COSC 2011 Section N

Tuesday, March 27 2001

Overview

- Trees and Binary Trees
  - Quick review of definitions and examples
- Tree Algorithms
  - Depth, Height
- Tree and Binary Tree Traversals
  - Preorder, postorder, inorder
- Binary Search Tree

Trees: Terminology and Basic Properties

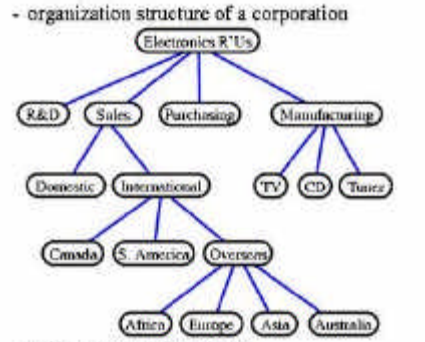
- Definitions (continued)
  - **Ancestor**:
    Either the node itself or an ancestor of the parent of the node.
  - **Descendant**:
    A node v is a descendant of a node u if u is an ancestor of v
  - **Descendant**:
    A node v is a descendant of a node u if u is an ancestor of v

Trees: Terminology and Basic Properties (continued)

- **Subtree:**
  The subtree of tree T rooted at a node v is the tree consisting of all descendants of v in T (including v itself).
- **Ordered Tree**:
  A linear ordering defined for the children of each node. We can identify the children as being the first, second, third etc.
- Note the Recursive Definitions for **Ancestor**, **Descendant** and **Subtree**!

Trees: Examples

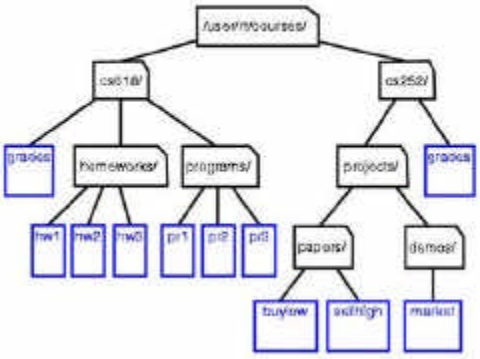
- organization structure of a corporation



- table of contents of a book

Trees: Another Example

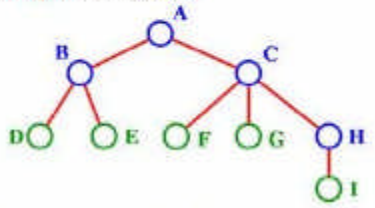
- Unix or DOS/Windows file system



- Internal nodes: directories
- External nodes: regular files.

Trees: Terminology

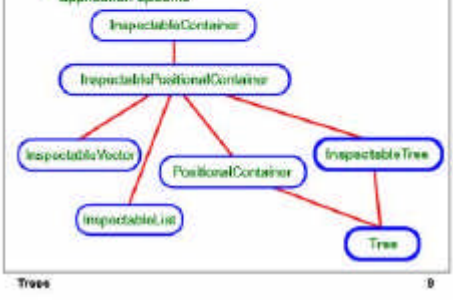
- *A* is the *root* node.
- *B* is the *parent* of D and E.
- *C* is the *sibling* of B
- *D* and *E* are the *children* of B
- *D, E, F, G, I* are *external nodes*, or *leaves*
- *A, B, C, H* are *internal nodes*
- The *depth* (*level*) of *E* is *2*
- The *height* of the tree is *3*
- The *degree* of node *B* is *2*



Property: (*# edges*) = (*#nodes*) – 1

ADTs for Trees

- generic container methods
  - size(), isEmpty(), elements()
- positional container methods
  - positions(), swapElements(p,q), replaceElement(p,e)
- query methods
  - isRoot(p), isInternal(p), isExternal(p)
- accessor methods
  - root(), parent(p), children(p)
- update methods
  - application specific



Trees: Binary Trees (1)

- Binary Tree
  - Ordered tree.
  - Each node has a maximum of two children.
- Definitions:
  - **Proper** Binary tree:

    Each node has either zero or two children.Every internal node has exactly two children.
  - **Left or Right Child**

    Each child of a node is labeled left or right child. Left child comes before the right child.

Trees: Binary Trees (1)

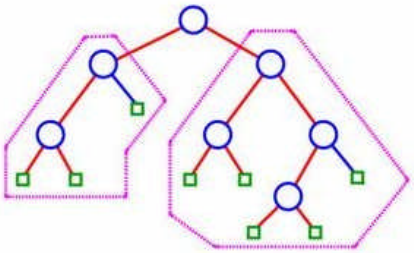
◆**Left and Right Subtree**

The subtree rooted at a left or right child of an internal node v.

Trees: Binary Trees (2)

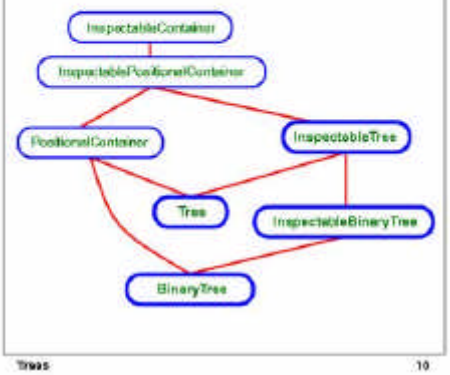
■ Recursive Definition:

- A ***binary tree*** is either
  - an external node (leaf), or
  - an internal node (the ***root***) and two binary trees (***left subtree*** and ***right subtree***)



ADTs for Binary Trees

- accessor methods
  - leftChild(p), rightChild(p), sibling(p)
- update methods
  - expandExternal(p), removeAboveExternal(p)
  - other application specific methods



Trees 10

Tree Algorithms: (1)

■ Assumptions:

◆Accessor methods *root()* & *parent(v)* take O(1) time.

◆Query methods isInternal(v) *isExternal(v)* & *isRoot(v)* take O(1) time.

◆Accessor method children(v) takes $O(c_v)$ time where c_v is the number of children of v.

Tree Algorithms: (2)

- Assumptions:
  - Generic methods *swapElements(v, w)* & *repalceElements(v, e)* take O(1) time.
  - Generic methods *elements*() & positions*(v)*, which return Iterators, take O(n) time.
  - Iterator methods take O(1) time.

Tree Algorithms: (3)

- **Depth** of a node *v* in a Tree:
  - The number of ancestors of the node excluding v itself.
  - Recursive Definition:
    - If *v* is the root, depth = 0.
    - Otherwise, depth of *v* is one plus the depth of the parent of *v*.

Tree Algorithms: (4)

- **Recursive Algorithm to compute Depth of node v**:
  - Calls itself recursively on the parent of *v*, and adds 1 to the return value.

```
Algorithm depth(T, v):
  if T.isRoot(v) then
    return 0
  else
    return 1+depth(T, T.parent(v))
```


Tree Algorithms: (5)

- **Height** of a node *v* in a Tree:
  - Recursive Definition:
    - If *v* is an external node, height = 0.
    - Otherwise, height of *v* is one plus the max. height of a child of *v*.
  - **Height** of a tree is the height of the root.
  - Height of a tree equals the max. depth of an external node of the tree.

Tree Algorithms: (6)

```
Algorithm height(T):
  h = 0
  for each v ∈ T.positions() do
    if T.isExternal(v) then
      h = max(h, depth(T, v))
  return h
```

- Not very efficient!
  - Worst case running time is $O(n^2)$.

Tree Algorithms: (7)

- **More Efficient Algorithm:**
  - **Uses recursive defn. of height.**
  - **Running Time is O(n)**

```
Algorithm height2(T, v):
  if T.isExternal(v) then
    return 0
  else
    h = 0
    for each w ∈ T.children(v) do
      h = max(h, height2(T, w))
  return 1 +h
```


Tree Traversals: (1)

- **A *traversal* of a tree T:**
  - Systematic way of accessing or "visiting" all nodes of T.
  - Specific action associated with "visit" to a node depends on the application - could be anything!
- Different Types of Traversals Available:
  - Differ in the way the nodes are visited.

Tree Traversals: (2)

- ***Preorder Traversal*:**
  - Root of the tree is visited first.
  - Subtrees rooted at the root's children are then visited recursively.

- preorder traversal

```
Algorithm preOrder(v)
  "visit" node v
  for each child w of v do
    recursively perform preOrder(w)
```

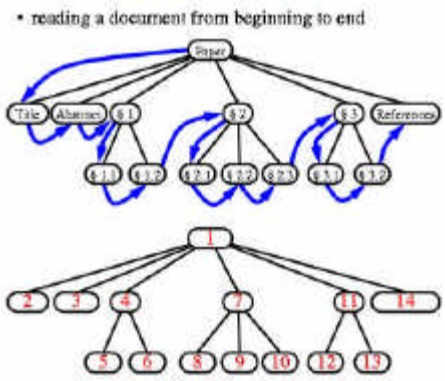
- reading a document from beginning to end

Tree Traversals: (3)

- Useful for:
  - Producing a linear ordering of the nodes of a tree where parents must always come before children.
- Efficient way to access all the nodes of a tree:
  - Assume visiting a node takes O(1) time.
  - At each node $O(1+c_v)$ where c_v is # of children of v.
    - ⋆Running Time is O(n).

Tree Traversals: (4)

- Example:
  - Document tree – if external nodes are removed, traversal examines Table of Contents.



Tree Traversals: (5)

- ***Postorder Traversal***:
  - Opposite of the preorder traversal.
  - Recursively traverses the subtrees rooted at the children of the root first, then visits the root.
  - Will visit a node v after it has visited all other nodes in the subtree rooted at v.

Tree Traversals: (6)

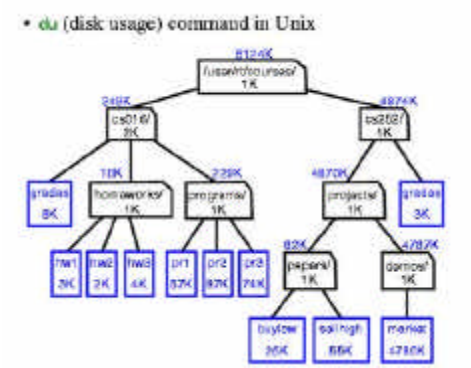
• postorder traversal

```
Algorithm postOrder(v)
  for each child w of v do
    recursively perform postOrder(w)
  "visit" node v
```

- Useful For:
  - Solving problems where we wish to compute some property for each node *v* but computing that property, requires we have already computed the property for the children of *v*.

Tree Traversals: (7)

- Example: File System Tree
  - Compute the disk space used by a directory.

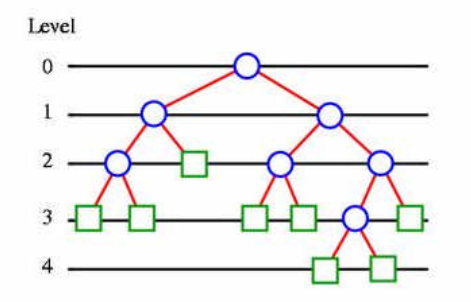


Tree Traversals: (8)

- ***Preorder*** and ***Postorder*** are common ways to traverse a tree, but other traversals are available:
  - Can visit all nodes at depth d before going to depth d+1.
    - ⋆ Use a queue!
- Don't necessarily need to use recursion!
  - Preorder and postorder can be done iteratively with a stack.

Binary Tree Properties (1)

- Level:
  - Set of all nodes of a tree T at the same depth d, as the ***level*** d of T.



Binary Tree Properties (2)

- Level 0 has one node – the root, level 1 at most 2 nodes, level 2 at most 4 nodes….
  - ⋆ Level d has at most 2^d nodes.
  - ⋆ Maximum number of nodes on the levels of a binary tree grows exponentially as we go down the tree.

Binary Tree Properties (3)

- Let T be a proper (non-empty) binary tree with n nodes, let h be the height of T, then:
  1. Number of external nodes in T is at least **h+1** and at most $\mathbf{2^h}$.
  2. The number of internal nodes in T is at least h and at most $\mathbf{2^h-1}$.
  3. Total number of nodes in T is at least 2h+1 and at most $\mathbf{2^{h+1} - 1}$.
  4. Height of T is at least **log(n+1)-1** and at most **(n-1)/2**. That is, **log(n+1) – 1 £ h £ (n-1)/2**
  5. **Number of external nodes = 1+ number of internal nodes.**