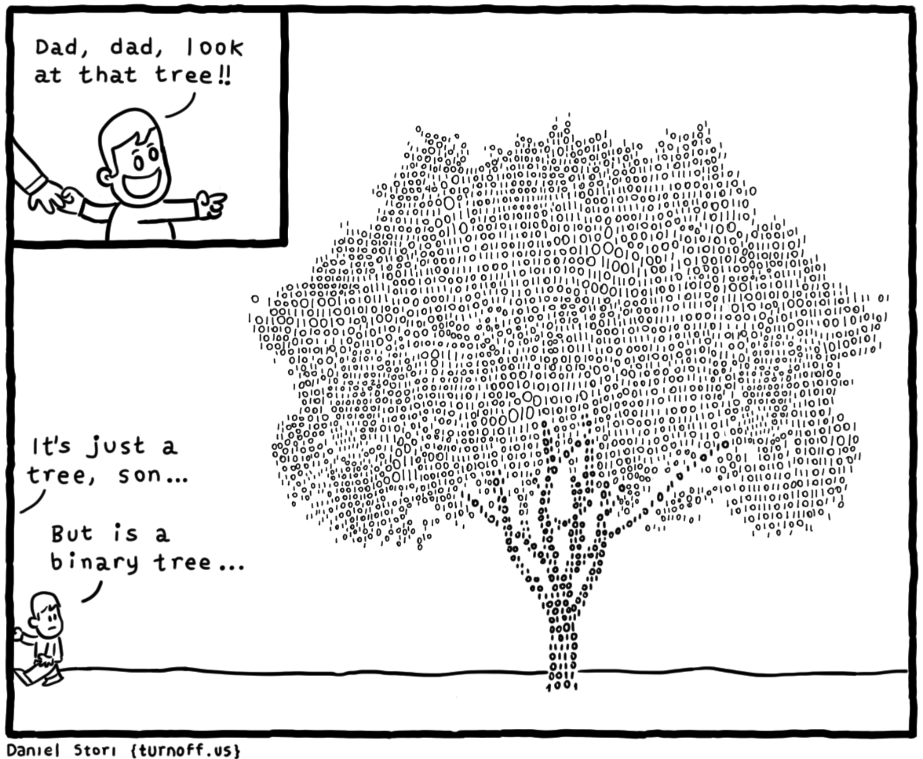
BBM 201 Data structures

Lecture 11: Trees



2019-2020 Fall

Content

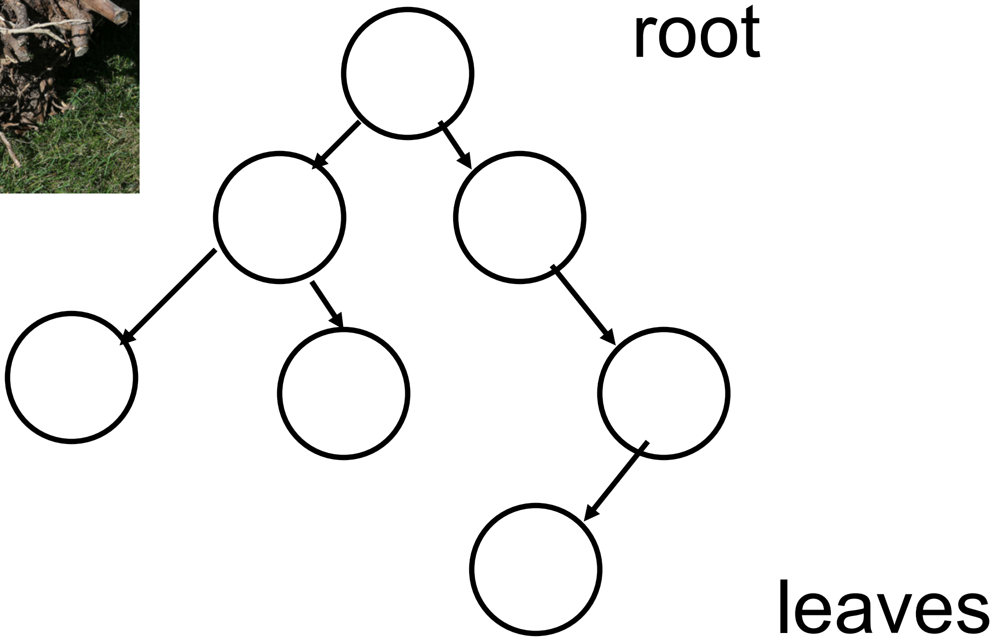
- Terminology
- The Binary Tree
- The Binary Search Tree

Terminology

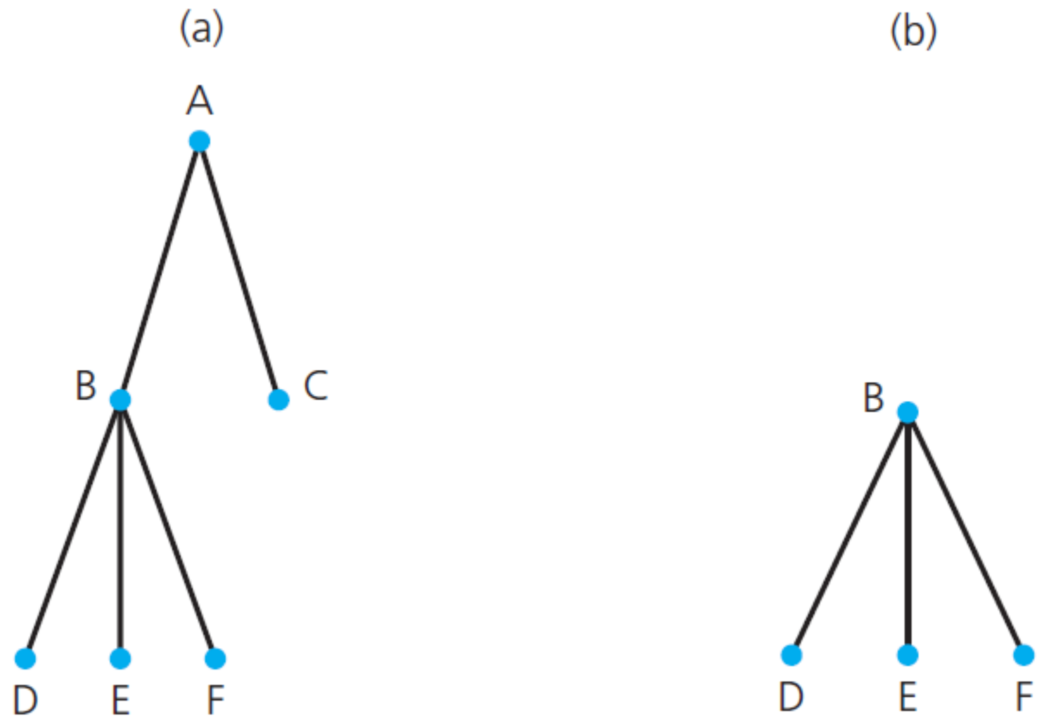
- Trees are used to represent relationships
- Trees are hierarchical in nature
  - "Parent-child" relationship exists between nodes in tree.
  - Generalized to ancestor and descendant
  - Lines between the nodes are called edges
- A subtree in a tree is any node in the tree together with all of its descendants

Terminology

- Only access point is the root
- All nodes, except the root, have one parent
  - like the inheritance hierarchy in Java
- Traditionally trees are drawn upside down



Terminology



(a) A tree;
(b) a subtree of the tree in part a

Terminology

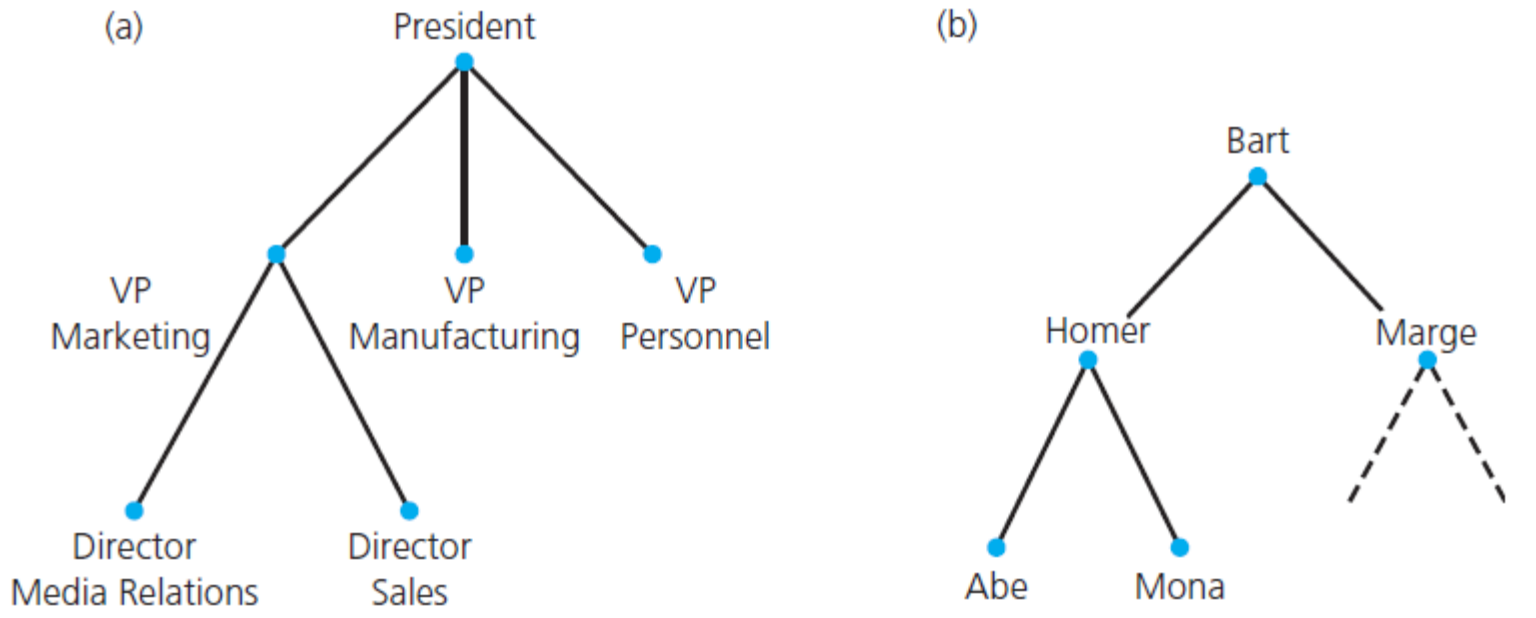
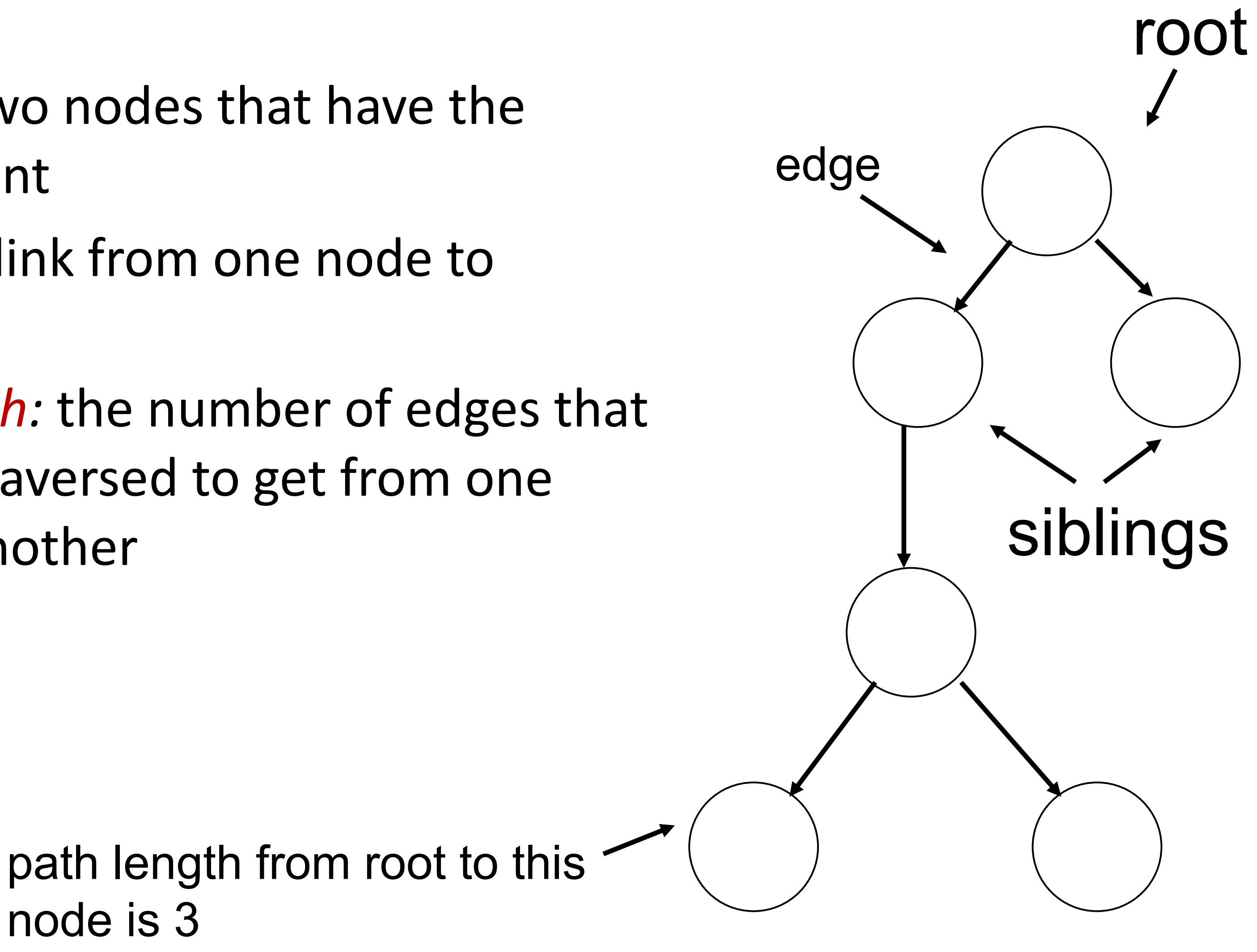


FIGURE 15-2 (a) An organization chart; (b) a family tree

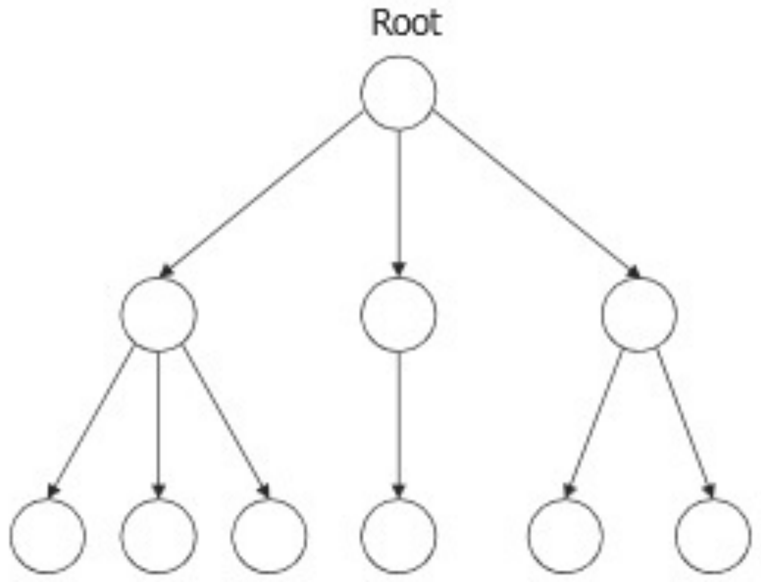
Properties of Trees and Nodes

- *siblings:* two nodes that have the same parent
- *edge:* the link from one node to another
- *path length:* the number of edges that must be traversed to get from one node to another



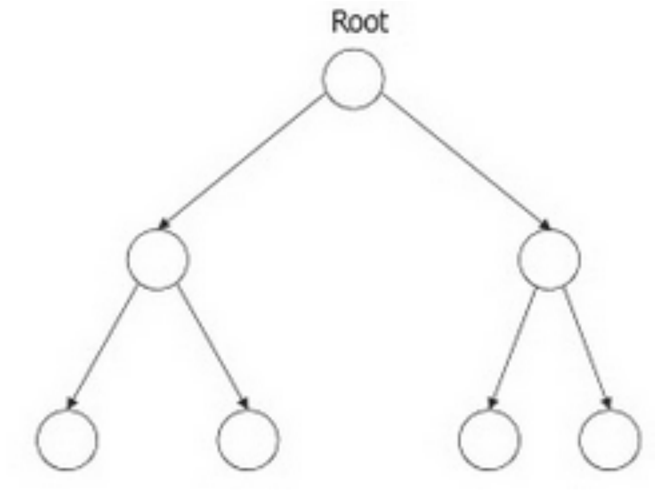
General Tree

- A general tree is a data structure in that each node can have infinite number of children
- A general tree cannot be empty.



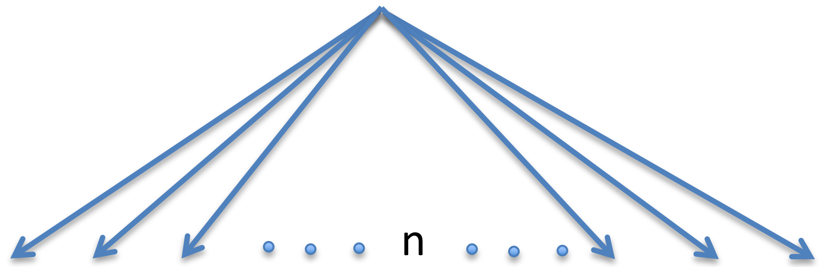
Binary Tree

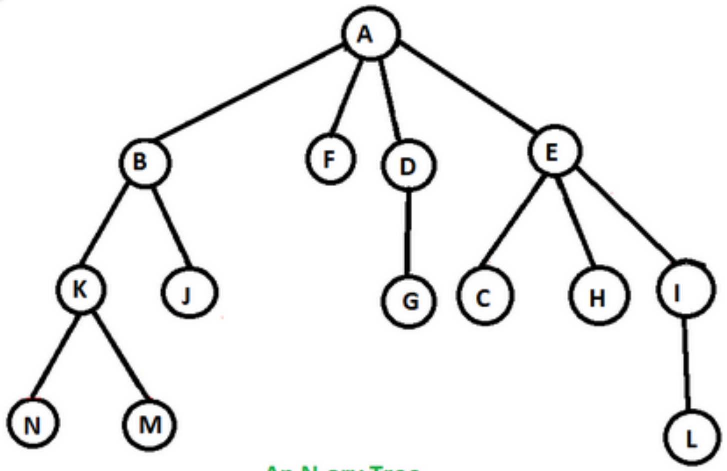
- A Binary tree is a data structure in that each node has at most **two children nodes:** left and right.
- A Binary tree can be **empty**.



n-ary tree

– A generalization of a binary tree whose nodes each can have no more than **n** children.





An N-ary Tree.

Example: Algebraic Expressions.

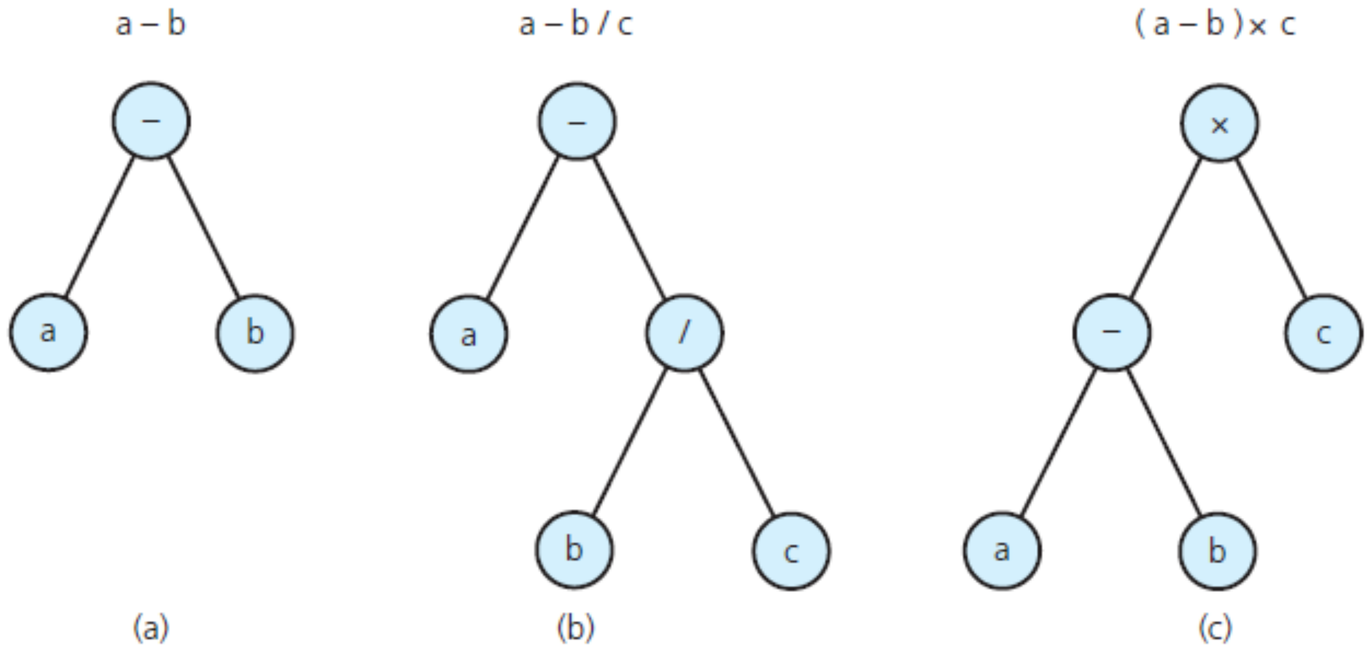
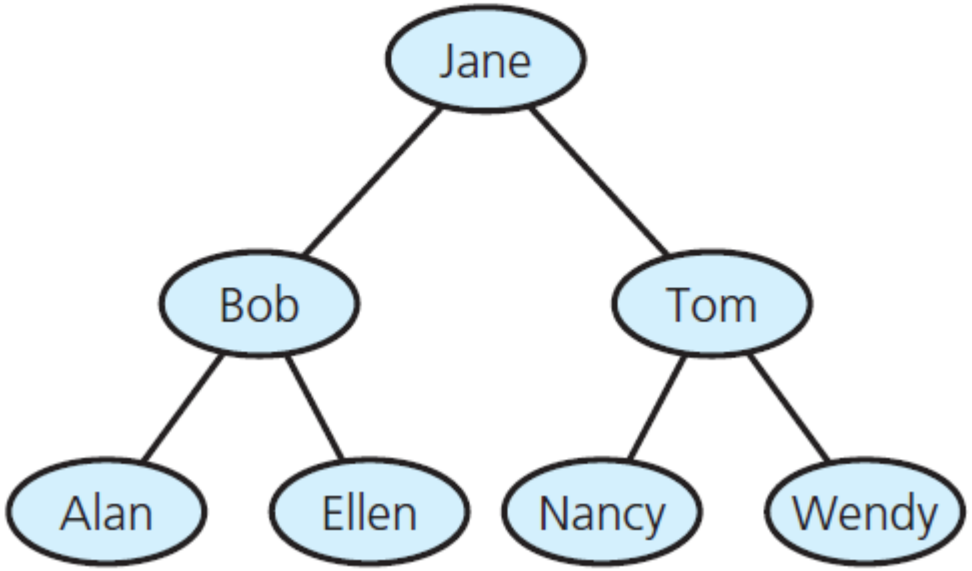


FIGURE 15-3 Binary trees that represent algebraic expressions

Level of a Node

- Definition of the level of a node n :
  - If n is the root of T, it is at level 1.
  - If n is not the root of T, its level is 1 greater than the level of its parent.



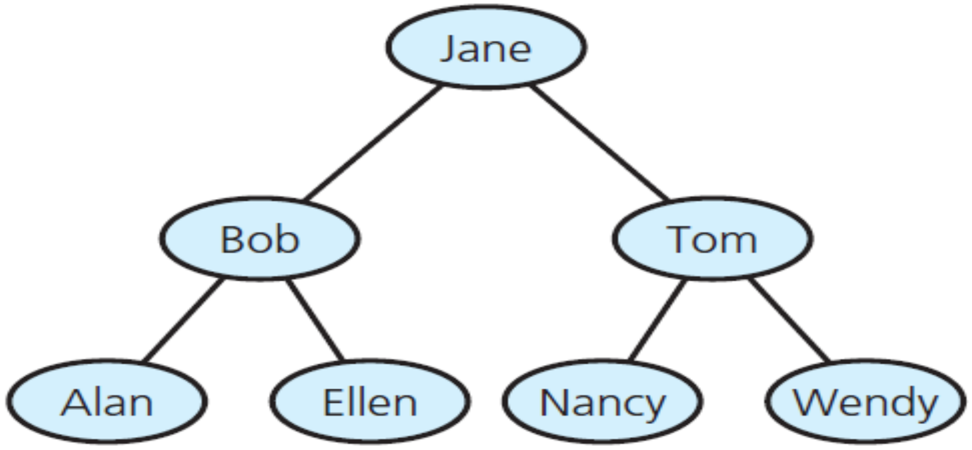
Level = 1

Level = 2

Level = 3

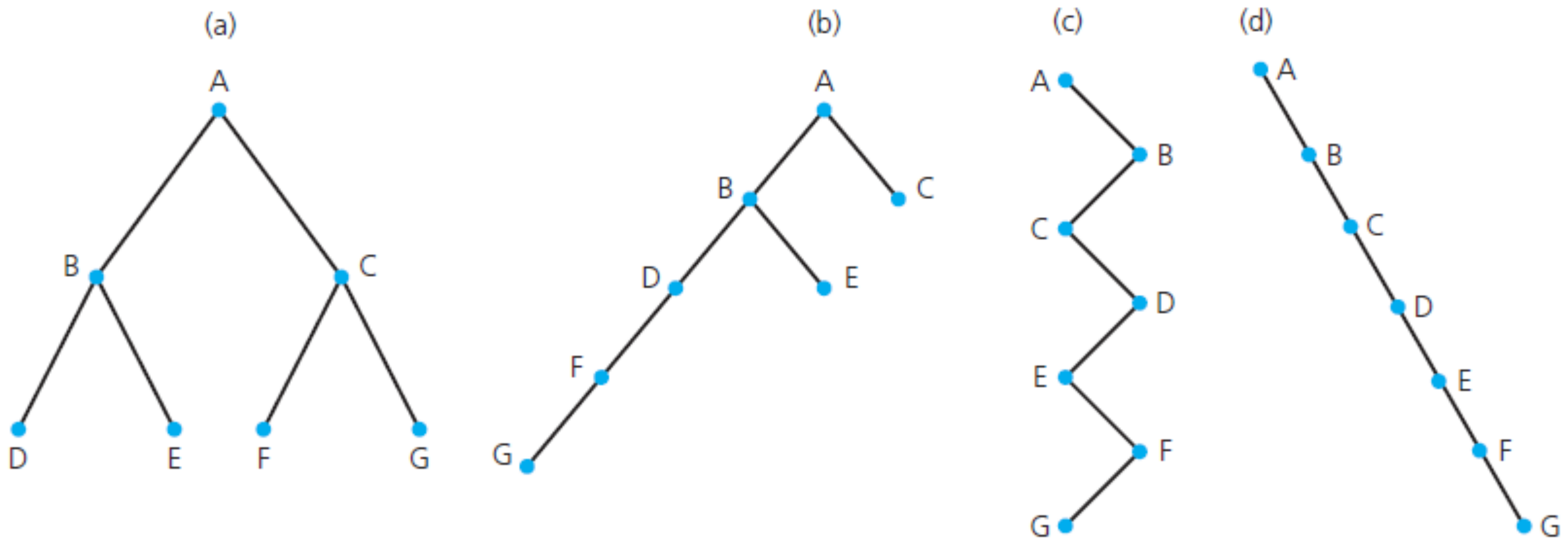
Height of Trees

- *The height of a node is the number of edges on the longest downward path between that node and a leaf.*



Height of tree = 2

The Height of Trees

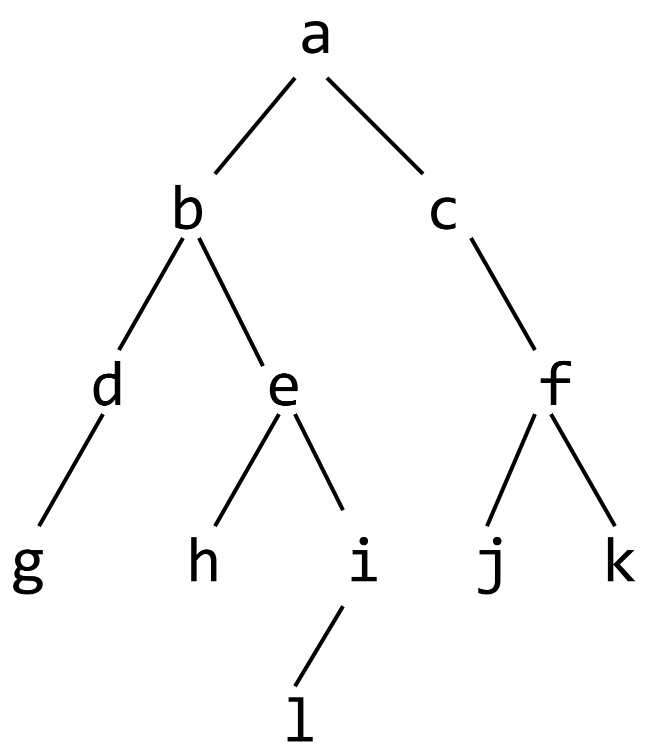


Height 2 Height 4 Height 6 Height 6

Binary trees with the same nodes but different heights

Depth of a Tree

- The path length from the root of the tree to this node.



The depth of a node is its distance from the root

a is at depth zero

e is at depth 2

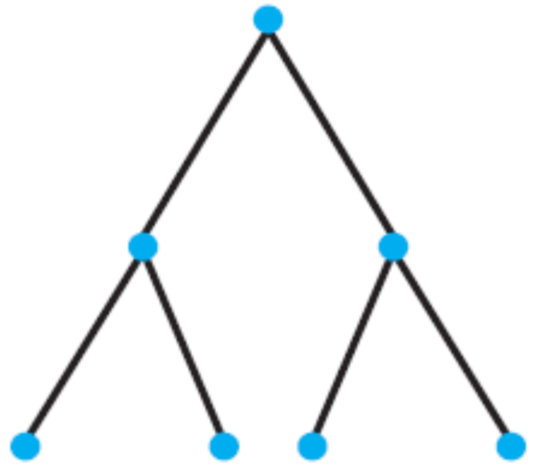
The depth of a binary tree is the depth of its deepest node

This tree has depth 4

Full, Complete, and Balanced Binary Trees

Full Binary Trees

- Definition of a full binary tree
  - If T is empty, T is a full binary tree of height 0.
  - If T is not empty and has height h > 0, T is a full binary tree if its root's subtrees are both full binary trees of height h – 1.
  - Every node other than the leaves has two children.



Facts about Full Binary Trees

- You cannot add nodes to a full binary tree without increasing its height.
- The number of nodes that a full binary tree of height h can have is $2^{(h+1)} - 1$.
- The height of a full binary tree with n nodes is

  $$\log_2(n+1) - 1$$
- The height of a complete binary tree with n nodes is

  $$\text{floor}(\log_2 n)$$

Complete Binary Trees

Every level, except possibly the last, is completely filled, and all nodes are as far left as possible

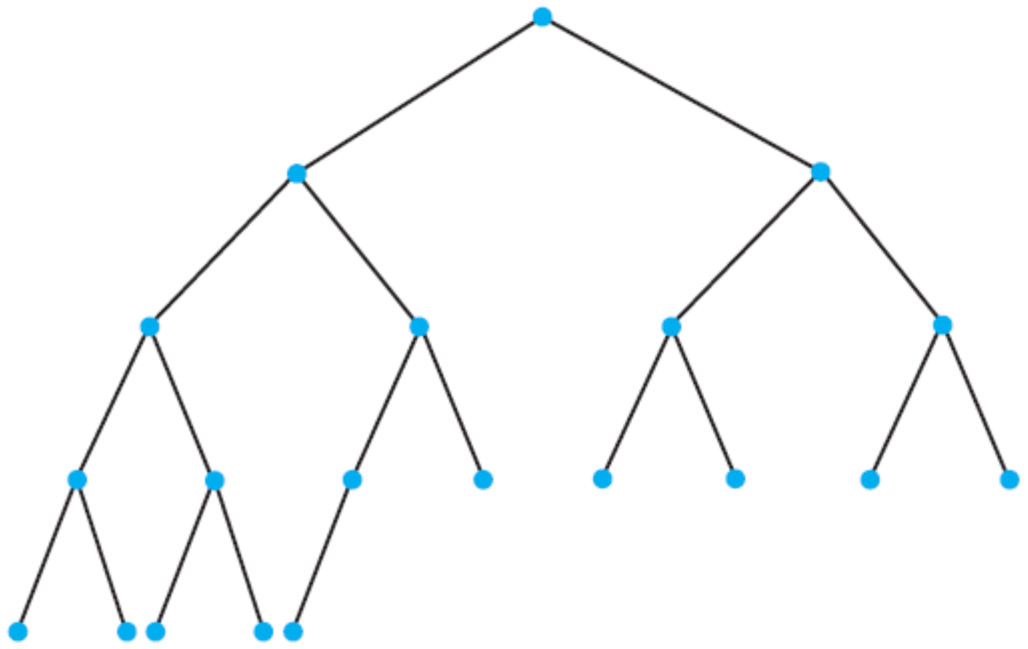
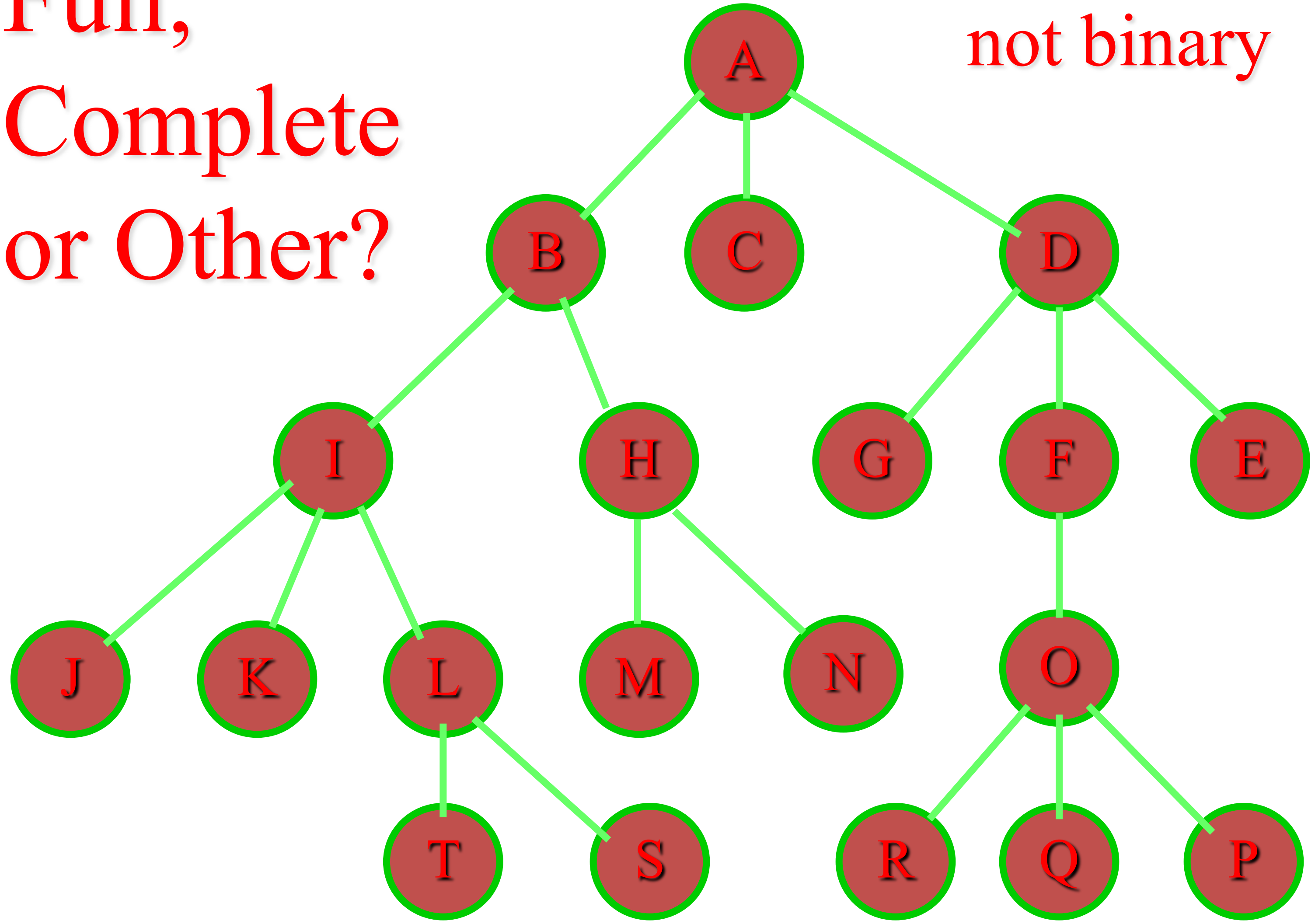


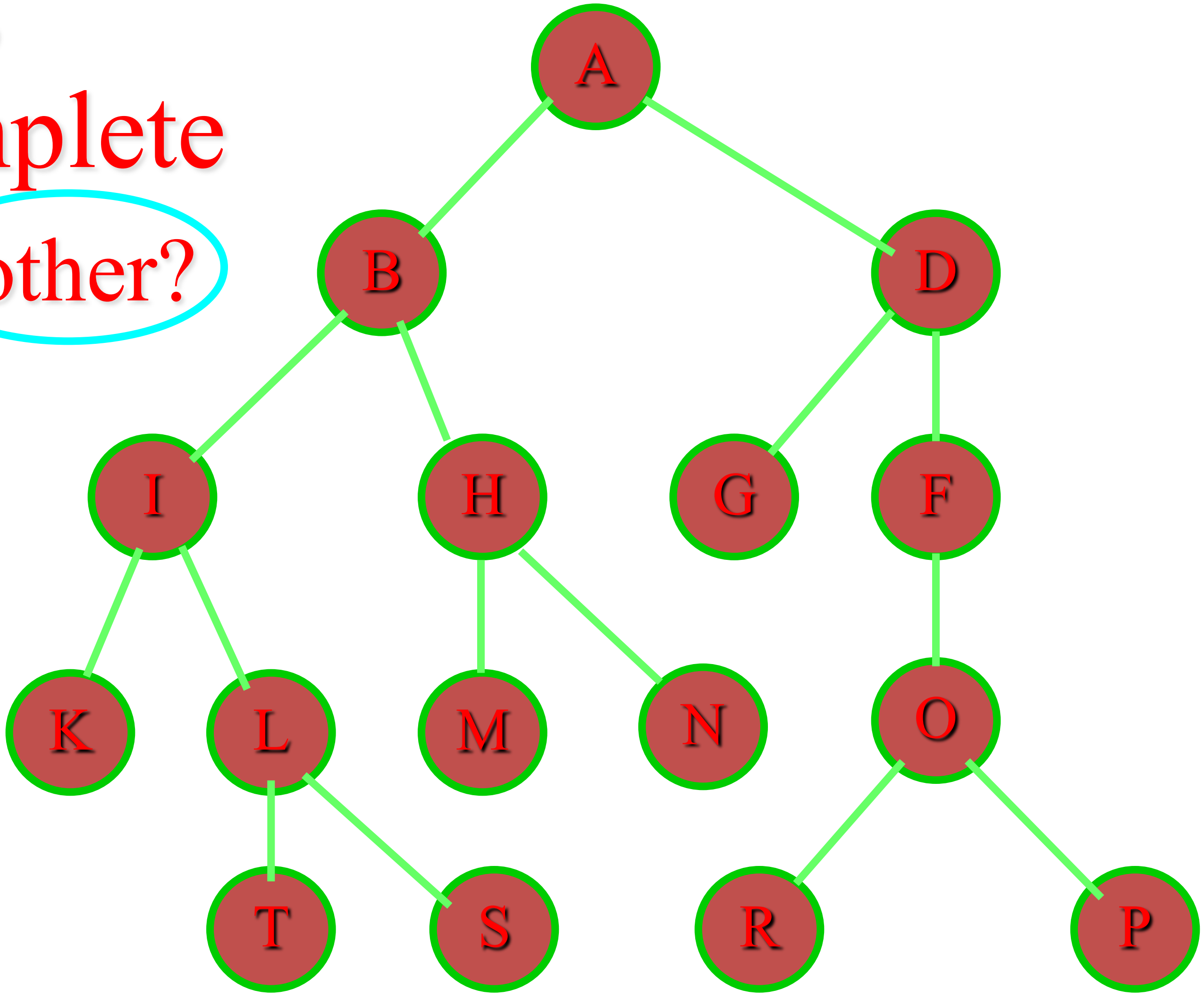
FIGURE 15-7 A complete binary tree

Full, Complete or Other?

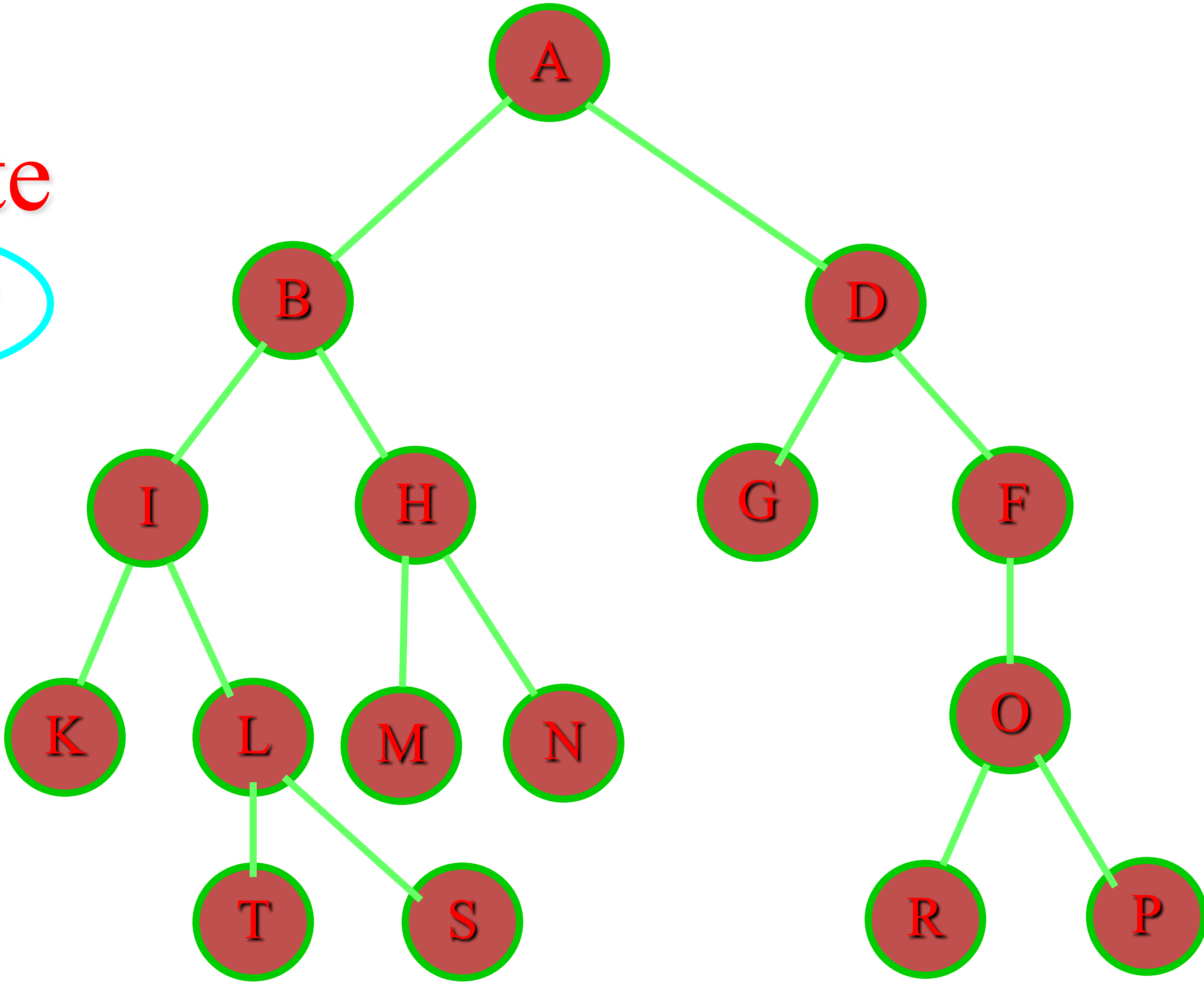
not binary



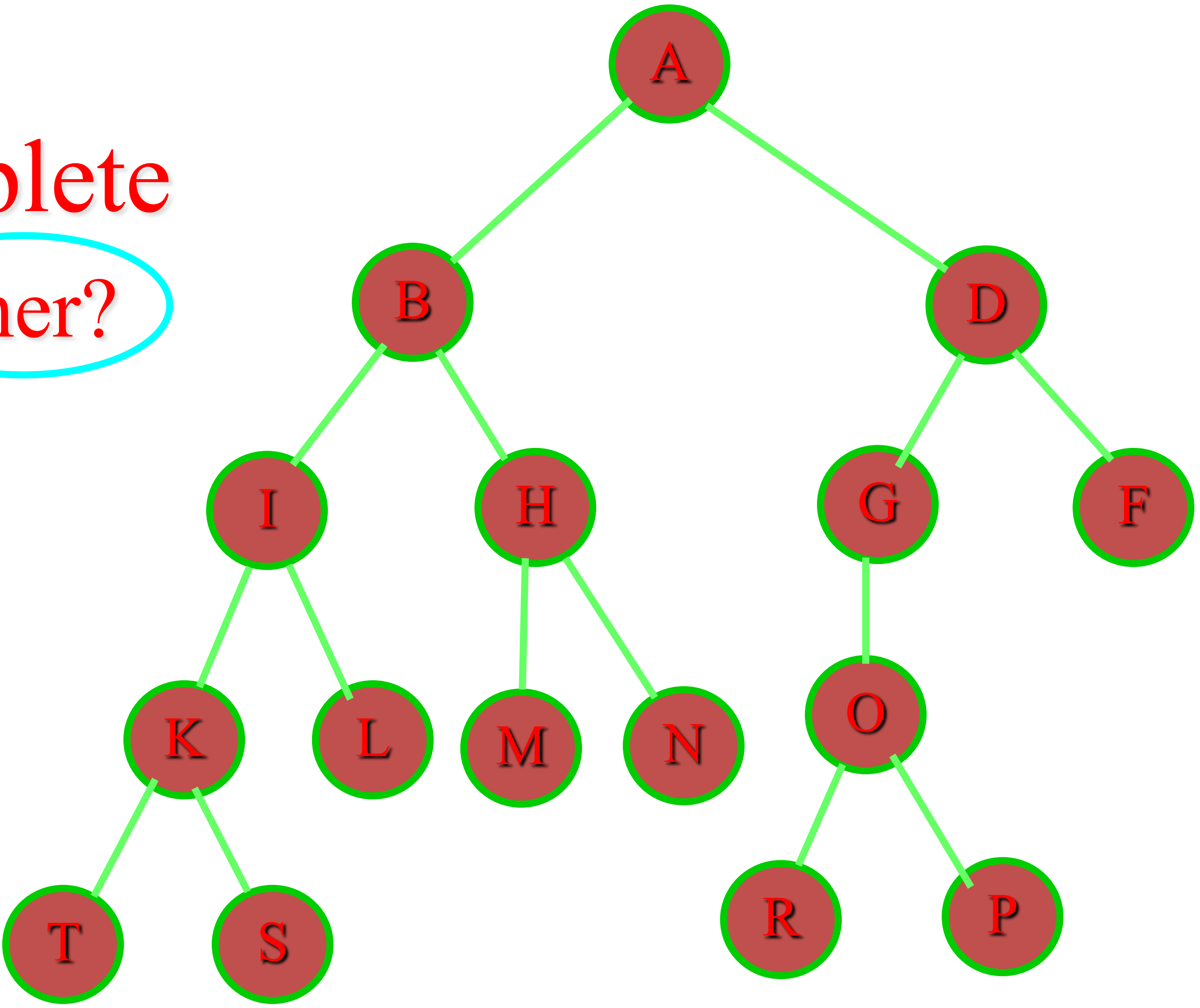
Full,
Complete
or [✓ other?]



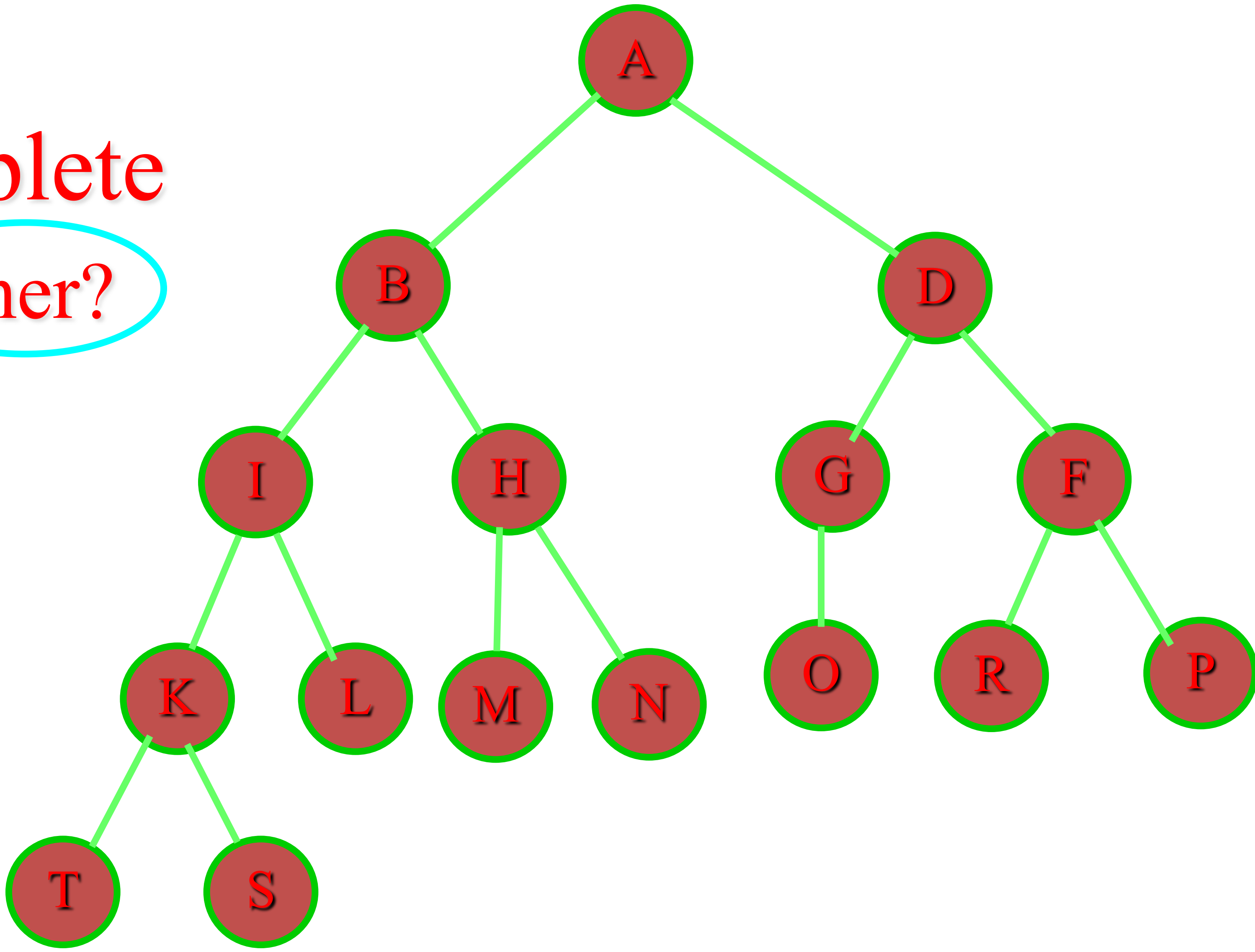
Full, Complete or [✓ other?]



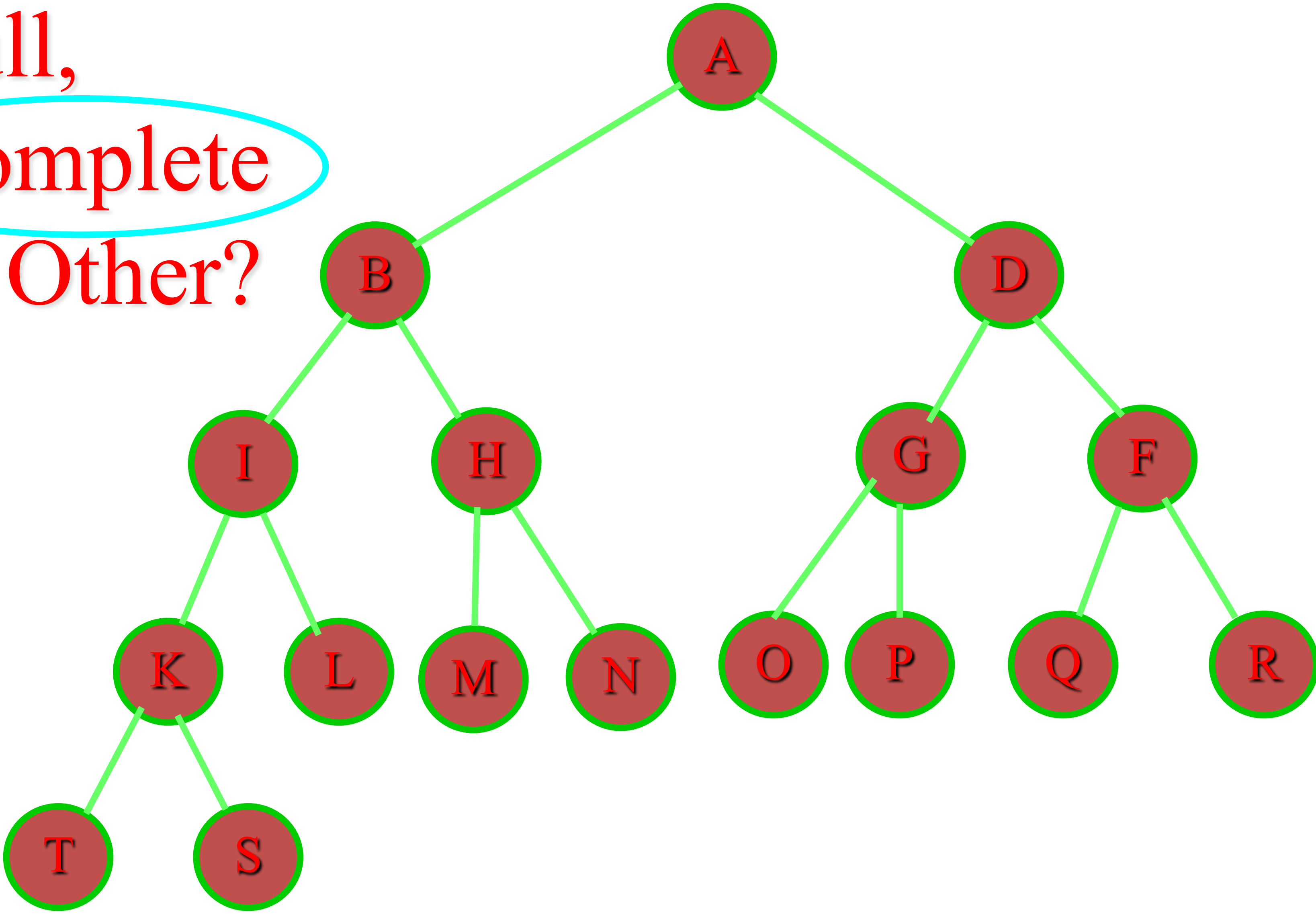
Full, Complete or [✓ other?]



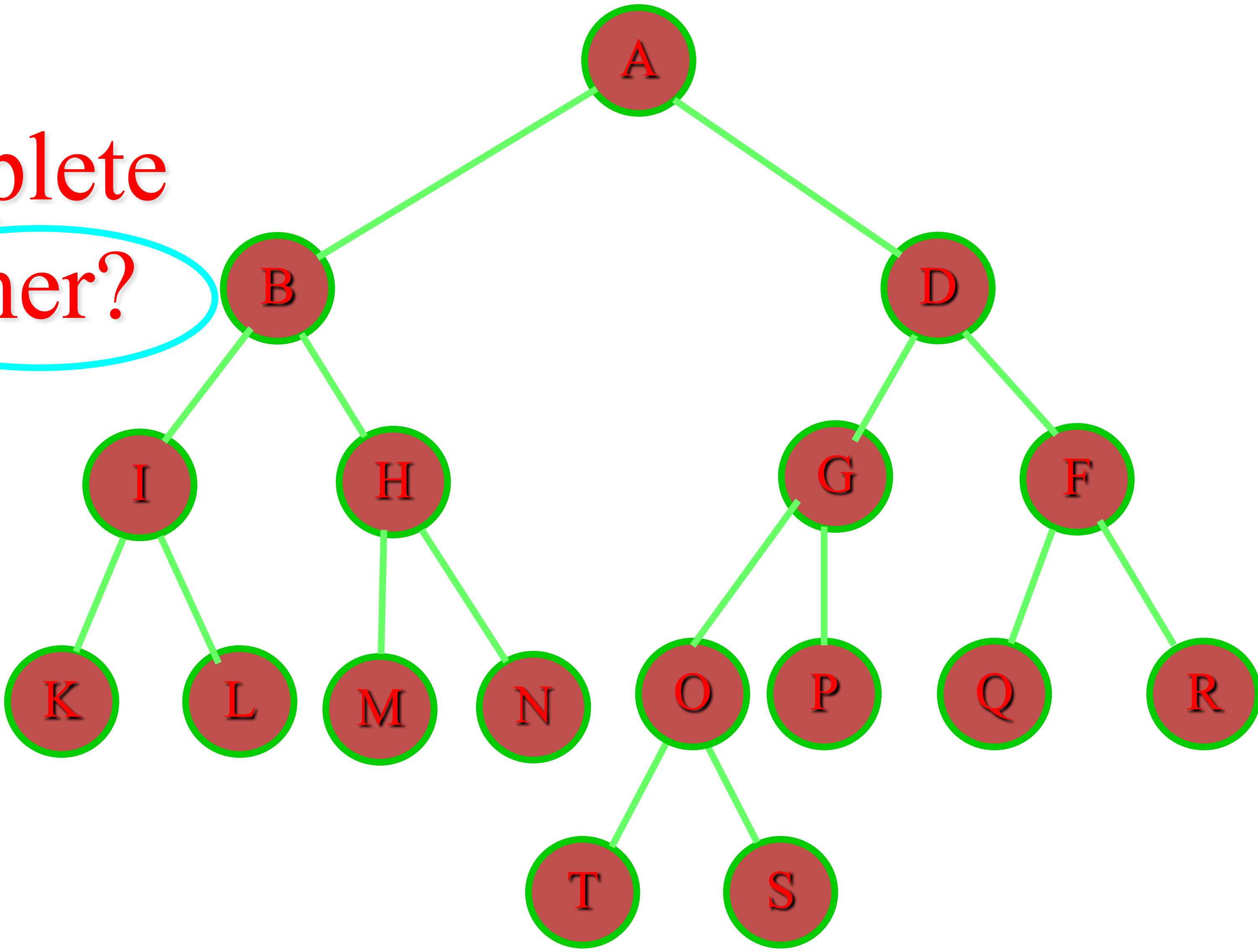
Full, Complete or [✓ other?]



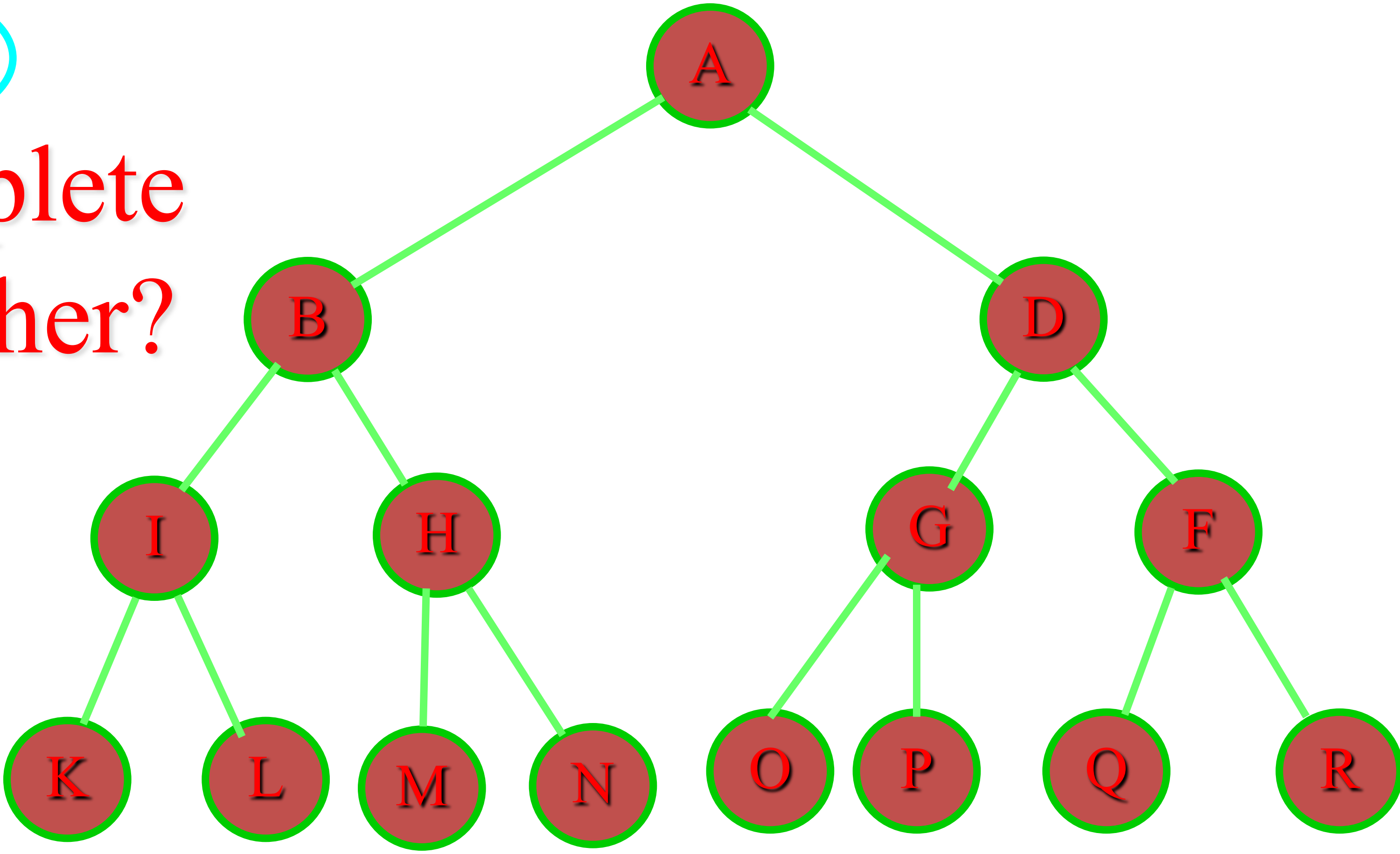
Full,
[✓ Complete]
or Other?



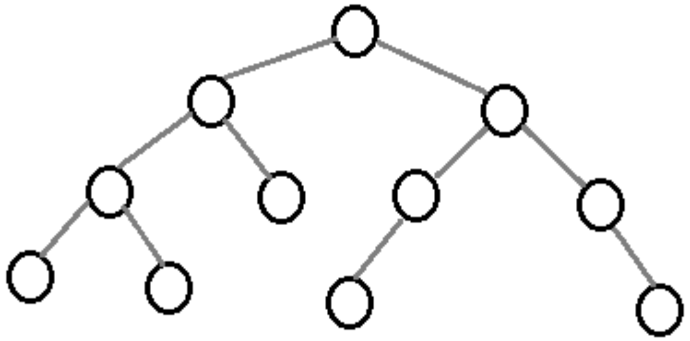
Full, Complete or other?

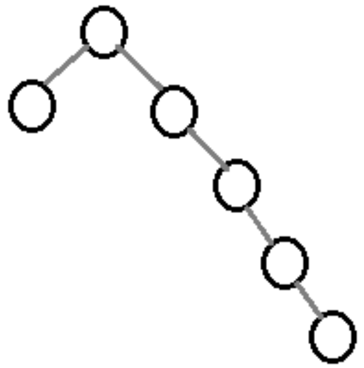


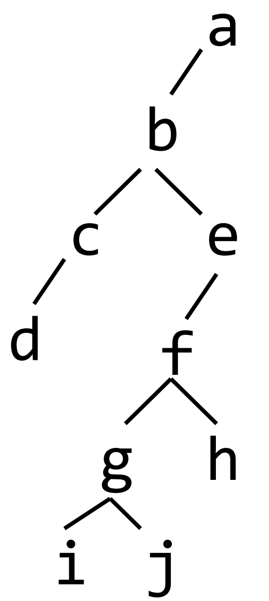
Full, Complete or Other?



- A balanced binary tree has the minimum possible height for the leaves

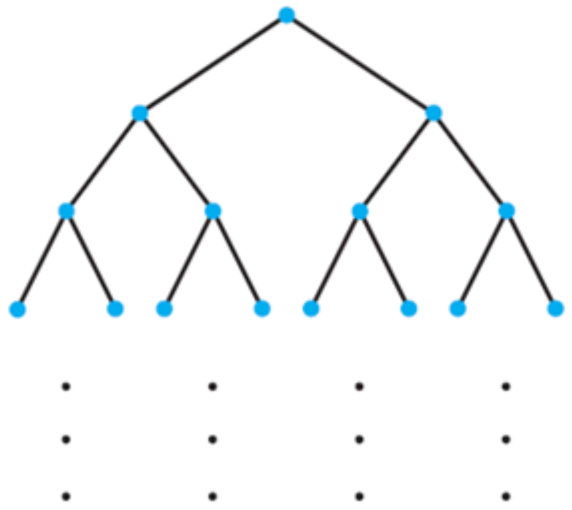

Balanced Binary Tree


Unbalanced Binary Tree



An unbalanced binary tree

Number of Nodes in a Binary Tree



Level	Number of nodes at this level	Total number of nodes at this level and all previous levels
1	$1 = 2^0$	$1 = 2^1 - 1$
2	$2 = 2^1$	$3 = 2^2 - 1$
3	$4 = 2^2$	$7 = 2^3 - 1$
4	$8 = 2^3$	$15 = 2^4 - 1$
⋮	⋮	⋮
h	2^{h-1}	2^h-1

depth: h-1

number of levels: h

Traversals of a Binary Tree

- General form of recursive traversal algorithm

1. **Preorder Traversal**

   Each node is processed before any node in either of its subtrees

2. **Inorder Traversal**

   Each node is processed after all nodes in its left subtree and before any node in its right subtree

3. **Postorder Traversal**

   Each node is processed after all nodes in both of its subtrees

Traversals of a Binary Tree

- **Preorder traversal** while duplicating nodes and values can make a complete duplicate of a binary tree. It can also be used to make a prefix expression (Polish notation) from expression trees: traverse the expression tree pre-orderly.

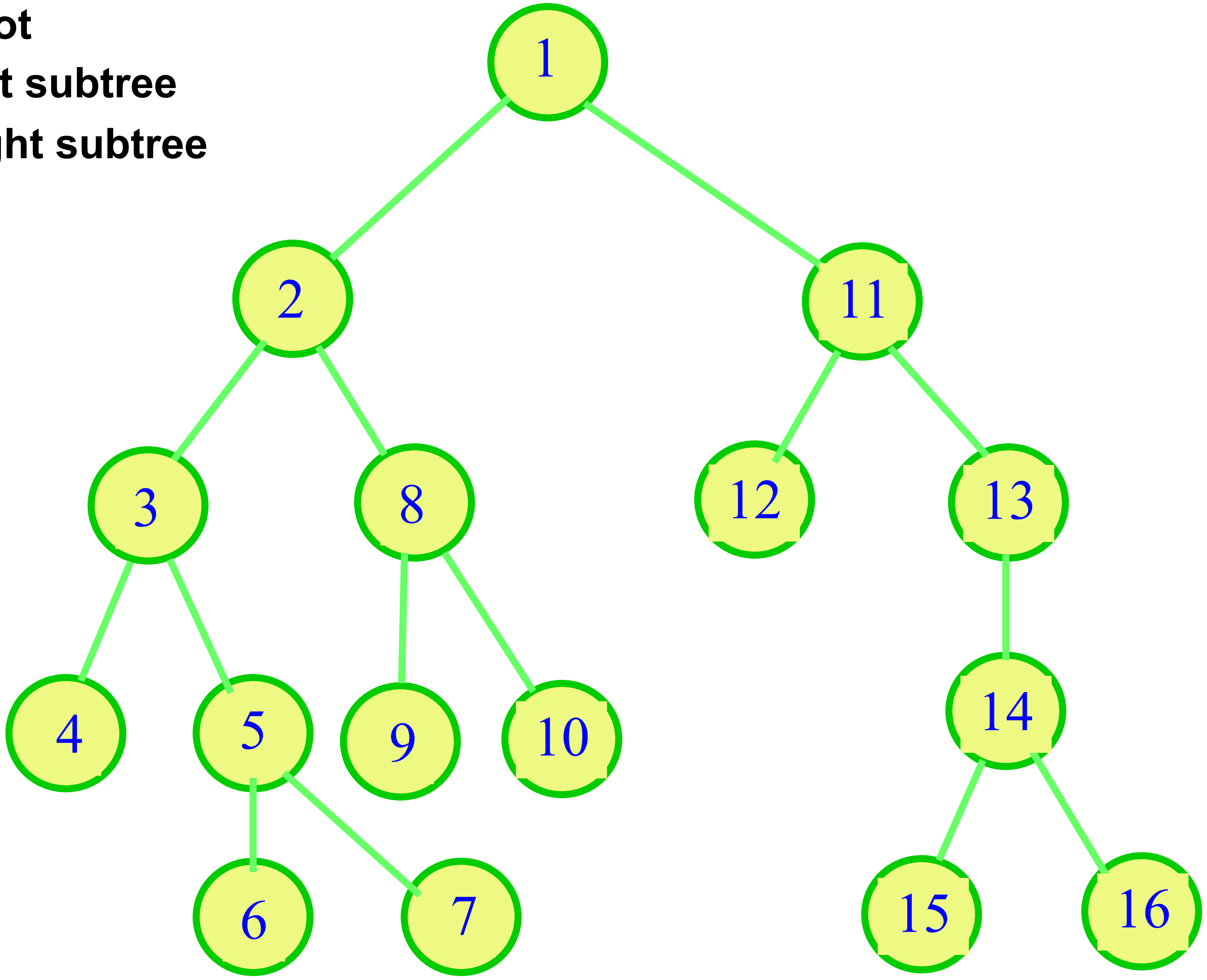
- **Inorder traversal** is very commonly used on binary search trees because it returns values from the underlying set in order, according to the comparator that set up the binary search tree (hence the name).

- **Postorder traversal** while deleting or freeing nodes and values can delete or free an entire binary tree. It can also generate a postfix representation of a binary tree.

It boils down to the logistical needs of an algorithm. For example, if you don't use post-order traversal during deletion, then you lose the references you need for deleting the child trees.

Preorder Traversal

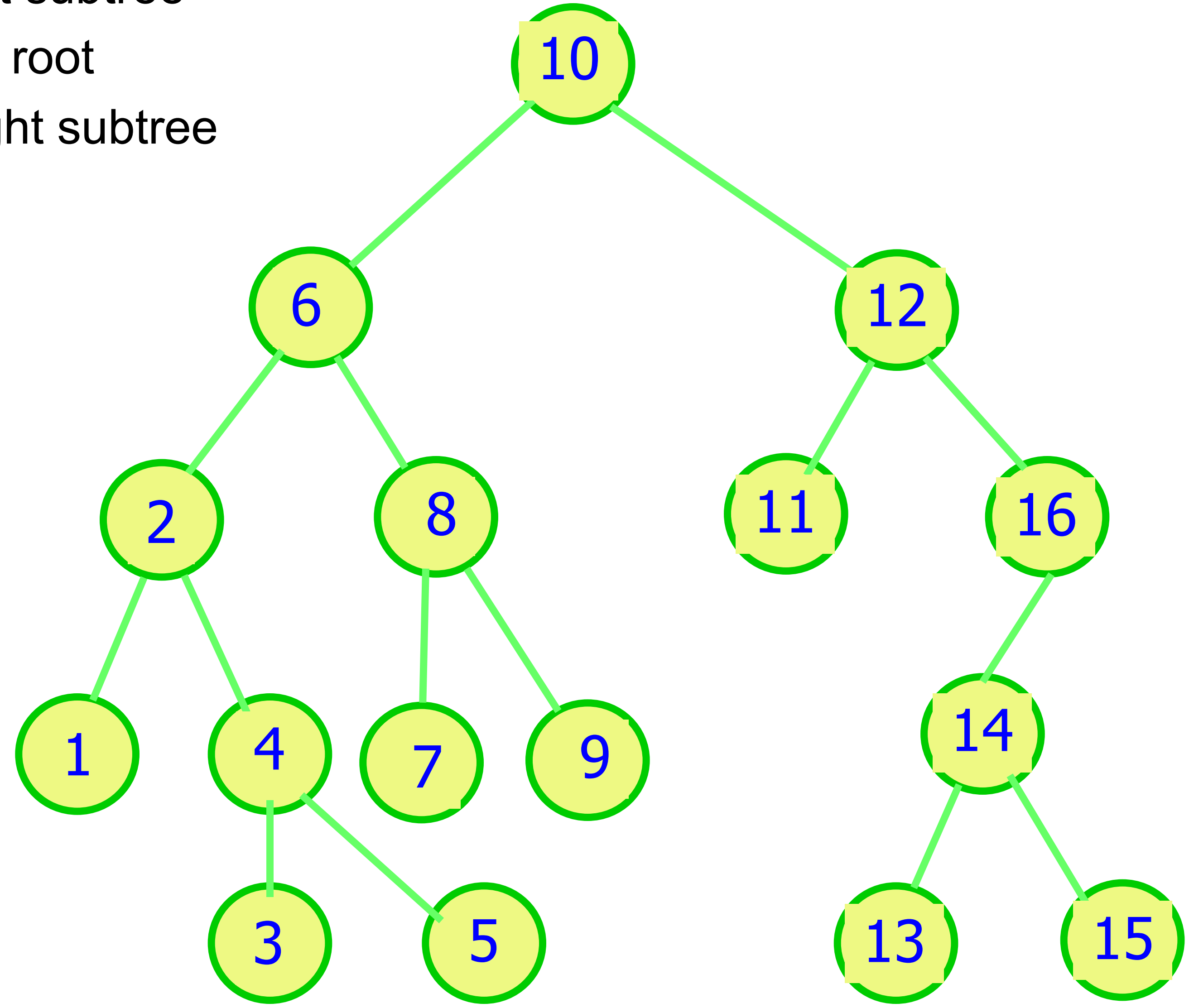
1. Visit the root
2. Visit the left subtree
3. Visit the right subtree



```
Algorithm TraversePreorder(n)
   Process node n
   if n is an internal node then
      TraversePreorder( n -> leftChild)
      TraversePreorder( n -> rightChild)
```

Inorder Traversal

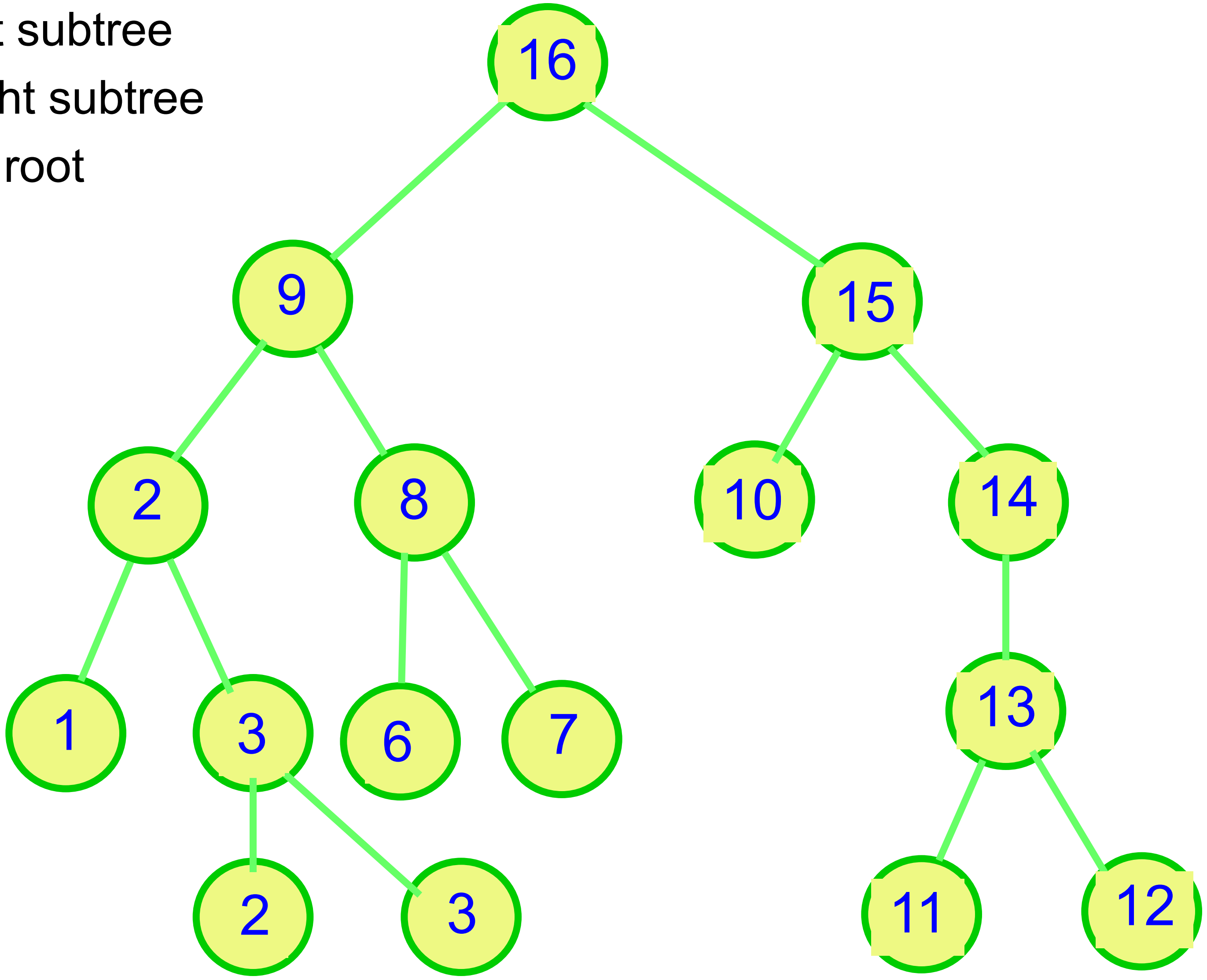
1. Visit Left subtree
2. Visit the root
3. Visit Right subtree



```
Algorithm TraverseInorder(n)
   if n is an internal node then
      TraverseInorder( n -> leftChild)
   Process node n
   if n is an internal node then
      TraverseInorder( n -> rightChild)
```

Postorder Traversals

1. Visit Left subtree
2. Visit Right subtree
3. Visit the root



```
Algorithm TraversePostorder(n)
    if n is an internal node then
        TraversePostorder( n -> leftChild)
        TraversePostorder( n -> rightChild)
    Process node n
```

Traversals of a Binary Tree

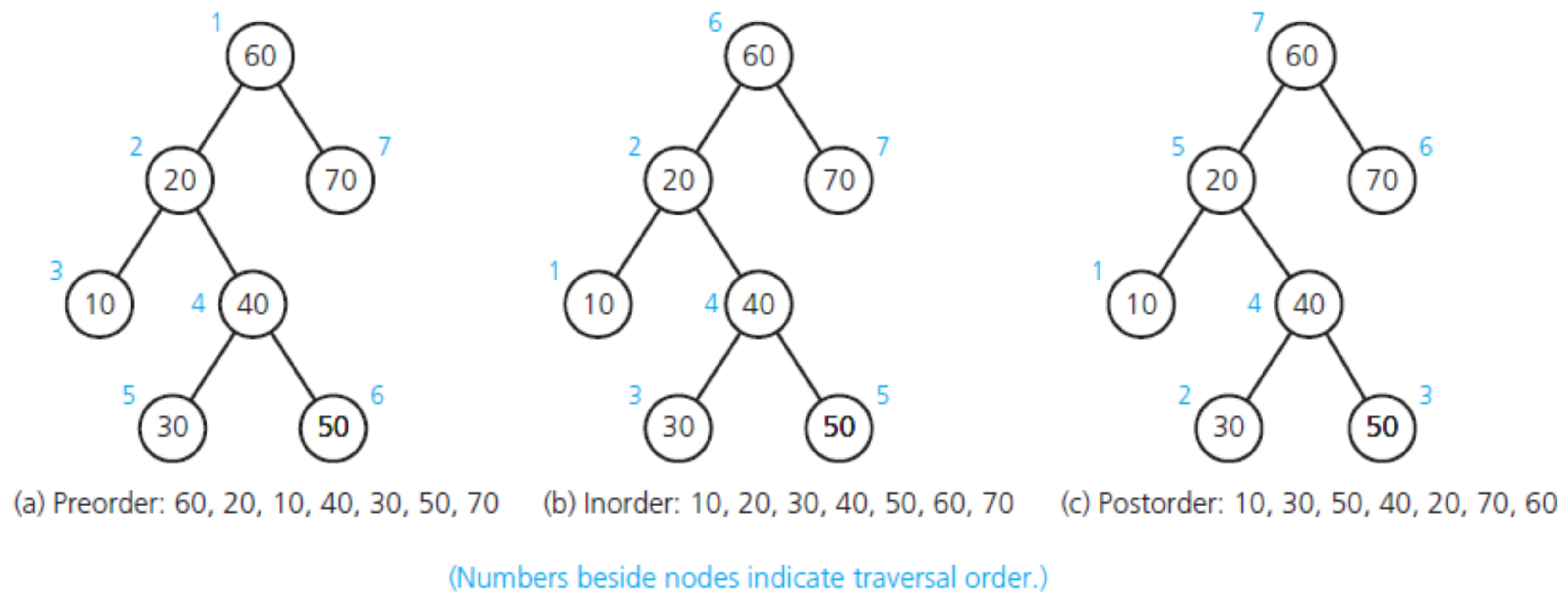
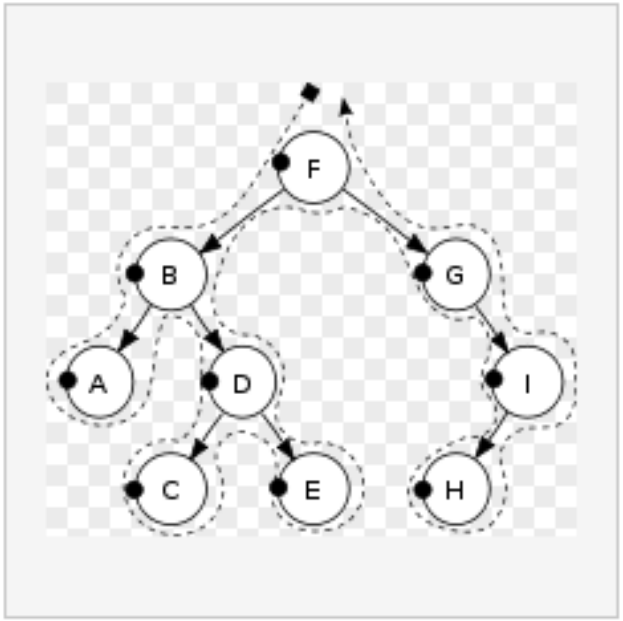
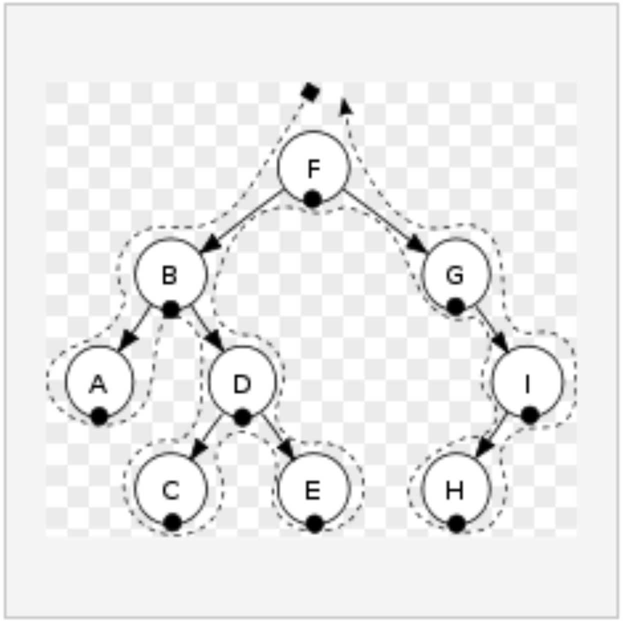

1 60
2 20
7 70
3 10
4 40
5 30
6 50
(a) Preorder: 60, 20, 10, 40, 30, 50, 70
6 60
2 20
7 70
1 10
4 40
3 30
5 50
(b) Inorder: 10, 20, 30, 40, 50, 60, 70
7 60
5 20
6 70
1 10
4 40
2 30
3 50
(c) Postorder: 10, 30, 50, 40, 20, 70, 60
(Numbers beside nodes indicate traversal order.)

FIGURE 15-11 Three traversals of a binary tree

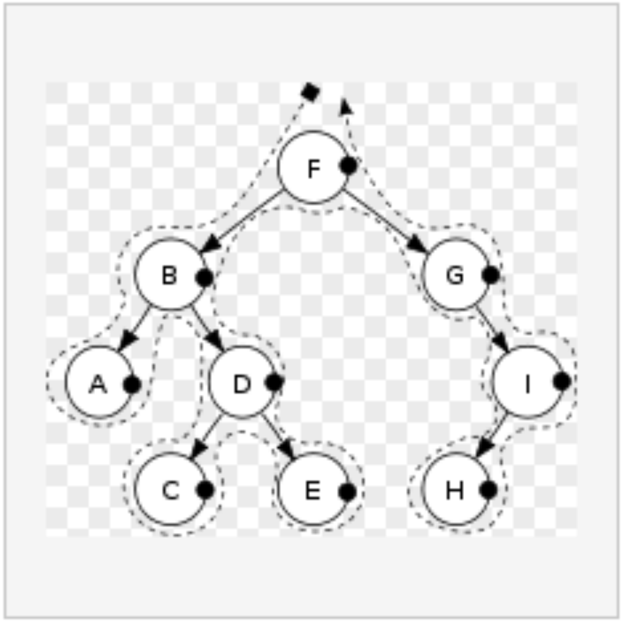
The 3 different types of traversal



Pre-order Traversal
FBADCEGIH



In-order Traversal
ABCDEFGHI



Post-order Traversal
ACEDBHIGF

Binary Tree Operations

- Test whether a binary tree is empty.
- Get the height of a binary tree.
- Get the number of nodes in a binary tree.
- Get the data in a binary tree’s root.
- Set the data in a binary tree’s root.
- Add a new node containing a given data item to a binary tree.

Binary Tree Operations

- Remove the node containing a given data item from a binary tree.
- Remove all nodes from a binary tree.
- Retrieve a specific entry in a binary tree.
- Test whether a binary tree contains a specific entry.
- Traverse the nodes in a binary tree in preorder, inorder, or postorder.

Represention of Binary Tree ADT

A binary tree can be represented using

- **Linked List**
- **Array**

Note : Array is suitable only for full and complete binary trees

```
struct node
{
  int key_value;
  struct node *left;
  struct node *right;
};

struct node *root = 0;
```

```
void preorder(node *p)
{
    if (p != NULL)
    {
        printf(p->key_value);
        preorder(p->left);
        preorder(p->right);
    }
}
```

```
void inorder(node *p)
{
    if (p != NULL)
    {
        inorder(p->left);
        printf(p->key_value);
        inorder(p->right);
    }
}
```

```
void postorder(node *p)
{
    if (p != NULL)
    {
        postorder(p->left);
        postorder(p->right);
        printf(p->key_value);
    }
}
```

```
void destroy_tree(struct node *leaf)
{
  if( leaf != NULL )
  {
      destroy_tree(leaf->left);
      destroy_tree(leaf->right);
      free( leaf );
  }
}
```

The Binary Search Tree

- Binary tree is ill suited for searching a specific item
- Binary *search* tree solves the problem
- Properties of each node, *n*
  - *n*'s value is greater than all values in the left subtree T_L
  - *n*'s value is less than all values in the right subtree T_R
  - Both T_R and T_L are binary search trees.

The Binary Search Tree

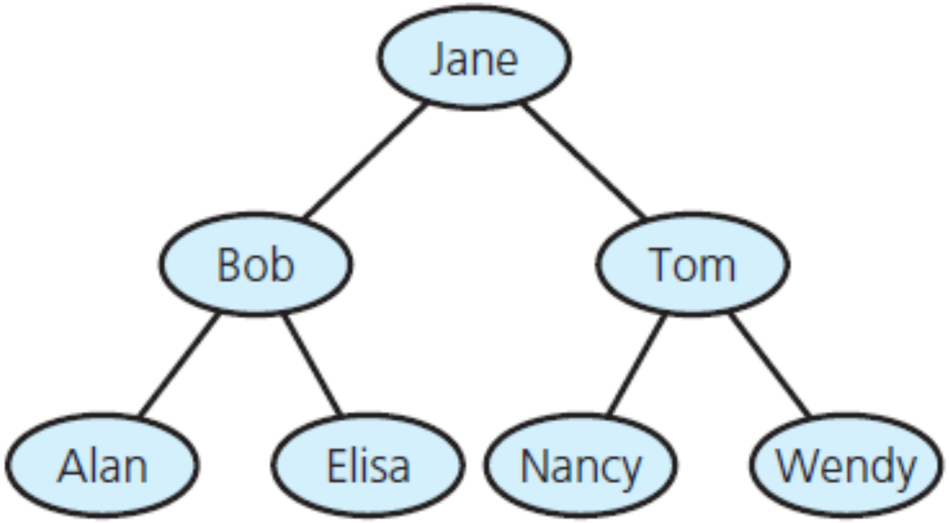


FIGURE 15-13 A binary search tree of names

The Binary Search Tree

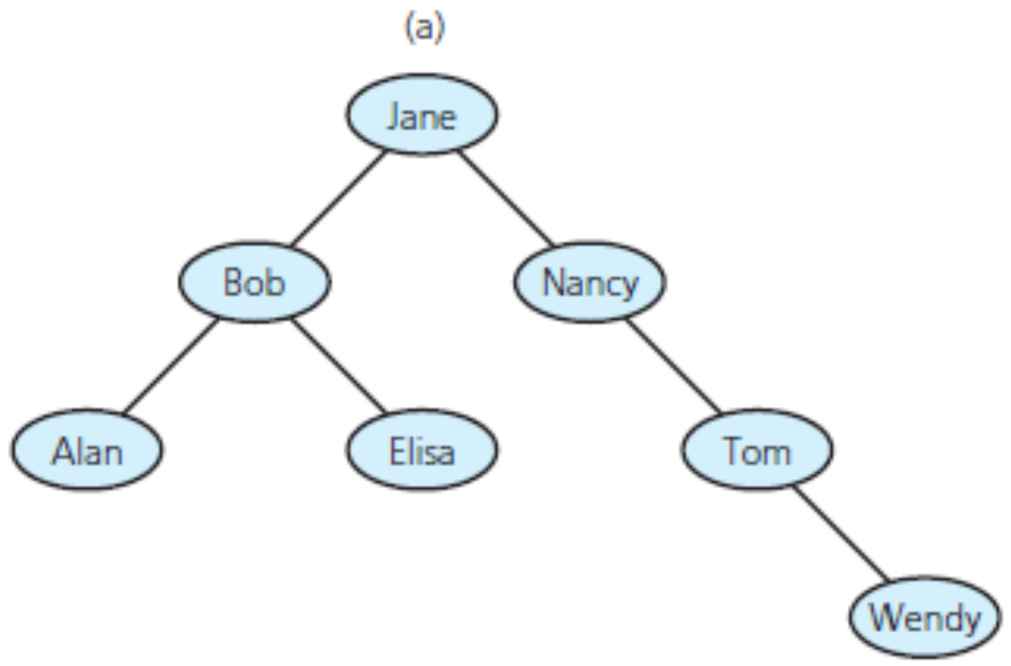


FIGURE 15-14 Binary search trees with the same data as in Figure 15-13

The Binary Search Tree

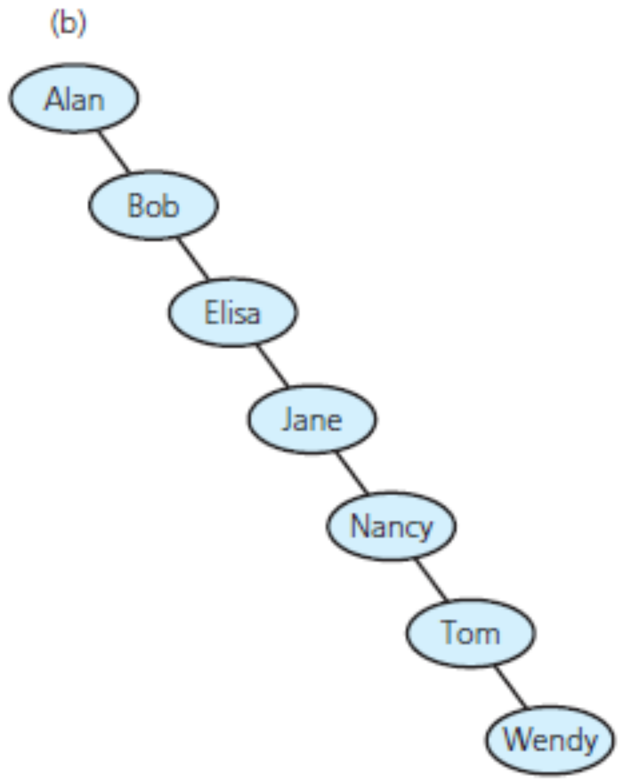
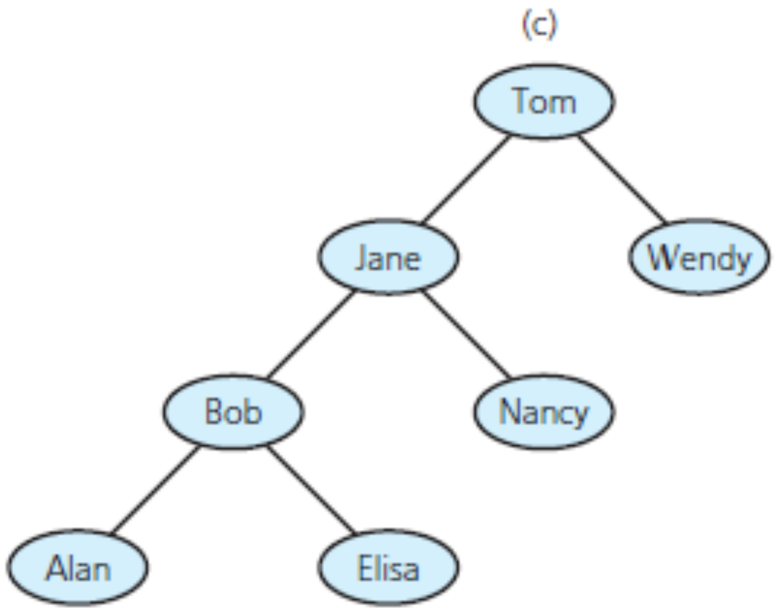


FIGURE 15-14 Binary search trees with the same data as in Figure 15-13

The Binary Search Tree



Binary search trees with the same data as in Figure 15-13

Binary Search Tree Operations

- Test whether a binary search tree is empty.
- Get height of a binary search tree.
- Get number of nodes in a binary search tree.
- Get data in binary search tree's root.
- Insert new item into the binary search tree.
- Remove given item from the binary search tree.

Binary Search Tree Operations

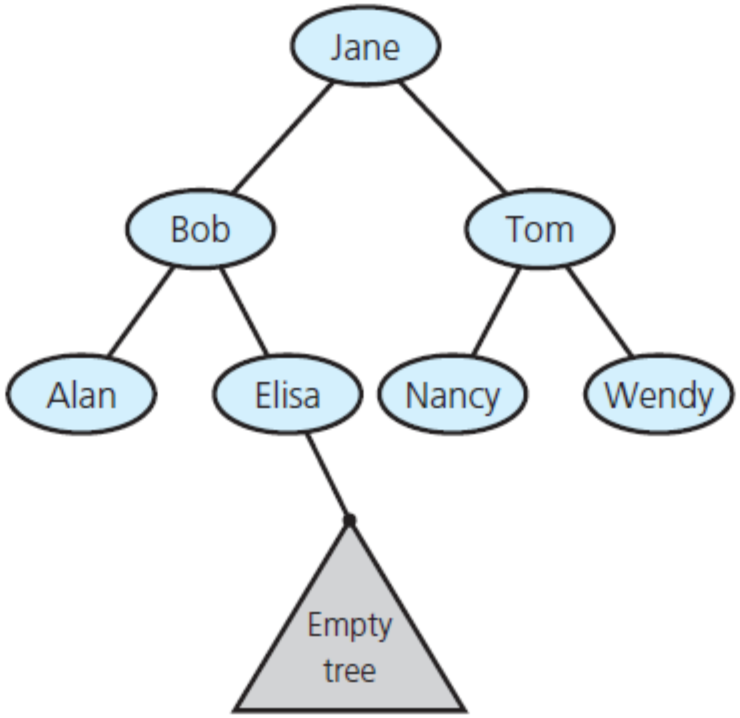
- Remove all entries from a binary search tree.
- Retrieve given item from a binary search tree.
- Test whether a binary search tree contains a specific entry.
- Traverse items in a binary search tree in
  - Preorder
  - Inorder
  - Postorder.

Searching a Binary Search Tree

- Search algorithm for binary search tree

```
struct node* search(int key, struct node *leaf)
{
  if( leaf != NULL )
  {
     if( key == leaf->key_value )
     {
        return leaf;
     }
     else if( key < leaf->key_value )
     {
        return search(key, leaf->left);
     }
     else
     {
        return search(key, leaf->right);
     }
  }
  else return 0;
}
```

Creating a Binary Search Tree



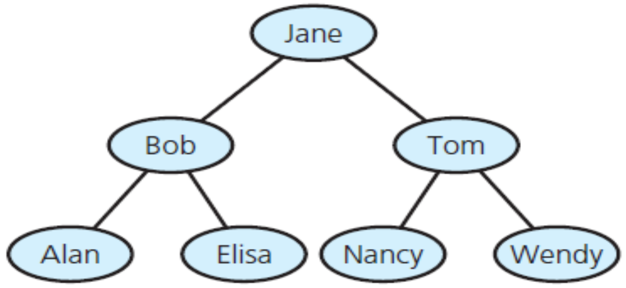
Empty subtree where the `search` algorithm terminates when looking for Frank

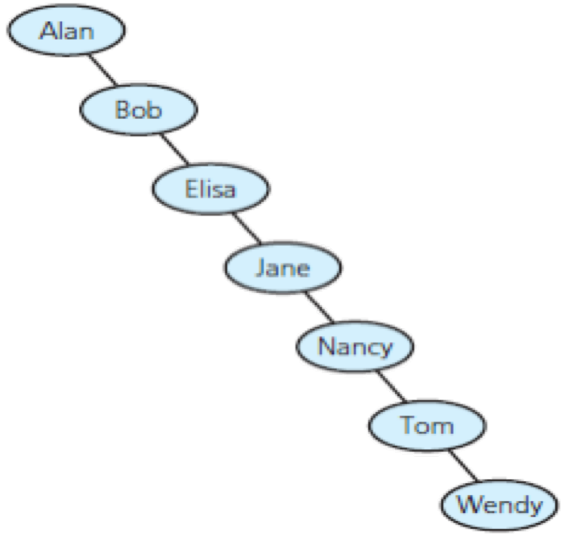
```
struct node *newNode(int key) {
   struct node *temp =  (struct node *)malloc(sizeof(struct node));
   temp->key = key;
   temp->left = temp->right = NULL;
   return temp;
}

struct node* insert(int key, struct node *leaf)
{
   /* If empty, return a new node */
   if (leaf == NULL) return newNode(key);

   /* Otherwise, recur down the tree */
   if (key < leaf->key)
      leaf->left  = insert(leaf->left, key);
   else if (key > leaf->key)
      leaf->right = insert(leaf->right, key);

   /* return the (unchanged) node pointer */
   return leaf;
}
```

Efficiency of Binary Search Tree Operations

Operation	Average case	Worst case
Retrieval	O(log *n*)	O(*n*)
Insertion	O(log *n*)	O(*n*)
Removal	O(log *n*)	O(*n*)
Traversal	O(*n*)	O(*n*)

The Big O for the retrieval, insertion, removal, and traversal operations of the ADT binary search tree


Jane
Bob
Tom
Alan
Elisa
Nancy
Wendy


Alan
Bob
Elisa
Jane
Nancy
Tom
Wendy