BBM 201
Data structures

Trees

2015-2016 Fall

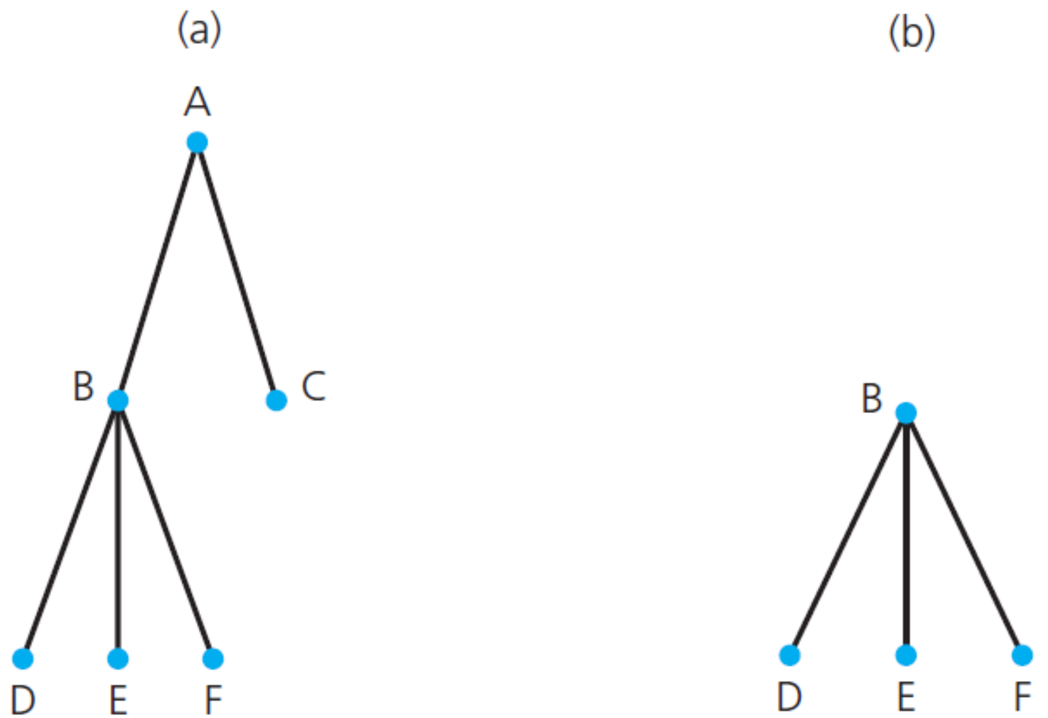
Content

- Terminology
- The ADT Binary Tree
- The ADT Binary Search Tree

Terminology

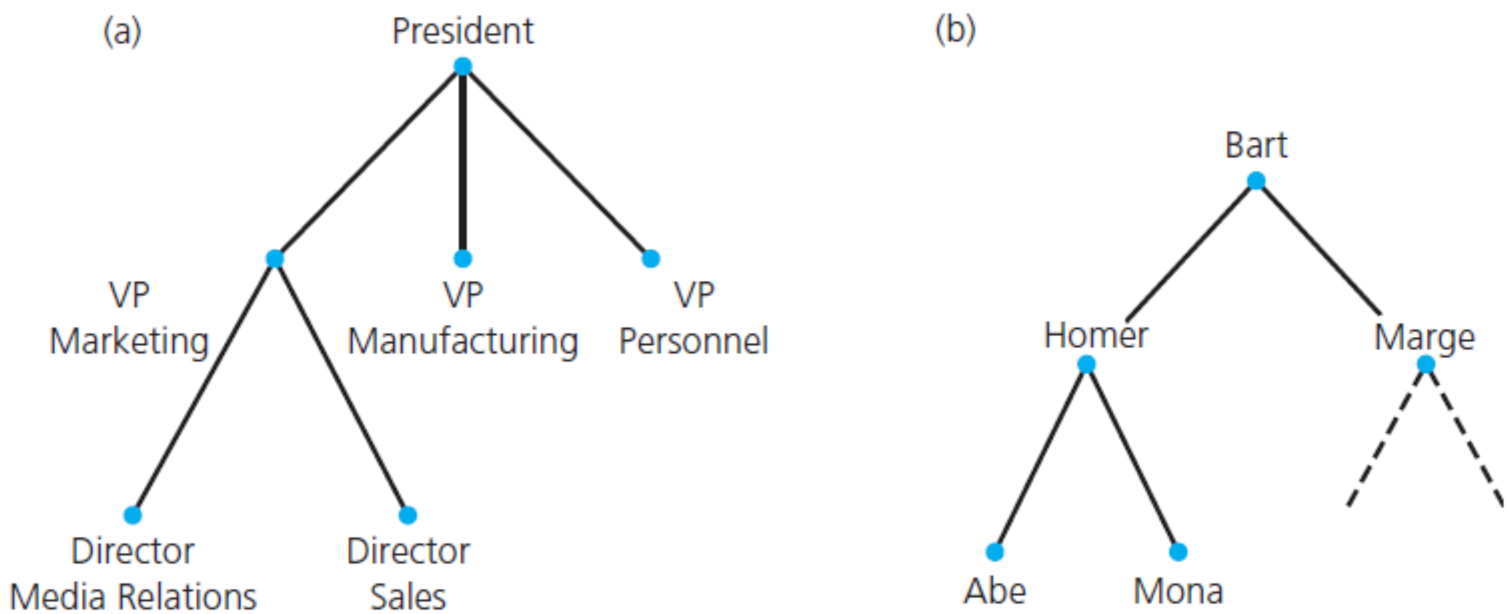
- Use trees to represent relationships
- Trees are hierarchical in nature
  - "Parent-child" relationship exists between nodes in tree.
  - Generalized to ancestor and descendant
  - Lines between the nodes are called edges
- A subtree in a tree is any node in the tree together with all of its descendants

Terminology



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

Terminology



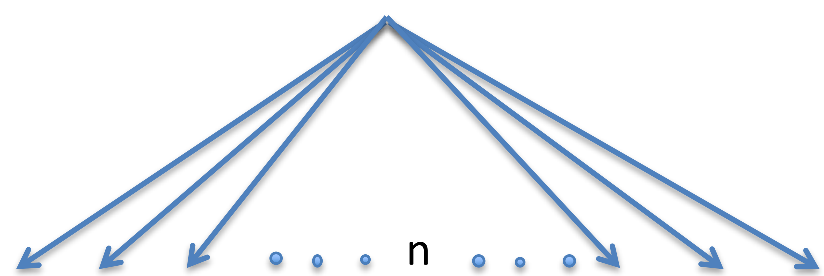
(a) An organization chart; (b) a family tree

Kinds of Trees

- General Tree
  - Set T of one or more nodes such that T is partitioned into disjoint subsets
  - A single node r , the root
  - Sets that are general trees, called subtrees of r

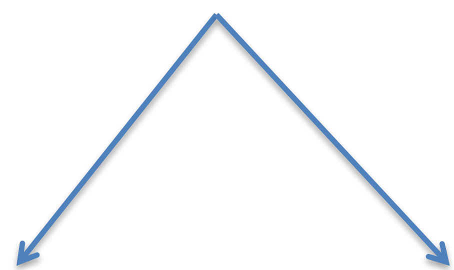
Kinds of Trees

- n -ary tree
  - set T of nodes that is either empty or partitioned into disjoint subsets:
  - A single node r , the root
  - n possibly empty sets that are n -ary subtrees of r

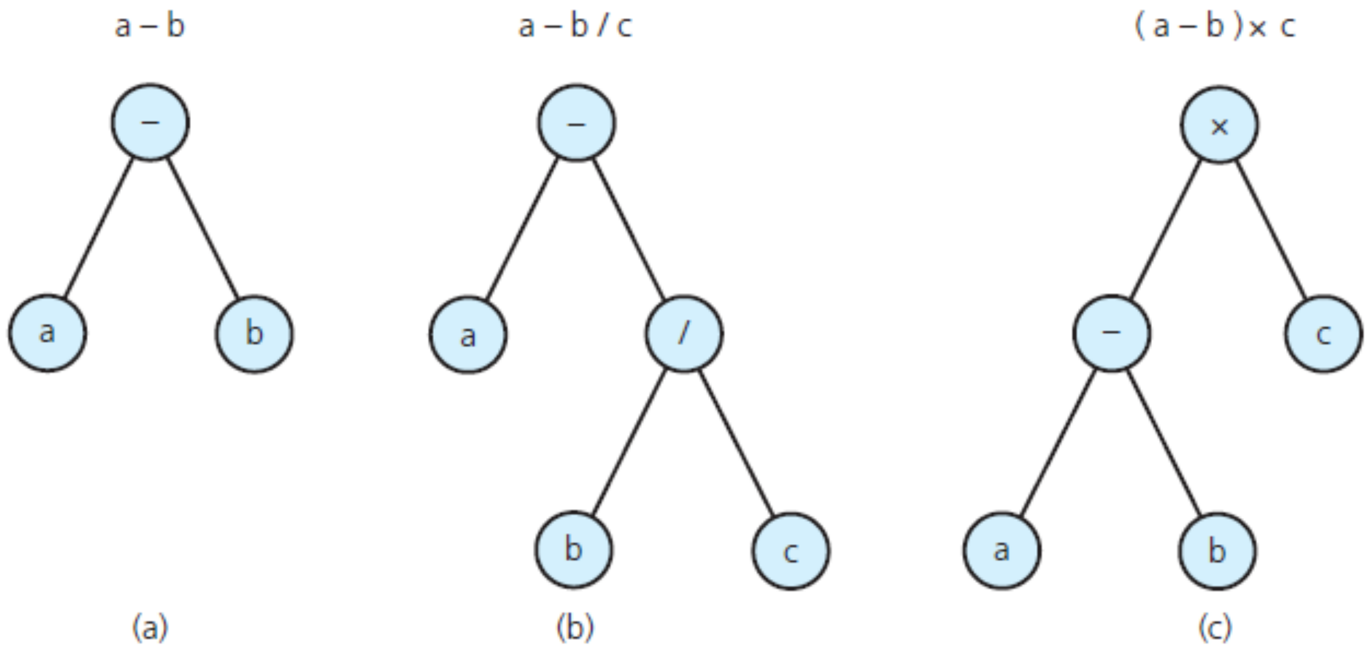


Kinds of Trees

- Binary tree
  - Set T of nodes that is either empty or partitioned into disjoint subsets
  - Single node r , the root
  - Two possibly empty sets that are binary trees, called left and right subtrees of r



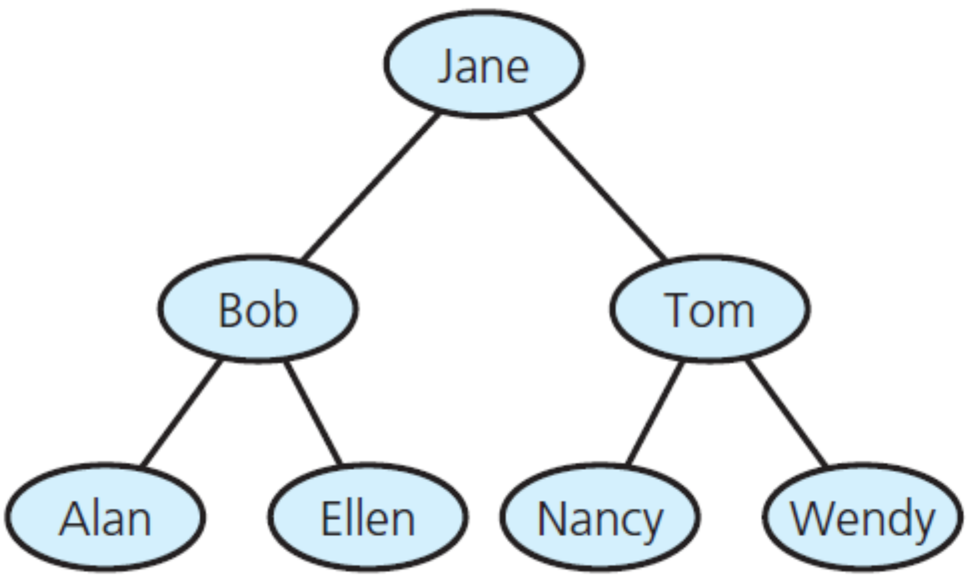
Example: Algebraic Expressions.



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.



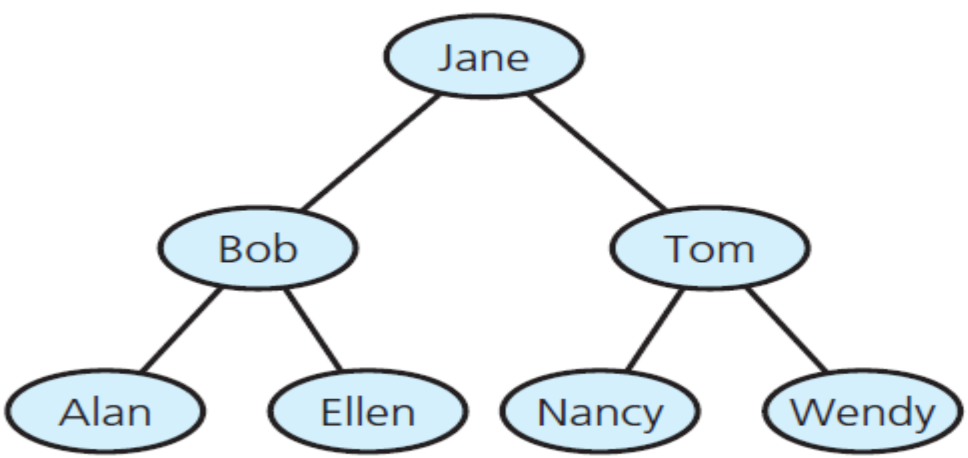
Level = 1

Level = 2

Level = 3

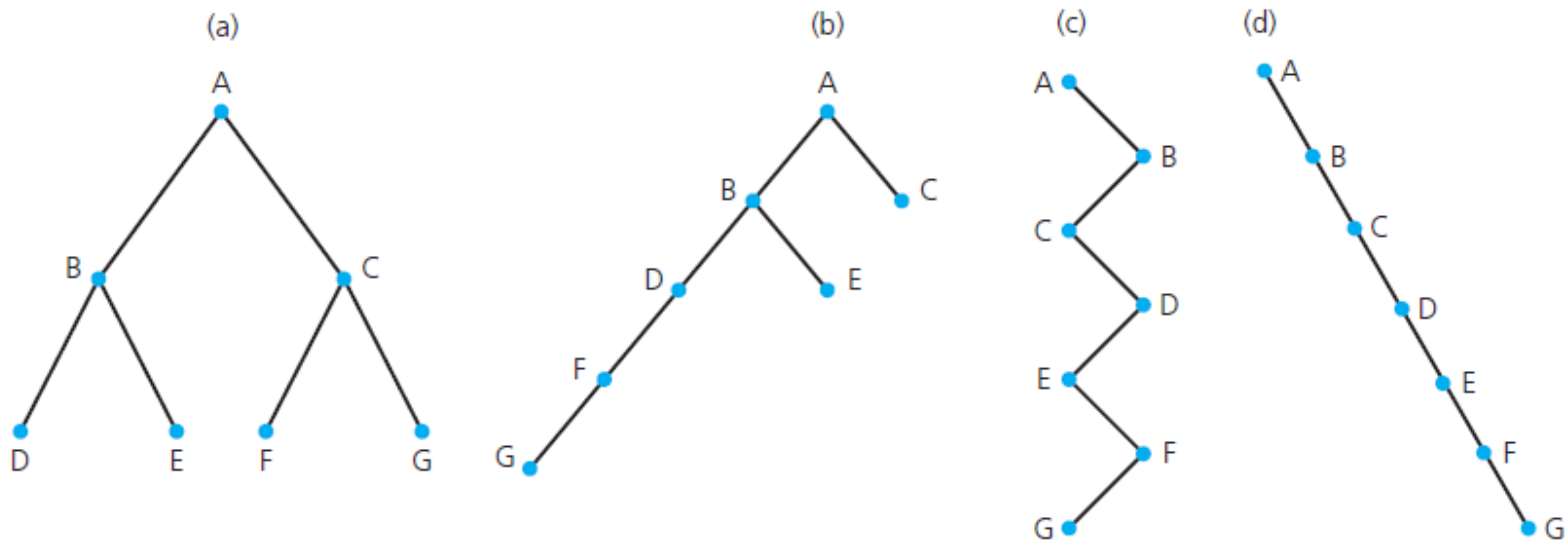
Height of Trees

- Height of a tree T in terms of the levels of its nodes
  - If T is empty, its height is -1.
  - The height of a node is the number of edges from the node to the deepest leaf.



Height of tree = 2

The Height of Trees



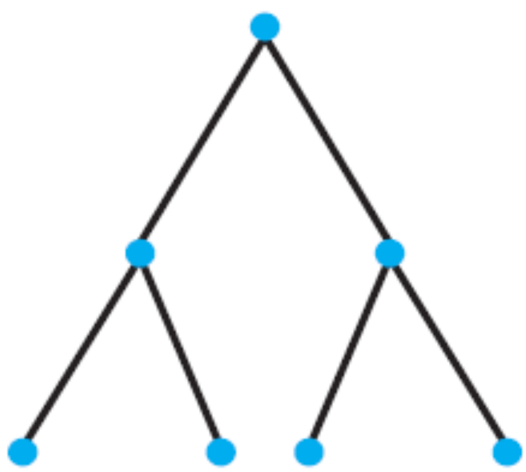
Height 2 Height 4 Height 6 Height 6

Binary trees with the same nodes but different heights

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 -1.
  - If T is not empty and has height h > -1, 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.



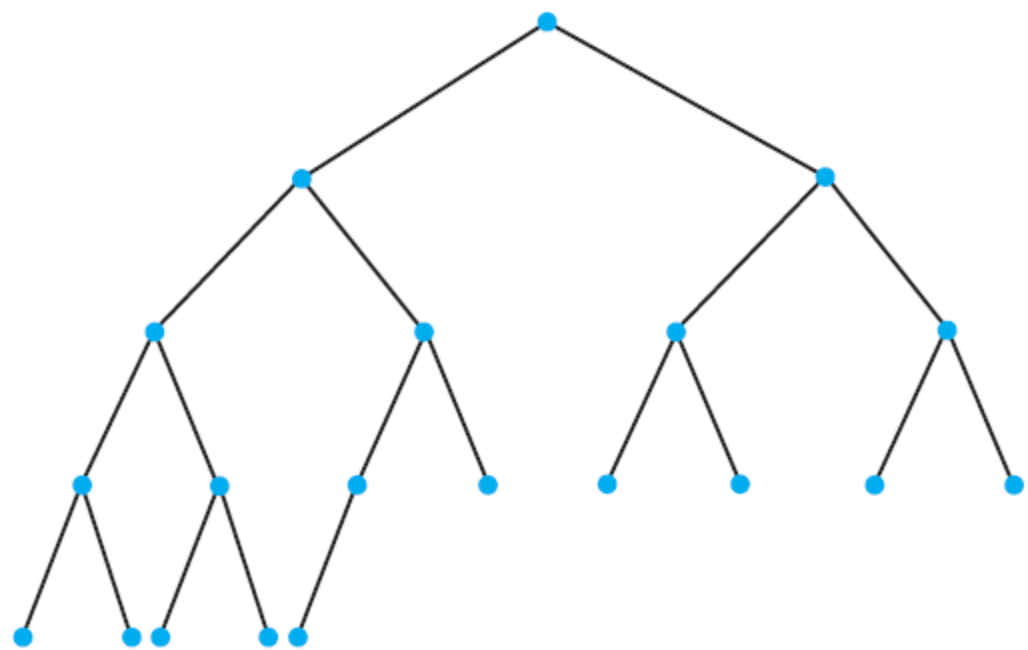
Facts about Full Binary Trees

- 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 $\mathrm{floor}(\log_2 n)$

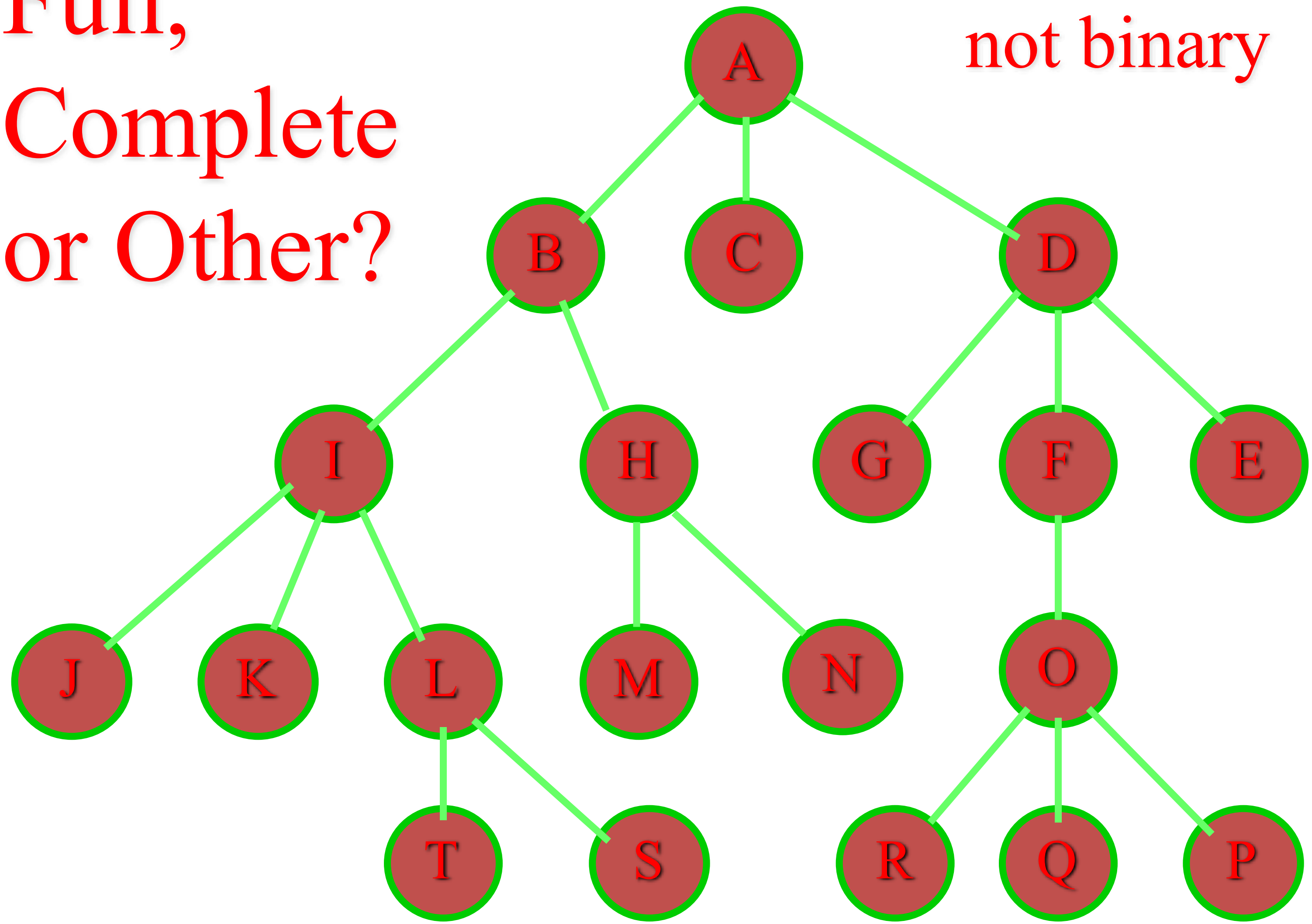
Complete Binary Trees

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

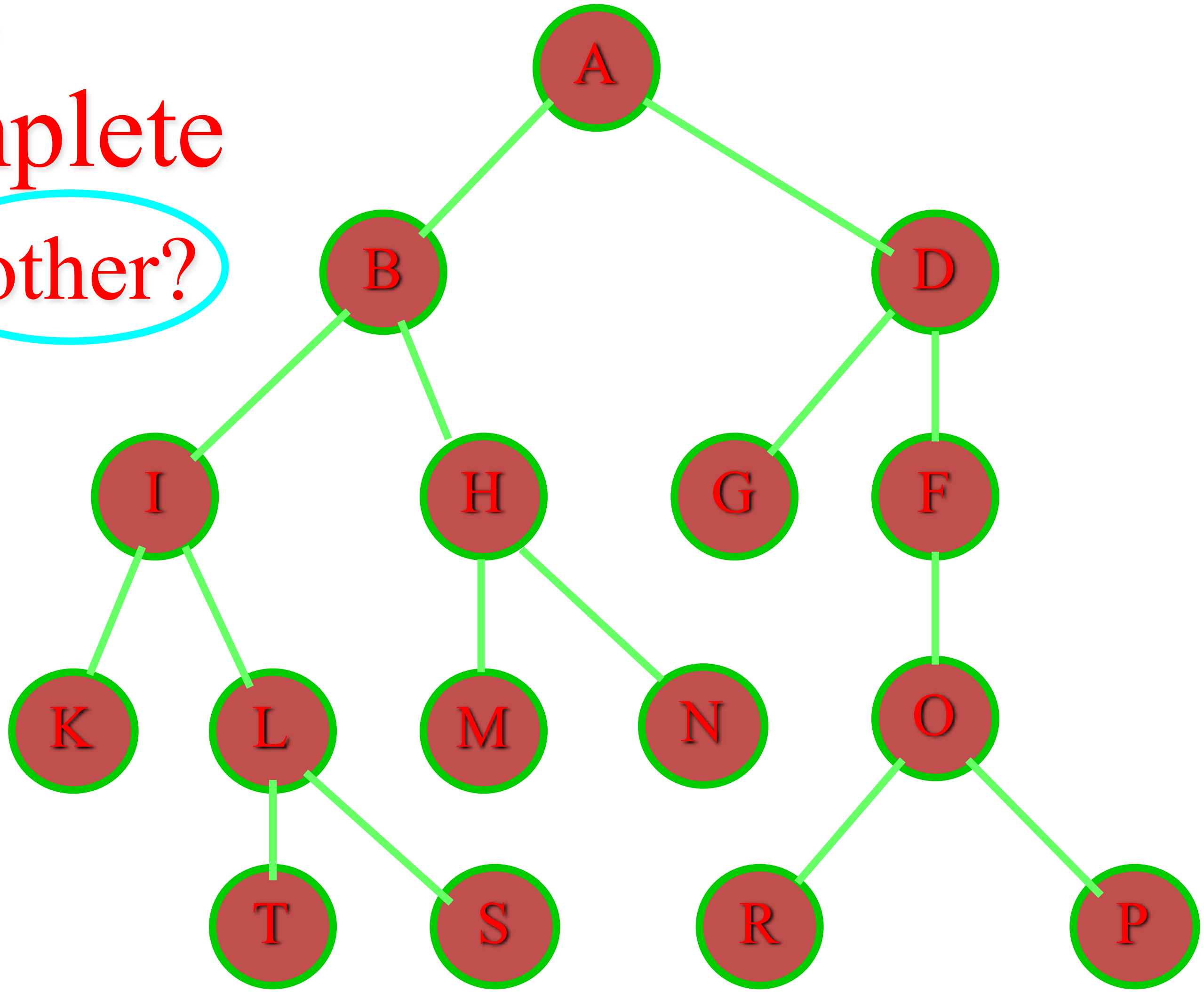
A complete binary tree



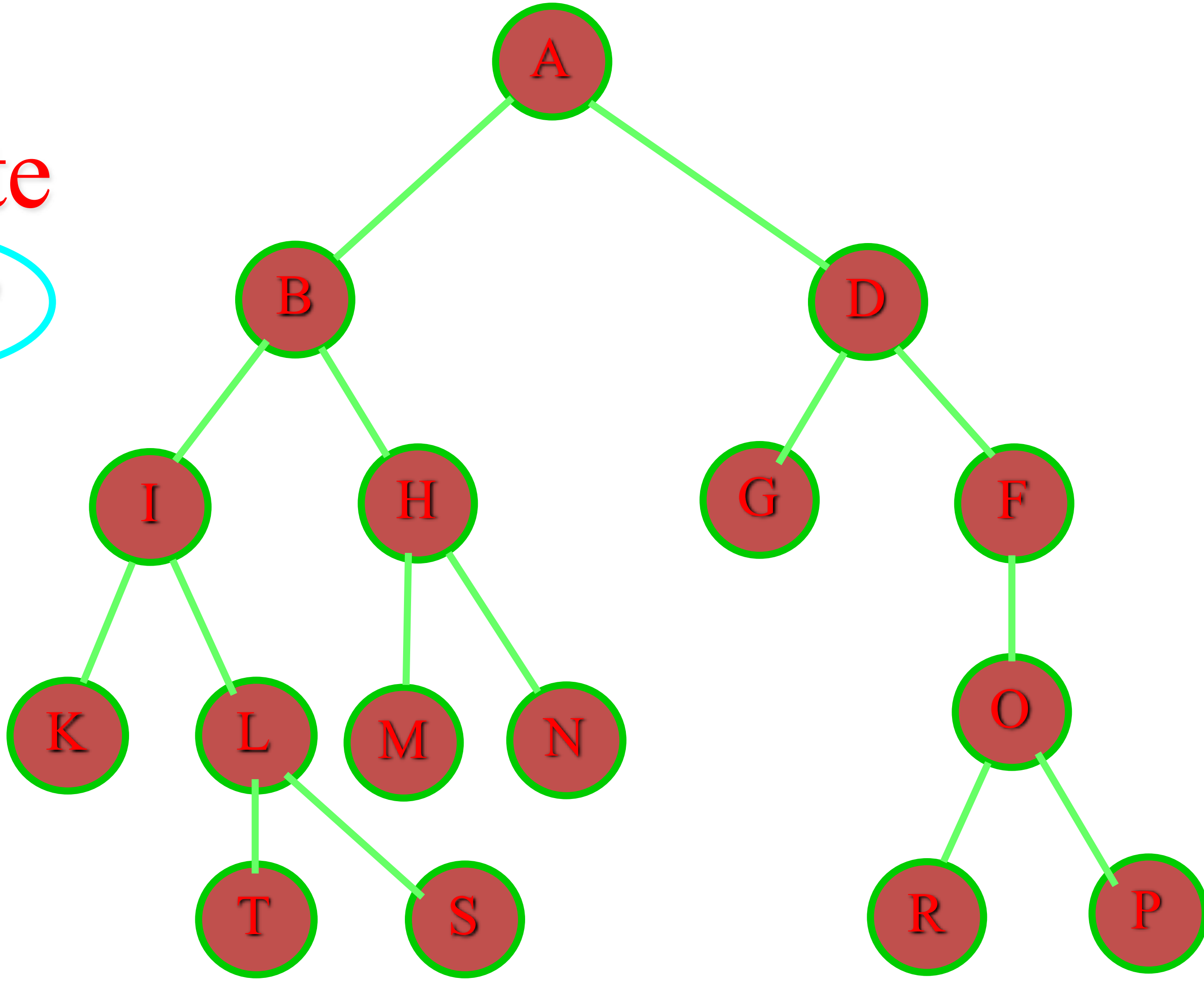
Full,
Complete
or Other?



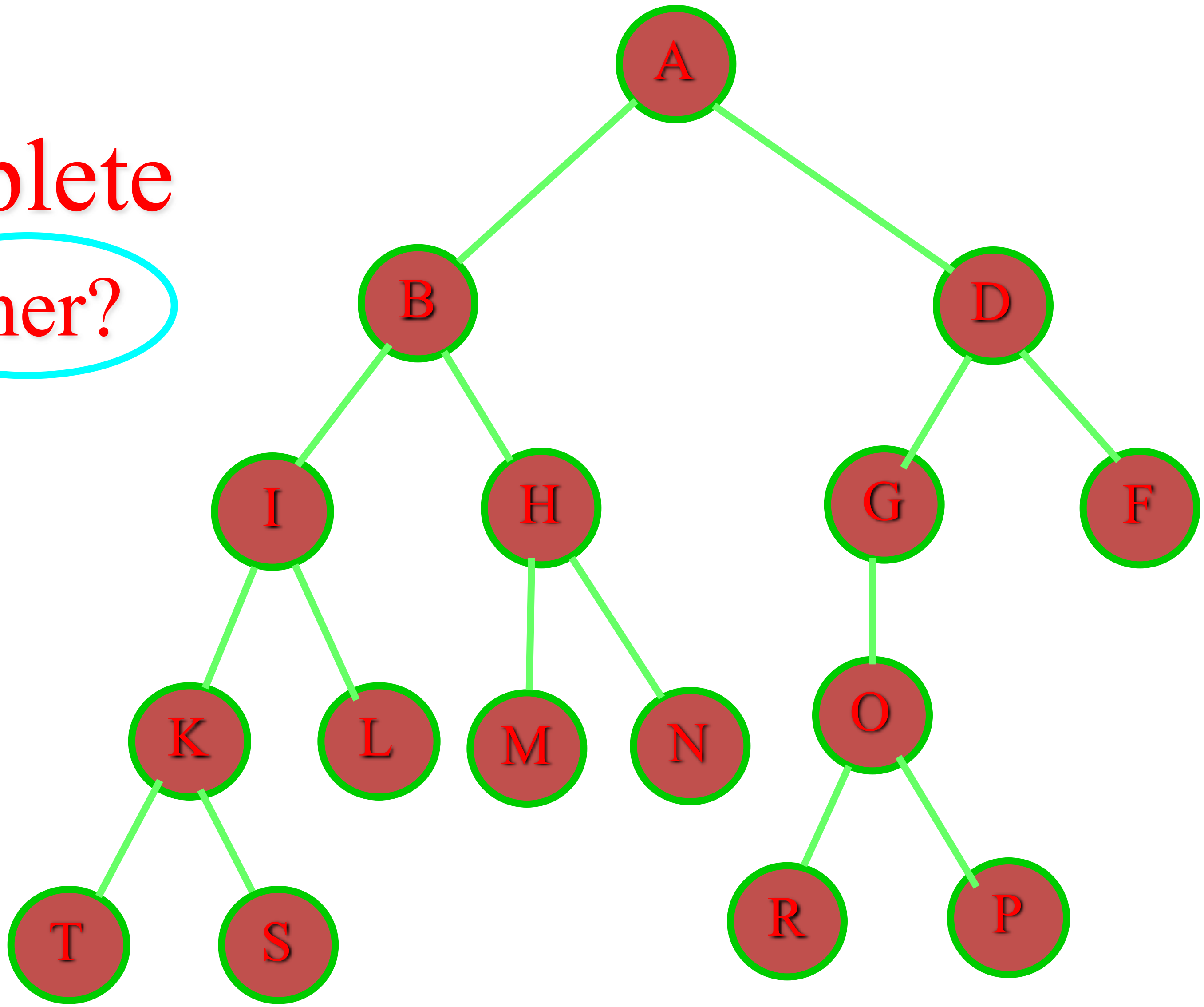
Full,
Complete
or other?



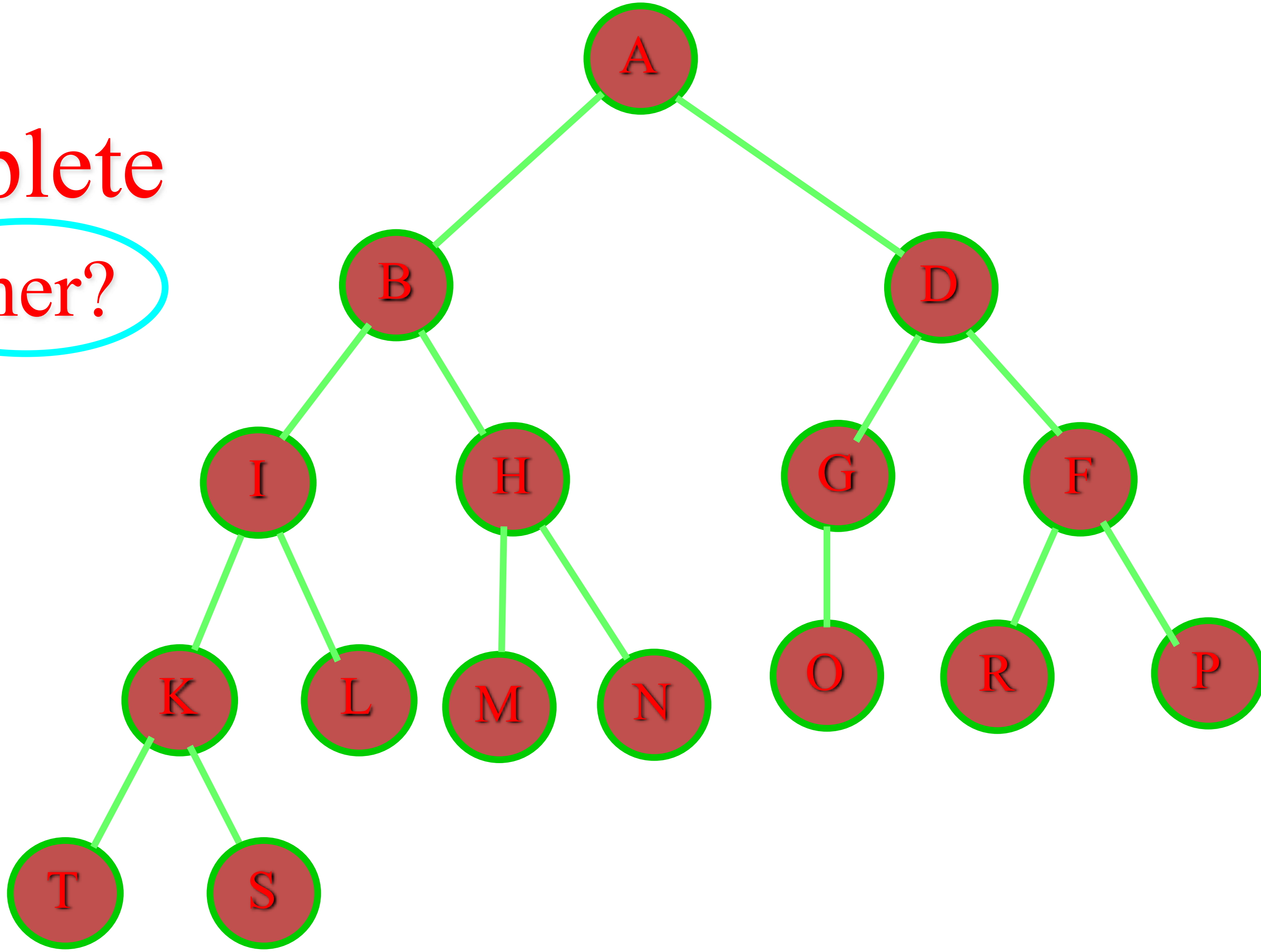
Full, Complete or other?



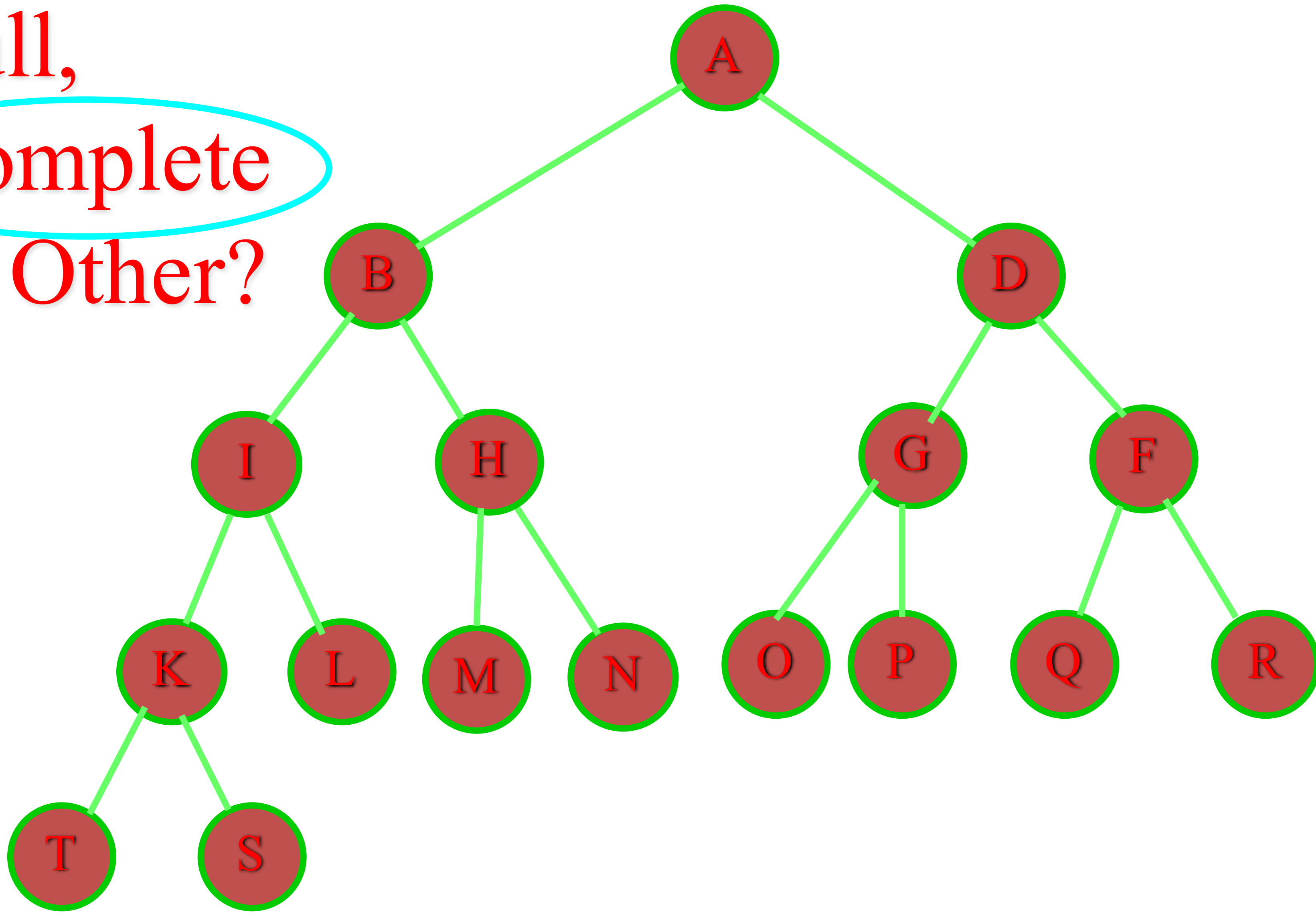
Full, Complete or other?



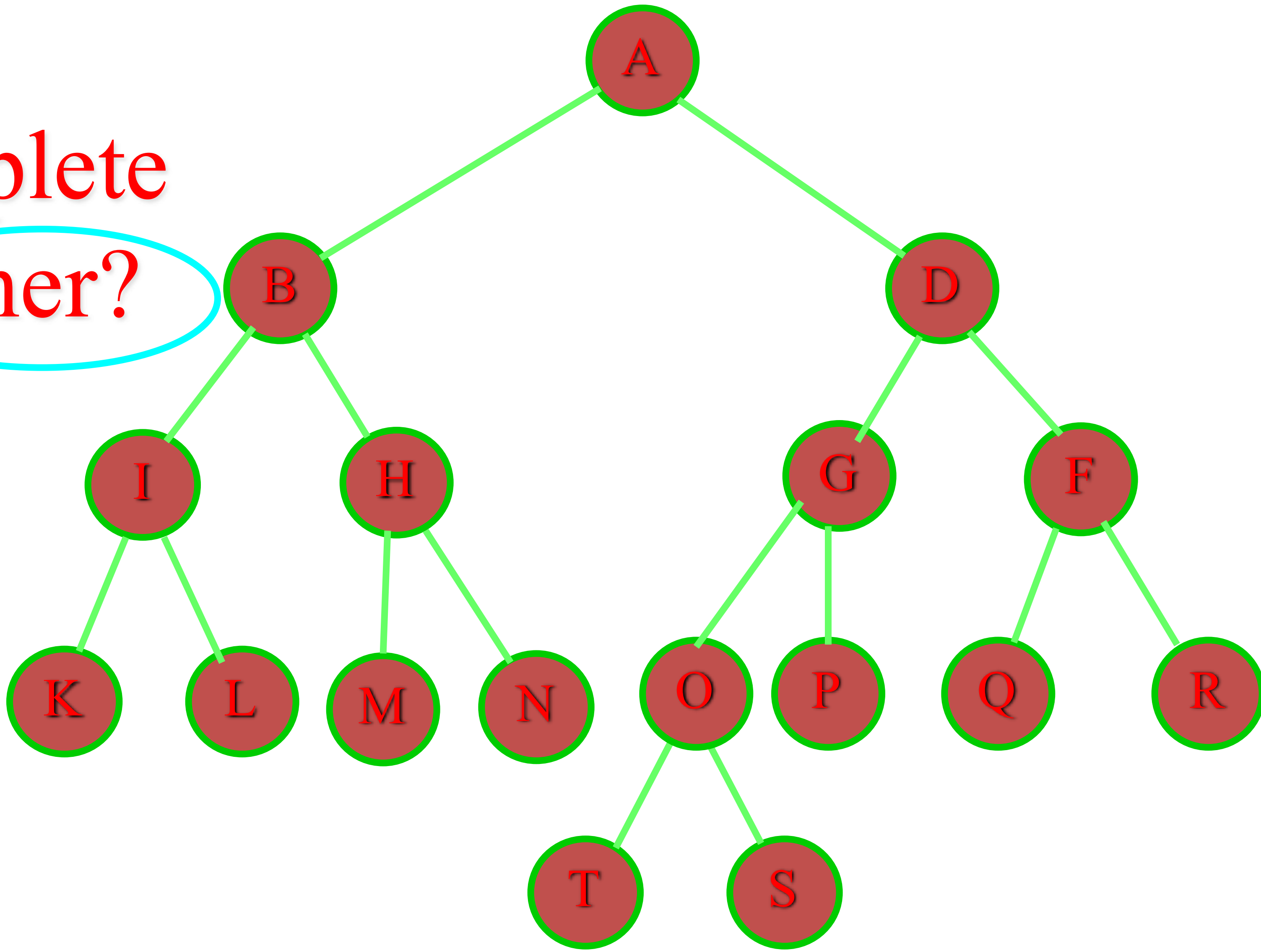
Full,
Complete
or other?



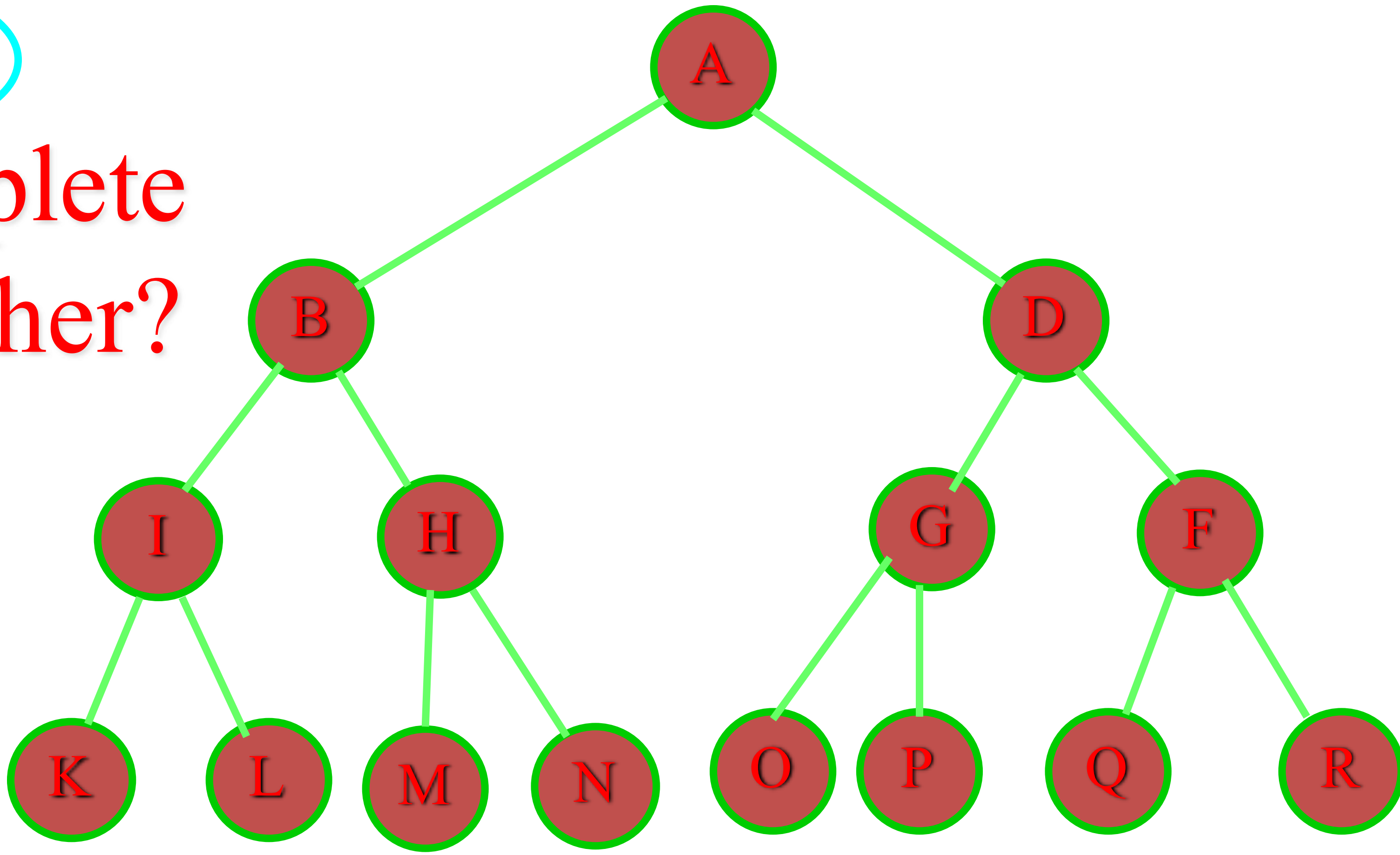
Full,
Complete
or Other?



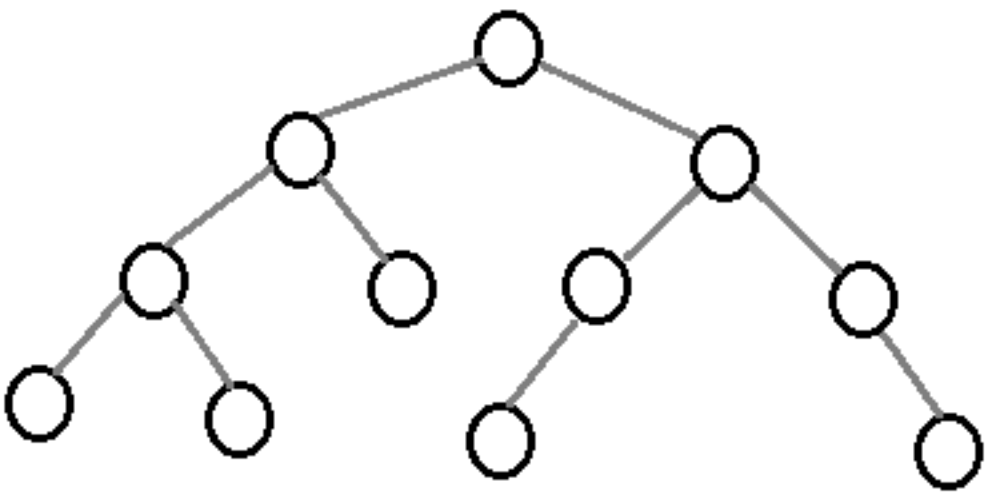
Full, Complete or other?



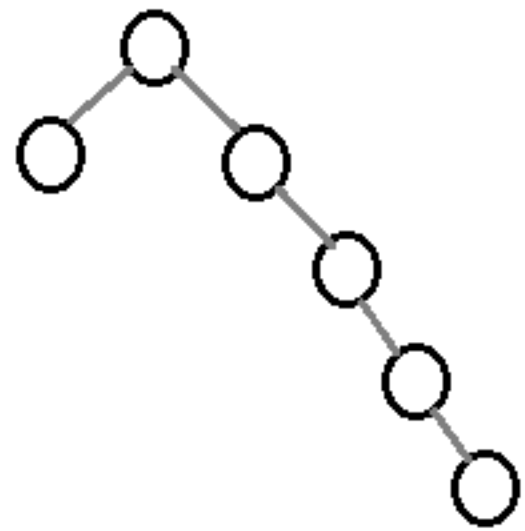
Full, Complete or Other?



- A balanced binary tree : Difference between the height of left subtree and the height of right subtree for every node is not more than k (mostly 1).



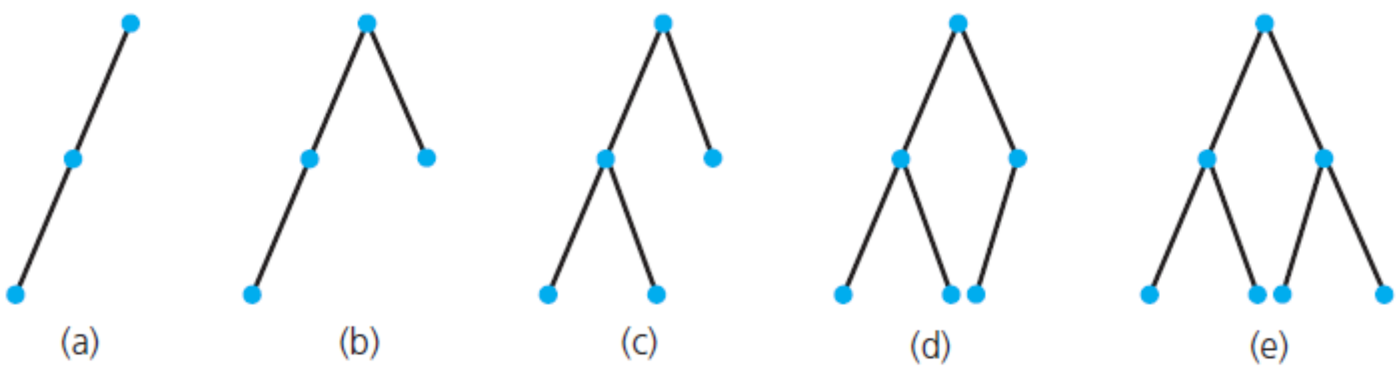
Balanced Binary Tree



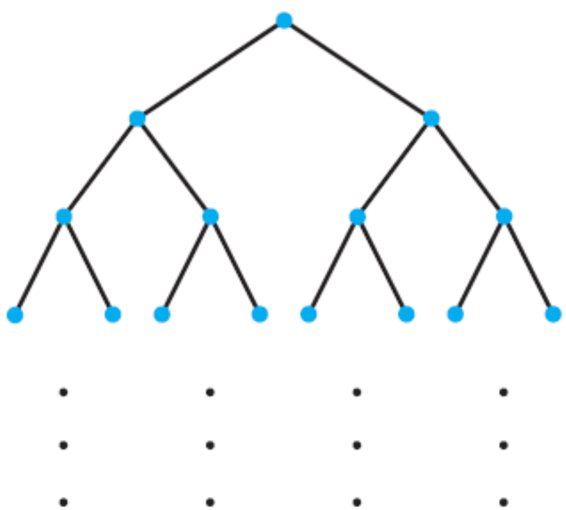
Unbalanced Binary Tree

The Maximum and Minimum Heights of a Binary Tree

- The maximum height of an n -node binary tree is n-1 .



The Maximum and Minimum Heights of a Binary Tree



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

Counting the nodes in a full binary tree of height 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**

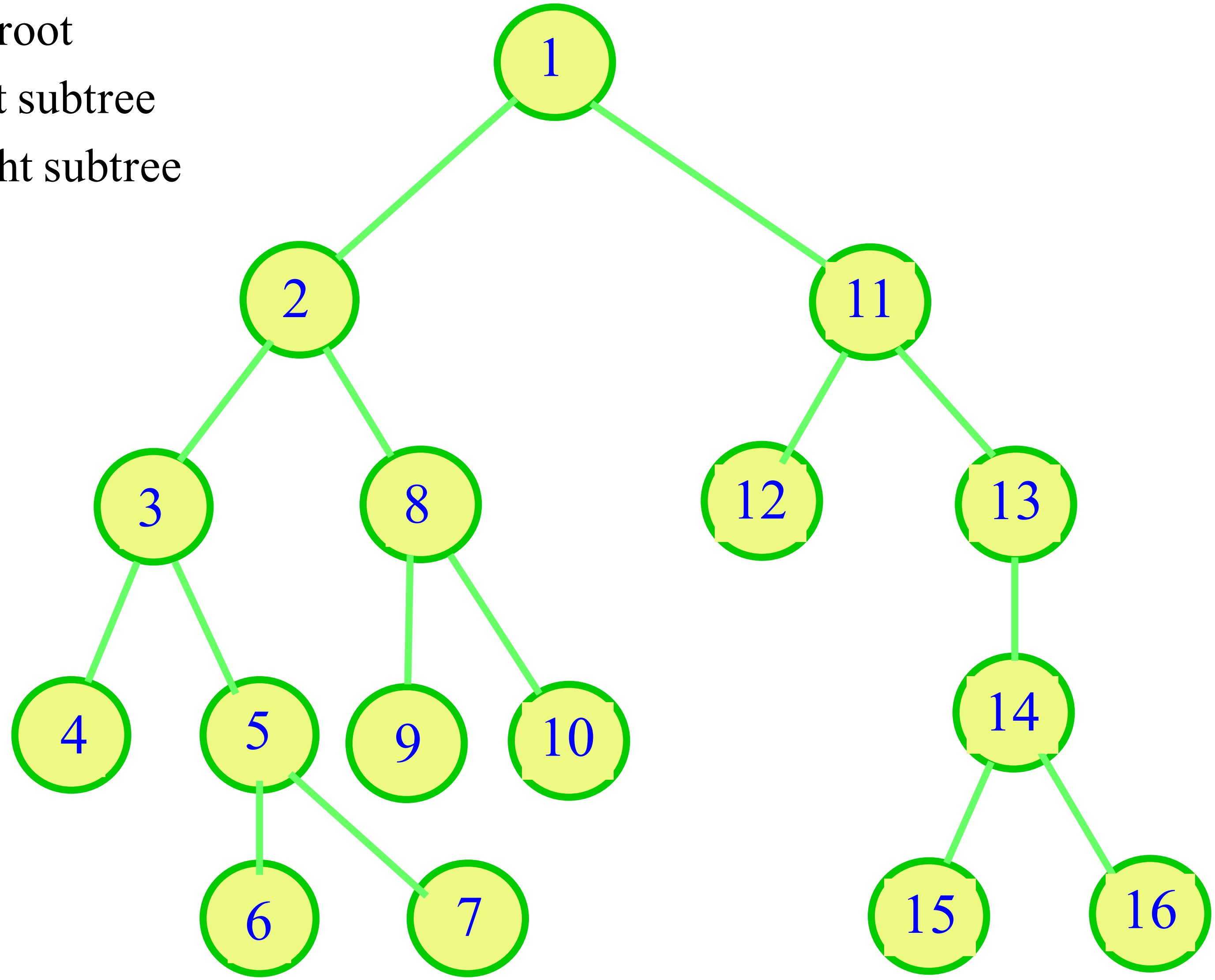
- Each node is processed after all nodes in its left subtree and before any node in its right subtree

3. **Postorder Traversal**

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

Preorder Traversals

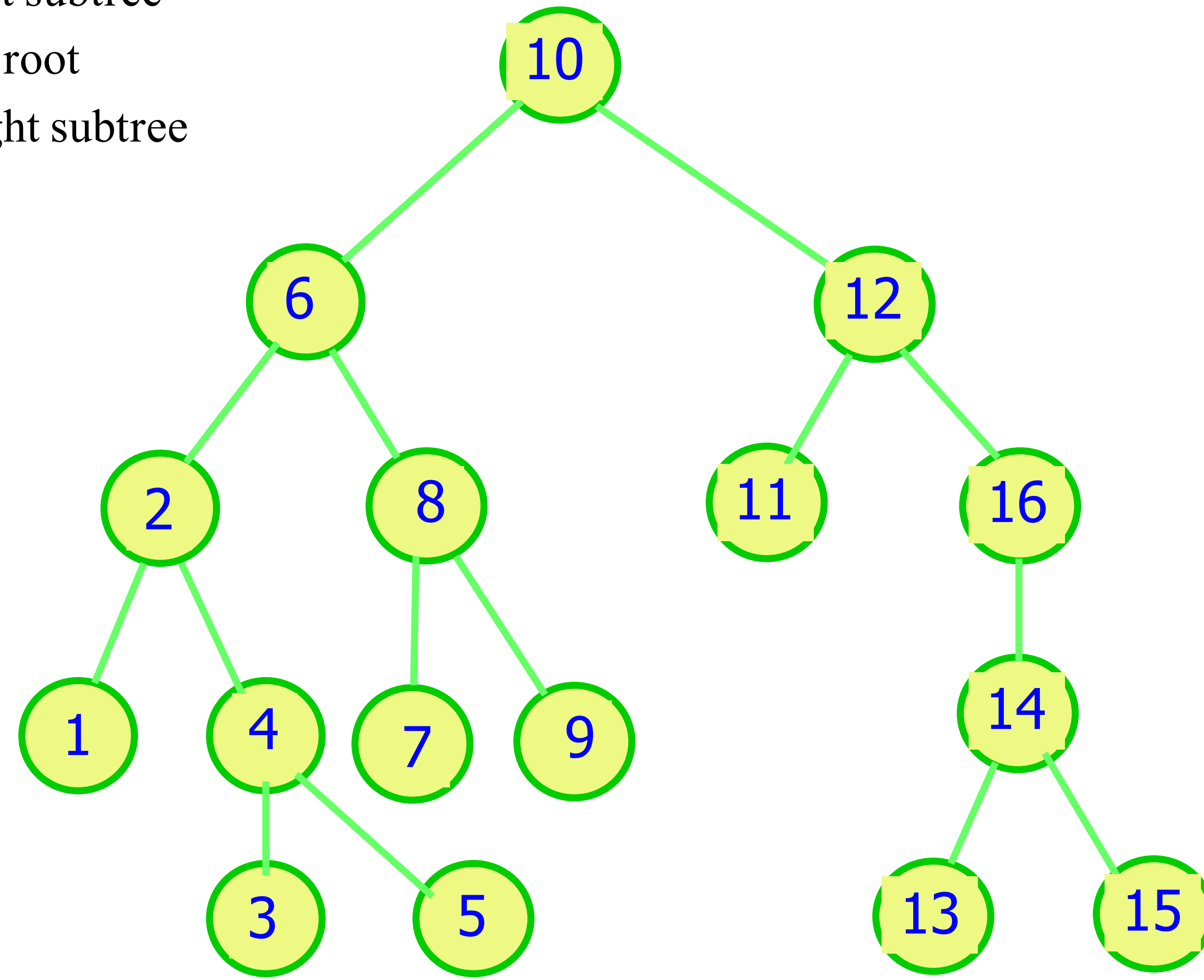
1. Visit the root
2. Visit Left subtree
3. Visit Right subtree



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

Inorder Traversals

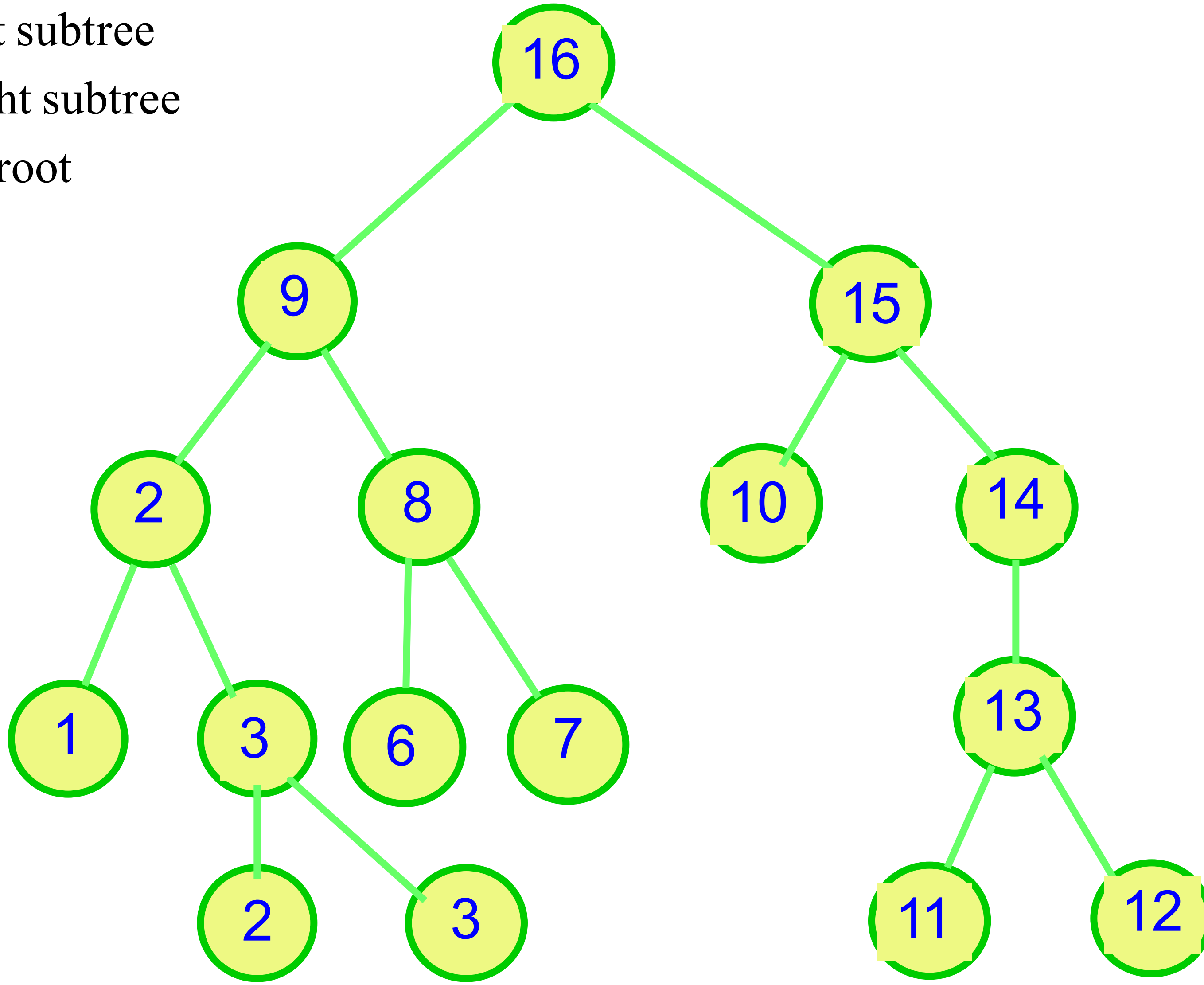
1. Visit Left subtree
2. Visit the root
3. Visit Right subtree



```
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)
```

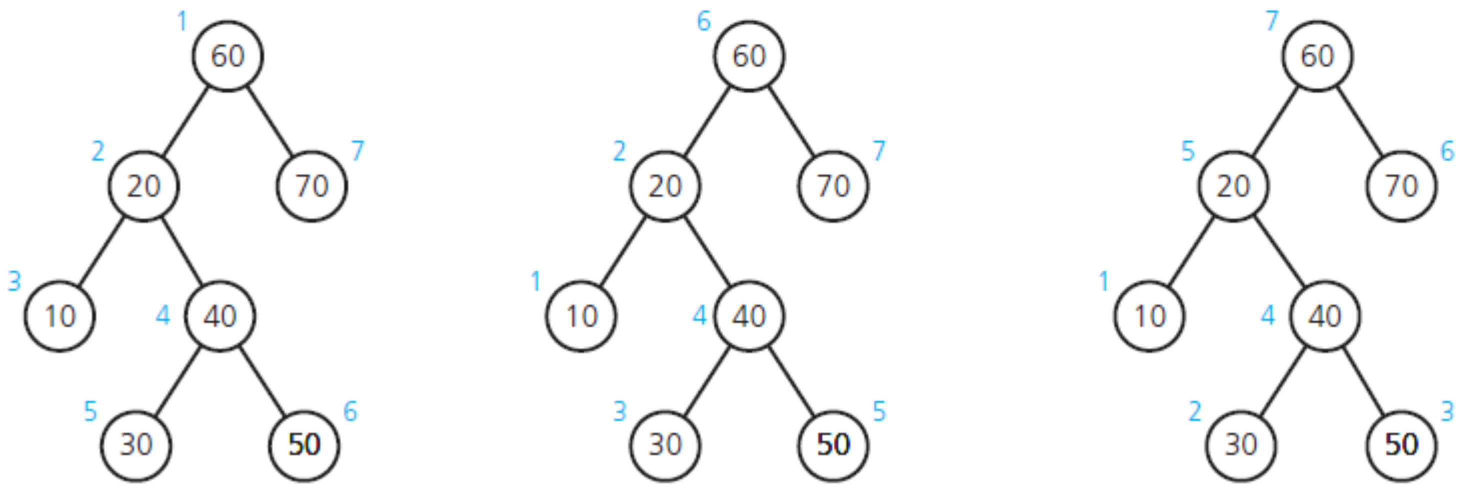
Postorder Traversals

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

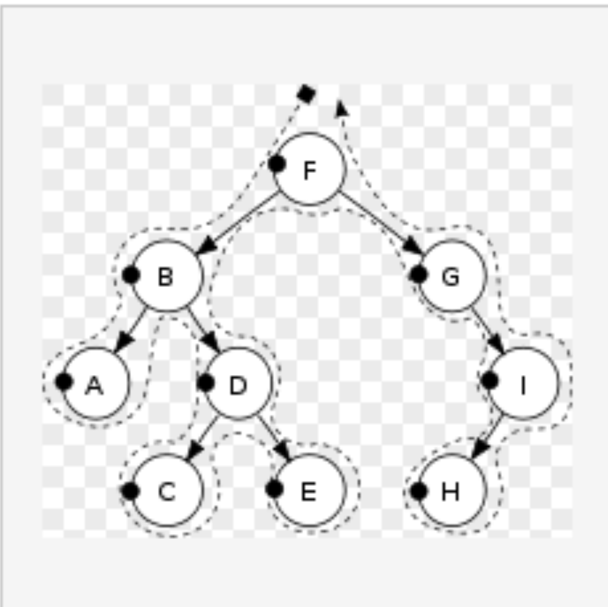


(a) Preorder: 60, 20, 10, 40, 30, 50, 70 (b) Inorder: 10, 20, 30, 40, 50, 60, 70 (c) Postorder: 10, 30, 50, 40, 20, 70, 60

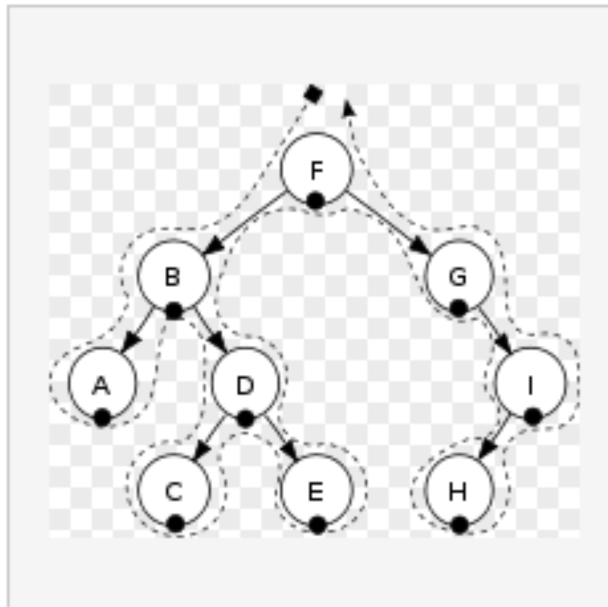
(Numbers beside nodes indicate traversal order.)

FIGURE 15-11 Three traversals of a binary tree

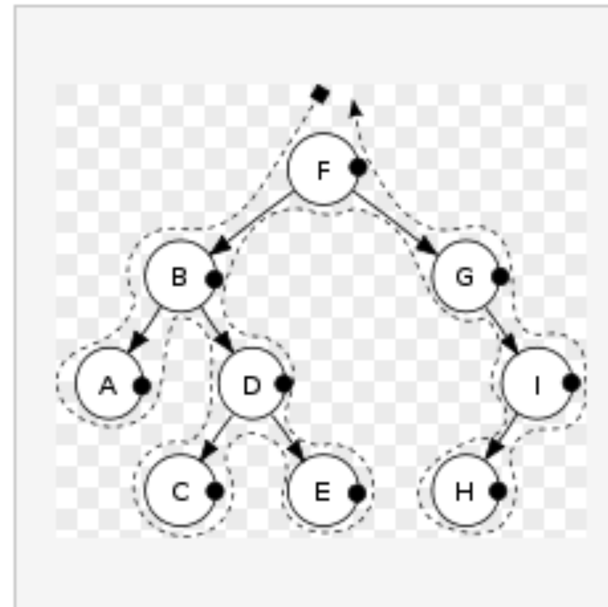
The 3 different types of traversal



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.

Binary Tree Operations

Elements: Any data type

Structure: A binary tree either is empty OR a node, called the root node, together with two binary trees, which are disjoint from each other and the root node. These are called left and right subtrees of the root

Domain: Number of elements is bounded

Operations:

Operation	**Specification**
void **empty**()	**Precondition/Requires:** none. **Processing/Results:** returns true if the binary tree (BT) has no nodes.
void **traverse**(Order ord)	**Precondition/Requires:** BT is not empty. **Processing/Results:** Traverses the binary tree according to the value of ord (1) ord = preOrder: traverses the tree using preorder traversal (2) ord = inOrder: traverses the tree using inorder traversal (3) ord = postOrder: traverses the tree using postorder traversal

Operation	**Specification**
void **find** (Relative rel)	**Precondition/Requires:** BT is not empty. **Processing/Results:** the current node of BT is determined by Relative and the current node prior to the operation as follows (always return true unless indicated so): (1) rel = Root: current = root (2) rel = Parent: if the current node has a parent then parent is the current node; otherwise returns false (3) rel = LeftChild: if the current node has a leftchild then it will be the current node; otherwise returns false (4) rel = RightChild: same as above but for rightchild.
void **insert**(Relative rel, Type val)	**Precondition/Requires:** either (1) BT is empty and rel = Root; or (2) BT not empty and rel $\neq$ Root . **Processing/Results:** as follows: (1) rel = Root: create a root node with data = val. (2) rel = Parent: nonsense case. (3) rel = LeftChild: if current node does not have a leftchild then make one with data = val. (4) rel = RightChild: same as above but for rightchild. In all the above cases if the insertion was successful then it will be designated as current node and returns true, otherwise current remains unchanged and returns false.

Operation	Specification
void **update**(Type)	**Precondition/Requires:** BT is not empty. **Processing/Results:** update the value of data of the current node.
Type **retrieve**()	**Precondition/Requires:** BT is not empty. **Processing/Results:** returns data of the current node.
void **delete_sub**()	**Precondition:** BT is not empty. **Process:** the subtree whose root node was the current node before this operation is deleted from the tree. In case the resulting tree is not empty then current = root.

Note: *Relative is enumerated type and is confined to the values {Root, Parent, LeftChild, RightChild}*

Represention of Binary Tree ADT

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

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

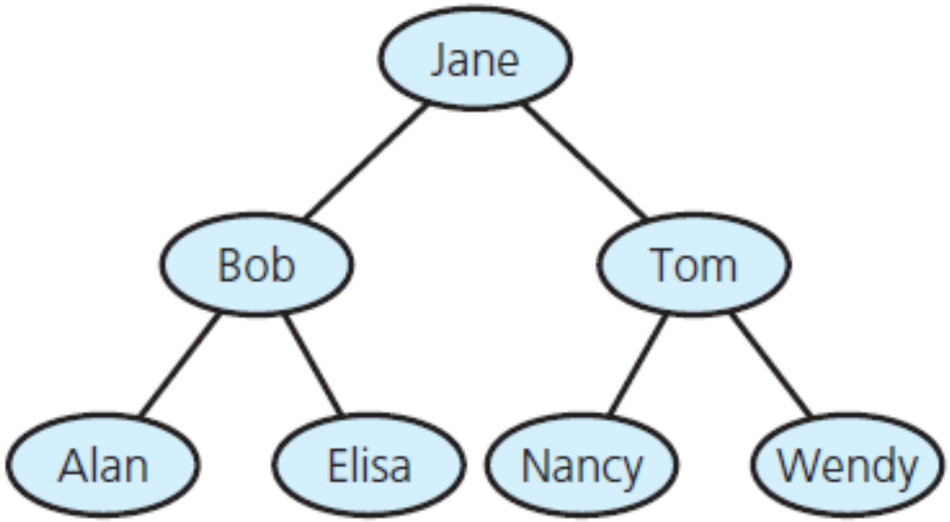
```
void postorder(node  *p)
{
   if (p != NULL)
   {
      postorder(p->left);
      postorder(p->right);
      printf(p->key_value);
   }
}
```

```
void destroy_tree(struct node *leaf)
{
  if( leaf != 0 )
  {
     destroy_tree(leaf->left);
     destroy_tree(leaf->right);
     free( leaf );
  }
}
```

The ADT Binary Search Tree

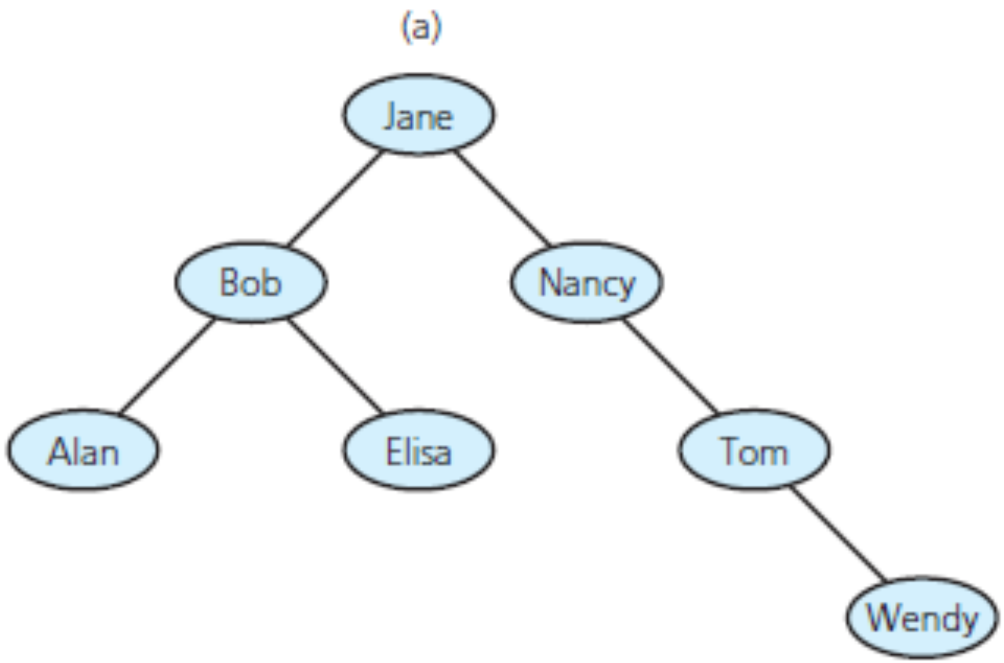
- ADT binary tree suited for search for specific item
- Binary *search* tree solves problem
- Properties of each node, n
  - n's value greater than all values in left subtree T_L
  - n's value less than all values in right subtree T_R
  - Both T_R and T_L are binary search trees.

The ADT Binary Search Tree



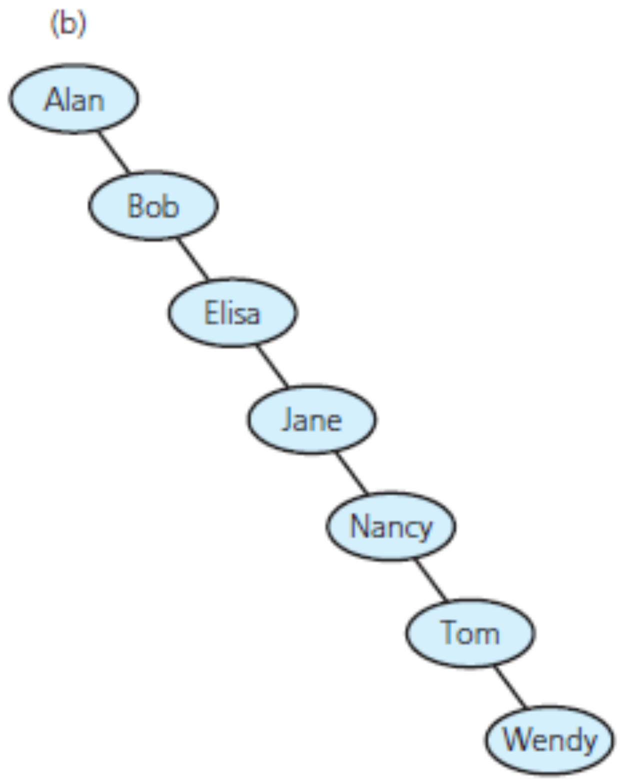
A binary search tree of names

The ADT Binary Search Tree



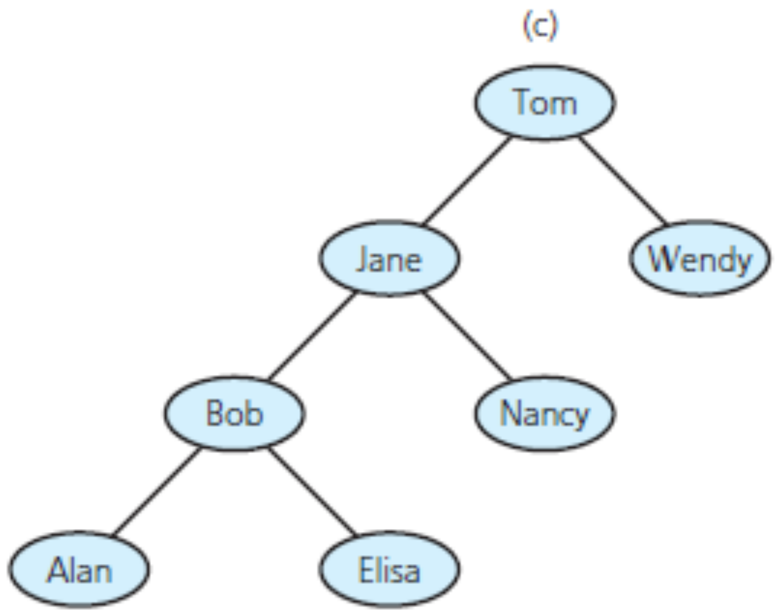
Binary search trees with the same data

The ADT Binary Search Tree



Binary search trees with the same data

The ADT Binary Search Tree



Binary search trees with the same data

Binary Search Tree Operations

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

Binary Search Tree Operations

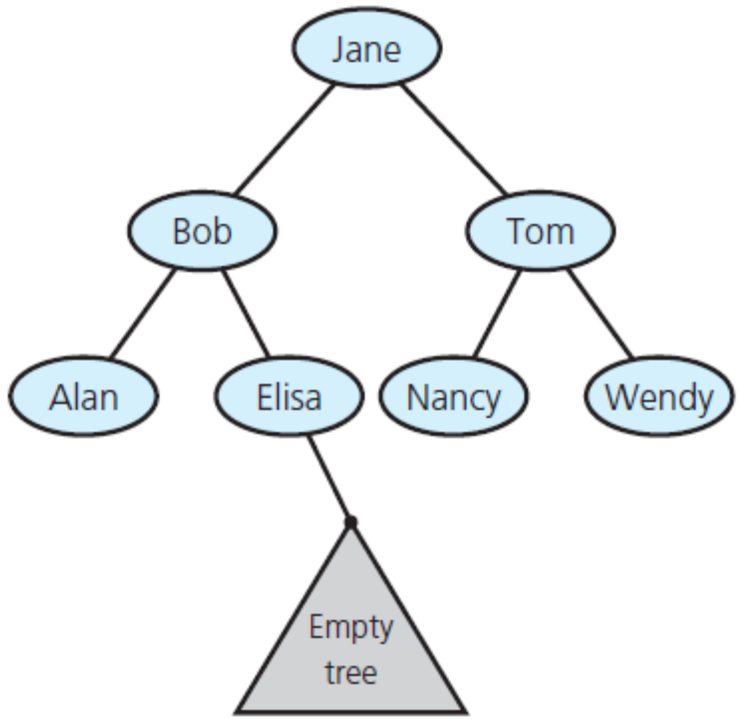
- Remove all entries from binary search tree.
- Retrieve given item from binary search tree.
- Test whether binary search tree contains specific entry.
- Traverse items in binary search tree in
  - Preorder
  - Inorder
  - Postorder.

Searching a Binary Search Tree

- Search algorithm for binary search tree

```
struct node *search(int key, struct node *leaf)
{
  if( leaf != 0 )
  {
      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

```c
void insert(int key, struct node **leaf)
{
    if( *leaf == 0 )
    {
        *leaf = (struct node*) malloc( sizeof( struct node ) );
        (*leaf)->key_value = key;
        /* initialize the children to null */
        (*leaf)->left = 0;
        (*leaf)->right = 0;
    }
    else if(key < (*leaf)->key_value)
    {
        insert( key, &(*leaf)->left );
    }
    else if(key > (*leaf)->key_value)
    {
        insert( key, &(*leaf)->right );
    }
}
```

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