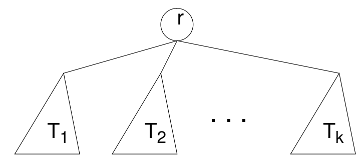
CSE 326
Lecture 10

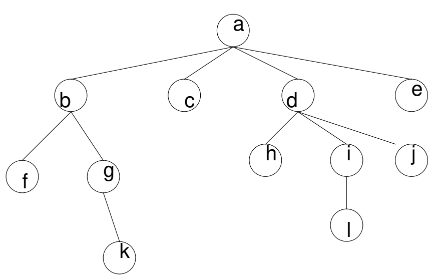
Spring Quarter 1995

Tree Terminology

- Recursive Definiton of trees
  - Empty tree Λ has no root.
  - Given trees T_1, T_2, ..., T_k and a node r a treeT is formed by making the root r whose children are the roots of T_1, T_2, ... , T_k respectively and the parent of the roots of T_1, T_2, ... , T_k is r.

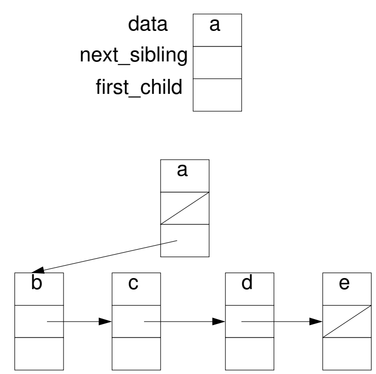


Example

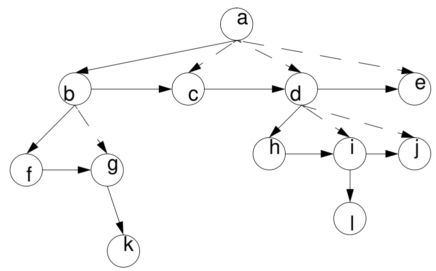


- root
- node
- children
- parent
- sibling
- descendent
- ancestor
- subtree
- leaf
- depth
- height

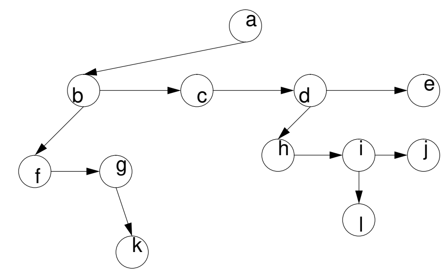
Basic Tree Data Structure



View of Tree Data Structure



Actual Data Structure for the Tree

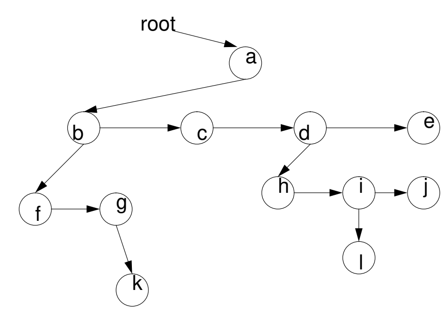


Preorder Traversal

- Preorder - Visit the node, then recursively visit each of the subtrees.

```
preorder(n) {
local node pointer c;
if not(n=null) then {
   Do_something;
   c := n.first_child;
   while not (c=null) do {
     preorder(c);
     c := c.next_sibling}}}
```


Preorder Traversal Example



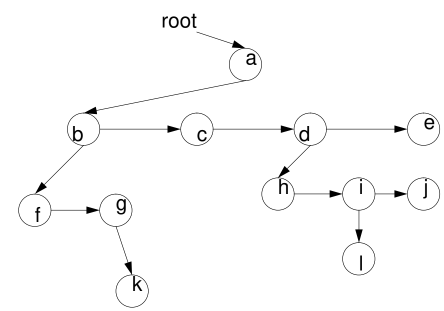
Computing the Depth of all Nodes

- Add a field called "depth"

```
Depth(n,d) {
local node pointer c;
if not(n=null) then {
  n.depth :=d;
  c := n.first_child;
  while not (c=null) do {
      Depth(c,d+1)
      c := c.next_sibling}}}
```

- Call Depth(root,0) to set the depth field correctly.

Depth Calculation

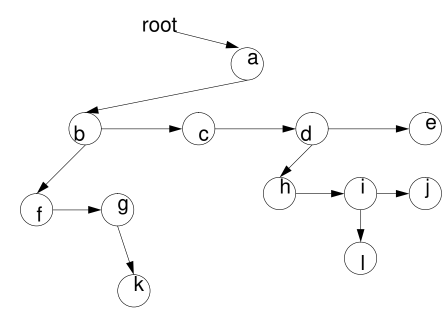


Postorder Traversal

- Postorder - Recursively visit each of the subtrees, then visit the node.

```
postorder(n) {
local node pointer c;
if not(n = null) then {
  c := n.first_child;
  while not(c=null) do {
     postorder(c);
     c := c.next_sibling};
  Do_something}}
```


Postorder Traversal Example



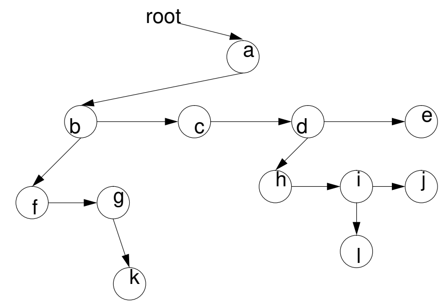
Computing the Height of Each Node

- We have a new field "height" for each node

```
Height(n) {
local node pointer c;
if not(n=null) then
  if n.first_child = nil then n.height := 0 else {
     h := 0;
     c := n.first_child;
     while not (c=null) do {
        Height(c);
        h := max(h,c.height);
        c := c.next_sibling};
     n.height := h +1 }}}
```

- Call Height(root) to correctly set all height fields

Height Example



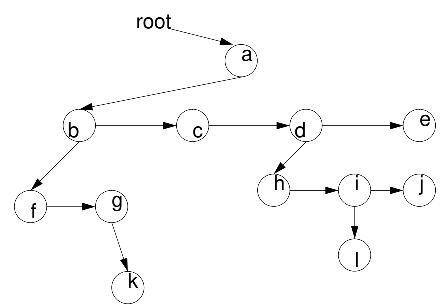
Computing the Size of a Tree with a Function

- count(n) {

```
local node pointer c, integer m;
if n = null then return(0) else {
   c := n.first_child;
   m := 1;
   while not (c=null) do {
      m := m + count(c);
      c := c.next_sibling}
   return(m) }}
```

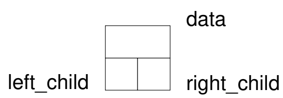
- Call count(root) to compute the number of nodes in the tree.

Size of Tree Example

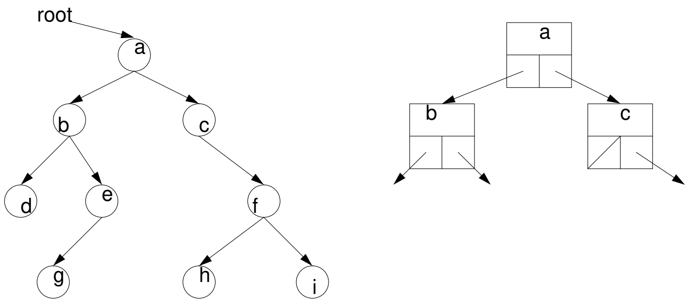


Binary Trees

- Each node has at most two children.



A Binary Tree Example



Inorder Traversal

- Inorder - Recursively visit the left subtree, then visit the node, then recursively visit the right subtree.

  inorder(n) {

  if not(n = null) then {
      inorder(left_child(n));
      Do_something;
      inorder(right_child(n)) }}

Binary Search Tree

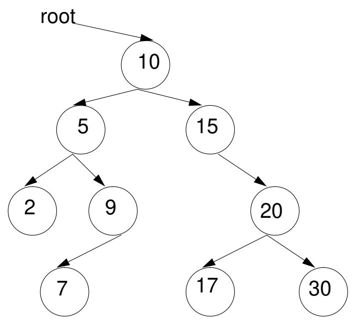
- A binary search tree has a number at each node and the tree has the following property.

  For every node n, all the numbers in the left subtree of n are less than or equal to the number at n, and all the numbers in the right subtree of n are greater than or equal to the number at n.

  Concisely:

  left subtree of node $\leq$ node $\leq$ right subtree of node

Example Binary Search Tree

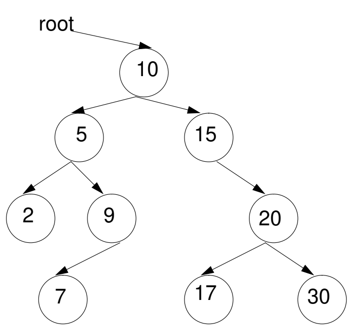


Listing the Data in Order

- List(n)

  if not(n=null) then {
  List(left_child(n));
  print(n.data);
  List(right_child(n)) }}
- Call List(root) to list all the data in order.

Example of Listing in Order



Warning - Not a Binary Search Tree