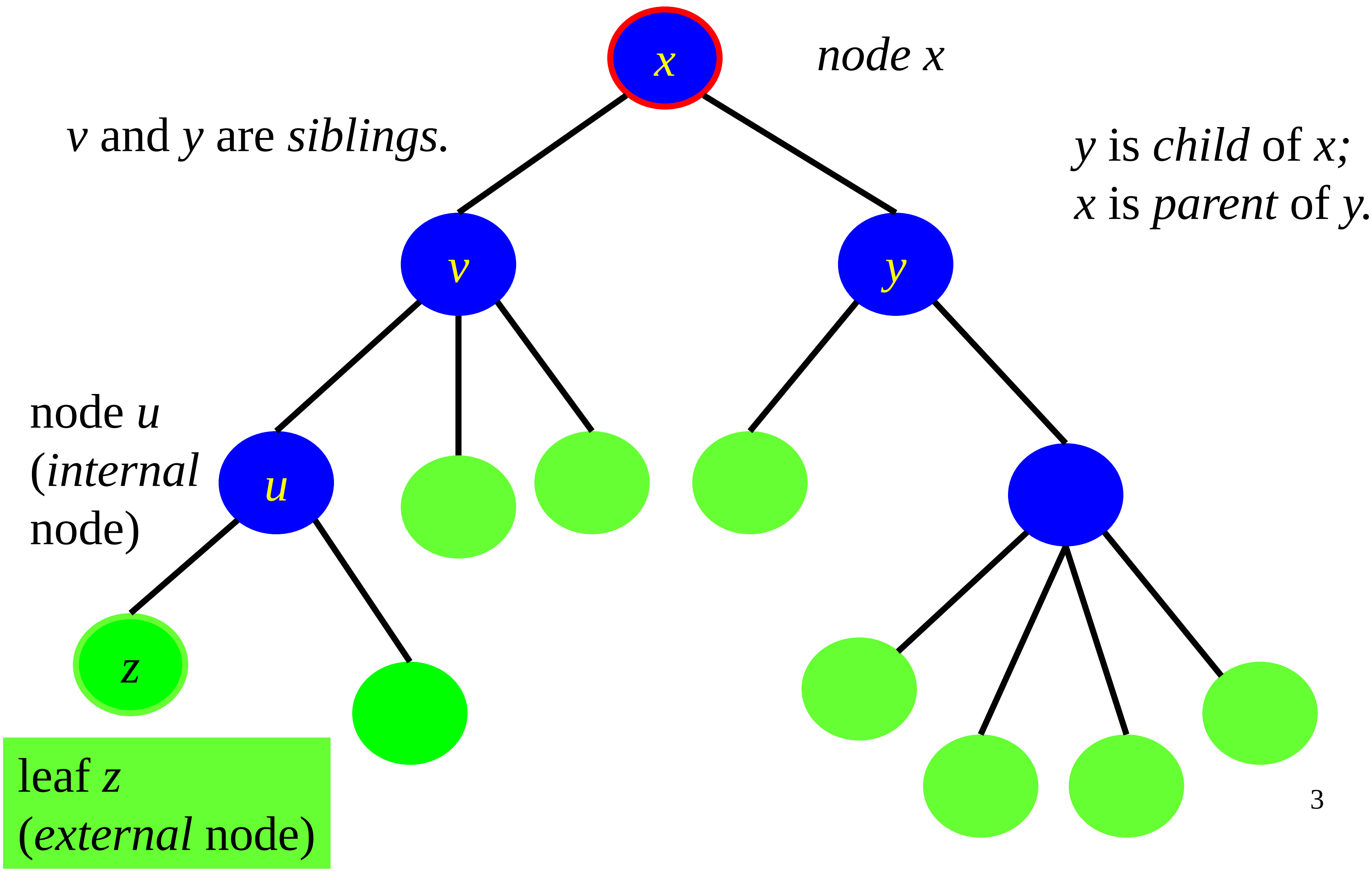
CSC 225

Algorithms and Data Structures I
Fall 2014
Rich Little

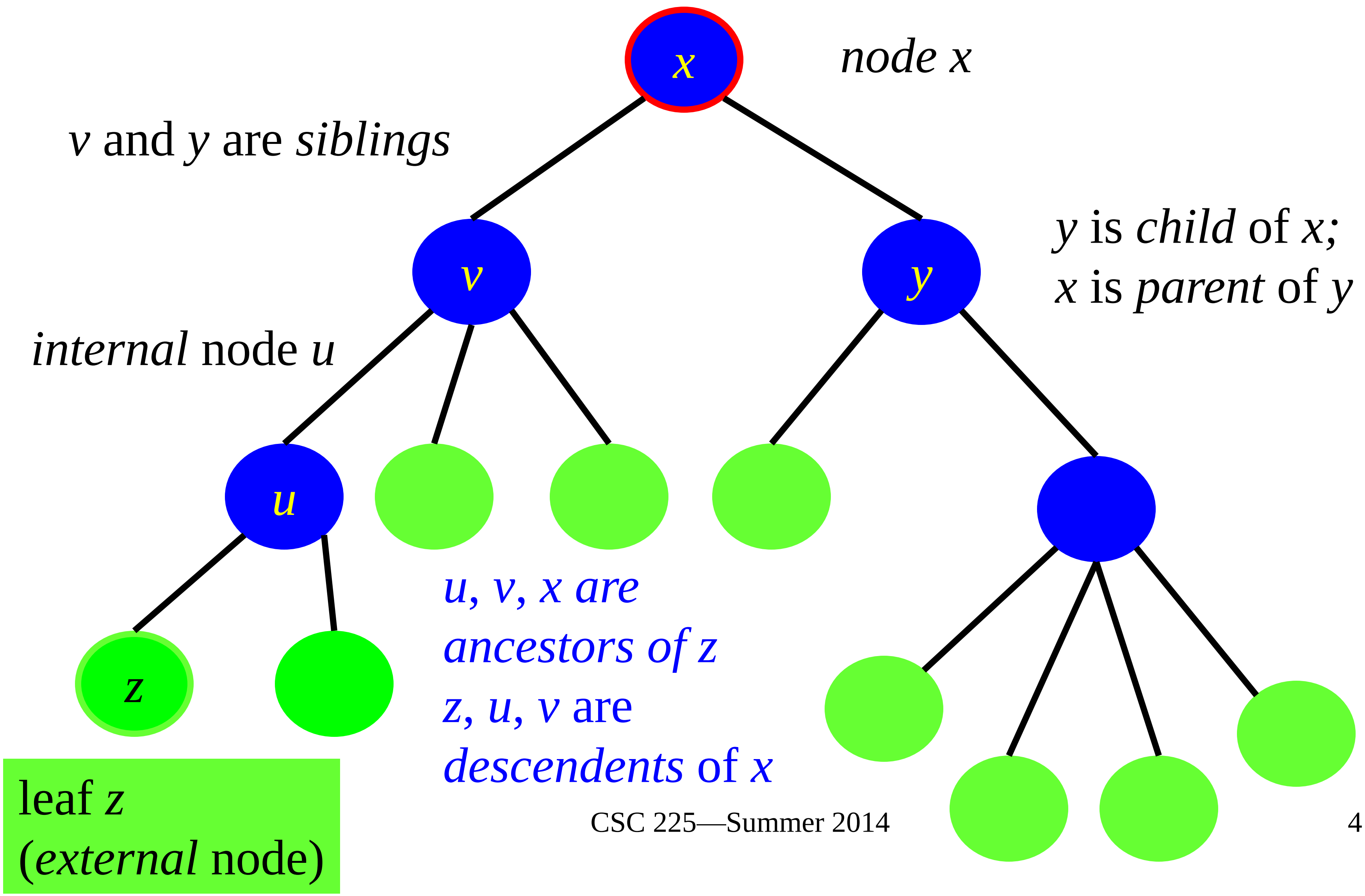
Trees

- A (*rooted*) *tree T* is a set of *nodes* in a *parent-child relationship* with the following properties:
  - ➢ *T* has a special node *r*, called the *root* of *T*
  - ➢ Each node *v* of *T* different from *r* has a *parent* node *u*

Trees


node *x*
v and *y* are *siblings*.
y is *child* of *x*;
x is *parent* of *y*.
x
v
y
node *u*
(*internal*
node)
u
z
leaf *z*
(*external* node)

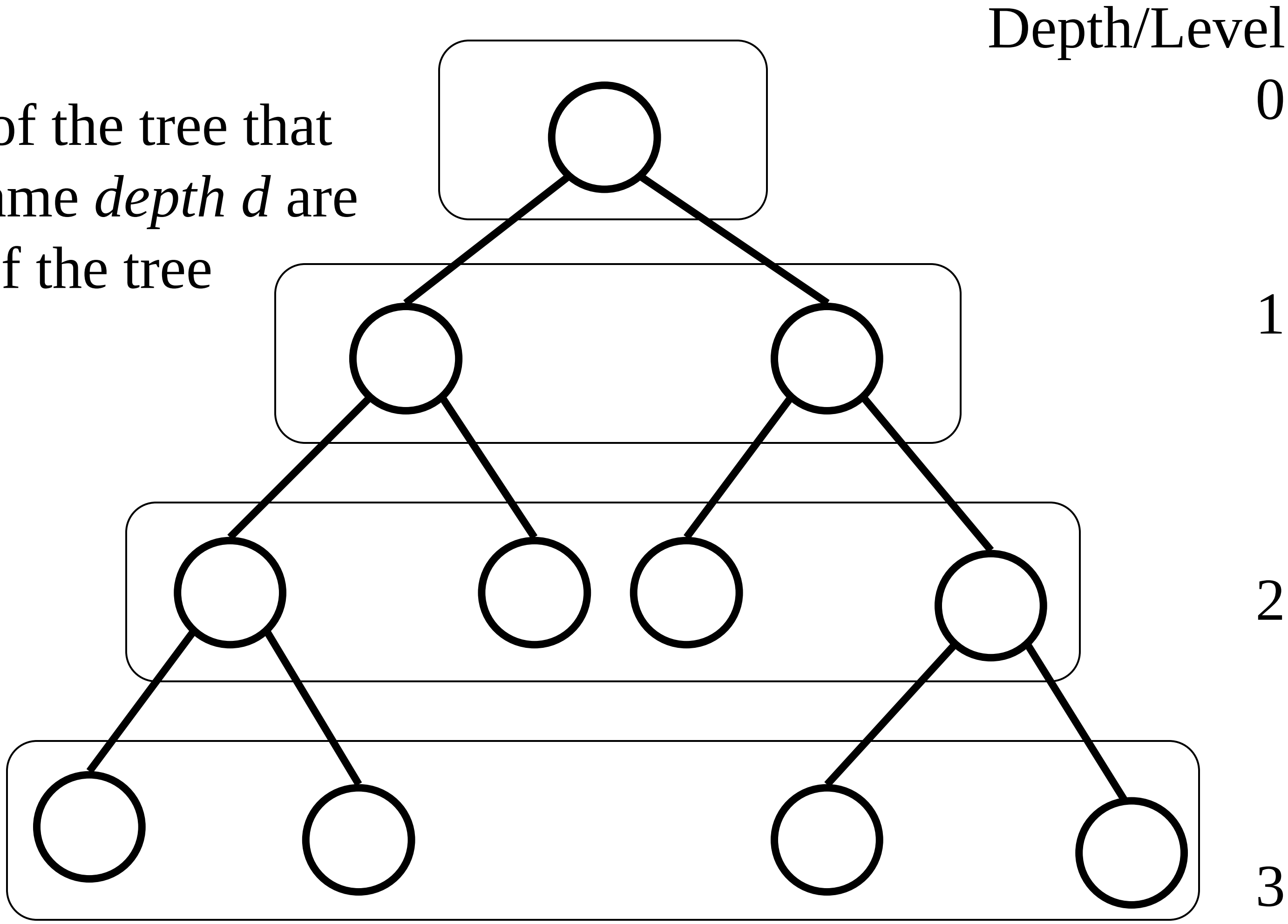
Trees



x
node x
v and y are siblings
v
y
y is child of x;
x is parent of y
internal node u
u
z
u, v, x are
ancestors of z
z, u, v are
descendents of x
leaf z
(external node)

Depth and Levels in Trees

All nodes of the tree that have the same *depth d* are at *level d* of the tree



Height of a Tree

Definition: The *height* of a a tree *T* rooted at node *v* is (recursively) defined to be

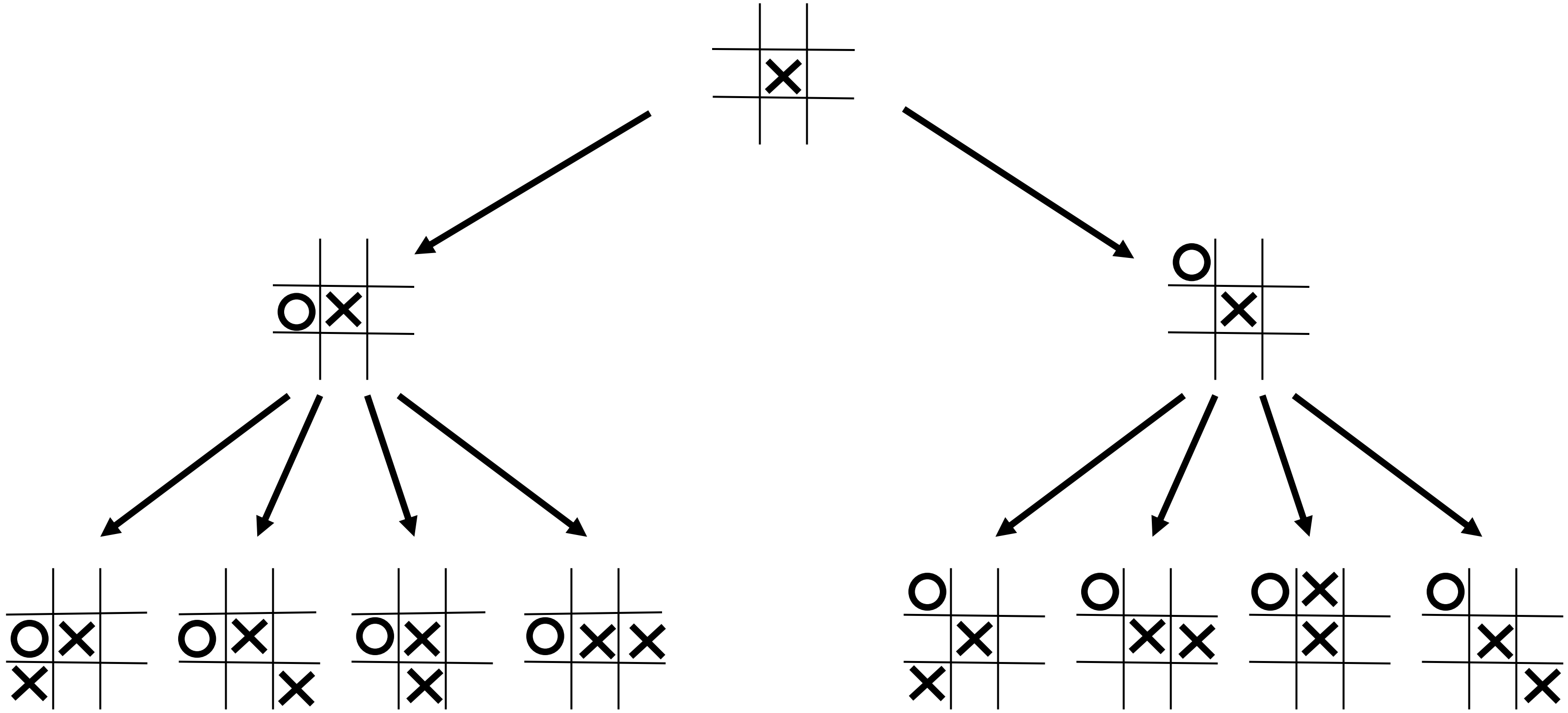
- The height is 0 if *v* is a leaf node
- The height is equal to 1 plus the maximum height of any child of *v*, otherwise.

Applications of Trees

- Phylogenetic trees
- Data structure (search trees)
- Search trees for exponential algorithms (branch-and-bound techniques, fixed-parameter tractable algorithms)
- Visualization of algorithms (and tool for complexity analysis)
- Decision trees
- Parse trees
- Expression trees
- Forests

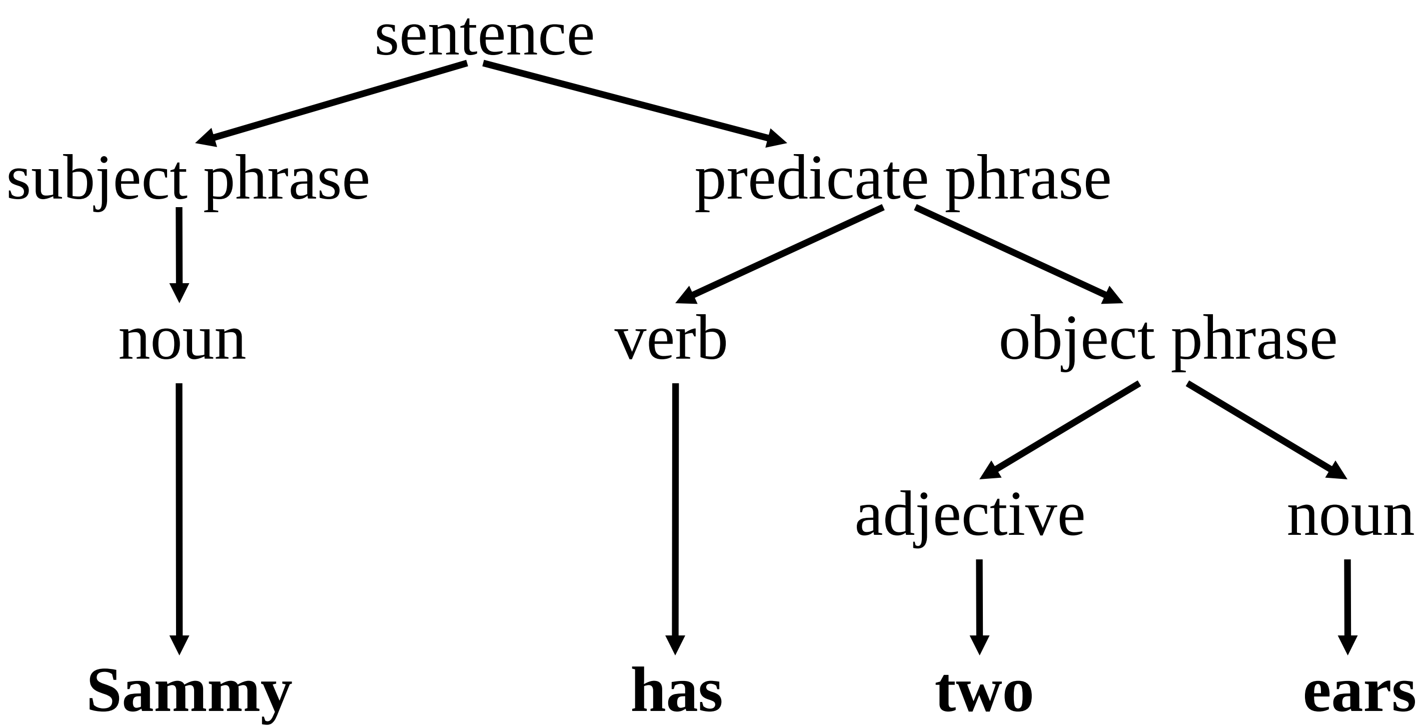
Decision Trees

(after Gross & Yellen, 1999, p. 93)



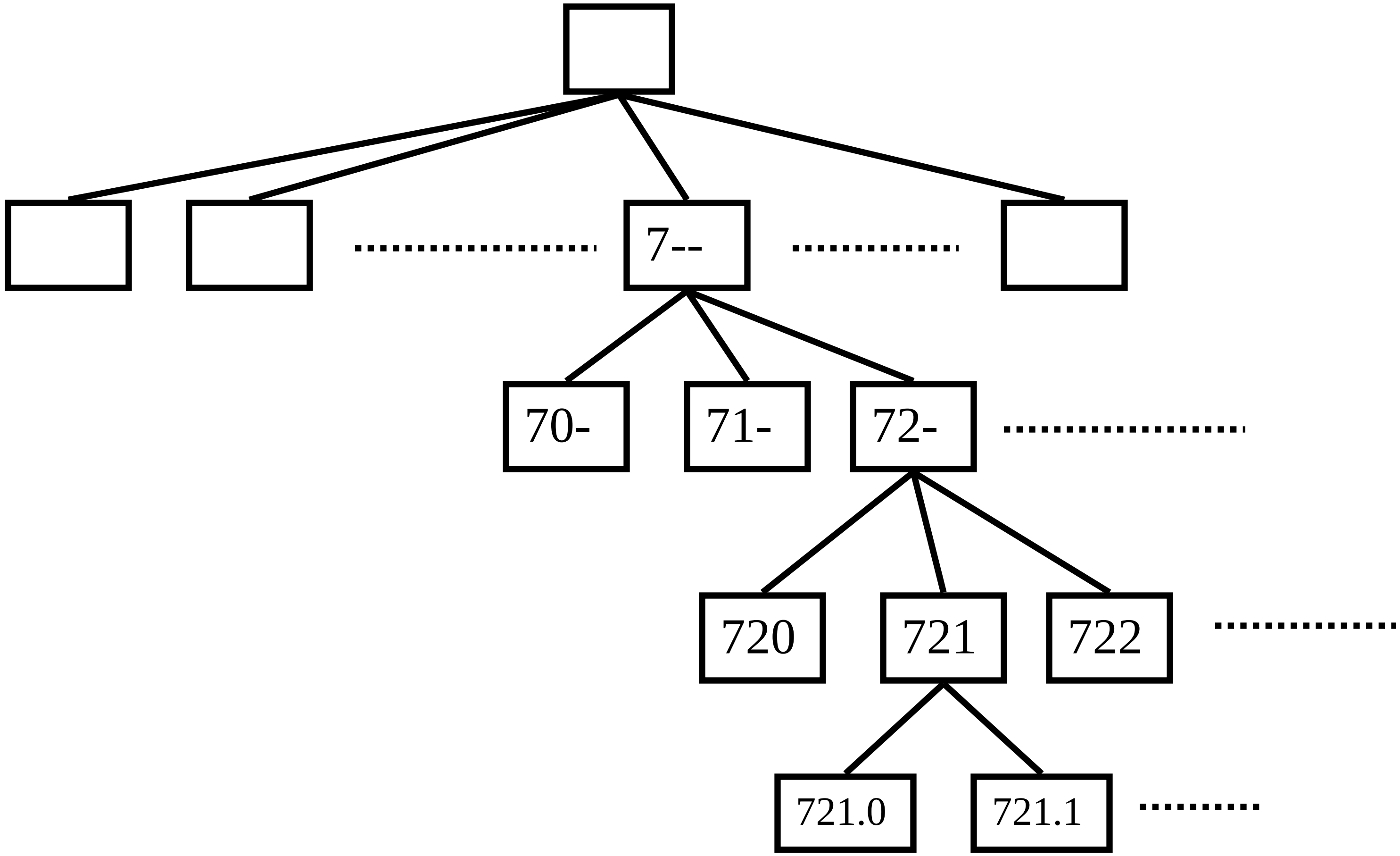
The first three moves of tic-tac-toe

Sentence Parsing
(after Gross & Yellen, 1999, p. 93)



Data Organization: DDCS for libraries

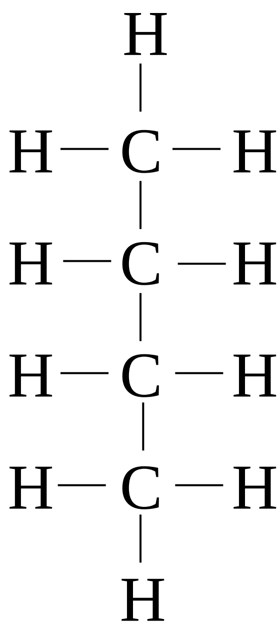
(after Gross & Yellen, 1999, p. 93)



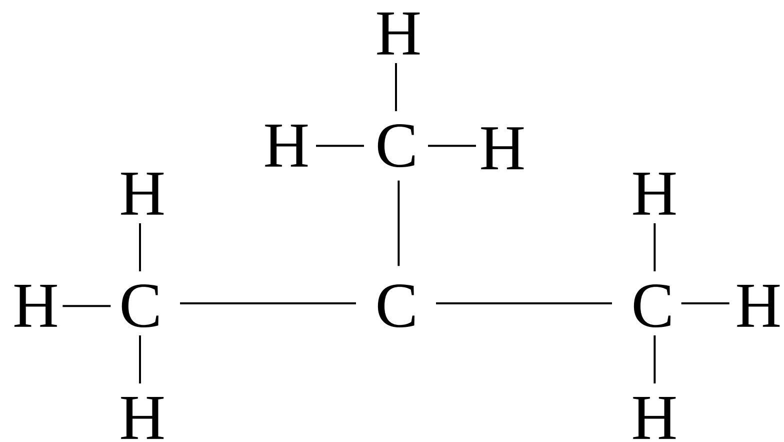
DDDC = *Dewey Decimal Classification System*

Chemical Isomers

(after Grimaldi, 1994, p. 610)



(a)

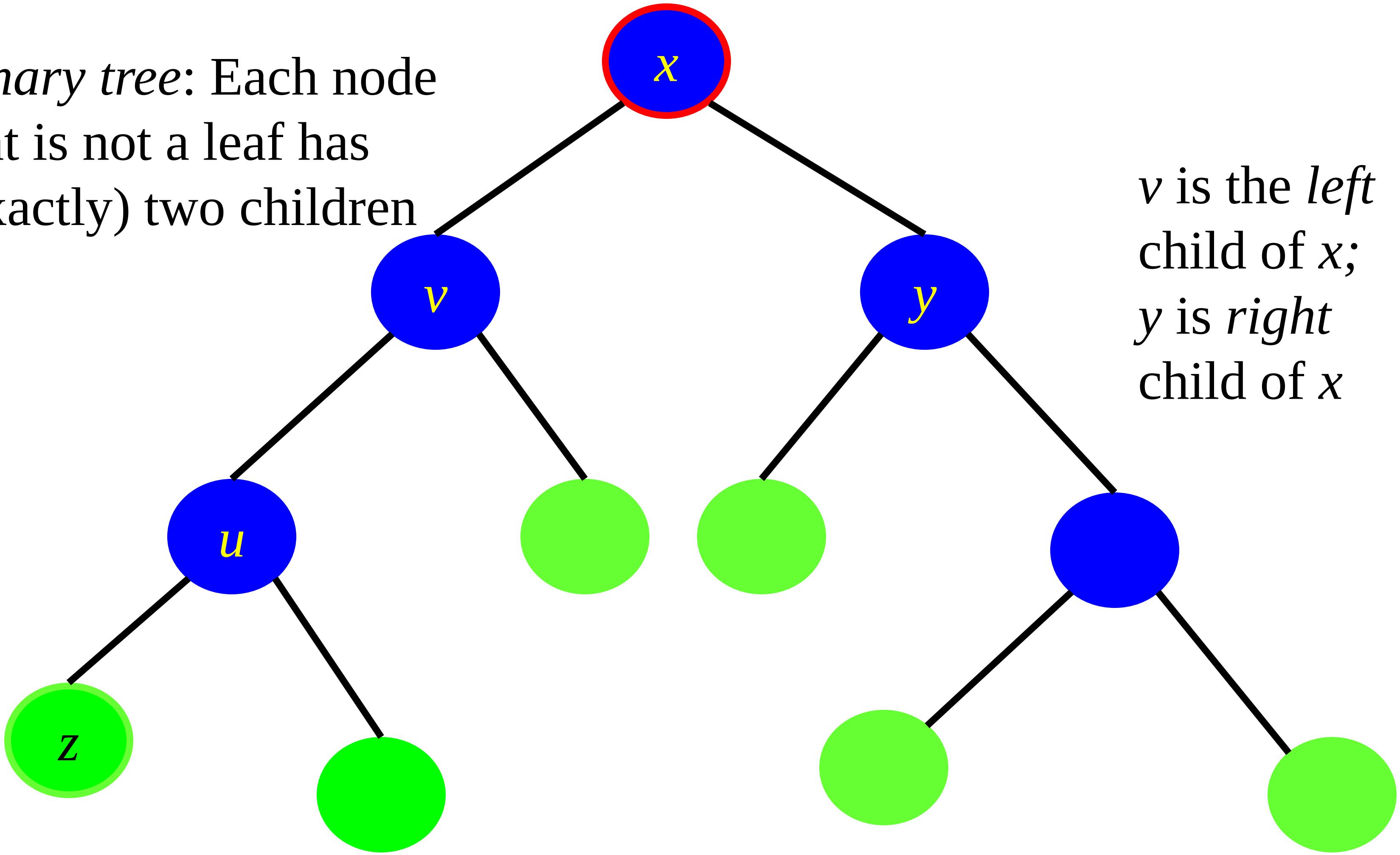


(b)

Two trees, each with 14 vertices (labeled with C's and H's) and 13 edges.
Each vertex has degree 4 (C, carbon atom) or degree 1 (H, hydrogen atom).

Binary Trees

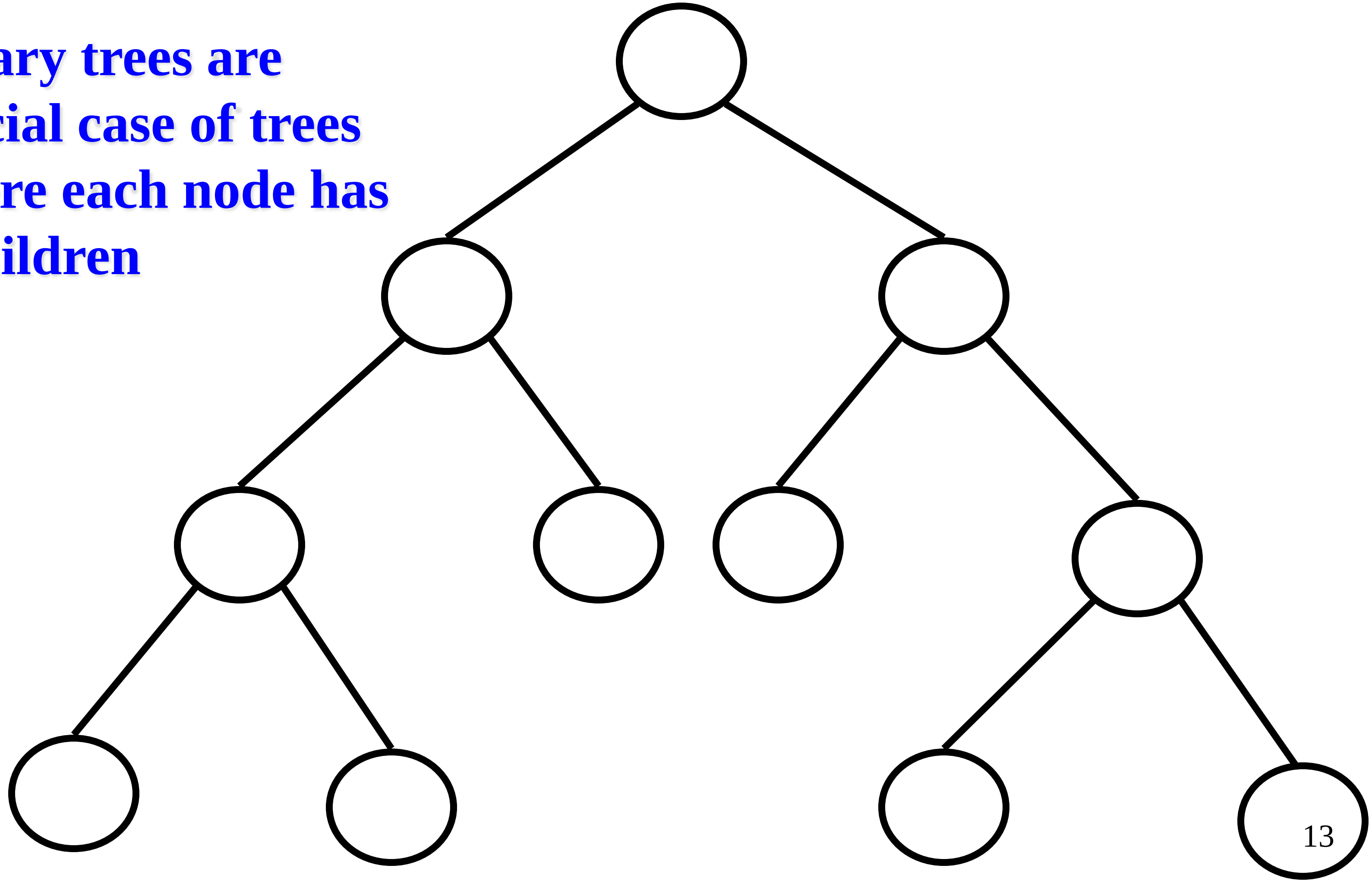
Binary tree: Each node that is not a leaf has (exactly) two children



v is the *left* child of *x*; *y* is *right* child of *x*

Binary Trees

Binary trees are special case of trees where each node has 2 children

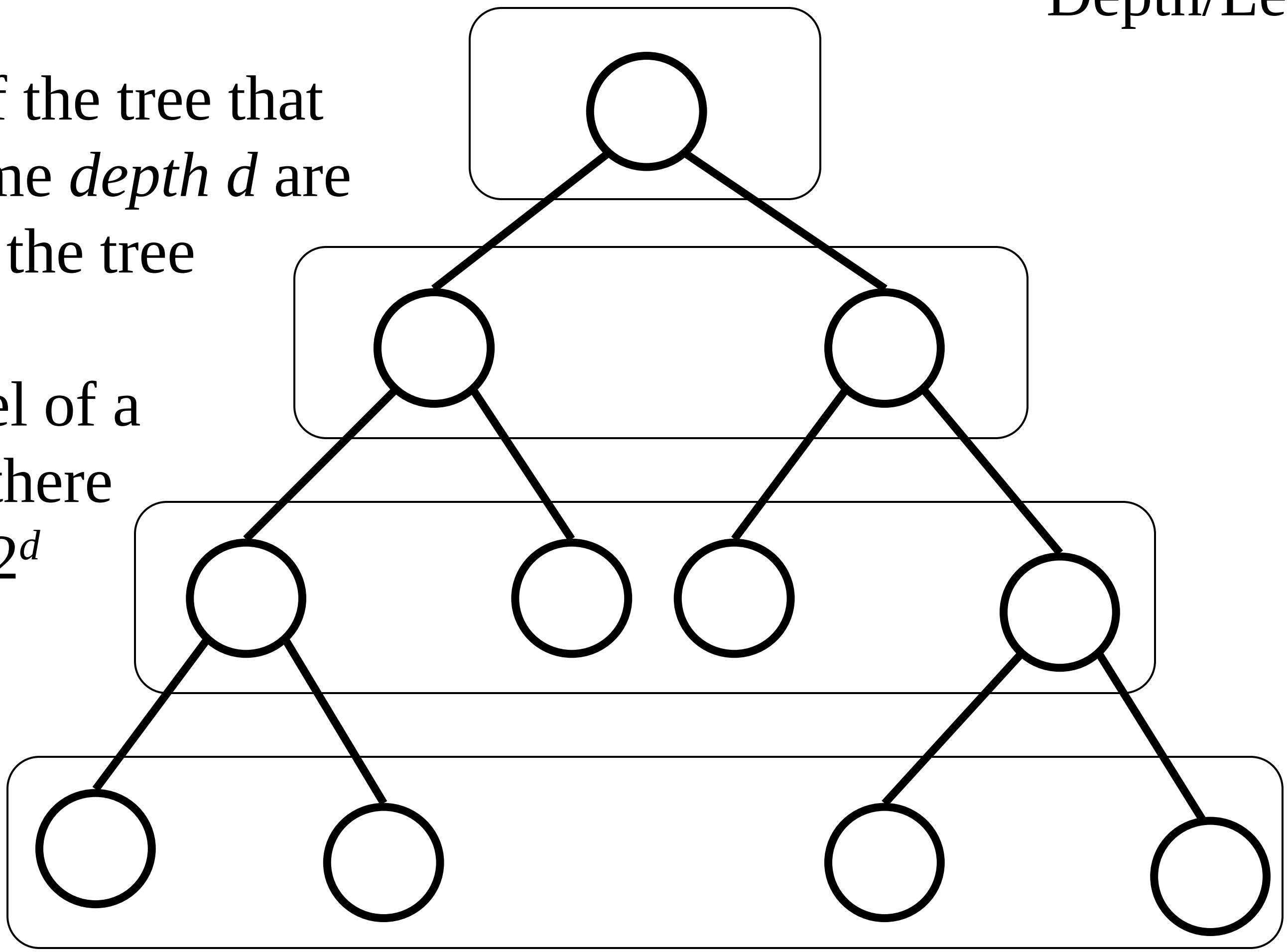


Depth and Levels in Trees

All nodes of the tree that have the same *depth d* are at *level d* of the tree

At each level of a *binary* tree there are at most 2^d nodes

Depth/Level



0

1

2

3

Properties of Binary Trees

- Let T be a binary tree with n nodes and let h denote the height of T
  - The number of *leaves* in T is at least $h + 1$
  - The number of internal nodes in T is at least h and at most $2^h - 1$
  - $n \leq 2^{h+1} - 1$
  - $\log(n+1) - 1 \leq h \leq (n-1)/2$
  - # of leaves = 1 + # of internal nodes

ADT Tree

- Stores *elements* at positions, which are defined relative to their neighbour positions (i.e., parents, children, siblings)
- The *positions* in a tree are its nodes
- Supported methods by a node/position object:
  - **root():** returns the root of the tree
  - **element(t):** returns the object at this position
  - **parent(*v*):** returns the parent of node *v;* an error occurs if *v* is root
  - **children(*v*):** returns an iterator to enumerate the children of node *v*

ADT Tree …

- **isInternal():** Test whether node *v* is internal
- **isExternal():** Test whether node *v* is external (i.e., leaf)
- **isRoot(*v*):** Test whether node *v* is root.
- **size():** returns the number of nodes in the tree
- **elements():** returns an iterator of all the elements stored at nodes of the tree (i.e., pre-, in-, post-, level-order)
- **positions():** returns an iterator of all the positions of the tree (i.e., pre-, in-, post-, level-order)
- **swapElements(*v*,*w*):** Swap the elements stored at nodes *v* and *w*
- **replaceElements(*v*,*e*):** Replace the element stored at node *v* with *e* and return the original element stored at node *v*

ADT Binary Tree

- Specialization of a tree ADT that supports the accessor methods
  - **leftChild(*v*):** returns the left child of *v;* an error occurs if *v* is a leaf.
  - **rightChild(*v*):** returns the right child of *v;* an error occurs if *v* is a leaf.
  - **sibling(*v*):** returns the sibling of *v;* an error occurs if *v* is the root.

Tree Algorithms: depth

Definition: The *depth* of a node v in a tree T is (recursively) defined to be

- 0, if v is the root of T
- The depth of the parent of v + 1, otherwise

Algorithm depth(T,v):

Input: Tree T, node v in T

Output: the depth of v in T (i.e., the number of ancestors of v in T, excluding v itself)

Tree Algorithms: height

Definition: The *height* of a a tree *T* rooted at node *v* is (recursively) defined to be

- 0 if *v* is a leaf.
- 1+ the maximum height of a child of *v*, otherwise.

Algorithm height(*T*,*v*):

Input: Tree *T*, node *v* in *T*

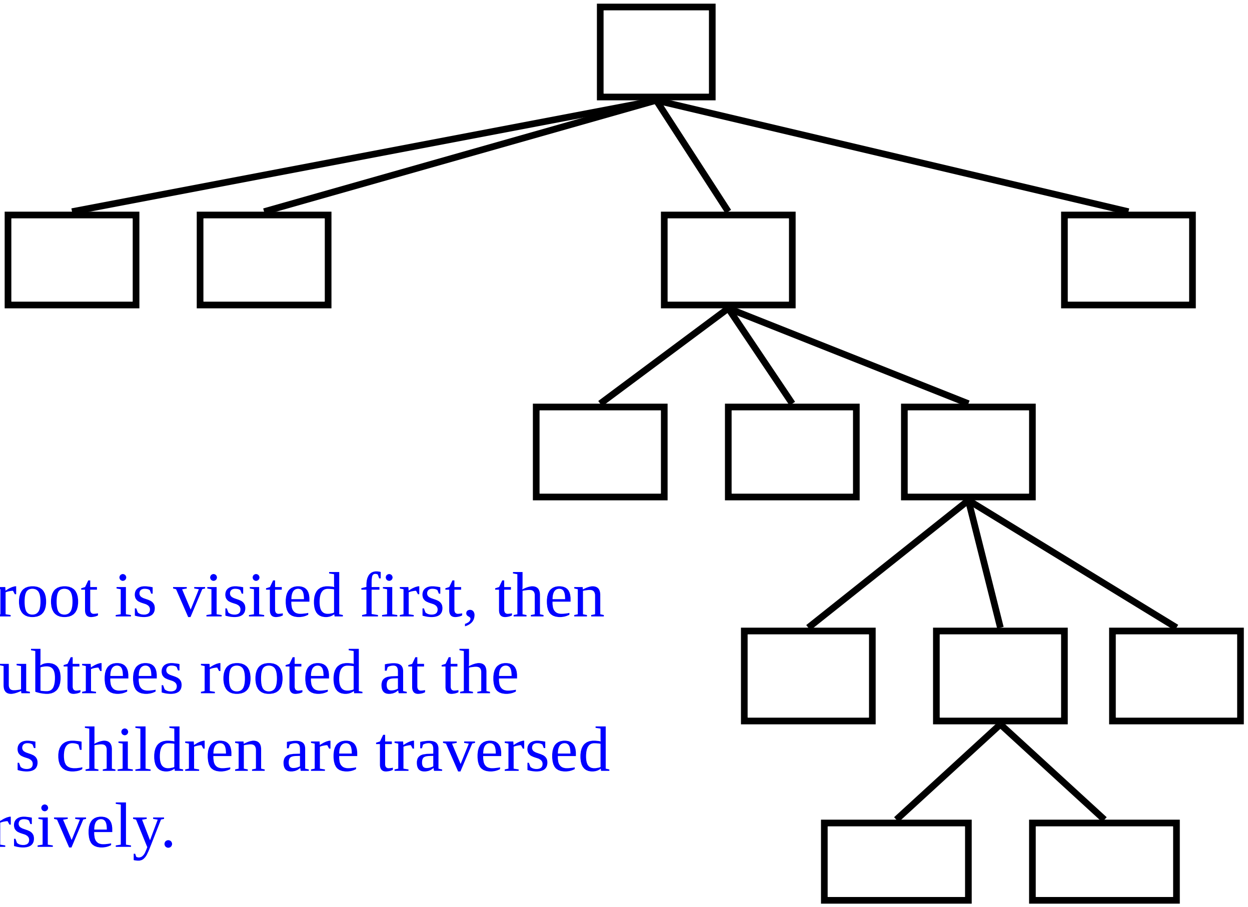
Output: the height of tree *T* rooted at node *v* (i.e., the maximum depth of a leaf of the tree *T* rooted by *v* in *T*)

Tree Traversals

- n-ary tree traversals
  - Preorder
  - Postorder
  - Level order

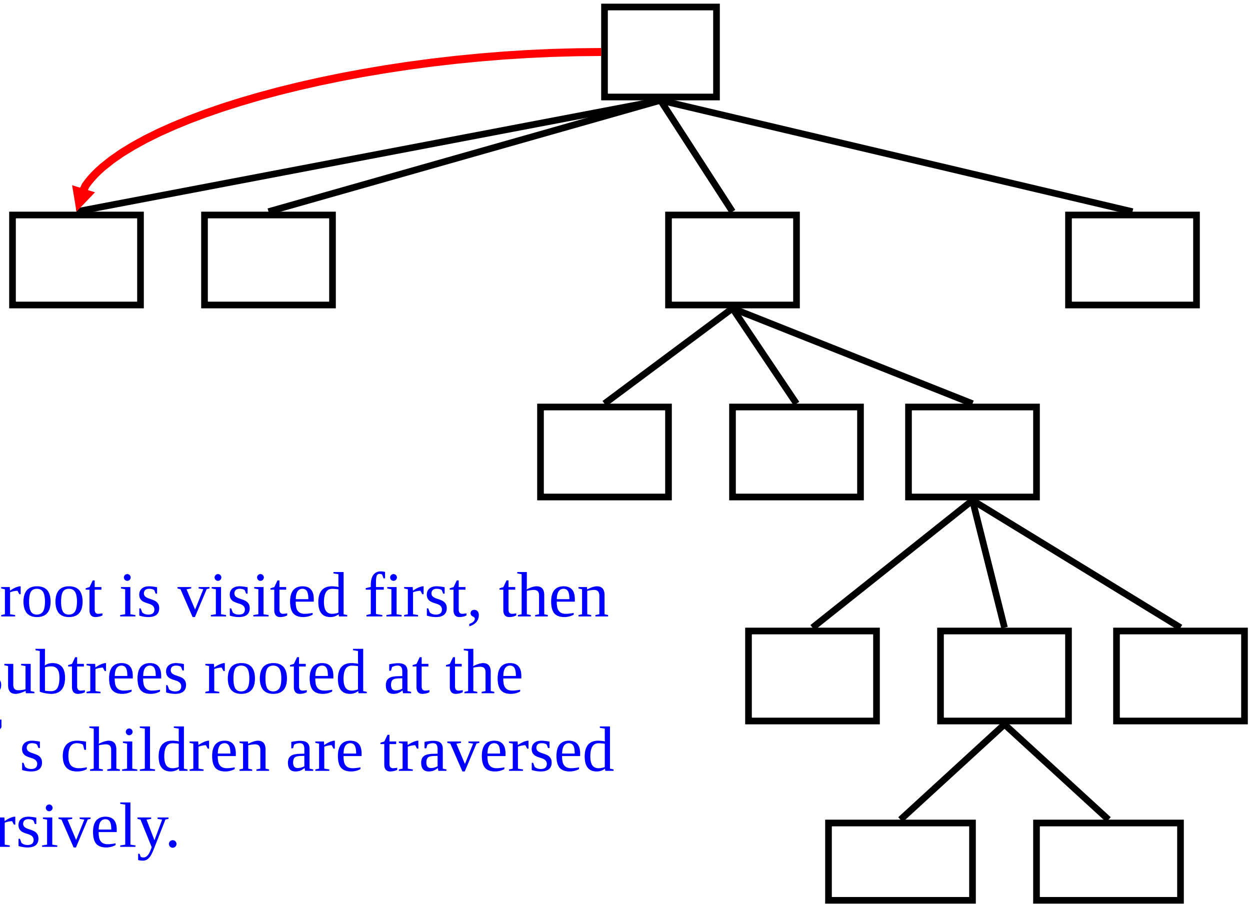
- Binary tree traversals
  - Preorder
  - Postorder
  - Inorder
  - Level order

Preorder Traversal
Depth First Search



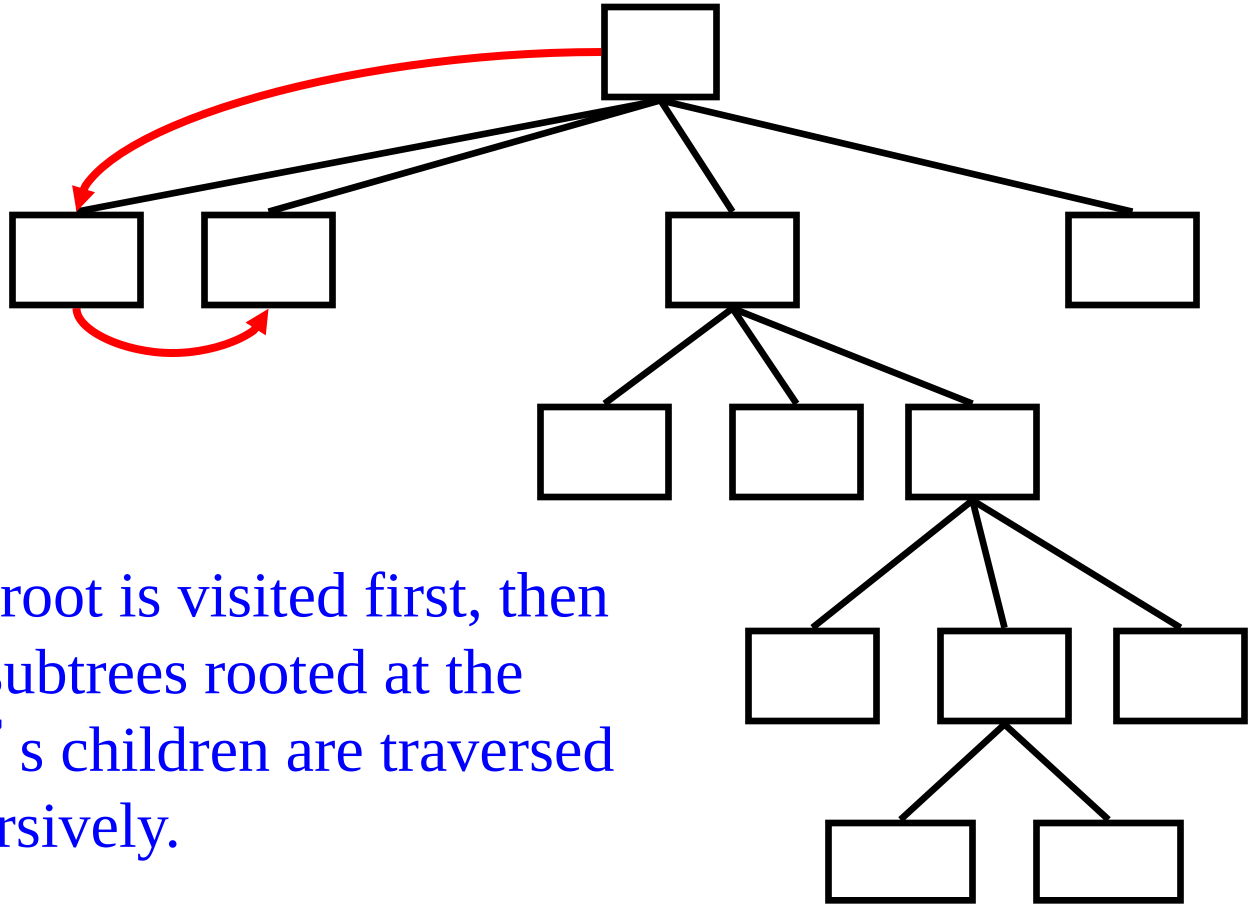
The root is visited first, then the subtrees rooted at the root's children are traversed recursively.

Preorder Traversal
Depth First Search



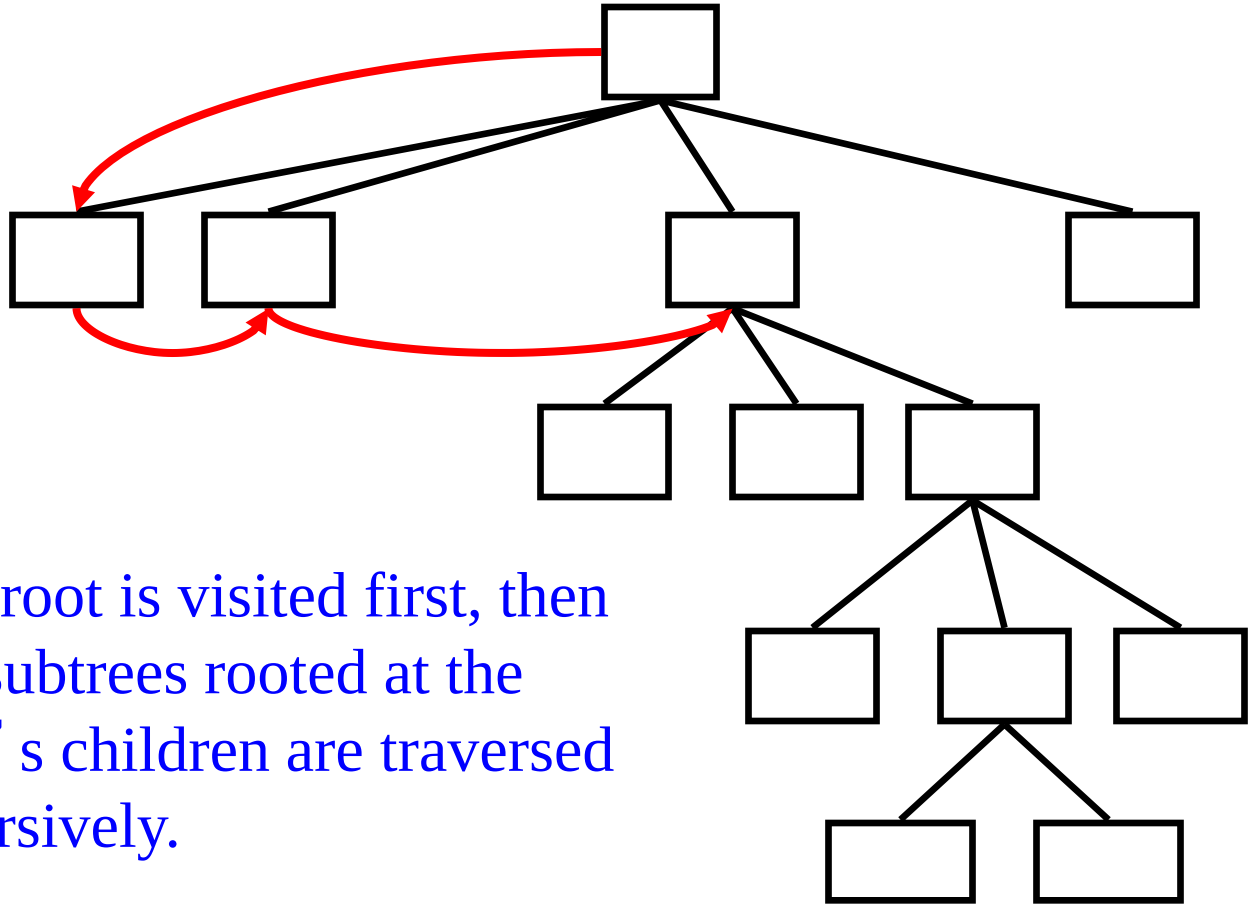
The root is visited first, then the subtrees rooted at the root’s children are traversed recursively.

Preorder Traversal
Depth First Search



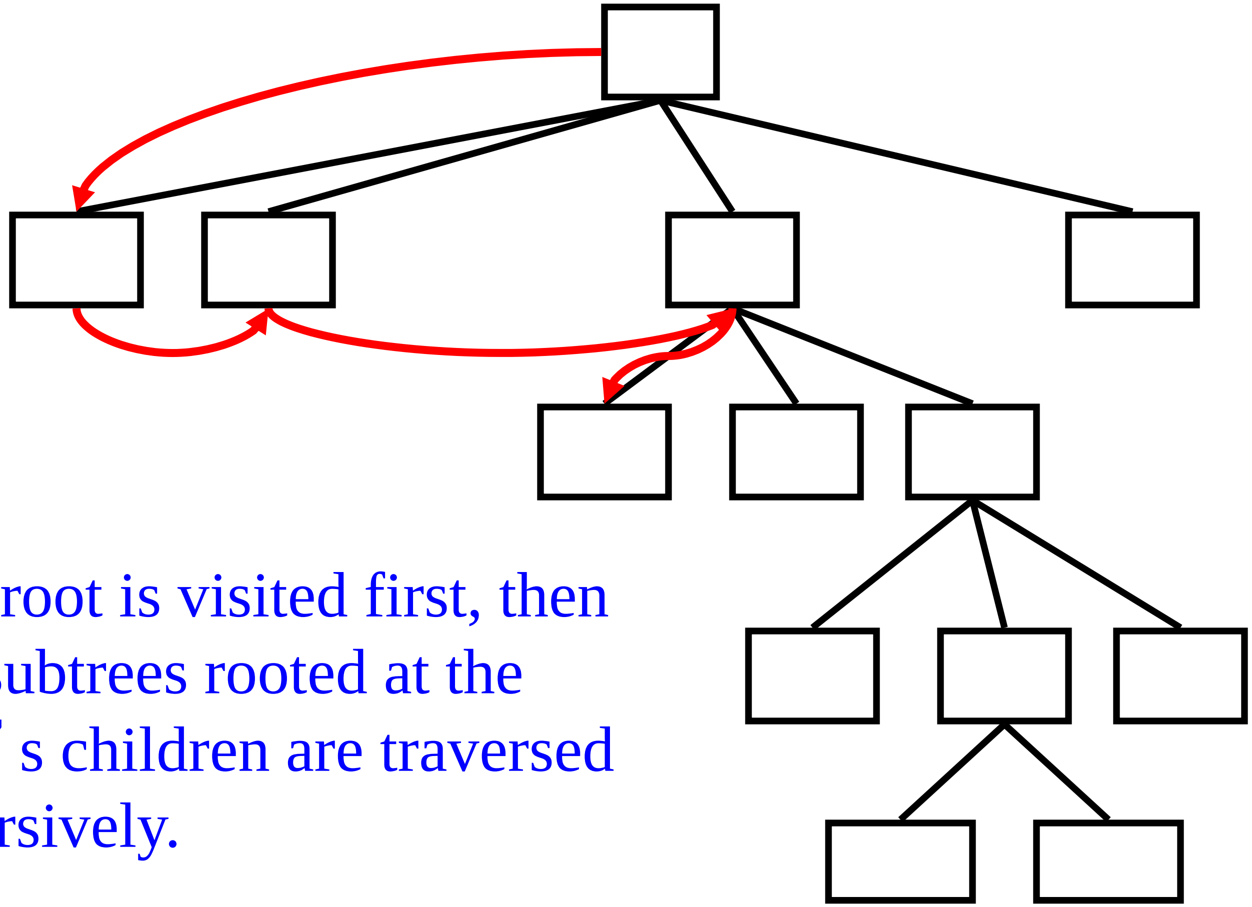
The root is visited first, then the subtrees rooted at the root's children are traversed recursively.

Preorder Traversal
Depth First Search



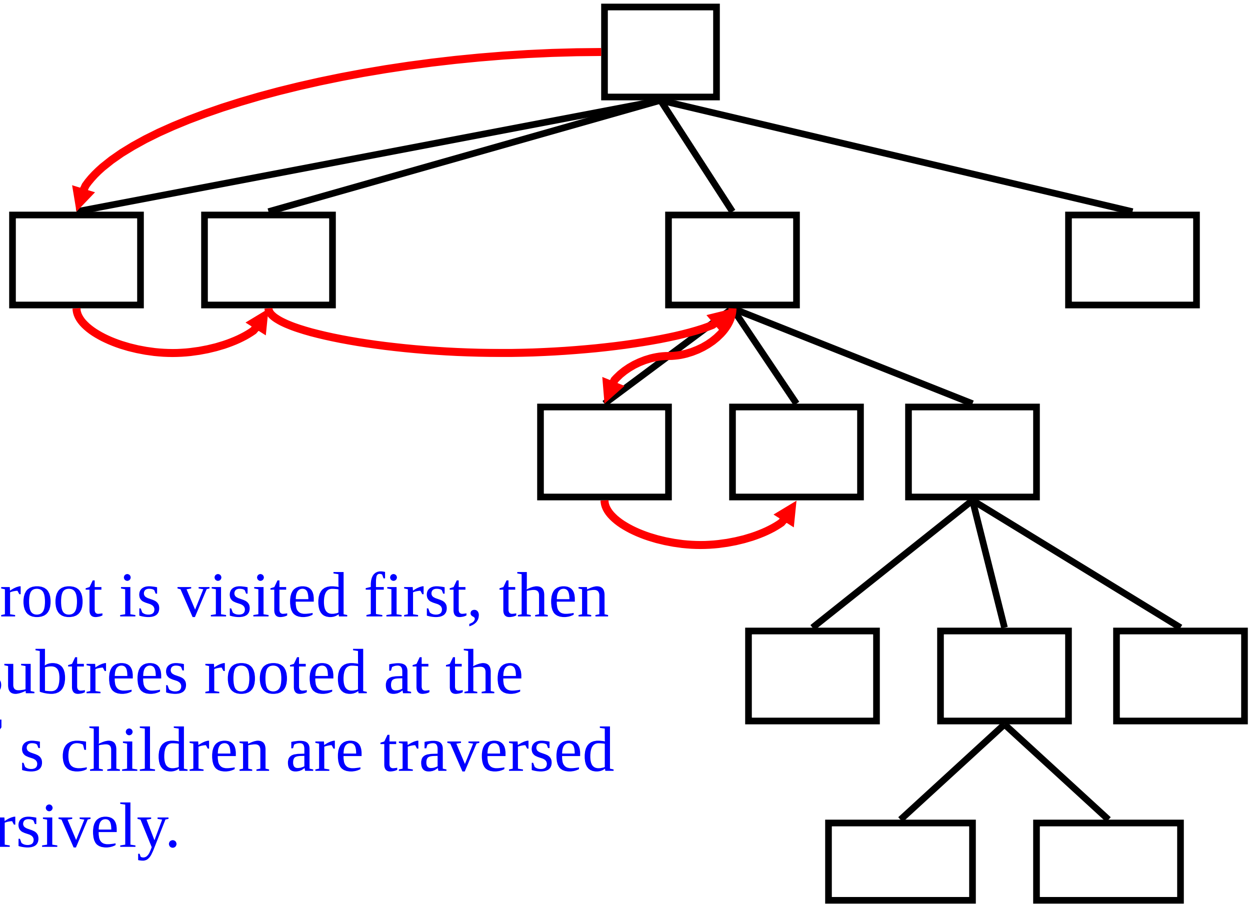
The root is visited first, then the subtrees rooted at the root’s children are traversed recursively.

Preorder Traversal
Depth First Search



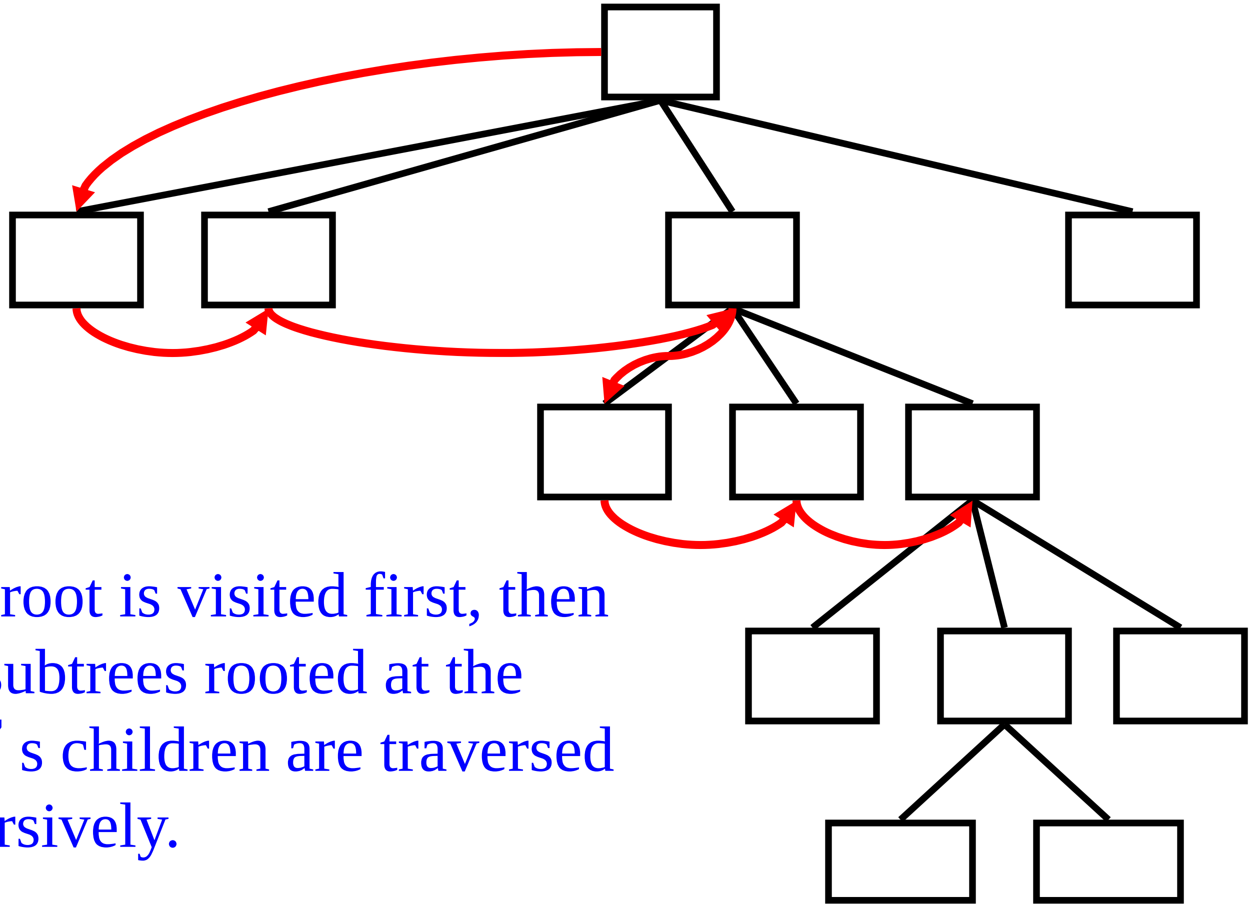
The root is visited first, then the subtrees rooted at the root's children are traversed recursively.

Preorder Traversal
Depth First Search



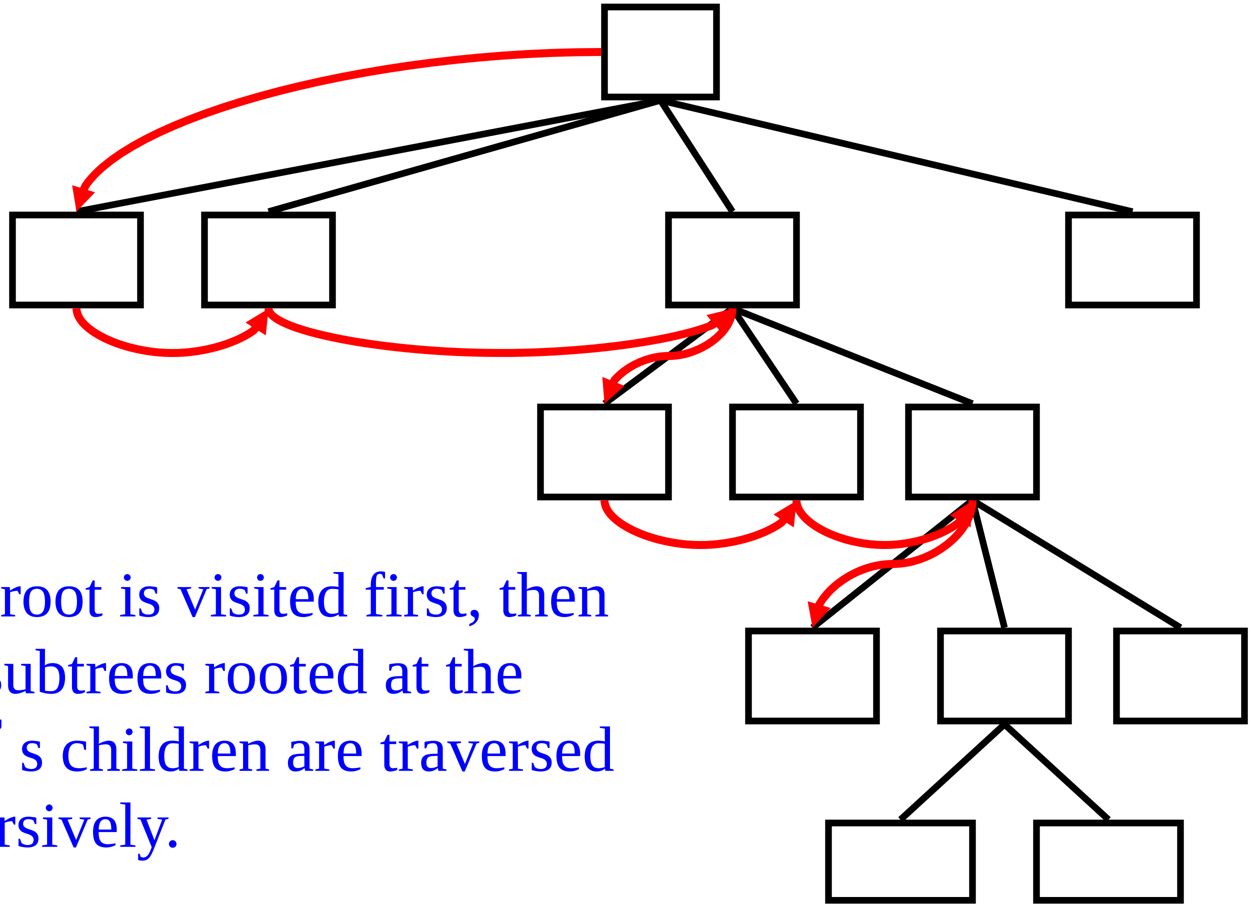
The root is visited first, then the subtrees rooted at the root’s children are traversed recursively.

Preorder Traversal
Depth First Search



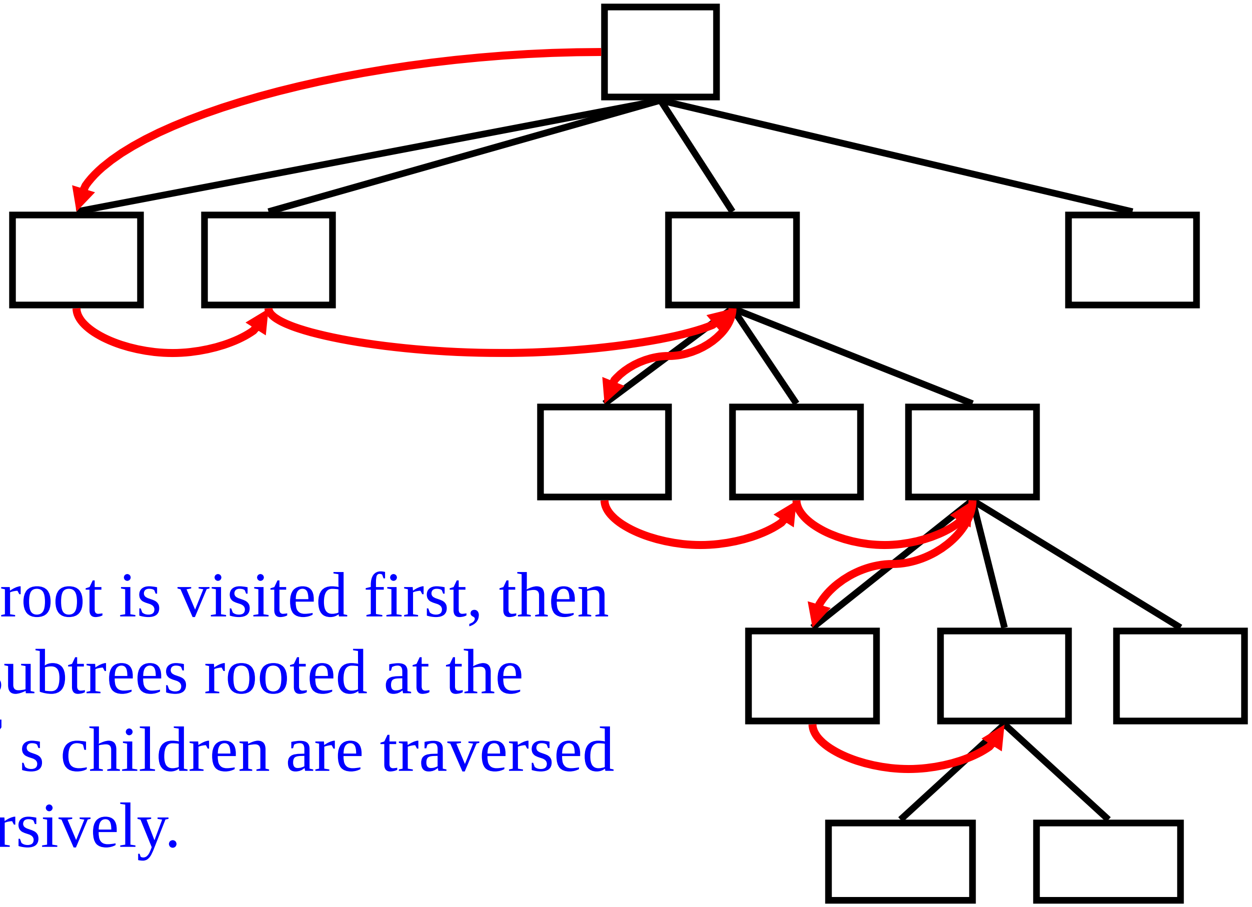
The root is visited first, then the subtrees rooted at the root's children are traversed recursively.

Preorder Traversal
Depth First Search



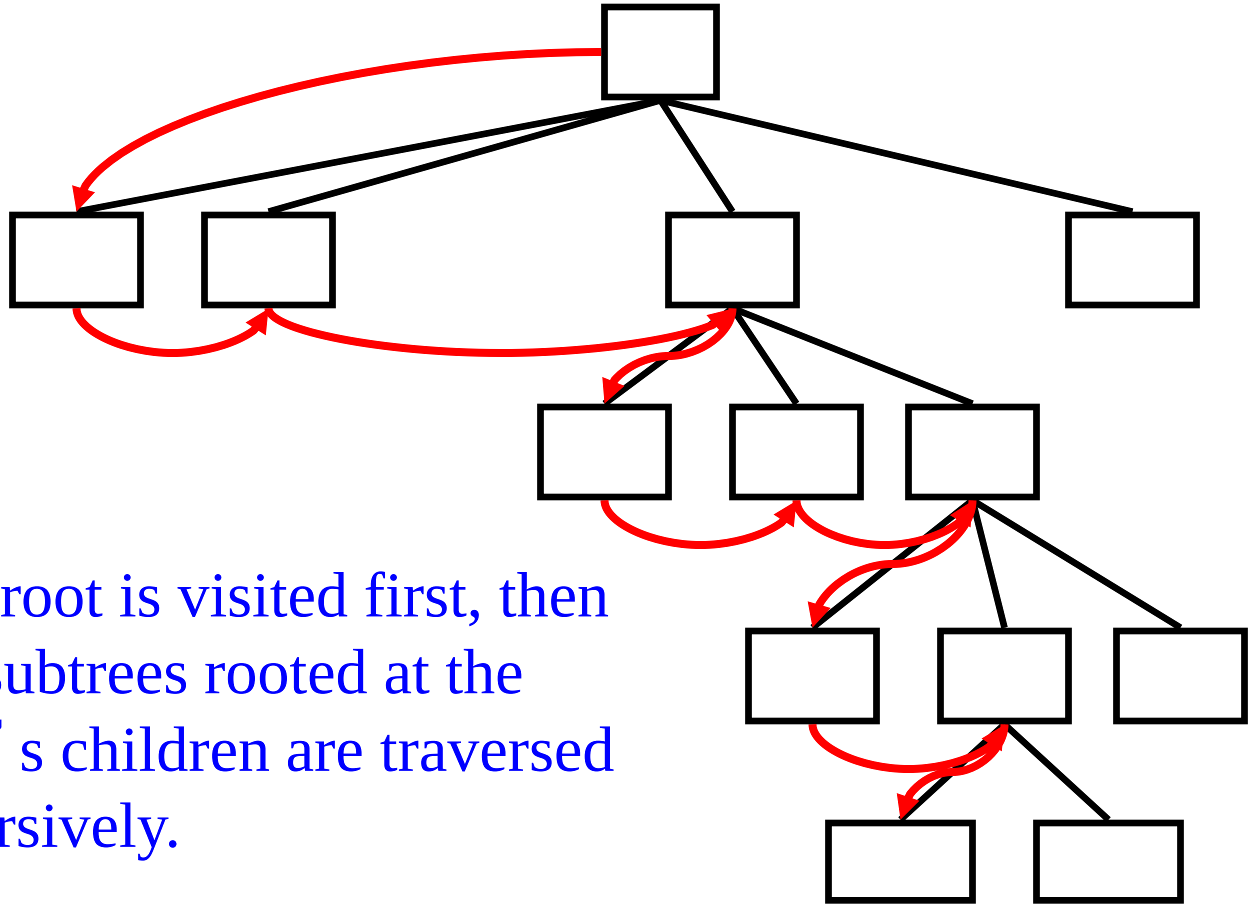
The root is visited first, then the subtrees rooted at the root's children are traversed recursively.

Preorder Traversal
Depth First Search



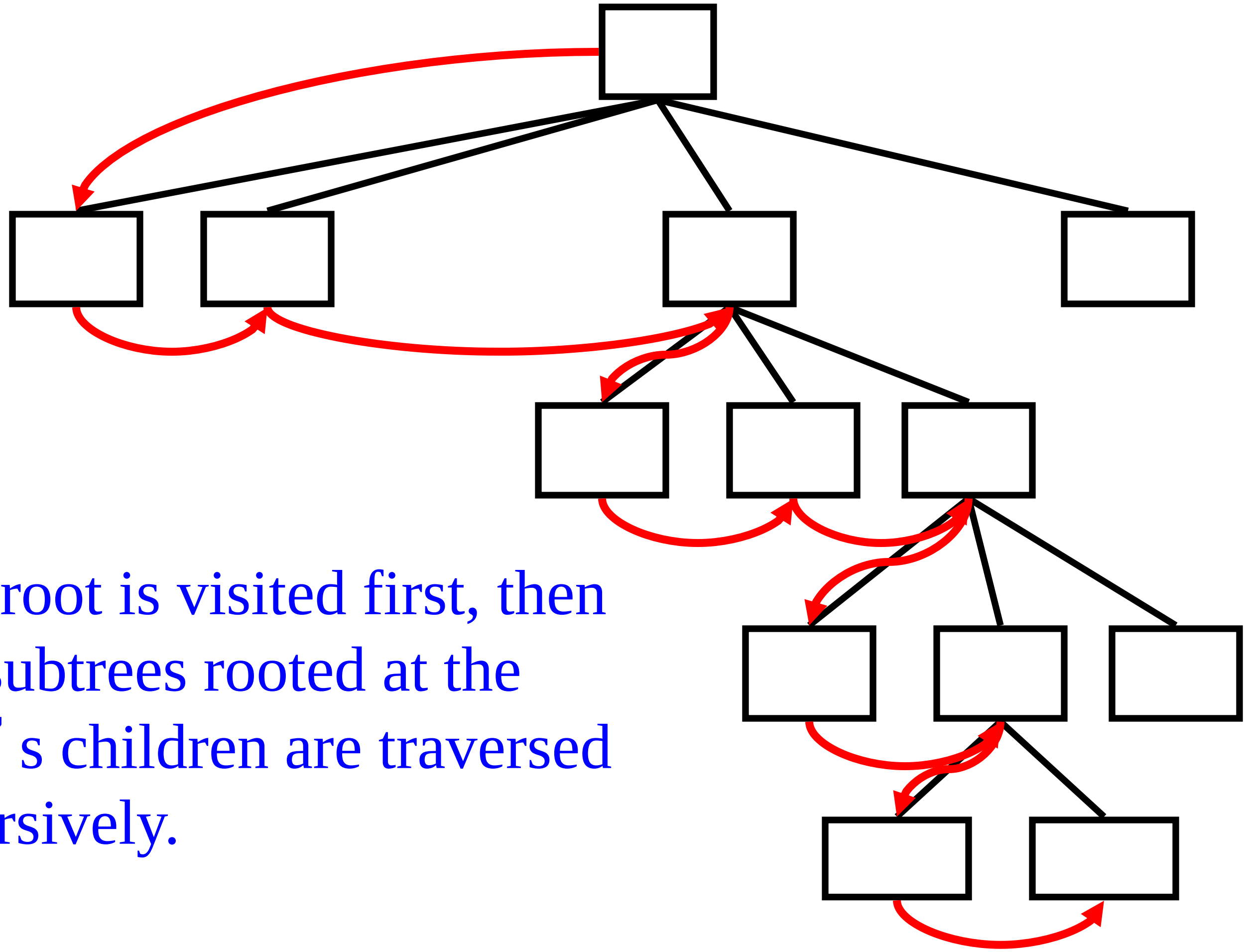
The root is visited first, then the subtrees rooted at the root's children are traversed recursively.

Preorder Traversal
Depth First Search



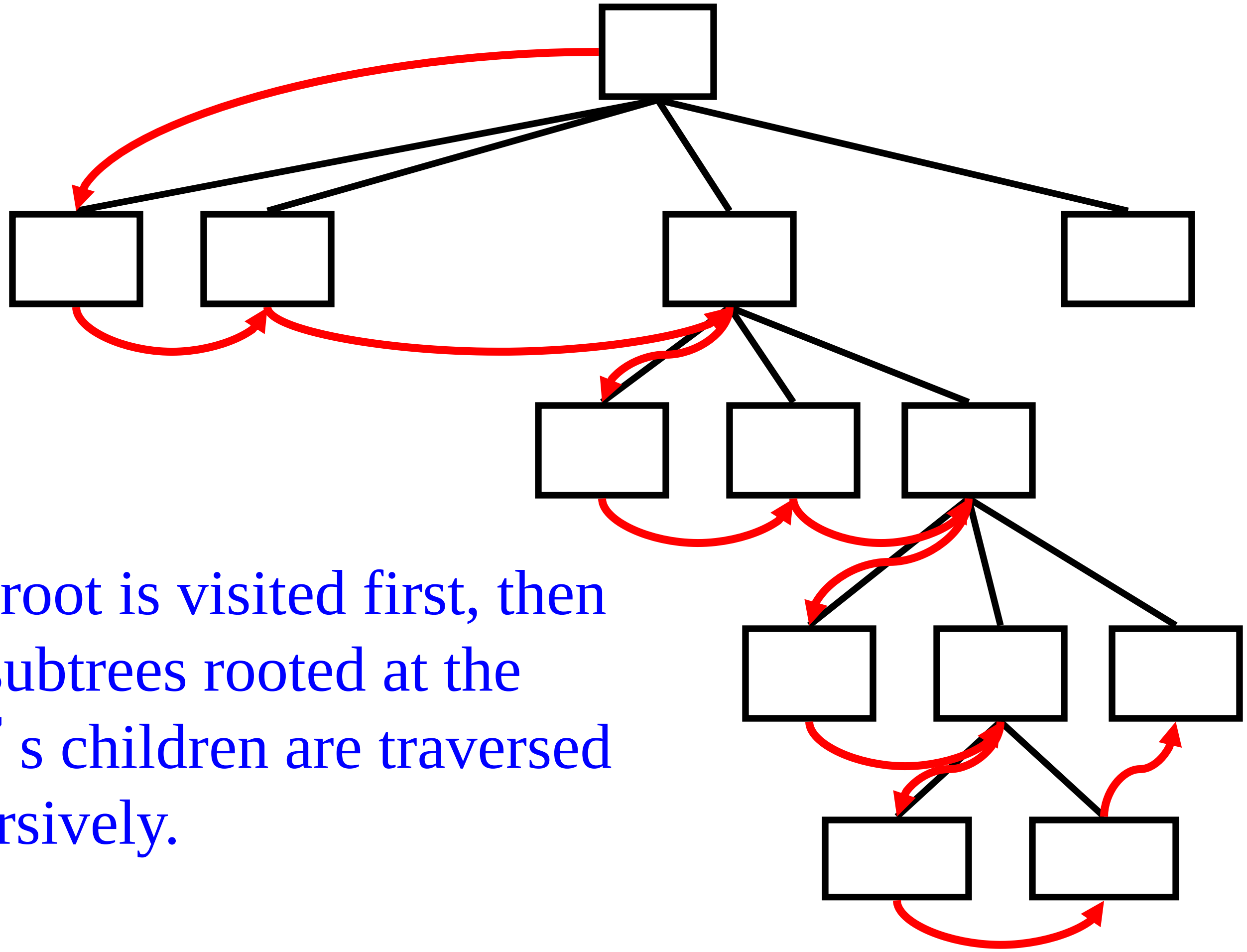
The root is visited first, then the subtrees rooted at the root's children are traversed recursively.

Preorder Traversal
Depth First Search



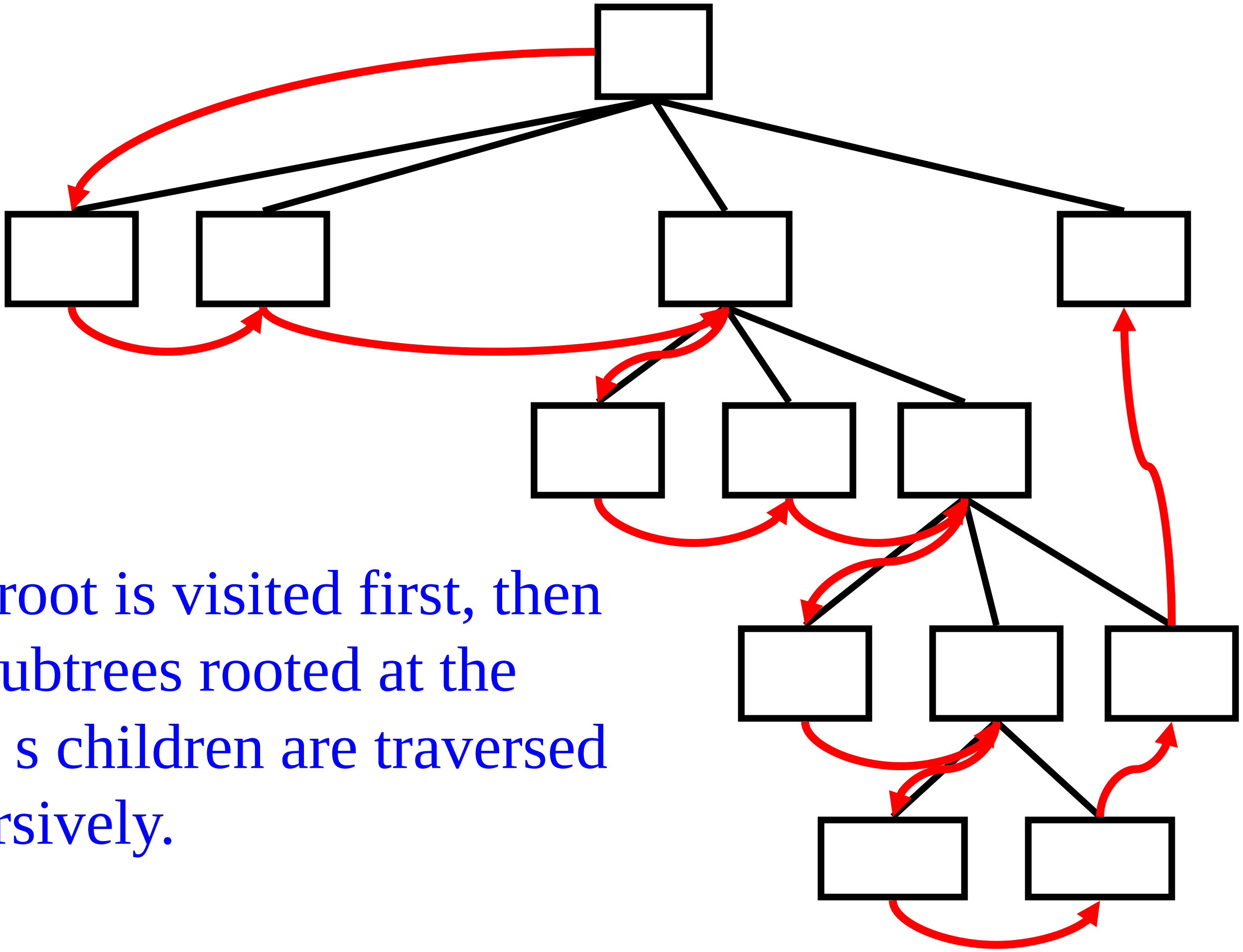
The root is visited first, then the subtrees rooted at the root's children are traversed recursively.

Preorder Traversal
Depth First Search



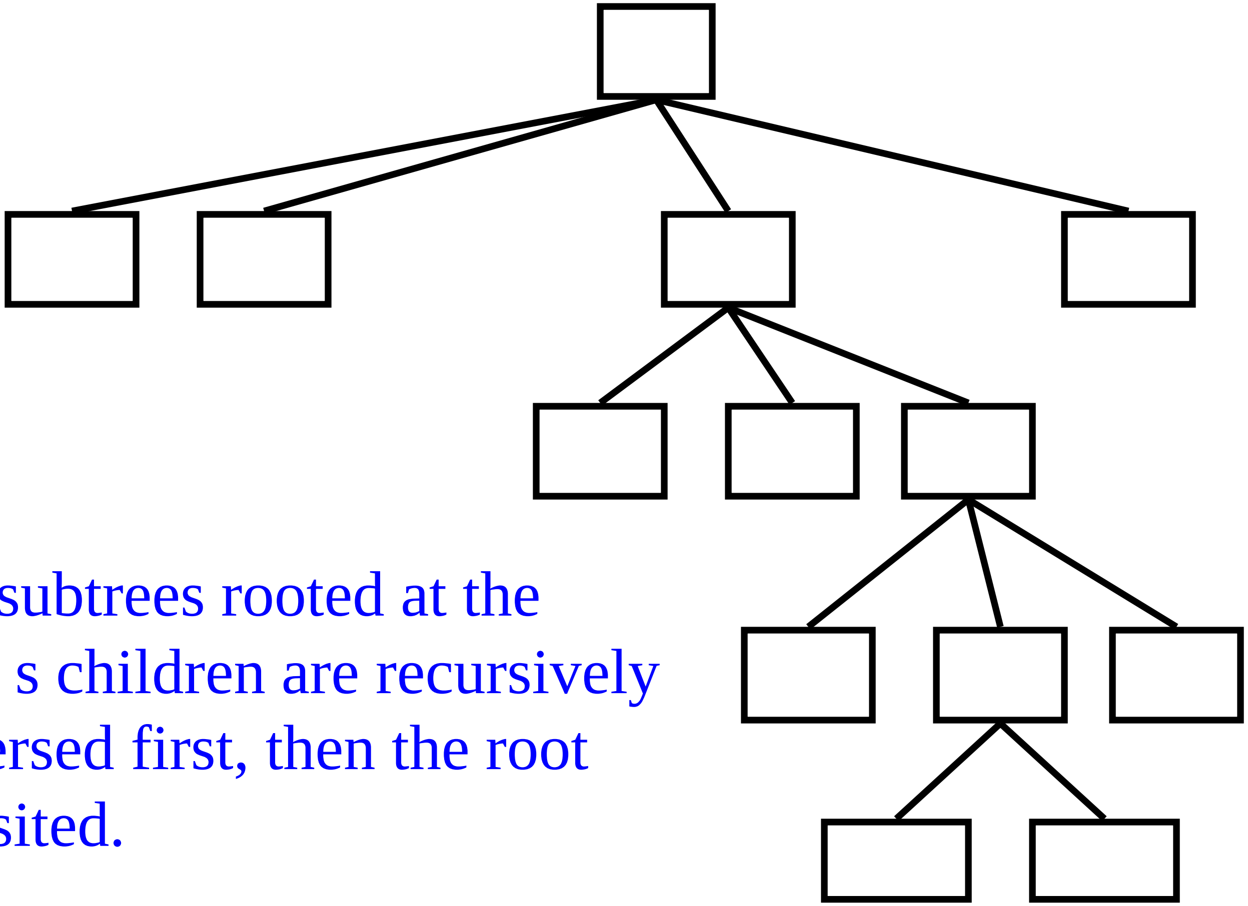
The root is visited first, then the subtrees rooted at the root's children are traversed recursively.

Preorder Traversal
Depth First Search



The root is visited first, then the subtrees rooted at the root's children are traversed recursively.

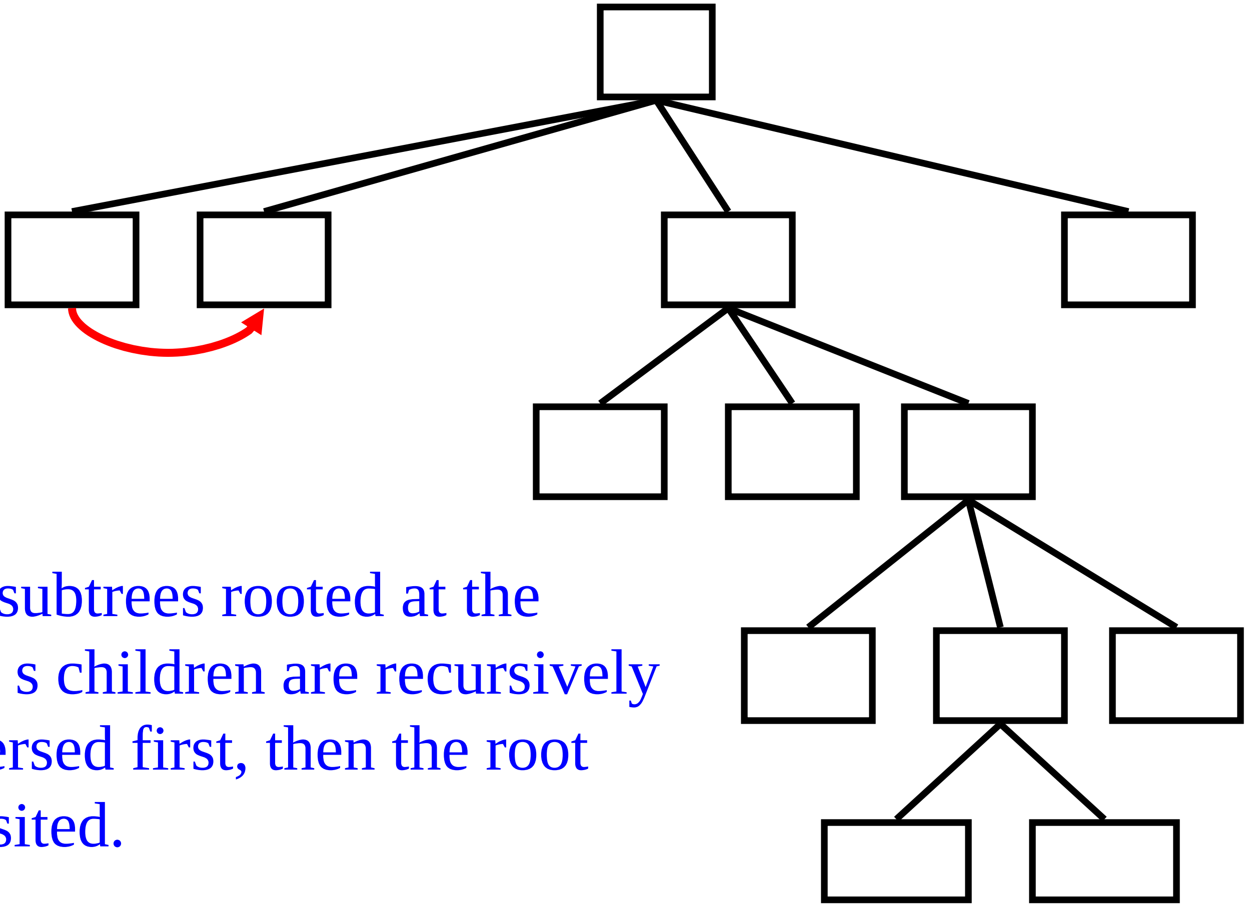
Postorder Traversal



The subtrees rooted at the root's children are recursively traversed first, then the root is visited.

.

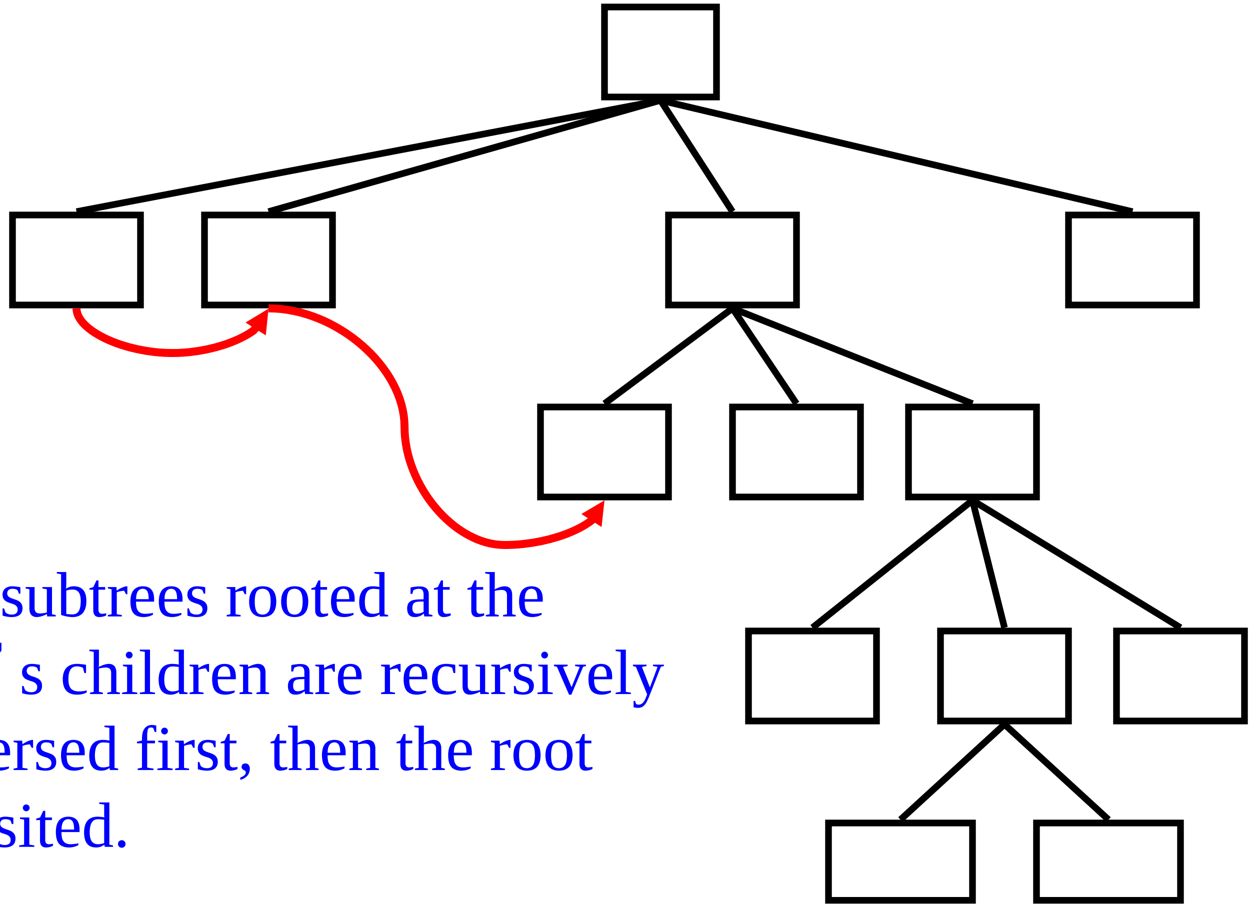
Postorder Traversal



The subtrees rooted at the root's children are recursively traversed first, then the root is visited.

.

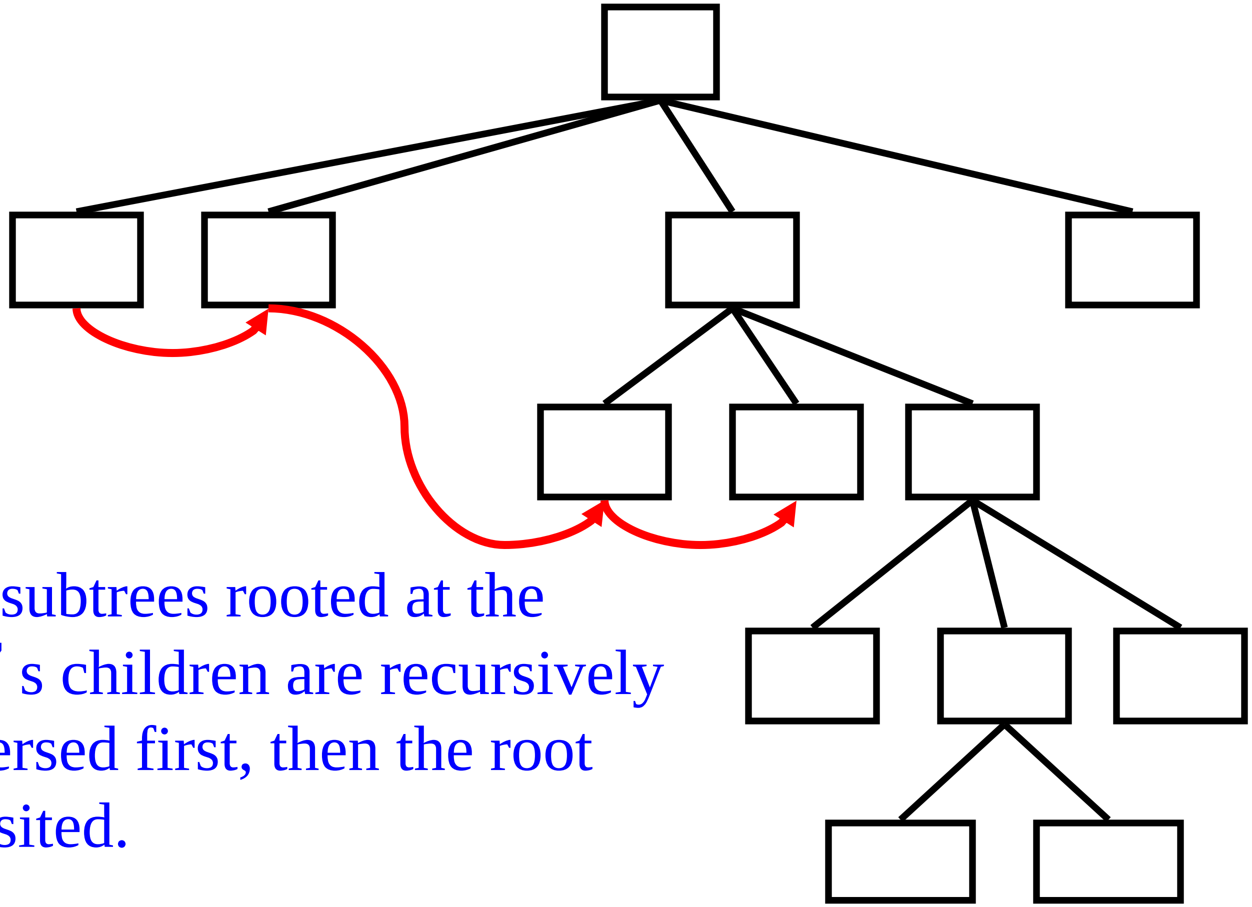
Postorder Traversal



The subtrees rooted at the root’s children are recursively traversed first, then the root is visited.

.

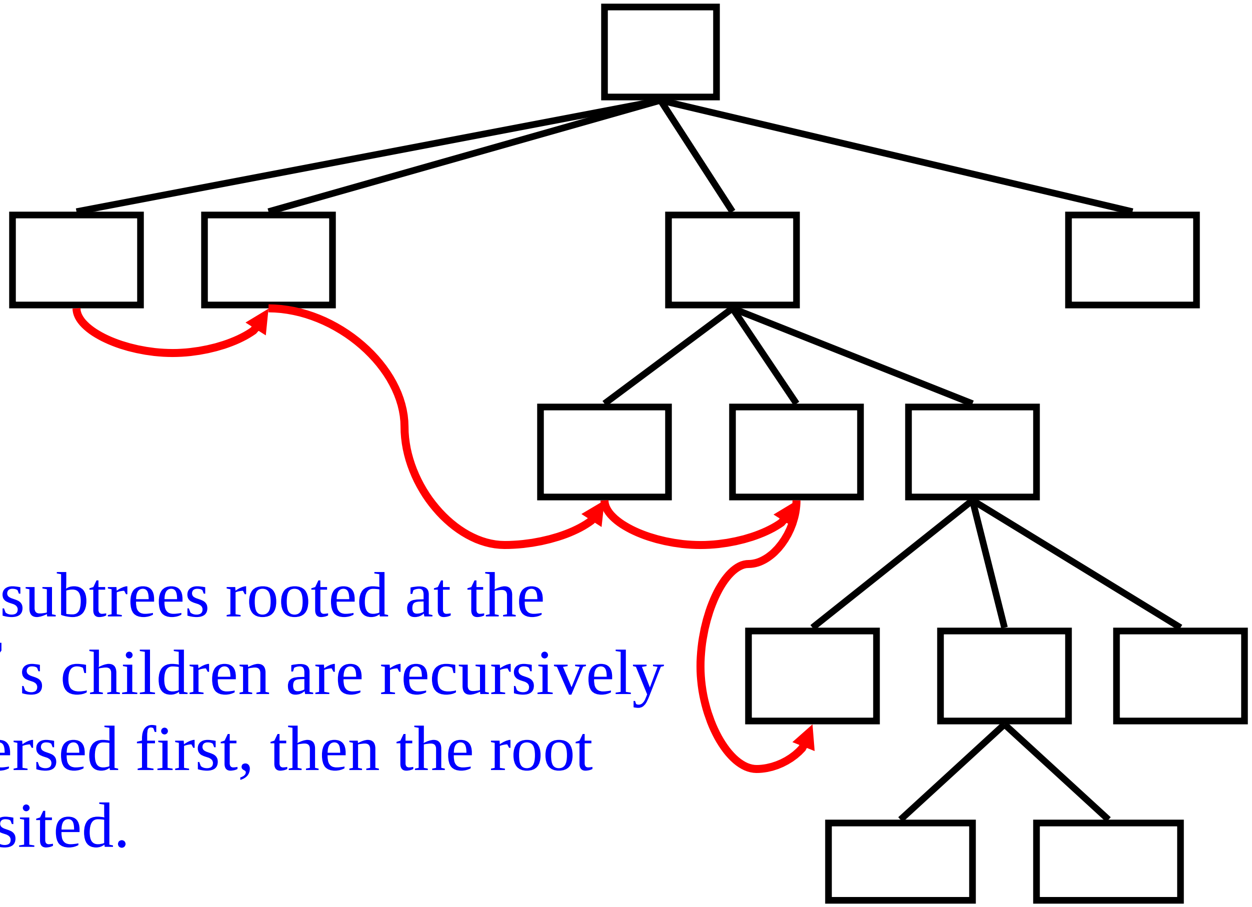
Postorder Traversal



The subtrees rooted at the root's children are recursively traversed first, then the root is visited.

.

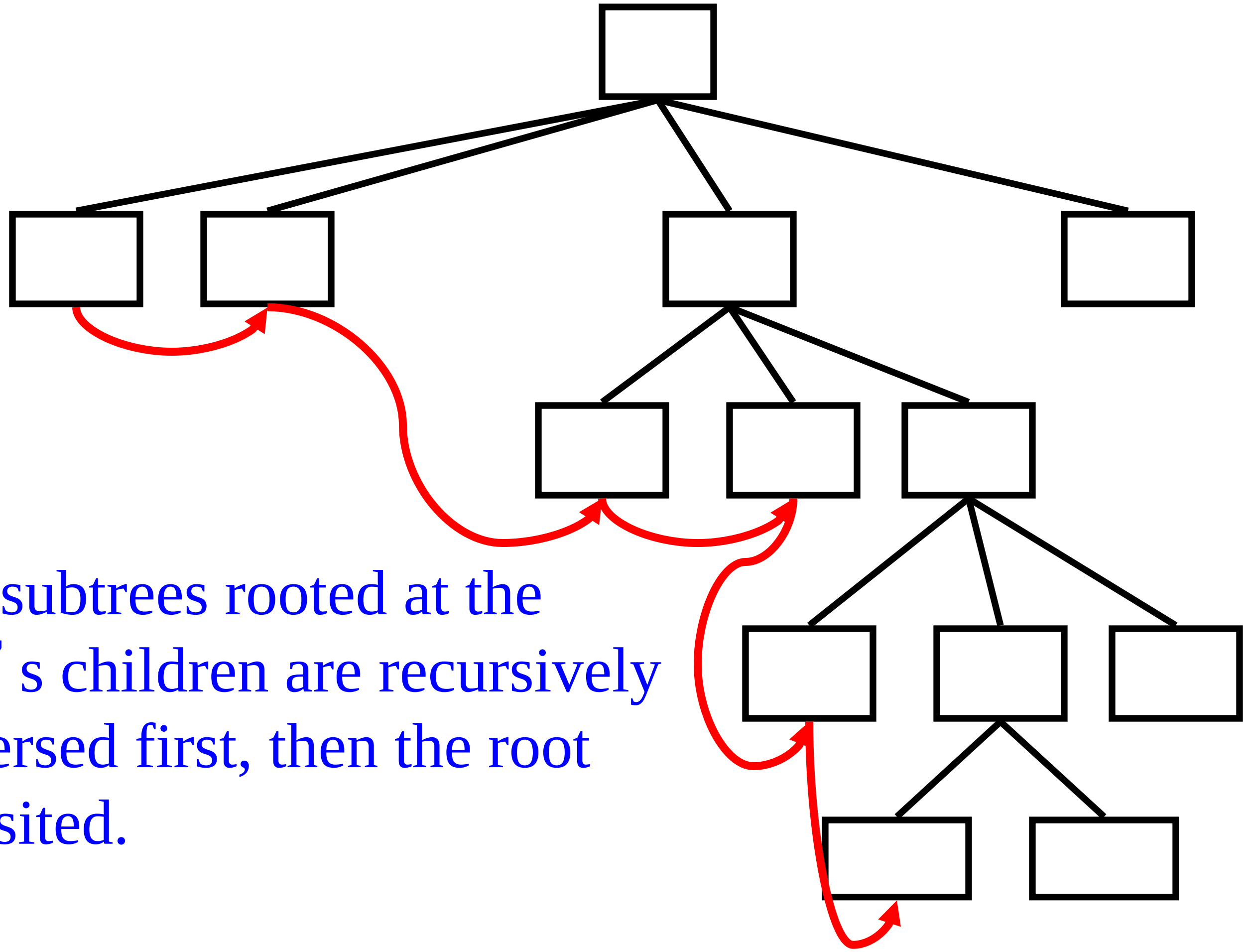
Postorder Traversal



The subtrees rooted at the root’s children are recursively traversed first, then the root is visited.

.

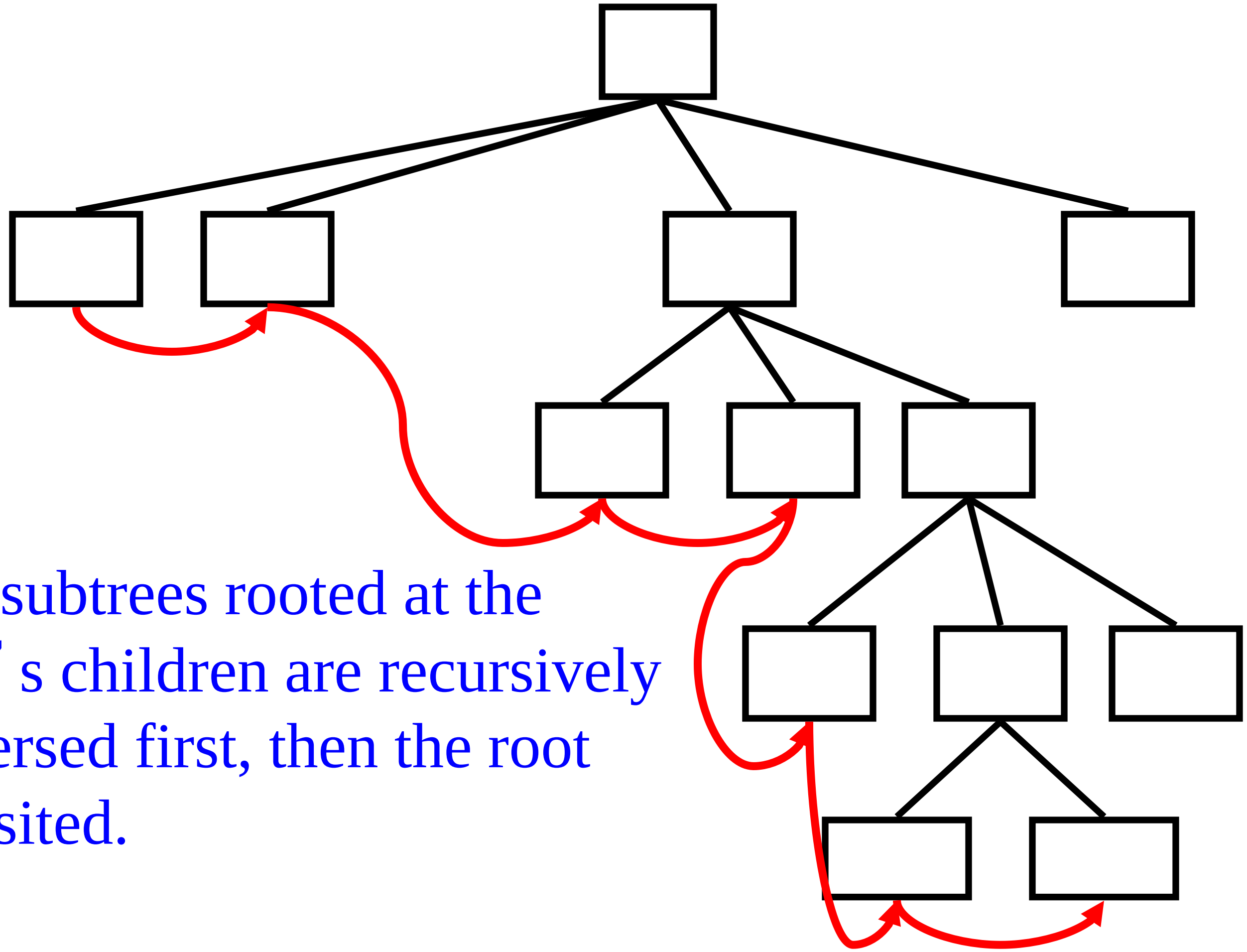
Postorder Traversal



The subtrees rooted at the root’s children are recursively traversed first, then the root is visited.

.

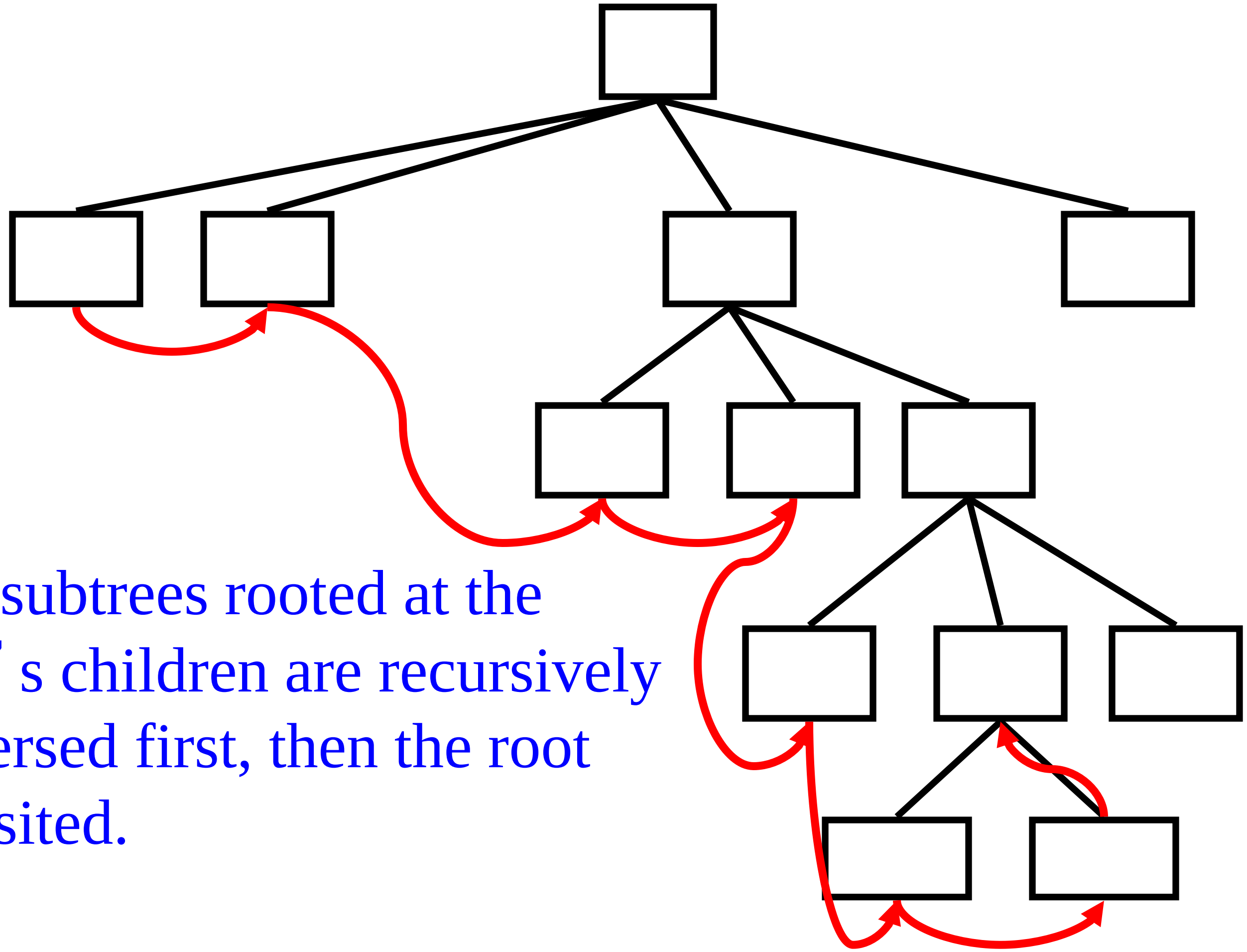
Postorder Traversal



The subtrees rooted at the root's children are recursively traversed first, then the root is visited.

.

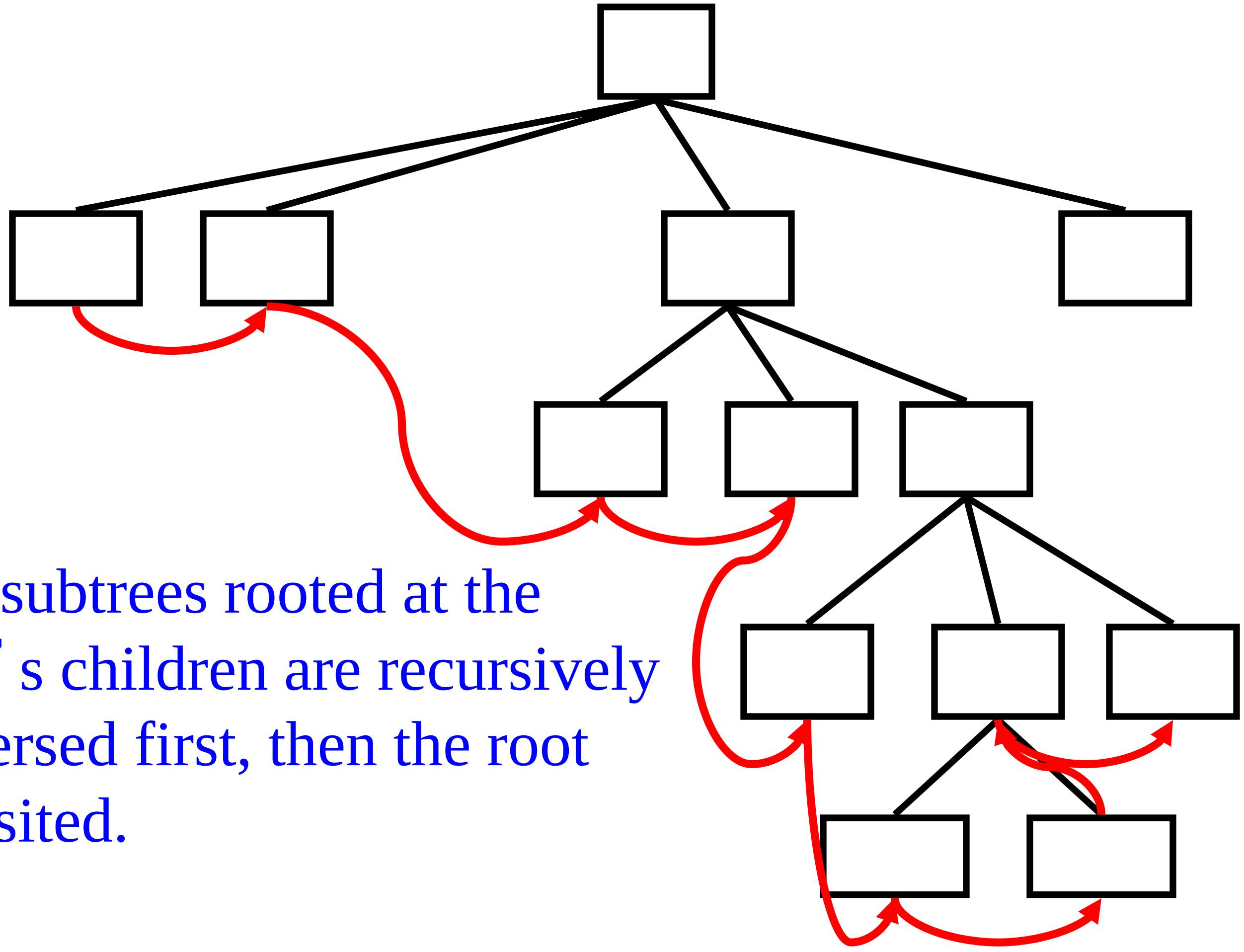
Postorder Traversal



The subtrees rooted at the root's children are recursively traversed first, then the root is visited.

.

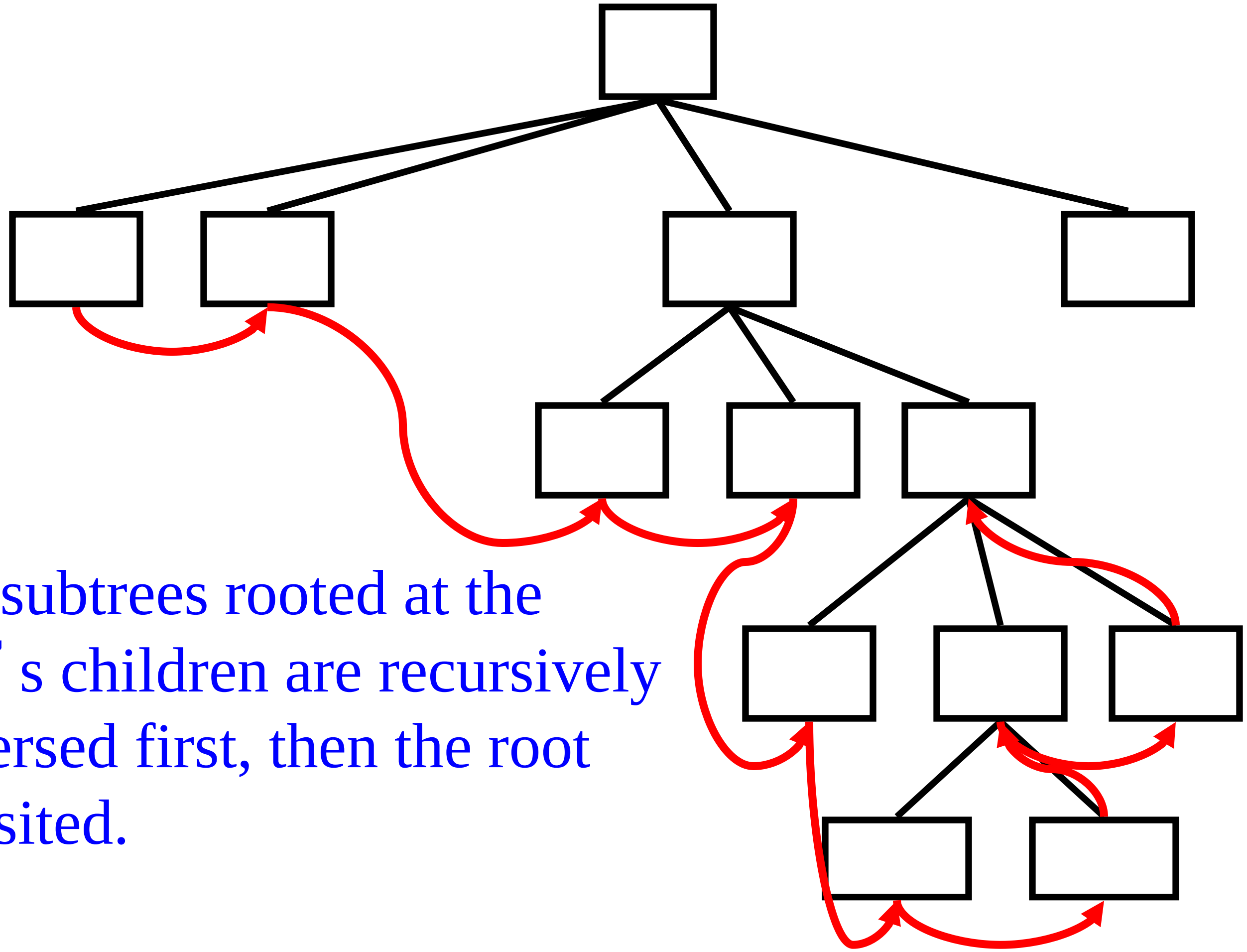
Postorder Traversal



The subtrees rooted at the root’s children are recursively traversed first, then the root is visited.

.

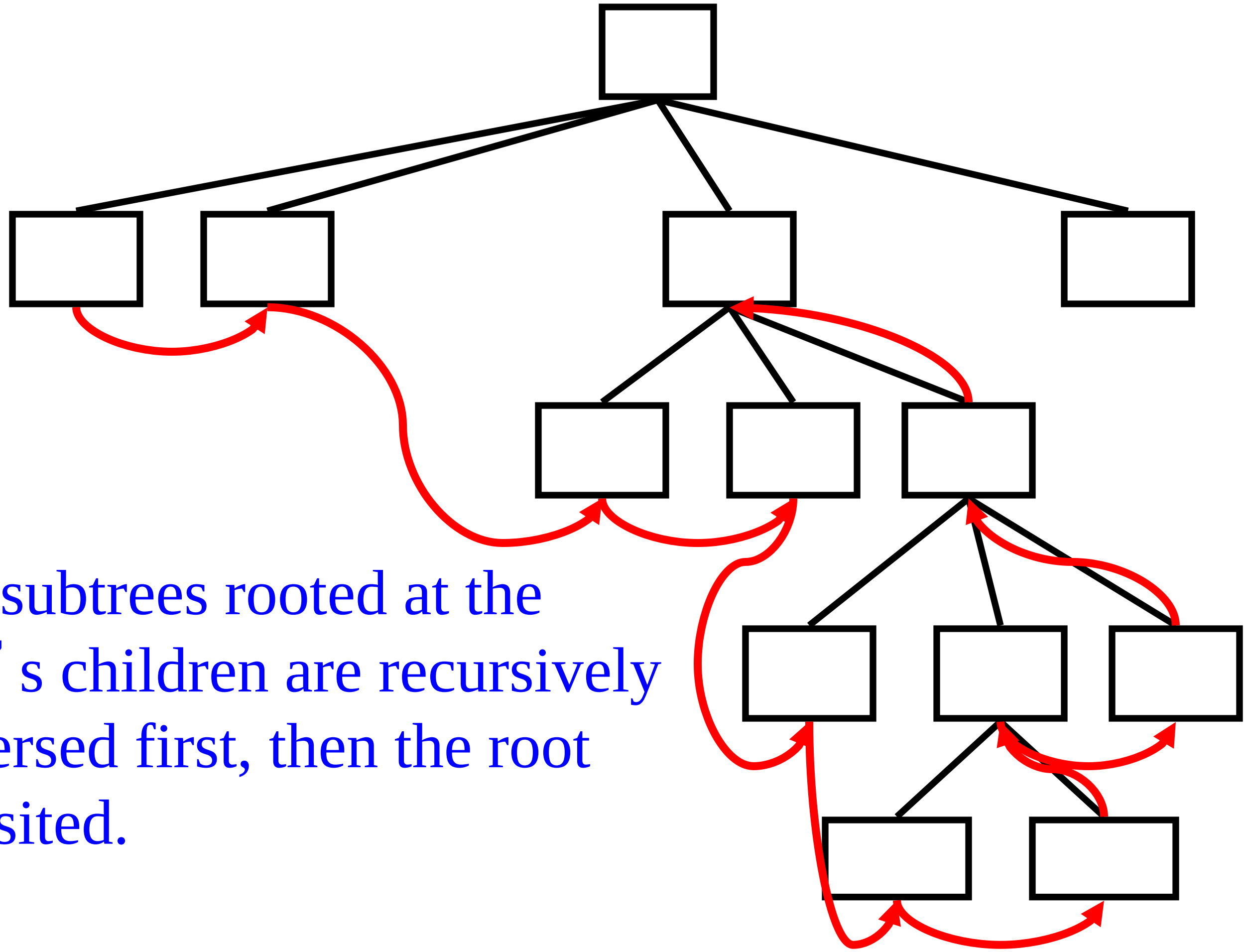
Postorder Traversal



The subtrees rooted at the root's children are recursively traversed first, then the root is visited.

.

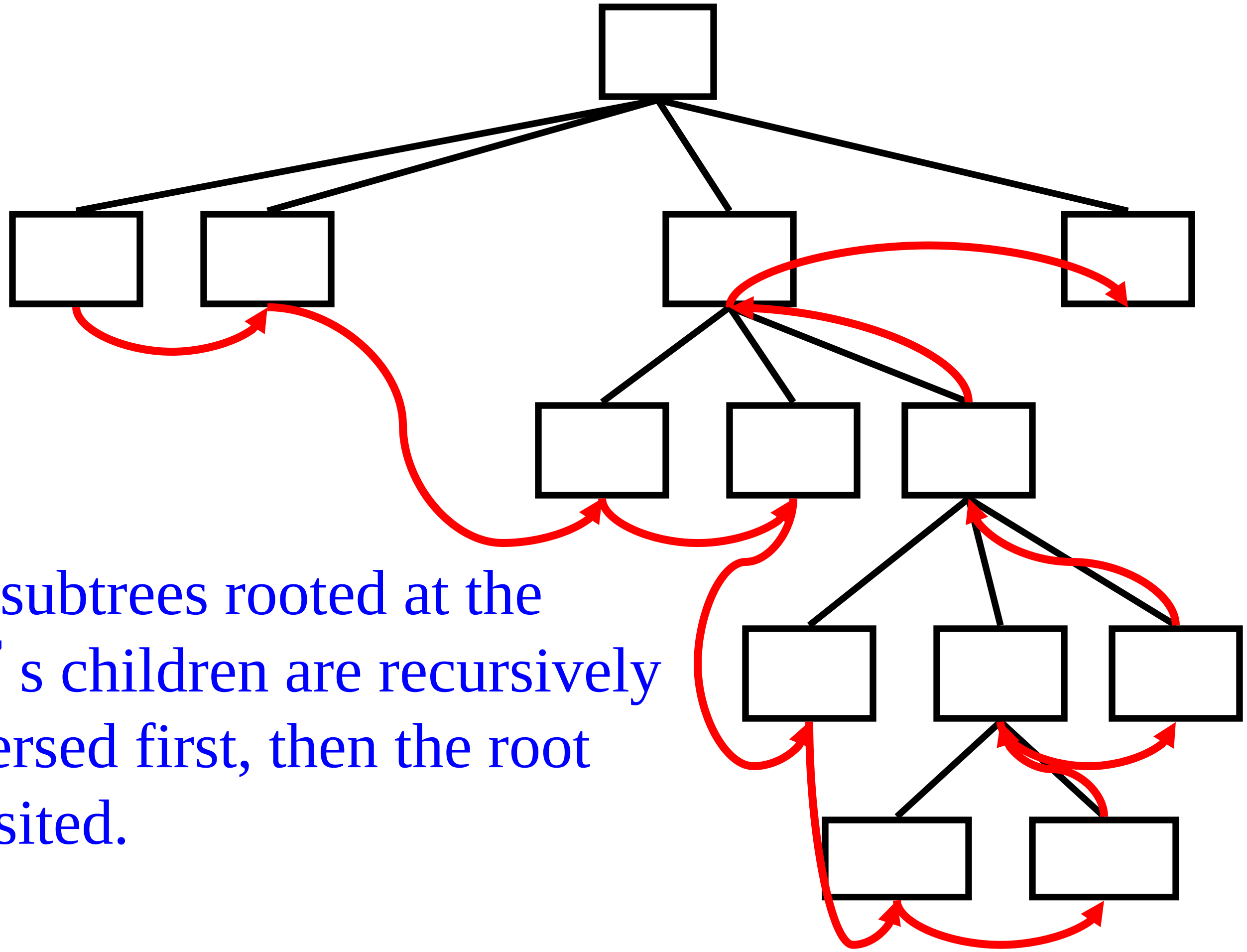
Postorder Traversal



The subtrees rooted at the root's children are recursively traversed first, then the root is visited.

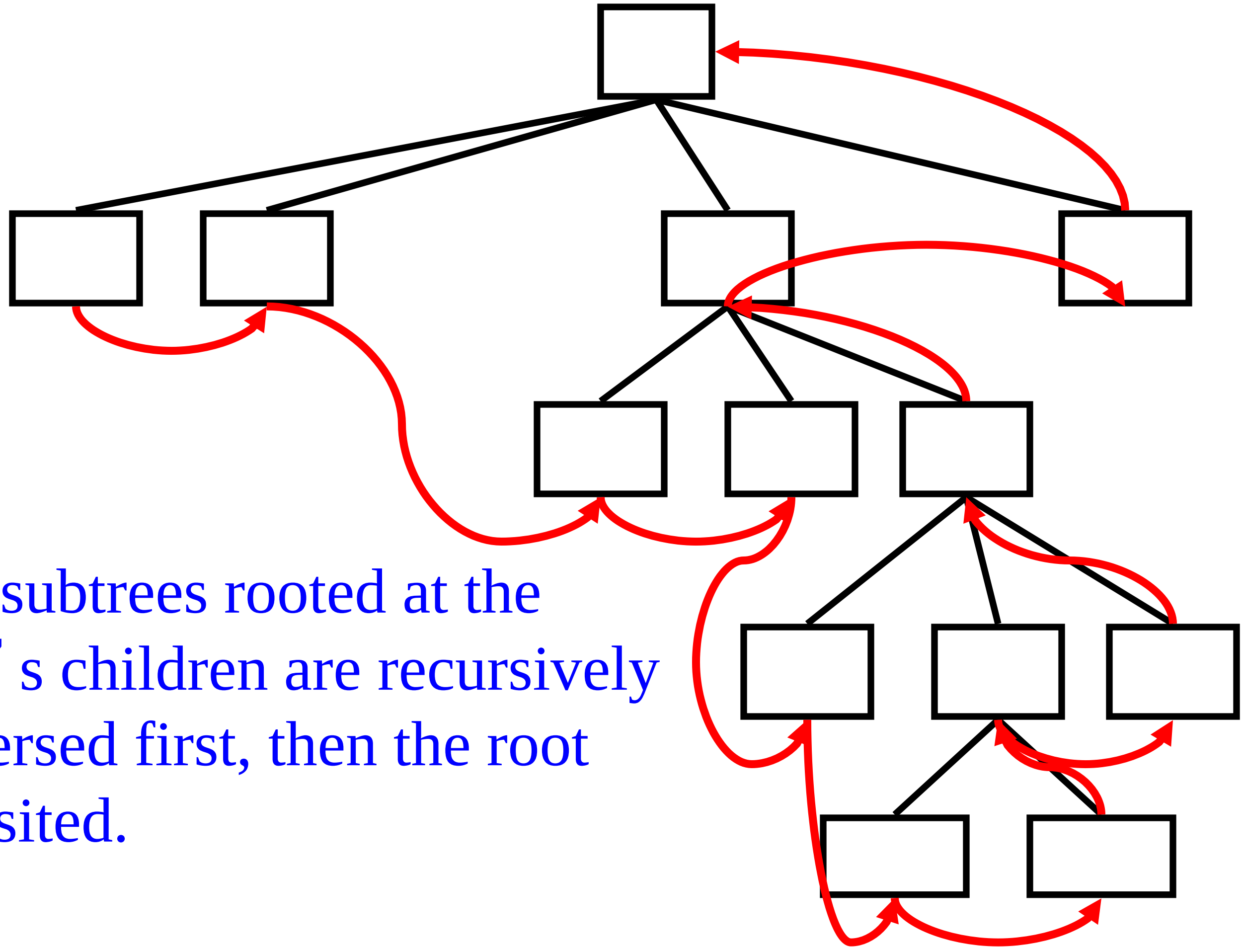
.

Postorder Traversal



The subtrees rooted at the root’s children are recursively traversed first, then the root is visited.
.

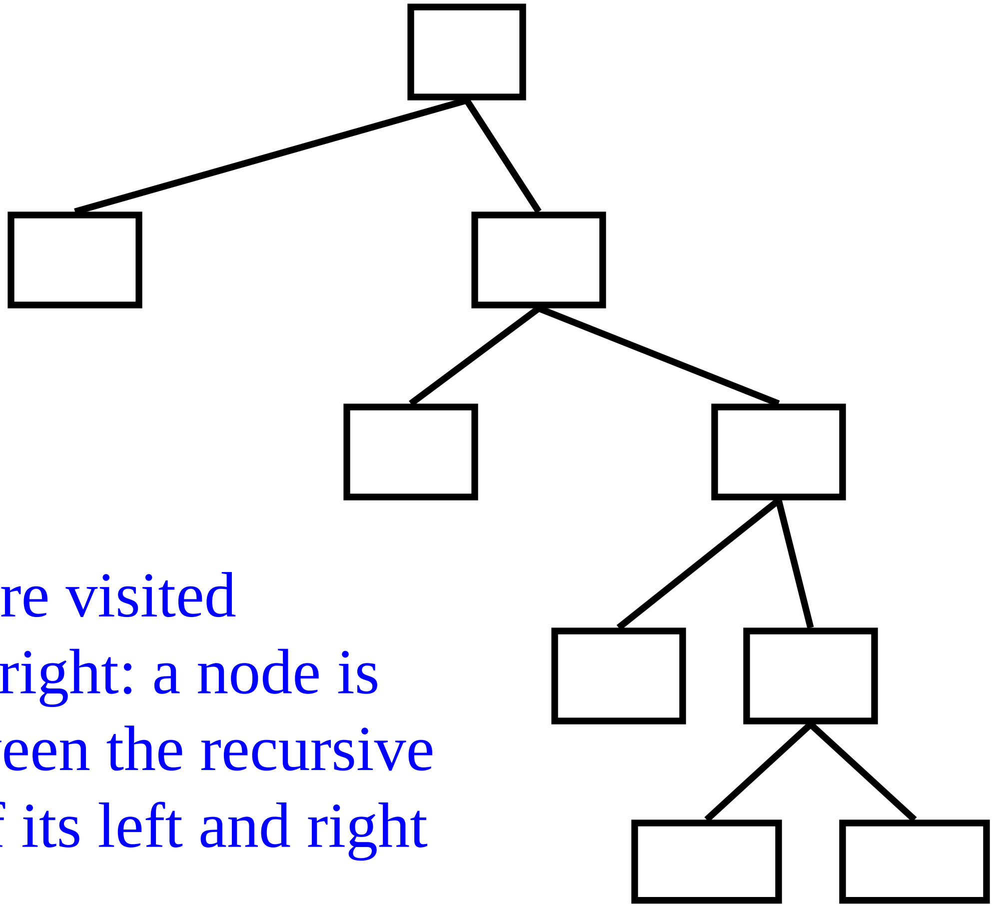
Postorder Traversal



The subtrees rooted at the root's children are recursively traversed first, then the root is visited.

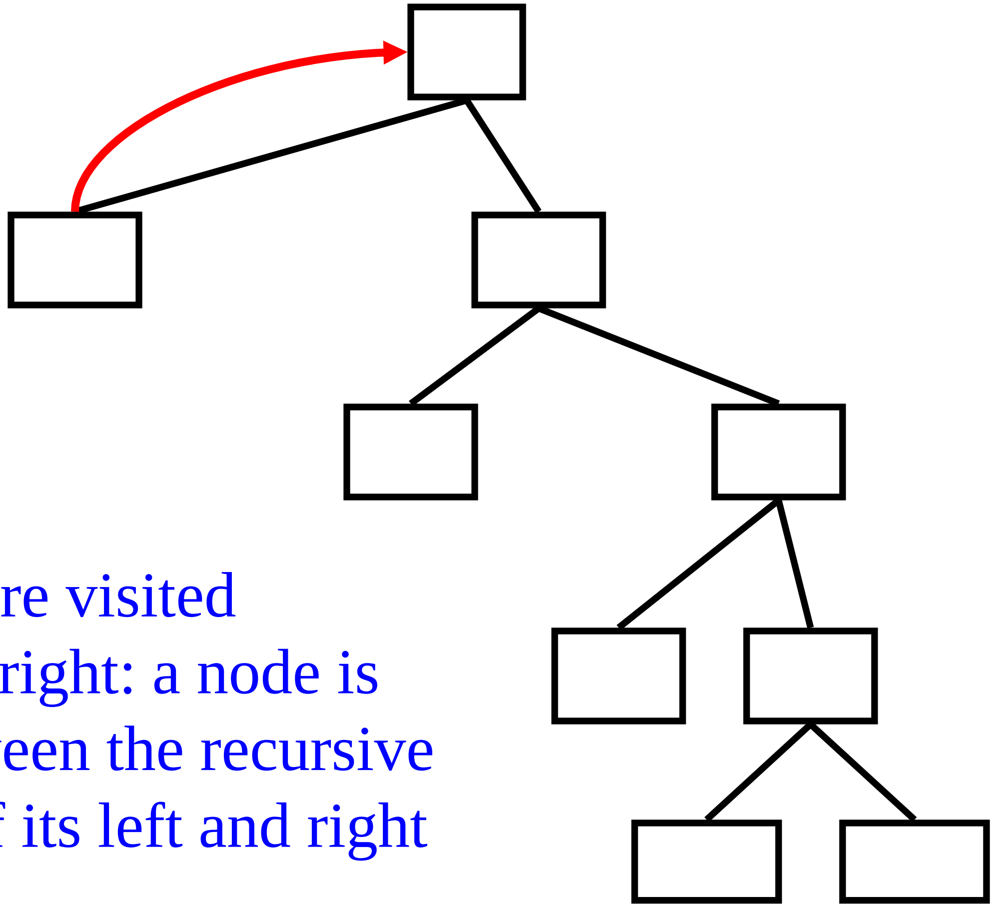
.

Inorder Traversal for Binary Trees



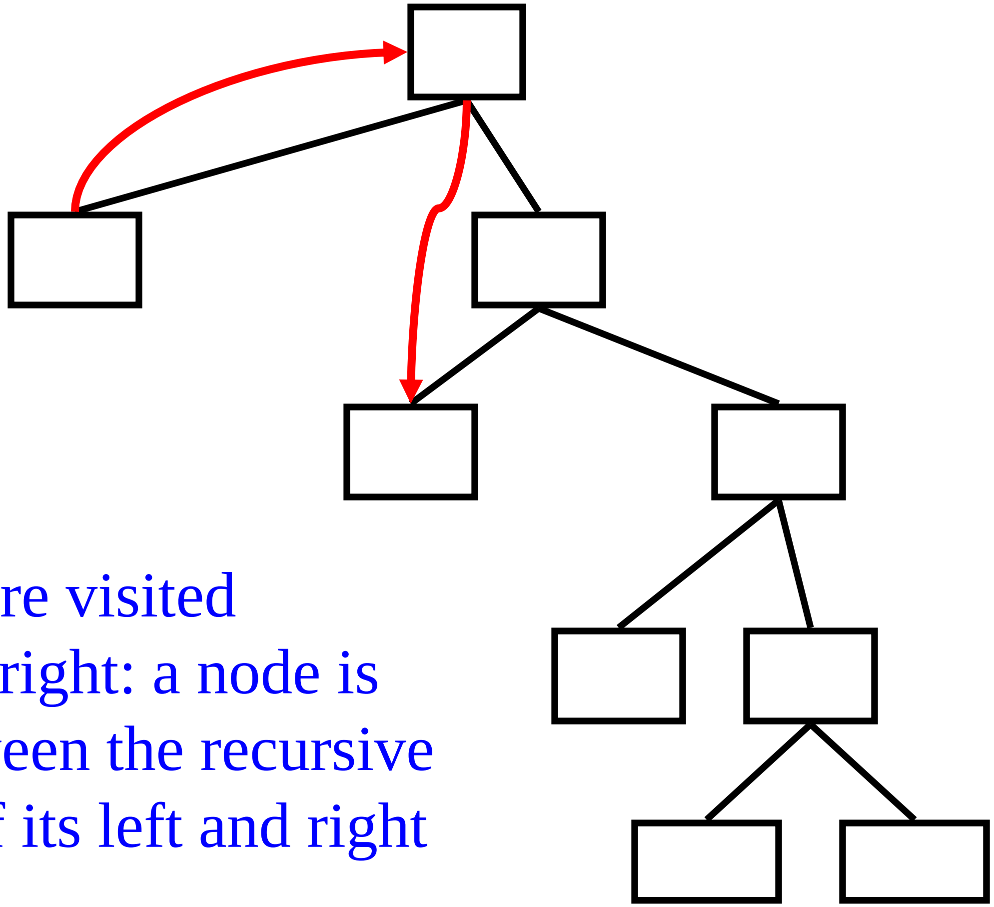
The nodes are visited from left to right: a node is visited between the recursive traversals of its left and right subtrees.

Inorder Traversal for Binary Trees



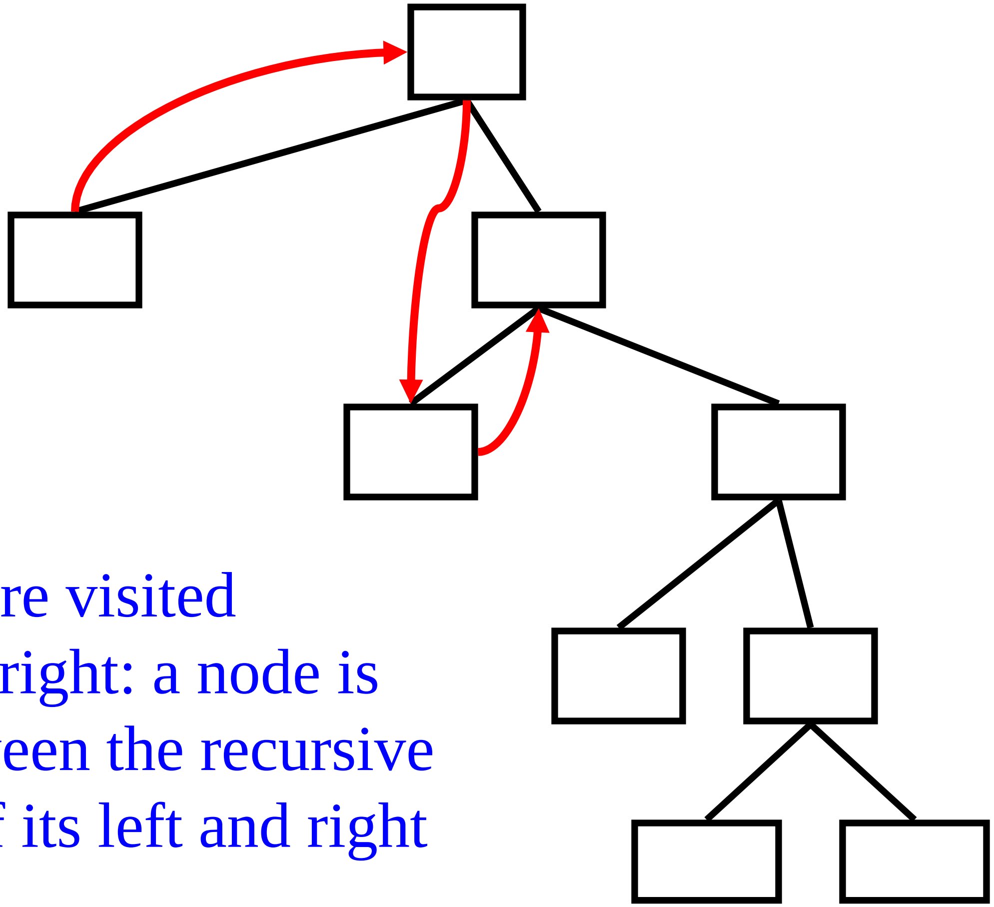
The nodes are visited from left to right: a node is visited between the recursive traversals of its left and right subtrees.

Inorder Traversal for Binary Trees



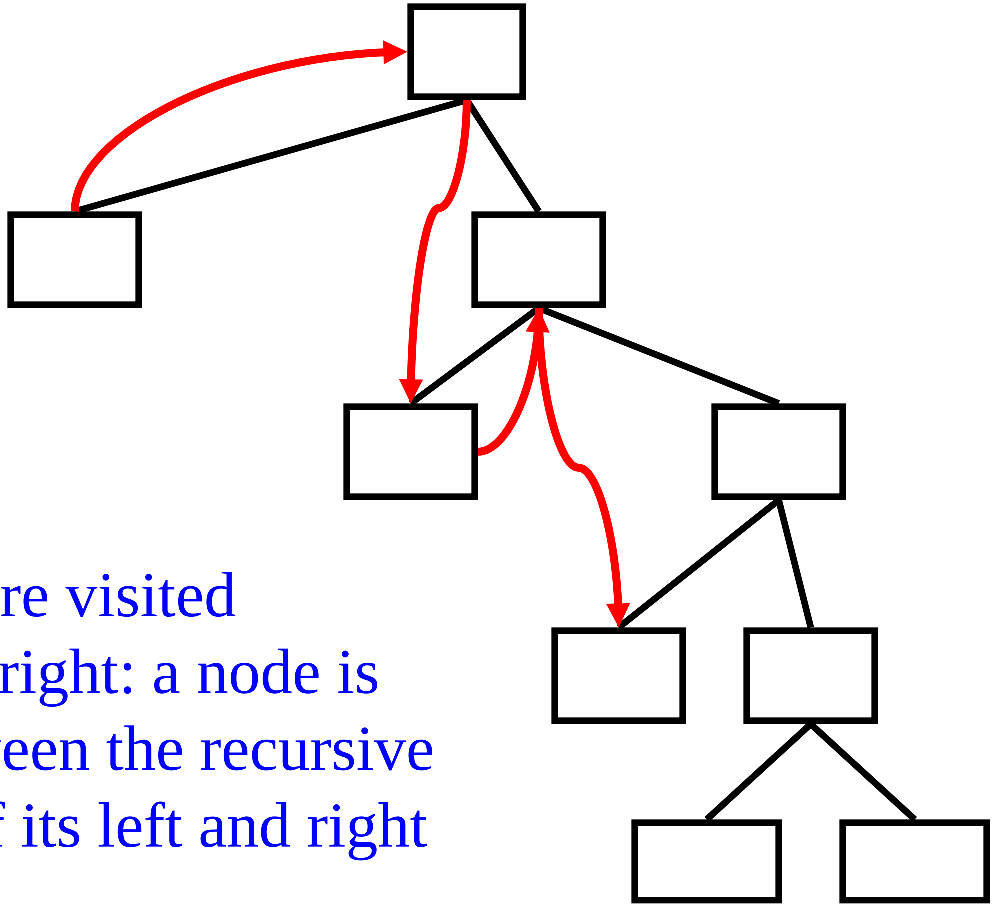
The nodes are visited from left to right: a node is visited between the recursive traversals of its left and right subtrees.

Inorder Traversal for Binary Trees



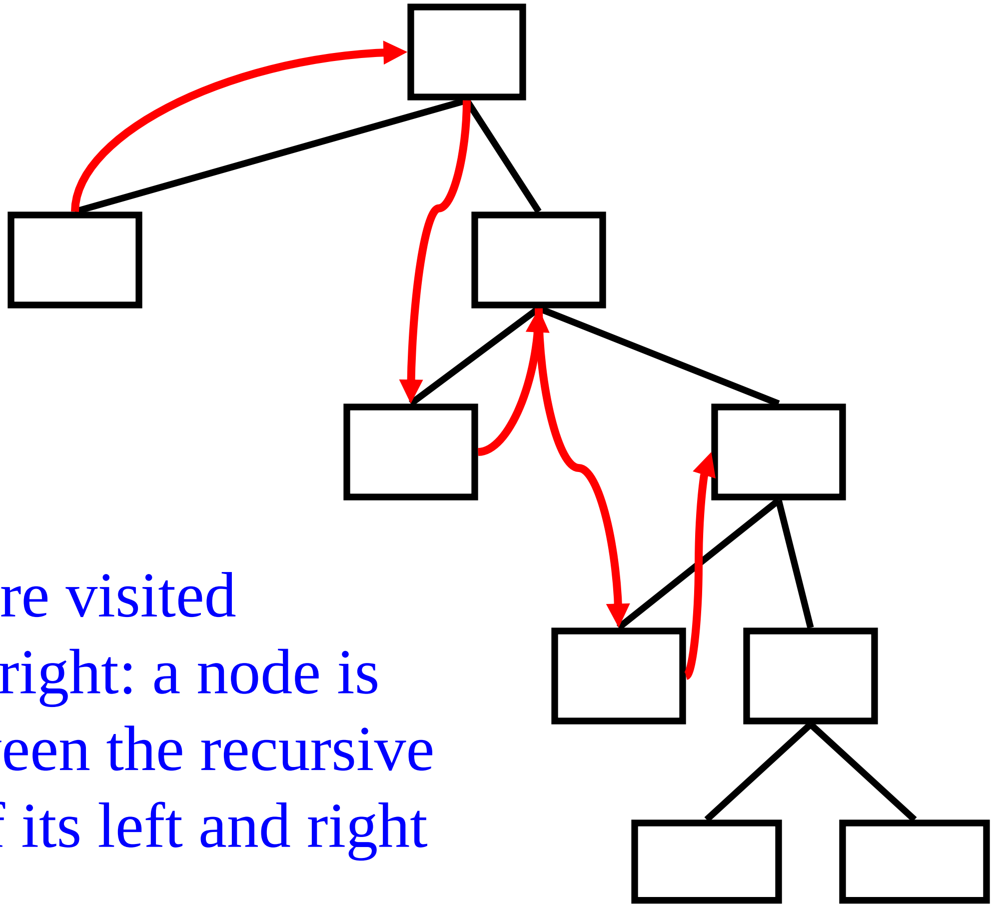
The nodes are visited from left to right: a node is visited between the recursive traversals of its left and right subtrees.

Inorder Traversal for Binary Trees



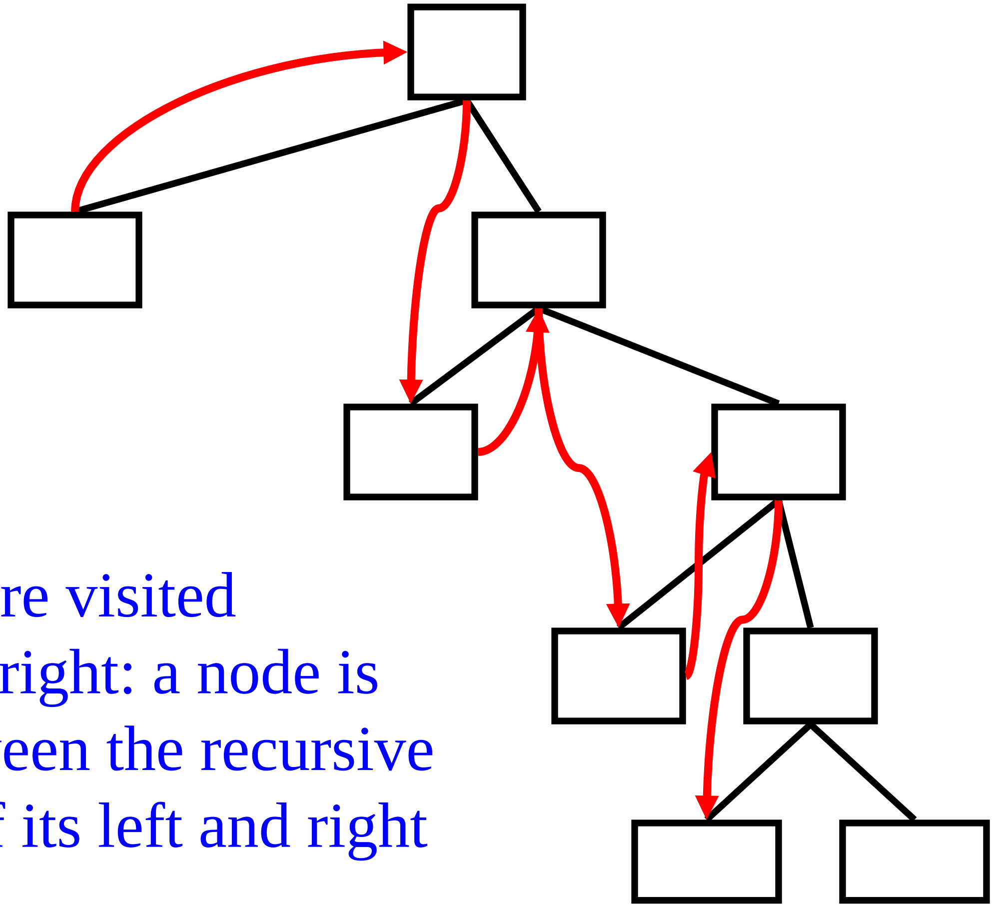
The nodes are visited from left to right: a node is visited between the recursive traversals of its left and right subtrees.

Inorder Traversal for Binary Trees



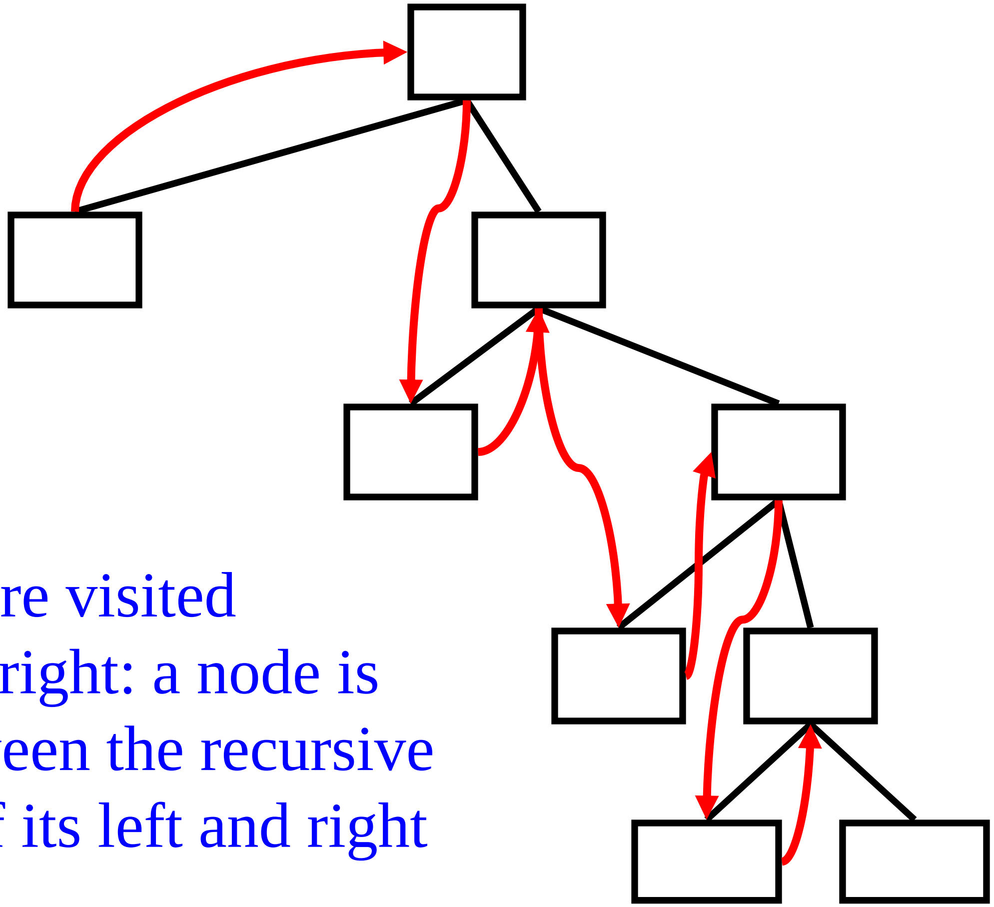
The nodes are visited from left to right: a node is visited between the recursive traversals of its left and right subtrees.

Inorder Traversal for Binary Trees



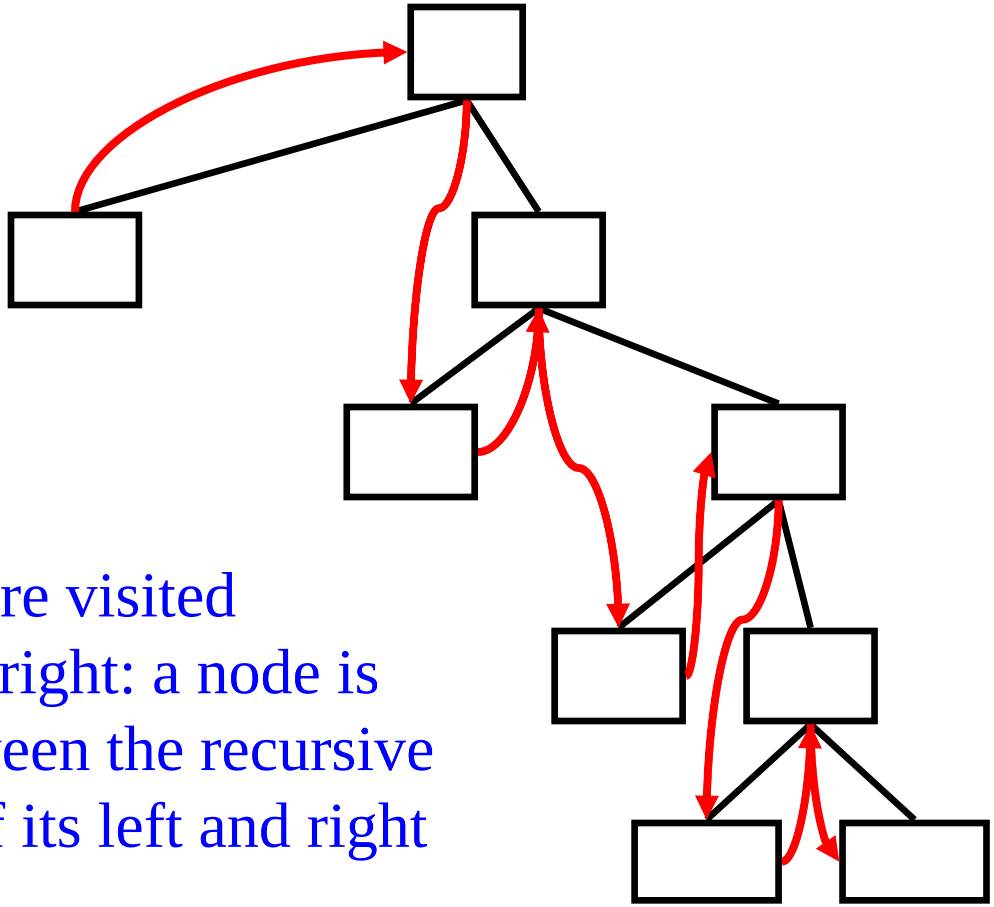
The nodes are visited from left to right: a node is visited between the recursive traversals of its left and right subtrees.

Inorder Traversal for Binary Trees



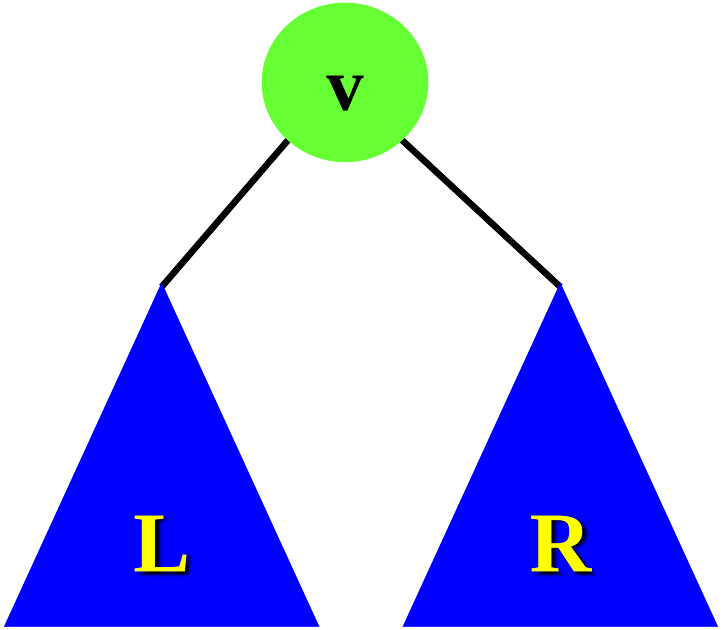
The nodes are visited from left to right: a node is visited between the recursive traversals of its left and right subtrees.

Inorder Traversal for Binary Trees



The nodes are visited from left to right: a node is visited between the recursive traversals of its left and right subtrees.

Tree Traversals



- Preorder
  - v, L, R
- Postorder
  - L, R, v
- Inorder
  - L, v, R
- Levelorder
  - v, vL, vR, L, R

Tree Representations

- Dynamic data structures
  - Using references or pointers
- Heap encoding
  - Using an array

Dynamic Tree Data Structure

```
public class TreeNode {
        private TreeNode left;
        private TreeNode right;
        private TreeNode parent;
        private int data;

        void setLeft(TreeNode t) { left = t;}
        TreeNode getLeft() { return left; }

        void setRight(TreeNode t) { right = t;}
        TreeNode getRight() { return right; }

        void setParent(TreeNode p) { parent = p;}
        TreeNode getParent() { return parent; }

        void setData(int d) { data = d;}
        int getData() { return data; }
}
```

<table>
<tr><td colspan="2">data</td></tr>
<tr><td colspan="2">parent</td></tr>
<tr><td>left</td><td>right</td></tr>
</table>

Preorder Traversal
Depth First Search (DFS)

```
algorithm preorder(Tree t)
   if t <> null then
         processNode(t.v)
         preorder(t.left)
         preorder(t.right)
    end
end
```

Inorder Traversal

```
algorithm inorder(Tree t)
  if t <> null then
      inorder(t.left)
      processNode(t.v)
      inorder(t.right)
   end
end
```

Postorder Traversal

```
algorithm postorder(Tree t)
  if t <> null then
      postorder(t.left)
      postorder(t.right)
      processNode(t.v)
   end
end
```

Levelorder Traversal
Breadth First Search (BFS)

```
algorithm levelorder(Tree t)
    Queue q
    q.enqueue(t)
    while not q.empty() do
        t = q.dequeue()
        processNode(t)
        if t.left <> null then q.enqueue(t.left) end
        if t.right <> null then q.enqueue(t.right) end
    end
end
```

Non-recursive Preorder Traversal Depth First Search (DFS)

```
algorithm dfs(Tree t)
    if t = null then return end
    Stack s
    s. push(t)
    while not s.empty() do
        t = s.pop()
        processNode(t)
        if t.right <> null then s.push(t.right) end
        if t.left <> null then s.push(t.left) end
    end
end
```

Running Time of Tree Traversals

- Each node is visited a fixed number of times (i.e., 3 times)
- **Theorem**
  - The time complexity of Preorder, Inorder, Postorder is $O(n)$

Tree Traversal Summary

Pre-order	Depth first	Stack
Inorder	Symmetric	Stack
Postorder	Bottom up	Stack
Levelorder	Breadth first	Queue

Tree Traversal Numberings

```
pre = 1;   in = 1;   post = 1
algorithm eulerNumberings(Tree t)
   if t <> null then
         t.pre = pre;   pre = pre + 1
         eulerNumberings(t.left)
         t.in = in;   in = in + 1
         eulerNumberings(t.right)
         t.post = post;   post = post + 1
    end
end
```

Heap Encoding

Parent of A[k] is at A[k/2]

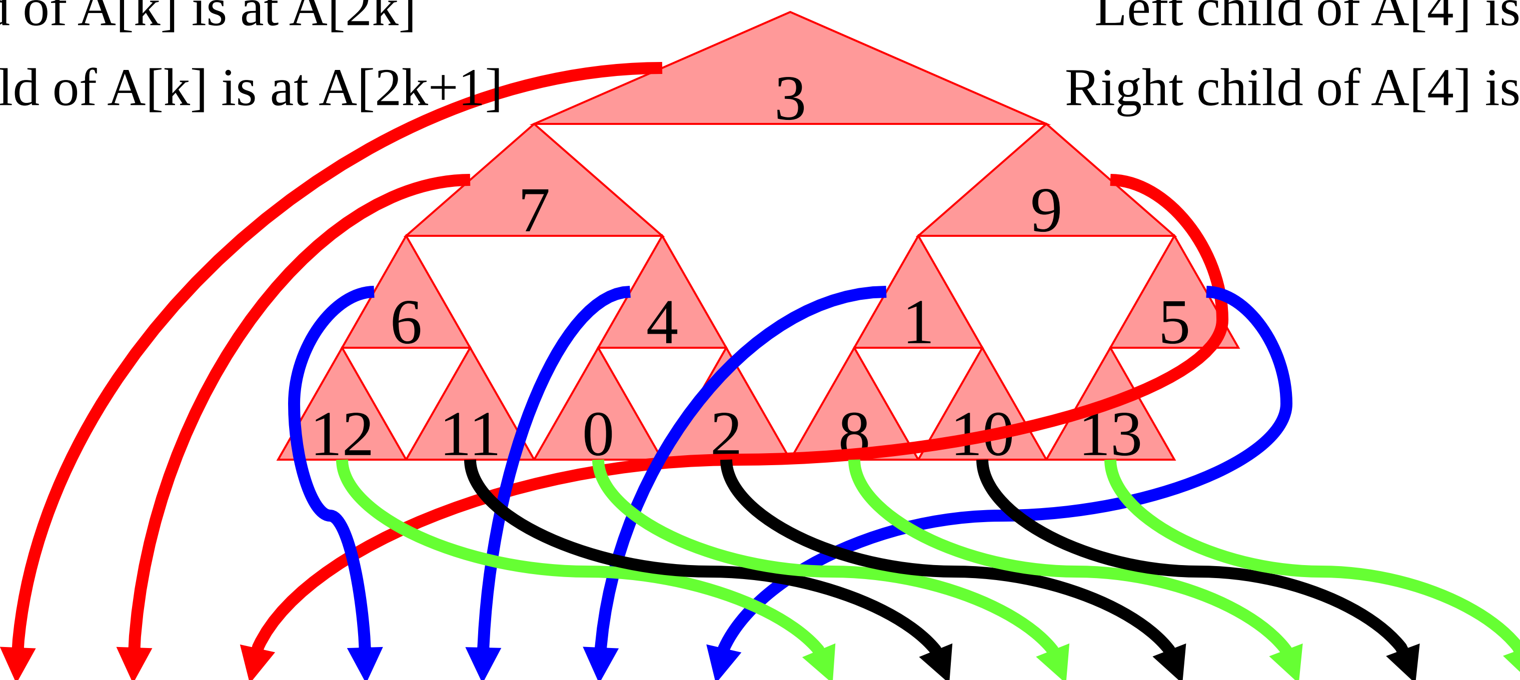
Left child of A[k] is at A[2k]

Right child of A[k] is at A[2k+1]

Parent of A[6] is at A[3]

Left child of A[4] is at A[8]

Right child of A[4] is at A[9]



A	3	7	9	6	4	1	5	12	11	0	2	8	10	13
index	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Note index starts at 1

Data Structures for *binary* Trees

Operation	Time with vector-based structure	Time with linked structure
positions, elements traversals (iterators): pre-, in-, post-, level-order	$O(n)$	$O(n)$
size, isEmpty	O(1)	O(1)
swapElements, replaceElement	O(1)	O(1)
leftChild, rightChild, sibling	O(1)	O(1)
isInternal, isExternal, isRoot	O(1)	O(1)
root, parent, child	O(1)	O(1)