Topic 14

The BinaryTree ADT

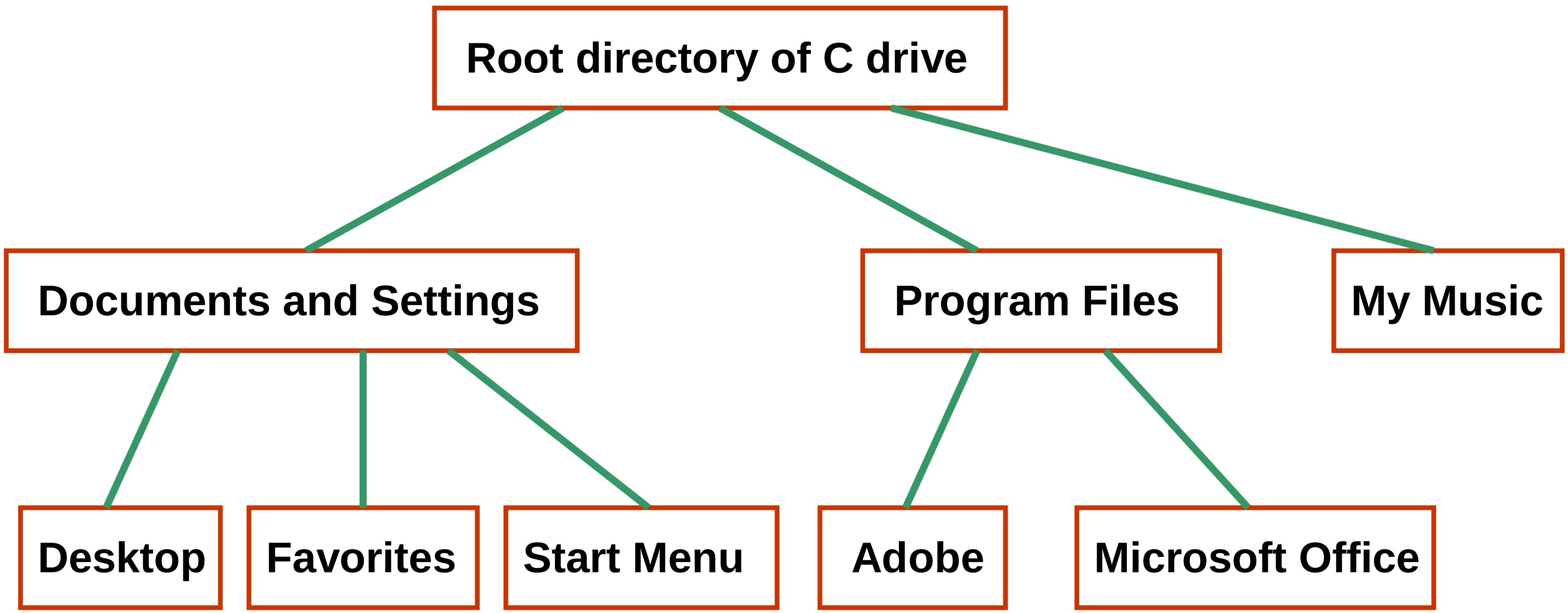
Objectives

- Define trees as data structures
- Define the terms associated with trees
- Discuss tree traversal algorithms
- Discuss a binary tree implementation
- Examine a binary tree example

Trees

- A ***tree*** is a ***nonlinear*** data structure used to represent entities that are in some ***hierarchical*** relationship
- ***Examples*** in real life:
  - Family tree
  - Table of contents of a book
  - Class inheritance hierarchy in Java
  - Computer file system (folders and subfolders)
  - Decision trees
  - Top-down design

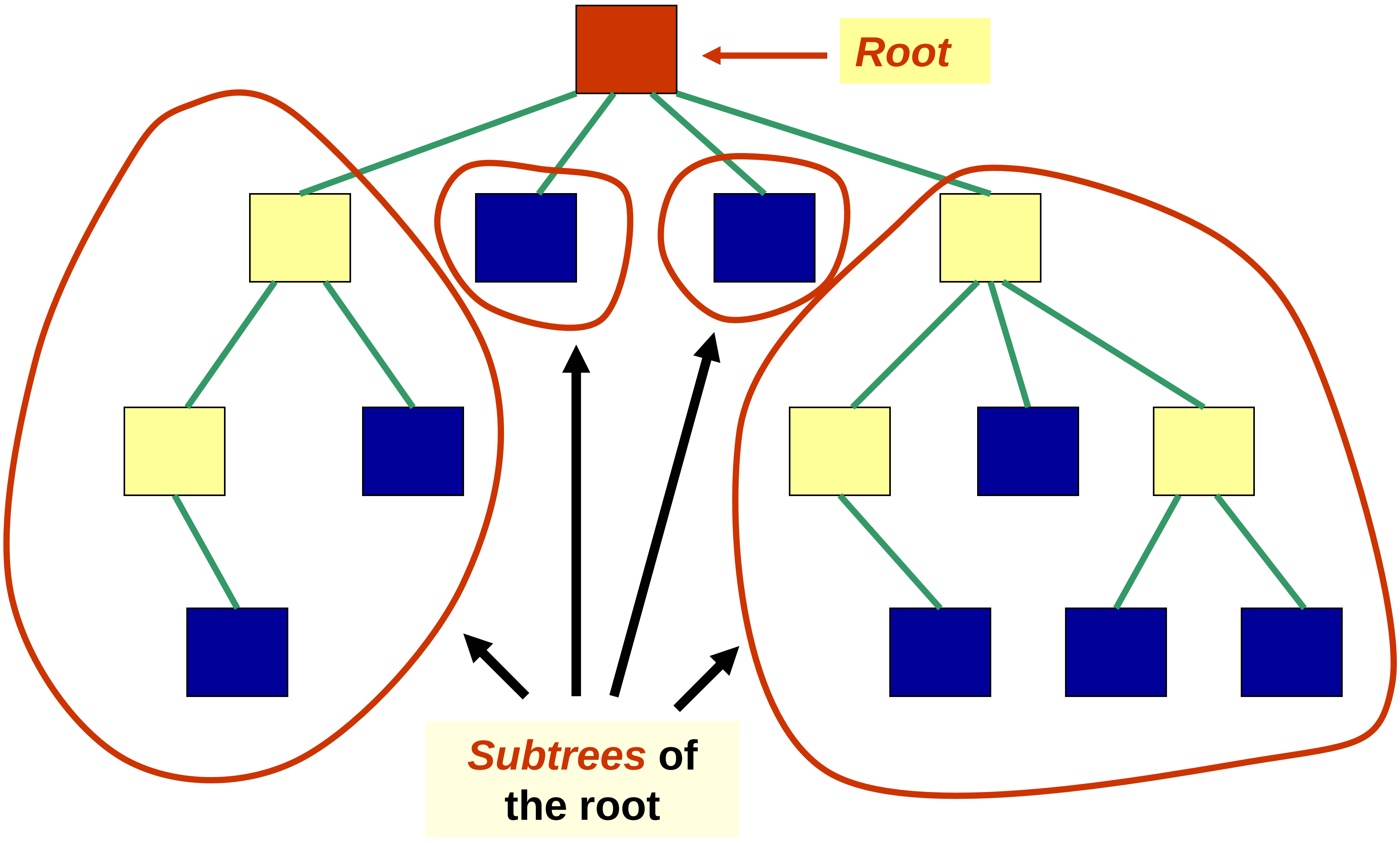
Example: Computer File System



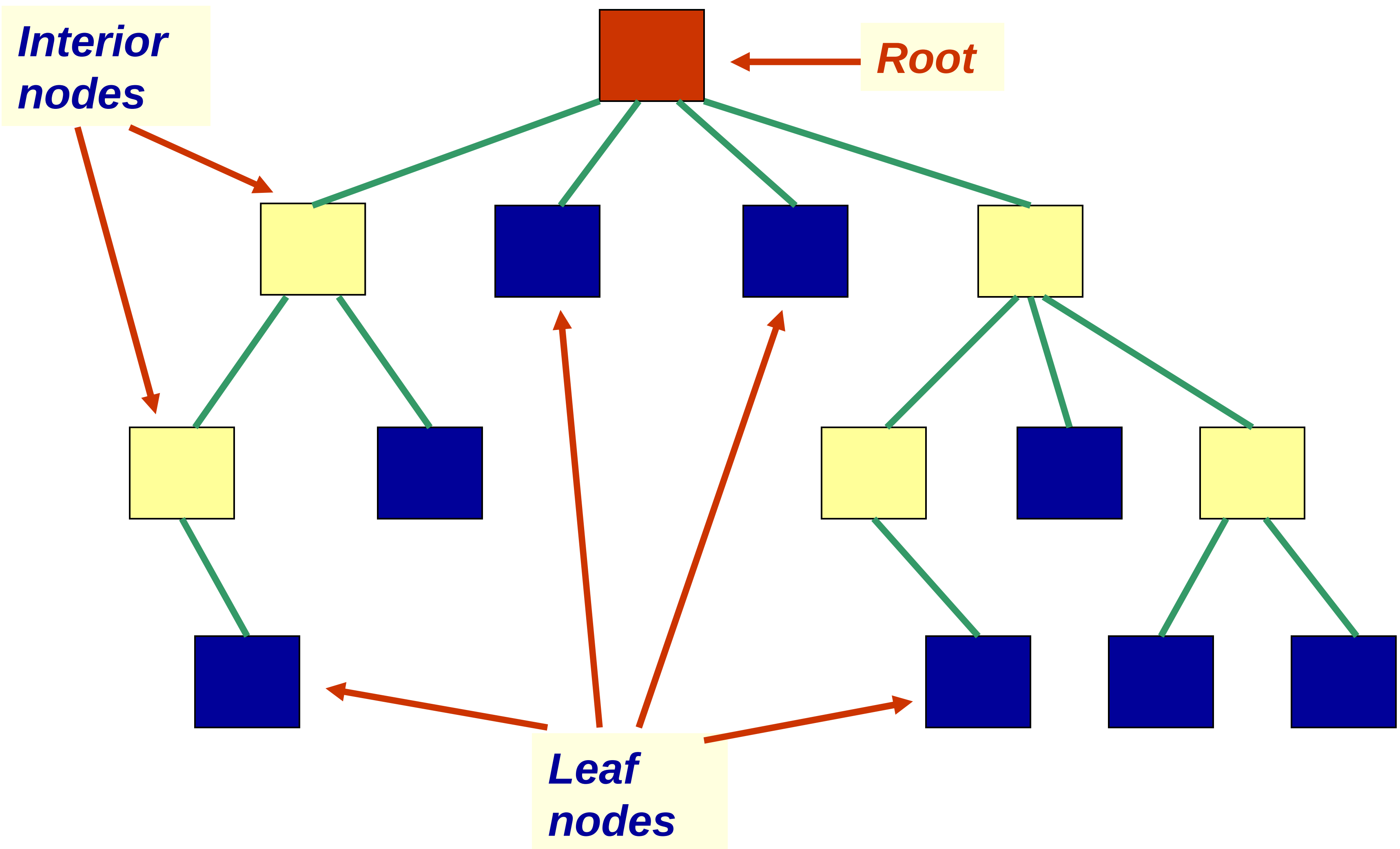
Tree Definition

- ***Tree:*** a set of elements of the same type such that
  - It is empty
  - Or, it has a distinguished element called the ***root*** from which descend zero or more ***trees*** ***(subtrees)***
- What kind of definition is this?
  - What is the base case?
  - What is the recursive part?

Tree Definition



Tree Terminology



Tree Terminology

- ***Nodes***: the elements in the tree
- ***Edges***: connections between nodes
- ***Root***: ***the*** distinguished element that is the origin of the tree
  - There is only one root node in a tree
- ***Leaf node***: a node without an edge to another node
- ***Interior node***: a node that is not a leaf node
- ***Empty tree*** has no nodes and no edges

Tree Terminology

- ***Parent*** or ***predecessor***: the node directly above in the hierarchy
  - A node can have only one parent
- ***Child*** or ***successor***: a node directly below in the hierarchy
- ***Siblings***: nodes that have the same parent
- ***Ancestors*** of a node*:* its parent, the parent of its parent, etc.
- ***Descendants*** of a node: its children, the children of its children, etc.

Discussion

- Does a leaf node have any children?
- Does the root node have a parent?
- How many parents does every node other than the root node have?

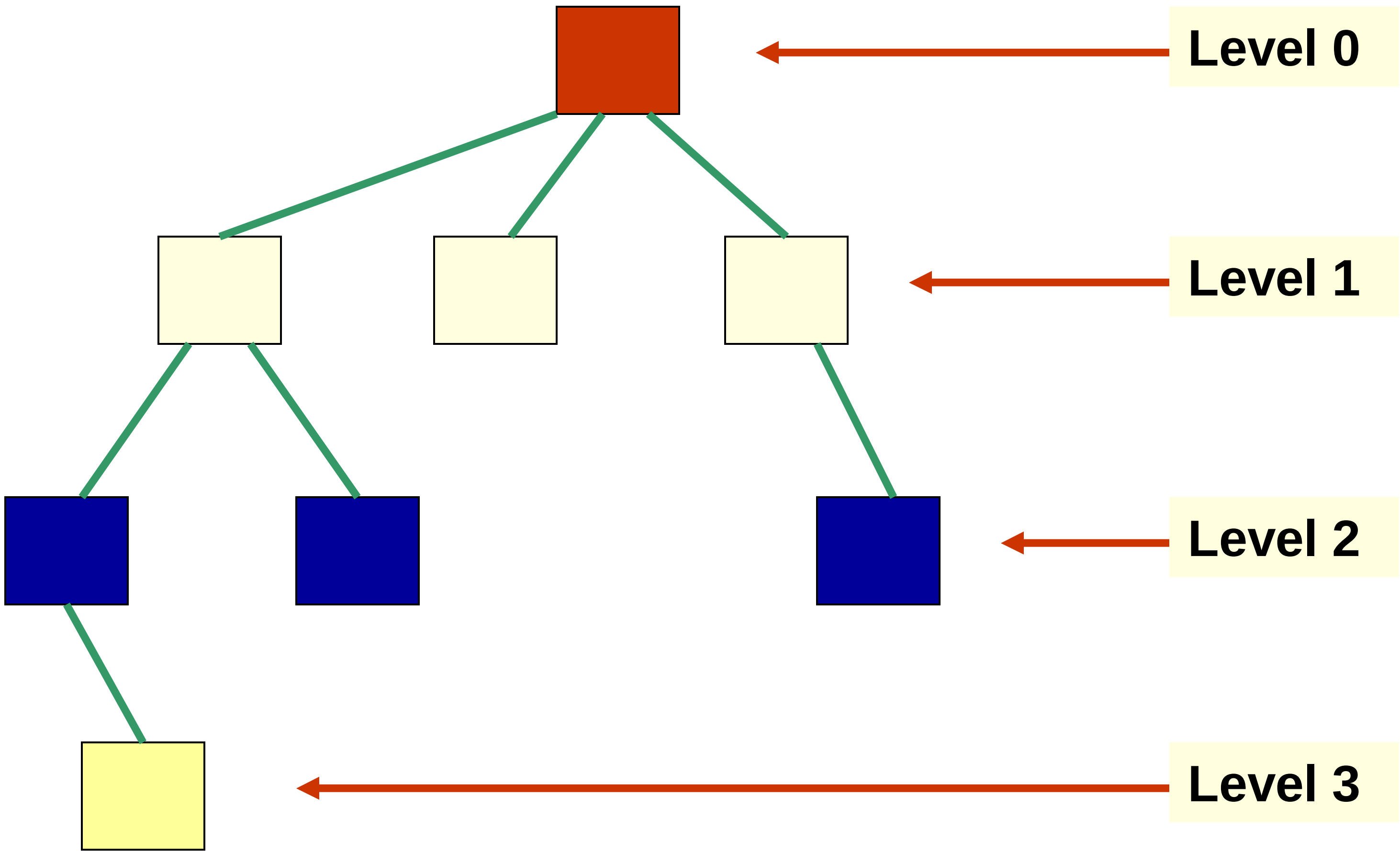
Height of a Tree

- A ***path*** is a sequence of edges leading from one node to another
- ***Length of a path***: number of edges on the path
- ***Height of a (non-empty) tree*** : length of the longest path from the root to a leaf
  - What is the height of a tree that has only a root node?
  - By convention, the ***height of an empty tree*** is **-1**

Level of a Node

- ***Level of a node*** ***:*** number of ***edges*** between root and node
- It can be defined ***recursively***:
  - Level of root node is ***0***
  - Level of a node that is not the root node is ***level of its parent + 1***
- ***Question****: What is the level of a node in terms of path length?*
- ***Question****: What is the height of a tree in terms of levels?*

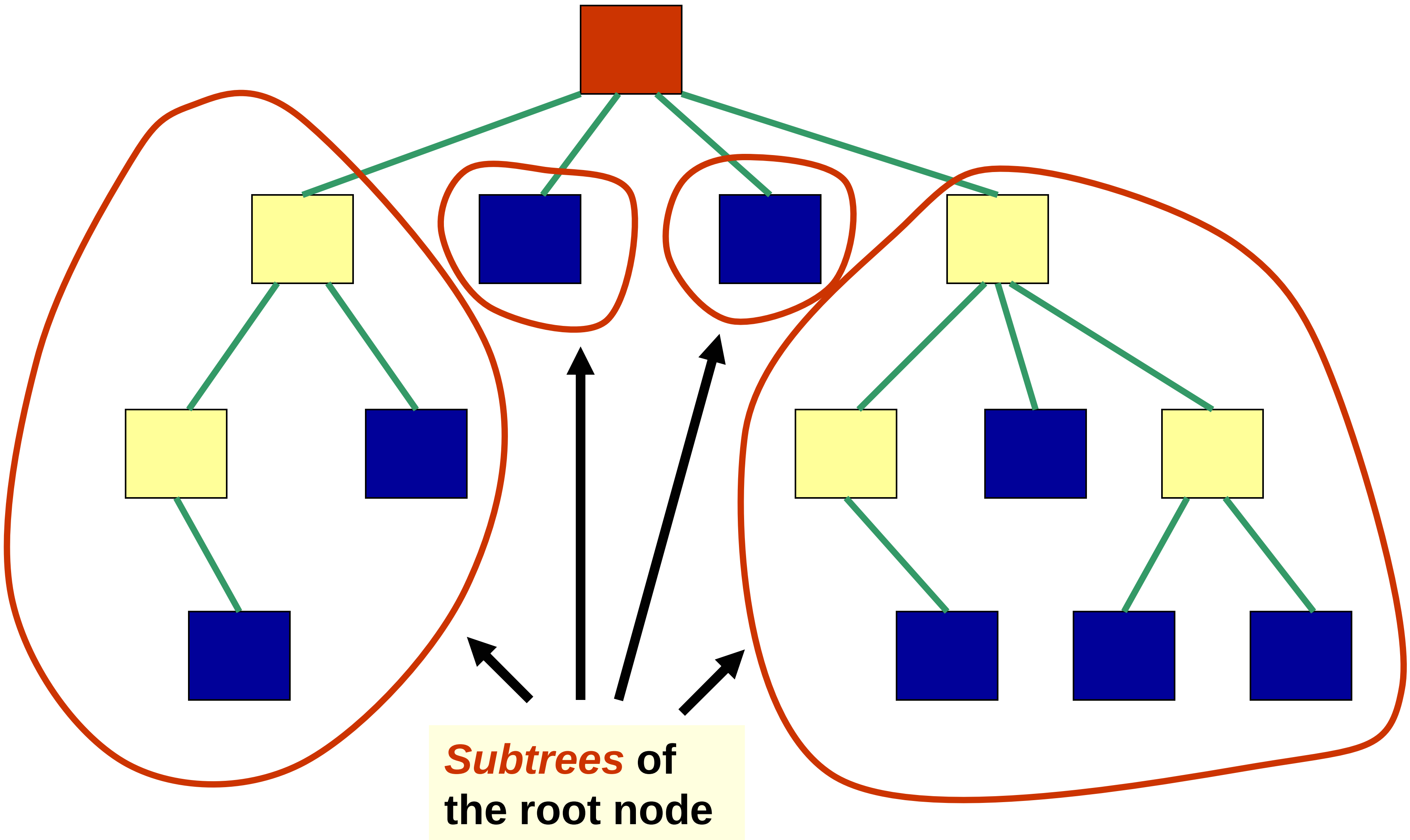
Level of a Node



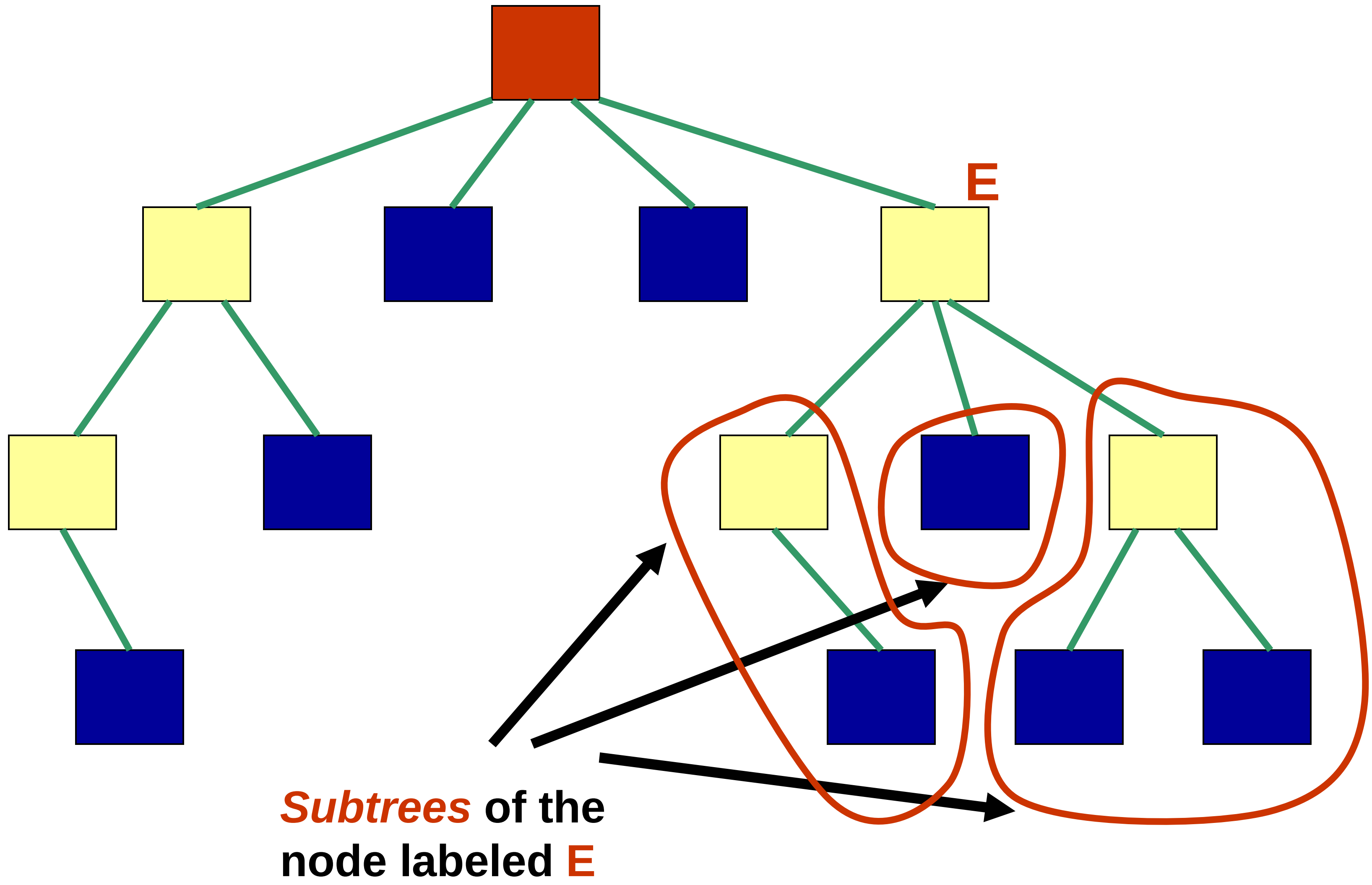
Subtrees

- ***Subtree*** of a node: consists of a child node and all its descendants
  - A subtree is itself a tree
  - A node may have many subtrees

Subtrees



Subtrees



More Tree Terminology

- ***Degree*** or ***arity*** of a ***node***: the number of children it has
- ***Degree*** or ***arity*** of a ***tree***: the ***maximum*** of the degrees of the tree's nodes

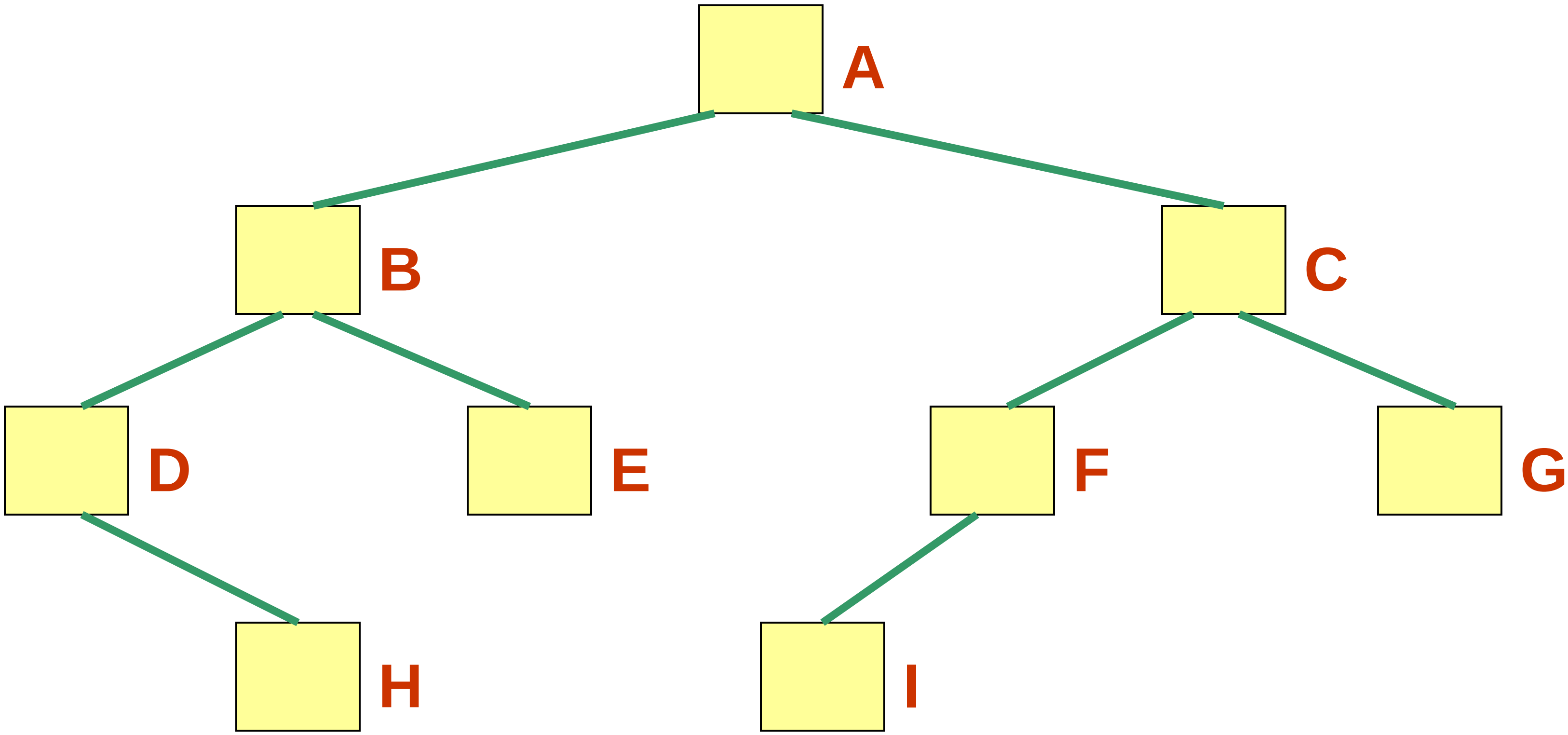
Binary Trees

- ***General tree***: a tree each of whose nodes may have any number of children
- ***n-ary tree***: a tree each of whose nodes may have no more than ***n*** children
- ***Binary tree***: a tree each of whose nodes may have no more than ***2*** children
  - ***i.e.*** a binary tree is a tree with ***degree (arity) 2***
  - The children (if present) are called the ***left child*** and ***right child***

Binary Trees

- ***Recursive definition*** of a ***binary tree***: it is
  - The empty tree
  - Or, a tree which has a root whose left and right subtrees are binary trees
- A binary tree is a ***positional*** tree, ***i.e.*** it matters whether the subtree is left or right

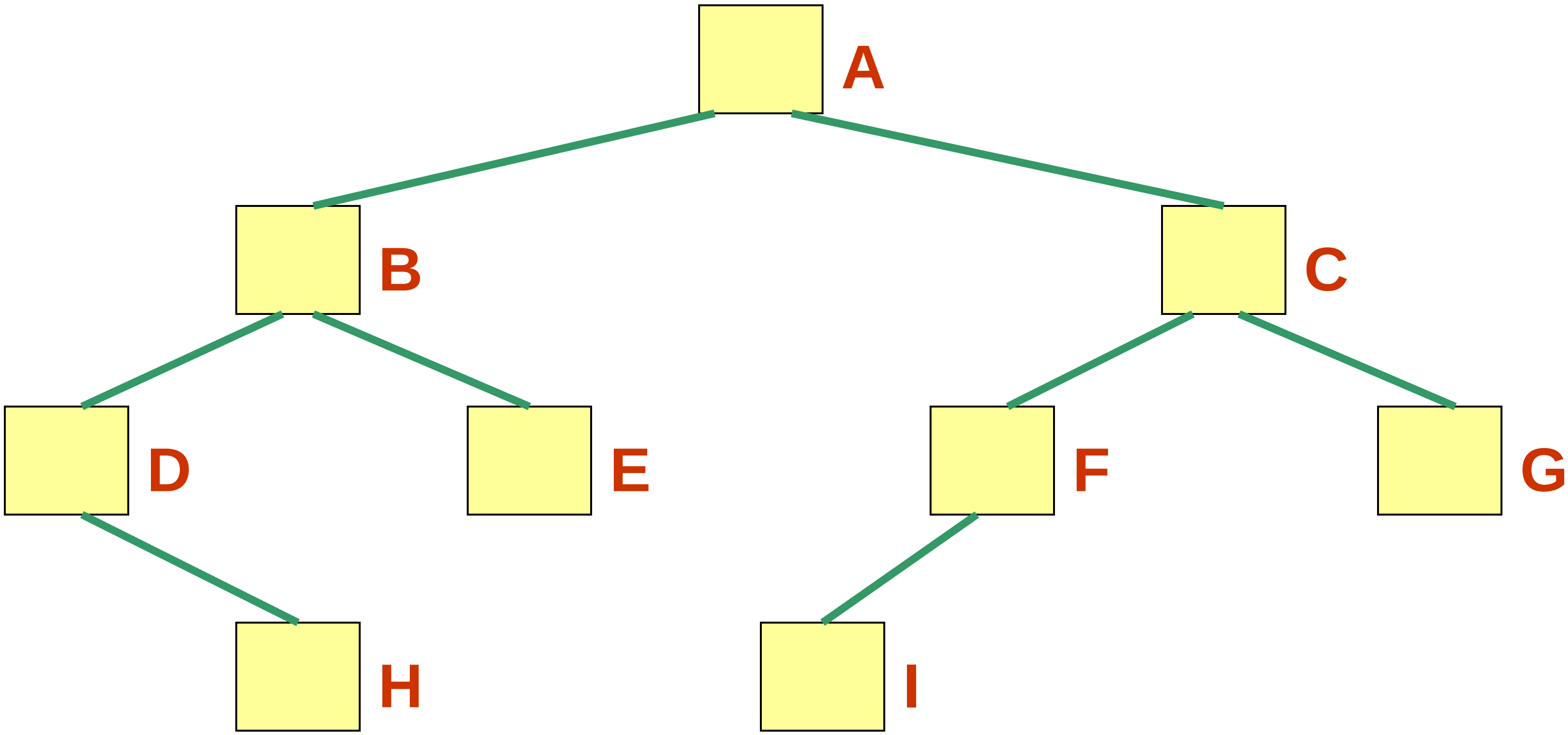
Binary Tree



Tree Traversals

- A ***traversal*** of a tree requires that each node of the tree be ***visited*** once
  - ***Example***: a typical reason to traverse a tree is to display the data stored at each node of the tree
- Standard traversal orderings:
  - ***preorder***
  - ***inorder***
  - ***postorder***
  - ***level-order***

Traversals



We'll trace the different traversals using this tree; recursive calls, returns, and "visits" will be numbered in the order they occur

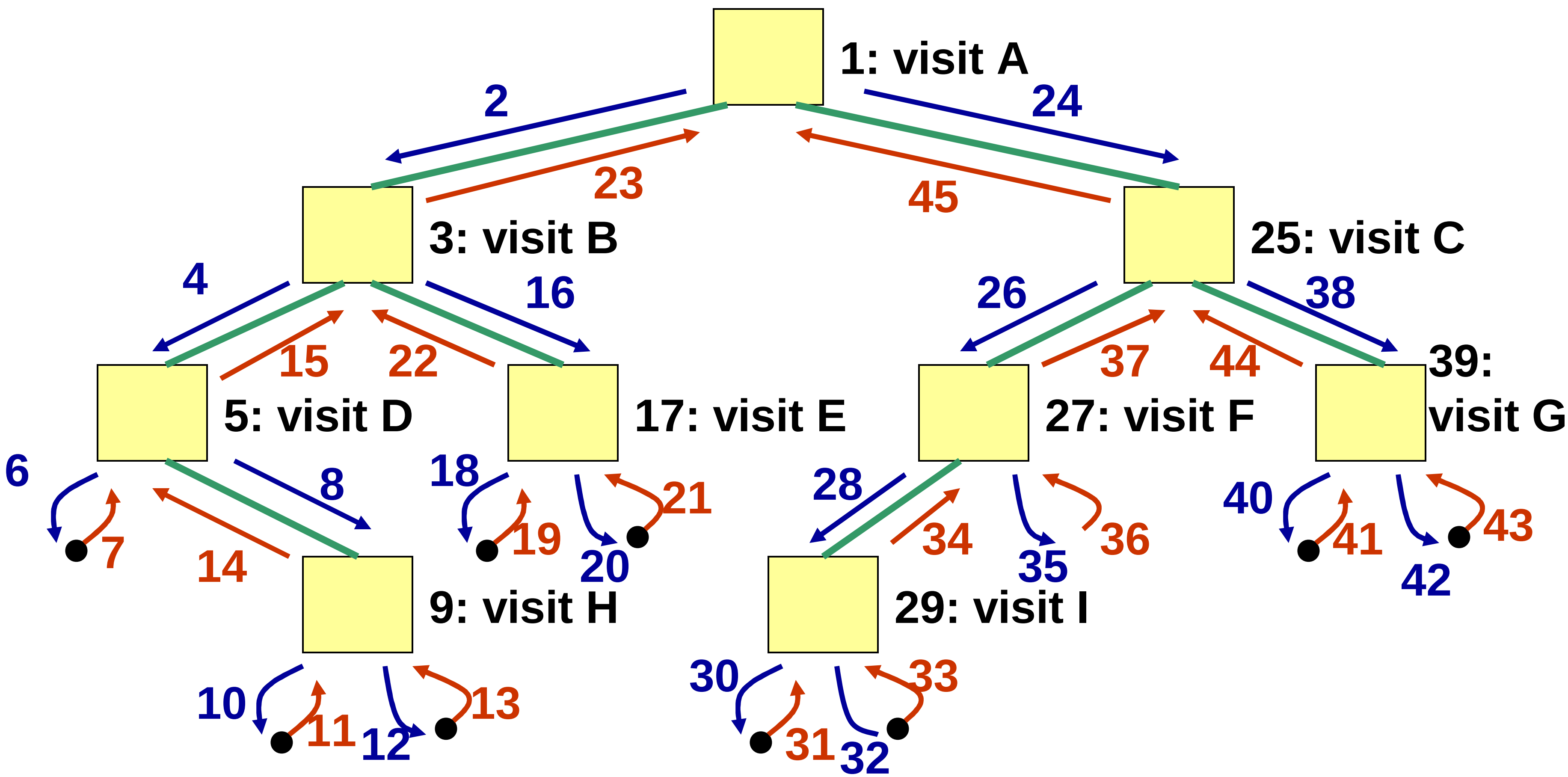
Preorder Traversal

- Start at the root
- Visit each node, followed by its children; we will choose to visit left child before right

- ***Recursive algorithm*** for preorder traversal:
  - If tree is not empty,
    - Visit root node of tree
    - Perform preorder traversal of its left subtree
    - Perform preorder traversal of its right subtree

  - *What is the base case?*
  - *What is the recursive part?*

Preorder Traversal



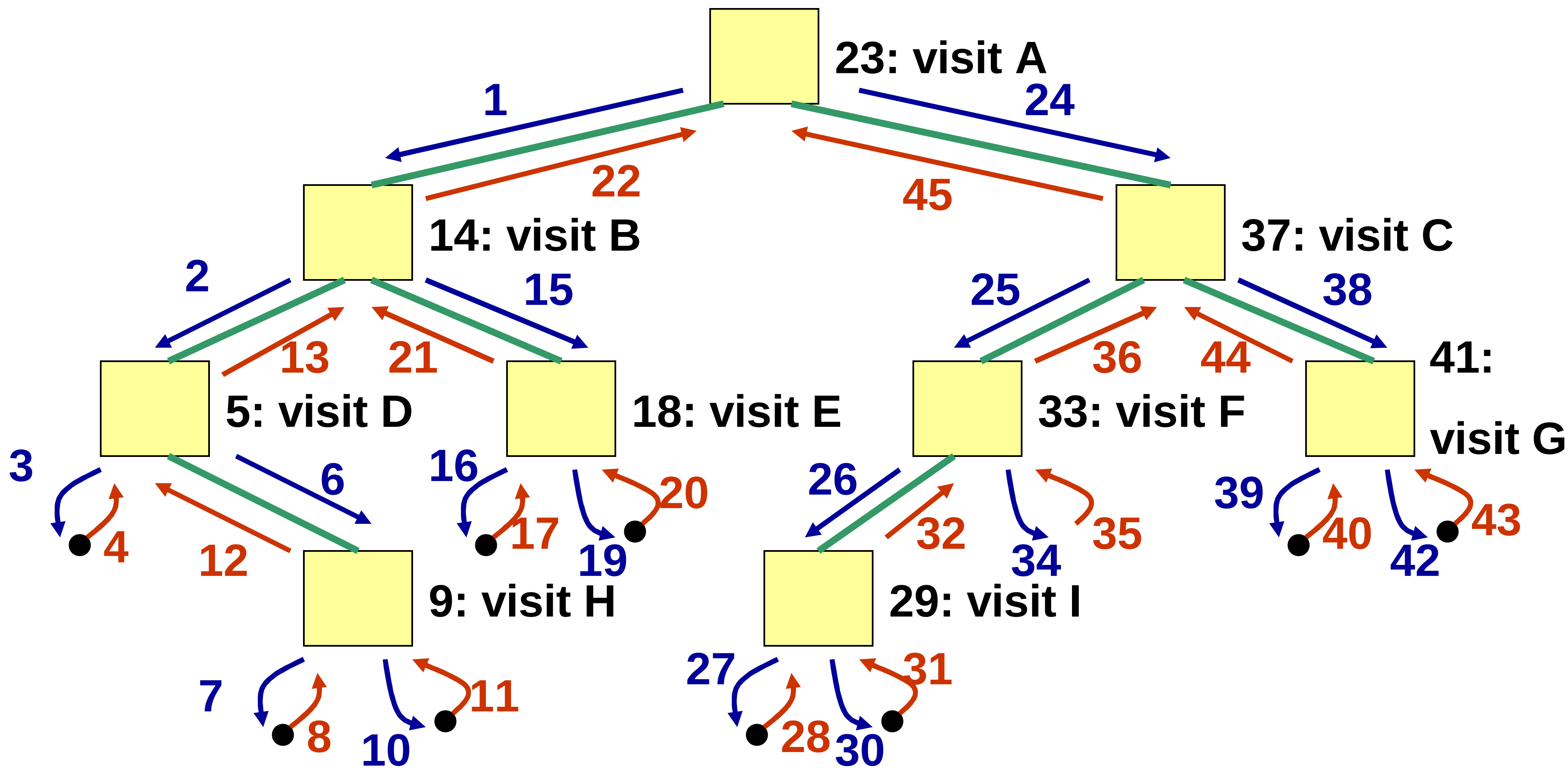
Nodes are visited in the order ABDHECFIG

Inorder Traversal

- Start at the root
- Visit the left child of each node, then the node, then any remaining nodes

- ***Recursive algorithm*** for inorder traversal
  - If tree is not empty,
    - Perform inorder traversal of left subtree of root
    - Visit root node of tree
    - Perform inorder traversal of its right subtree

Inorder Traversal



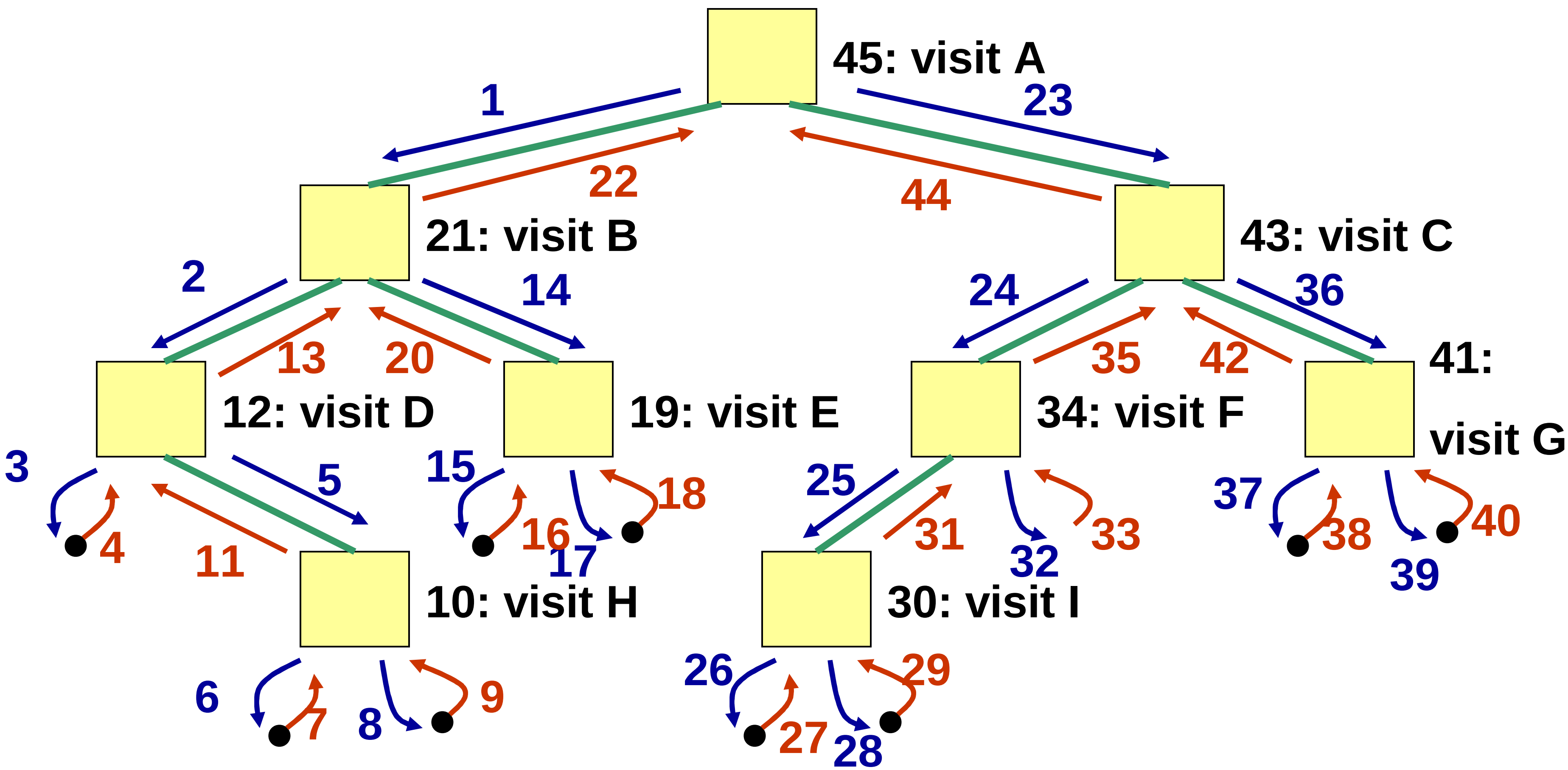
Nodes are visited in the order DHBEAIFCG

Postorder Traversal

- Start at the root
- Visit the children of each node, then the node

- ***Recursive algorithm*** for postorder traversal
  - If tree is not empty,
    - Perform postorder traversal of left subtree of root
    - Perform postorder traversal of right subtree of root
    - Visit root node of tree

Postorder Traversal



Nodes are visited in the order HDEBIFGCA

Discussion

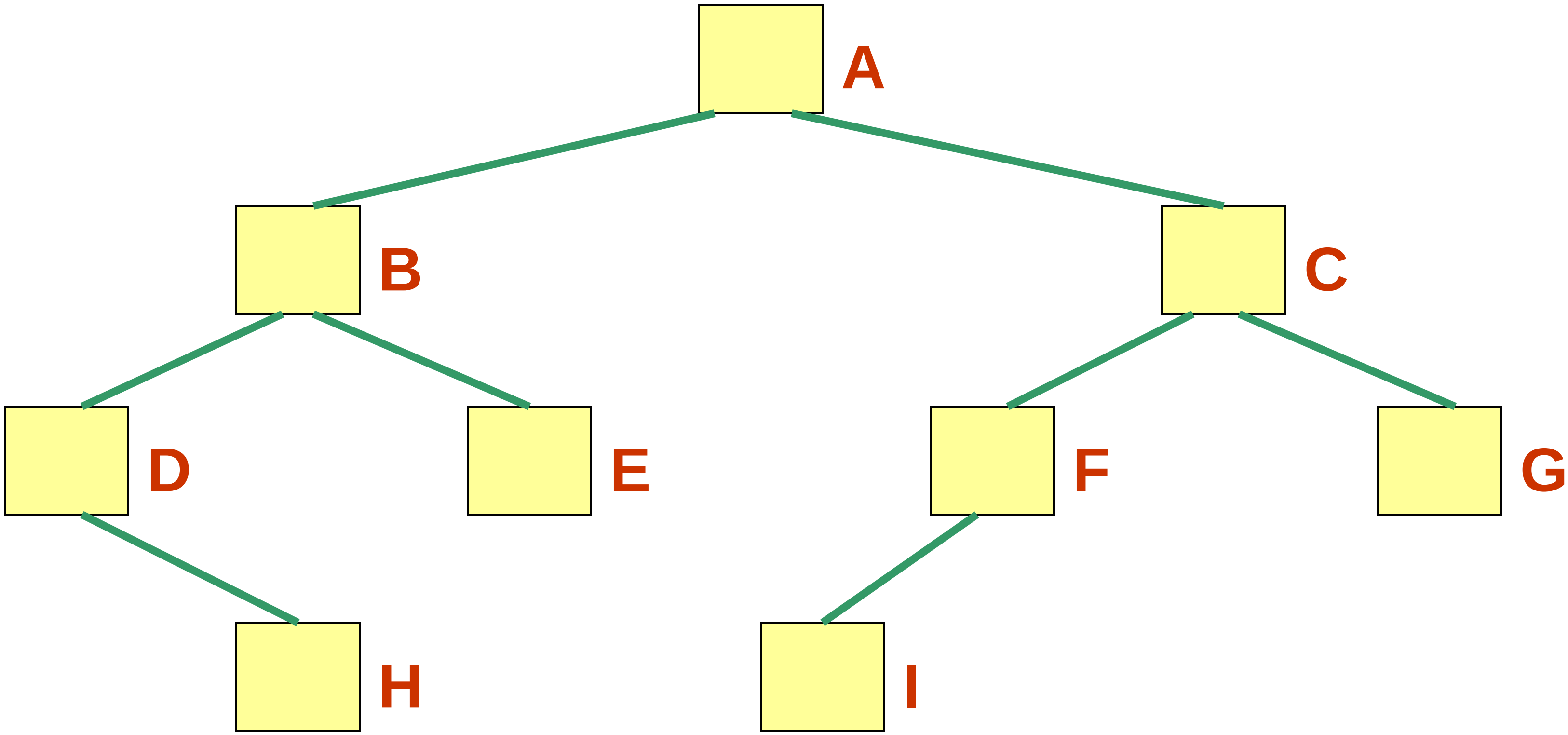
- Note that the ***relative order*** of the recursive calls in ***preorder***, ***inorder*** and ***postorder*** traversals is the same
- The only differences stem from ***where*** the visiting of the root node of a subtree actually takes place

Level Order Traversal

- Start at the root
- Visit the nodes at each level, from left to right

- Is there a recursive algorithm for a level order traversal?

Level Order Traversal



Nodes will be visited in the order ABCDEFGHI

Iterative Binary Tree Traversals

- In recursive tree traversals, the ***Java call stack*** keeps track of where we are in the tree (by means of the call frames for each call)
- In ***iterative traversals***, the programmer needs to keep track!
  - An iterative traversal uses a ***container*** to store references to nodes not yet visited
  - Order of visiting will depend on the type of container being used (**stack**, **queue**, etc.)

An Iterative Traversal Algorithm

```
// Assumption: the tree is not empty

Create an empty container to hold references to nodes
      yet to be  visited.

Put reference to the root node in the container.

While the container is not empty {

   Remove a reference x from the container.

   Visit the node x points to.

   Put references to non-empty children of x in the container.

   }

}
```

Iterative Binary Tree Traversals

- Container is a **stack**: if we push the *right* successor of a node before the *left* successor, we get ***preorder*** traversal
- Container is a **queue**: if we enqueue the *left* successor before the *right*, we get a ***level order traversal***
- ***Exercise***: Trace the iterative tree traversal algorithm using as containers
    - a stack
    - a queue

Traversal Analysis

- Consider a binary tree with n nodes
- How many recursive calls are there at most?
  - For each node, 2 recursive calls at most
  - So, 2*n recursive calls at most
- So, a traversal is **O(n)**

Operations on a Binary Tree

- What might we want to do with a binary tree?
  - Add an element (but where?)
  - Remove an element (but from where?)
  - Is the tree empty?
  - Get size of the tree (i.e. how many elements)
  - Traverse the tree (in preorder, inorder, postorder, level order)

Discussion

- It is difficult to have a general ***add*** operation, until we know the purpose of the tree (we will discuss binary search trees later)
  - We could add "***randomly***": go either right or left, and add at the first available spot

Discussion

- Similarly, where would a general ***remove*** operation remove from?
  - We could arbitrarily choose to remove, say, the leftmost leaf
  - If random choice, what would happen to the children and descendants of the element that was removed? What does the parent of the removed element now point to?
  - What if the removed element is the root?

Possible Binary Tree Operations

Operation	Description
removeLeftSubtree	Removes the left subtree of the root
removeRightSubtree	Removes the right subtree of the root
removeAllElements	Removes all elements from the tree
isEmpty	Determines whether the tree is empty
size	Determines the number of elements in the tree
contains	Determines if a particular element is in the tree
find	Returns a reference to the specified target, if found
toString	Returns a string representation of tree’s contents
iteratorInOrder	Returns an iterator for an inorder traversal
iteratorPreOrder	Returns an iterator for a preorder traversal
iteratorPostOrder	Returns an iterator for a postorder traversal
iteratorLevelOrder	Returns an iterator for a levelorder traversal

Binary Tree Operations

- Our textbook has a smaller set of operations for the BinaryTreeADT
  - See BinaryTreeADT.java

UML Description of the **BinaryTreeADT** interface

<<interface>>

BinaryTreeADT

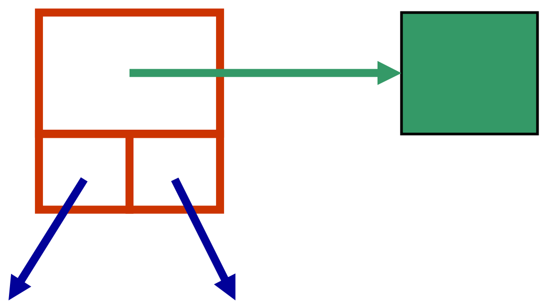
getRoot()
isEmpty( )
size( )
contains( )
find( )
toString()
iteratorInOrder( )
iteratorPreOrder( )
iteratorPostOrder( )
iteratorLevelOrder( )

Linked Binary Tree Implementation

- To represent the binary tree, we will use a linked structure of nodes
  - **root**: reference to the node that is the root of the tree
  - **count**: keeps track of the number of nodes in the tree
- First, how will we represent a ***node of a binary tree***?

Binary Tree Node

- A binary tree node will contain
  - a reference to a data element
  - references to its left and right children



left and right children are binary tree nodes themselves

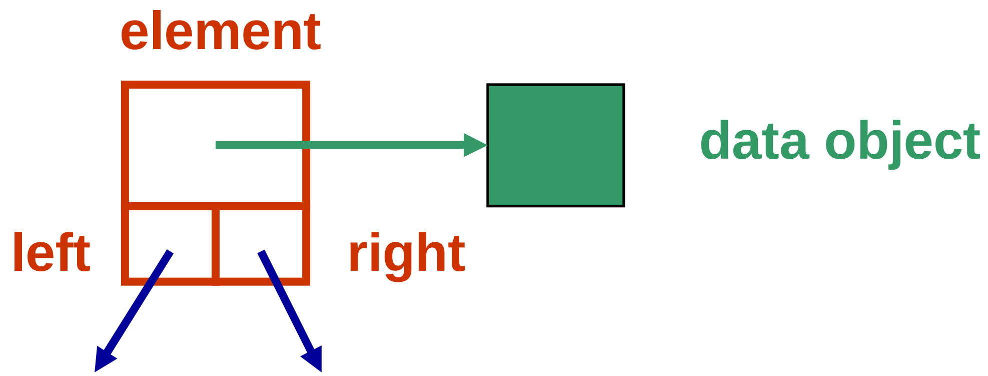
BinaryTreeNode class

- Represents a node in a binary tree
- Attributes:
  - element: reference to data element
  - left: reference to left child of the node
  - right: reference to right child of the node

- See BinaryTreeNode.java
  - Note that the attributes here are protected
    - This means that they can be accessed directly from any class that is in the same package as BinaryTreeNode.java

A BinaryTreeNode Object

```
protected T element;
protected BinaryTreeNode<T> left, right;
```



Note that either or both of the left and right references could be null

LinkedBinaryTree Class

- Attributes:

  protected BinaryTreeNode<T> root;
  protected int count;

- The attributes are protected so that they can be accessed directly in any subclass of the LinkedBinaryTree class
  - We will be looking at a very useful kind of binary tree called a Binary Search Tree later

LinkedBinaryTree Class

- Constructors:

```
//Creates empty binary tree
public LinkedBinaryTree() {
    count = 0;
    root = null;
}
//Creates binary tree with specified element as its root
public LinkedBinaryTree (T element) {
    count = 1;
    root = new BinaryTreeNode<T> (element);
 }
```

```
/* Returns a reference to the specified target element if it is
   found in this binary tree.
   Throws an ElementNotFoundException if not found. */

  public T find(T targetElement) throws
                                    ElementNotFoundException
  {
    BinaryTreeNode<T> current =
                             findAgain( targetElement, root );
    if ( current == null )
      throw new ElementNotFoundException("binary tree");

    return (current.element);
  }
```

find method

Discussion

- What is element in this statement from the method?

      return (current.element);

  - If element were private rather than protected in BinaryTreeNode.java, what would be need in order to access it?
- We will now look at the helper method findAgain …

```
private BinaryTreeNode<T> findAgain(T targetElement,
                                    BinaryTreeNode<T> next)
{
    if (next == null)
        return null;
    if (next.element.equals(targetElement))
        return next;
    BinaryTreeNode<T> temp =
                    findAgain(targetElement, next.left);
    if (temp == null)
        temp = findAgain(targetElement, next.right);
    return temp;
}
```

findAgain helper method

Discussion

- What kind of method is findAgain?
  - What is the base case?
    - There are two!
  - What is the recursive part?

```java
/* Performs an inorder traversal on this binary tree by
   calling a recursive inorder method that starts with
   the root.
   Returns  an inorder iterator over this binary tree */
  public Iterator<T> iteratorInOrder()
  {
    ArrayUnorderedList<T> tempList =
                             new ArrayUnorderedList<T>();

    inorder (root, tempList);
    return tempList.iterator();
  }
```

iteratorInOrder method

Discussion

- iteratorInOrder is returning an iterator object
  - It will perform the iteration in ***inorder***
- But where is that iterator coming from?

```
return tempList.iterator();
```

- Let’s now look at the helper method inorder …

```
/* Performs a recursive inorder traversal.
   Parameters are: the node to be used as the root
   for this traversal, the temporary list for use in this
   traversal  */
  protected void inorder (BinaryTreeNode<T> node,
                          ArrayUnorderedList<T> tempList)
  {
    if (node != null)
    {
      inorder (node.left, tempList);
      tempList.addToRear(node.element);
      inorder (node.right, tempList);
    }
  }
```

inorder helper method

Discussion

- Recall the ***recursive algorithm*** for inorder traversal:
  - If tree is not empty,
    - Perform inorder traversal of left subtree of root
    - Visit root node of tree
    - Perform inorder traversal of its right subtree
- That’s exactly the order that is being implemented here!
    - What is “visiting” the root node here?

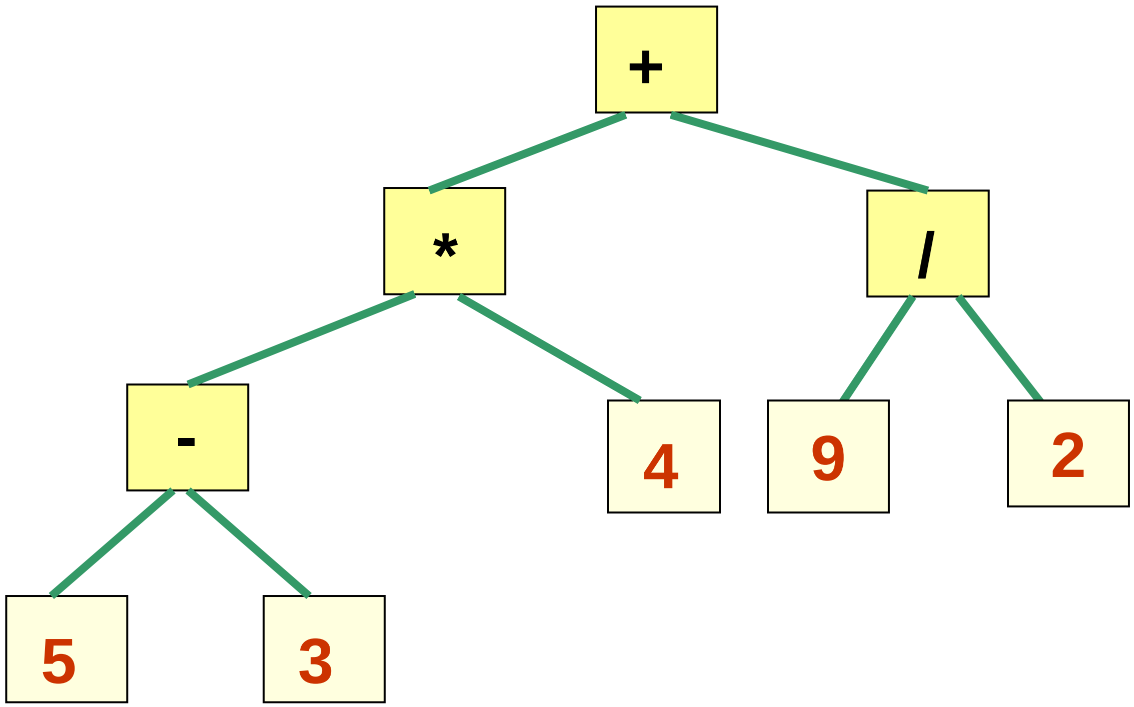
Discussion

- The data elements of the tree (i.e. items of type T) are being *temporarily* added to an unordered list, in *inorder* order
  - Why use an unordered list??
    - Why not? We already have this collection, with its iterator operation that we can use!

Using Binary Trees: Expression Trees

- Programs that manipulate or evaluate arithmetic expressions can use binary trees to hold the expressions
- An ***expression tree*** represents an arithmetic expression such as
  (5 – 3) * 4 + 9 / 2
  - Root node and interior nodes contain ***operations***
  - Leaf nodes contain ***operands***

Example: An Expression Tree

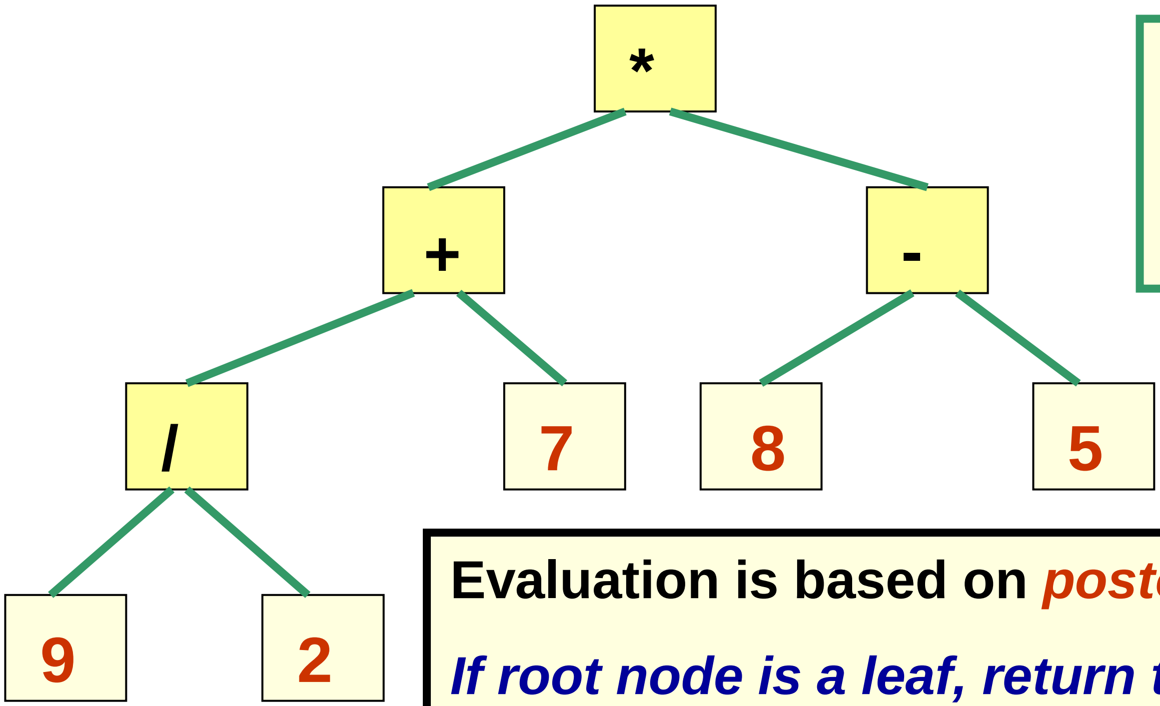


(5 – 3) * 4 + 9 / 2

Evaluating Expression Trees

- We can use an expression tree to ***evaluate an expression***
  - We start the evaluation at the *bottom left*
  - What kind of traversal is this?

Evaluating an Expression Tree



This tree represents the expression

(9 / 2 + 7) * (8 – 5)

Evaluation is based on *postorder* traversal:

If root node is a leaf, return the associated value.

Recursively evaluate expression in left subtree.

Recursively evaluate expression in right subtree.

Perform operation in root node on these two values, and return result.

Building an Expression Tree

- Now we know how to evaluate an expression represented by an expression tree
- But, how do we *build* an expression tree?
  - We will build it from the postfix form of the expression
- ***Exercise***: develop the algorithm by following the diagrams on the next pages

Building an Expression Tree

- The algorithm will use a stack of ExpressionTree objects
  - An ExpressionTree is a special case of a binary tree
    - The ExpressionTree constructor has 3 parameters:
      - Reference to data item
      - Reference to left child
      - Reference to right child
- *That's all you need to know to develop the algorithm!*

Build an expression tree from the postfix expression 5 3 - 4 * 9 +

Token 5

Processing Step(s)

```
push(new ExpressionTree(5,null,null));
```

Expression Tree Stack (top at right)

5

Token 3

Processing Step(s)

```
push(new ExpressionTree(3,null,null));
```

Expression Tree Stack (top at right)

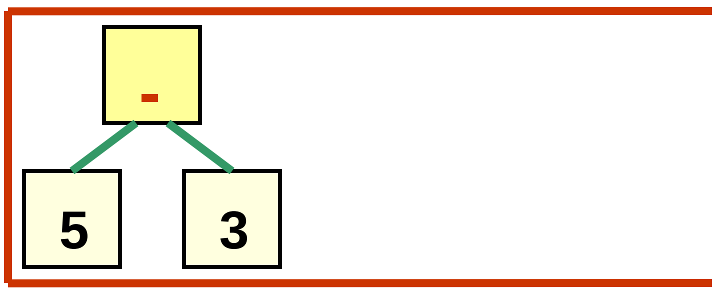
5	3

Token **Processing Step(s)**

-

```
op2 = pop
op1 = pop
push(new ExpressionTree(-,op1,op2));
```

Expression Tree Stack (top at right)



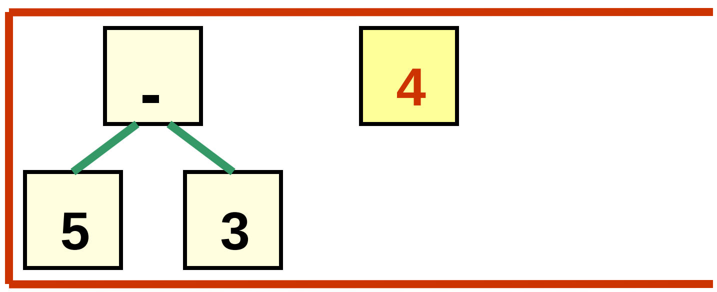
Token

4

Processing Step(s)

```
push(new ExpressionTree(4,null,null));
```

Expression Tree Stack (top at right)



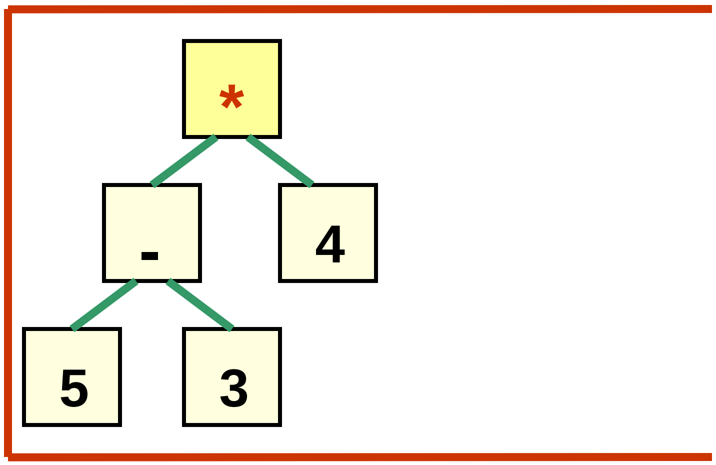
Token

*

Processing Step(s)

```
op2 = pop
op1 = pop
push(new ExpressionTree(*,op1,op2));
```

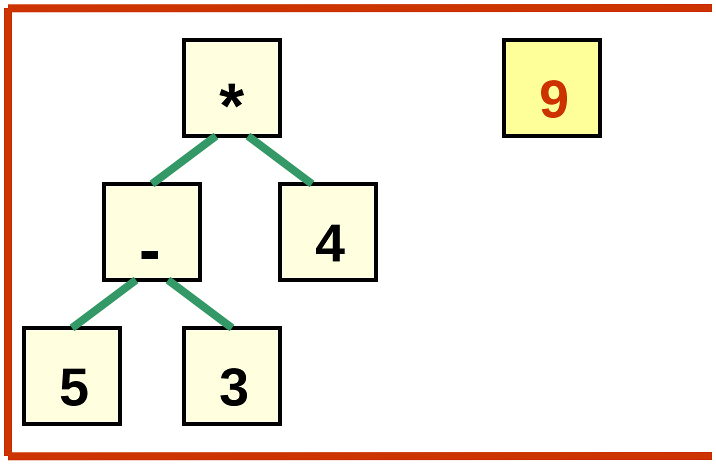
Expression Tree Stack (top at right)



Token **Processing Step(s)**


```
push(new ExpressionTree(9,null,null));
```

Expression Tree Stack (top at right)



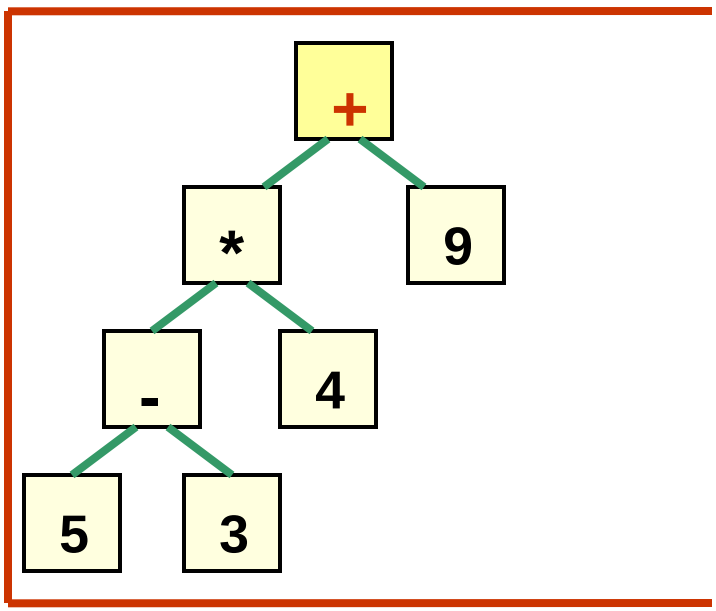
Token

+

Processing Step(s)

```
op2 = pop
op1 = pop
push(new ExpressionTree(+,op1,op2));
```

Expression Tree Stack (top at right)



End of the expression has been reached, and the full expression tree is the only tree left on the stack