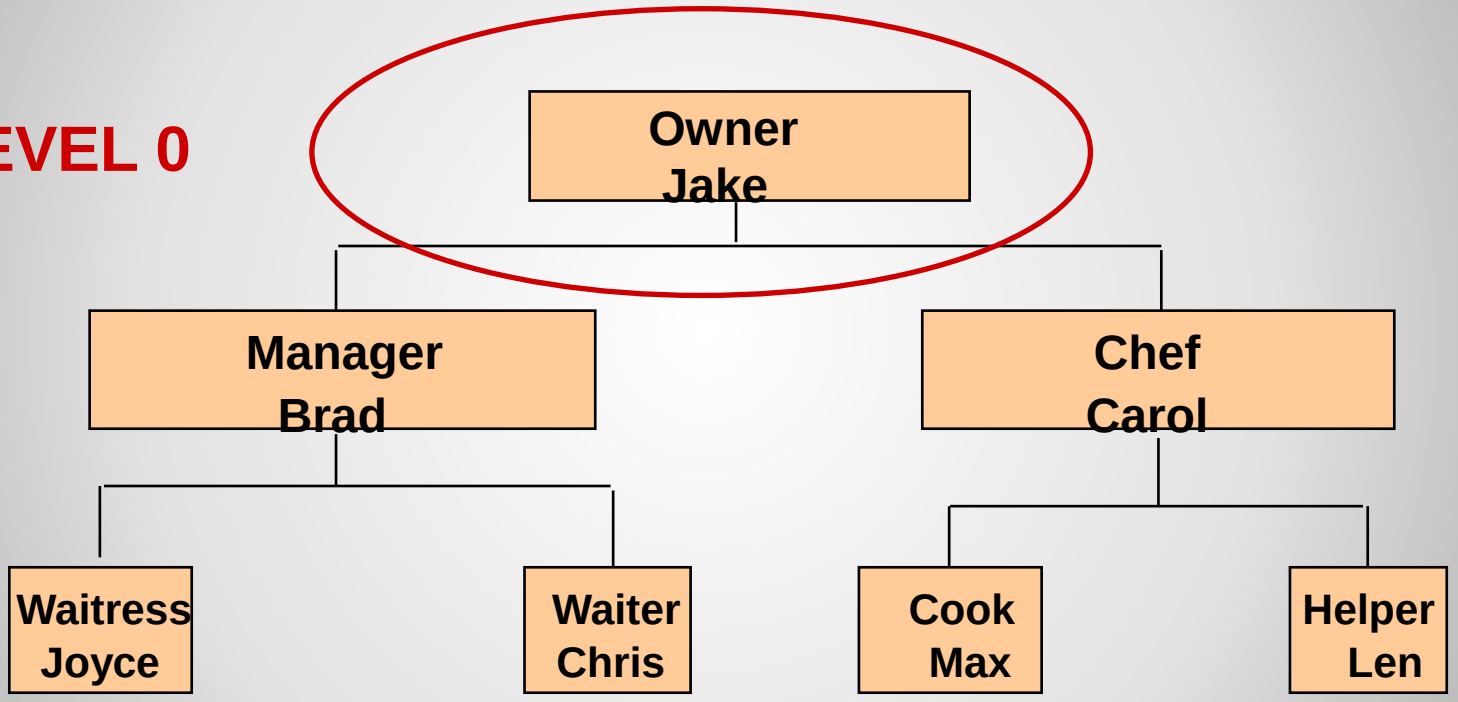




LEAF NODES

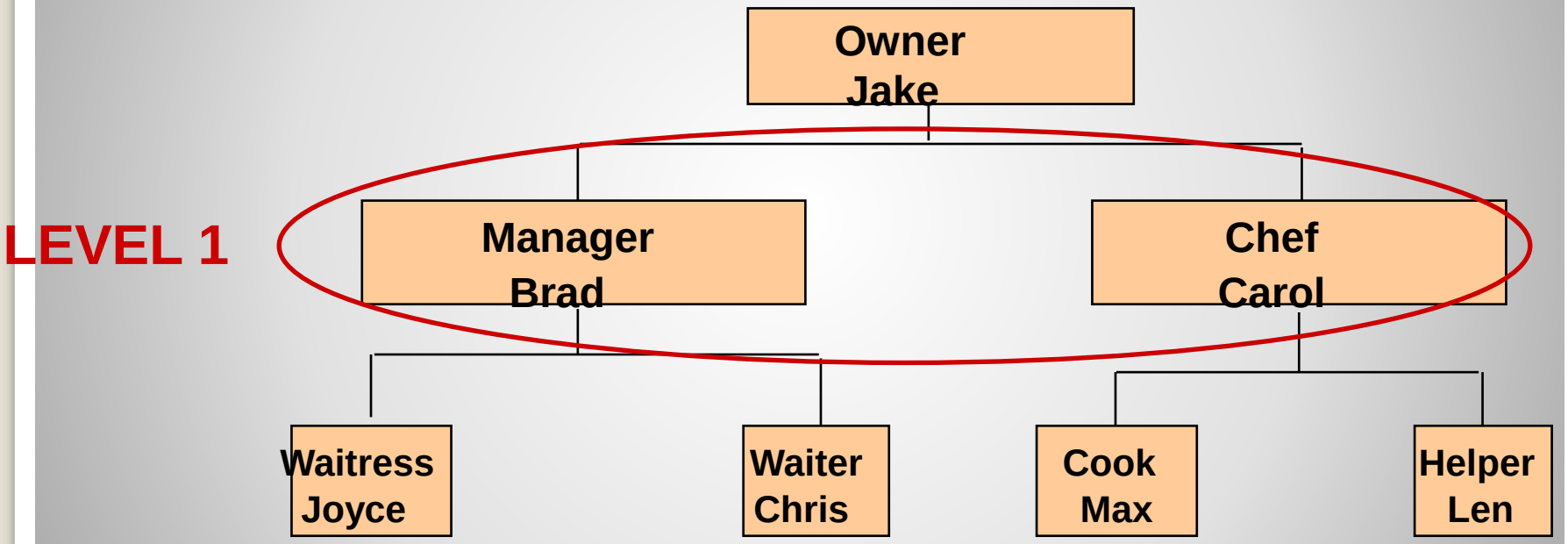
Trees

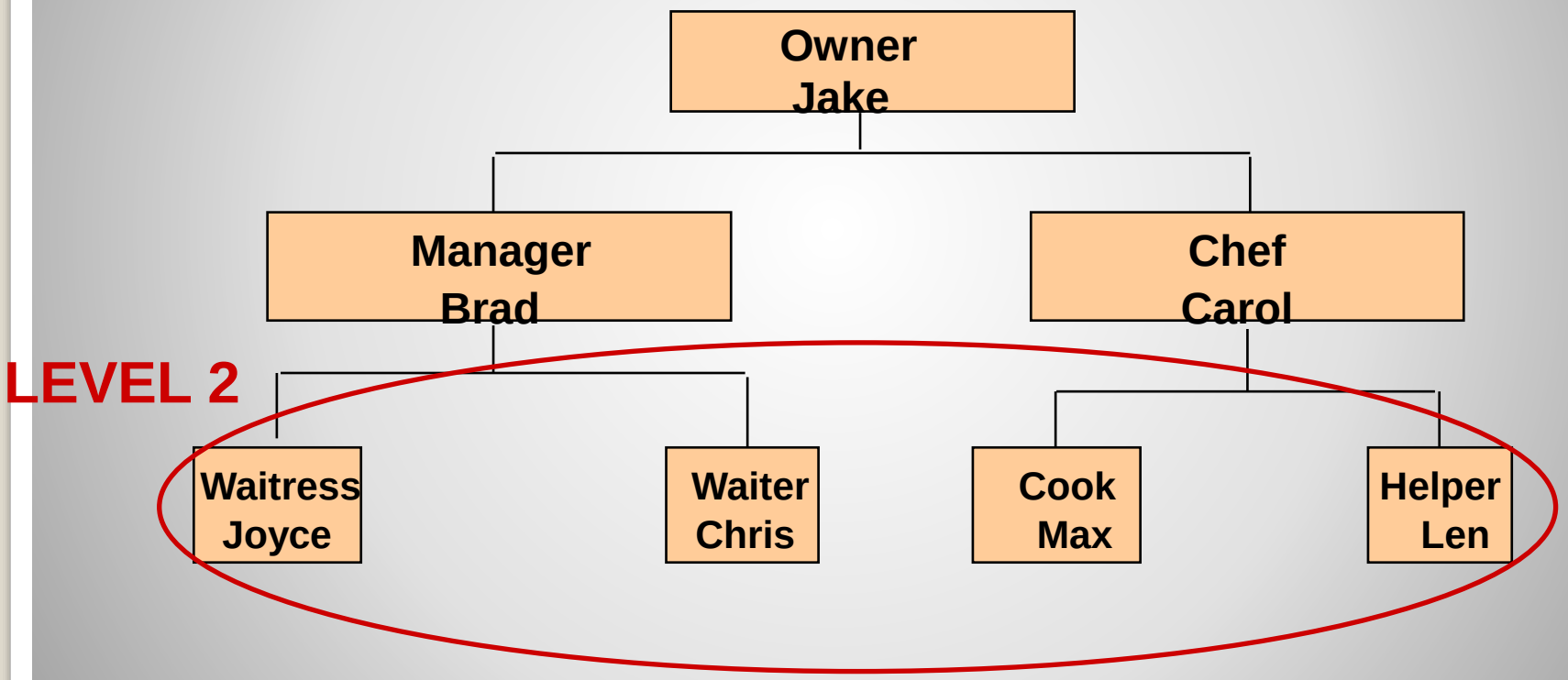
LEVEL 0



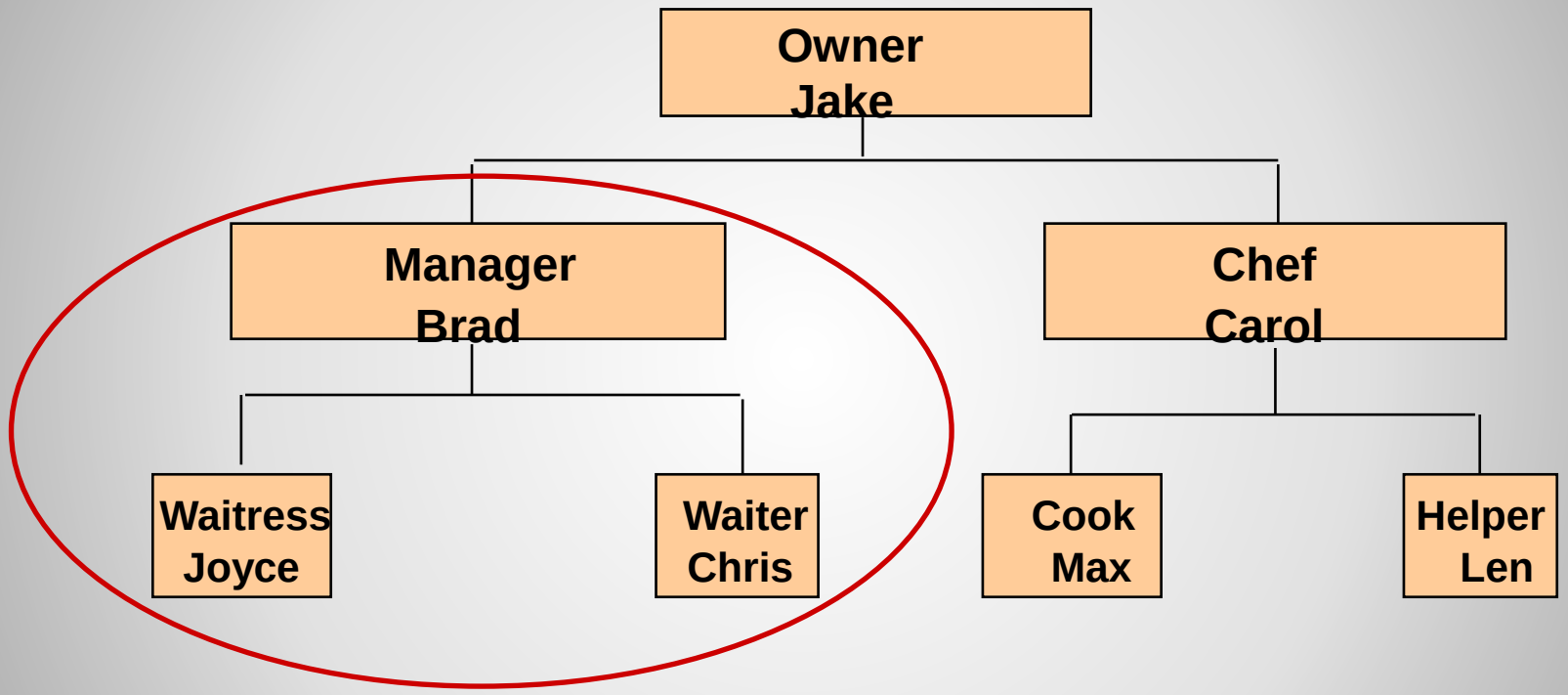
Trees

Trees



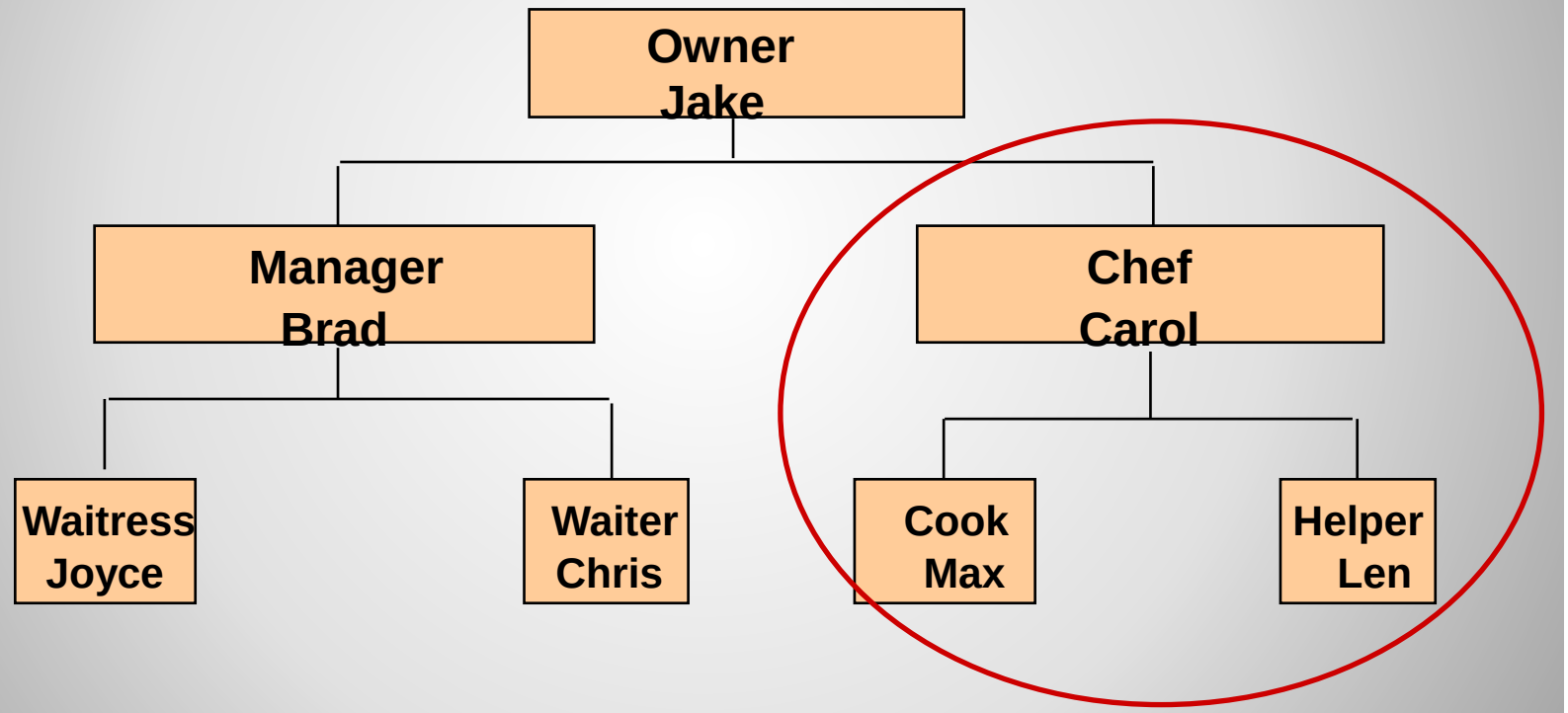


Trees



LEFT SUBTREE OF ROOT NODE

Trees



Trees

RIGHT SUBTREE OF ROOT NODE

Binary Tree

A structure with a unique starting node (the root), in which each node is capable of having two child nodes and a unique path exists from the root to every other node

Root

The top node of a tree structure; a node with no parent

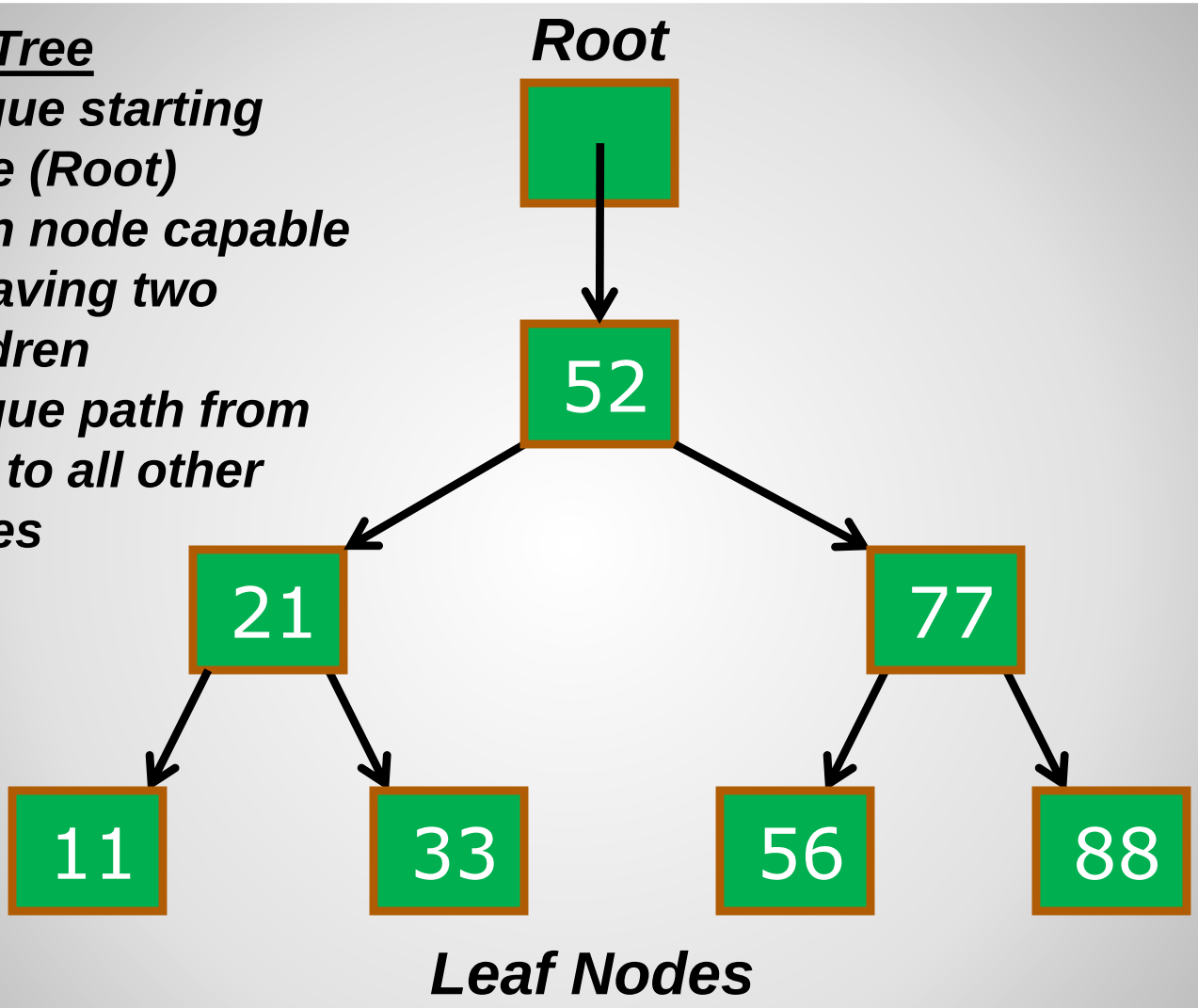
Leaf Node

A tree node that has no children

Binary Tree

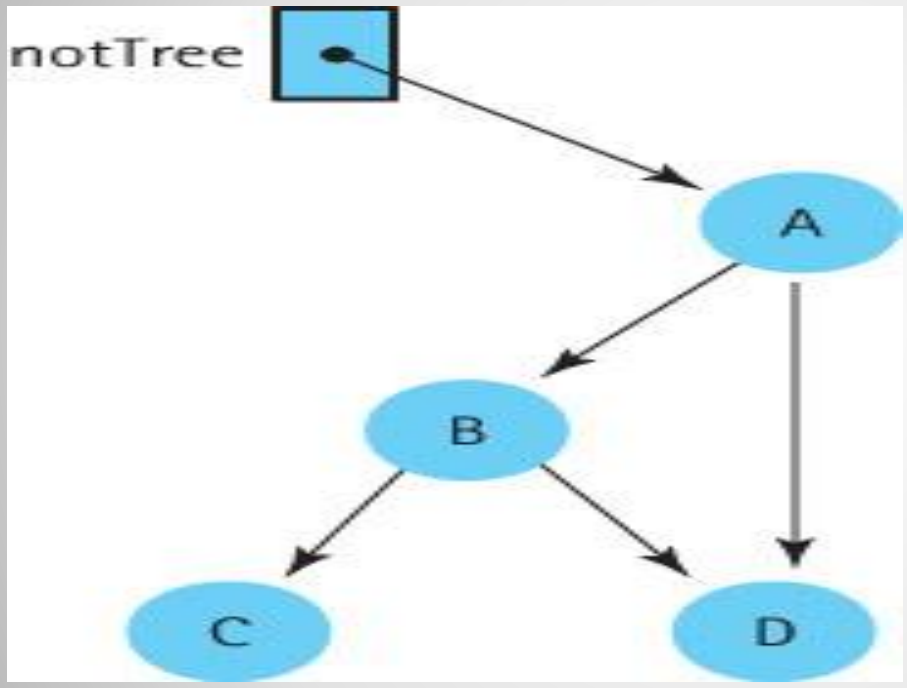
Binary Tree

1. ***Unique starting node (Root)***
2. ***Each node capable of having two children***
3. ***Unique path from root to all other nodes***

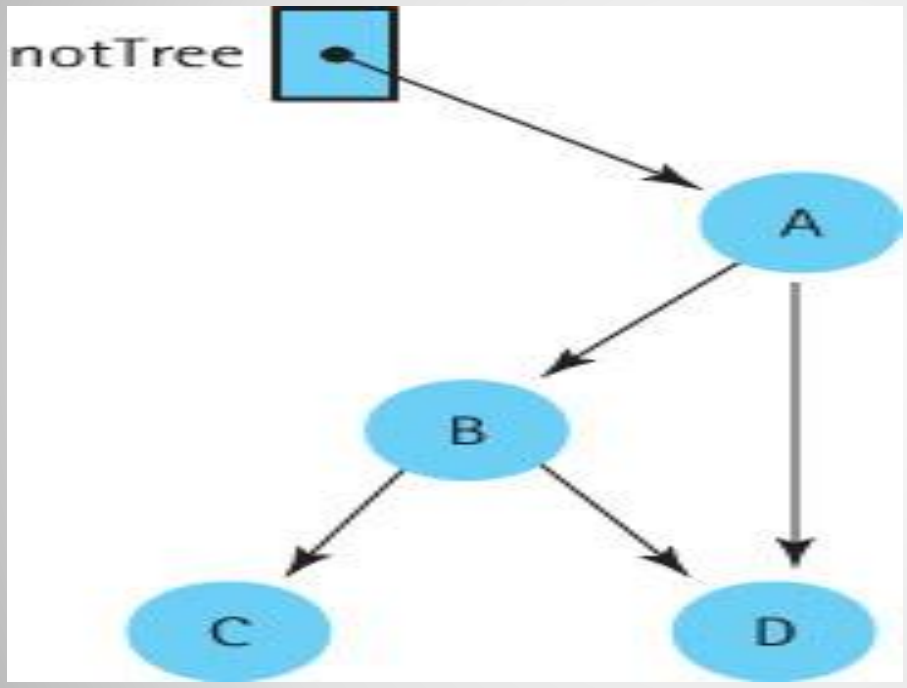


Binary Tree

Trees



Why is this not a tree ?



Why is this not a tree ?

Answer:
No unique path from root to item D

Trees

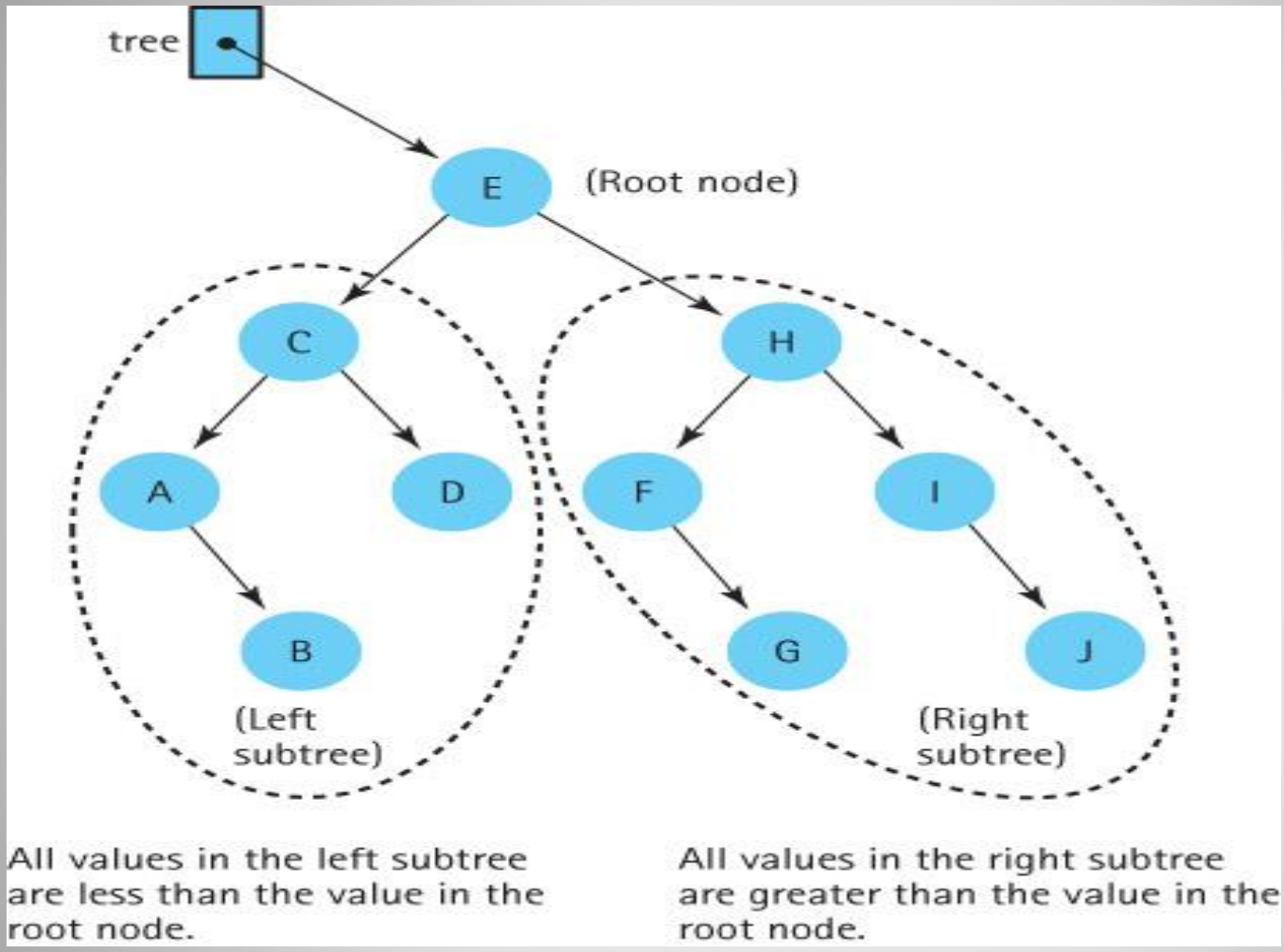
Binary Search Trees (BST)

Search Property

A binary tree in which the key value in any node is greater than the key value in the left child and any of its children and less than the key value in the right child and any of its children

A BST is a binary tree with the search property.

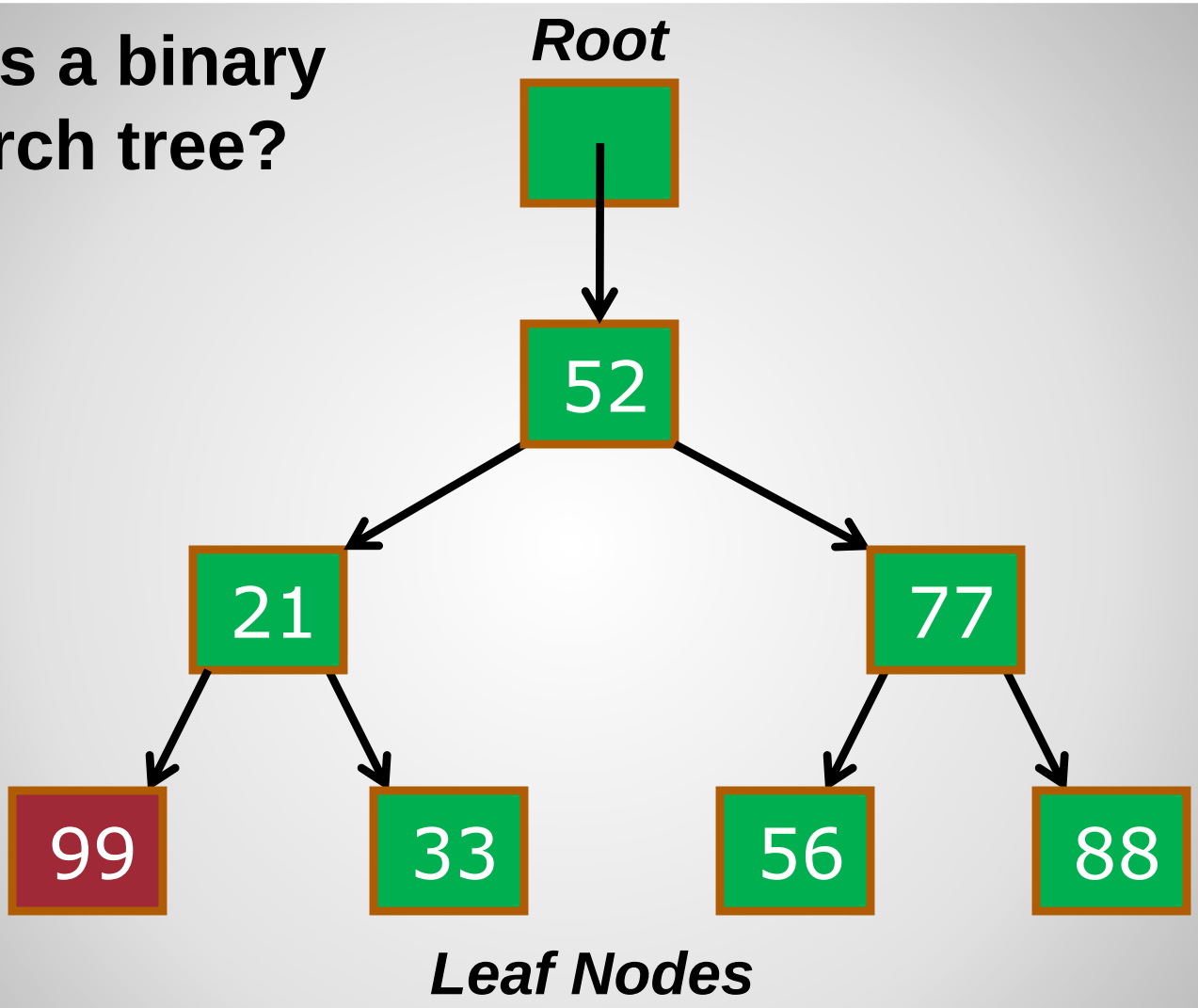
Binary Search Tree



Each node is the root of a subtree

Binary Search Trees

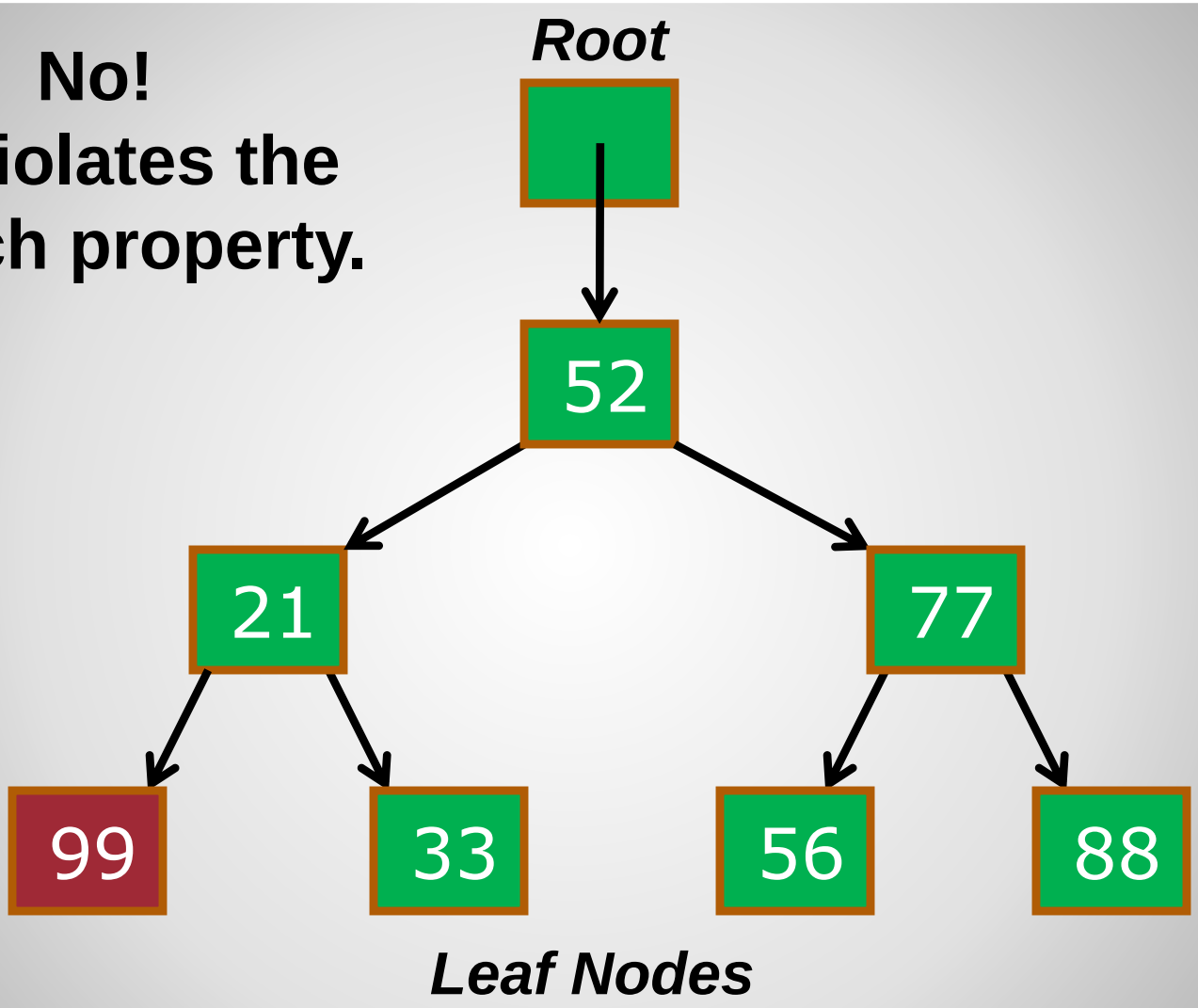
Is this a binary search tree?



Binary Search Tree

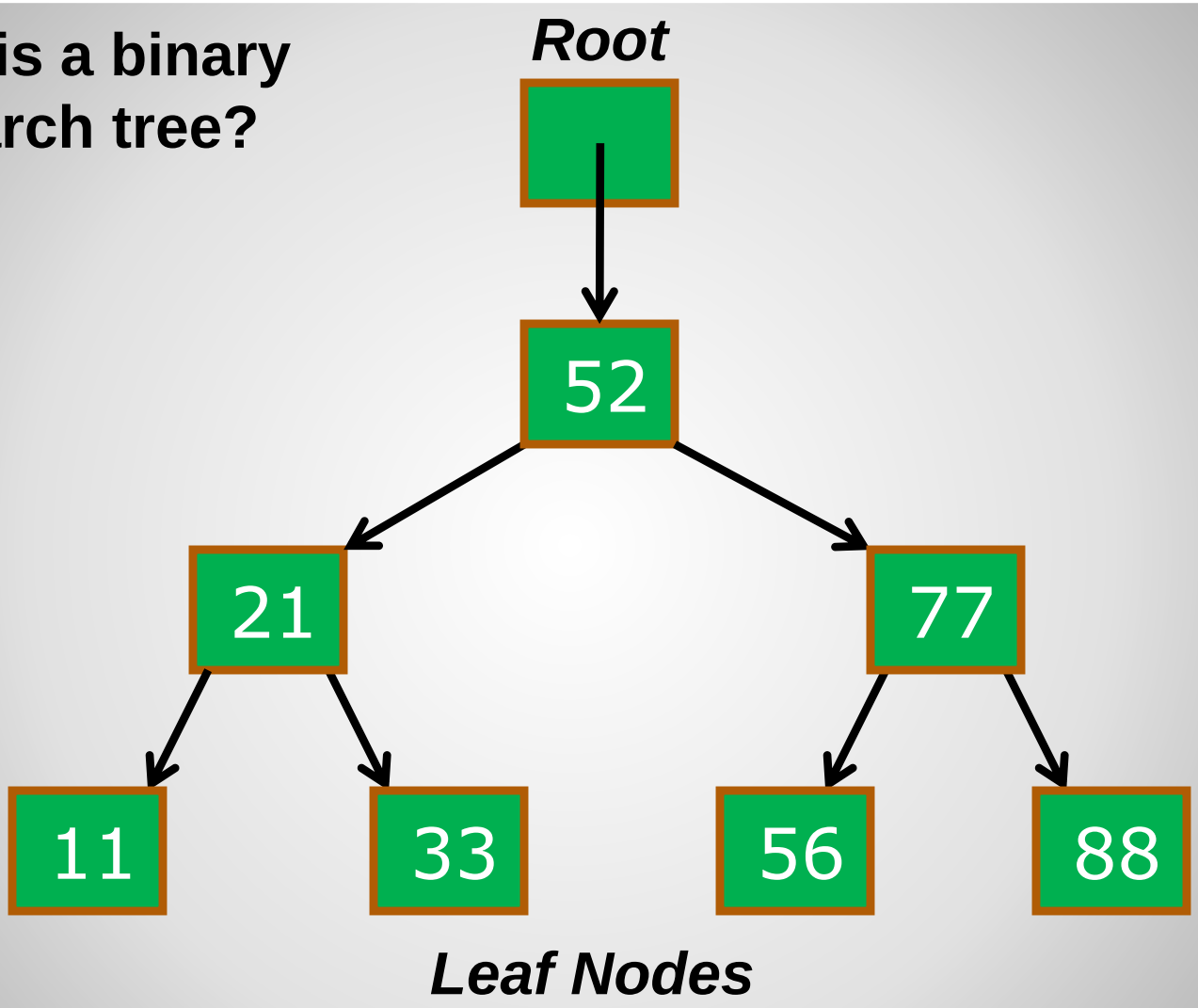
No!

99 Violates the search property.



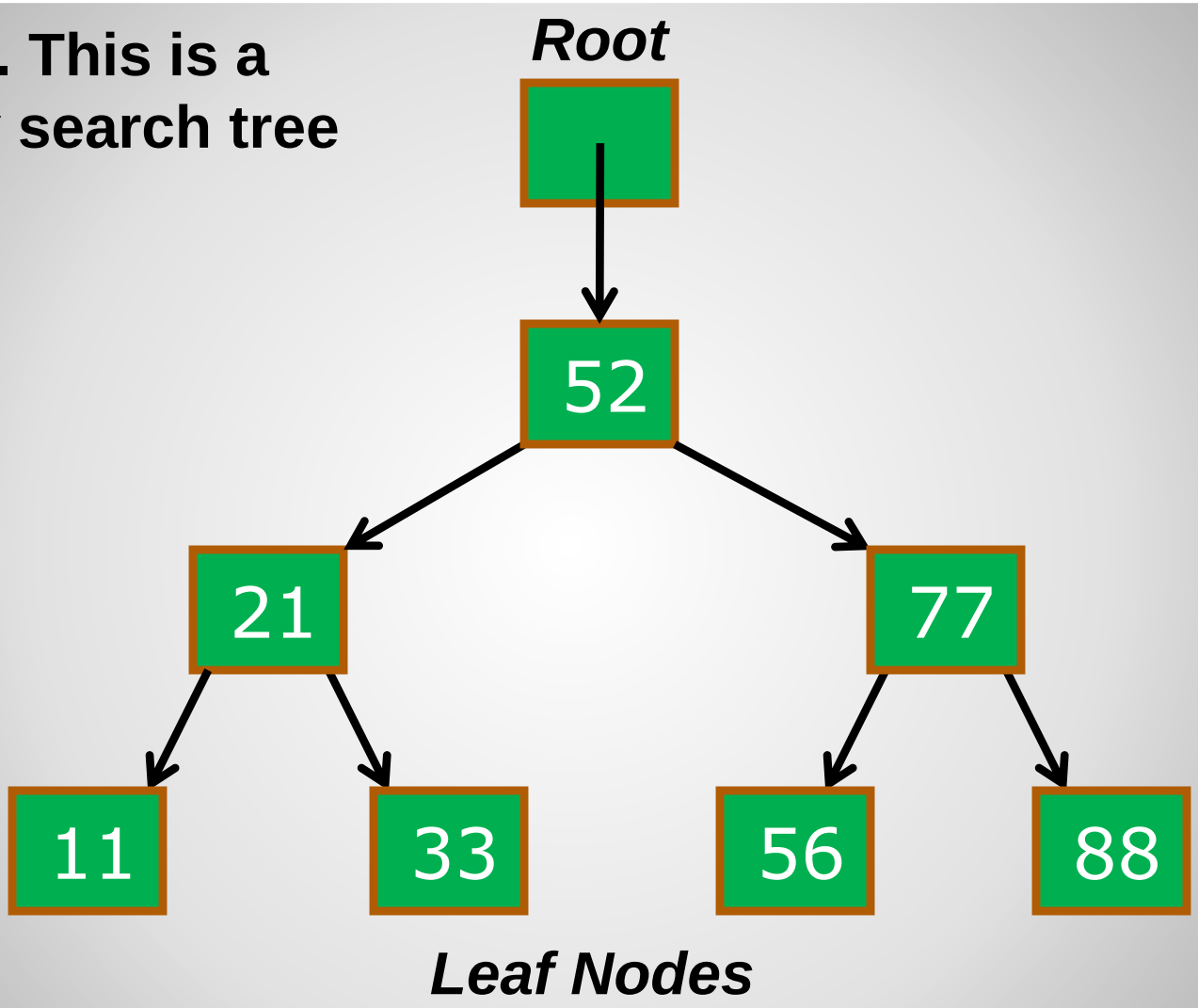
Binary Search Tree

Is this a binary search tree?



Binary Search Tree

Yes. This is a binary search tree



Binary Search Tree

Level

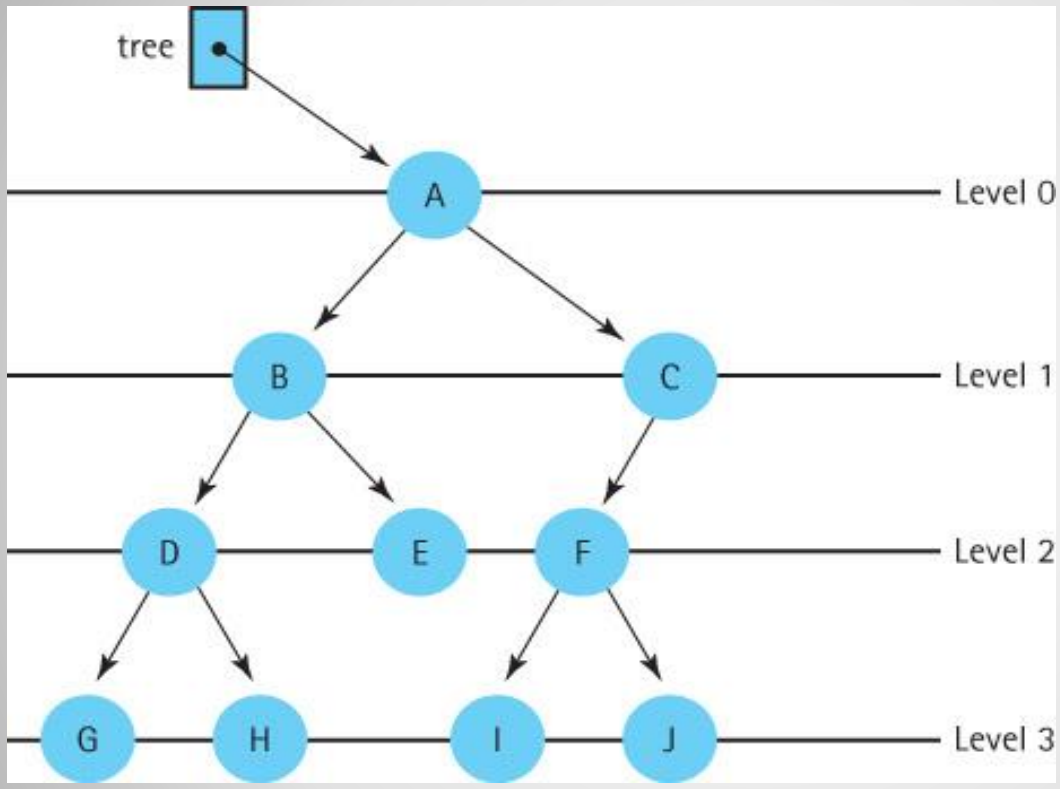
The distance of a node from the root; the root is level 0.

Height

The maximum level.

Binary Search Tree

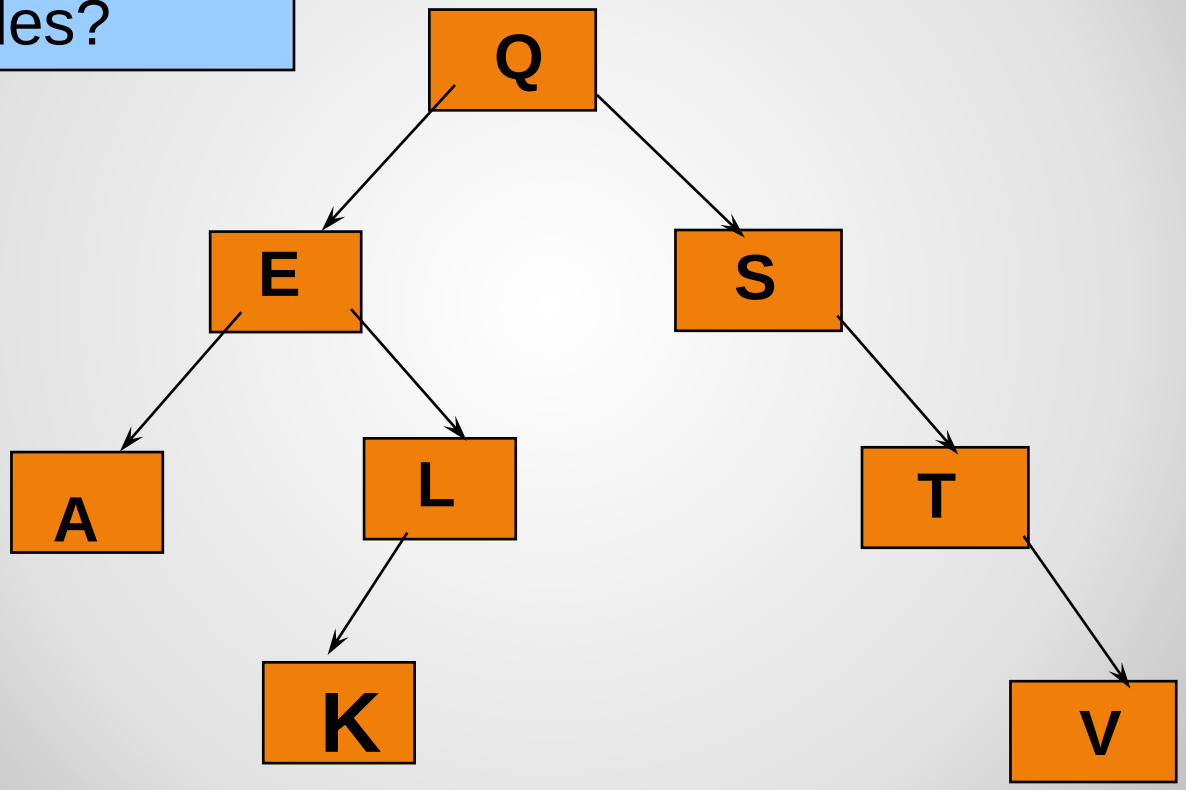
Trees



Level
Distance of a node from the root

Height
The maximum level

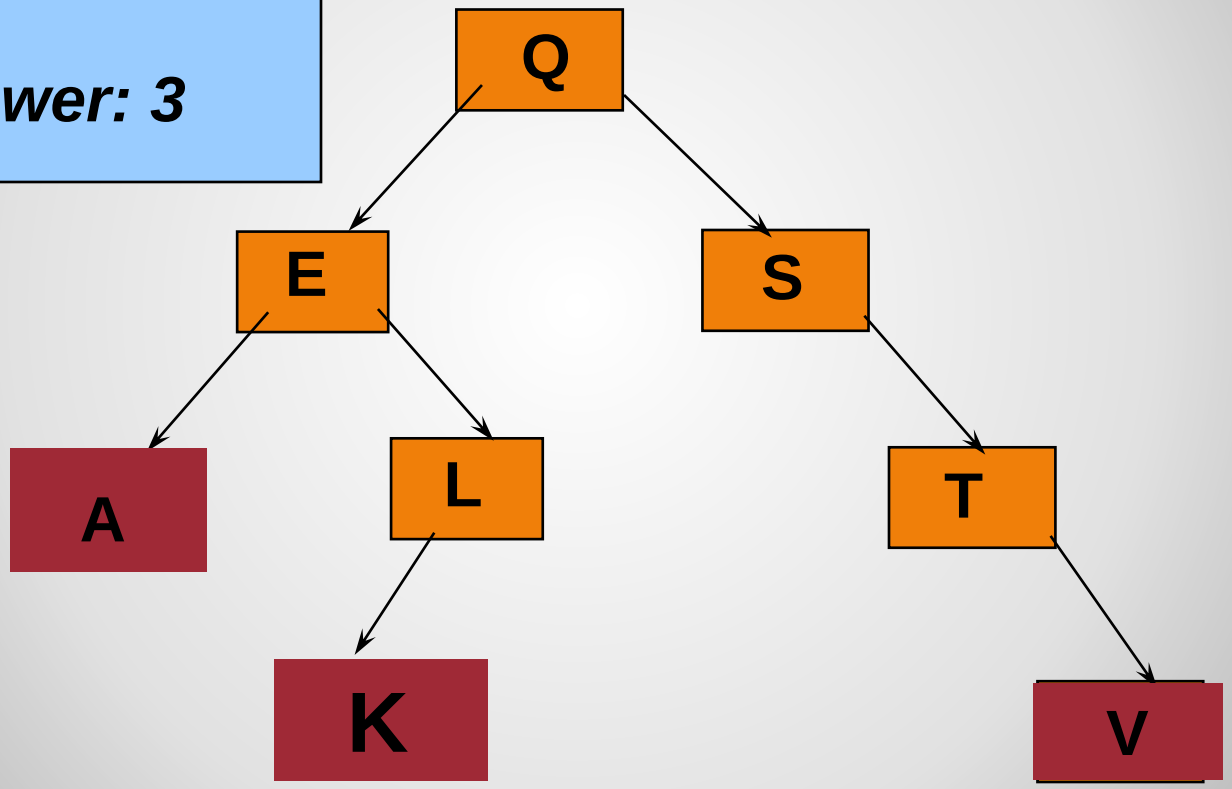
How many leaf nodes?



Trees

How many leaf nodes?

Answer: 3



Trees

Descendants

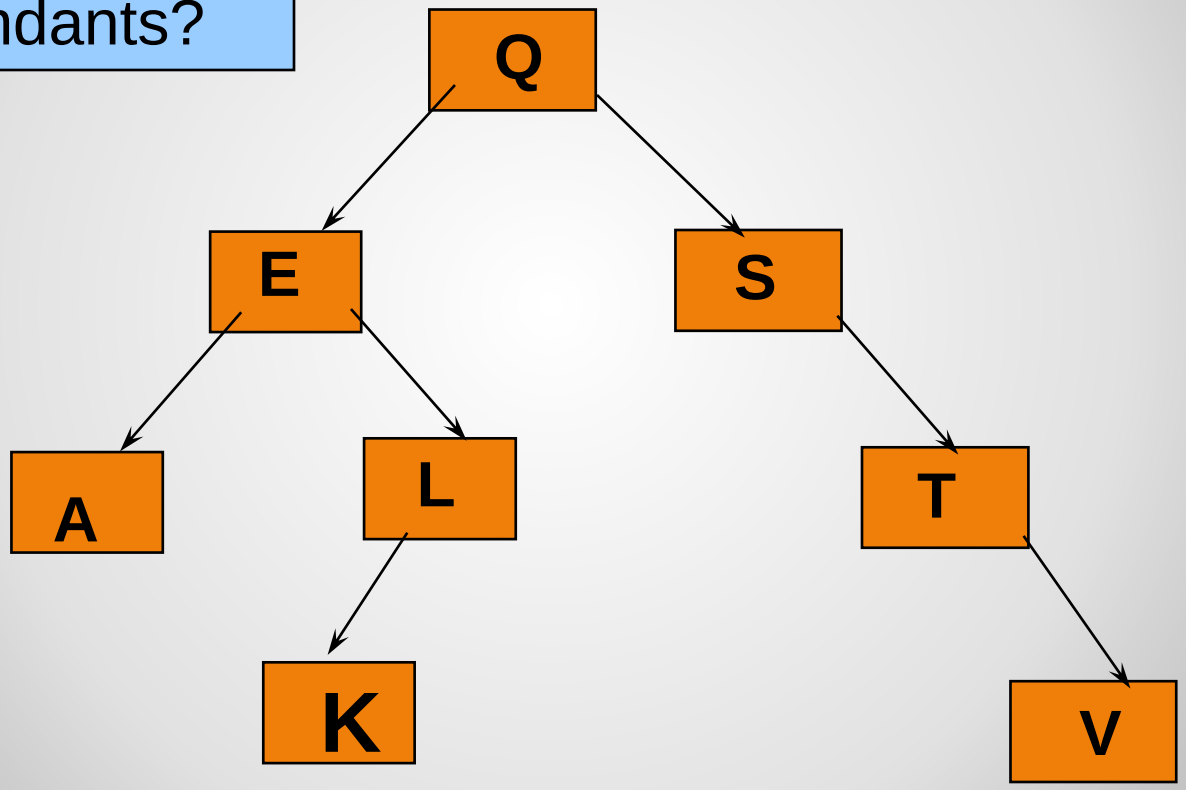
These are the children of a given node. For any given node, say n, the descendants of n are all of the children of n and all of the children of the children of n and so on.

Ancestors

A node is the ancestor of another node if it is the parent of that node, or the parent of some other ancestor of that node.

Trees

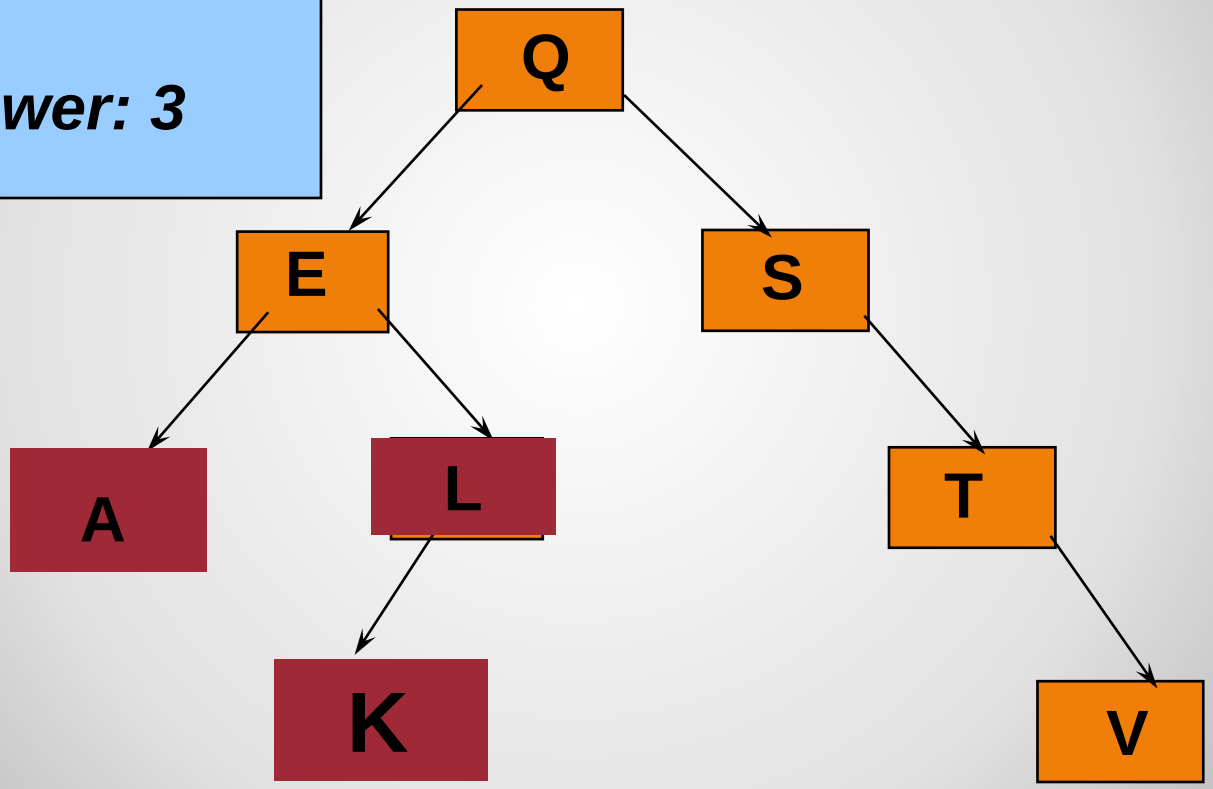
E has how many descendants?



Trees

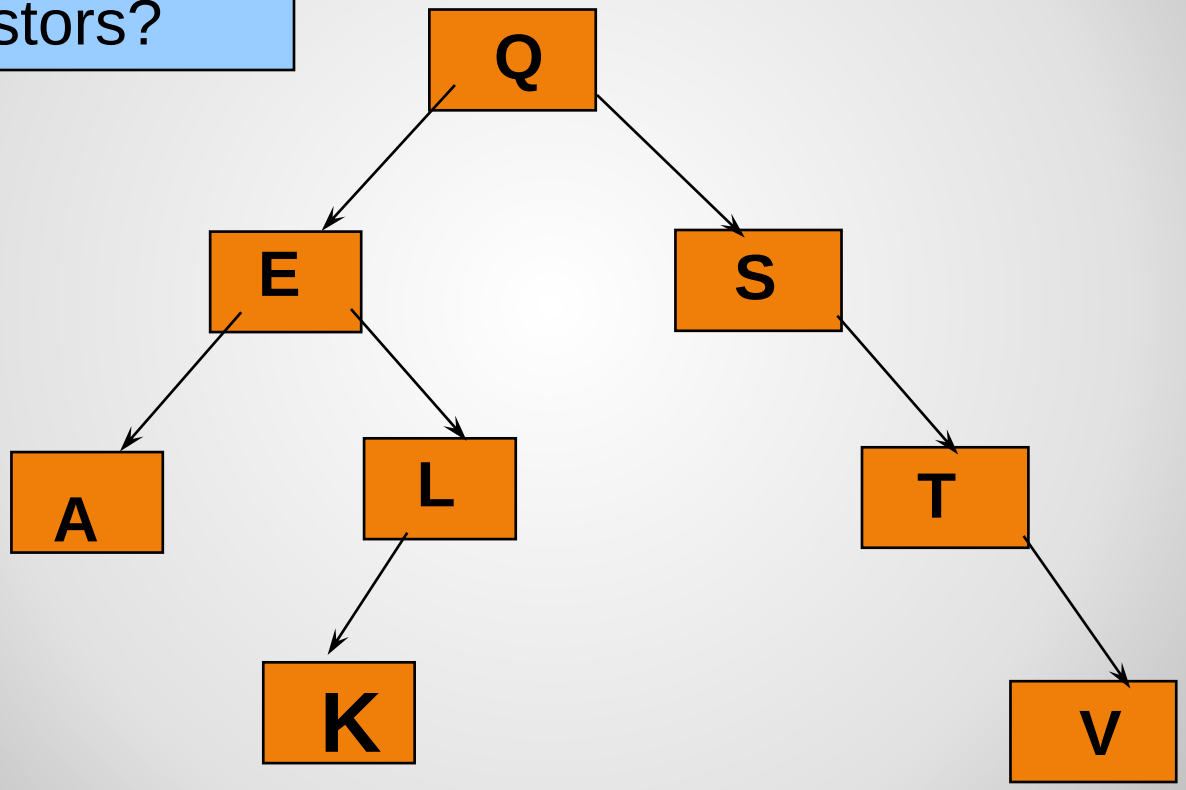
E has how many descendants?

Answer: 3



Trees

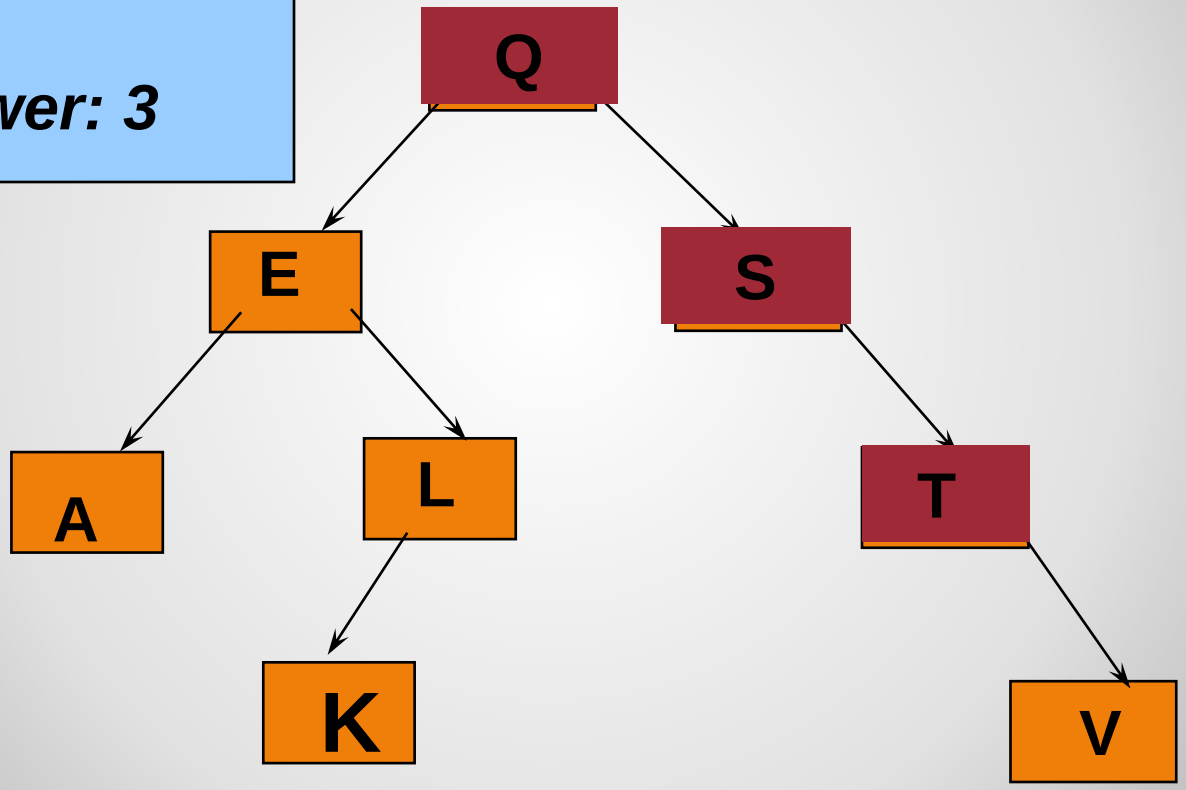
V has how many ancestors?



Trees

V has how many ancestors?

Answer: 3

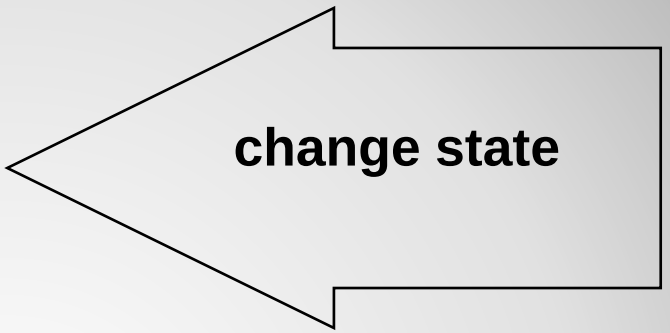


Trees

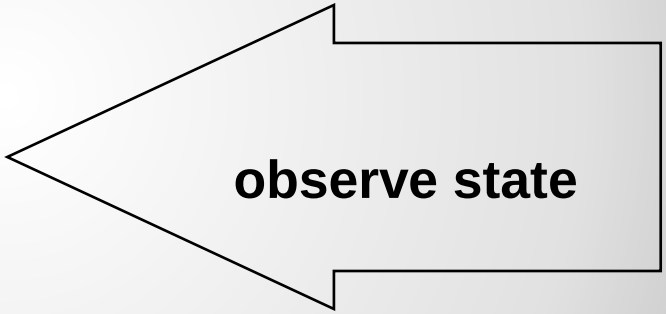
What operations would be appropriate for a binary search tree?

Binary Search Tree

- **Transformers**
  - *InsertItem*
  - *DeleteItem*



- **Observers**
  - *IsEmpty*
  - *IsFull*
  - *GetLength*
  - *RetrieveItem*
  - *Print*



Binary Search Tree

- What member variables do we need for a linked implementation?

Binary Search Tree (linked)

- Here is the Tree Interface we will be using:

```
public interface Tree {
    public void insertItem(int item);
    public void deleteItem(int item);
    public boolean hasItem(int target);
    public int retrieveItem(int target) throws Exception;
    public void makeEmpty();
    public int getLength();
}
```

Tree Interface

BinarySearchTree Class

- We will write a BinarySearchTree class that implements our Tree interface.

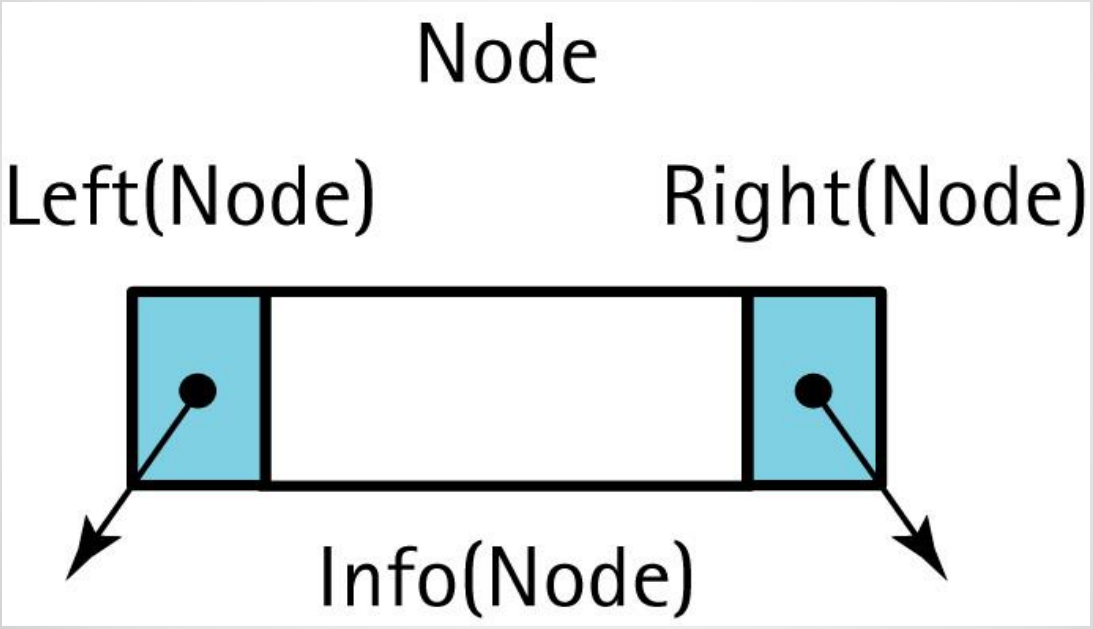
```
public class BinarySearchTree implements Tree
{
   // Implementation code goes here
}
```

Node

- The binary search tree data structure requires that we keep more information at EACH place inside of it.
- Each item in the tree will be a "Node" (not just the data).
- A node stores the data and a reference to the next node
- It should be defined as an inner class within the binary search tree class.

```
class Node {
   Declare int data
   Declare Node left
   Declare Node right
}
```

- Picture of a tree node



Node

- Tree **private** members

```
public class BinarySearchTree implements Tree {
    Declare Node root

    // Public members go here…
}
```

BinarySearchTree Class Member Variables

```
getLength() returns int
   return getLength(root)
```

Call the recursive getLength

What should the access modifiers be for each getLength?

```
getLength(Node tree) returns int
    if (tree equals null)
        return 0

    return getLength(tree.left) + getLength(tree.right) + 1
```

1. Get left subtree length

2. Get right subtree length

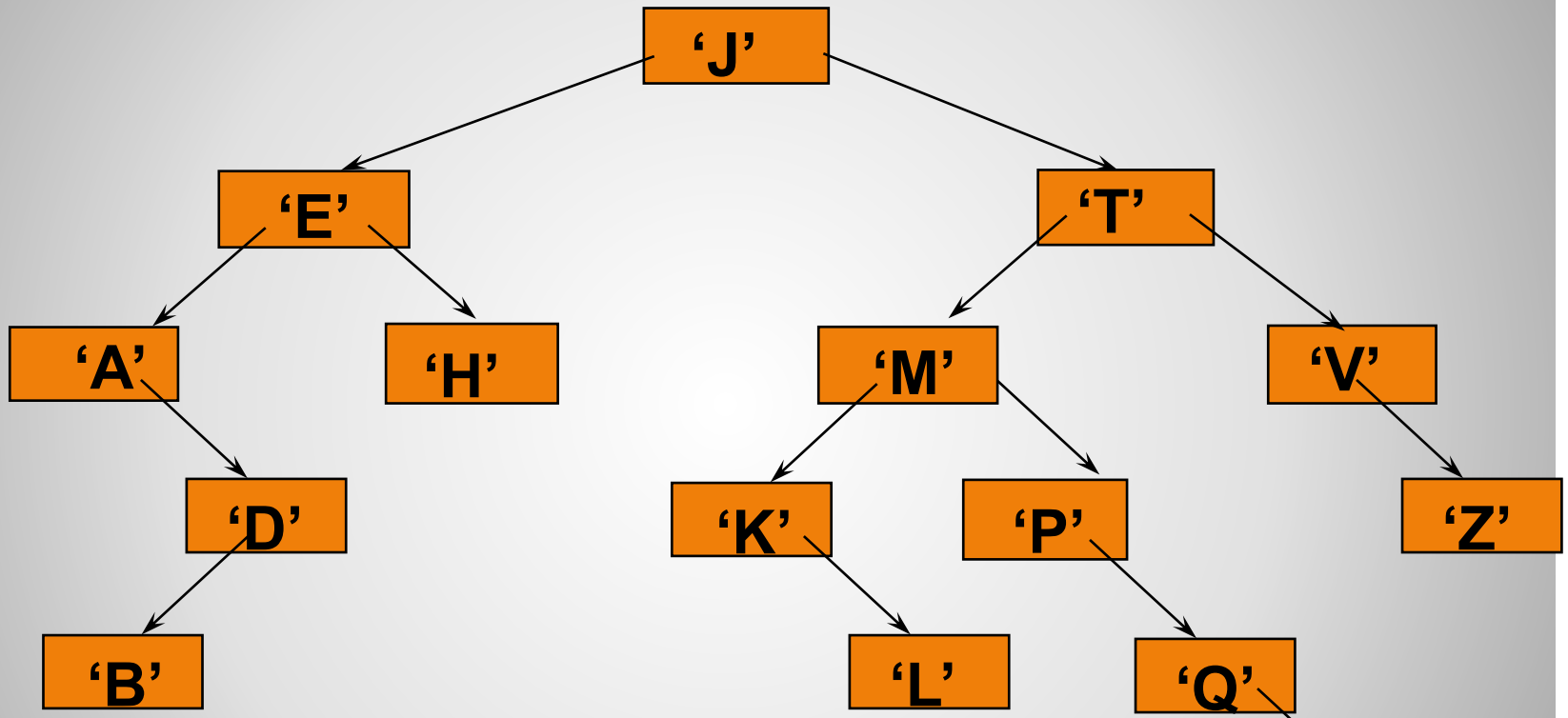
3. Add 1 for the current node

Binary Search Tree (linked)

- Now we will walk through searching for items in the tree…

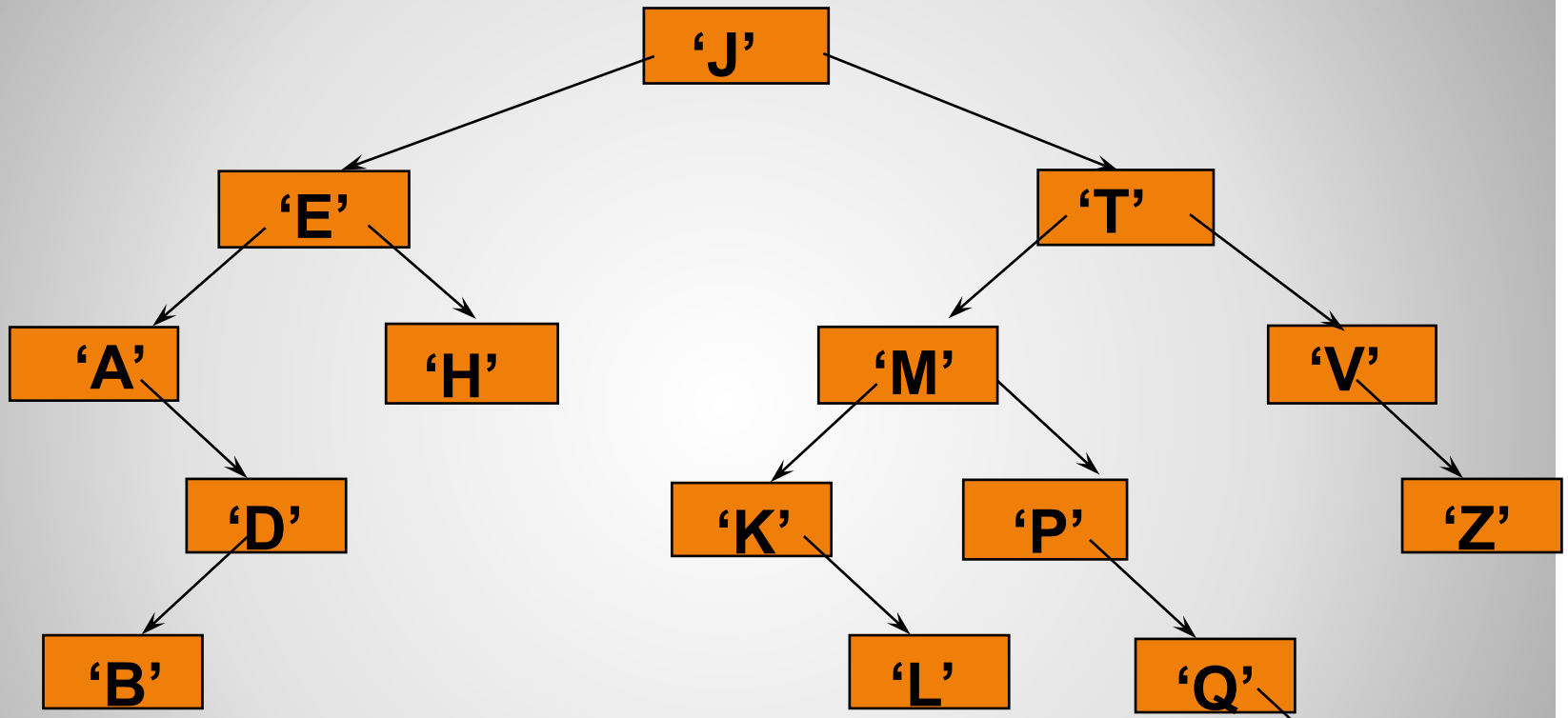
Recursive Search

Recursive Search



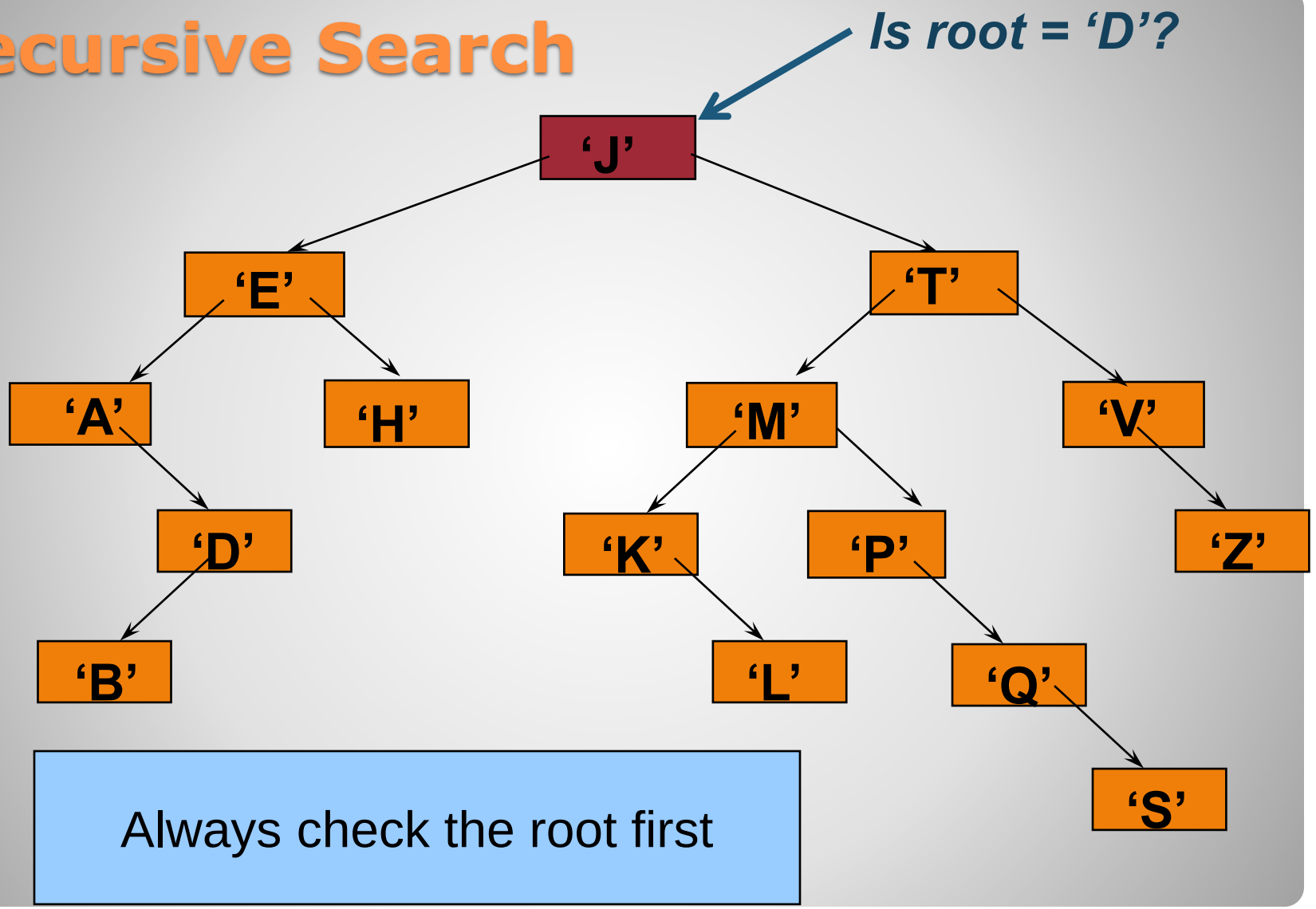
Are 'D', 'Q', and 'N' in the tree?

Recursive Search



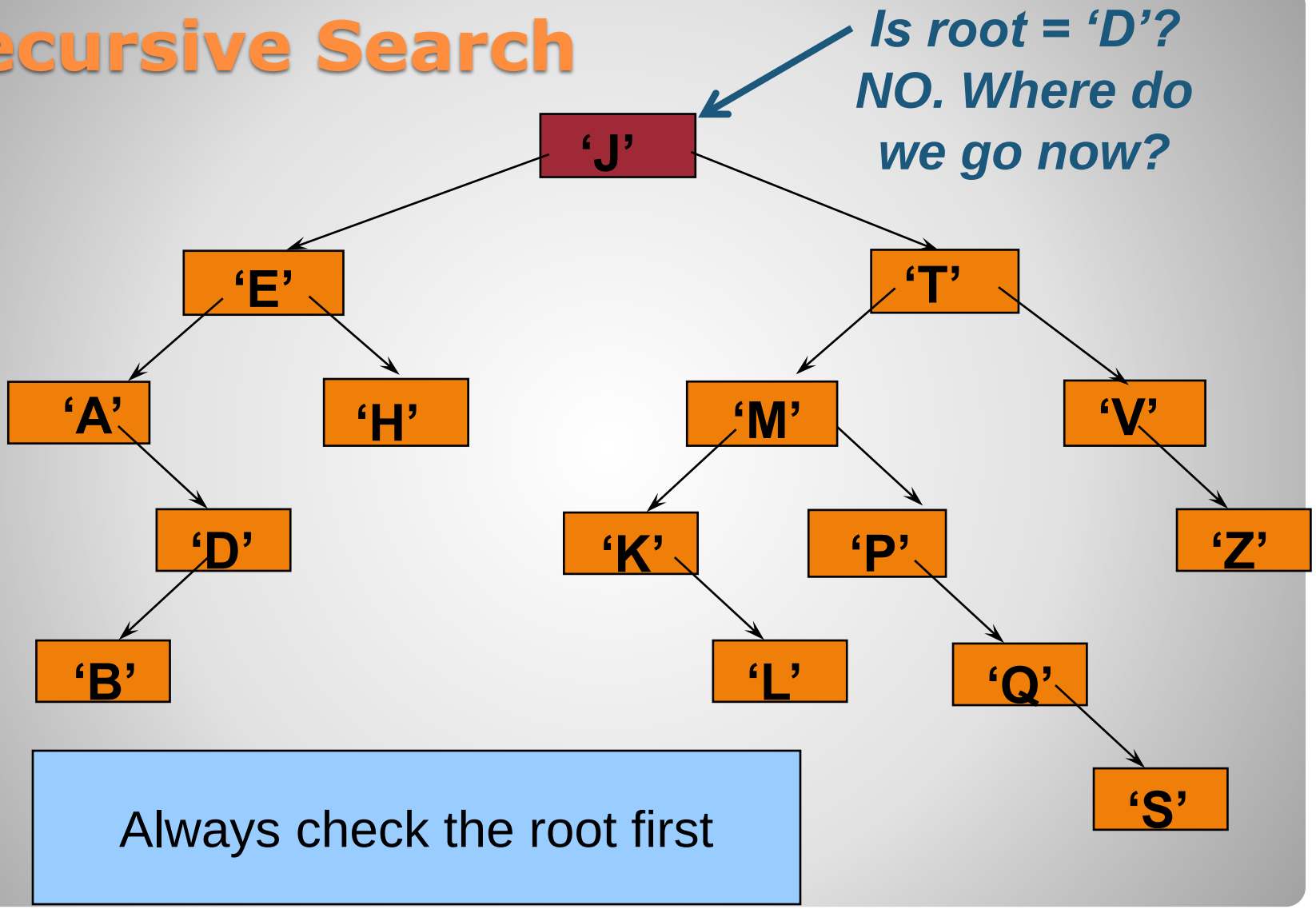
How does a search D proceed?

Recursive Search



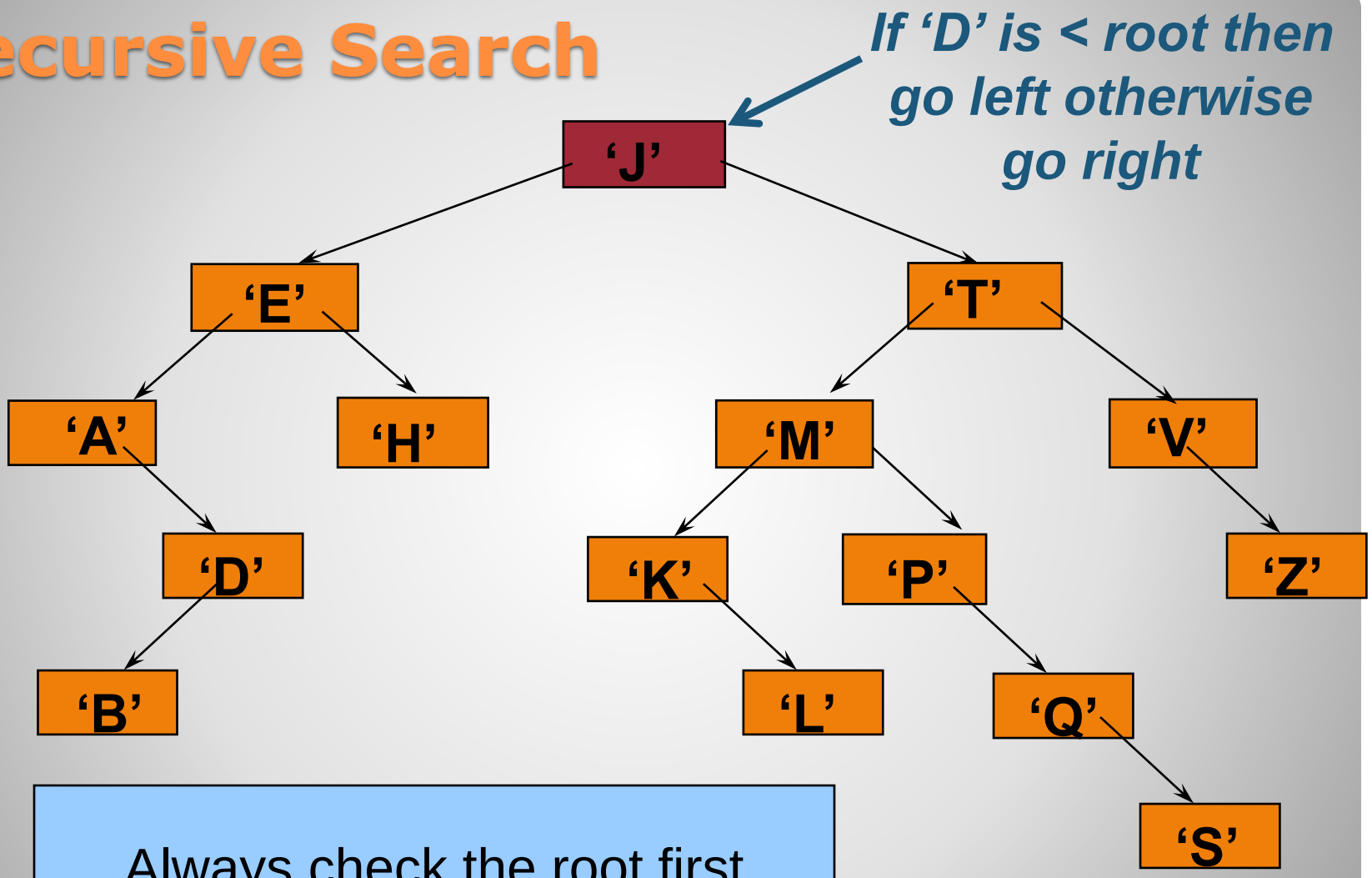
Always check the root first

Recursive Search



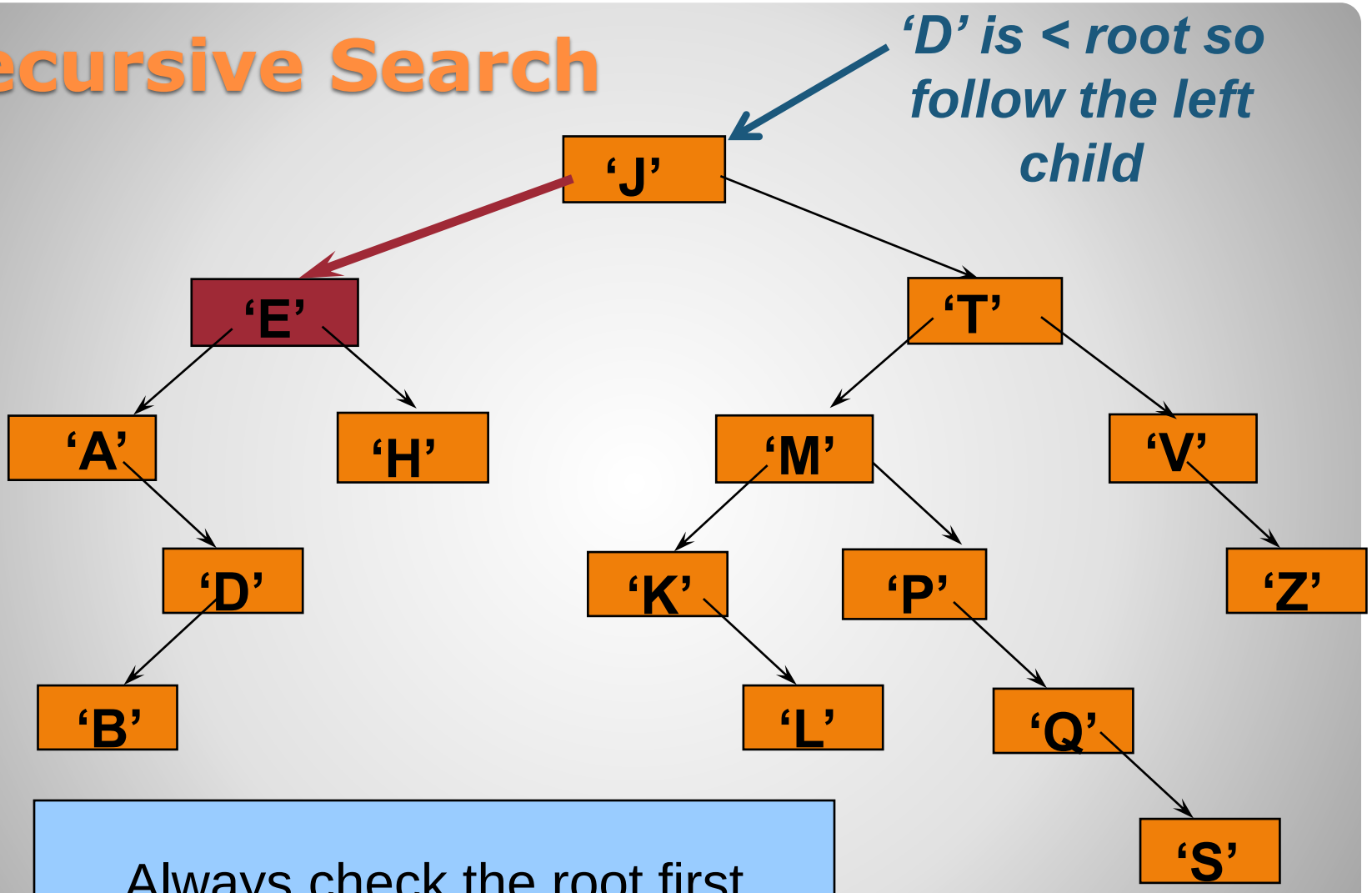
Always check the root first

Recursive Search



Always check the root first

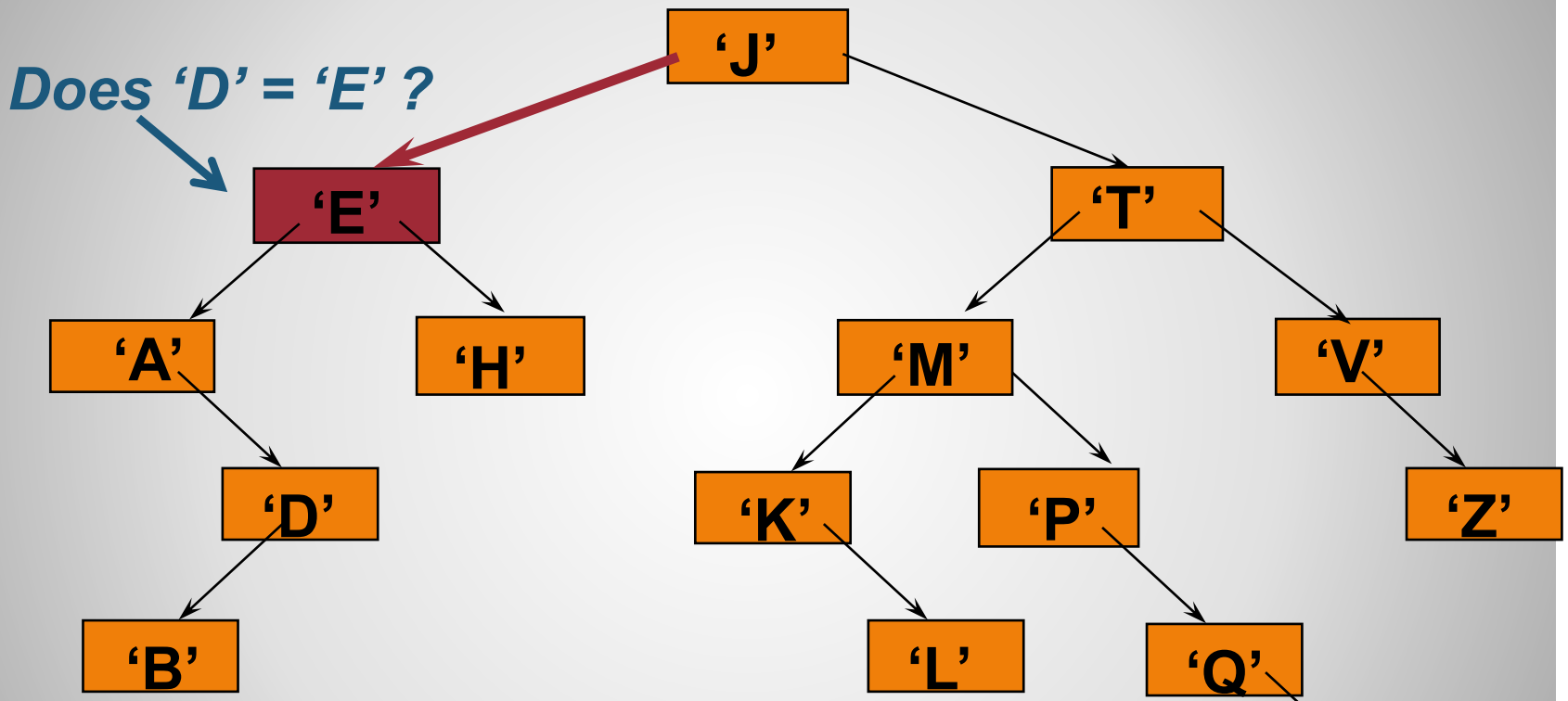
Recursive Search



Always check the root first

Recursive Search

'E' is the "root" of a subtree

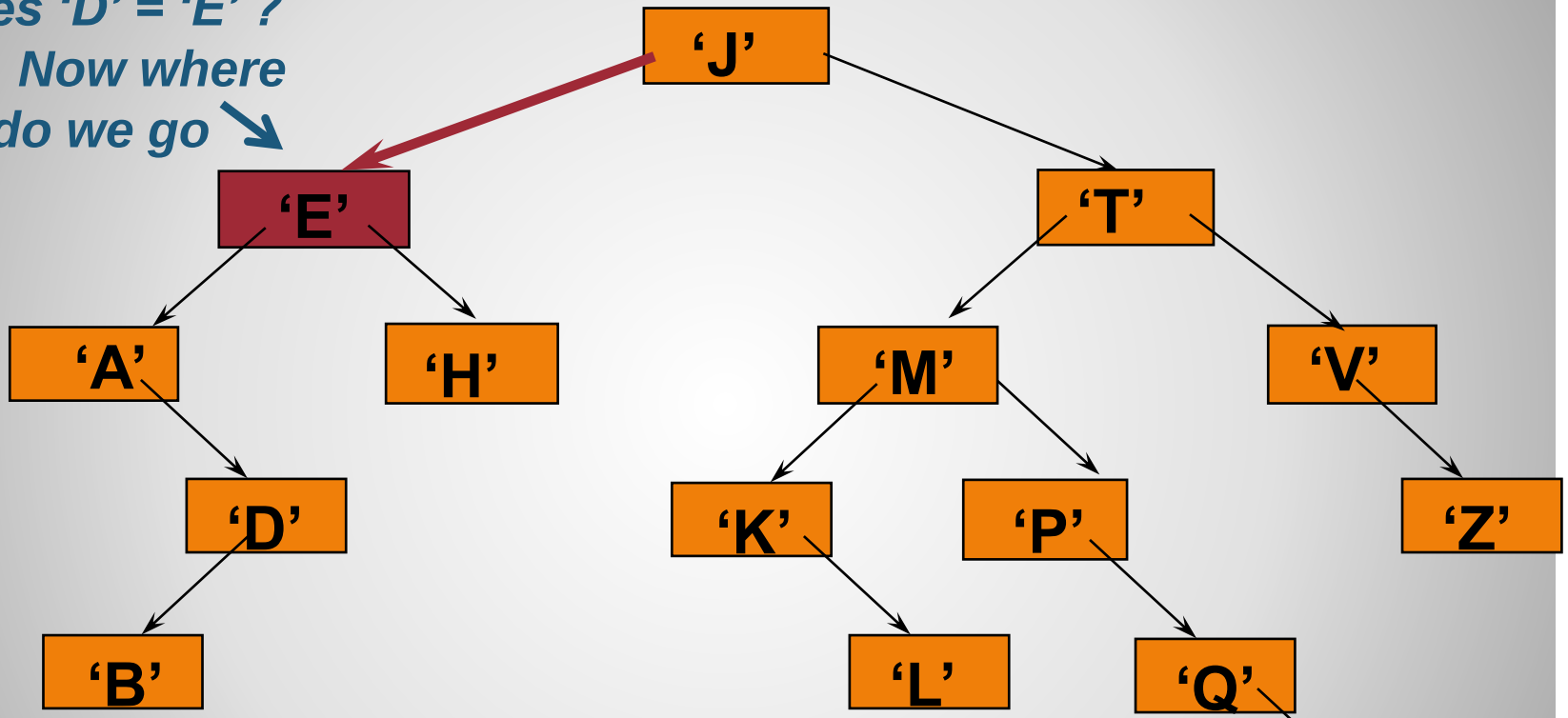


Always check the root first

Recursive Search

'E' is the "root" of a subtree

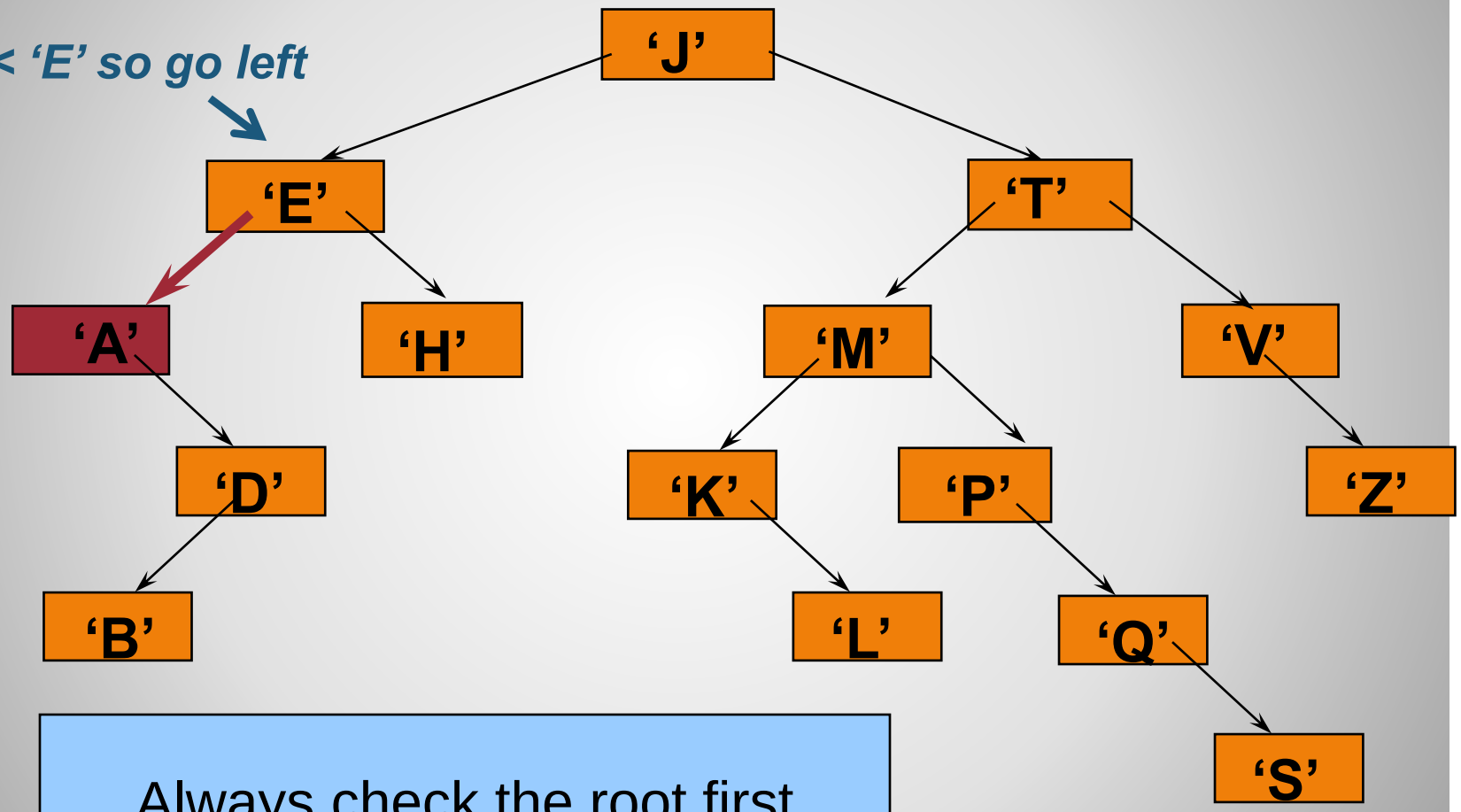
Does 'D' = 'E' ?
NO. Now where do we go ↘



Always check the root first

Recursive Search

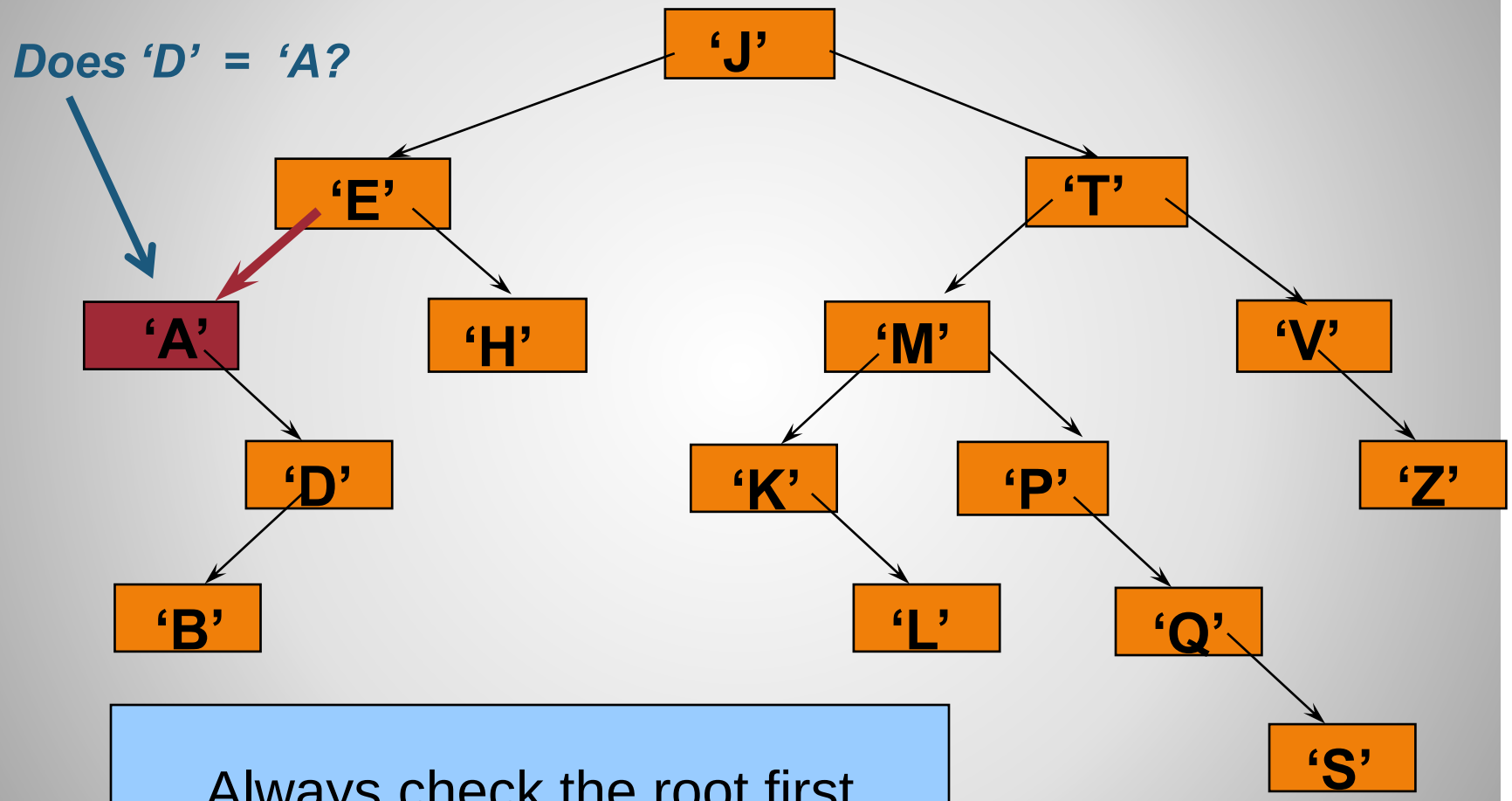
'A' is the "root" of a subtree



Always check the root first

Recursive Search

'A' is the "root" of a subtree

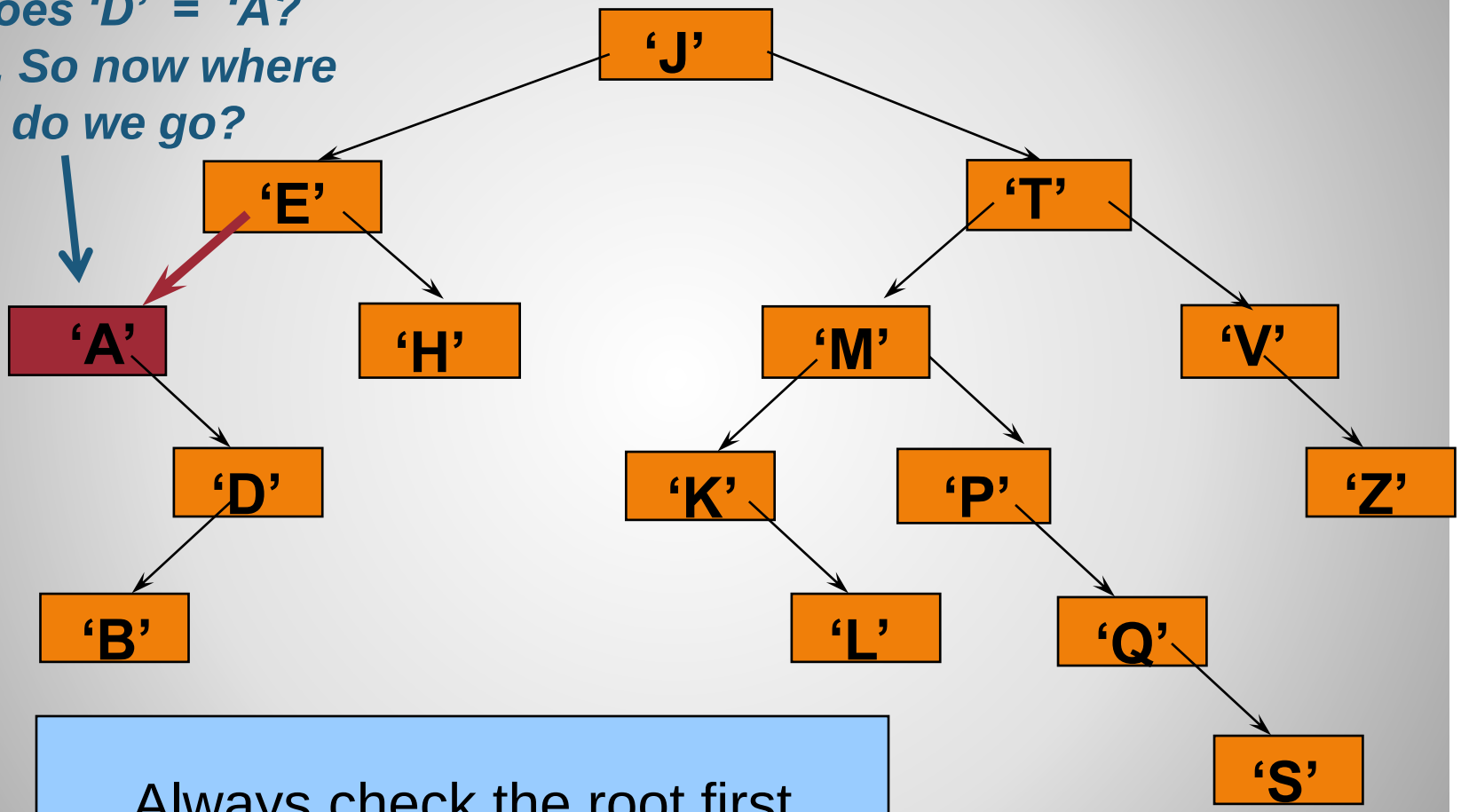


Always check the root first

Recursive Search

'A' is the "root" of a subtree

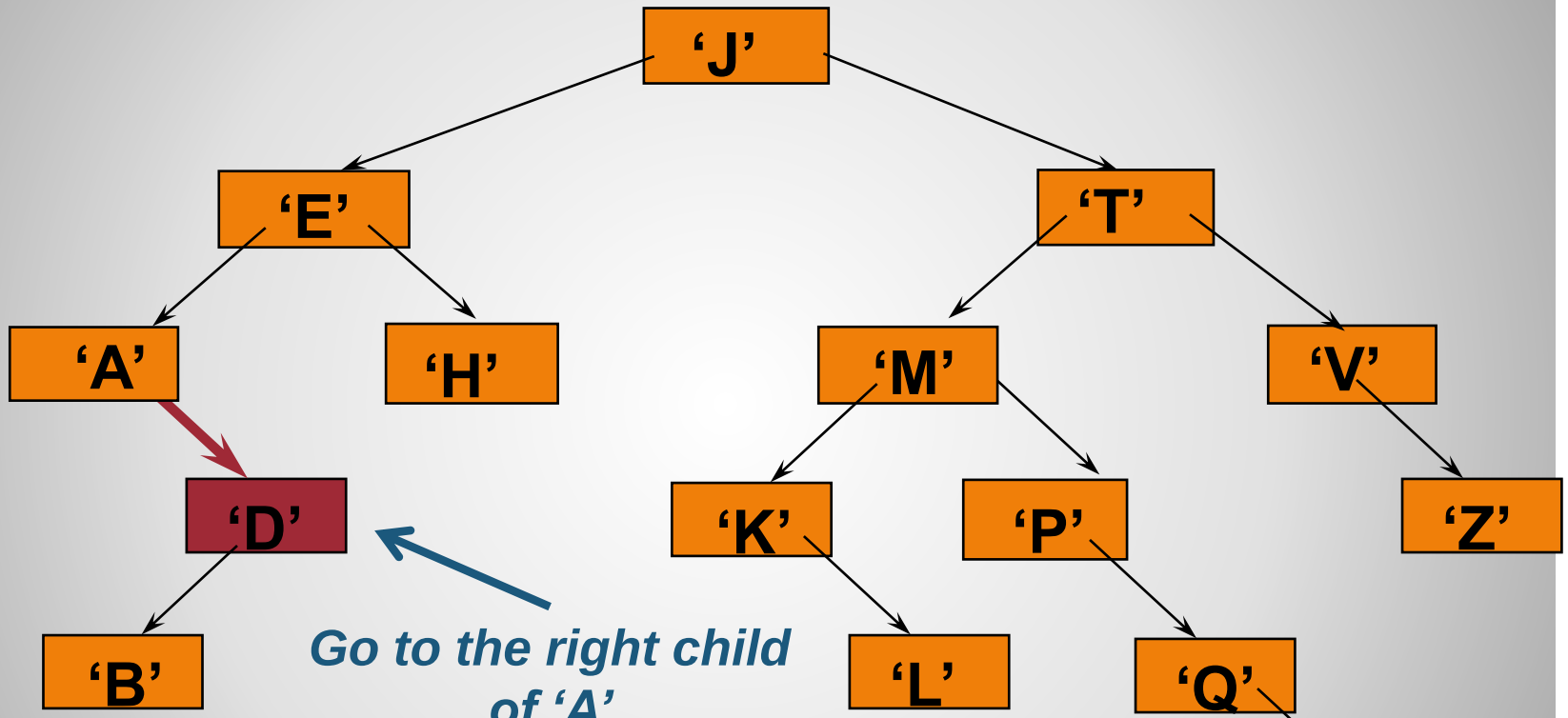
Does 'D' = 'A? NO. So now where do we go?



Always check the root first

Recursive Search

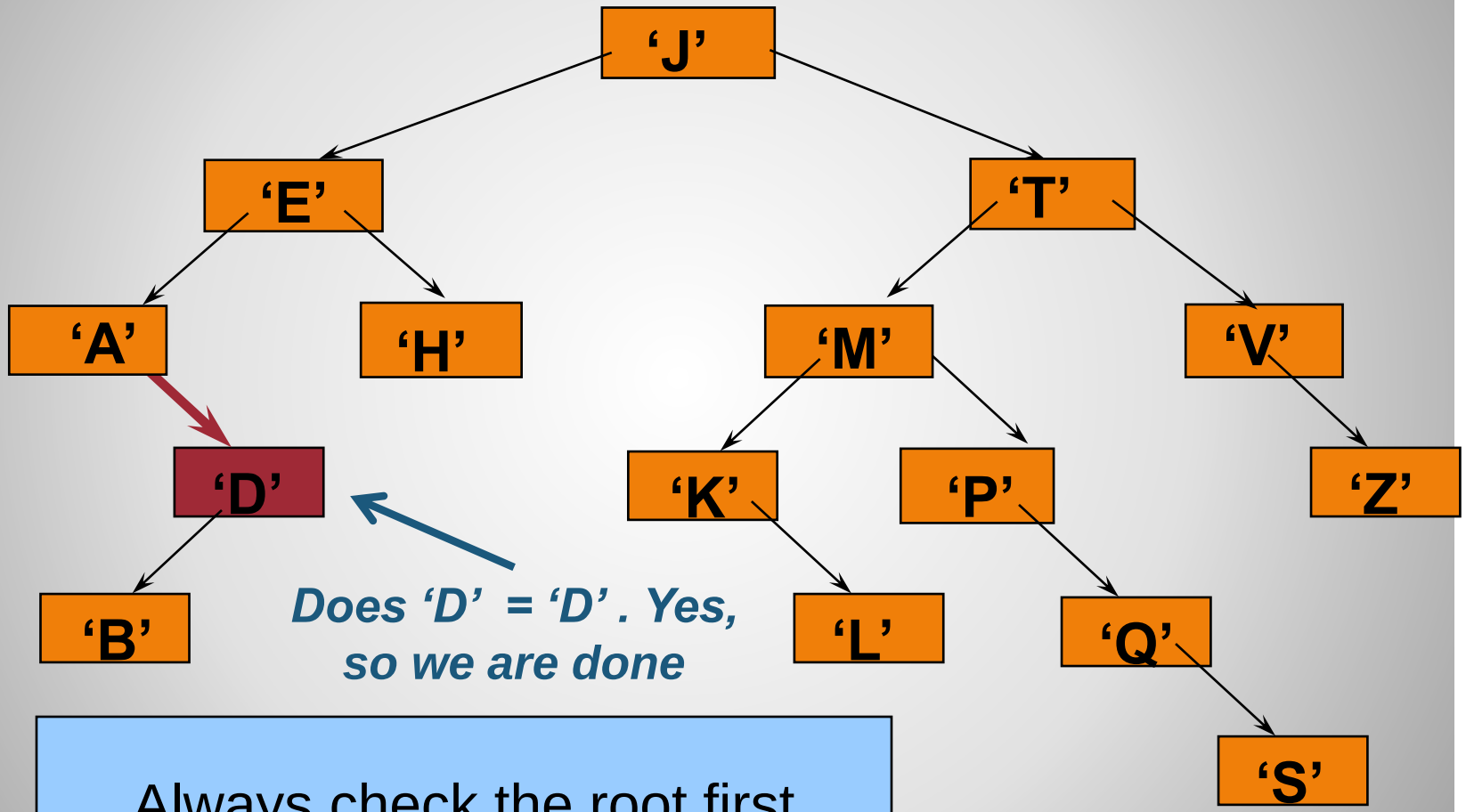
'D' is the "root" of a subtree



Always check the root first

Recursive Search

'D' is the "root" of a subtree



Always check the root first

```
retrieveItem(Node tree, int target) returns int
   if (tree equals null)
      return Integer.MIN_VALUE

   else if (target < tree.data)
      return retrieveItem(tree.left, target)

   else if (target > tree.data)
      return retrieveItem(tree.right, target)

   else
      return tree.data
```

At bottom of tree so item not there

Check the left subtree

Check the right subtree

FOUND IT!!!

Binary Search Tree (linked)

- **IsFull**

  Similar to IsFull for the linked implementations of the other data structures we have covered.

- **IsEmpty**

  Similar to IsEmpty for the linked implementations of the other data structures we have covered.

Binary Search Tree (linked)

Insert

Now we will insert items into a binary search tree…

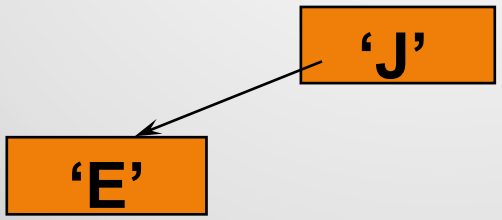
Binary Search Tree (linked)

Shape of BST

- Shape depends on the order of item insertion
- Insert the elements ‘J’ ‘E’ ‘F’ ‘T’ ‘A’ in that order
- The first value inserted is always put in the root

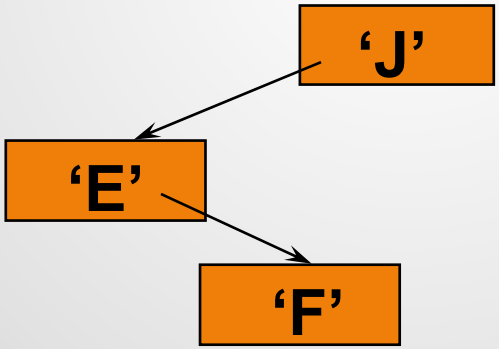
Shape of BST

- Thereafter, each value to be inserted
- compares itself to the value in the root node
- moves left it is less or
- moves right if it is greater
- *When does the process stop?*



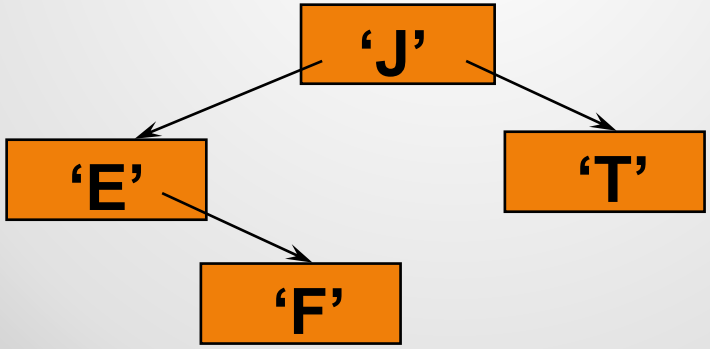
Shape of BST

- Trace path to insert ‘F’



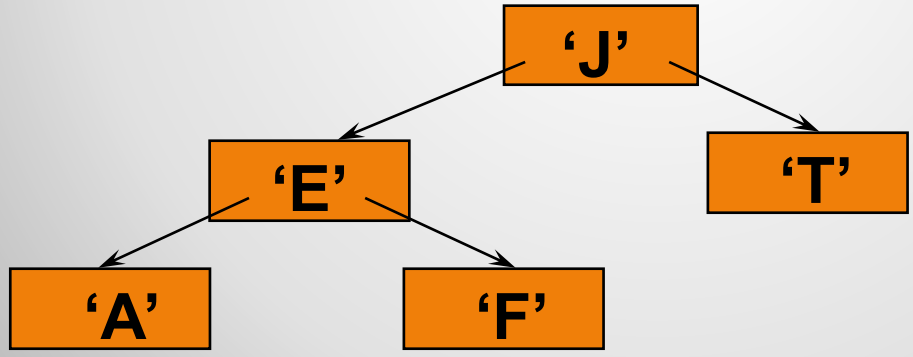
Shape of BST

- Trace path to insert ‘T’



Shape of BST

- Trace path to insert ‘A’



Shape of BST

- Now build tree by inserting ‘A’ ‘E’ ‘F’ ‘J’ ‘T’ in that order

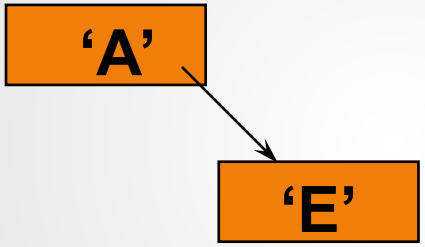
And the moral is?

Shape of BST

- Now build tree by inserting **‘A’** ‘E’ ‘F’ ‘J’ ‘T’ in that order

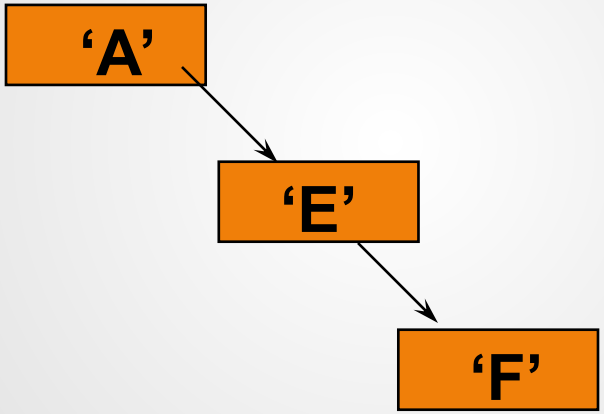
Shape of BST

- Now build tree by inserting ‘A’ **‘E’** ‘F’ ‘J’ ‘T’ in that order



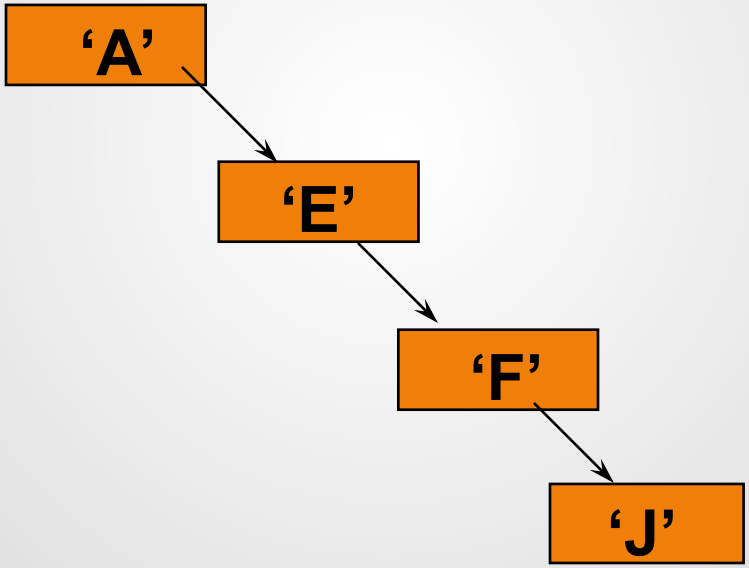
Shape of BST

- Now build tree by inserting ‘A’ ‘E’ **‘F’** ‘J’ ‘T’ in that order



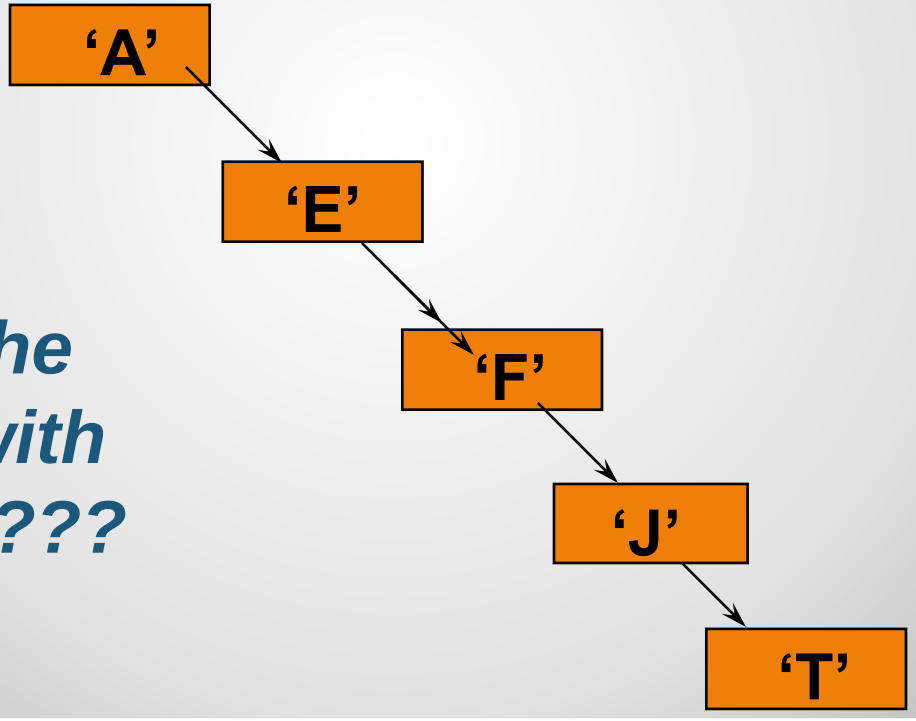
Shape of BST

- Now build tree by inserting ‘A’ ‘E’ ‘F’ **‘J’** ‘T’ in that order



Shape of BST

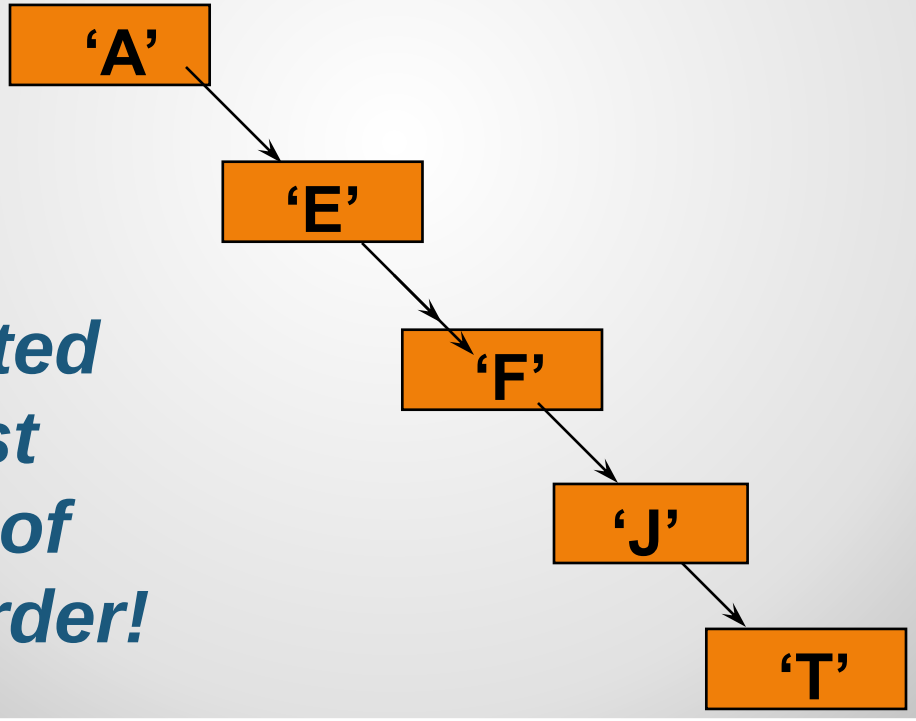
- Now build tree by inserting ‘A’ ‘E’ ‘F’ ‘J’ **‘T’** in that order



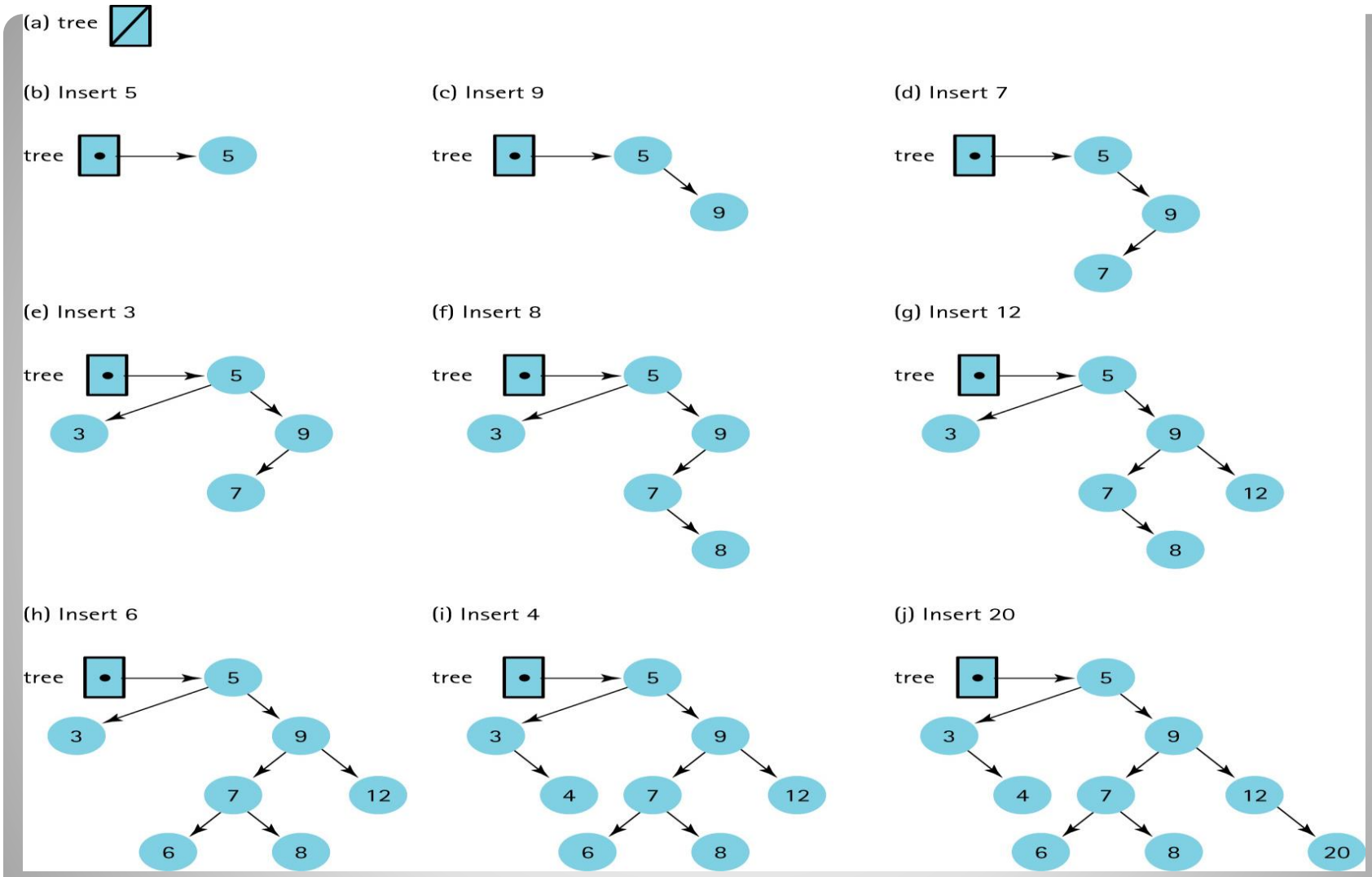
What is the problem with this shape???

Shape of BST

- Now build tree by inserting ‘A’ ‘E’ ‘F’ ‘J’ **‘T’** in that order



Degenerated into a list because of insertion order!



Recursive Insertion

```
insertItem(int item)
    root = insertItem(root, item)


insertItem(Node tree, int item) returns Node
    if (tree equals null)
        return new Node instance with item in it



    if (item < tree.data)
        // Set left child to the node that is returned
        Set tree.left to insertItem(tree.left, item)
    else if (item > tree.data)
        // Set right child to the node that is returned
        Set tree.right to insertItem(tree.right, item)

    return tree
```

insertItem(int) – Public method
insertItem(Node, int) – Private helper method

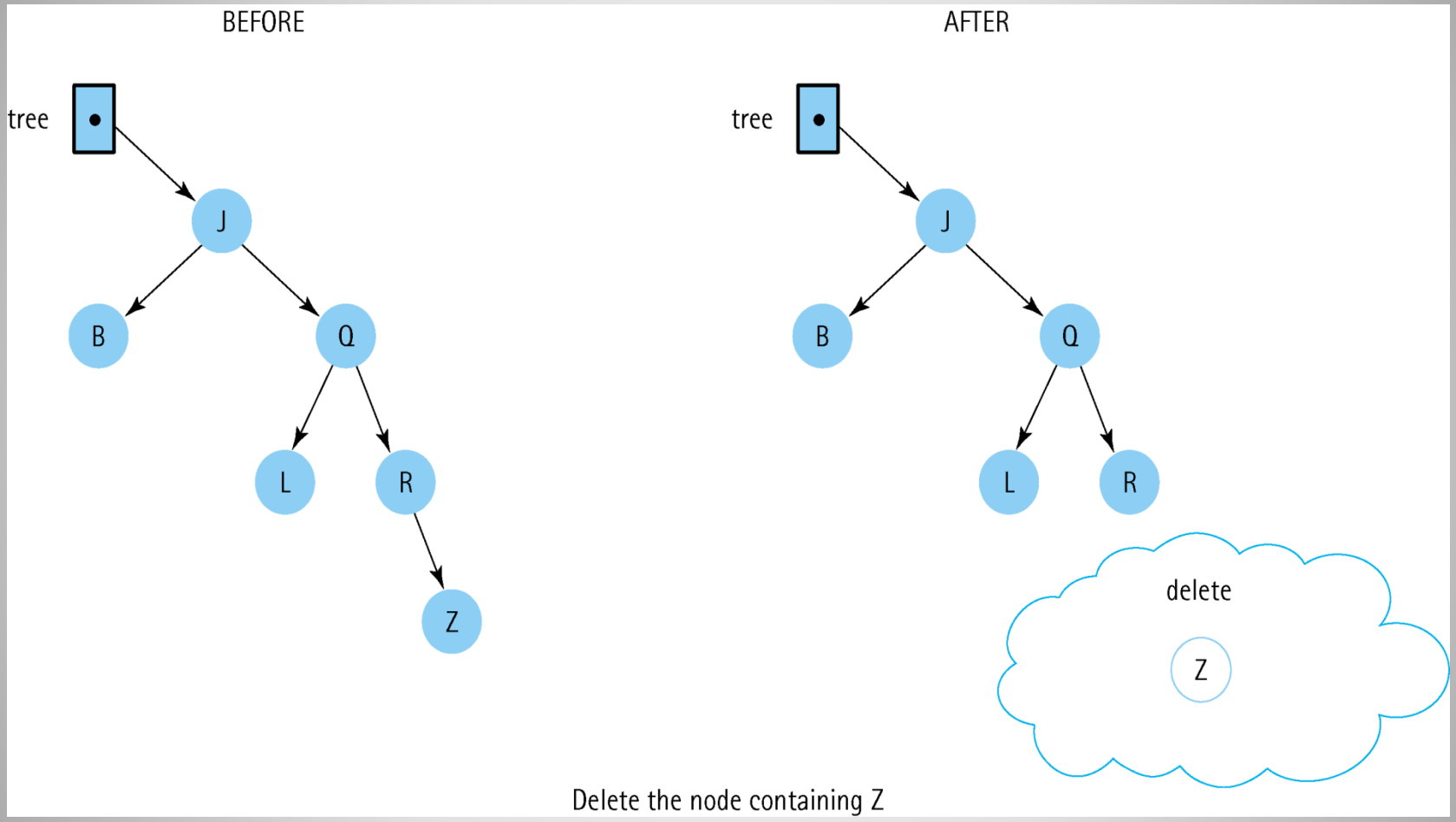
Recursive Insertion

Delete

Now we will delete items from a binary search tree…

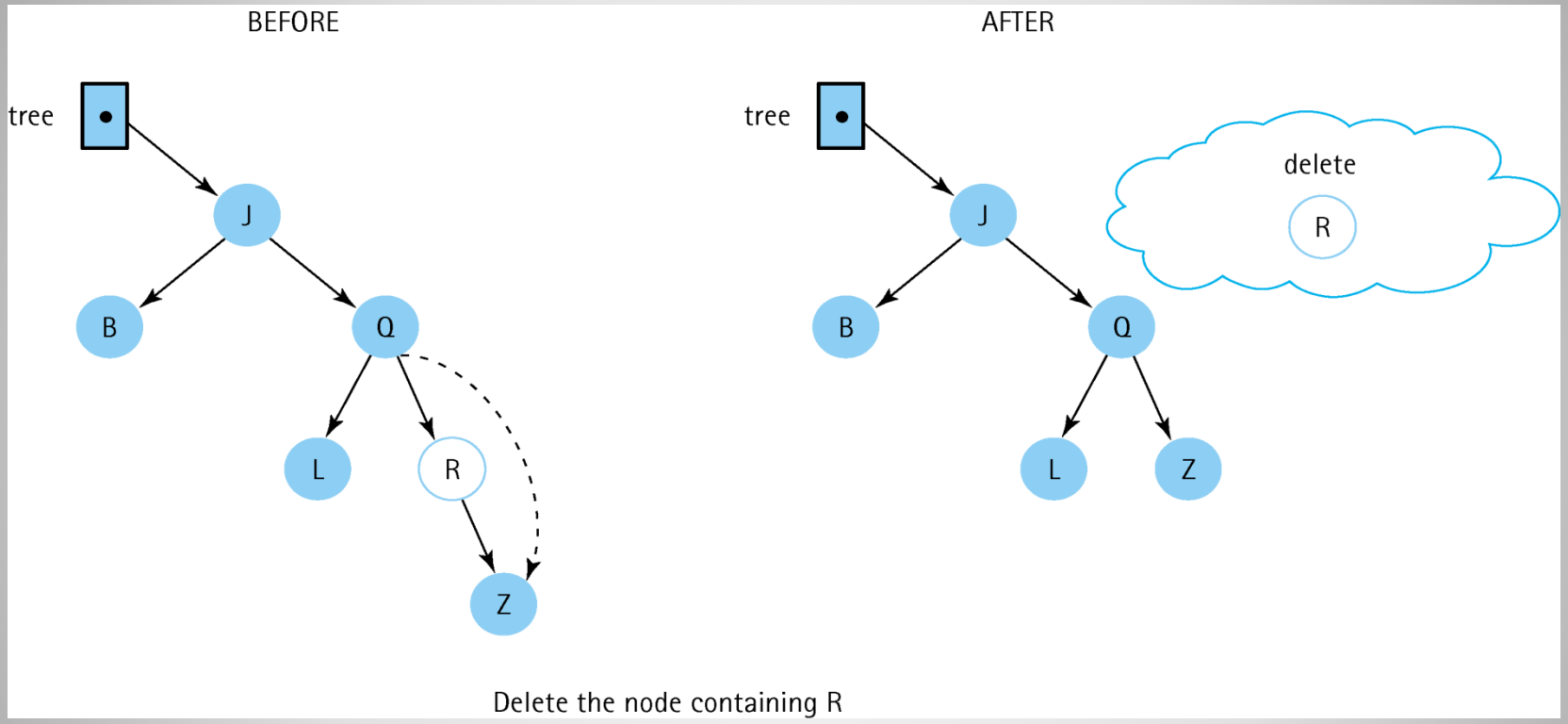
Binary Search Tree (linked)

Delete ‘Z’



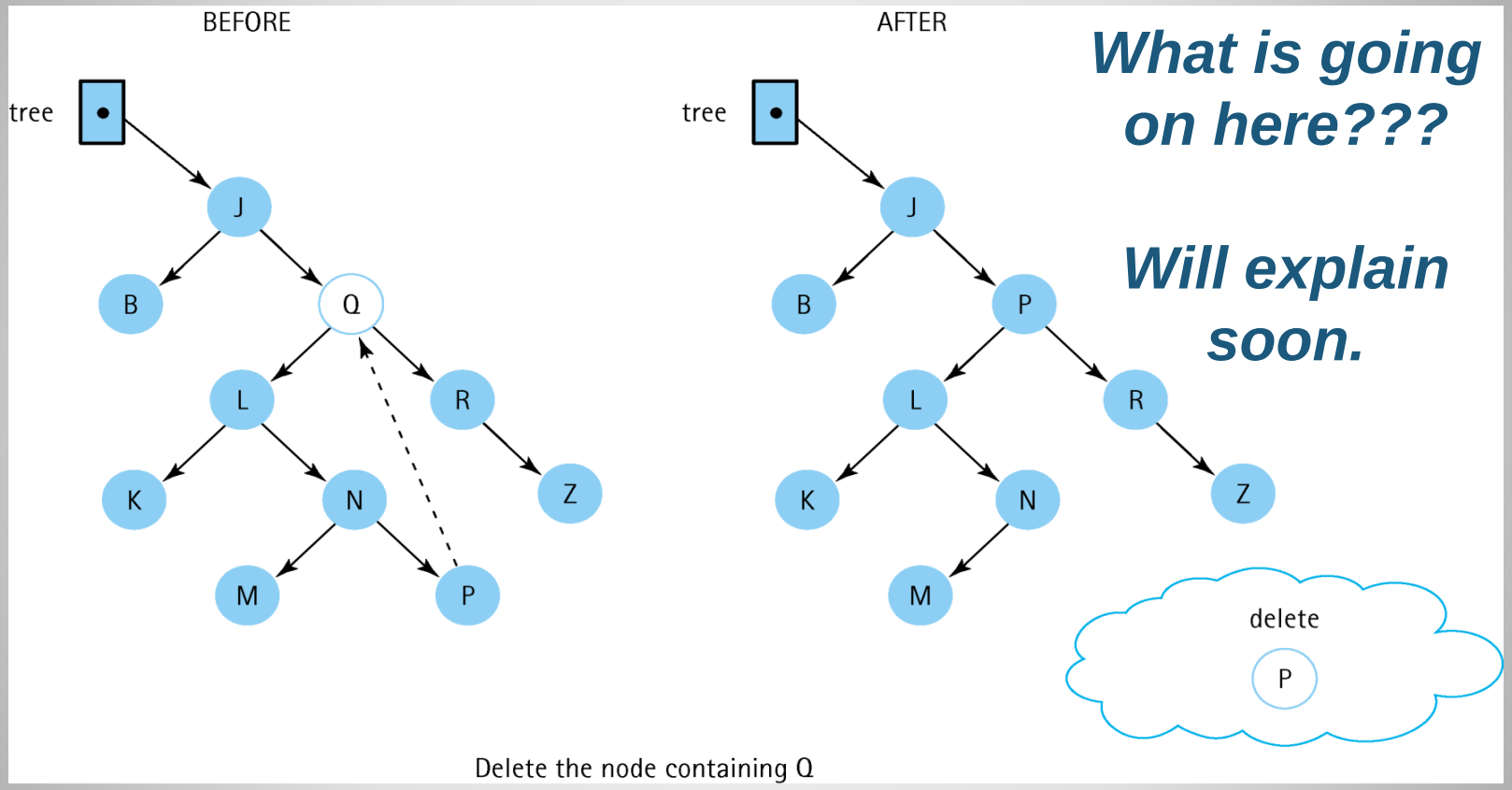
Recursive Deletion

Delete 'R'



Recursive Deletion

Delete 'Q'



Recursive Deletion

- **Can you summarize the three deletion cases?**

Recursive Deletion

Three Deletion Cases

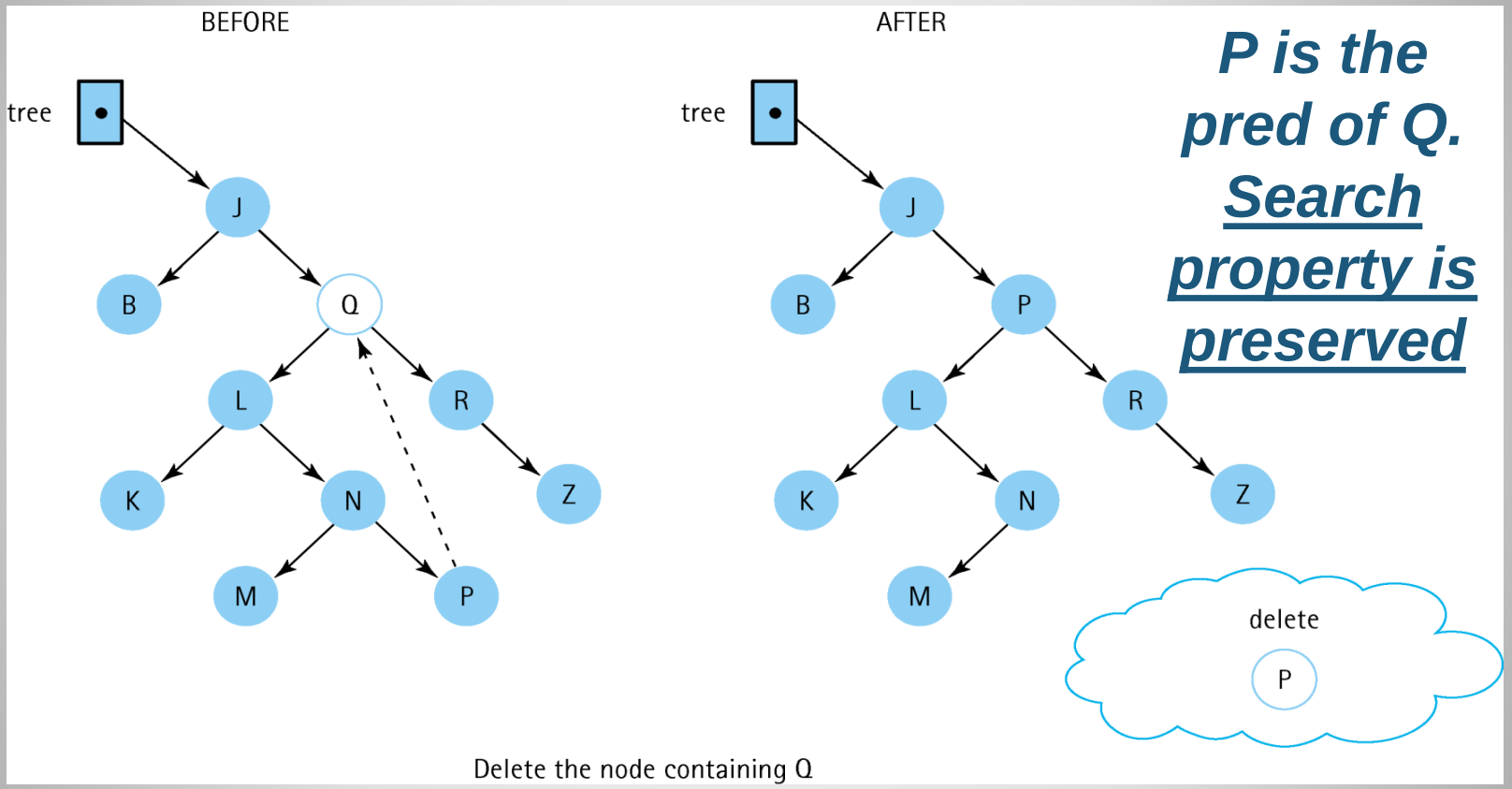
1. **Leaf – Delete node is a leaf. Delete the node and set the child pointer of parent to null.**

2. **One child – Delete node has one child. Delete the node and fix the child pointer of the parent to point to the appropriate child of the node to be deleted.**

3. **Two children – Delete node has two children. Copy the data from the predecessor into the node to be deleted. The predecessor is the largest value in the left subtree of the node to be deleted. Finally, delete the predcessor node.**

Recursive Deletion

Delete ‘Q’



Recursive Deletion

Delete Pseudocode for When Node is Found

```
if (Left(tree) is NOT NULL) AND (Right(tree) is NOT NULL)
   Find predecessor
   Set Info(tree) to Info(predecessor)
   Delete predecessor
else if Left(tree) is NOT NULL
   Set tree to Left(tree)
else if Right(tree) is NOT NULL
   Set tree to Right(tree)
else
   Set tree to NULL
```

Recursive Deletion

```
public Node deleteItem(Node tree, int item) {
    if (tree == null) {
        return null;
    } else if (item < tree.data) {
        tree.left = deleteItem(tree.left, item);
    } else if (item > tree.data) {
        tree.right = deleteItem(tree.right, item);
    } else {
      if (tree.left != null && tree.right != null) {
          Node predNode = findPredecessorNode(tree.left);
          tree.data = predNode.data;
          tree.left = deleteItem(tree.left, predNode.data);
       } else if (tree.left != null) {
          tree = tree.left;
       } else if (tree.right != null) {
          tree = tree.right;
       } else {
         tree = null;
       }
    }
    return tree;
}
```

Deleting node with two child

Copy pred

Delete pred

Deleting node with one child
Set tree to the child (tree gets returned at end of method)

Deleting Leaf. If we get here, then we are deleting a leaf node

Recursive Deletion

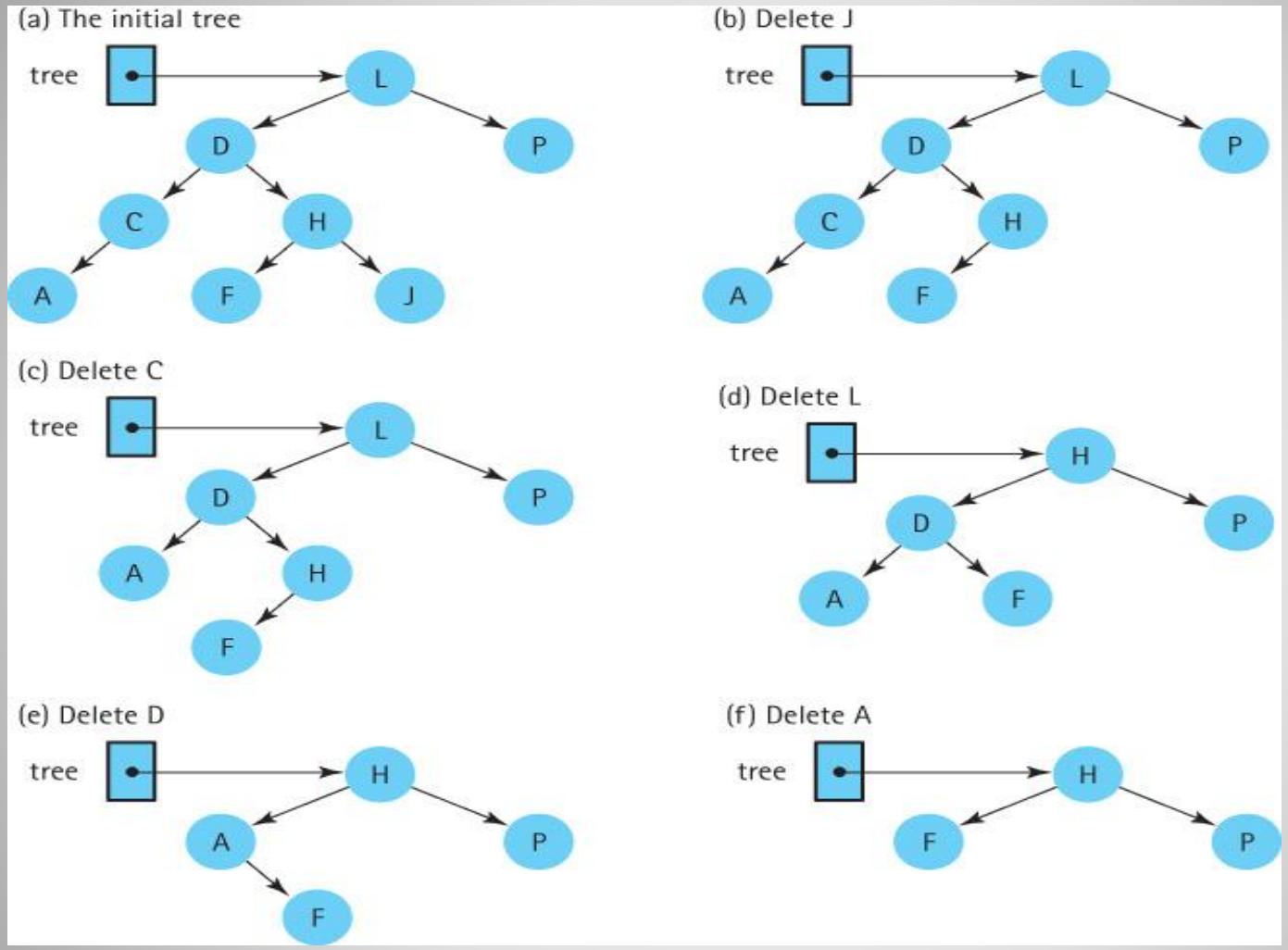
```
private Node findPredecessorNode(Node tree)
{
   Node current = tree;

   while (current.right != null)
   {
      current = current.right;
   }

   return current;
}
```

Keep going to the right in the tree

Recursive Deletion



Recursive Deletion

- Do in-class problem for binary search tree.

In-Class Problem

Clearing The Tree

- How do we clear the tree?
- Here is makeEmpty()…

Clearing The Tree

- Let the garbage collector handle releasing all children of the root (they will all eventually be garbage collected).

makeEmpty()

Set root to null

Clearing The Tree

```
makeEmpty()
   makeEmpty(root)
   root = null
```

This is what you would do to explicitly release each node (postorder traversal)

```
makeEmpty(Node tree)
   if (tree not equal to null)
         makeEmpty(tree.left)
         makeEmpty(tree.right)
         Set tree to null
```

Clear the left subtree

Clear the right subtree

Make the current node a candidate for garbage collection

Traversal (Unsorted List) - REVIEW

- Visit all nodes of the data structure
- How do you traverse an ***unsorted list*** that uses a linked implementation?

Traversals - REVIEW

Traversal (Unsorted List) - REVIEW

- Visit all nodes of the data structure
- How do you traverse an ***unsorted list*** that uses a linked implementation?

ANSWER

1. Create a temporary pointer that points to the start of the list.
2. Use that pointer to "visit" each node on the list.
3. Keep going until that pointer is null. If the pointer is null, then you reached the end of the list.

Traversal (Binary Search Tree)

- Visit all nodes of the data structure
- How do you traverse a ***binary search tree*** that uses a linked implementation?

Traversal

- Visit all nodes of the data structure
- How do you traverse a *binary search tree* that uses a linked implementation?

ANSWER

There are three common ways to traverse a binary search tree.

1. Inorder
2. Preorder
3. Postorder

Traversals

- Will traverse the binary search tree recursively.
- **Inorder** – Process the left subtree (recursively) then "visit" the current node then process the right subtree (recursively)
- **Preorder** – "Visit" the current node then process the left subtree (recursively) then process the right subtree (recursively).
- **Postorder** – Process the left subtree (recursively) then process the right subtree recursively then "visit" the current node.

Traversals

- The pre-, post-, and in- refer to when the current node is processed with respect to the left and right subtrees.
- Preorder - Current node is processed BEFORE the left and right subtrees.
- Postorder - Current node is processed AFTER the left and right subtrees.
- Inorder - Current node is processed AFTER the left subtree but BEFORE the right subtree. Inorder will have the effect of "visiting" each node according to the order of the key of the tree.

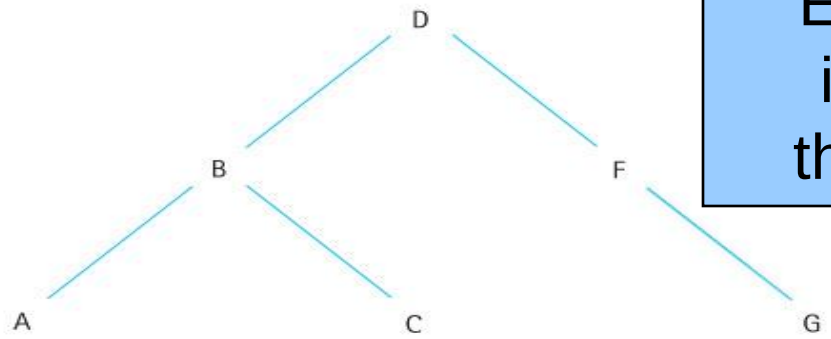
Inorder(tree)
if tree is not NULL
Inorder(Left(tree))
Visit Info(tree)
Inorder(Right(tree))

PostOrder(tree)
if tree is not NULL
Postorder(Left(tree))
Postorder(Right(tree)
Visit Info(tree)

PreOrder(tree)
if tree is not NULL
Visit Info(tree)
Preorder(Left(tree))
Preorder(Right(tree))

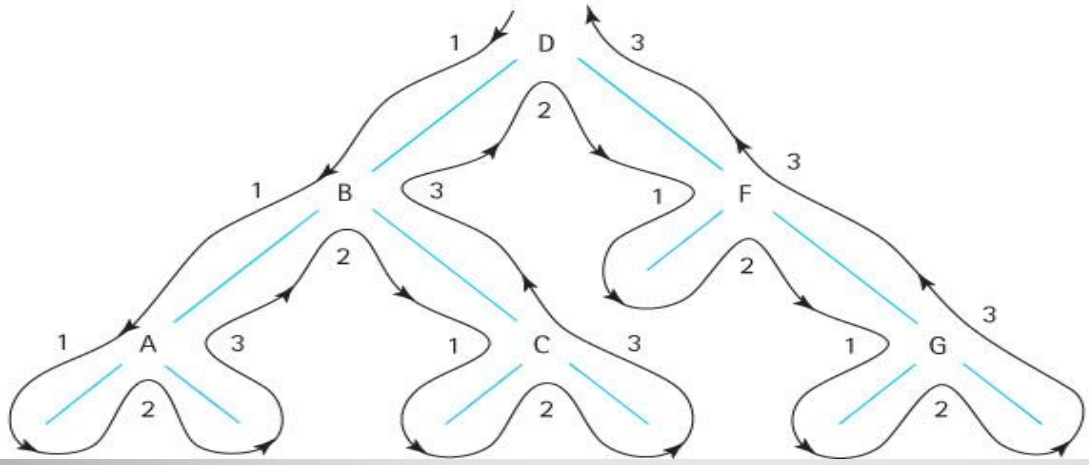
Traversals

A binary tree



Each node is visited three times

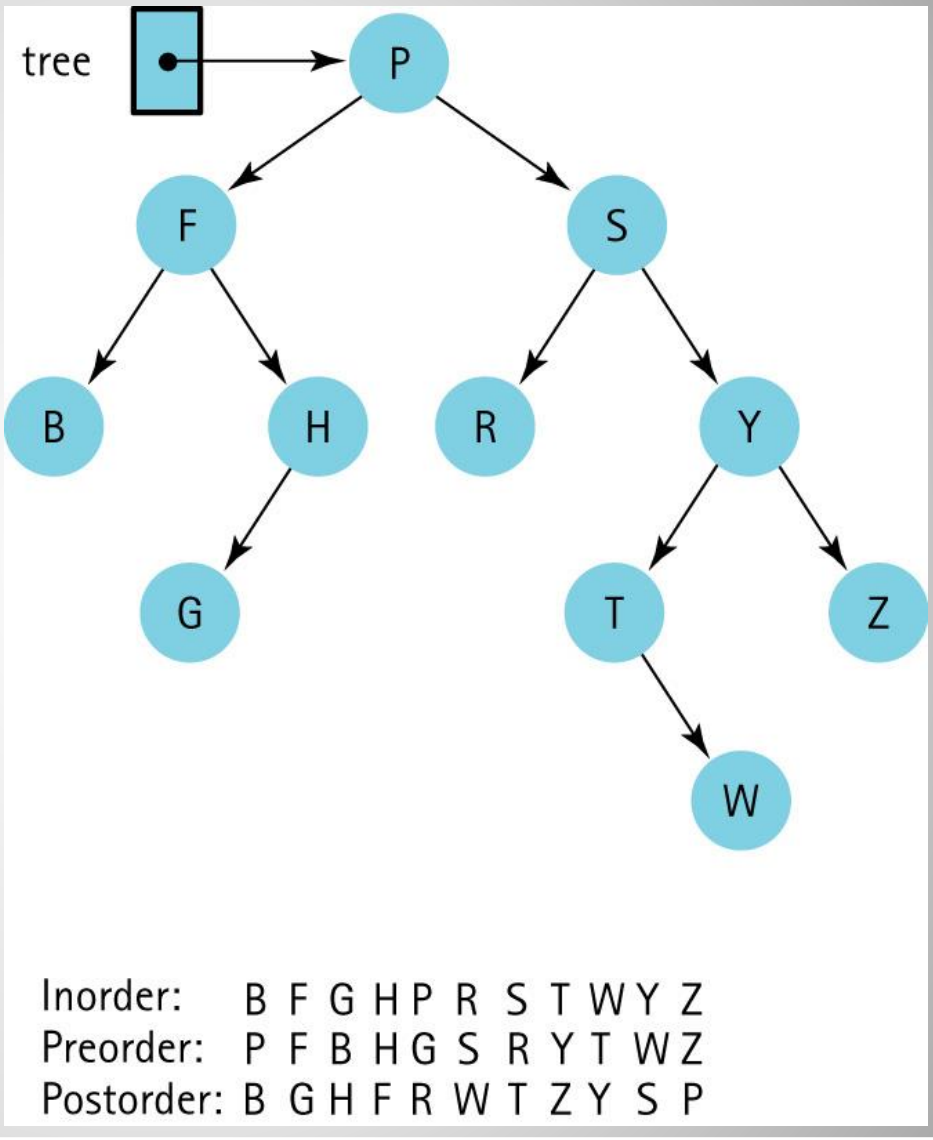
The extended tree



Preorder: DBACFG
Inorder: ABCDFG
Postorder: ACBGFD

Traversals

Traversals



Depth-First Traversal

- **Depth-First Traversal** – Go as deep as you can then backtrack.
- Instead of recursion you can use a **stack** to help with traversal.
- Note: Inorder, preorder, and postorder traversals are examples of depth-first traversals.

DepthFirstTraversal w/Stack (preorder)

```
stack.Push(root)

do
        stack.Pop(node)

        Visit node

        Push all children of node onto stack
while !stack.IsEmpty()
```

Put root on stack

Take the top node off the stack

Push node's children on stack

Keep going while stack is not empty

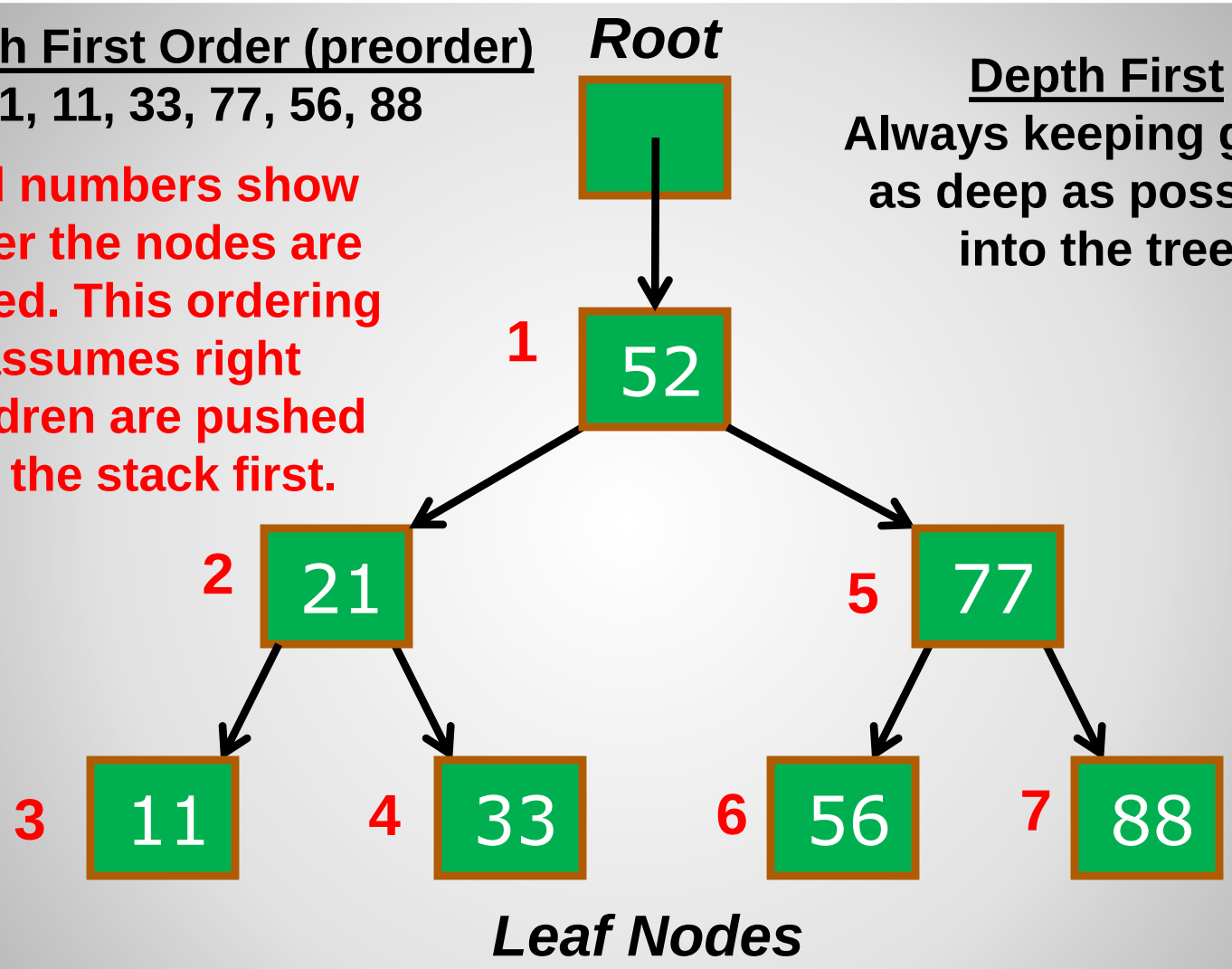
Note: Could use recursive methods instead of stack.

Depth-First Traversal

Depth First Order (preorder)
52, 21, 11, 33, 77, 56, 88

Red numbers show order the nodes are visited. This ordering assumes right children are pushed on the stack first.

Depth First
Always keeping going as deep as possible into the tree



Depth-First Traversal

Breadth-First Traversal

- **Breadth-First Traversal** – Process all nodes in one level then move on to nodes in the next level.
- Use a **queue** to help with traversal.

BreadthFirstTraversal

```
queue.Enqueue(root)

do
        queue.Dequeue(node)

        Visit node

        Enqueue all children of node onto queue
while !queue.IsEmpty()
```

Put root on queue

Take the front node off the queue

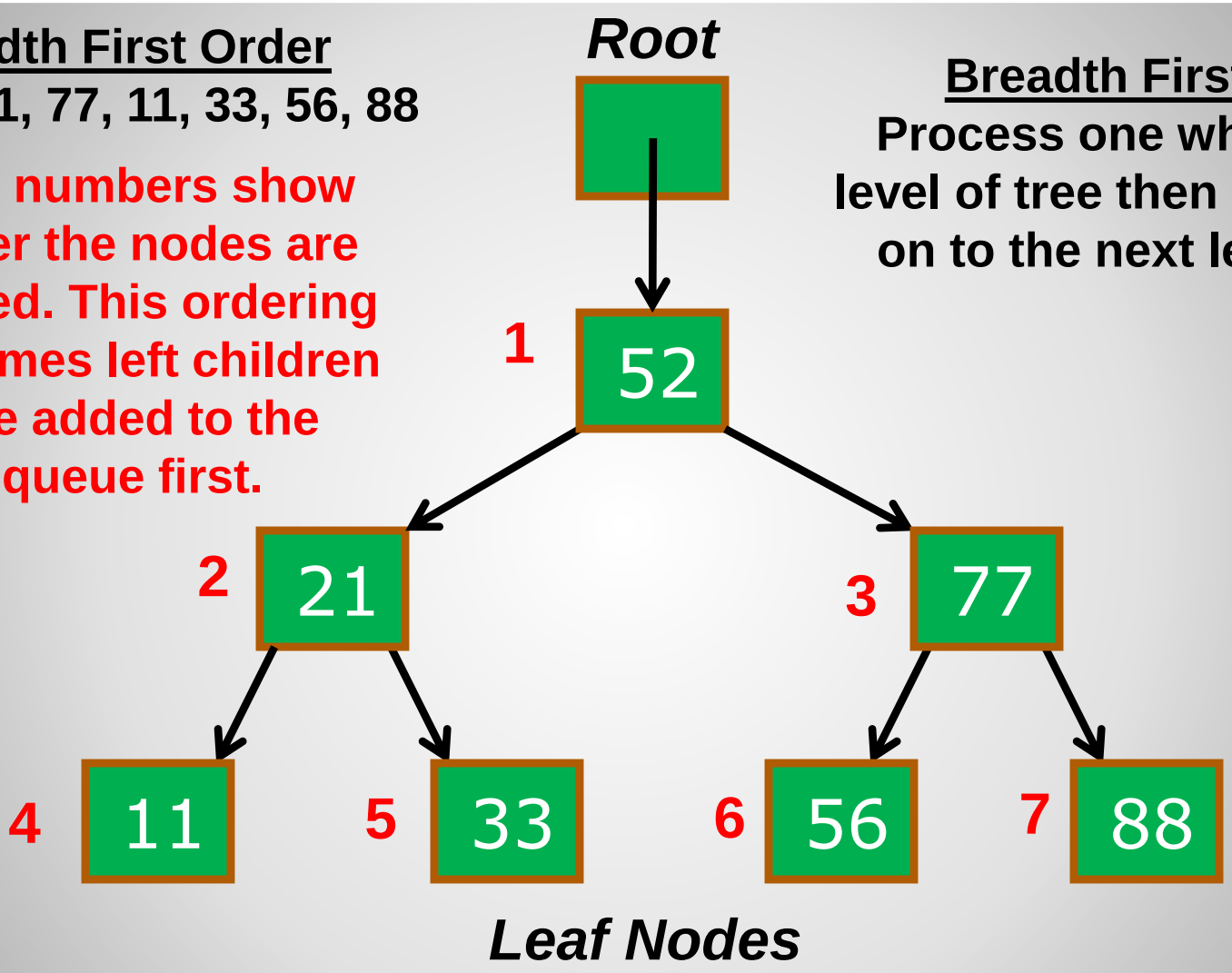
Add node's children to the end of queue

Keep going while queue is not empty

Breadth-First Traversal

Breadth First Order
52, 21, 77, 11, 33, 56, 88

Red numbers show order the nodes are visited. This ordering assumes left children are added to the queue first.

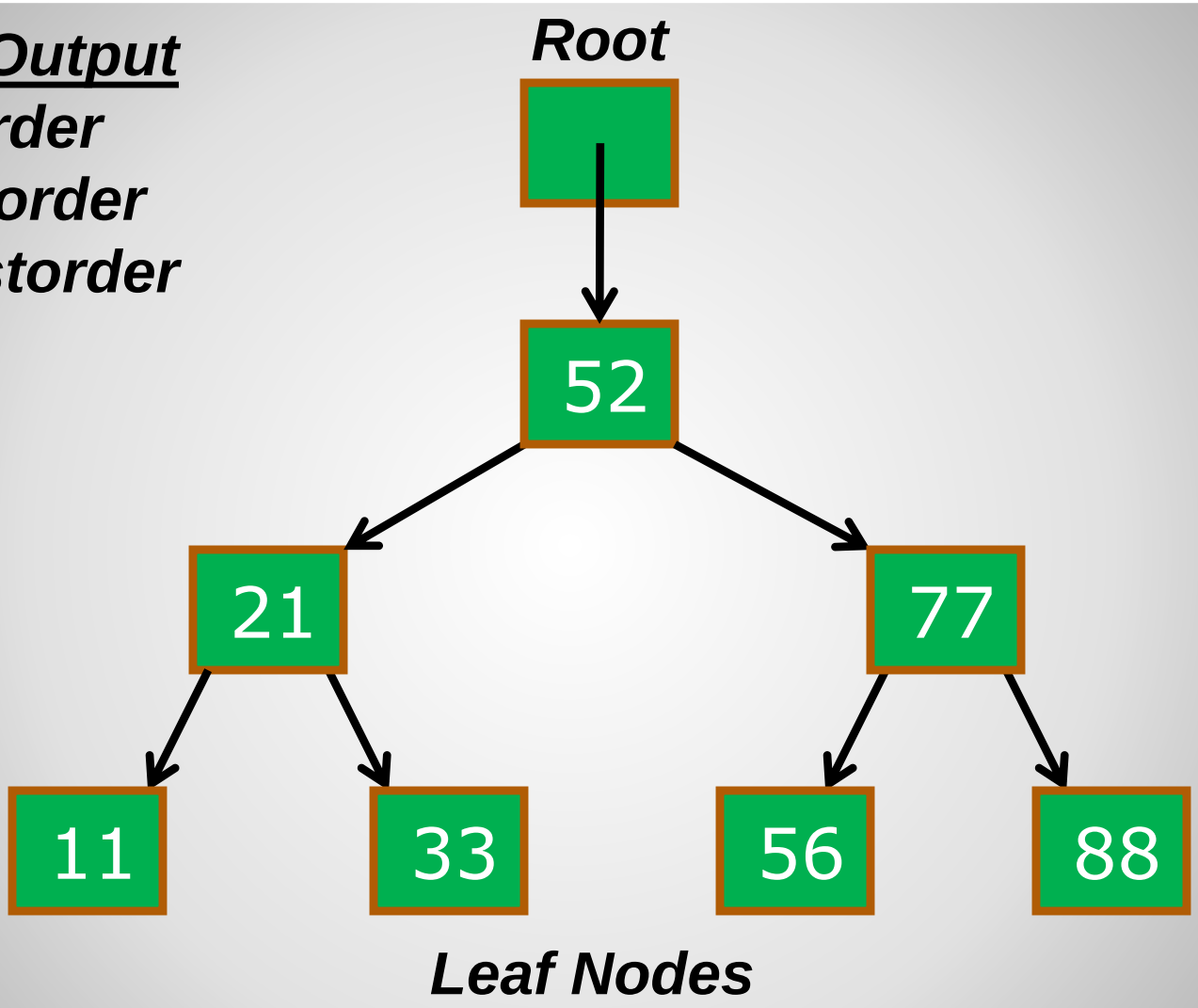


Breadth First
Process one whole level of tree then move on to the next level

Breadth-First Traversal

Show Output

1. *Inorder*
2. *Preorder*
3. *Postorder*



Traversals Practice

Show Output

1. ***Inorder***

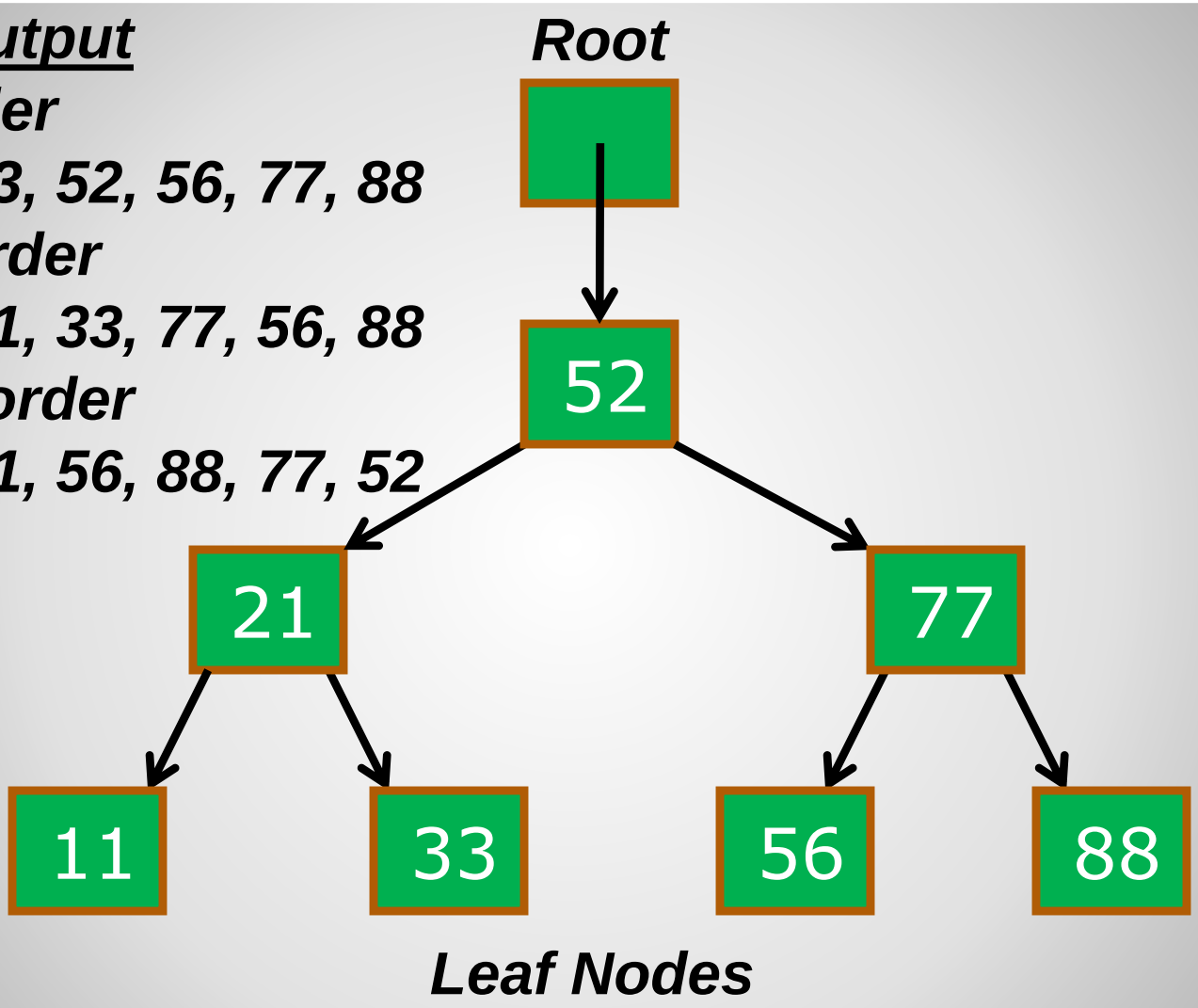
11, 21, 33, 52, 56, 77, 88

2. ***Preorder***

52, 21, 11, 33, 77, 56, 88

3. ***Postorder***

11, 33, 21, 56, 88, 77, 52



Traversals Practice

- **End of Slides**

End of Slides