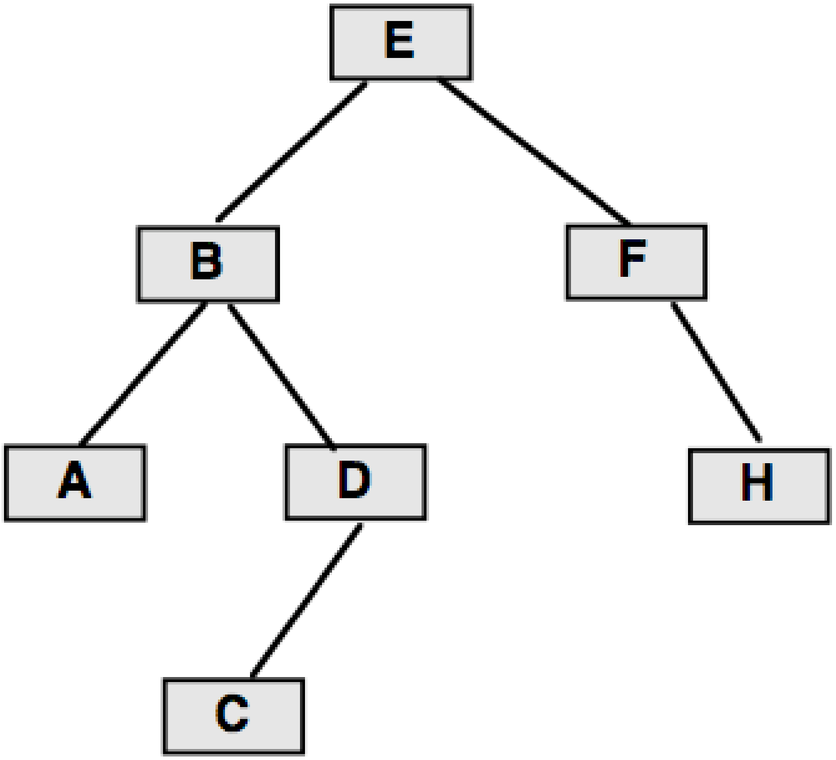
Tree Algorithms

Tree Traversals

- An important class of algorithms is to traverse an entire data structure – visit every element in some fixed order
- For trees there are two types of traversals, each with their variations
  - » Breadth first traversal
    - > Level by level
      - – **Left to right across a level, or, right to left across a level**
  - » Depth first traversal
    - > Go as deep as possible before going along a level
      - – **preorder, inorder, postorder – each going clockwise or anti-clockwise around the tree**

Breadth First Traversal

- Visit and process the nodes in one of the following orders
  - » E B F A D H C or E F B H D A C



Breadth First Traversal – 2

- Queue saves pointers to tree nodes for later processing

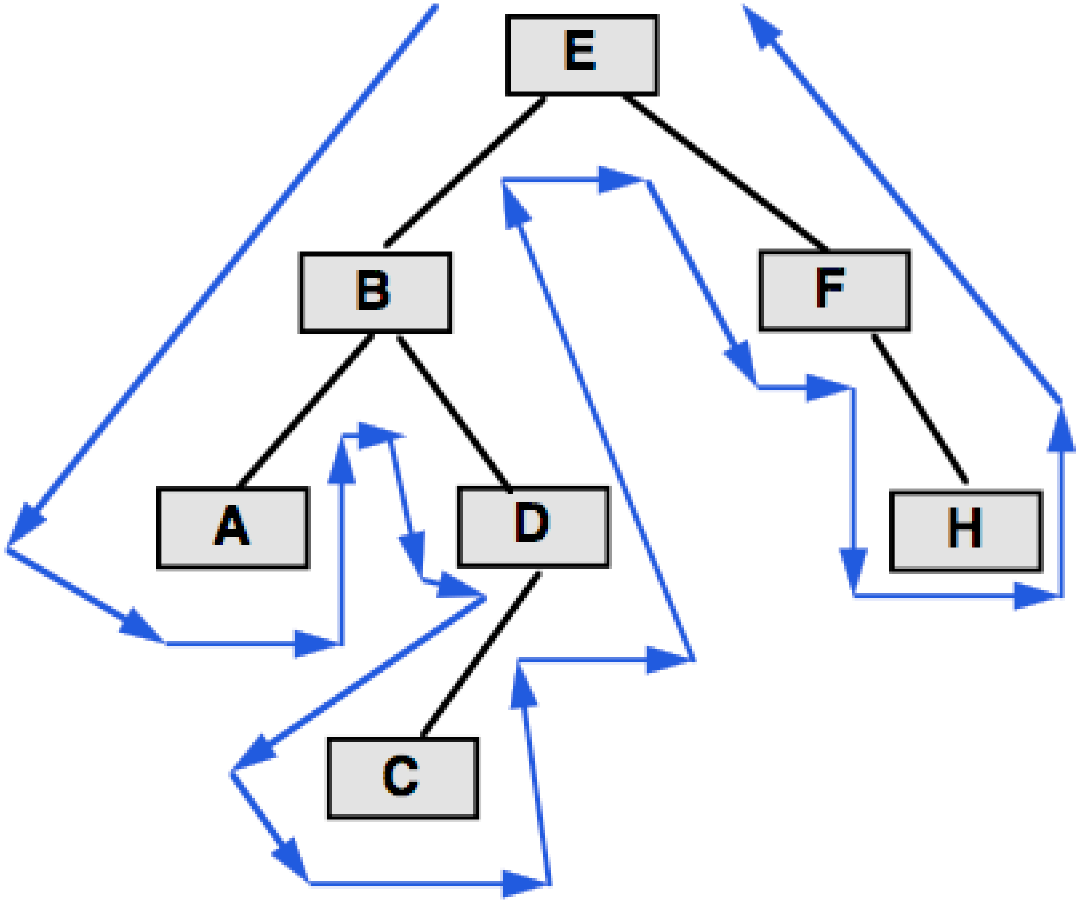
```
    Require $root \neq void$
    Ensure $\forall node \in Tree \cdot processed(node)$

Q ← new Queue
Q.put ( root )
Loop Invariant
    $\forall node \in \{n \in tree(root)\} \setminus \{n1 \in Q \cdot \forall n2 \in tree(n1) \cdot n2\} \cdot processed(Node)$
while ~ empty ( Q ) do
    node ← Q.take                // Put children in Queue
    if node.left ≠ void then Q.put ( node.left ) fi
    if node.right ≠ void then Q.put ( node.right ) fi
    process ( node )             // Visit the node
end
```

An example of the Template Pattern

Breadth First Traversal – 3

- Exercises
  - **Apply the algorithm to the example in the slide *Breadth First Traversal***
  - **What changes are required in the algorithm to change the order of processing nodes within a level?**
  - **What changes are required in the algorithm to handle a general tree?**

Depth First Traversal



Preorder – process on the way down
E B A D C F H

Inorder – process while going underneath
A B C D E F H

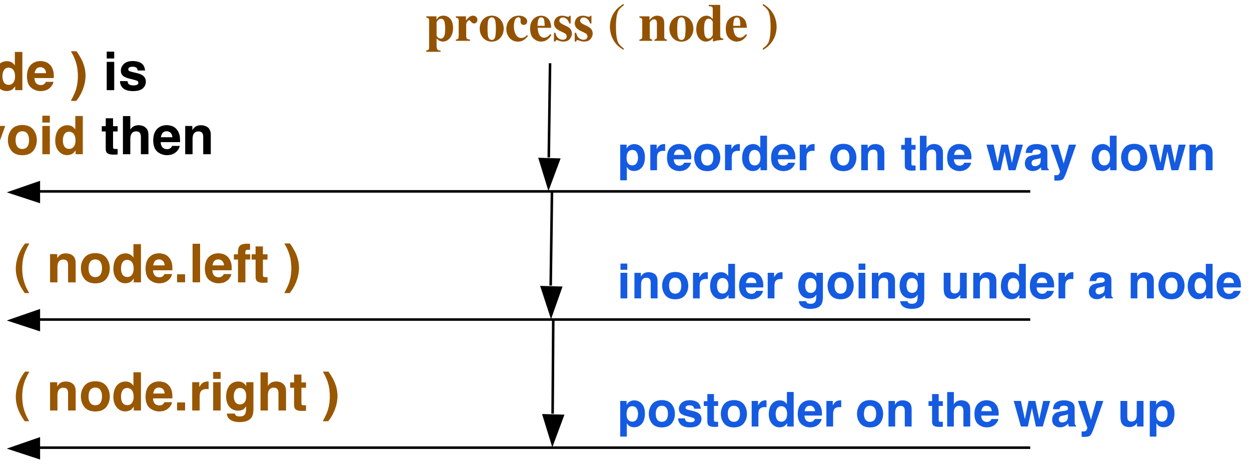
Postorder – process on the way up
A C D B H F E

Have another 3 orderings by reversing the arrows

Depth First Traversal – 2

- Depth first traversal uses a stack to save pointers to nodes for later processing
- Recursion uses a stack, so a recursive algorithm is a natural for depth first traversal

An example of the Template Pattern



process (node)

```
traverse ( node ) is
  if node ≠ void then

    traverse ( node.left )

    traverse ( node.right )

  fi
end
```

preorder on the way down

inorder going under a node

postorder on the way up

Depth First Traversal – 3

- Exercises
  - **Apply the algorithm to the example in the slide *Breadth First Traversal***
  - **What changes are required in the algorithm to reverse the order of processing nodes for each of preorder, inorder and postorder?**
  - **What changes are required in the algorithm to handle a general tree?**

Node Depth General Case

- O(N) algorithm, where N is the number of nodes in the Tree
  - **O(D_{node}), where D_{node} is the depth of the node**
  - **Note the assumption that general tree nodes have a pointer to the parent**
    - **Depth is undefined for empty tree**

```
Require tree ≠ Void ∧ node ∈ tree
Ensure  Result = pathLength ( node, tree )

depth ( node, tree ) : Integer is
    if node = tree.root then Result ← 0
    else Result ← 1 + depth (node.parent, tree )
    fi
end
```

Node Depth Binary Tree

- Permit node = Void on recursion to simplify algorithm

```
Require client tree ≠ Void ∧ node ∈ tree
Ensure  Result = pathLength ( node, tree )
   integer depth2 ( node, tree ) is depth_sup ( node, tree, 0 ) end

Require True
Ensure     ( node ∉ tree ∧ Result = – 1 )
       ∨  ( node ∈ tree ∧ Result = pathLength ( node, tree ) )

depth_sup ( node, tree, depth ) : Integer is
      if node = Void then Result ← –1
      elseif node = tree.root then Result ← depth
      else Result ← max ( depth_sup ( node, tree.left, depth+1 )
                         , depth_sup ( node, tree.right, depth+1)
      fi
   end
```

Tree Height General Case

- An O(N) algorithm, N is the number of nodes in the tree

```
Require node ≠ Void          Height is undefined for empty tree
Ensure ~hasChildren ( node ) → Result = 0
           hasChildren ( node ) →
              Result = 1 + max / ⟨ c : children ( node ) • height  ( c ) ⟩

height1 ( node ) : Integer is
   if ~hasChildren ( node ) then Result ← 0
   else children ← childrenOf ( node )
         height ← 0
         for child in children do  height ← max ( height
                                                 , height1 (child) )
         end
         Result ← 1 + height
   fi
end
```

Binary_op / sequence
reduce the sequence
using the operator

Tree Height General Case – 2

- An O(N^2) algorithm, N is the number of nodes in the tree – from page 274 of the textbook
    - » **Why is this O(N^2)**

```
height_tb ( Tree ) : Integeris
    height ← 0
    for node in externalNodes(T) do
        height ← max ( height, depth (Tree, node) )
    end
    Result ← height
end
```

Tree Height Binary Tree

```
height2 ( node ) : Integer is
   if node.left = Void then
      if node.right = Void then Result ← 0
                          else Result ← 1 + height2 ( node.right )
     fi
   else
      if node.right = Void then Result ← 1 + height2 ( node.left )
      else Result ← 1 + max ( height2 ( node.left )
                                 , height2 ( node.right ) )
      fi
   fi
end
```

Tree Height Binary Tree – 2

- Simplify algorithm by defining height of empty tree as –1
  - » **Use mathematical properties of integers and arithmetic**

```
Require client node ≠ Void
        recursion  True

height3 ( node ) : Integer is
    if node = Void then Result ← –1
    else Result ← 1 + max ( height3 ( node.left )
                          , height3 ( node.right ) )
    fi
end
```

Tree Height Binary Tree – 3

- Lesson from previous slide – do not treat tree empty tree as special case
- Special cases complicate algorithms

Binary_op / sequence
reduce the sequence
using the operator

Can call for empty tree

```
Require True
Ensure  Result = 1 + max / ⟨ c : children ( node )
                             • height  (node) ⟩
```

```
height4 ( node ) : Integer is
    if node = Void then Result ← 0        Empty tree has 0 height
    else Result ← 1 + max ( height4 ( node.left )
                          , height4 ( node.right ) )
    fi
end
```

Inorder Traversal Binary Tree

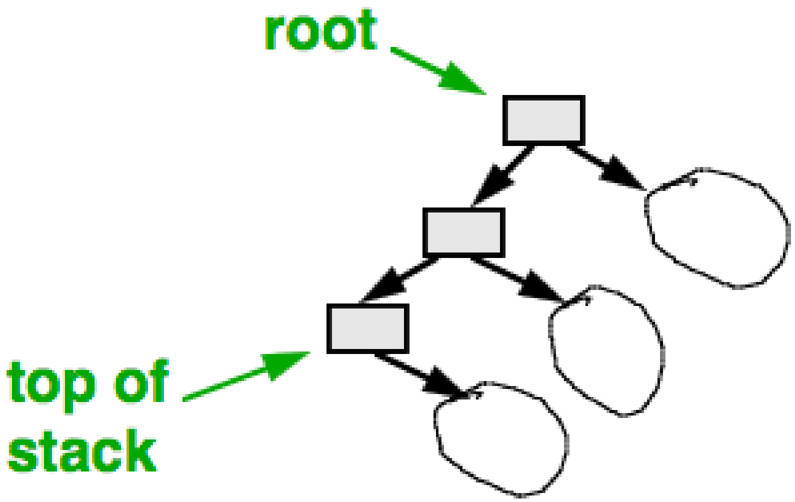
- Binary tree has 8 different traversal orders
  - » 6 for depth first plus 2 for breadth first
    - > Template comes from slides on Enumeration

Require True
Ensure Nodes returned in inorder sequence

```
public Enumeration elements () {
   return new Enumeration() {
      public boolean hasMoreElements() { Provide the definition – 1 }
      public Object nextElement()   { Provide the definition – 2  }
      Declare variables needed by the enumerator – 3
       { Initialization (constructor) program for the enumerator – 4
     }}
```

Inorder Traversal Binary Tree – 2

```
// Declare variables needed by the enumerator – 3

private Stack btStack = new Stack();

{ Initialization (constructor) program for the enumerator – 4
  // Simulate recursion by programming our own stack.  Need to get to
  // the leftmost node as it is the first in the enumeration.

   Node node = tree;
   while node != null) {
      btStack.add ( node );
      node = node .left;
   }
```



Inorder Traversal Binary Tree – 3

```
// Provide definition – 2

Require  True
Ensure    Result = another element to get

public boolean hasMoreElements()  {
    return !btStack.isEmpty();
}
```

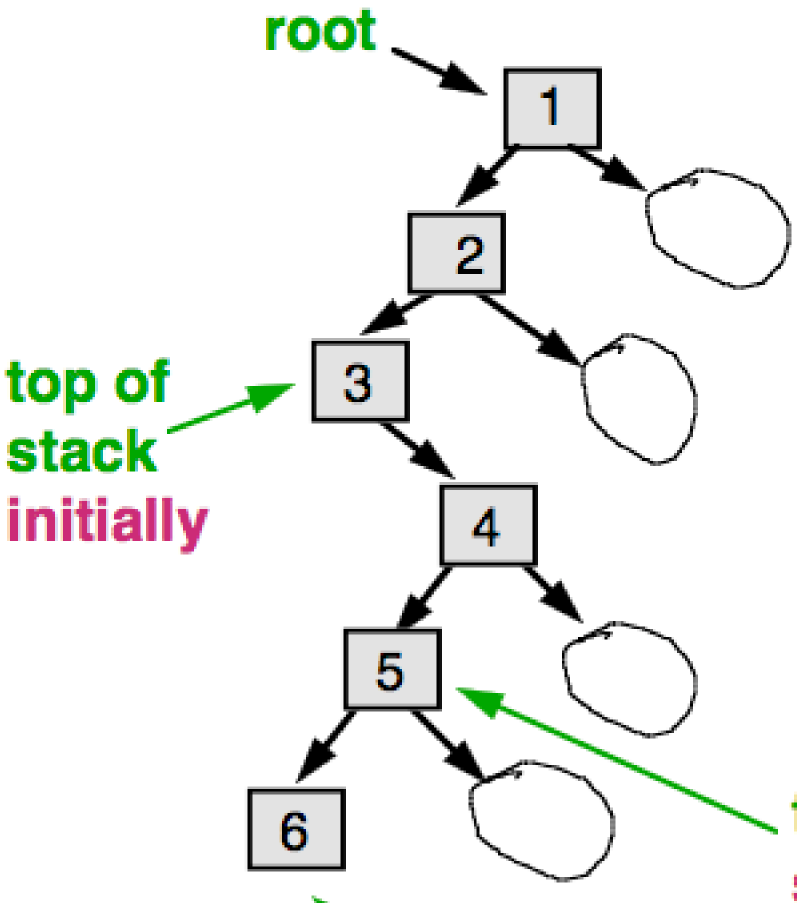
Inorder Traversal Binary Tree – 4

// Provide definition – 3

Require hasMoreElements
Ensure Result = next element in sequence and it is removed from the sequence

```
public Object nextElement()  {
    Node node = (Node) btStack.remove();
    Object result = node.datum       // next item to return
    if (node.right != null) {        // Find next node in sequence
        node = node.right;
        do { btStack.add(node);      // Get leftmost node in right
            node = node.left;        // subtree
        } while (node != null);
    }
    return result;
}
```

Inorder Traversal Binary Tree – 5



top of stack initially

top of stack after second call to nextElement

top of stack after first call to nextElement

InitStack	after call 1	after call 2
3	6	5
2	5	4
1	4	2
	2	1
	1	

An enumerator is always one element ahead of the user

Class Structure

- ADT definitions can found in textbook and in the FlexOr Library