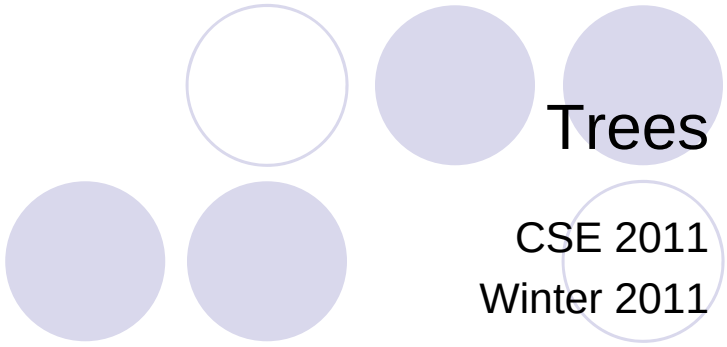


Trees

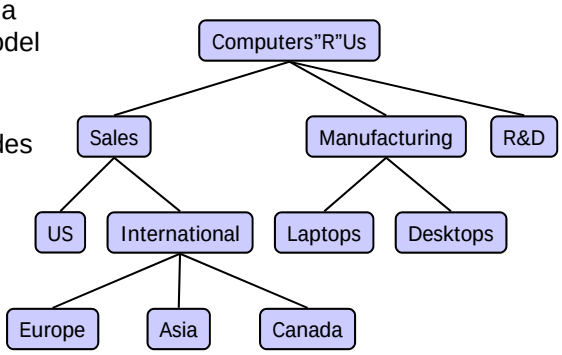
CSE 2011
Winter 2011

Trees

- Linear access time of linked lists is prohibitive
  - Does there exist any simple data structure for which the running time of most operations (search, insert, delete) is O(log N)?
- Trees
  - Basic concepts
  - Tree traversal
  - Binary trees
  - Binary search trees
  - AVL trees

General Trees (7.1)

- In computer science, a tree is an abstract model of a hierarchical structure
- A tree consists of nodes with a parent-child relation
- Applications:
  - Organization charts
  - File systems
  - Programming environments



Recursive Definition

- A tree is a collection of nodes.
  - The collection can be empty.
  - Otherwise, a tree consists of a distinguished node *r* (the *root*), and zero or more nonempty *subtrees* T_1, T_2, . . . , T_k, each of whose roots is connected by a directed *edge* from *r.*

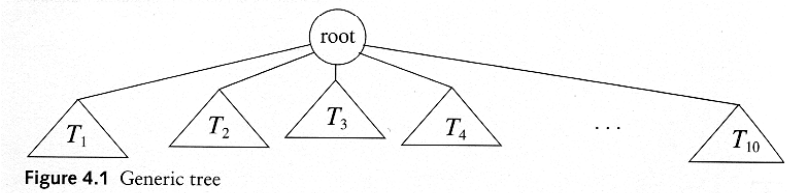


Figure 4.1 Generic tree

Applying the Recursive Definition

```
void operation ( T ) {
  if ( T is not empty )
    for every subtree T_i of T
      operation( T_i )
}
```

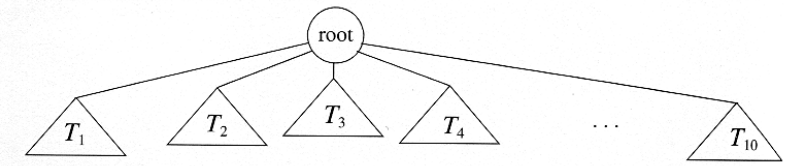


Figure 4.1 Generic tree

Terminologies

- *Child* and *Parent*
  - Every node except the root has one parent
  - A node can have zero or more children
- *Leaves*: nodes with no children
  - Also called *external* nodes
- *Internal* node: having one or more children
- *Siblings*: nodes having the same parent

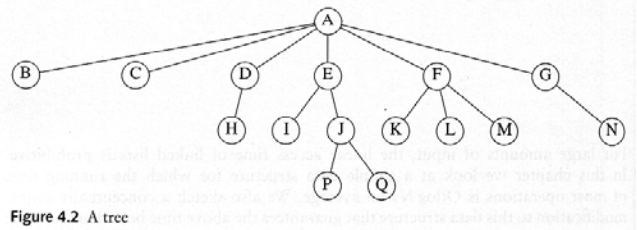


Figure 4.2 A tree

Terminologies (2)

- *Ancestor* and *descendant*
  - If there is a path from n1 to n2 then
    - n1 is an *ancestor* of n2 (parent, grandparent, great-grandparent, etc.)
    - n2 is a *descendant* of n1 (child, grandchild, great-grandchild, etc.)
  - A node is its own ancestor and descendant
  - *Proper ancestor* and *proper descendant* if n1 ≠ n2
- *Subtree*: tree consisting of a node and its descendants

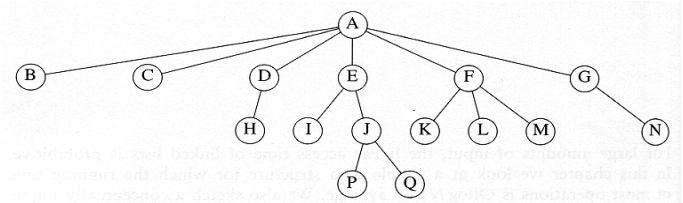


Figure 4.2 A tree

7

Terminologies (3)

- *Path*
  - a sequence of edges
- *Length of a path*
  - number of edges on the path
- *Depth* of a node
  - length of the unique path from the root to that node

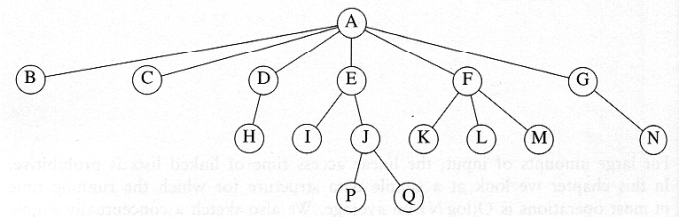


Figure 4.2 A tree

8

Terminologies (4)

- *Height* of a node
  - length of the longest path from that node to a leaf
  - all leaves are at height 0
- The height of a tree = the height of the root
  = the depth of the deepest leaf

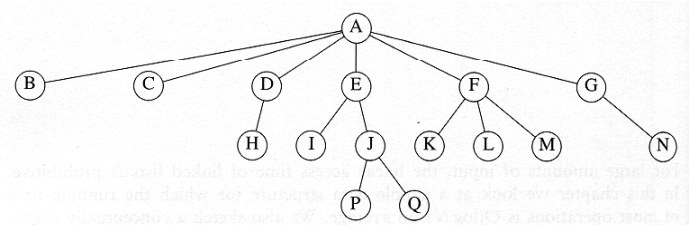


Figure 4.2 A tree

9

Example: UNIX Directory

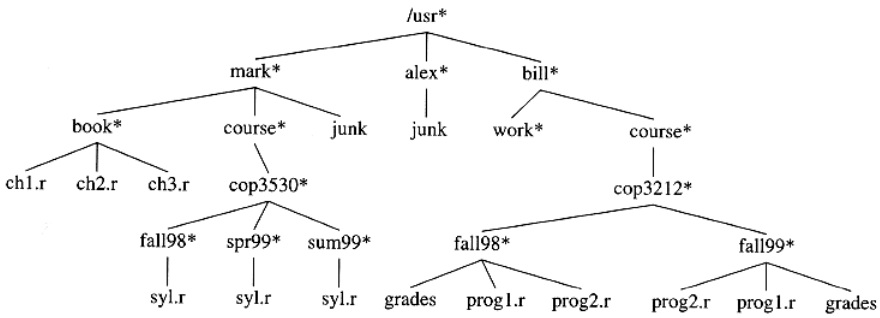


Figure 4.5 UNIX directory

10

Example: Expression Trees

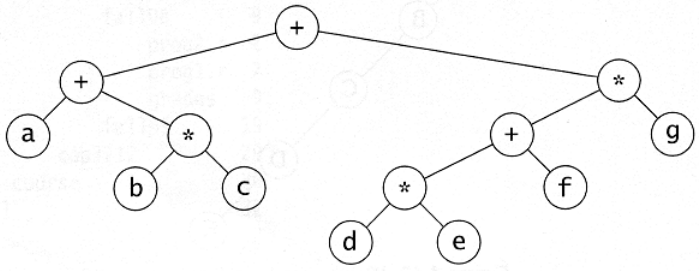


Figure 4.14 Expression tree for `(a + b * c) + ((d * e + f ) * g)`

- Leaves are operands (constants or variables)
- The internal nodes contain operators

Tree ADT

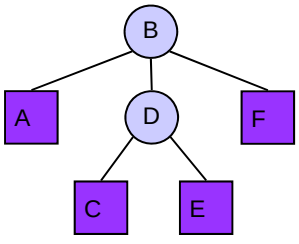
- We use positions to abstract nodes (position ≡ node)
- Generic methods:
  - integer size()
  - boolean isEmpty()
  - Iterator elements()
  - positionIterator positions()
- Accessor methods:
  - position root()
  - position parent(p)
  - positionIterator children(p)
- Query methods:
  - boolean isInternal(p)
  - boolean isExternal(p)
  - boolean isRoot(p)
- Update method:
  - object replace (p, e): replace with e and return element stored at node p
- Additional update methods may be defined by data structures implementing the Tree ADT

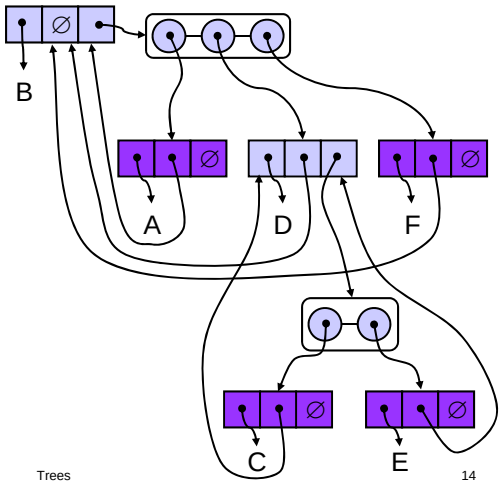
Implementing Trees

- Arrays ?
- Linked “lists” (pointers) ?

Linked Structure for Trees

- A node is represented by an object storing
  - Element
  - Parent node
  - Sequence of children nodes





Tree Traversal Algorithms (7.2)

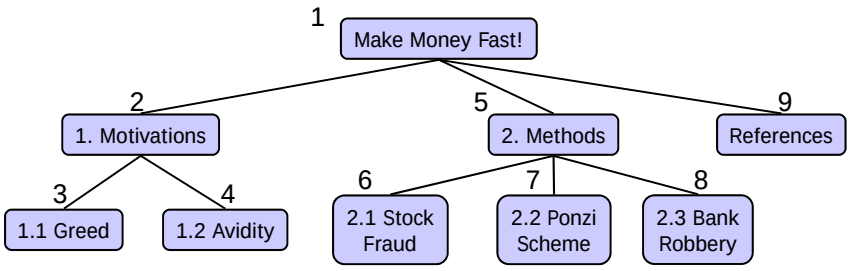
- Preorder
  - Visit *v* first
  - Then visit the descendants of *v*
- Postorder
  - Visit the descendants of *v* first
  - Then visit *v* last
- Depth and Height

15

Preorder Traversal

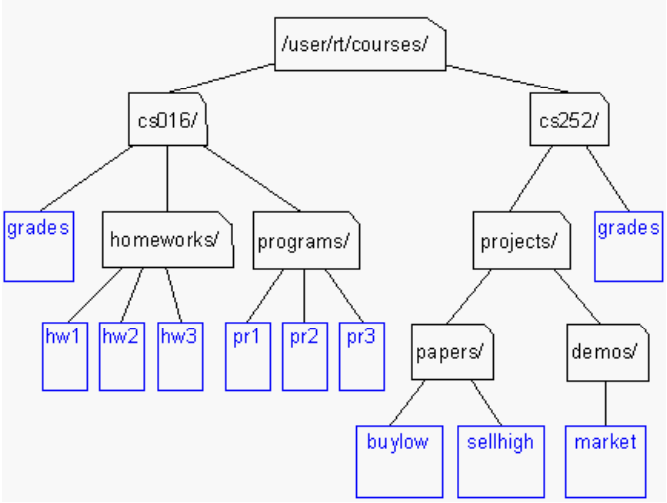
- A traversal visits the nodes of a tree in a systematic manner
- In a preorder traversal, a node is visited before its descendants
- Application: print a structured document

Algorithm ***preOrder***(***v***)
  visit(***v***)
  for each child ***w*** of ***v***
    preOrder (***w***)



16

An Example

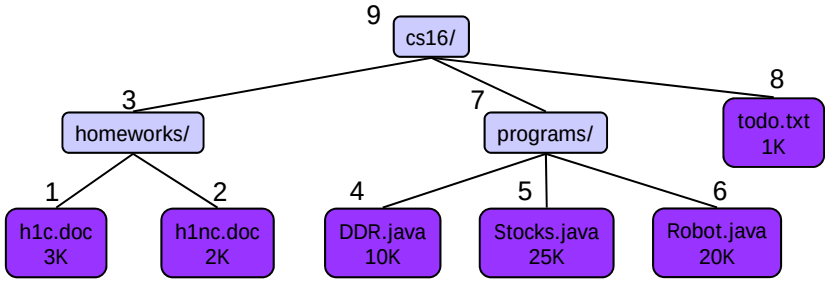


17

Postorder Traversal

- In a postorder traversal, a node is visited after its descendants
- Application: compute space used by files in a directory and its subdirectories

Algorithm ***postOrder***(***v***)
  for each child ***w*** of ***v***
    postOrder (***w***)
  visit(***v***)



18

Applications

- Either preorder traversal or postorder traversal can be used when the order of computation is not important.

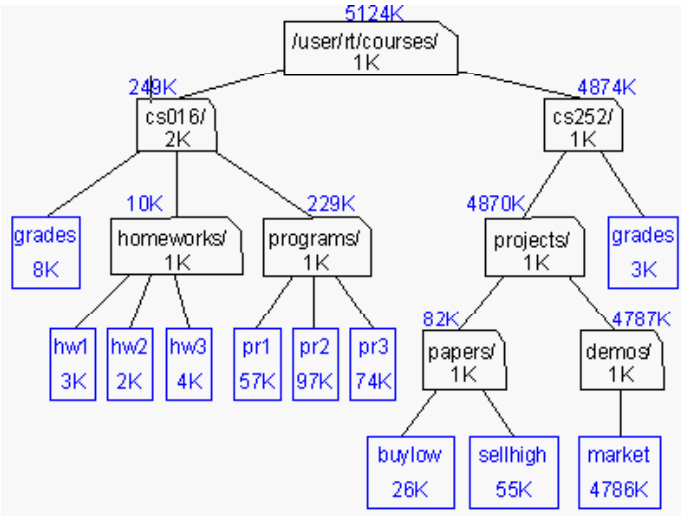
  Example: printing the contents of a tree (in any order)
- *Preorder* traversal is required when we must perform a computation for each node ***before*** performing any computations for its descendents.

  Example: Printing the headings of chapters, sections, sub-sections of a book.
- *Postorder* traversal is needed when the computation for a node *v* requires the computations for *v*'s children to be done first.

  Example: Given a file system, compute the disk space used by a directory.

19

Example: Computing Disk Space



20

Example: UNIX Directory Traversal

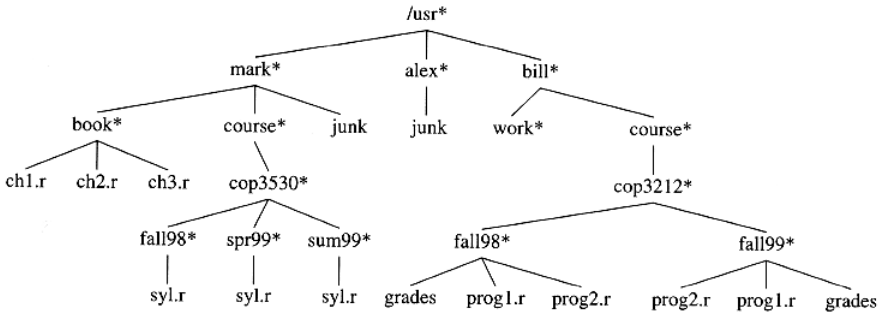


Figure 4.5 UNIX directory

Example: Unix Directory Traversal

Preorder

```
/usr
    mark
        book
            ch1.r
            ch2.r
            ch3.r
        course
            cop3530
                fall98
                    syl.r
                spr99
                    syl.r
                sum99
                    syl.r
        junk
    alex
        junk
    bill
        work
        course
            cop3212
                fall98
                    grades
                    prog1.r
                    prog2.r
                fall99
                    prog2.r
                    prog1.r
                    grades
```

Postorder

```
            ch1.r          3
            ch2.r          2
            ch3.r          4
        book              10
                syl.r      1
            fall98         2
                syl.r      5
            spr99          6
                syl.r      2
            sum99          3
        cop3530           12
    course                13
    junk                   6
mark                      30
    junk                   8
alex                       9
    work                   1
                grades     3
                prog1.r    4
                prog2.r    1
            fall98         9
                prog2.r    2
                prog1.r    7
                grades     9
            fall99        19
        cop3212           29
    course                30
bill                      32
/usr                      72
```

Depth

- *Depth* of node *v*: length of the unique path (number of edges) from the root to *v*.
- Recursive definition:
  - If *v* is the root, then depth of *v* is 0.
  - Otherwise, depth of *v* is 1 plus the depth of *v*'s parent.

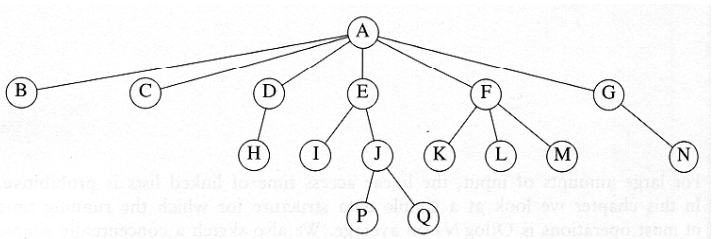


Figure 4.2 A tree

23

Algorithm ***depth***

```
Algorithm depth( T, v ) {
   if ( isRoot( v ) )
      return 0;
   return ( 1 + depth( T, parent( v ) ) );
}
```

Running time = ?

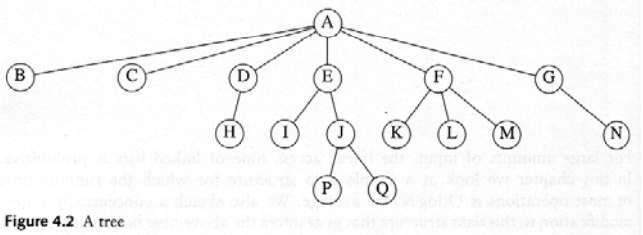


Figure 4.2 A tree

24

Height

- *Height* of a node
  - length of the longest path from that node to a leaf
  - all leaves are at height 0
- The height of a tree = the height of the root
  = the depth of the deepest leaf

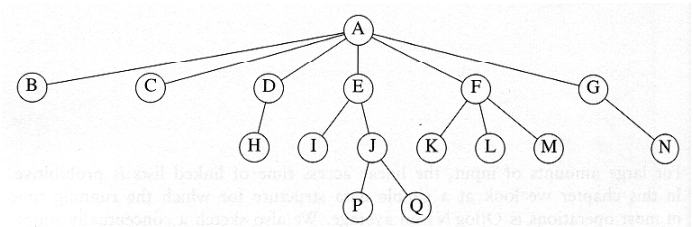


Figure 4.2 A tree

25

Computing the Height of a Tree

- The height of a tree = the depth of the deepest leaf

```
Algorithm height( T ) {
   h = 0;
   for every node v in T
      if( isExternal( v ) )
         h = max( h, depth( T, v ) );
}
```

Running time: $O(n) + \Sigma_v(d_v + 1)$ for all external nodes v

$\Sigma_v d_v = O(n^2)$ in the worst case $\Rightarrow$ not efficient

26

Recursive Definition of Height

- The height of a node v in a tree T is defined as follows:
  - If v is a leaf node, then height of v is 0.
  - Otherwise, height of v is 1 plus the maximum height of a child of v.

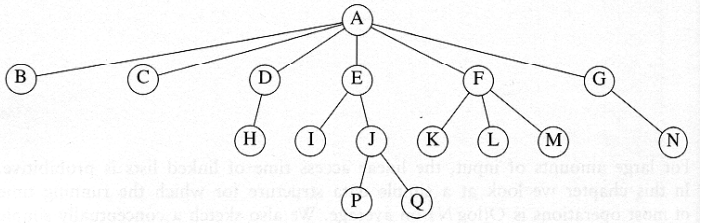


Figure 4.2 A tree

27

Algorithm ***height2***

```
Algorithm height2( T, v ) {
   if ( isExternal( v ) )
      return 0;
   h = 0;
   for every child w of v
      h = max( h, height2( T, w ) );
   return( 1 + h );
}
```

- Height of the tree: **`H = height2( T, root );`**
- Running time: $\Sigma_v(c_v + 1) \Rightarrow$ O(?)
- We visit each node exactly once.

28

Next time …

- Binary Trees (7.3)
- Binary Search Trees (10.1)

29