COSC160: Data Structures
Trees

Jeremy Bolton, PhD
Assistant Teaching Professor

GEORGETOWN UNIVERSITY

Outline

- I. Trees
  - I. Terminology
  - II. Traversals
    - I. DFS
      - I. InOrder
      - II. PreOrder
      - III. PostOrder
    - II. BFS
  - III. Pseudocode Examples

Trees

- A tree is a fundamental discrete structure used in computer science.
  - In computer science we are generally interested in *rooted* trees. Henceforth when we will refer to all rooted trees simply as trees.
  - Uses: compiler design, file hierarchy representation, data organization, …

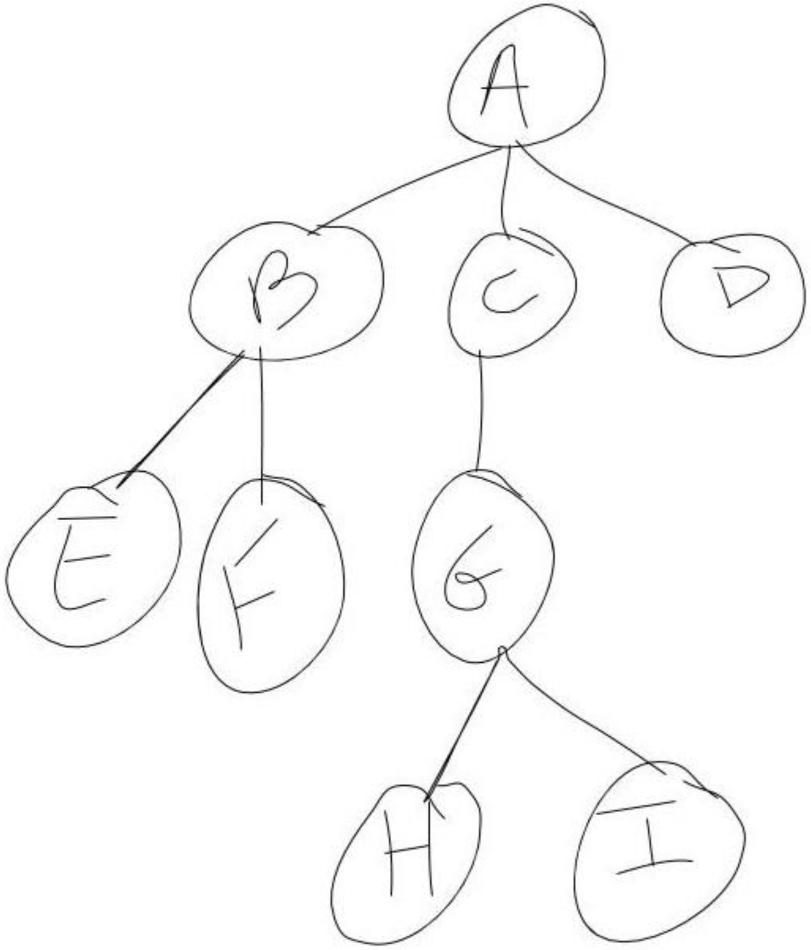
Trees

- Definitions:
  - A node is an entity, visually displayed as a circle, that may have a name and attributes.
  - A directed edge is a 2-tuple, e.g. edge1 = (node1,node2), that is said to connect two nodes. The edge is directed and so, node1 is connected to node2 but the reverse may not be so.
  - A path of length n between node i and node j is a sequence of edges where the first element of edge 1 is node i, the second element of edge n is node j, all neighboring edges in the sequence share the same connecting node.
    - EG Path from node 1 to node 5 of length 3: (1,2) (2,4) (4,5)
  - A (rooted) tree is a set of nodes and a set of directed edges with the following properties
    - One node is designated as the root
    - There is no path from any node (that is not the root) to the root node.
    - There is a unique *path* between the root node and each node.

Trees: terminology

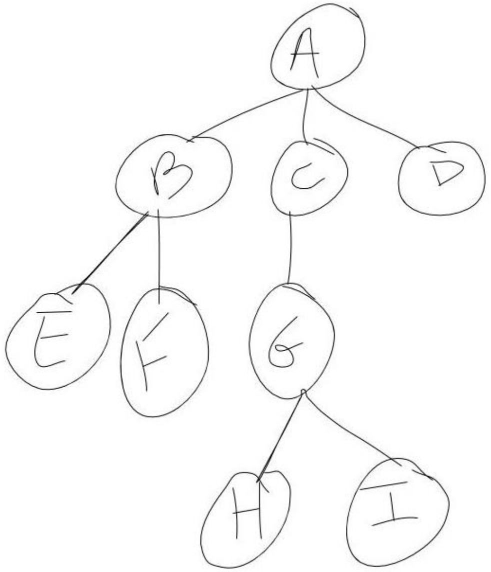
- A directed edge connects a parent node and a child node
  - EG, assume edge = (node A , node B). A is the parent of B and B is the child of A.
  - Observations:
    - The root node has no parent.
    - Nodes that have no children are called *leaves*.
  - Nodes that share a parent are called siblings
  - If there is a path from node i to node j, then i is an ancestor of j and j is a descendent of i.

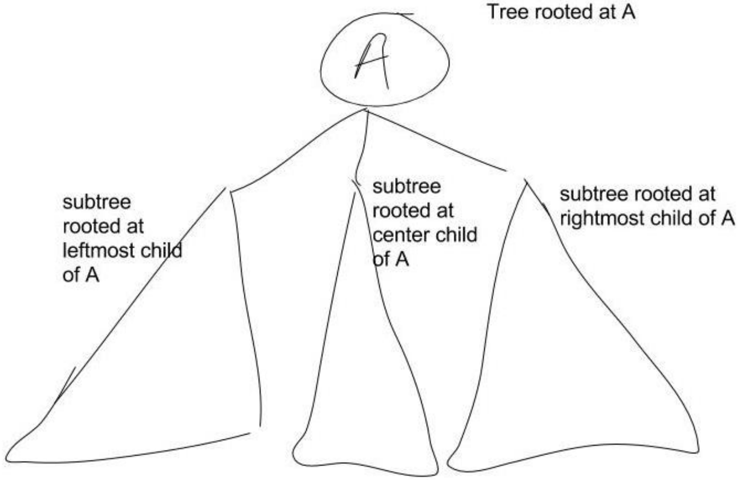
- The depth of a node j is the length of the path from the root to node j
  - The root node is at depth 0.
  - The depth of any node j is equal to the depth of the parent of node j + 1
  - The height of the tree is equal to the maximum depth over all nodes in the tree.



Recursive Definition of Trees

- The following are trees (shown as a set of nodes and set of edges)
  - The empty tree. No nodes, no edges
    - ({},{})
  - A (root) node
    - ({n_1},{})
  - If T_1 and T_2 are two trees, then T_3, formed by connecting any node from T_1 to the root node of T_2, is a tree.
    - $T_1 = (N_1, E_1)$
    - $T_2 = (N_2, E_2)$
    - $T_3 = (N_3, E_3)$,
      - where $N_3 = N_1 \cup N_2$
      - $E_3 = E_1 \cup E_2 \cup \{e\}$, where $e = (n_1, r_2)$ where $n_1 \in N_1$ and $\mathrm{r}_2 \in N_2$ and r_2 is the root of T_2
      - Observe: root of tree 1 is the root of tree 3



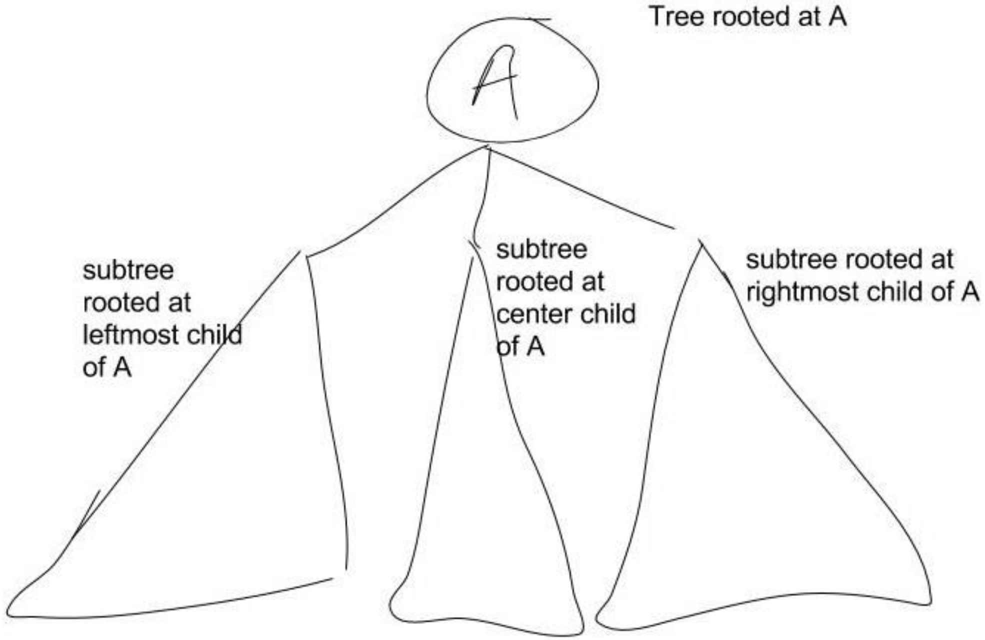


GEORGETOWN UNIVERSITY

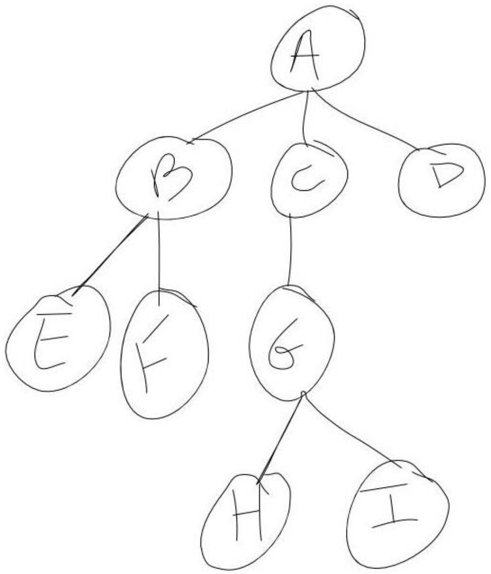
Subtrees

- A subtree st of a tree t is a tree whose set of nodes and edges are subsets of (n,e) = t.
  - $st = (N_1, E_1)$, where $t = (N, E\ )$, where $N_1 \subseteq N$ and $E_1 \subseteq E$ AND st is a tree.

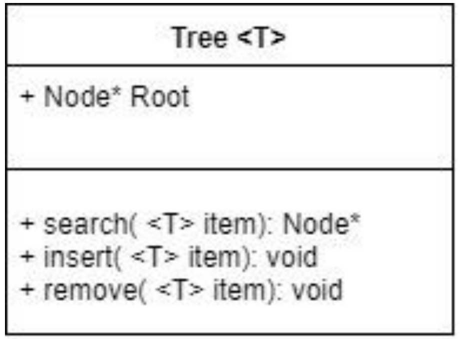
- Given the recursive definition of trees observe the following:
  - All nodes in a tree are a root of a subtree

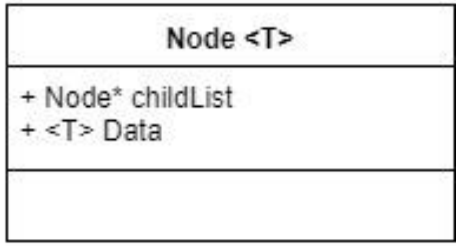


Tree Implementation (version 1)



- Designing a generic tree structure
  - Design concern: <u>the number of children</u> for any parent is unknown and may change dynamically (depending on usage).
  - Basic Design Option (list of children implementation):
    - Node class
      - Attributes include: data, <u>listOfChildren</u>
    - Tree class
      - Attributes include: rootNode
  - How can we design a list of children that is variable in size (between nodes) and may change in size?
    - Implement listOfChildren as Linked List

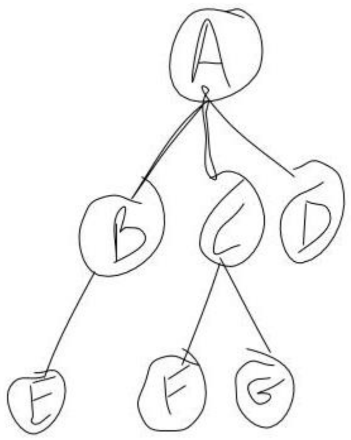




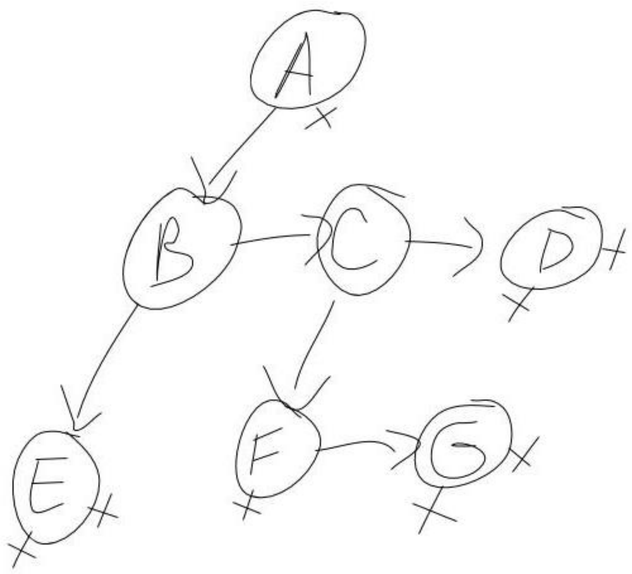
Tree Implementation (version 2)

- Rather than implementing a linked list of children, we can make the “number” of attributes for each node constant. “Alleviates” allocation concerns related to variable number of children per node.
  - left child / right sibling implementation
    - Instead of listOfChildren as a linked list, create a node class that has the following attributes
      - Node class
        - » Attributes include: data, leftMostChild, rightSibling

Actual Tree



Left Child / Right Sibling Representation

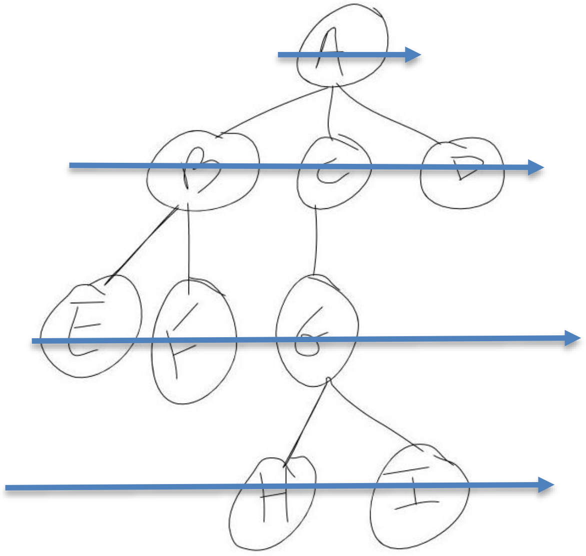


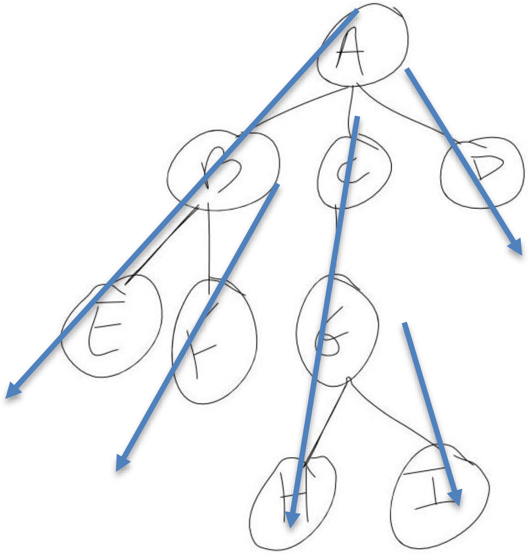
Node <T>
+ Node* leftChild + Node* rightSibling + <T> Data

Tree Traversals

- Traversing structures
  - Designing a traversal strategy is important for any data structure.
    - Visiting each element of a structure for processing, searching, …
    - EG Lists: sequential, divide and conquer traversal, …

- Traversing Trees (common approaches)
  - Breadth First Traversal (level-order traversal): traversing a tree node-by-node in order of depth.
    - Favors breadth
  - Depth First Traversal: traversing a tree node-by-node, starting at the root and proceeding as quickly as possibly to the leaves.
    - Favors depth
  - By standard, both traversal types traverse children in left to right order





Depth First Search (traversal)

- Pseudo code

Algorithm 1

Require: keyword 'this' is used to refer to calling Tree object

    function DFS(Tree this)

        $stack \leftarrow$ initialize new Stack

        $stack.push(this \rightarrow root)$

        while $\neg stack.isEmpty()$ **do**

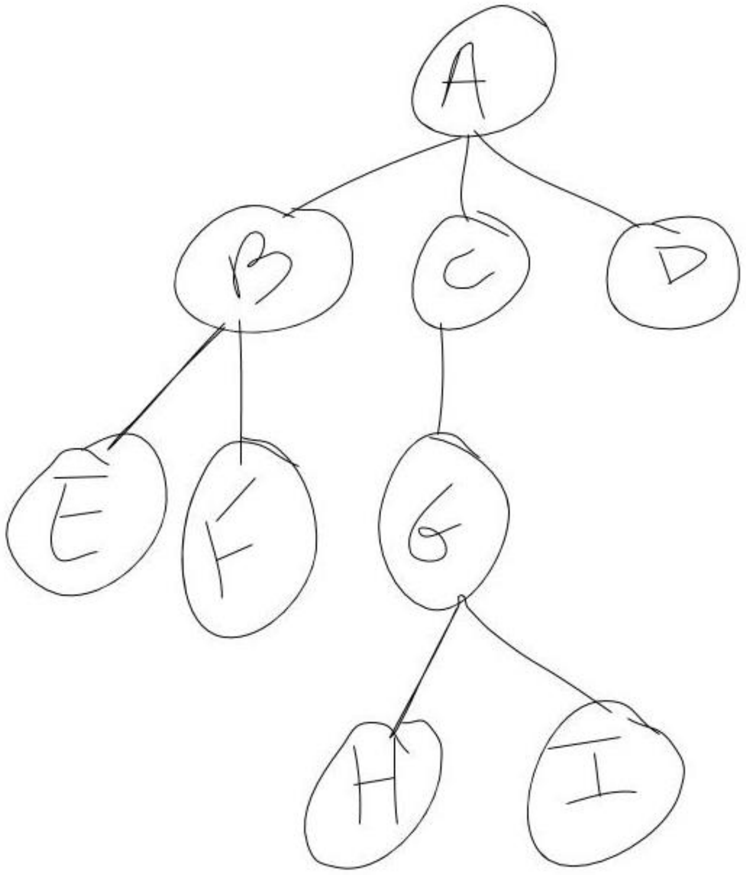
            $thisNode \leftarrow stack.pop()$

            // Process thisNode Here

            for $\forall c \in (thisNode \rightarrow childList)$ **do**

                $stack.push(c)$

- Try Example

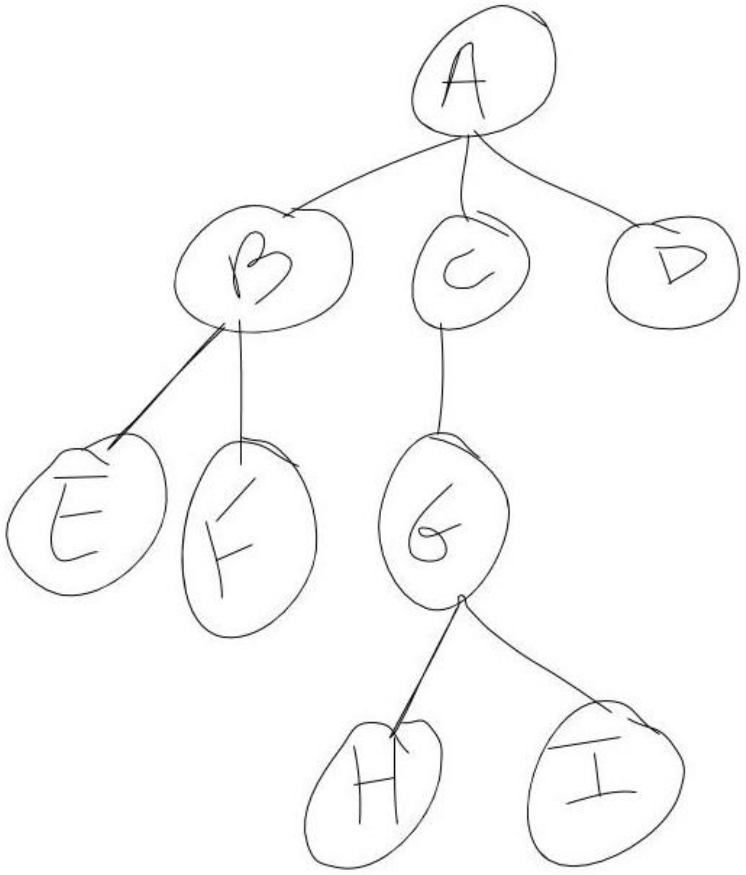


GEORGETOWN UNIVERSITY

DFS: recurrence perspective

- Given our recursive definition of a tree, it should come as no surprise that we can define operations on a tree (such as a traversal), recursively.

- Recurrence Idea
  - To perform a DFS on a tree with root r, perform a DFS on each subtree rooted by each child c.



DFS (recursive)

- Pseudo code
- Note: stack is implicit

Algorithm 1

Require: recursive function initially called with root as parameter

    function DFS(Node* thisNode)

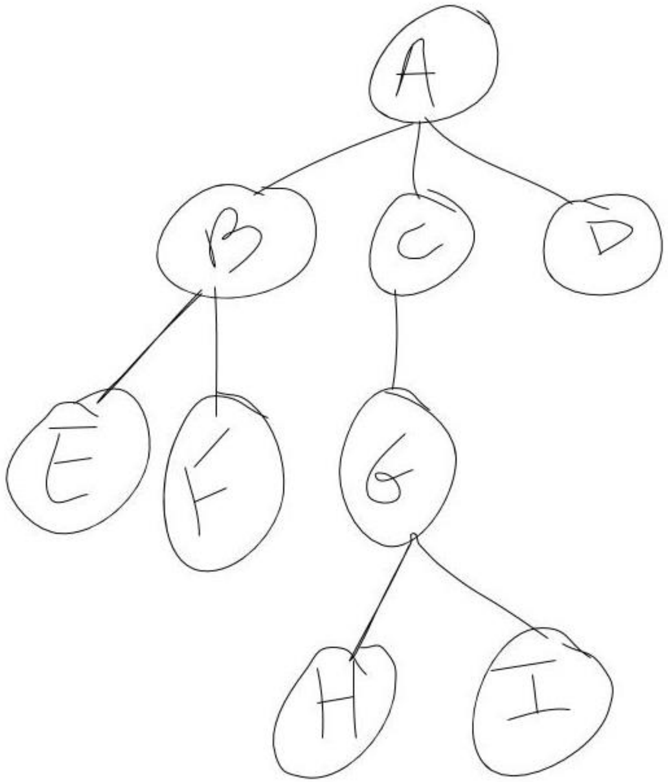
        // Process thisNode here for preorder traversal

        for $\forall c \in (thisNode \rightarrow childList)$ **do**

            $DFS(c)$

        // Process thisNode here for postorder traversal

- Try Example

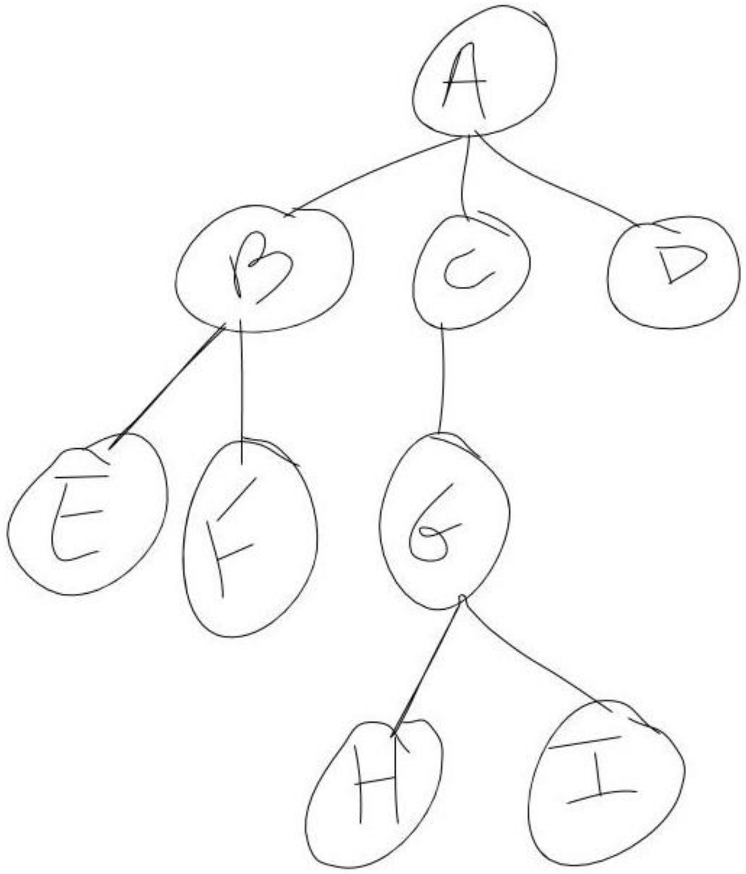


Breadth First Search (traversal)

- Pseudo code

Algorithm 1

Require: keyword this refers to calling Tree object
  function DFS(Tree this)
    $q \leftarrow$ initialize new fifoQueue
    $q.enqueue(this \rightarrow root)$
    while $\neg q.isEmpty()$ **do**
      $thisNode \leftarrow q.dequeue()$
      // Process thisNode here
      for $\forall c \in (thisNode \rightarrow childList)$ **do**
        $q.enqueue(c)$



- Try Example

Observations concerning DFS and BFS

- Observation: DFS lends itself to a simple recursive implementation. Why?
  - DFS makes use of a stack.
  - BFS makes use of a queue (FIFO).
  - A recursive, function call chain implicitly makes use of a stack (the runtime stack!)
    - Thus a recursive implementation is befitting (has a simple implementation)

DFS forms

- Common types of traversals
  - Prefix / Pre-order
    - A node is processed before processing its children
    - Example use: copy tree
  - Infix / In-order
    - A node is processed after j of its children are processed, where, in general, j is greater than 1 and less than the total number of children.
    - Example use: parser
  - Postfix / Post-order
    - A node is processed after all of its children are processed
    - Example use: deallocate tree

Pre-order

- Recursive Pseudo code

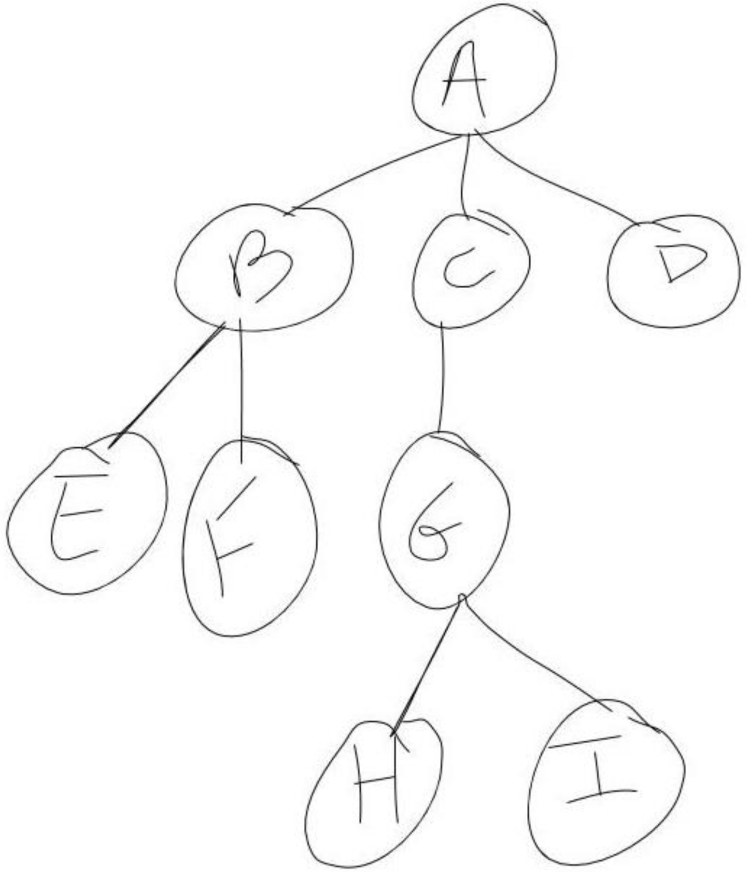
Algorithm 1

Require: recursive function initially called with root as parameter

```
function DFS(Node* thisNode)
    // Process thisNode here for preorder traversal
    for ∀c ∈ (thisNode → childList) do
        DFS(c)
    // Process thisNode here for postorder traversal
```

- Try Example:

  A-B-E-F-C-G-H-I-D



GEORGETOWN UNIVERSITY

In-order

- Pseudo code for inorder traversal. Process node after visiting left child

---

Algorithm 1

Require: recursive function initially called with root as parameter

  function DFS(Node* thisNode)

    // Process thisNode here for preorder traversal

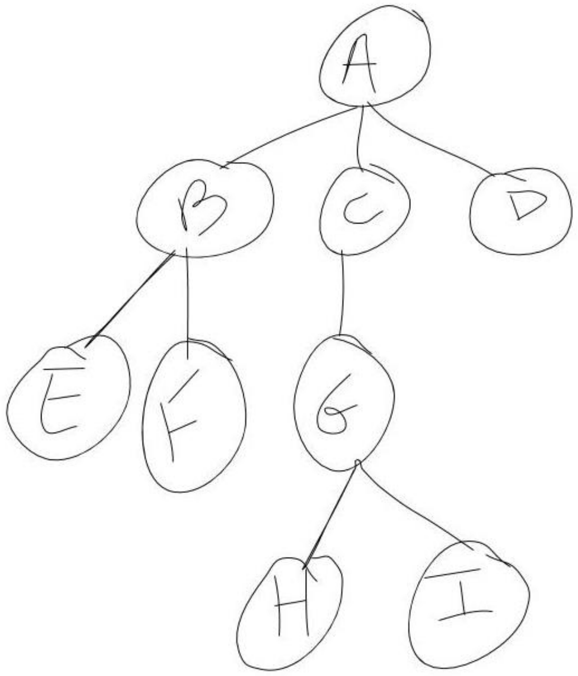
    for $\forall c \in (thisNode \rightarrow childList)$ **do**

      $DFS(c)$

    // Process thisNode here for postorder traversal

- Try Example

  E-B-F-A-H-G-I-C-D



Post-order

- Pseudo code for postorder traversal. Process node after visiting all children nodes

Algorithm 1

Require: recursive function initially called with root as parameter

    function DFS(Node* thisNode)

        // Process thisNode here for preorder traversal

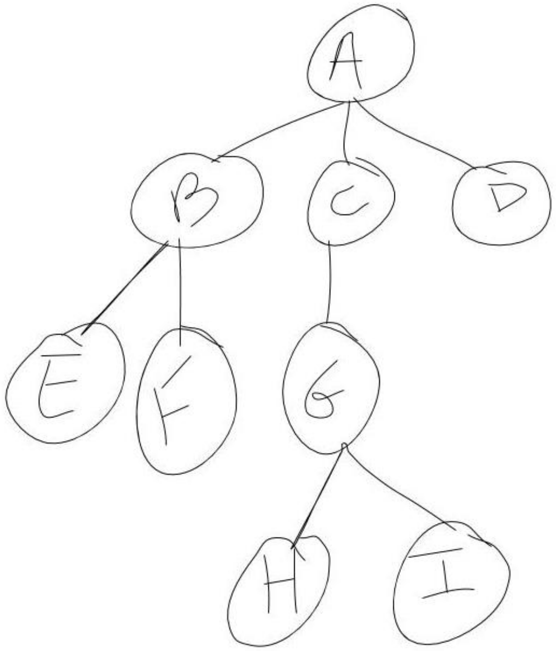
        for $\forall c \in (thisNode \rightarrow childList)$ **do**

            $DFS(c)$

        // Process thisNode here for postorder traversal

- Try Example

  E-F-B-H-I-G-C-D-A



Example Use

- Copy tree intuitively implements a preorder traversal

Algorithm 1

Require: initially thisNode is assumed root of a tree

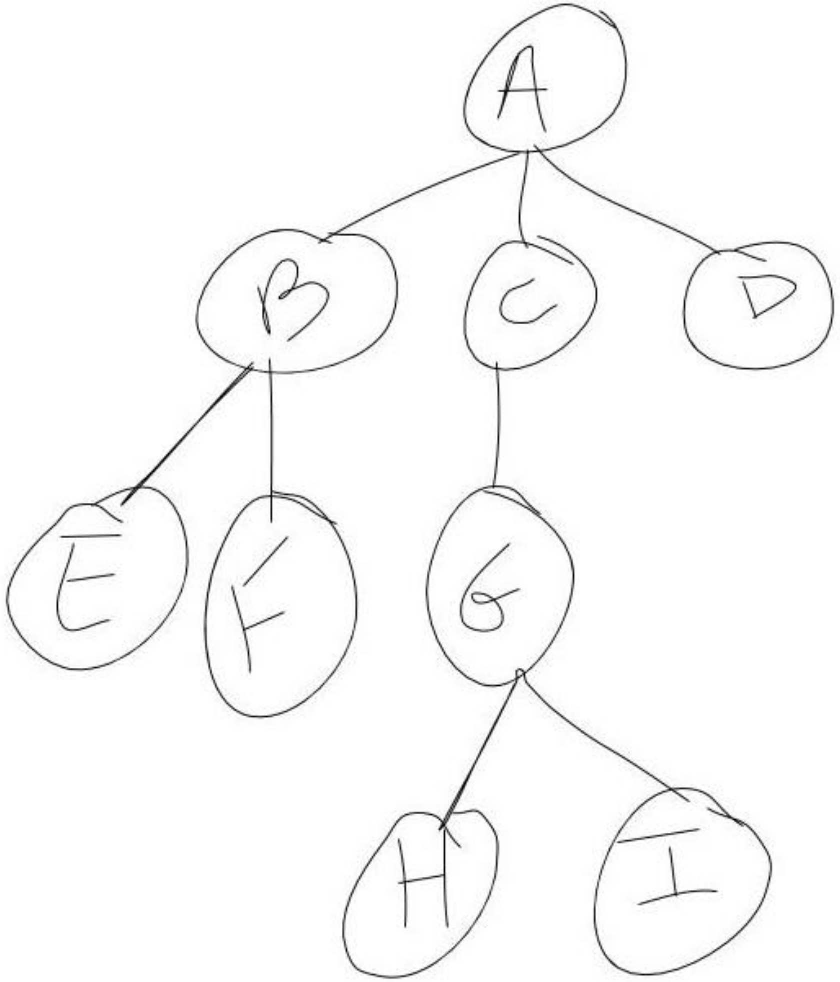
 function COPYTREE(Node* thisNode)

  $newNode \leftarrow newNode(thisNode \rightarrow data)$

  for $\forall c \in (thisNode \rightarrow childList)$ **do**

   $newNode.addToChildList(copyTree(c))$

  return $newNode$



Example Use

- Deallocate tree intuitively implements a postorder traversal

---

Algorithm 1

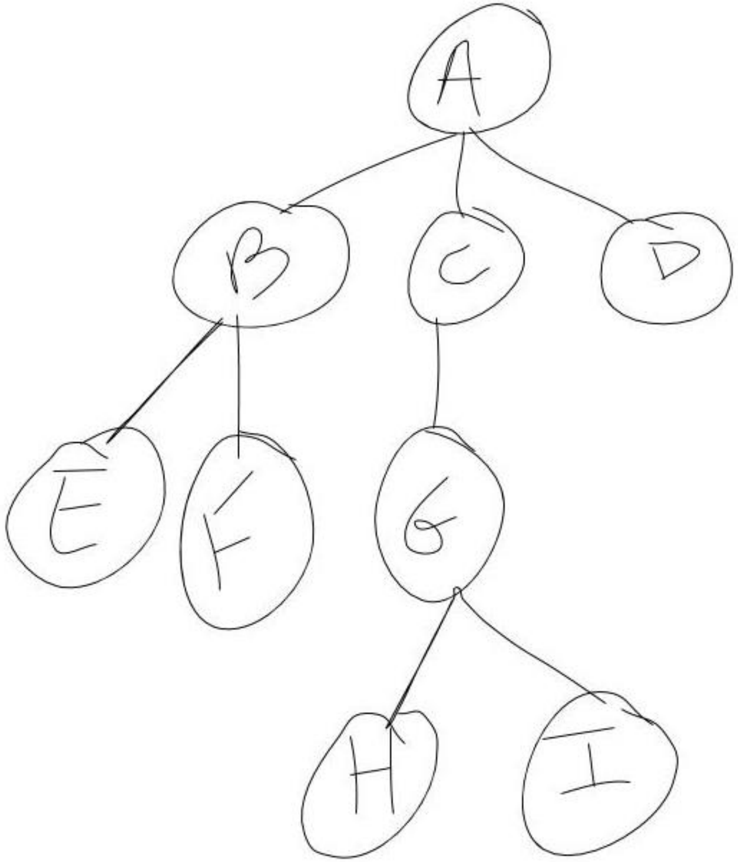
Require: initially thisNode is assumed root of a tree

  function DELETETREE(Node* thisNode)

    for $\forall c \in (thisNode \rightarrow childList)$ **do**

      $deleteTree(c)$)

    delete $thisNode$



Other examples

- As a practice exercise implement methods for the following:
  - Determine height of tree
  - Count number of elements
  - Make copy or delete
- Next Binary Trees
  - Similar to generic tree with extra constraint that each node has a maximum of two children, by standard, leftChild and rightChild.