Trees

Textbook Reading:

Chapter 4, Section 4.1, pp. 138-143

Sections 4.3 and 4.4, pp. 153-161.

Trees

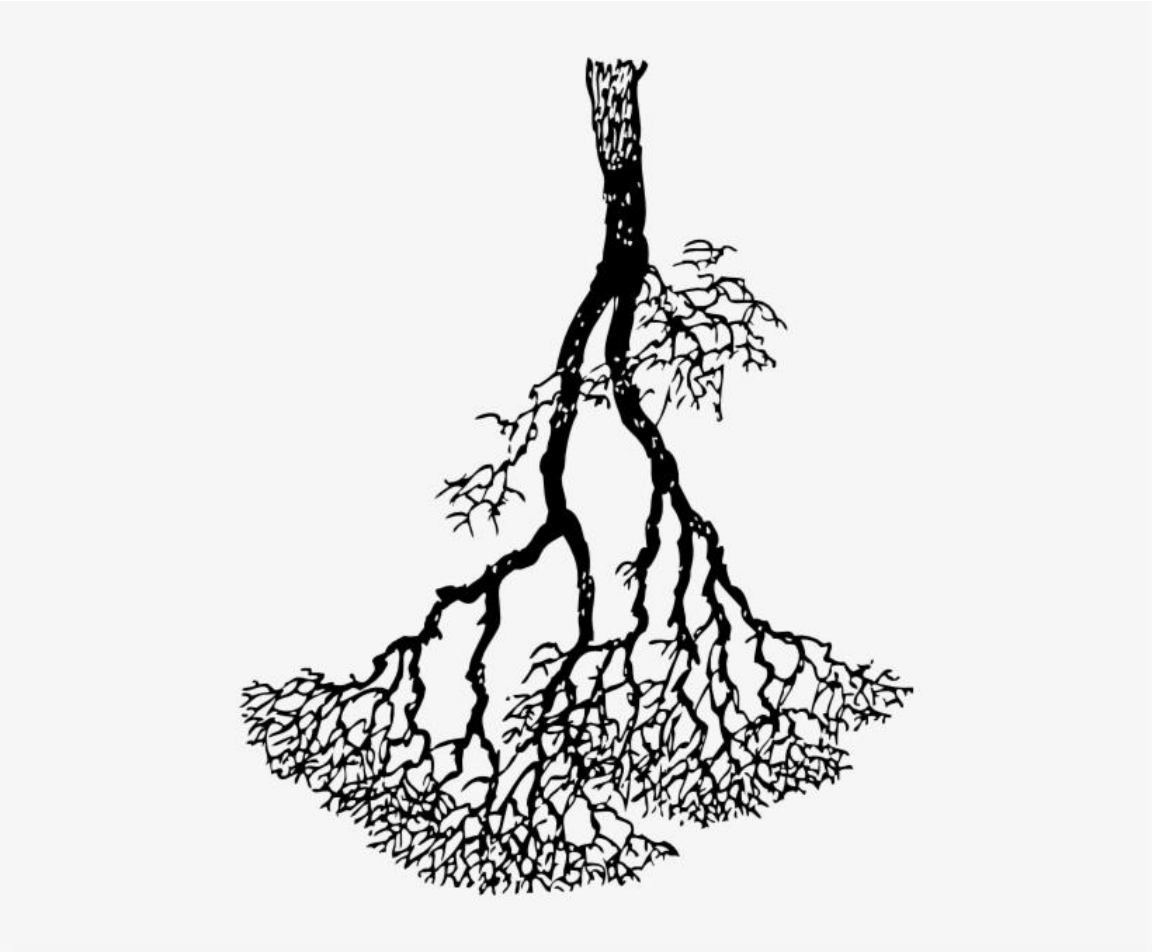
A tree is an important structure in CS with myriad applications.

- It models operations in networks such as broadcasting from a source and gathering at a sink.
- It is an important data structure used in many applications and algorithms.
- Mathematical properties of trees have important applications in the analysis of algorithms.

Important Types of Trees

- Top-Down Design Tree
- Tree of Recursive Calls
- Directory Tree in Unix
- Family Tree
- Expression Tree
- Decision Tree
- Game Tree
- State Space Tree for Backtracking and Branch-&-Bound
- Minimum Spanning Tree
- Shortest Path Tree
- Etc.

Trees in computing are usually illustrated growing from root down

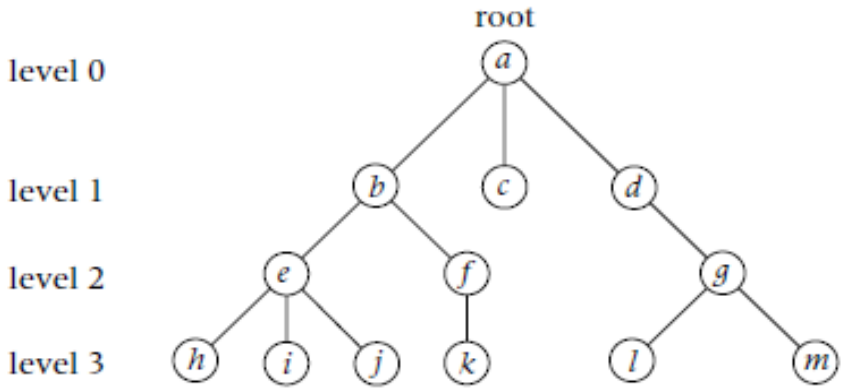


Tree Definition and Terminology

A **tree** consists of a set of **nodes** (also called **vertices**), where one node is identified as the **root** and each node different from the root has another node associated with it called its **parent**.

A node is joined to it parent using an **edge**.

The set of all nodes having the same parent *p* are called the **children** of *p*.



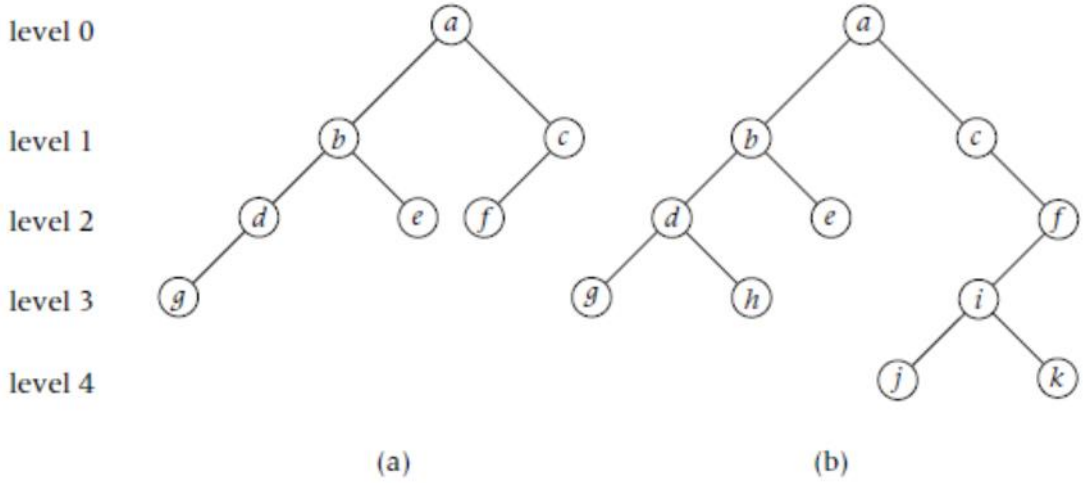
Sample tree of depth 3

Tree Terminology

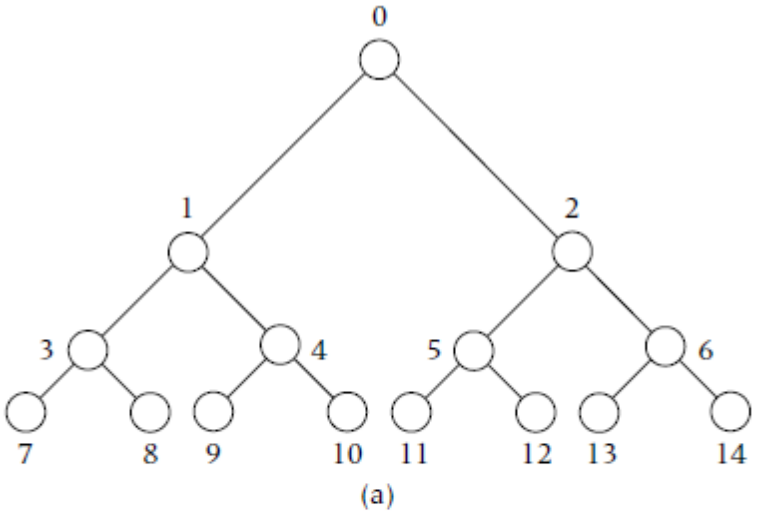
- The set of all nodes having the same parent *p* are called the **children** of *p*.
- If node *p* has parent *g* and child *c* then *g* is the **grandparent** of *c* and *c* is the **grandchild** of *g*.
- A node with no children is called a **leaf**.
- A node *X* is an **descendent** of *Y* if there is a path in the tree from *X* to *Y*. *Y* is an **ancestor** of *X*.
- The **depth** (also called **height**) is the highest level number, i.e., the maximum length of a path from the root to a leaf.

Binary Trees

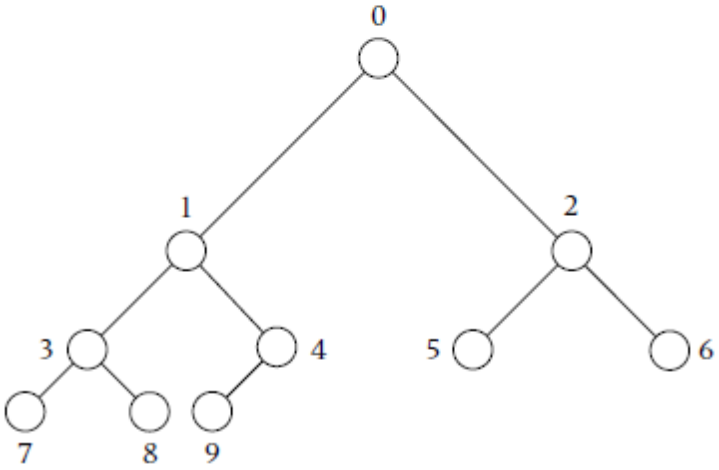
A binary tree is a tree where every node has **at most** 2 children and we identify children as either a left child or right child.



Full Tree and Complete Binary Trees



(a)



(b)

PSN. Implementing a full and complete tree

a) Draw complete binary with values on nodes implemented by the array:

0	1	2	3	4	5	6	7	8	9
10	20	60	88	1	2	7	11	-4	3

Given the index i of a node in a complete tree

b) Give formula for index of parent of i.

c) Give formulas for indices of left and right children of i.

Left-right child implementation of a binary tree

BinaryTreeNode = **record**

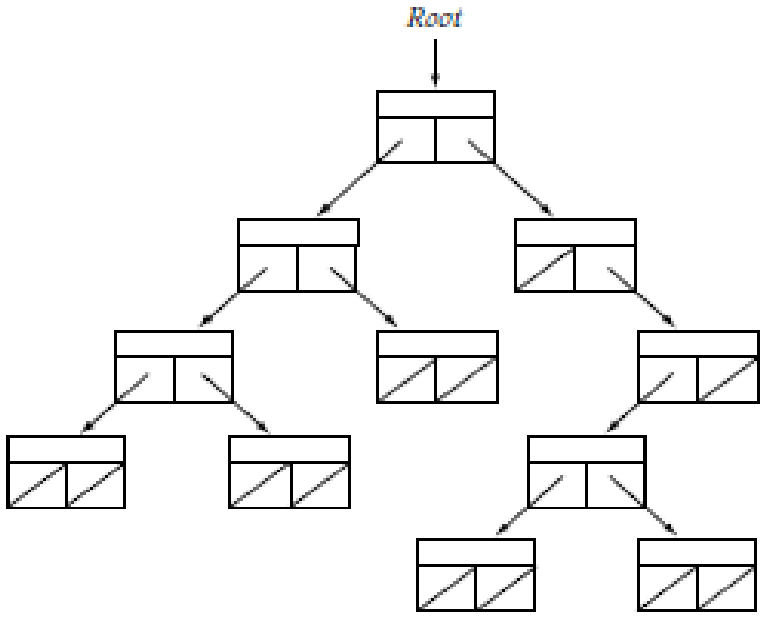
Info: *InfoType*

LeftChild: $\rightarrow$ *BinaryTreeNode*

RightChild: $\rightarrow$ *BinaryTreeNode*

end *BinaryTreeNode*

(a)



(b)

Binary Tree Traversals

The *preorder* traversal of *T* is defined recursively as follows:

If *T* is not empty, then

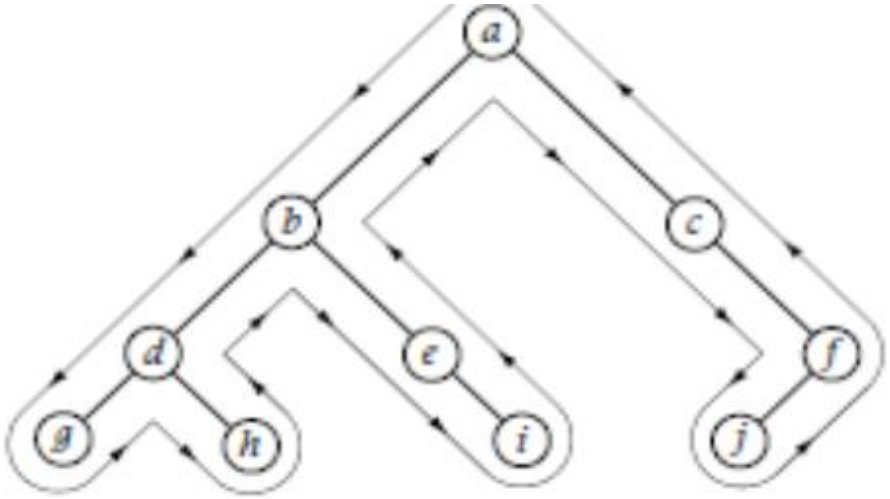
1. visit the root node *R*,
2. perform a preorder traversal of the left subtree *LT* of the root,
3. perform a preorder traversal of the right subtree *RT* of the root.

inorder and *postorder* traversals the same except that
1. is done after 2. and 3., respectively

Path around the Tree

- The resolution of the recursion determines a path around the input binary tree. The path of recursive calls is the same for preorder, inorder, postorder.
- The difference is when a node is visited, i.e., call to `Visit(Current->Item)`
- In a Preorder Traversal, it is visited the **first time** it is accessed;
- in an Inorder Traversal it is visited the **second time** it is accessed (if no left-child, it is considered accessed again after accessing the NULL left child pointer);
- in a Postorder Traversal it is visited the **last time** it is accessed.

Example Binary Tree Traversals



Preorder Traversal: *abdgheicfj*
Inorder Traversal: *gdhbeiacjf*
Postorder Traversal: *ghdiebjfca*

PSN. Code for Binary Tree Traversals

Give C++ code for Preorder, Inorder and Postorder traversals, invoking function Visit() to perform operations on data in the node.

Assume the nodes of the tree are implemented using the structure:

```
typedef int datatype;
struct Node {
    datatype item;
    Node *leftchild;
    Node *rightchild;
}; //END TREENODE

typedef *Node ptrNode
```

Variations

- A class could have been used to implement the binary tree nodes instead of a structure.
- In practice item is often a key in a record (implemented as a struct or class) with multiple fields.
- For example, the key could be the customer ID. Other fields could be the customer's name, address, account info, phone number, etc.

Applications of Binary Tree Traversals

a) Performing a deep copy of one binary tree to another (copy constructor).

b) Deleting all the nodes in a binary tree (destructor)

c) Evaluating a binary expression tree.

d) Outputting the items of a binary search tree in sorted order.

e) Storing a binary search tree in a file so that it can be reconstructed when the file is scanned sequentially.

g) Generating Huffman code

Deep Copy of a Binary Tree – Preorder Traversal

```
void CopyTree(ptrNode root, ptrNode& newroot)
{
// A deep copy of the binary tree that root points is performed with
newroot pointing to the root of copied tree

  if (root == NULL)
      newroot = NULL;
  else {
      newroot = new Node;
      newroot -> item = root -> item;
      CopyTree(root -> leftchild,newroot->leftchild);
      CopyTree(root -> rightchild,newroot->rightchild);
  }END else
} // END CopyTree()
```

Copy Constructor

Copy Constructor for class BinaryTree, where pointer to Root is pointer to Root in private part

```
BinaryTree::BinaryTree(const BinaryTree& tree)
{
   CopyTree(tree.Root, Root)
} //END Copy Constructor
```

Deleting a Binary Tree – Postorder Traversal

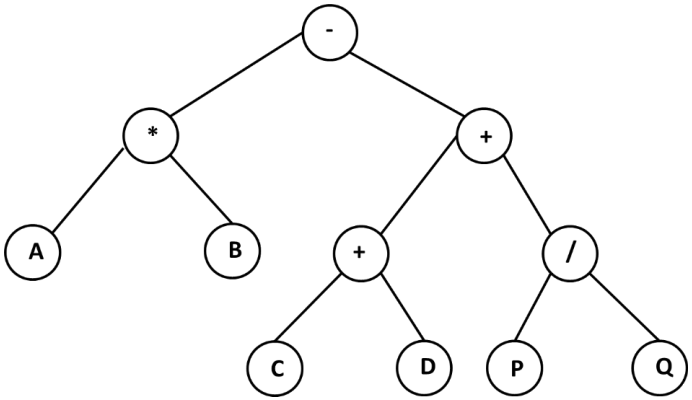
```
void DeleteTree(ptrNode root)
{
// All dynamically allocated nodes in the binary tree that
// root points to are deleted.

   if (root != NULL){
        DeleteTree(root -> leftchild);
        DeleteTree(root -> rightchild);
        delete root;
   } // END if
} // END Preorder()
```

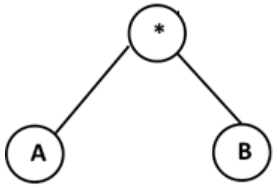
Destructor

Destructor for class BinaryTree, where pointer to Root is pointer to Root in private part:

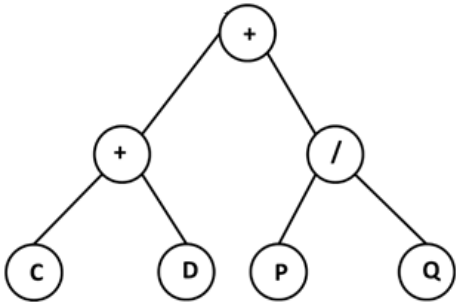
```
BinaryTree::~BinaryTree( )
        {
            DeleteTree(Root)
        } //END destructor
```

Expression Tree – Postorder





Evaluate left tree to get A*B

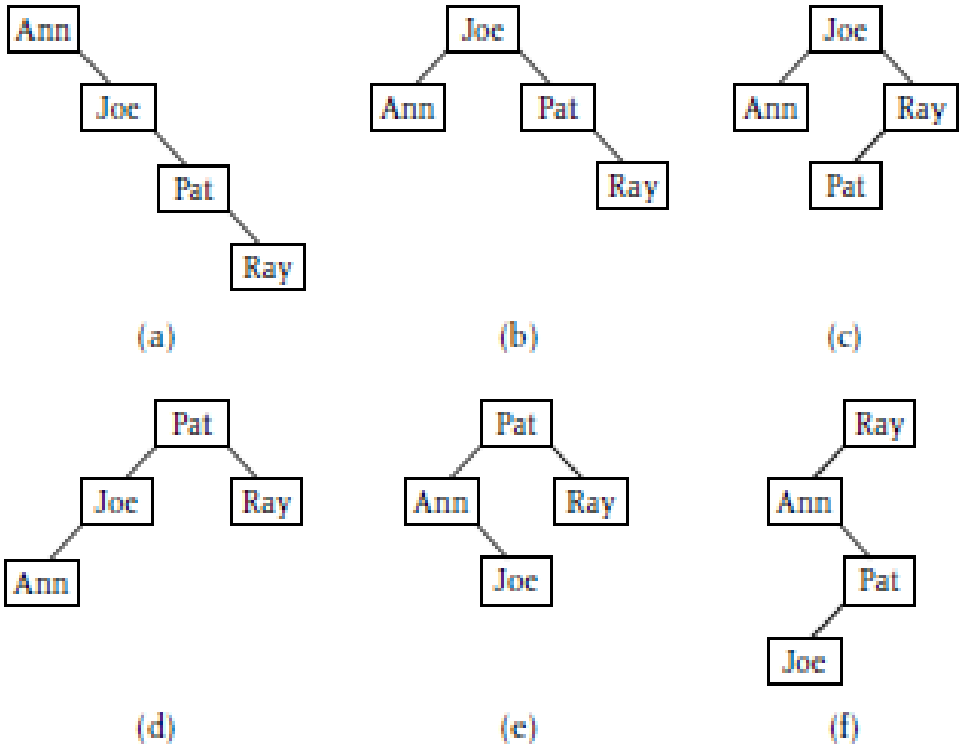


Evaluate right tree to get (C+D)+(P/Q)

Perform operation at root to get (A*B) – ((C+D)+(P/Q))

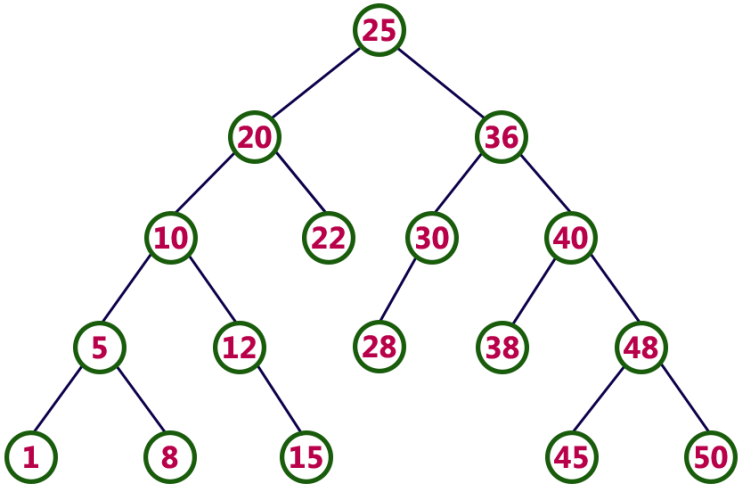
Binary Search Trees

A **binary search tree** is a binary tree with a key (value) associated with each node, so that for every node, all the keys in its left subtree are smaller and all the keys in its right subtree are larger.



Binary Search Tree – Inorder Traversal

Proposition. An inorder traversal of a binary search tree outputs the node keys in sorted order.



Inorder traversal: 1 5 8 10 12 15 20 22 25 28 30 36 38 40 45 48 50

Basis Step

Clearly, result is true for a binary search tree having one node.

Induction Step: Strong

Assume the result is true of all binary search trees have j nodes, where $1 \leq j \leq k$, i.e., performing an inorder traversal of a binary search tree having j nodes, outputs the keys in sorted order.

Now consider a binary search tree having $k + 1$ nodes.

Applying Induction Hypothesis

- Let *L* and *R* denote the left and right subtrees of *T*.

- Since both *L* and *R* have at most *k* nodes, we can apply the **induction hypothesis** to them, i.e., **performing an inorder traversal of *L* outputs the keys in sorted order. The same applies to *R***.

- Performing an inorder traversal of *T* involves
  - performing an inorder traversal of *L*,
  - visiting the root,
  - performing an inorder traversal of *R*.

- Since all the keys in *L* are less than the root key and all the keys in *R* are greater than the root key, it follows that an inorder traversal of *T* outputs the keys in sorted order.

This completes the induction step and the proof of the Proposition.

Illustration with Previous Sample Binary Search Tree

L root *R*

1 5 8 10 12 15 20 22 25 28 30 36 38 40 45 48 50

Applying Induction Hypothesis with *L*

output root key

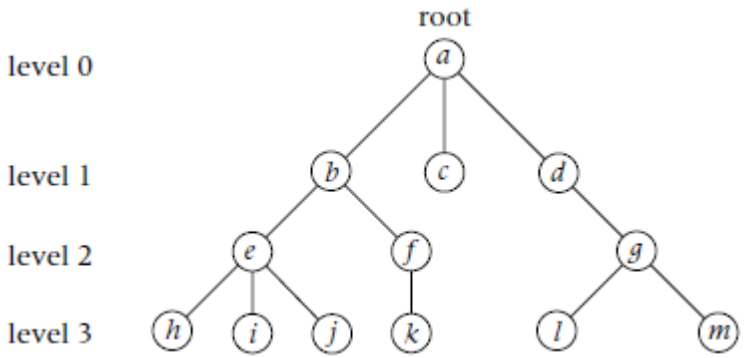
Applying Induction Hypothesis with *R*

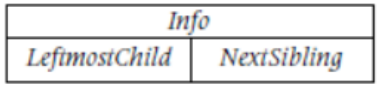
Treesort

1. Construct a binary tree T from the given list $L[0,n-1]$ by a sequence of insertions.
2. Perform an inorder traversal of T.

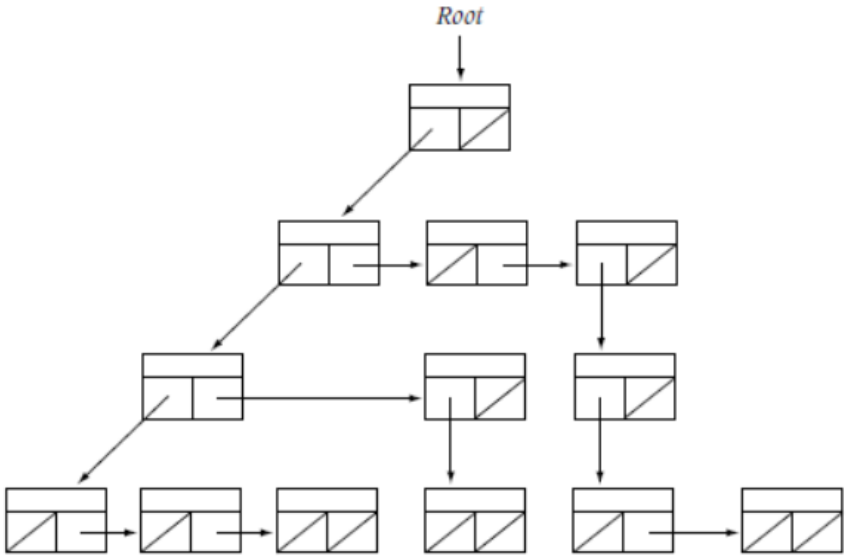
Child-Sibling Representation of a General Tree

TreeNode = **record**
    Info: *InfoType*
    LeftmostChild: → *TreeNode*
    NextSibling: → *TreeNode*
end *TreeNode*





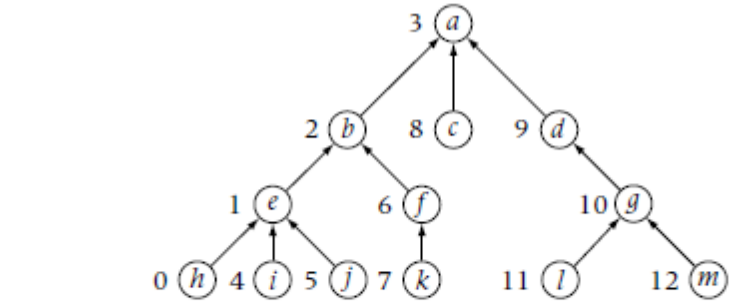
(a)



This implementation of a general tree leads naturally to a transformation, known as the **Knuth Transformation**, from a general tree to a binary tree. We simply think of the *LeftmostChild* as the left child of the node and the *NextSibling* as the right child.

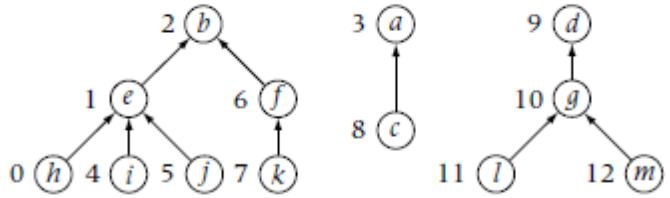
Parent Array Implementation of Trees and Forests

Path from 7 to 3: $7, Parent[7] = 6, Parent[6] = 2, Parent[2] = 3$



i	0	1	2	3	4	5	6	7	8	9	10	11	12
$Info[i]$	h	e	b	a	i	j	f	k	c	d	g	l	m
$Parent[i]$	1	2	3	−1	1	1	2	6	3	3	9	10	10

(a)



i	0	1	2	3	4	5	6	7	8	9	10	11	12
$Info[i]$	h	e	b	a	i	j	f	k	c	d	g	l	m
$Parent[i]$	1	2	−1	−1	1	1	2	6	3	−1	9	10	10

(b)

How do trees access the internet?

They log on.