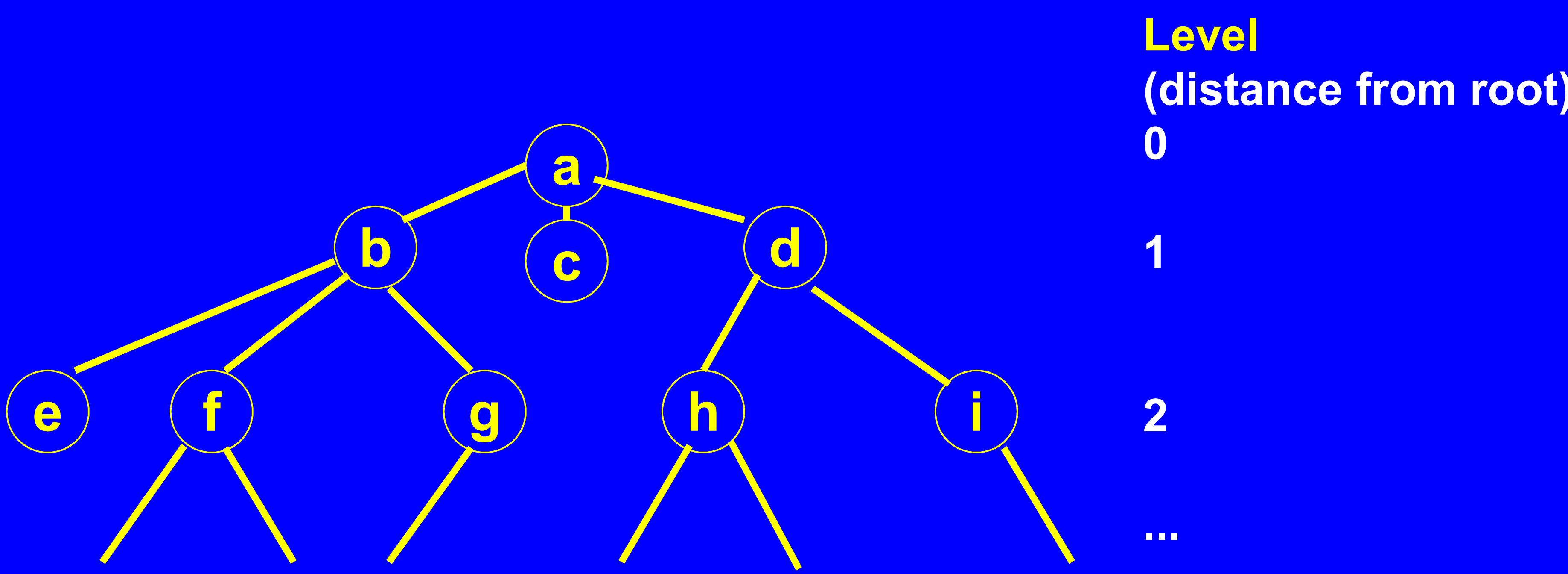
1.00 Lecture 28

Trees

Reading for next time: None.
Please look over the lecture notes before class.

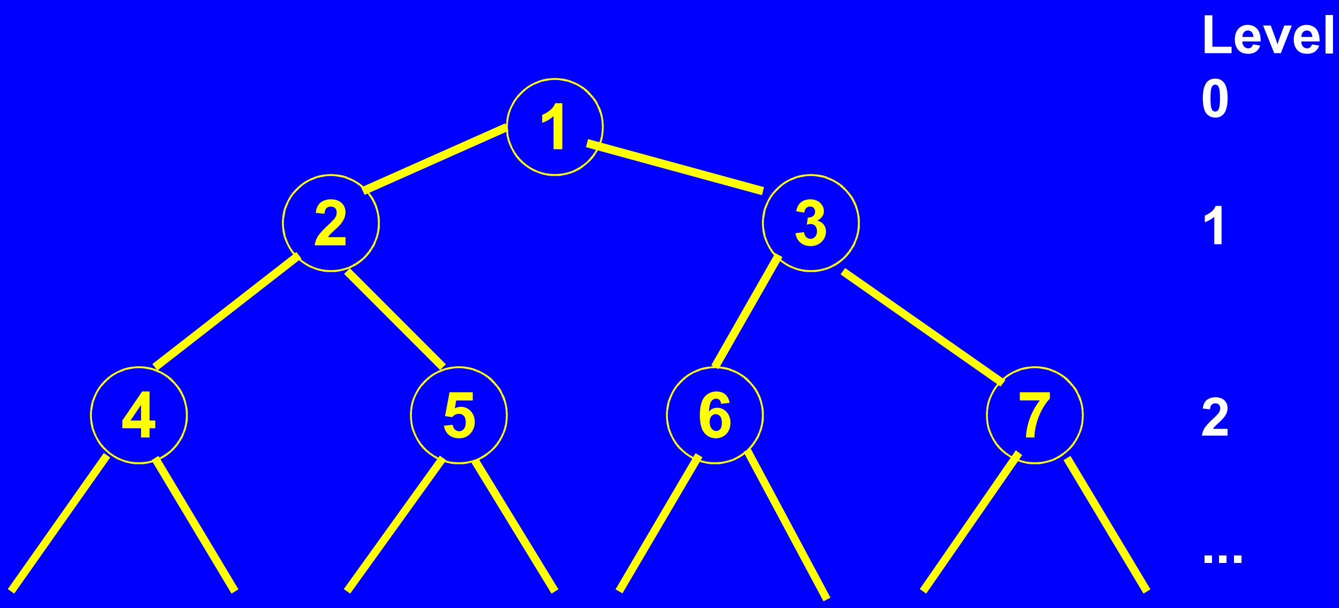
Tree definitions



Root: a
Degree (of node): number of subtrees
b:3, c:0, d:2
Leaf: node of degree 0: e, c
Branch: node of degree >0
Depth: max level in tree

Children: of a are b, c, d
Parent: of g is b
Siblings: children of same parent: b, c, d
Degree of tree: max degree of its nodes(3)
Ancestors: nodes on path to root:
g's ancestors are b and a

Binary tree definitions



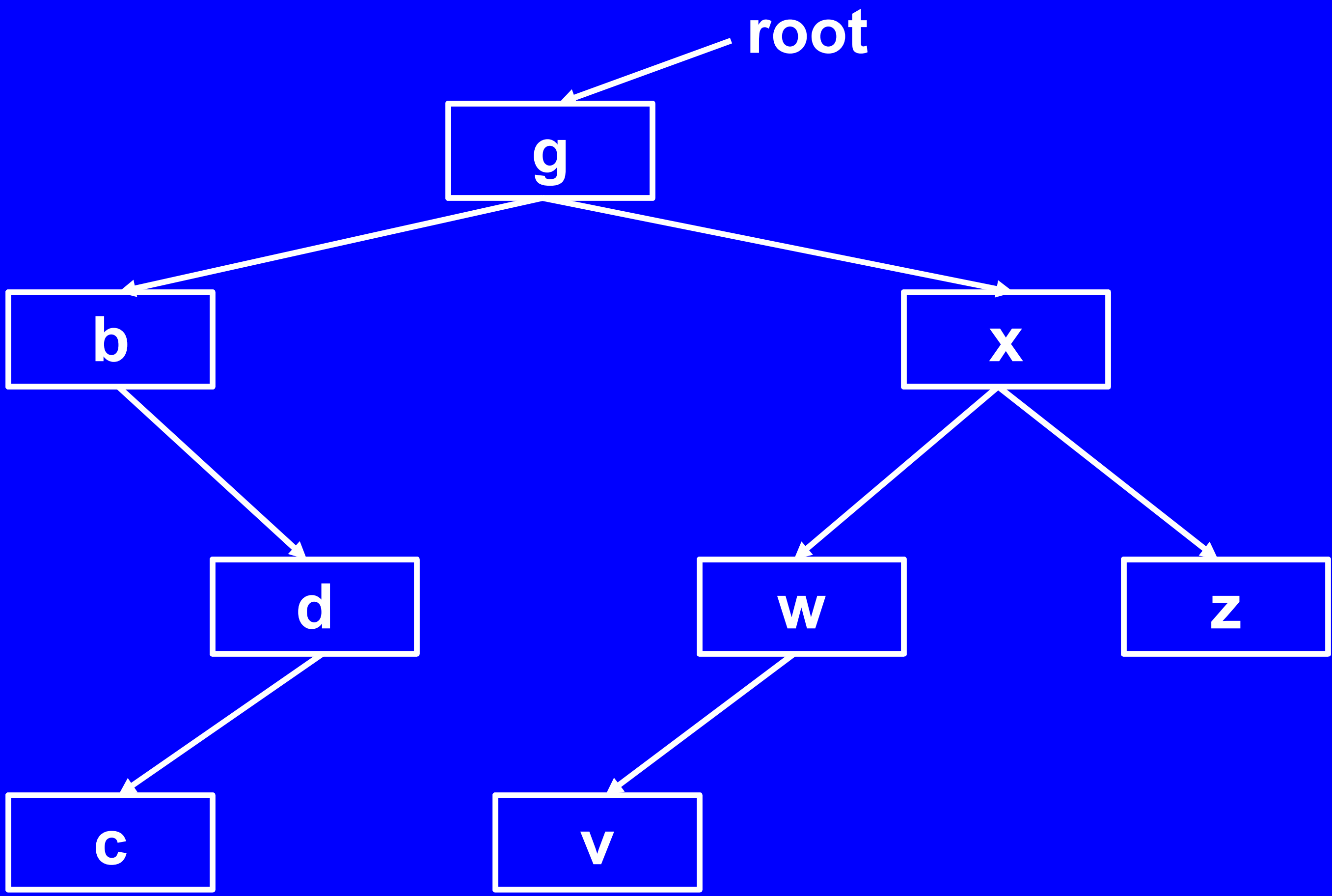
Max nodes on level i= 2^i
Max nodes in tree of depth k= $2^{k+1}-1$
(full tree of depth k)

Binary tree in a 1-D array:
Parent[i]= i/2
LeftChild[i]= 2i
RightChild[i]= 2i+1

Tree Traversal

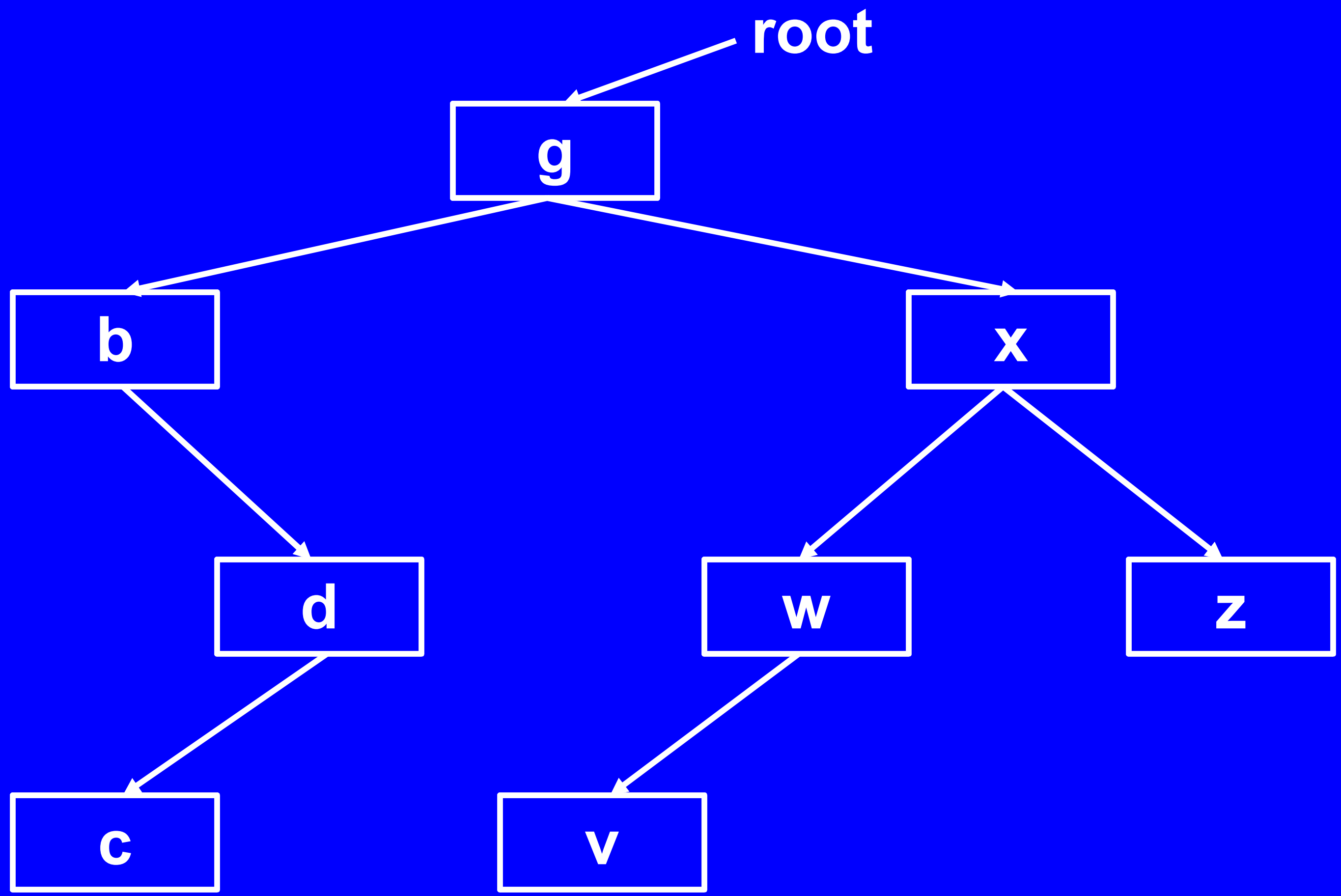
- Listing all the elements of a tree is more subtle than listing all the elements of a linked list, and there are a number of ways we can do it.
- We call a list of a tree's nodes a traversal if it lists each tree node exactly once.
- The three most commonly used traversal orders are recursively described as:
  - Inorder: traverse left subtree, visit current node, traverse right subtree
  - Postorder: traverse left subtree, traverse right subtree, visit current node
  - Preorder: visit current node, traverse left subtree, traverse right subtree

Tree traversal examples



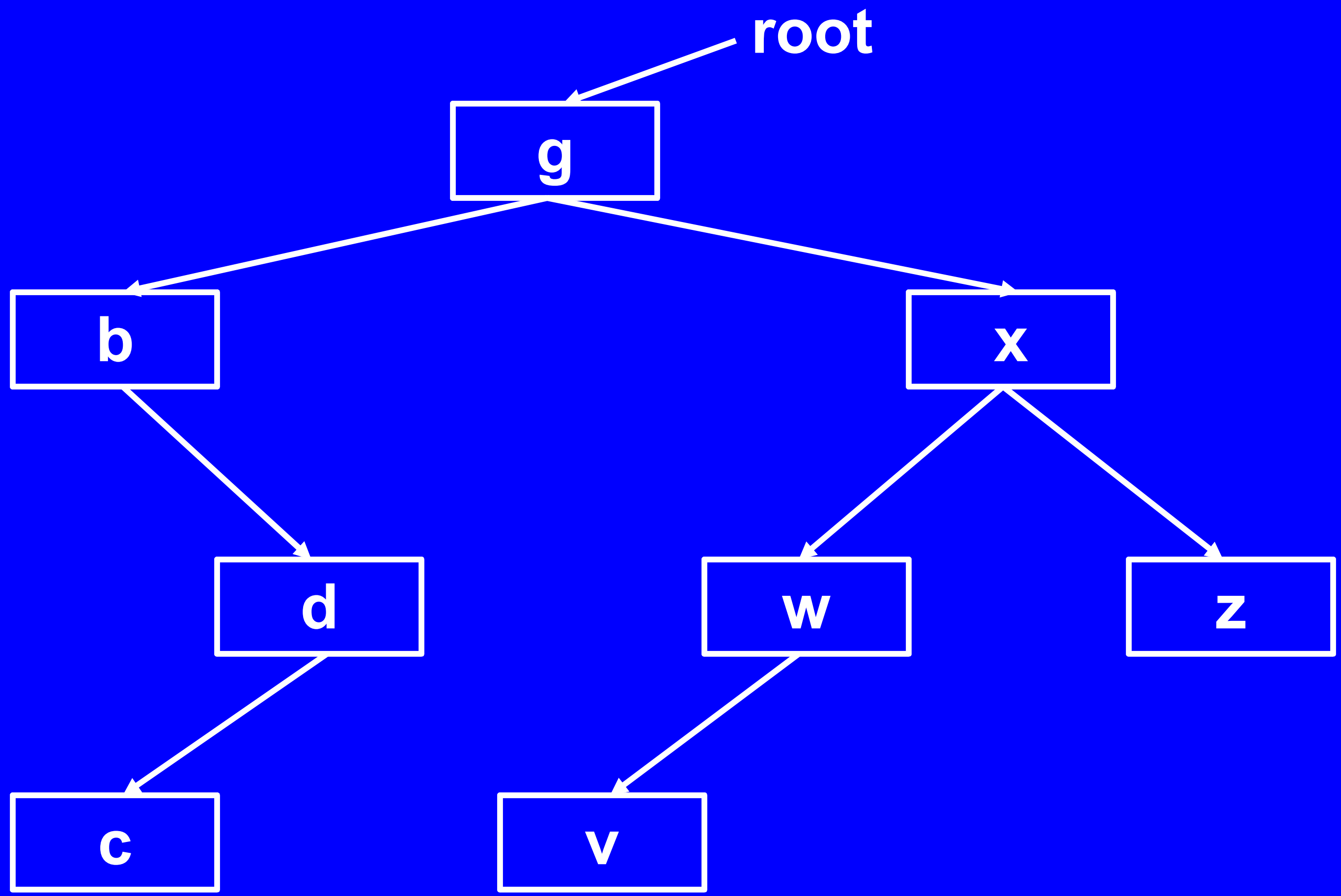
Inorder: start at root

Tree traversal examples



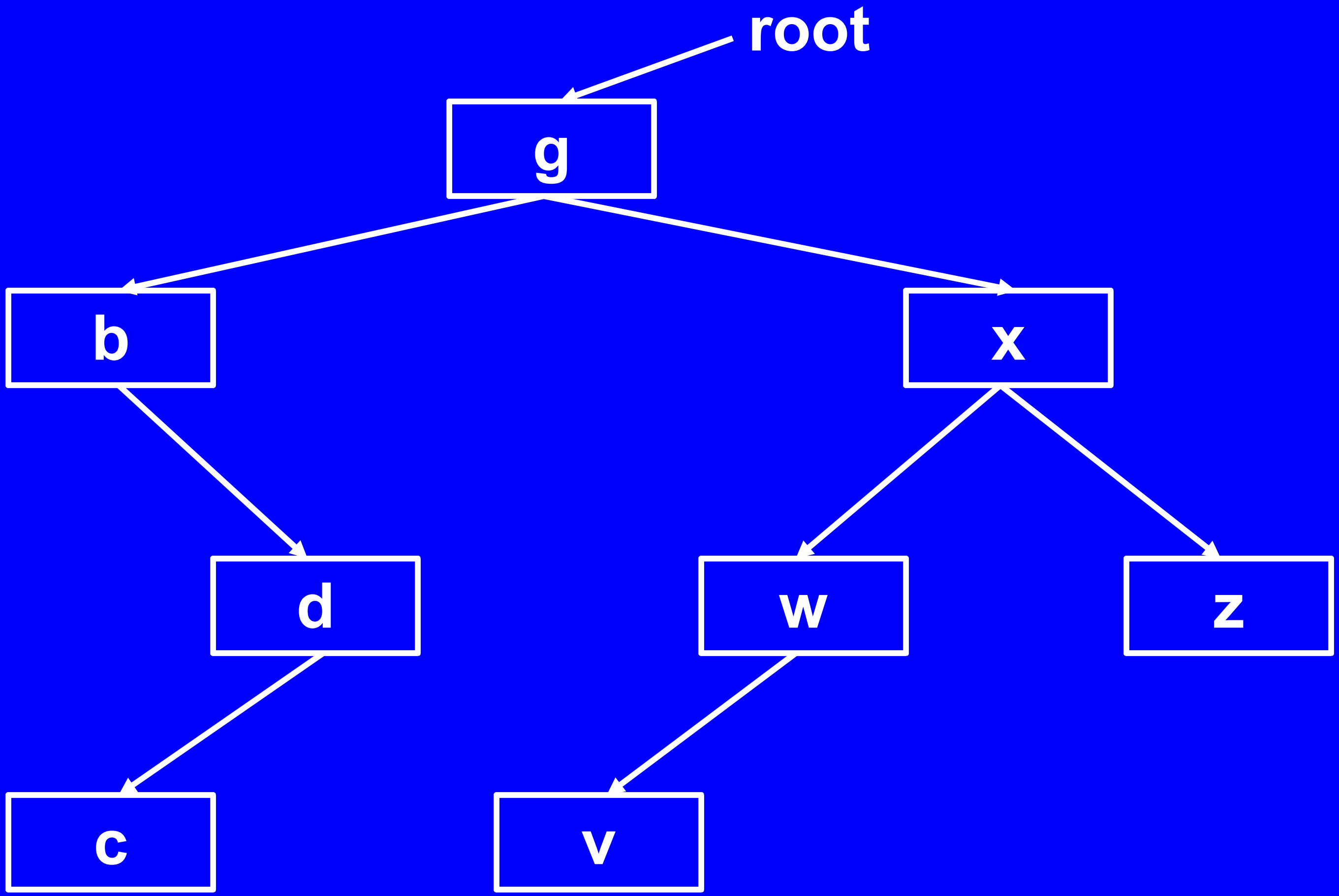
Inorder: b

Tree traversal examples



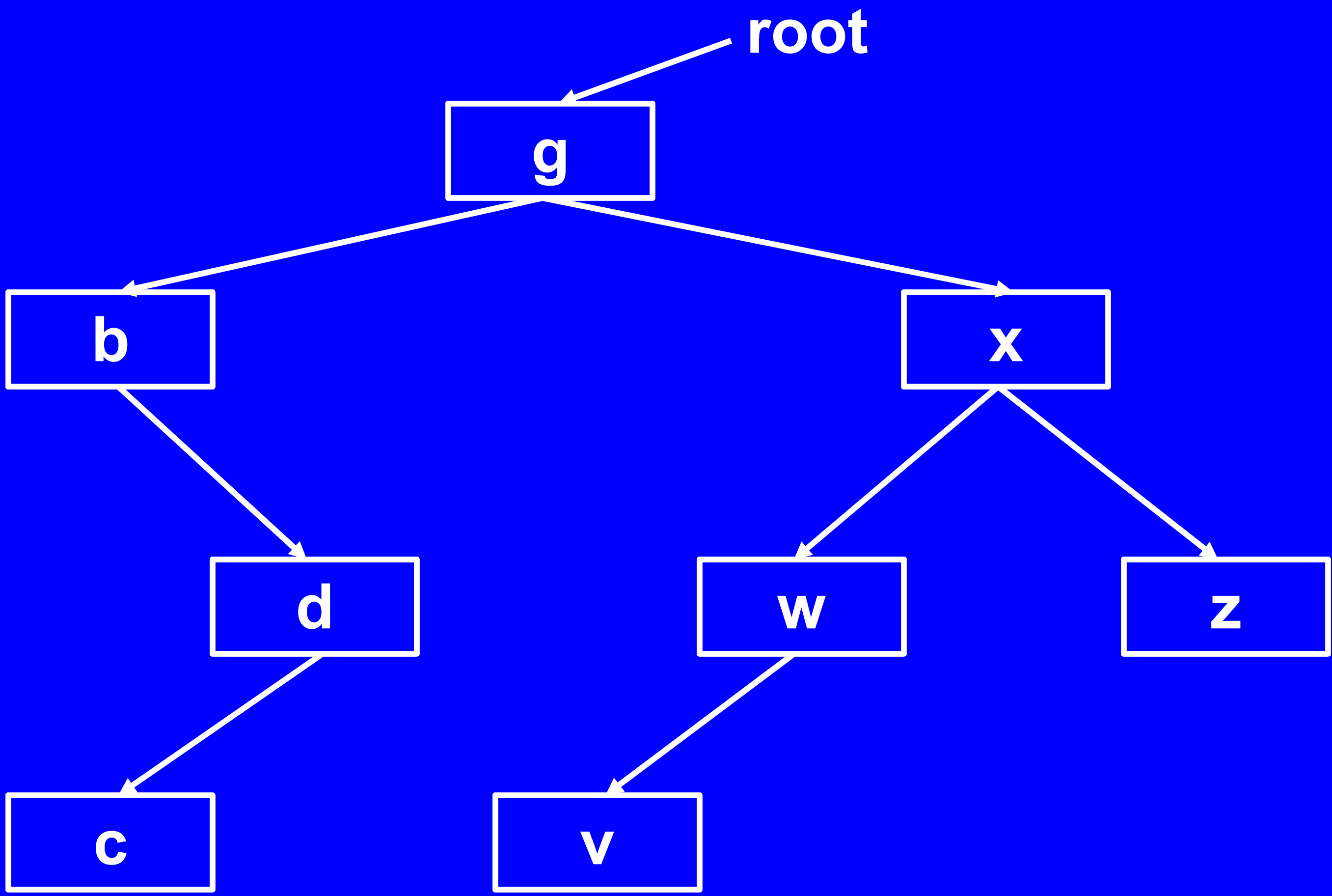
Inorder: b c

Tree traversal examples



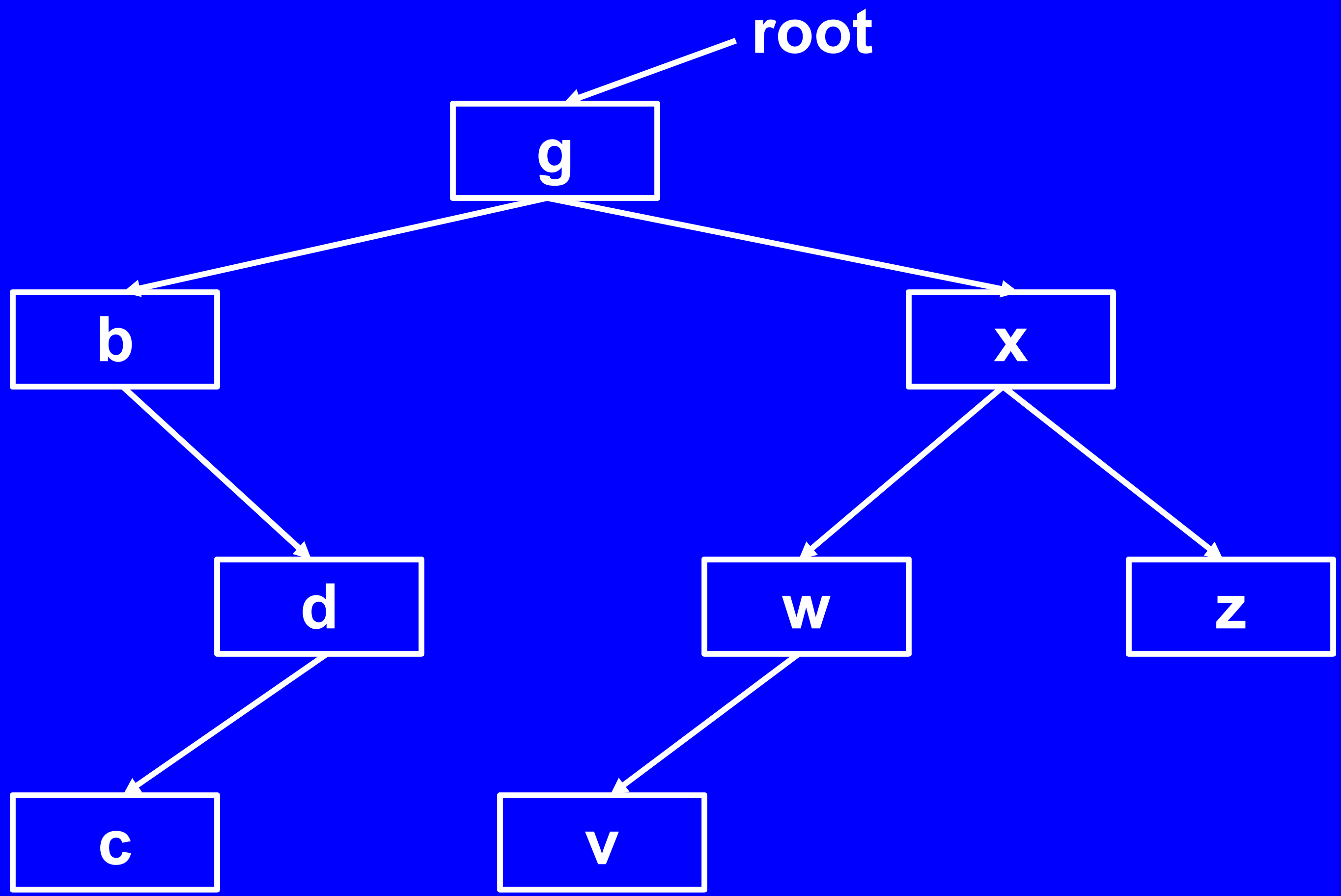
Inorder: b c d

Tree traversal examples



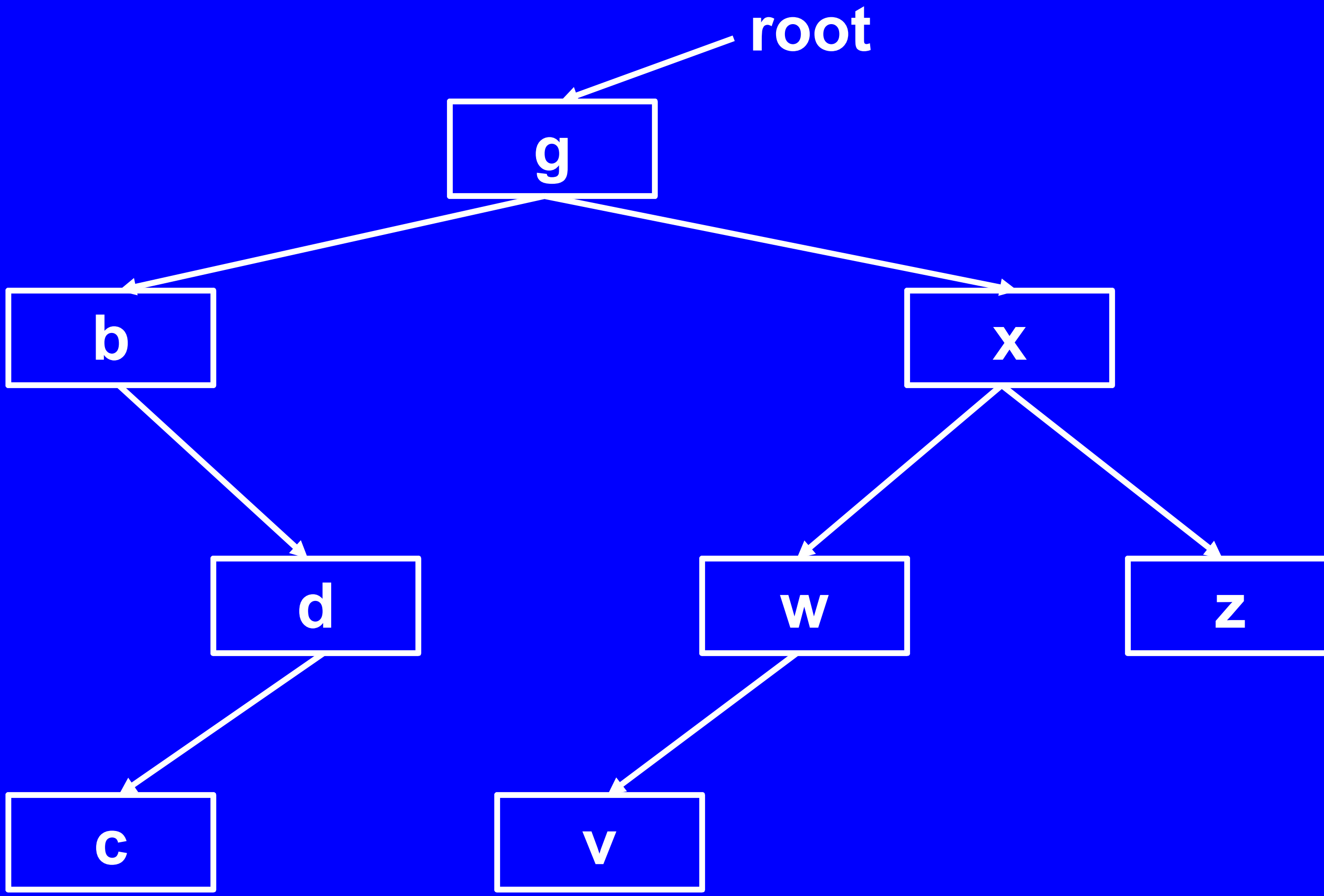
Inorder: b c d g

Tree traversal examples



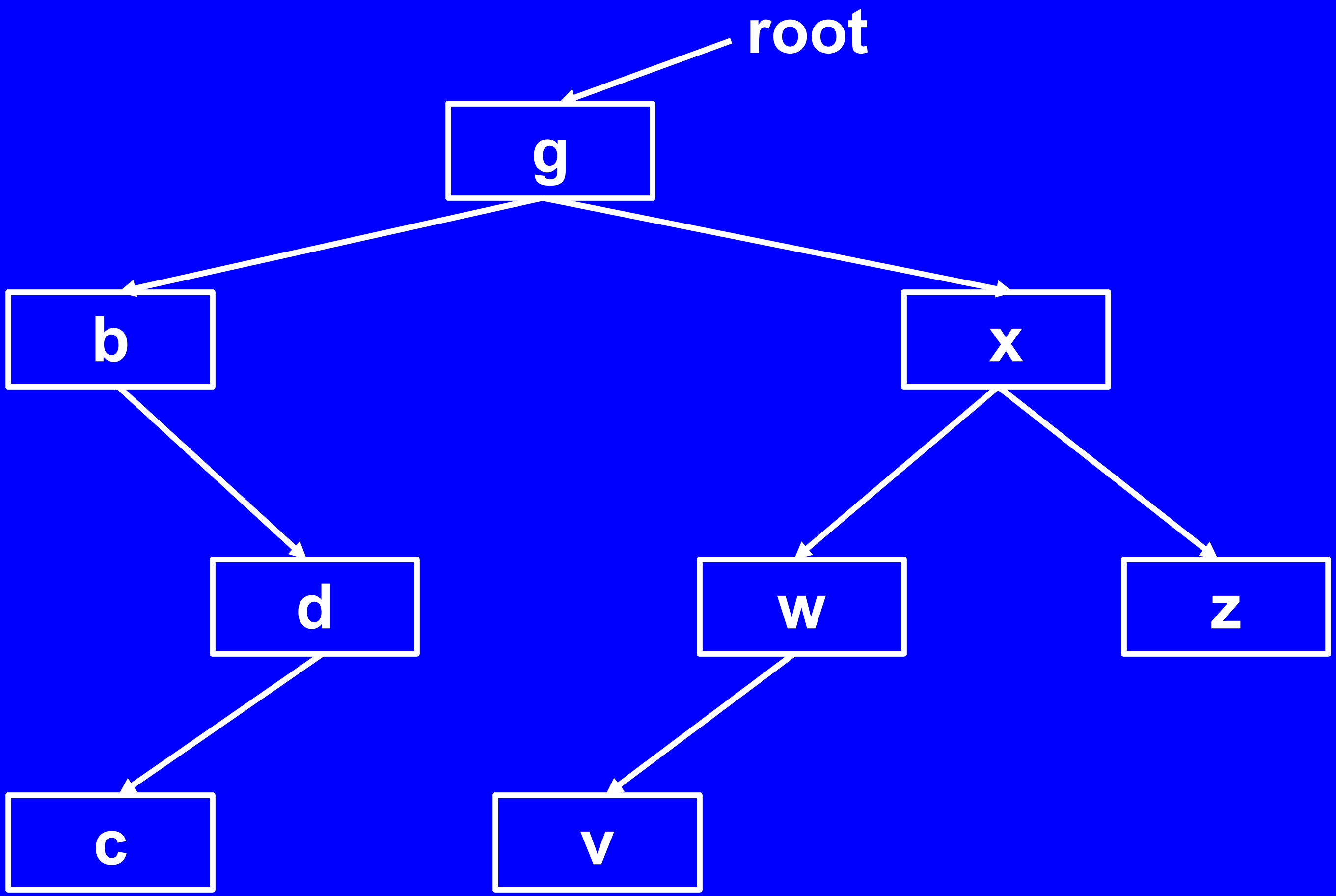
Inorder: b c d g v

Tree traversal examples



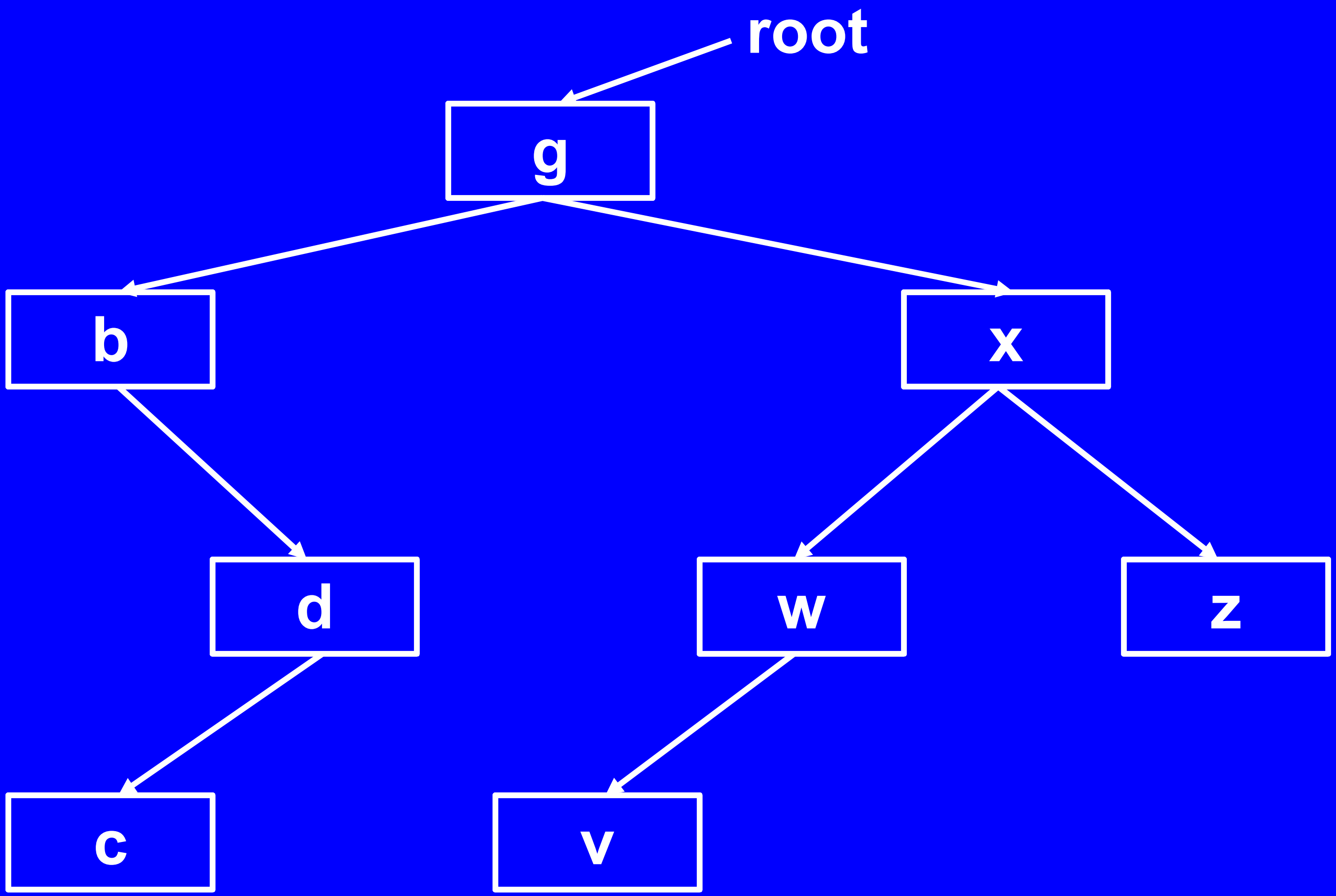
Inorder: b c d g v w x z

Tree traversal examples



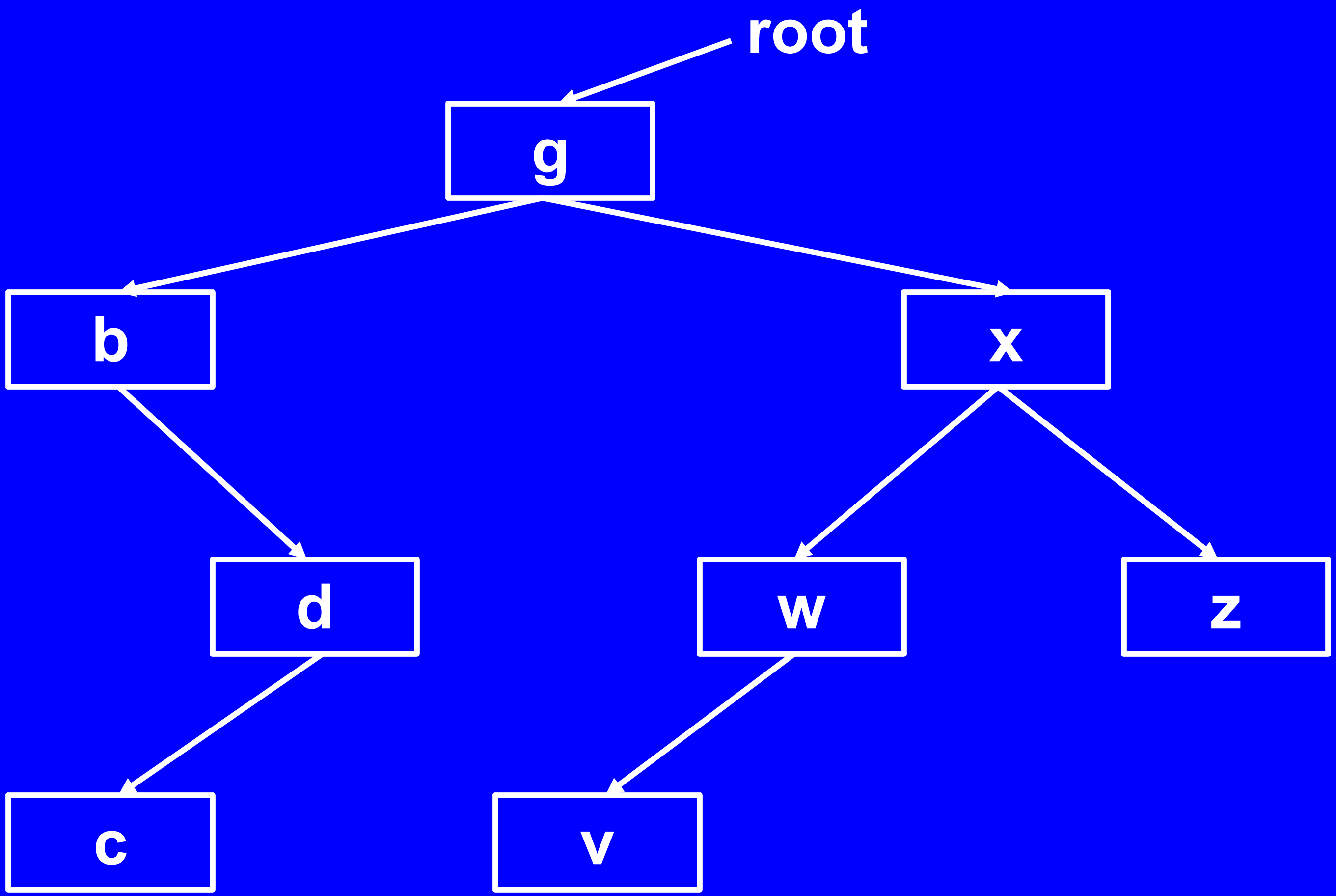
Postorder: start at root

Tree traversal examples



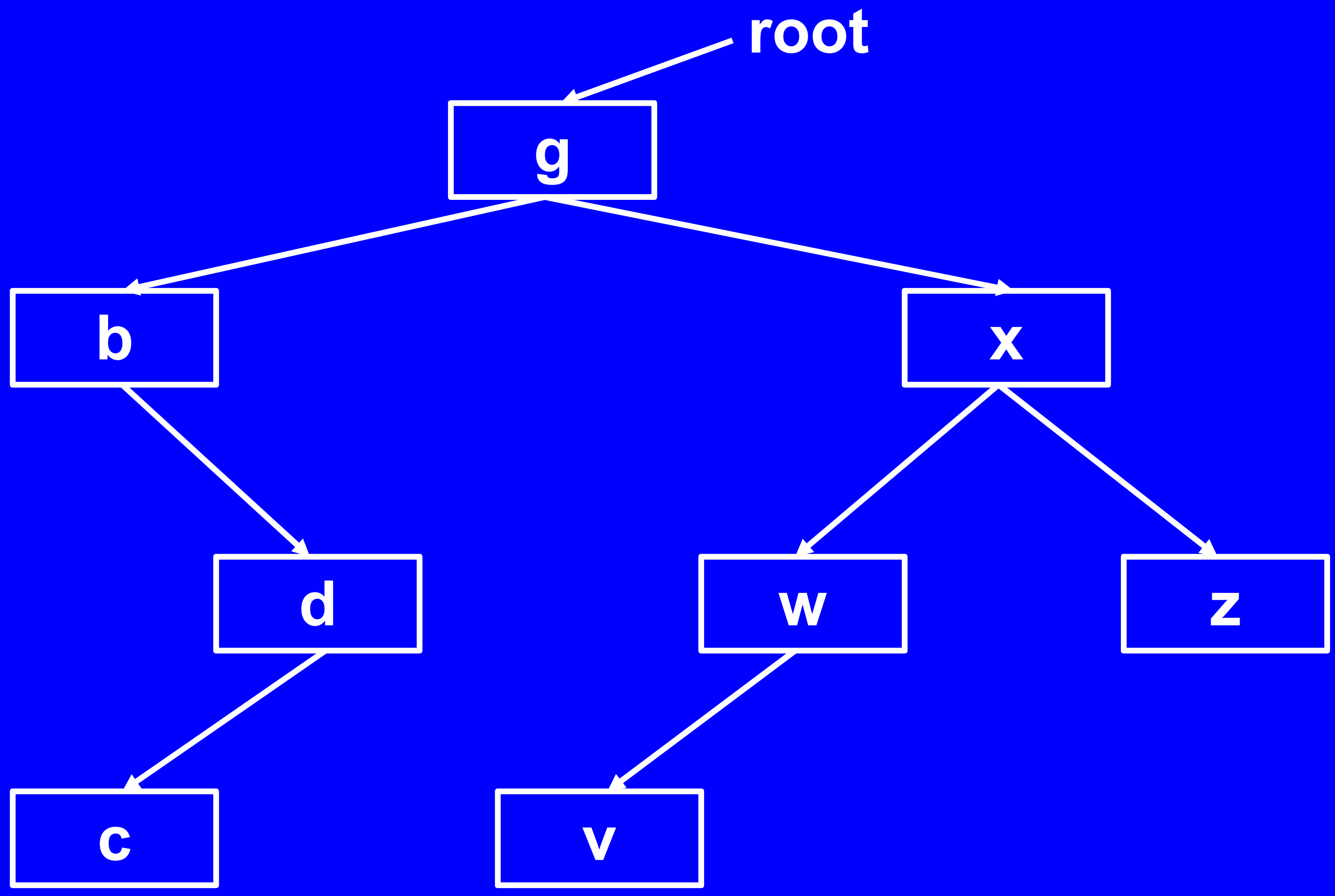
Postorder: c

Tree traversal examples



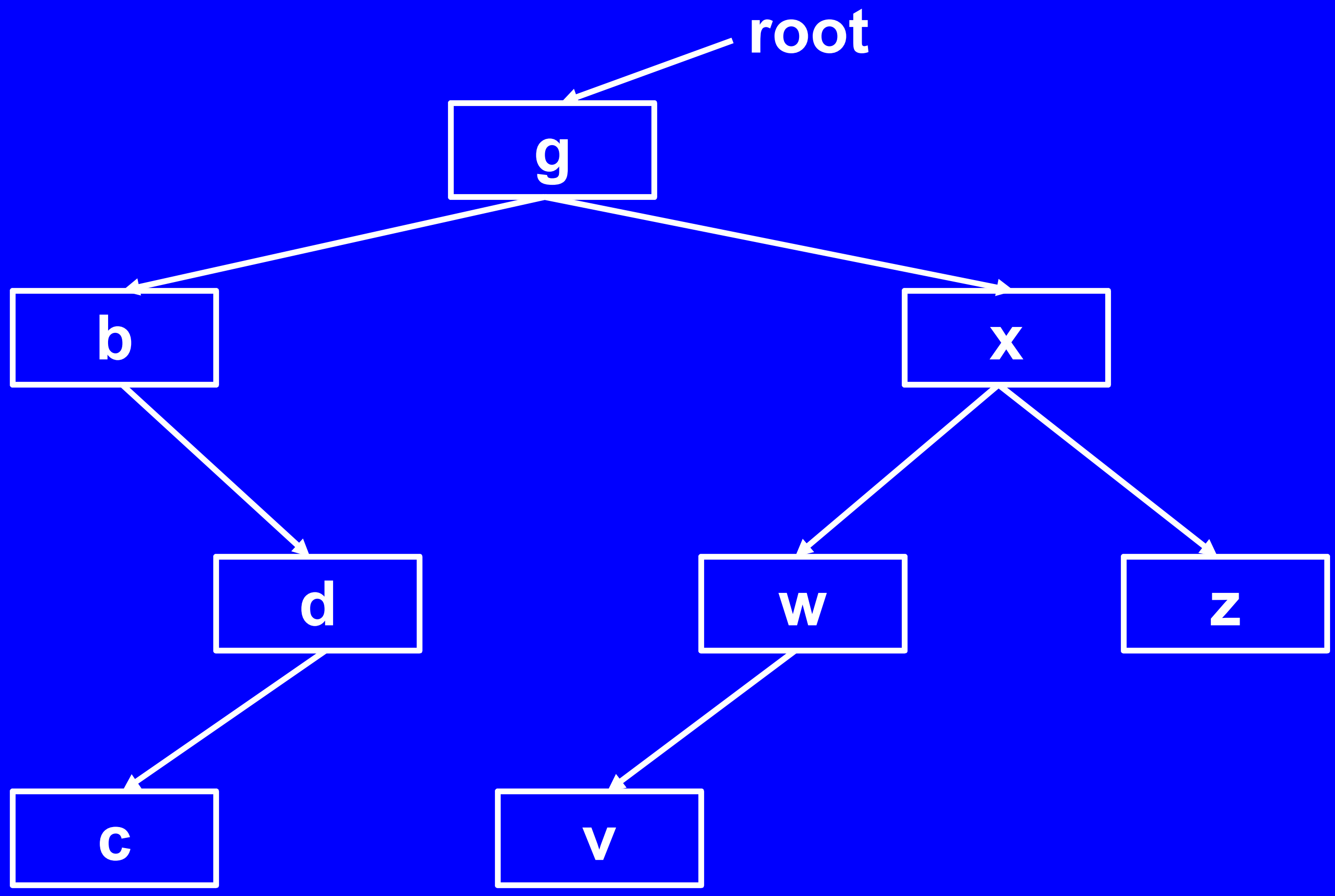
Postorder: c d

Tree traversal examples



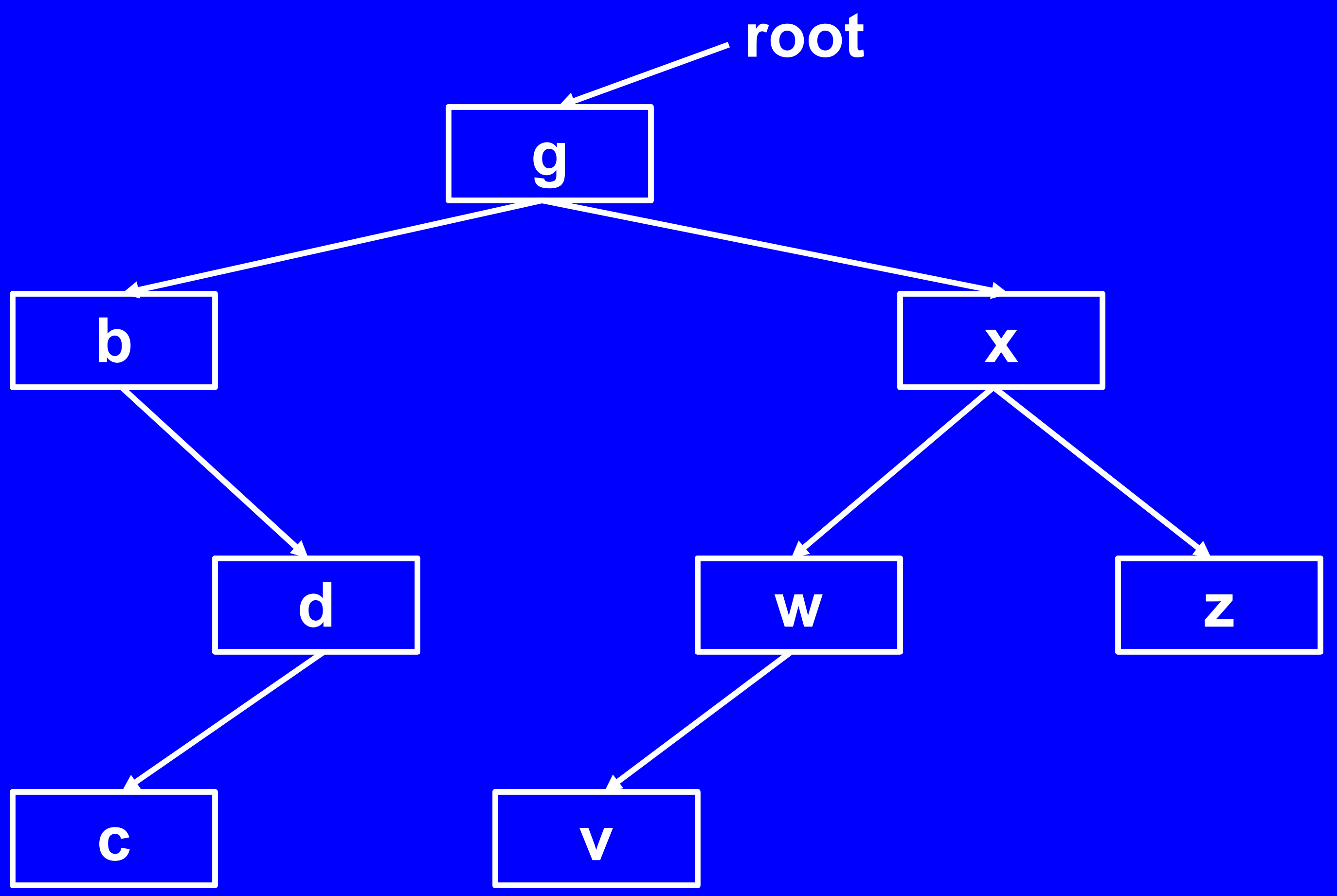
Postorder: c d b

Tree traversal examples



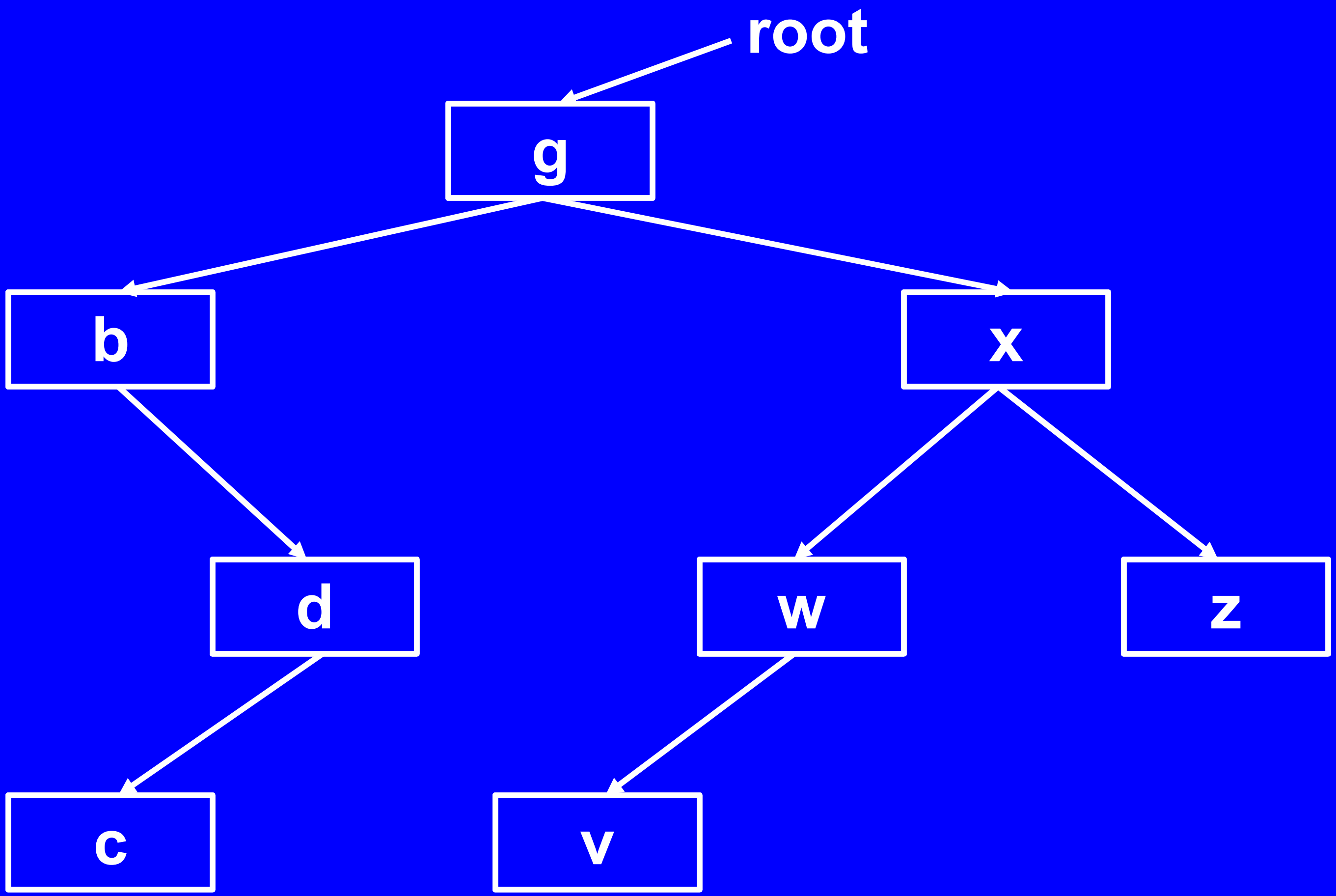
Postorder: c d b v

Tree traversal examples



Postorder: c d b v w

Tree traversal examples



Postorder: c d b v w z x g

Tree Traversal Exercise

- Download:
  - TreeTraversalApp
  - TreeTraversalView
  - VisualTreeNode
  - Screen
- Run TreeTraversalApp
  - Use the buttons on the bottom to explore tree definitions
  - Use the buttons on the top to explore the three typical tree traversals: inorder, preorder, and postorder.

Binary Search Trees

- There are many ways to build binary trees with varying properties:
  - In a heap or priority queue, the largest element is on top. In the rest of the heap, each element is larger than its children
  - In a binary search tree, the left subtree has nodes smaller than the parent, and the right subtree has nodes bigger than the parent
    - We saw that performing an *inorder* traversal of such a tree visited each node in order
- We'll build a binary search tree in this lecture

Writing a Binary Search Tree

- We'll build a `Tree` class:
  - One data member: `root`
  - One constructor: `Tree()`
  - Methods:
    - `insert`: build a tree, node by node
    - `inorder` traversal
    - `postorder` traversal
    - (we omit preorder)

Writing a BST, p.2

- We also build a `Node` inner class:
  - Three data members: `data, left, right`
    - `data` is a reference to an `Object`, so our `Node` is general
    - Our `data` Objects must implement the `Comparable` interface, which has one method:

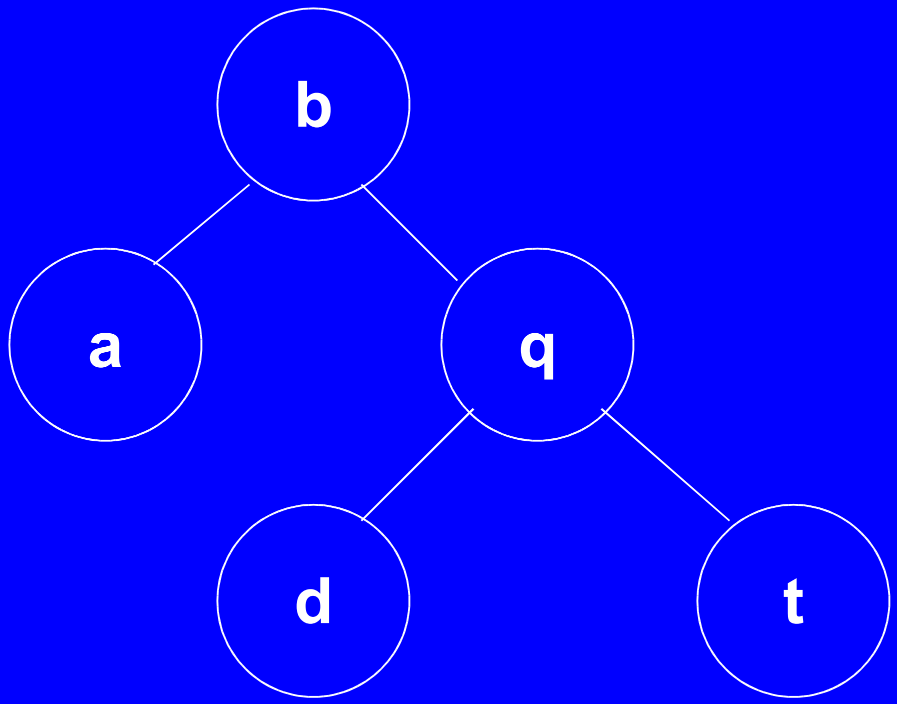
      ```
      int compareTo(Object other)
      ```
    - `compareTo` returns:
      - An int < 0 if (other < this)
      - 0 if (other equals this)
      - An int > 0 if (other > this)
  - Three methods, all used by corresponding methods in the `Tree` class:
    - `insertNode`
    - `traverseInorder`
    - `traversePostorder`
  - Methods are invoked on root node and then traverse the tree as needed

Exercise 1

- Draw the binary search tree that results from:

```
public class TreeTest {
    public static void main(String[] args) {
        Tree z= new Tree();
        z.insert("b");
        z.insert("q");
        z.insert("t");
        z.insert("d");
        z.insert("a");
        // Four more lines to appear in exercise 2
    }
}
```

Solution



Exercise 2

- What is the output if main() then contains the following 4 lines?

```
System.out.println("Inorder");
z.inorder();
System.out.println("Postorder");
z.postorder();
```

- Inorder is:
  - traverse left, visit (print) current, traverse right
- Postorder is:
  - Traverse left, traverse right, visit (print) current

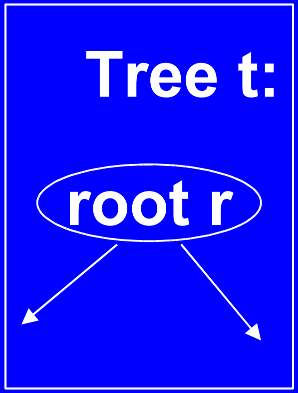
Solution

- Inorder:
  - a b d q t
- Postorder:
  - a d t q b

Tree and Node Classes

```
Tree:
  private Node root;
  public Tree() {root=null;}
  public void inorder() {…}
  public void postorder() {…}
  public void insert(n) {…}
  public boolean find(o) {…}
  public void print() {…}
```

Tree methods invoked on Tree object; they call Node methods invoked on the root node object




```
Node:
  public Comparable data;
  public Node left, right;
  public Node(o) {data=o;}
  public void traverseInorder(n) {…}
  public void traversePostorder(n) {…}
  public void insertNode(n) {…}
  public boolean findNode(o) {…}
  public void printNodes() {…}
```

Tree class

```
public class Tree {
    private Node root;

    public Tree() {
        root= null;     }

    public void inorder() {
        if (root != null)  root.traverseInorder(root);    }

    public void postorder() {
        if (root != null)  root.traversePostorder(root); }

    public void insert(Comparable o) {
        Node t= new Node(o);
        if (root==null)
            root= t;
        else
            root.insertNode(t);     }
```

Tree class, p.2

```
public boolean find(Comparable o) {
    if (root== null)
        return false;
    else
        return root.findNode(o);
}

public void print() {
    if (root != null)
        root.printNodes();
}
```

Node class: data, constructor

```
private class Node {
    public Comparable data;
    public Node left;
    public Node right;

    public Node(Comparable o) {
        data= o;
        left= null;
        right= null;
    }
```

Exercise 3: traversal

- Download TreeX, which contains Node
  - Rename it Tree if you wish (Eclipse: Refactor->Rename)
- Write the two traversal methods in Node:

```
public void traverseInorder( Node n) {
     if ( n != null ) {
      // Traverse left subtree
      // Print current Node
      // Traverse right subtree
     }
  }

  public void traversePostorder( Node n) {
     if ( n != null ) {
      // Traverse left subtree
      // Traverse right subtree
      // Print current Node
     }
  }
```

Solution

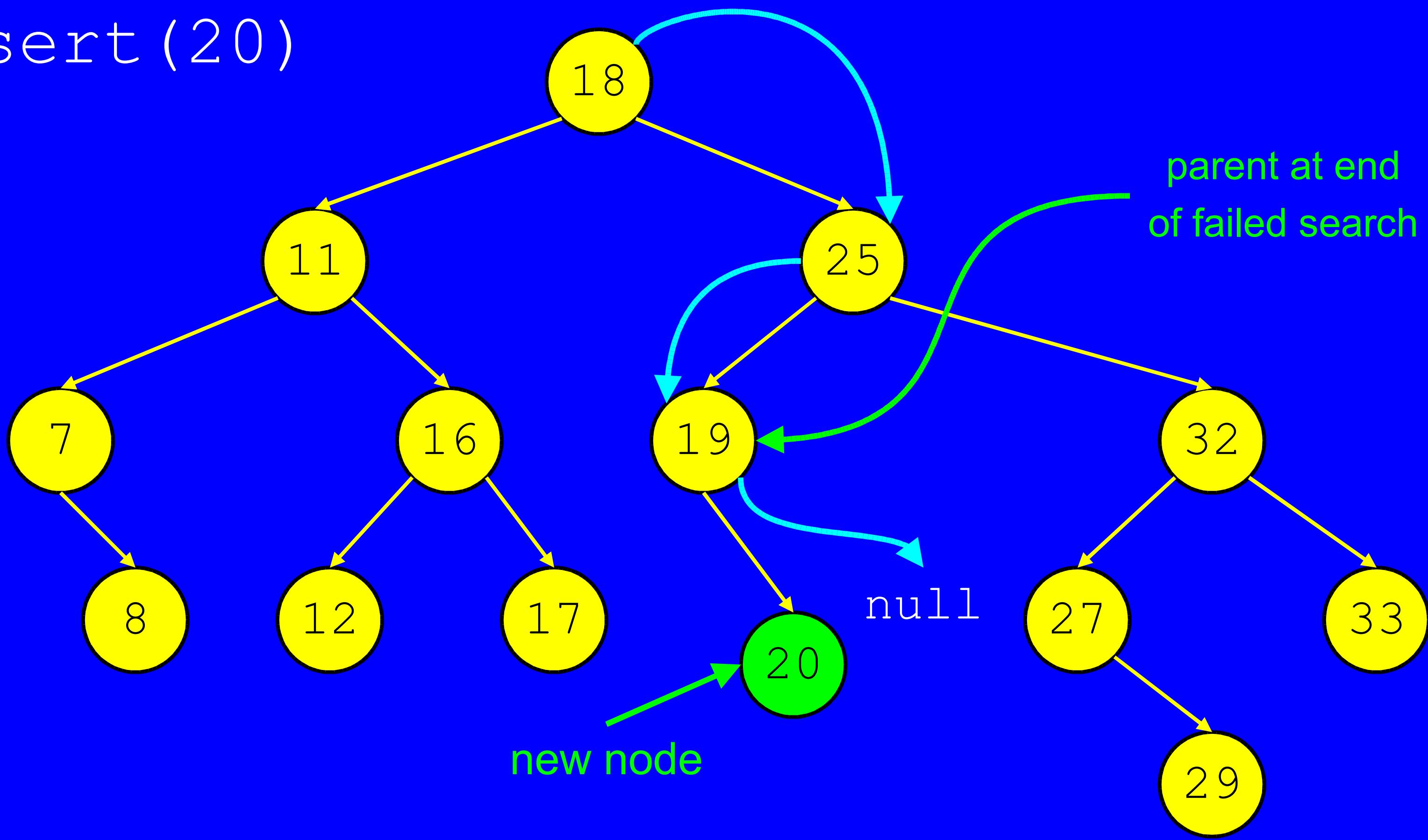
```
public void traverseInorder( Node n) {
     if ( n != null ) {
          traverseInorder( n.left);
          System.out.println( n.data);
          traverseInorder( n.right);
     }
 }

 public void traversePostorder( Node n) {
     if ( n != null ) {
          traversePostorder( n.left);
          traversePostorder( n.right);
          System.out.println( n.data);

     }
 }
```

Node class, insertNode

```
public void insertNode(Node n) {
    if (n.data.compareTo(data) < 0) {
        if (left==null)
            left= n;
        else
            left.insertNode(n);
    }
    else {
        if (right == null)
            right= n;
        else
            right.insertNode(n);
    }
}
```

`insert()` in Action

`insert(20)`



Exercise 4: Find Node

- This is very similar to insertNode:
  - In class Tree, we have:

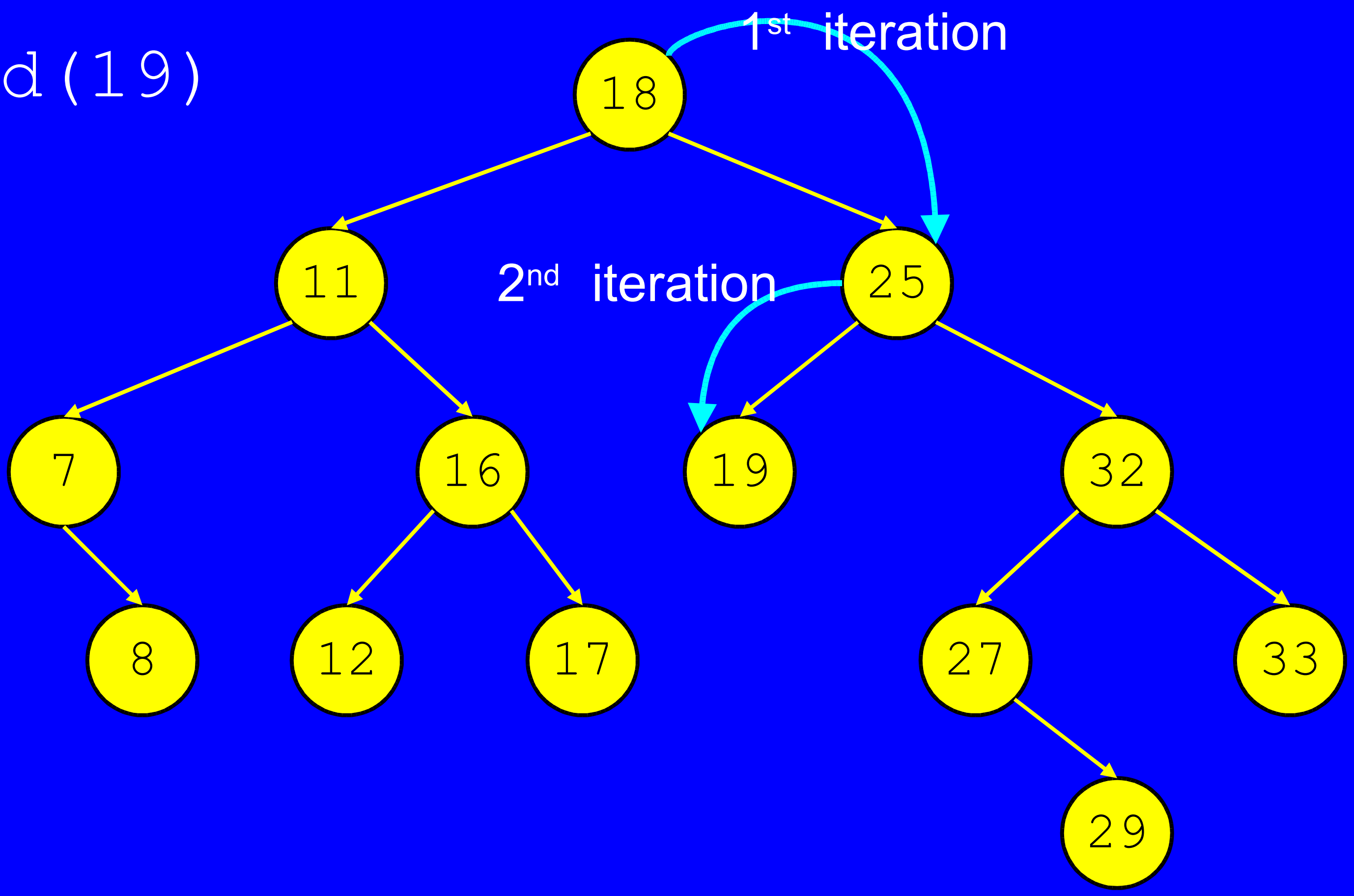
```
public boolean find(Comparable o) {
    if (root== null)
        return false;
    else
        return root.findNode(o);    }
```

  - In class Node, complete `findNode`:

```
public boolean findNode(Comparable o) {
    if (o.compareTo(data) < 0) {
        // Add code here to return result
    }
    else if (o.compareTo(data) > 0) {
        // Add code here to return result
    }
    else    // Equal
        return true;
}
```

`find()` in Action

`find(19)`



Solution

```
public boolean findNode(Comparable o) {
           if (o.compareTo(data) < 0) {
               if (left== null)
                   return false;
               else
                   return left.findNode(o);
           }
           else if (o.compareTo(data) > 0) {
               if (right == null)
                   return false;
               else
                   return right.findNode(o);
           }
           else    // Equal
               return true;
}
```

Exercise 5: Test

- Download TreeTest
- Run it to check if your Tree class gives the correct answers from Exercises 1 and 2
  - Inorder: a b d q t
  - Postorder: a d t q b
  - See if it finds b, a, d, q, t (yes) and x (no) in the tree

Keys and Values

- If binary search trees are ordered, then they must be ordered on some *key* possessed by every tree node.
- A node might contain nothing but the *key*, but it's often useful to allow each node to contain a *key* and a *value*.
- The *key* is used to look up the node. The *value* is extra data contained in the node indexed by the *key*.

Maps

- Such data structures with key/value pairs are usually called *maps*.
- As an example, consider the entries in a phone book as they might be entered in a binary search tree. The subscriber name, last name first, serves as the *key*, and the phone number serves as the *value*.

Maps

- Implementing tree structures with keys and values is a straightforward extension to what we just did. The Node contains:
  - Key
  - Value
  - Left
  - Right
- We add or modify methods to set or get the values associated with the keys
  - No change in logic
- Map example on next slides
  - This could be improved by having find() return the Object instead of a boolean whether it was found
  - You'd then have to check if the object is null, etc.
  - These are straightforward changes, but we show the simples implementation here

Phone class

```
public class Phone implements Comparable {
    private String name;                    // Name of person (key)
    private int phone;                      // Phone number (value)

    public Phone(String n, int p) {
         name= n;
         phone= p;
    }

    public int compareTo(Object other) {
         Phone o= (Phone) other;
         return o.name.compareTo(this.name);        // String compare
    }

    public String toString() {
         return("Name: "+ name +" phone: "+ phone);
    }
}
```

MapTest

```
public class MapTest {
   public static void main(String[] args) {
        Tree z= new Tree();
        z.insert(new Phone("Betty", 4411));
        z.insert(new Phone("Quantum", 1531));
        z.insert(new Phone("Thomas", 6651));
        z.insert(new Phone("Darlene", 8343));
        z.insert(new Phone("Alice", 6334));
        z.print();
        System.out.println("Inorder");
        z.inorder();
        System.out.println("Postorder");
        z.postorder();
        System.out.println("Search for phone numbers");
        System.out.println("Find Betty? " +
                 z.find(new Phone("Betty", -1)));
        System.out.println("Find Thomas? " +
                 z.find(new Phone("Thomas", -1)));
        System.out.println("Find Alan? " +
                 z.find(new Phone("Alan", -1)));
    }
}
```

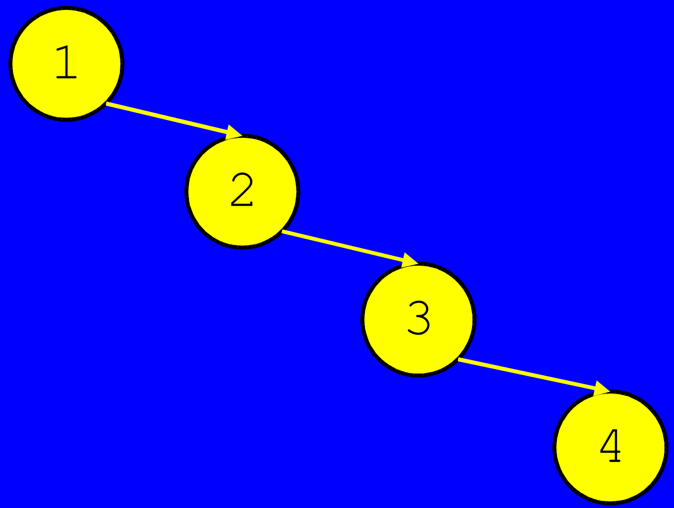
The Efficiency of Binary Search

- Intuitively, the basic operations on a binary search tree require *O(h)* time where h is the height of the tree.
- The height of a *balanced* binary tree is roughly $\log_2(n)$ where n is the number of elements if the tree remains approximately balanced.
- If keys are randomly inserted in a binary search tree, this condition will be met, and the tree will remain balanced enough so that search and insertion time will approximate *O(lg n)*.

Tree Balance

- There are some extremely simple and common cases, however, where keys will not be inserted in random order.
- Consider what will happen if you insert keys into a search tree from a sorted list. The tree will assume a degenerate form equivalent to the source list and search and insertion times will degrade to *O(n)*.
- There are many variants of trees, e.g., red-black trees, AVL trees, B-trees, that try to solve this problem by rebalancing the tree after operations that unbalance it.

Keys Inserted in Order



`delete()` Cases

Deleting nodes is messy. We just give a hint here:

There are three deletion cases we must consider:

1. **The deleted node has no children, e.g., node 29 below.**
2. **The deleted node has one child, e.g., node 7 below.**
3. **The deleted node has two children, e.g., node 25 below**