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
  - E F B H D A C
Breadth First Traversal – 2

• Queue saves pointers to tree nodes for later processing

Require $\text{root} \neq \text{void}$
Ensure $\forall \text{node} \in \text{Tree} \cdot \text{processed ( node )}$

$Q \leftarrow \text{new Queue}$
$Q.\text{put ( root )}$

Loop Invariant

$\forall \text{node} \in \{n \in \text{tree(root) } \}\setminus$
$\{n1 \in Q \cdot \forall n2 \in \text{tree(n1) } \cdot n2 \}$

• $\text{processed(Node)}$

while $\neg$ empty ( $Q$ ) do

node $\leftarrow Q.\text{take}$ \hspace{1em} // Put children in Queue

if node.left $\neq$ void then $Q.\text{put ( node.left )}$ fi
if node.right $\neq$ void then $Q.\text{put ( node.right )}$ fi

process ( node ) \hspace{1em} // Visit the node

end

An example of the Template Pattern
• 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.

```plaintext
traverse ( node ) is
  if node ≠ void then
    process ( node )
    traverse ( node.left )
    traverse ( node.right )
  fi
end
```

An example of the Template Pattern:

- preorder on the way down
- inorder going under a node
- postorder on the way up
• 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?
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 \( \text{tree} \neq \text{Void} \land \text{node} \in \text{tree} \)
Ensure \( \text{Result} = \text{pathLength} \left( \text{node}, \text{tree} \right) \)

\[
\text{depth} \left( \text{node}, \text{tree} \right) : \text{Integer is}
\]

if \( \text{node} = \text{tree.root} \) then Result \( \leftarrow 0 \)
else Result \( \leftarrow 1 + \text{depth} \left( \text{node.parent}, \text{tree} \right) \)
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
    elsif 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 $\neq$ Void \hspace{1cm} \textit{Height is undefined for empty tree}
Ensure $\neg$hasChildren ( node ) $\rightarrow$ Result = 0
hasChildren ( node ) $\rightarrow$
\hspace{1cm} Result = $1 + \max / \langle c : \text{children} ( \text{node} ) \cdot \text{height} (c) \rangle$

height1 ( node ) : Integer is
\hspace{1cm} if $\neg$hasChildren ( node ) then Result $\leftarrow$ 0
\hspace{1cm} else children $\leftarrow$ childrenOf ( node )
\hspace{1cm} height $\leftarrow$ 0
\hspace{1cm} for child in children do height $\leftarrow$ max ( height, height1 (child) )
\hspace{1cm} end
\hspace{1cm} Result $\leftarrow$ 1 + height
\hspace{1cm} fi
\hspace{1cm} end
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)

```plaintext
height_tb ( Tree ) : Integer is
    height ← 0
    for node in externalNodes(T) do
        height ← max ( height, depth (Tree, node) )
    end
    Result ← height
end
```
height2 ( node ) : Integer is
  if node.left = Void then
    if node.right = Void then Result ← 0
      end
    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 )
    fi
  fi
end
• 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
Lesson from previous slide – do not treat tree empty tree as special case

Special cases complicate algorithms

Require True Can call for empty tree
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

```java
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
    }
  }
}
```

© Gunnar Gotshalks
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;
  }
// 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) {
        node = node.right;
        do {
            btStack.add(node);  // Get leftmost node in right
            node = node.left;  // subtree
        } while (node != null);
    }
    return result;
}
An enumerator is always one element ahead of the user
• ADT definitions can be found in the textbook and in the FlexOr Library.