Data Structures & Algorithms (CS-212)

Week 9: Trees

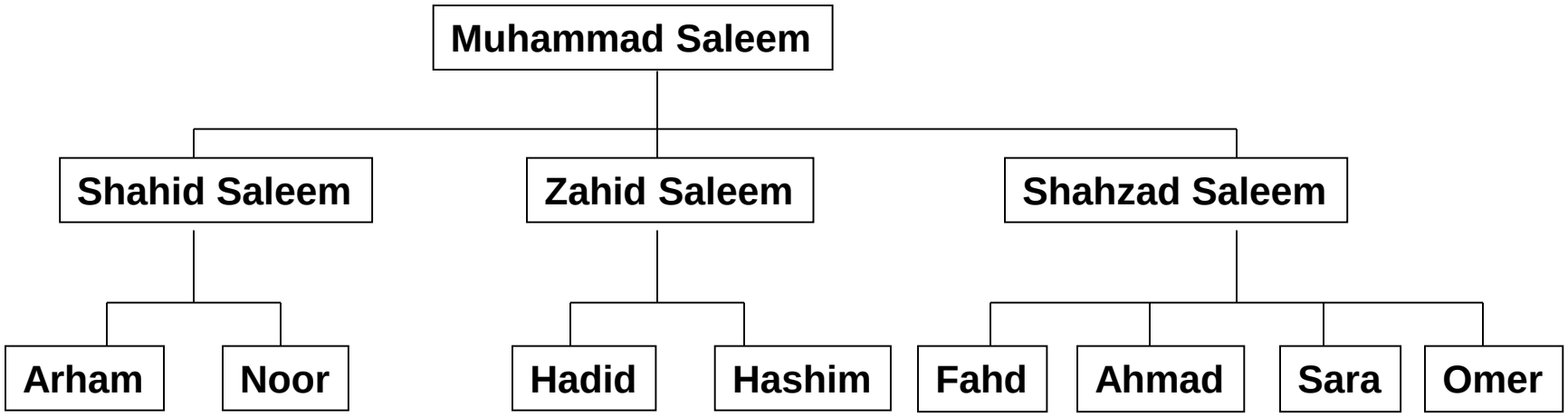
Trees

- So far we did linear data structures
- There are a number of applications where linear data structures are not appropriate.
- Trees are non-linear
- Allow much more efficient implementations of algorithms
- Also, tree structure is the natural form of many types of data
- Therefore, are seen commonly in file systems, GUIs, databases, websites etc.

Trees

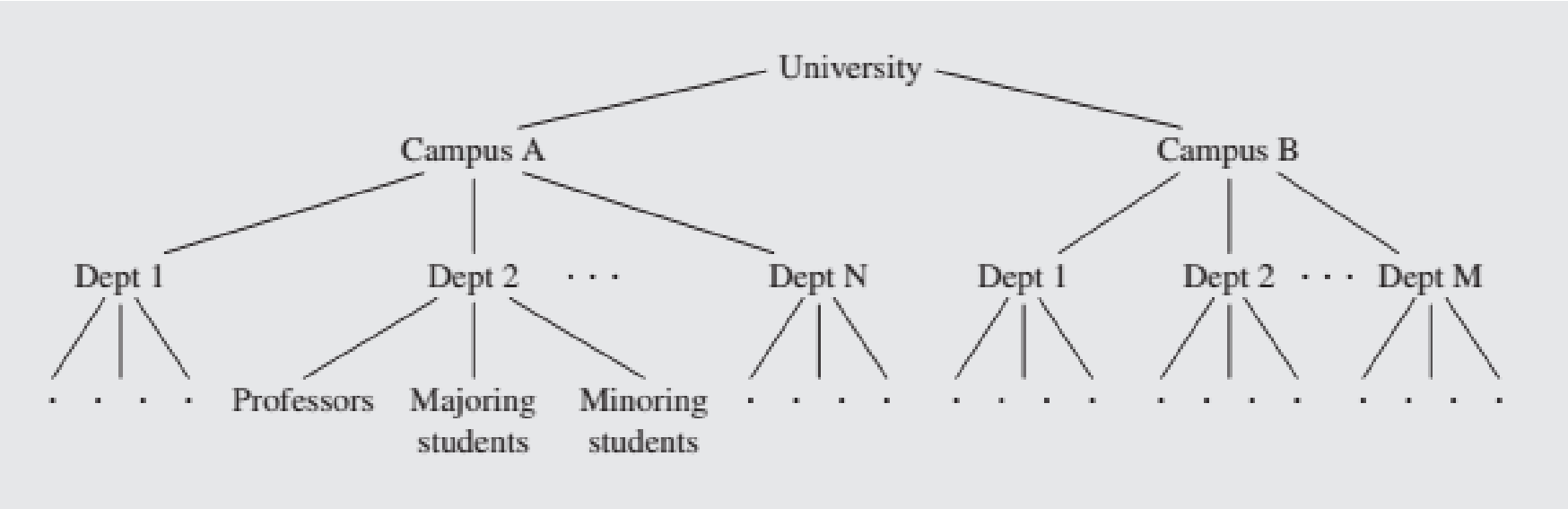
- For the linear context, in our earlier data structures, there was a before and after relationship between elements
- The relationship between elements in trees is hierarchical, with elements being above and below each other
- The terminology comes from family tree, so we will be using terminology such as parent, child, ancestor, descendant etc.

Tree of a Family



University Hierarchy

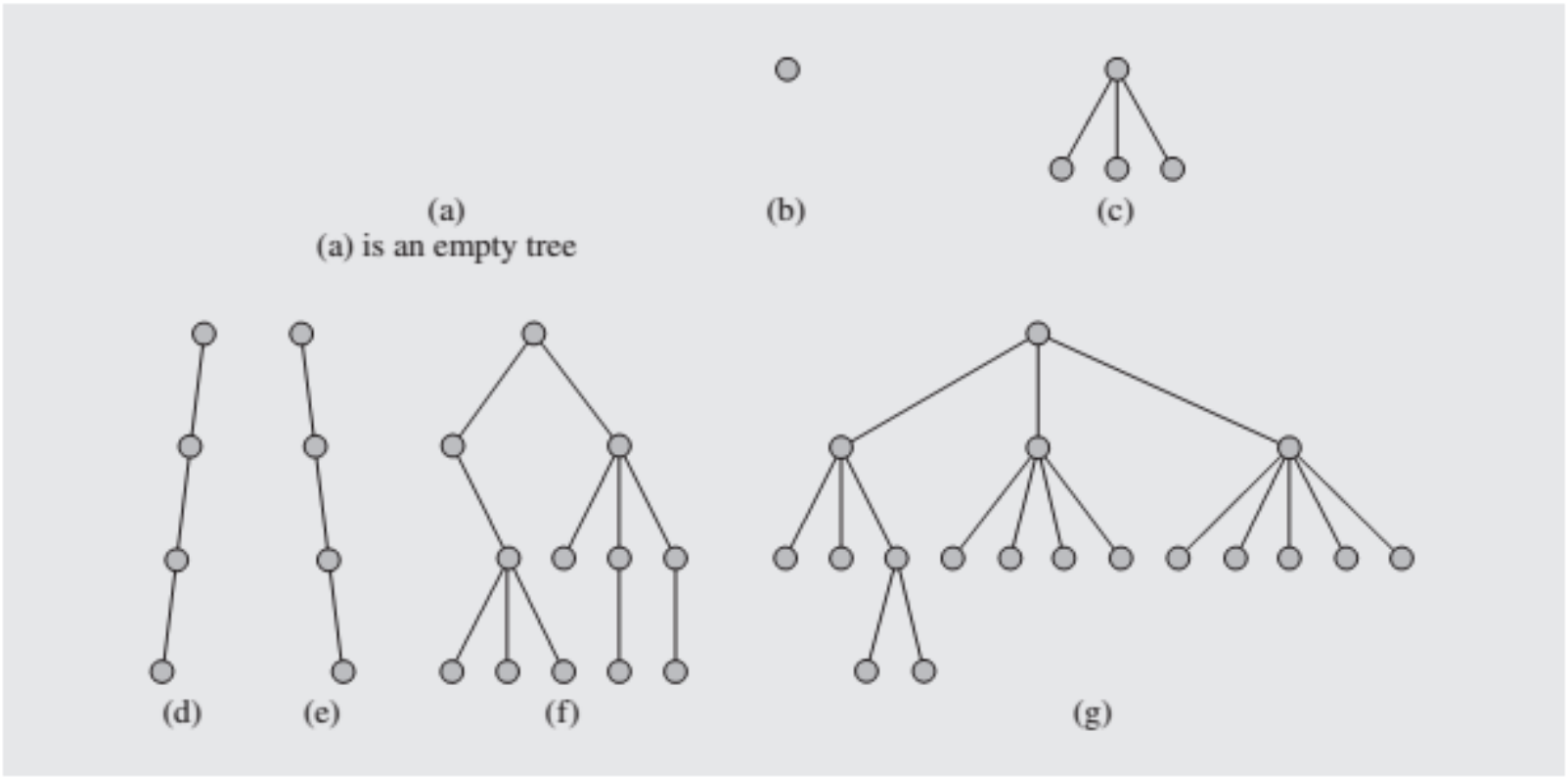
- Hierarchical structure of a university shown as a tree.



Tree

- A ***tree*** is an abstract data type that stores elements hierarchically
- With the exception of the top element, known as the root, each element in a tree has a ***parent*** element and zero or more ***children*** elements

FIGURE 6.1 Examples of trees.



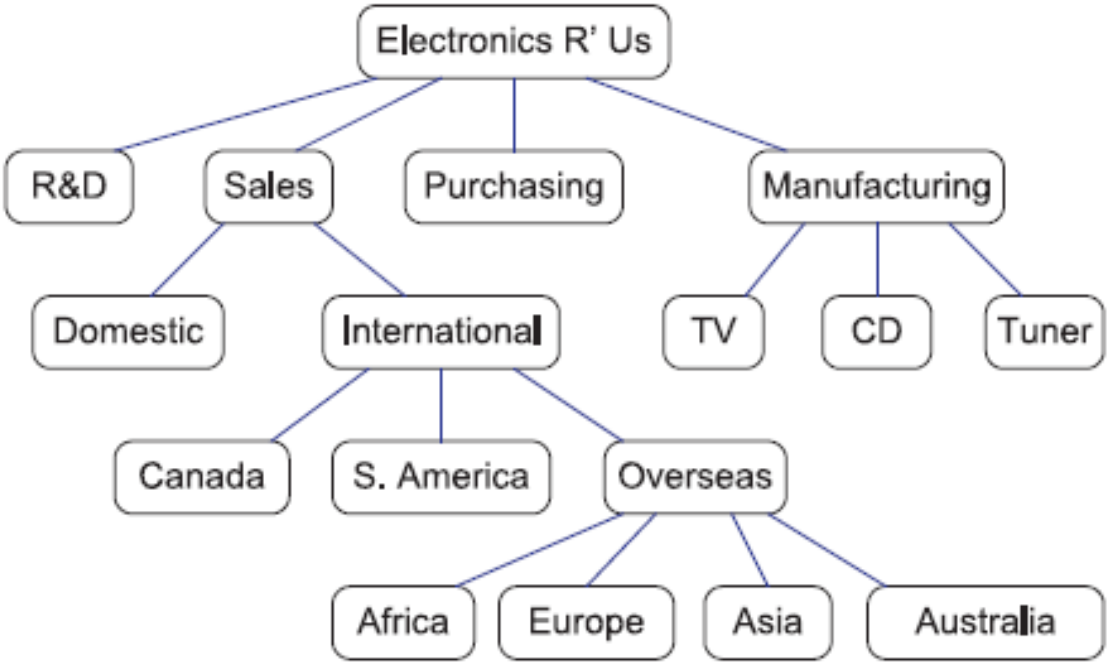


Figure 7.2: A tree with 17 nodes representing the organizational structure of a fictitious corporation. *Electronics R'Us* is stored at the root. The children of the root store *R&D*, *Sales*, *Purchasing*, and *Manufacturing*. The internal nodes store *Sales*, *International*, *Overseas*, *Electronics R'Us*, and *Manufacturing*.

Formal Tree Definition

Formally, we define ***tree*** T to be a set of ***nodes*** storing elements in a ***parent-child*** relationship with the following properties:

- If T is nonempty, it has a special node, called the ***root*** of T, that has no parent.
- Each node v of T different from the root has a unique ***parent*** node w; every node with parent w is a ***child*** of w.

Note that according to our definition, a tree can be empty, meaning that it doesn't have any nodes. This convention also allows us to define a tree recursively, such that a tree T is either empty or consists of a node r, called the root of T, and a (possibly empty) set of trees whose roots are the children of r.

Other Node Relationships

Two nodes that are children of the same parent are ***siblings***. A node v is ***external*** if v has no children. A node v is ***internal*** if it has one or more children. External nodes are also known as ***leaves***.

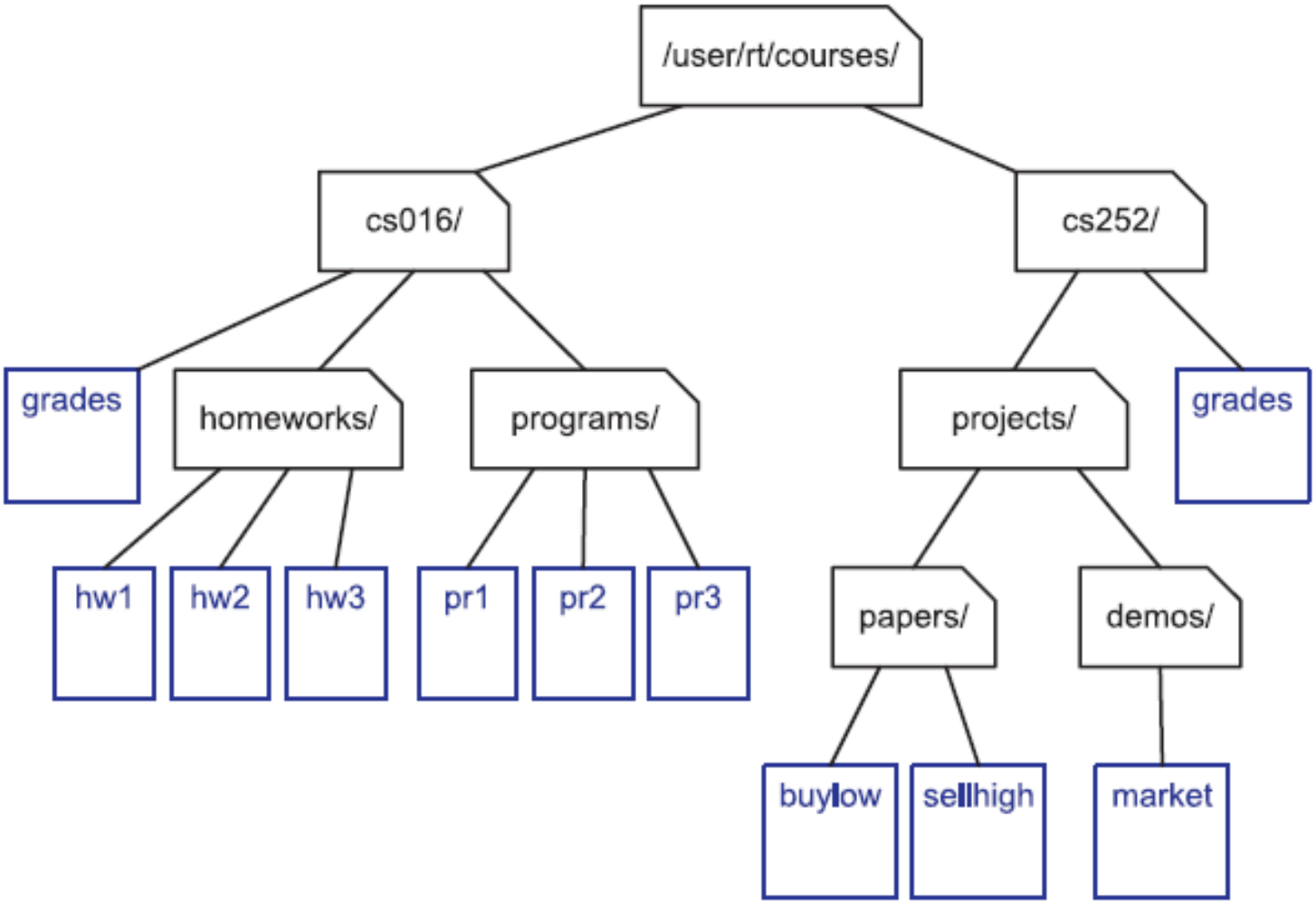
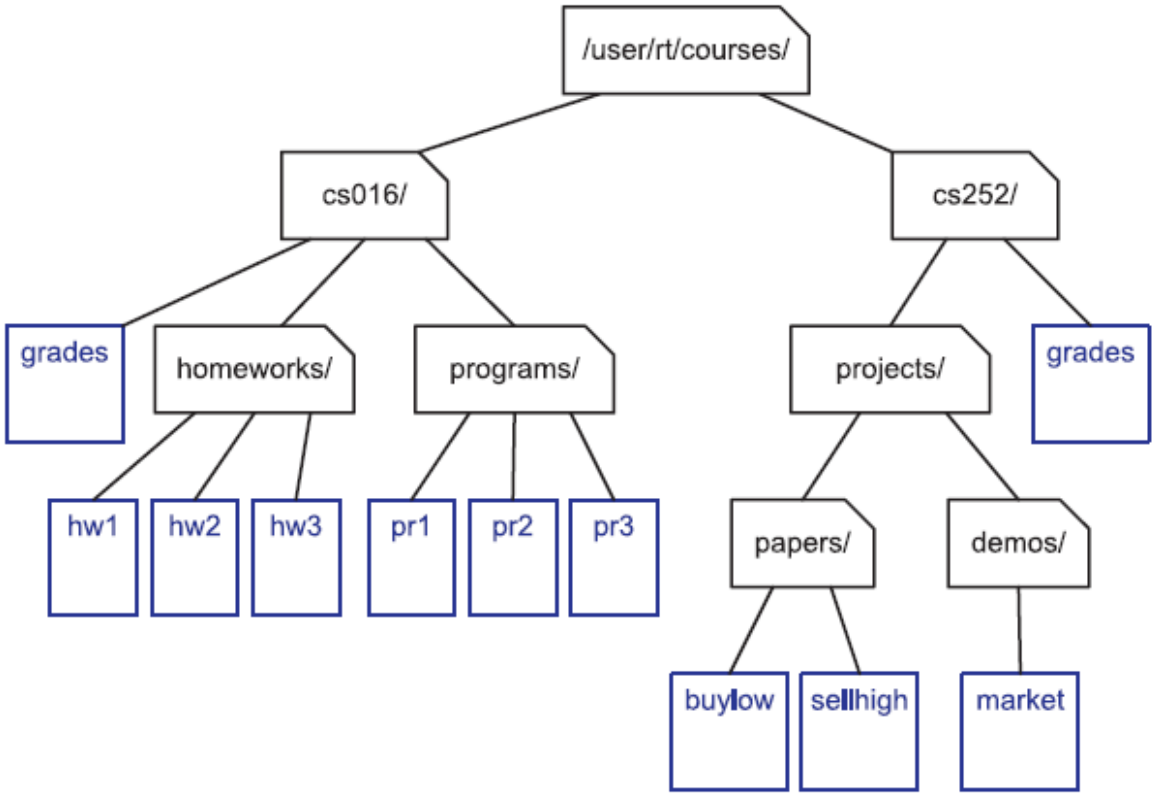


Figure 7.3: Tree representing a portion of a file system.

External or leaf nodes?



A node u is an ***ancestor*** of a node v if $u = v$ or u is an ancestor of the parent of v. Conversely, we say that a node v is a ***descendent*** of a node u if u is an ancestor of v. For example, in Figure 7.3, cs252/ is an ancestor of papers/, and pr3 is a descendent of cs016/. The ***subtree*** of T ***rooted*** at a node v is the tree consisting of all the descendents of v in T (including v itself). In Figure 7.3, the subtree rooted at cs016/ consists of the nodes cs016/, grades, homeworks/, programs/, hw1, hw2, hw3, pr1, pr2, and pr3.

Edges and Paths in Trees

An ***edge*** of tree T is a pair of nodes (u,v) such that u is the parent of v, or vice versa. A ***path*** of T is a sequence of nodes such that any two consecutive nodes in the sequence form an edge. For example, the tree in Figure 7.3 contains the path (cs252/, projects/, demos/, market).

Ordered Trees

A tree is ***ordered*** if there is a linear ordering defined for the children of each node; that is, we can identify children of a node as being the first, second, third, and so on. Such an ordering is determined by how the tree is to be used, and is usually indicated by drawing the tree with siblings arranged from left to right, corresponding to their linear relationship. Ordered trees typically indicate the linear order relationship existing between siblings by listing them in a sequence or iterator in the correct order.

Example 7.3: *A structured document, such as a book, is hierarchically organized as a tree whose internal nodes are chapters, sections, and subsections, and whose external nodes are paragraphs, tables, figures, the bibliography, and so on. (See Figure 7.4.) The root of the tree corresponds to the book itself. We could, in fact, consider expanding the tree further to show paragraphs consisting of sentences, sentences consisting of words, and words consisting of characters. In any case, such a tree is an example of an ordered tree, because there is a well-defined ordering among the children of each node.*

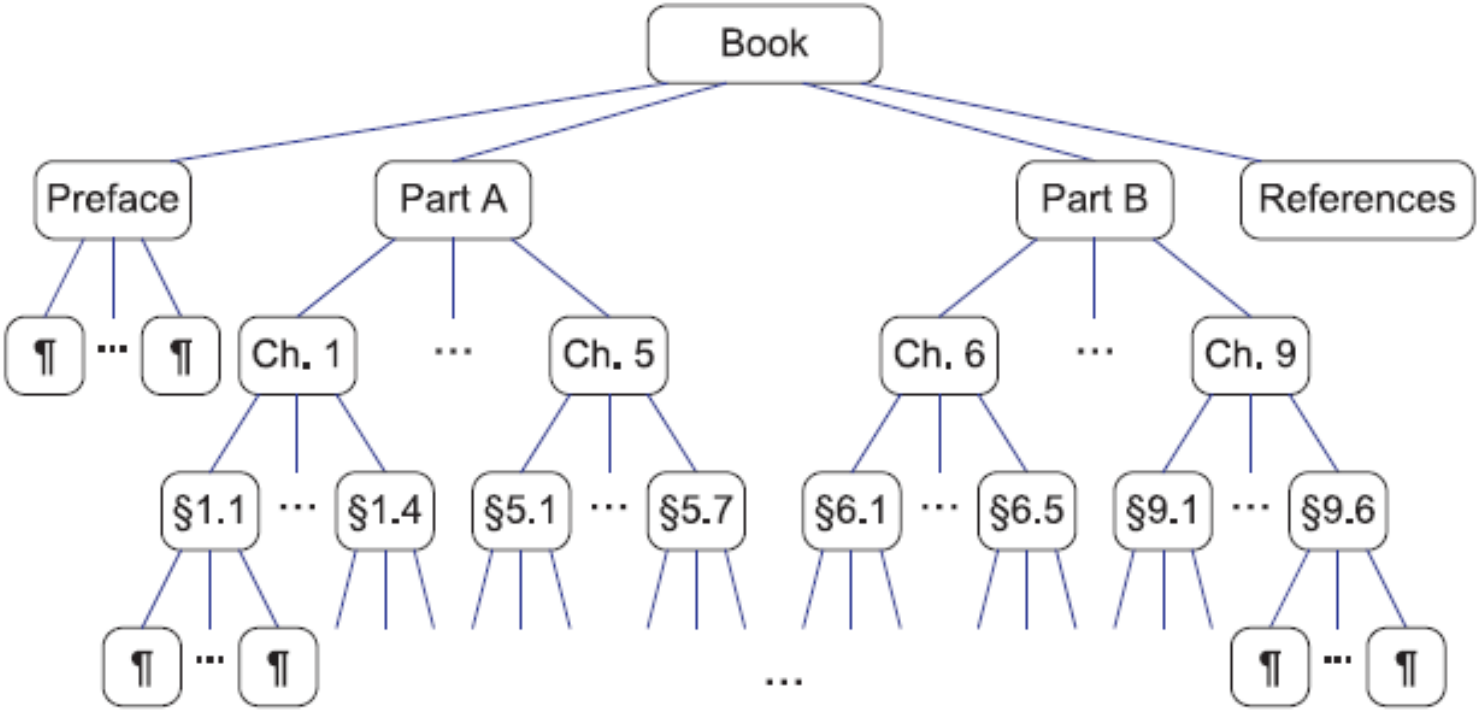


Figure 7.4: An ordered tree associated with a book.

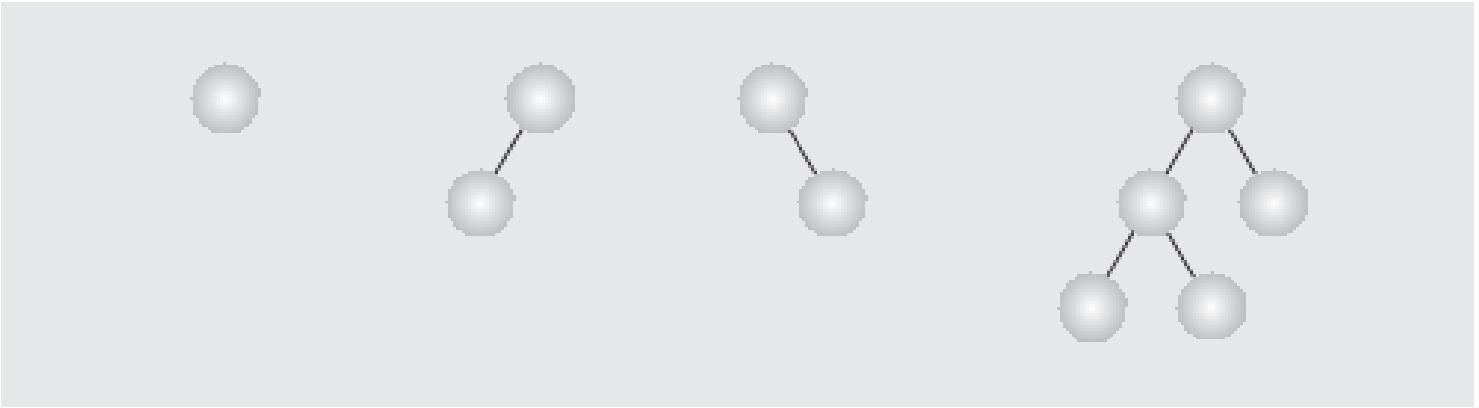
Tree Data Structure

- A linear linked list will not be able to capture the tree-like relationship with ease.
- Shortly, we will see that for applications that require searching, linear data structures are not suitable.
- A node in tree can have any number of children.
- We will focus our attention on *binary trees*.

Binary Tree

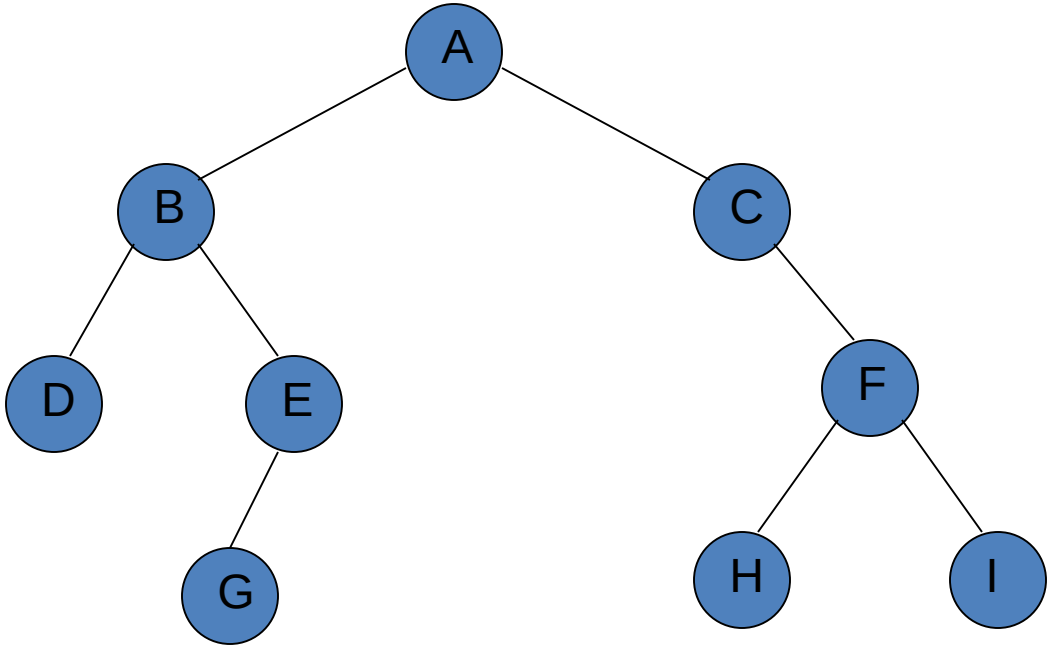
- A *binary tree* is a finite set of elements that is either empty or is partitioned into *three* disjoint subsets.
- The first subset contains a single element called the *root* of the tree.
- The other two subsets are themselves binary trees called the *left* and *right subtrees*.
- Each element of a binary tree is called a *node* of the tree.

FIGURE 6.4 Examples of binary trees.

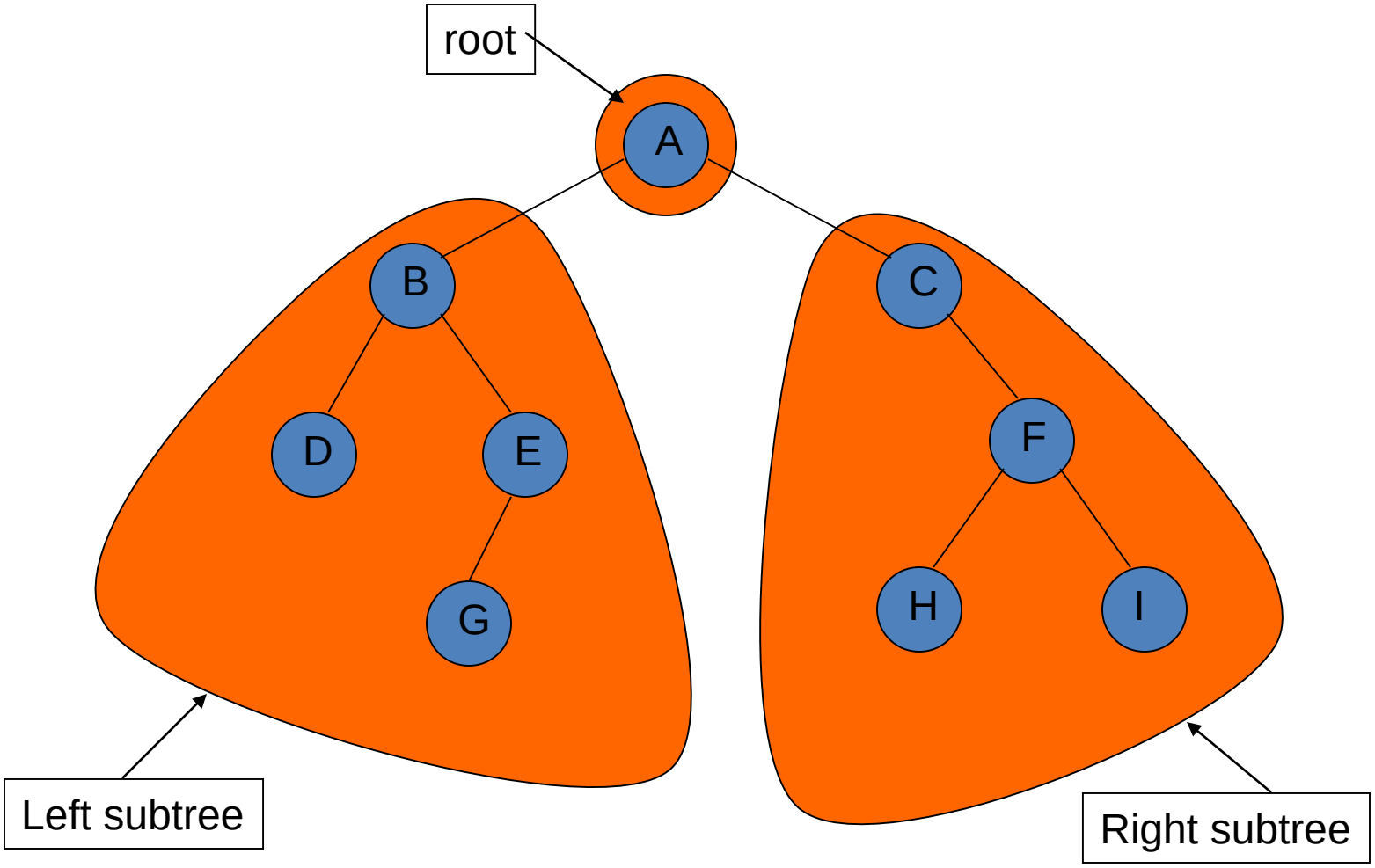


Binary Tree

- Binary tree with 9 nodes.

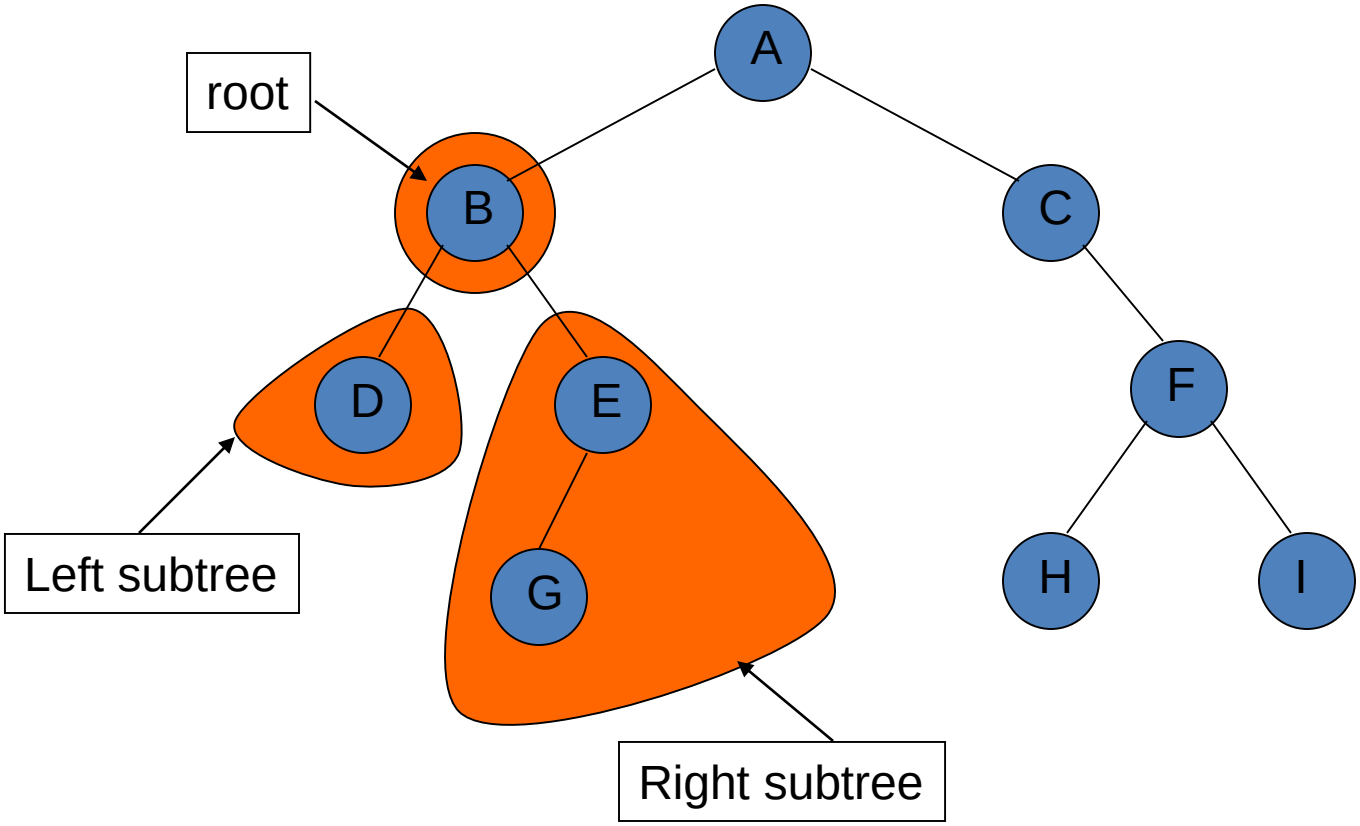


Binary Tree



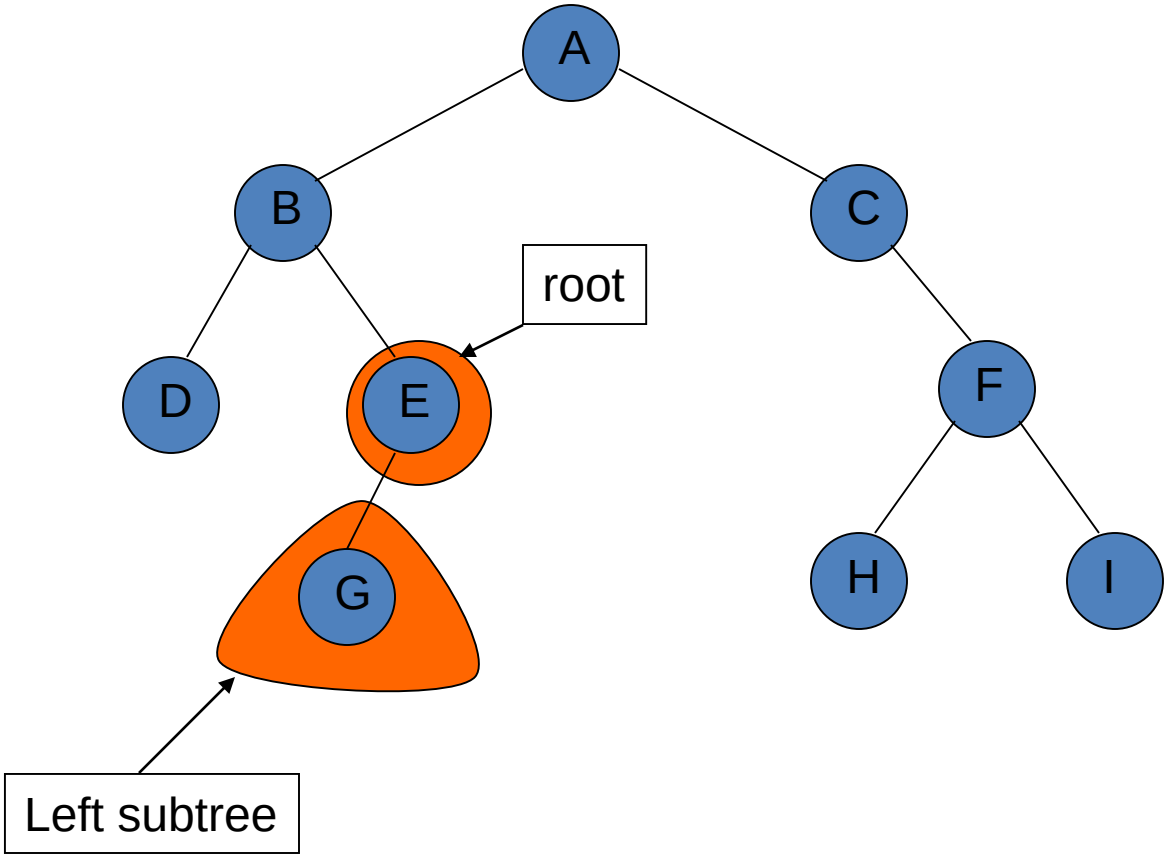
Binary Tree

- Recursive definition



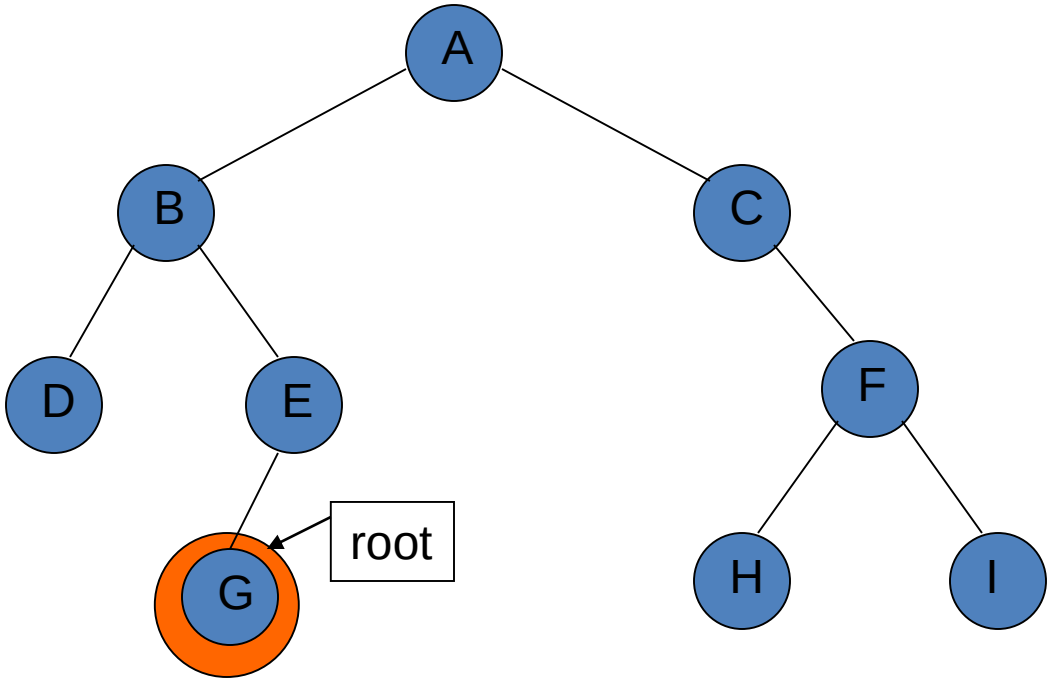
Binary Tree

- Recursive definition



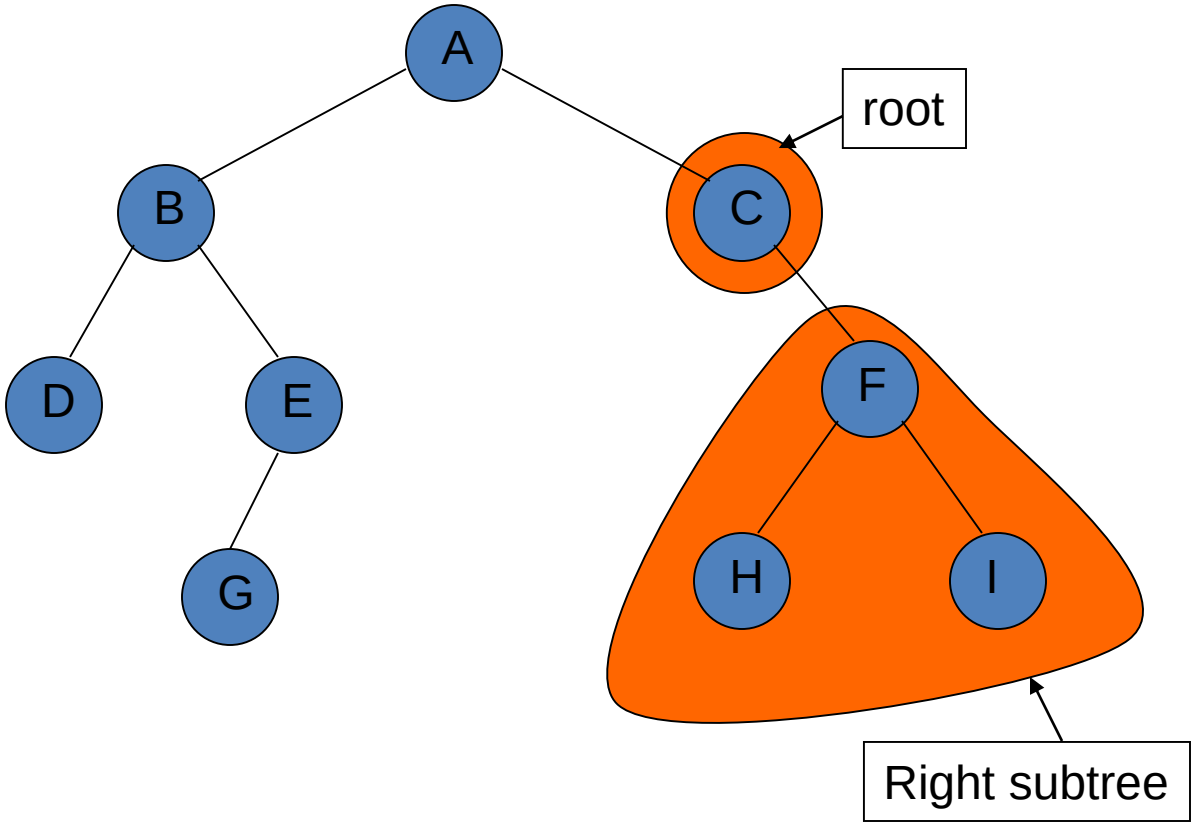
Binary Tree

- Recursive definition



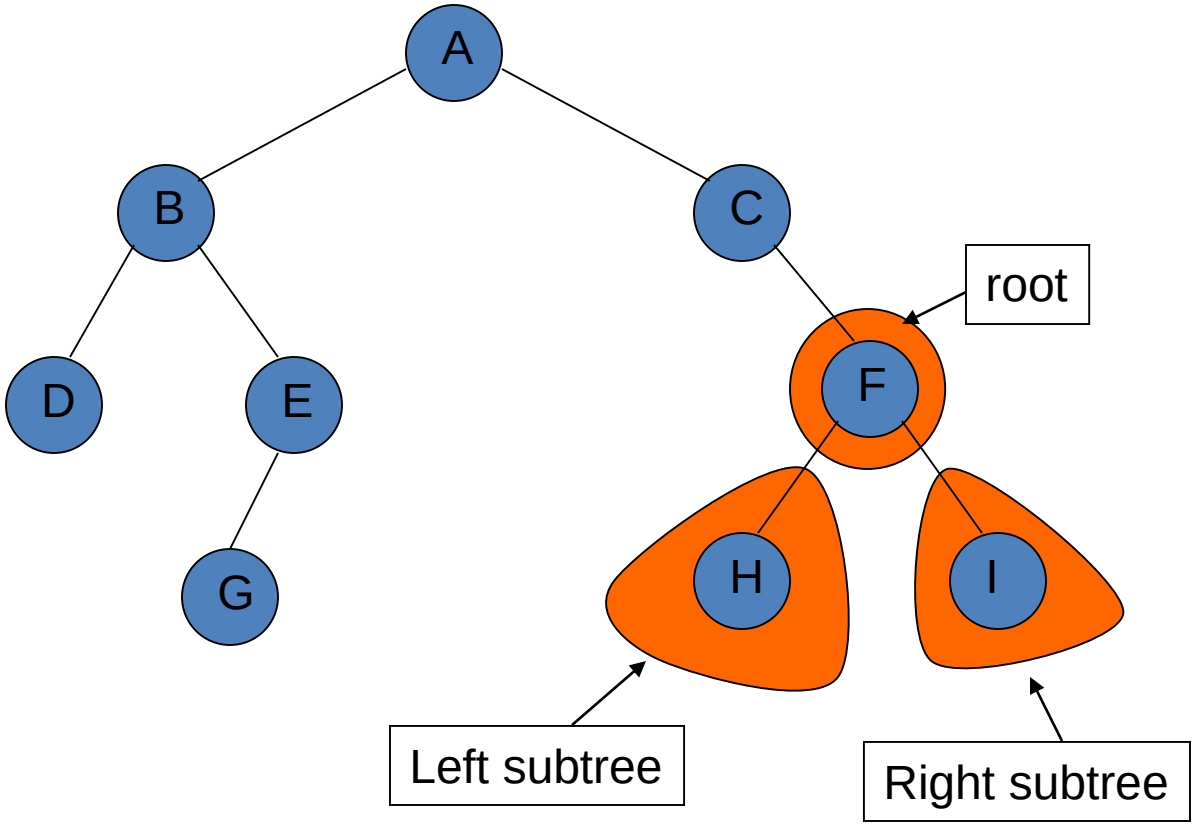
Binary Tree

- Recursive definition



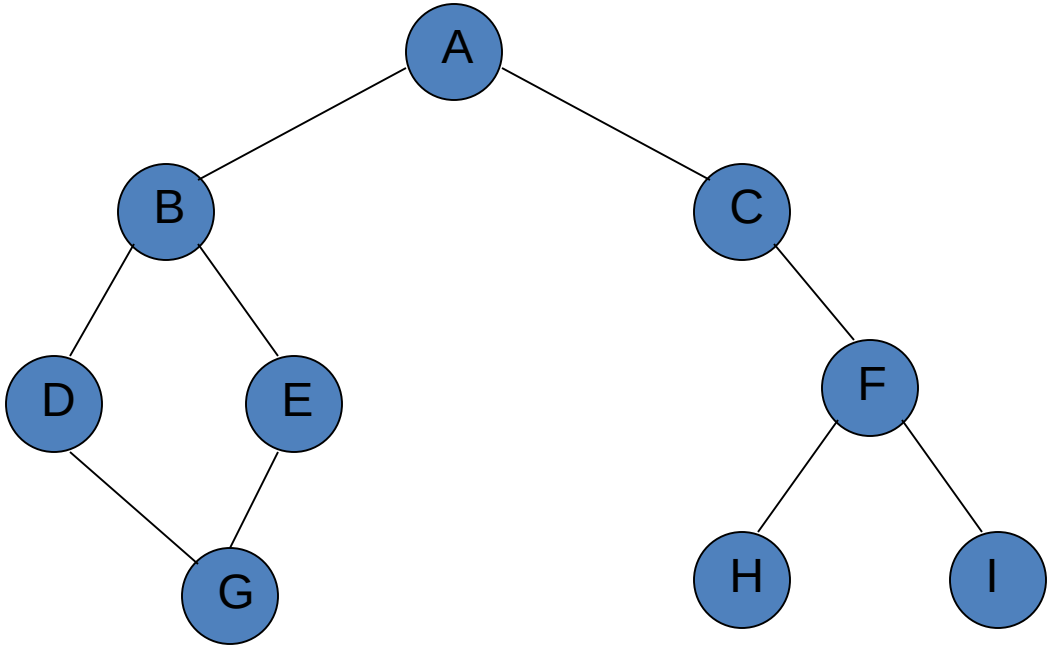
Binary Tree

- Recursive definition



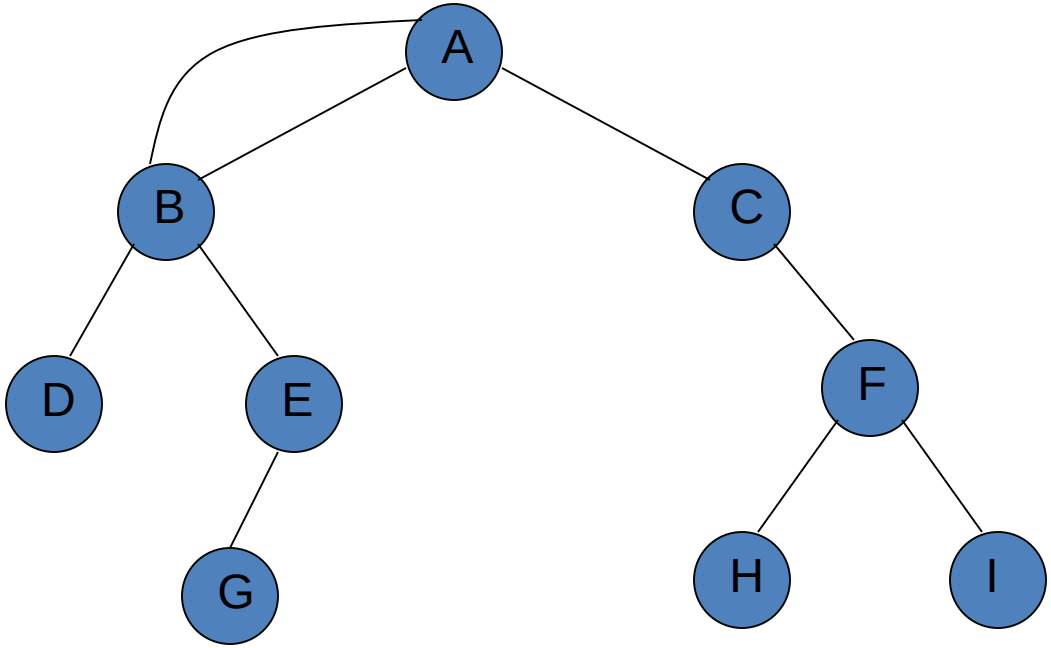
Not a Tree

- Structures that are not trees.



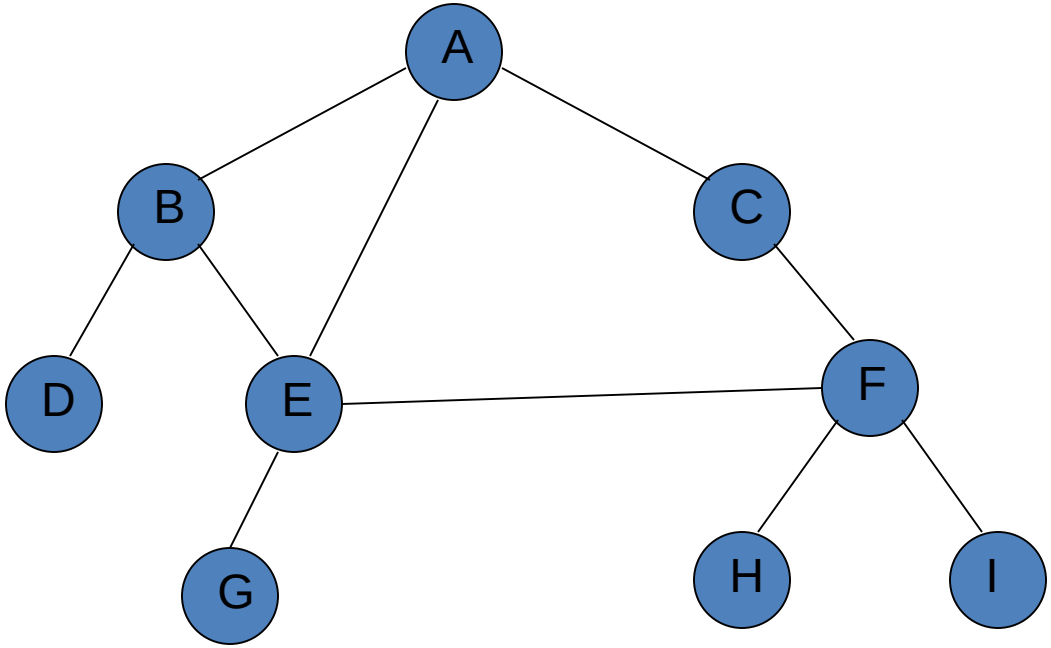
Not a Tree

- Structures that are not trees.

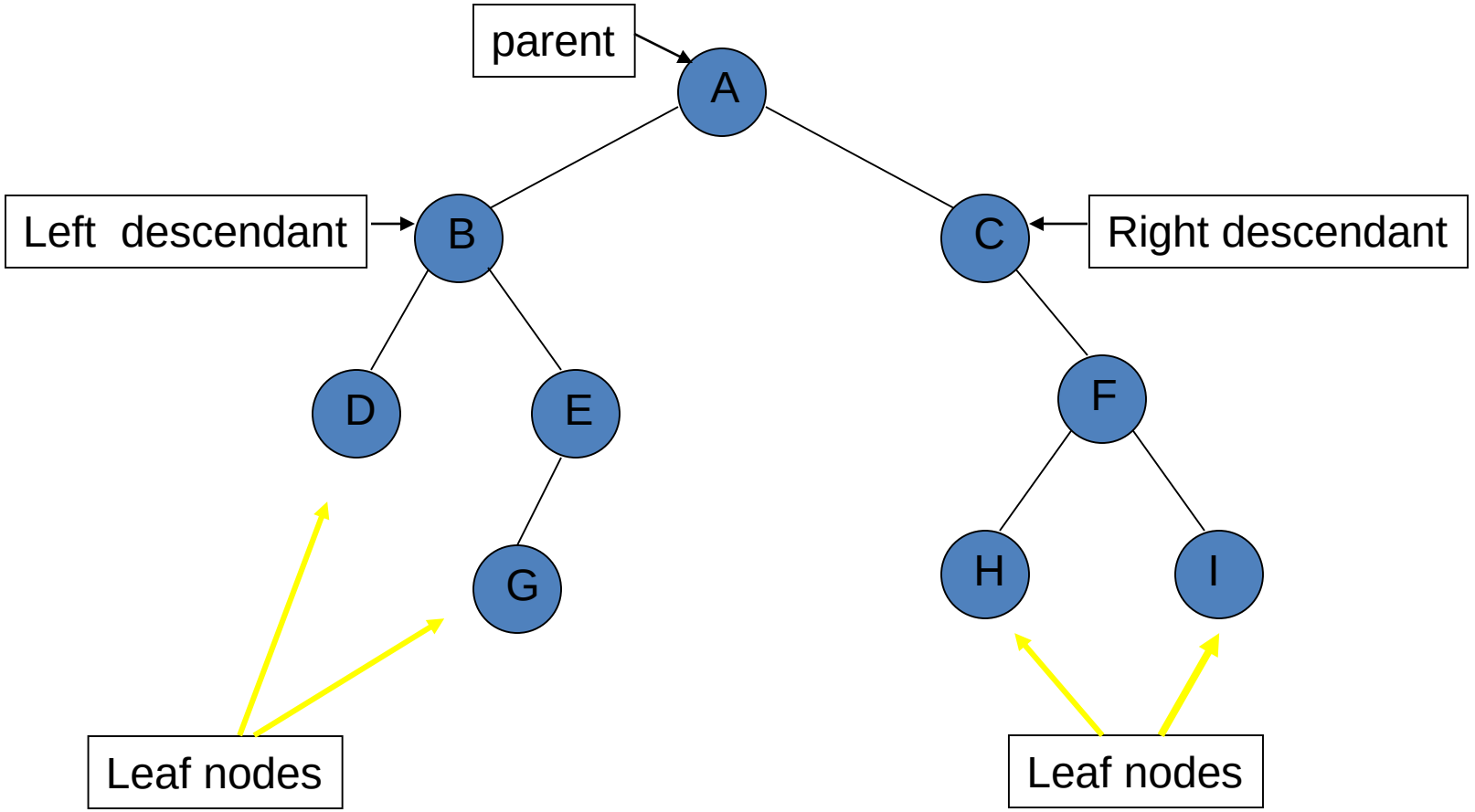


Not a Tree

- Structures that are not trees.

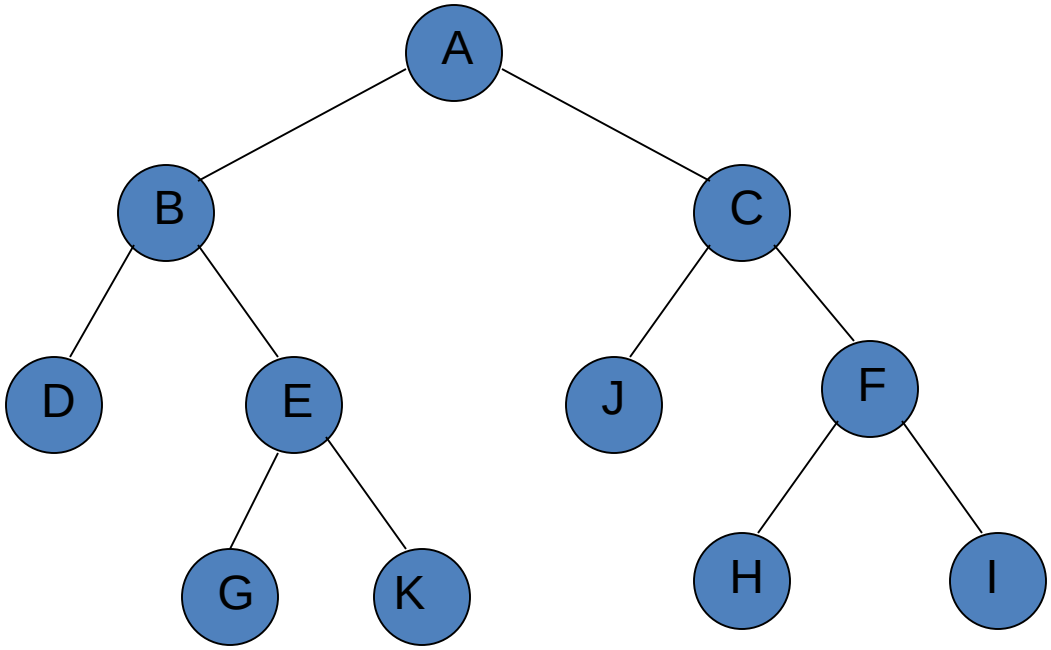


Binary Tree: Terminology



Binary Tree

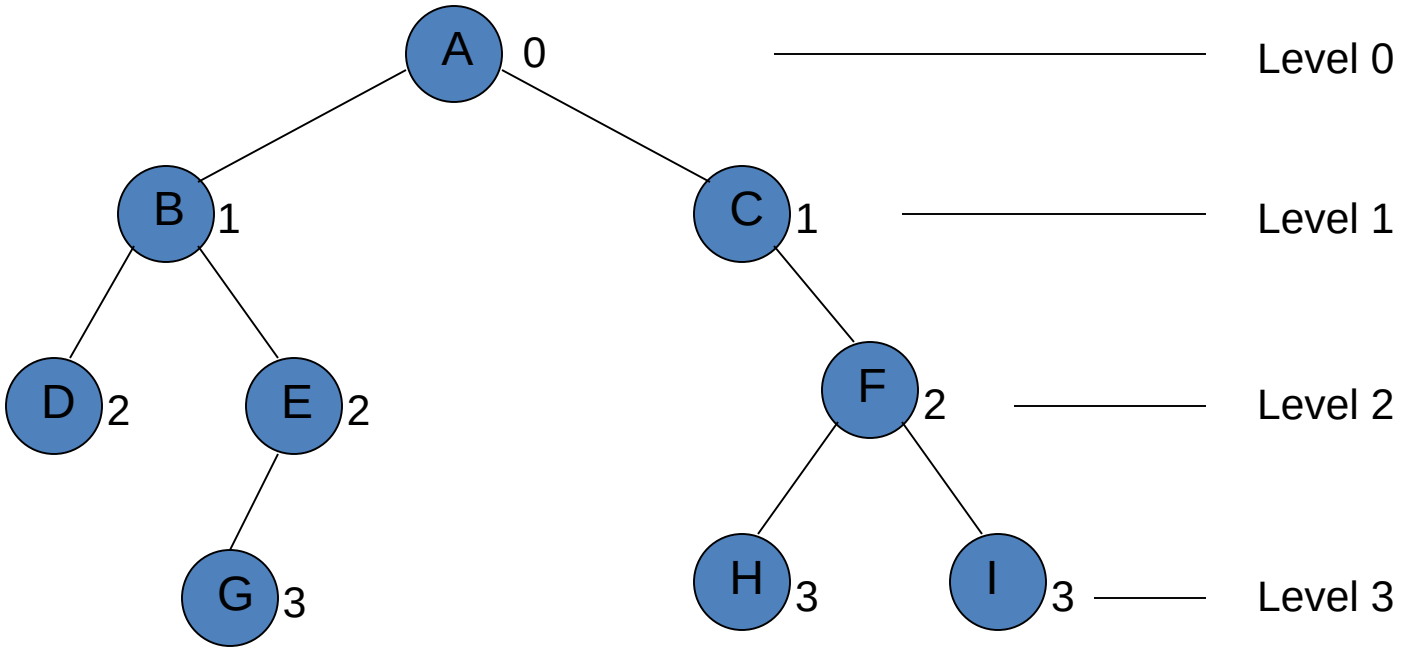
- If every ***non-leaf node*** in a binary tree has non-empty left and right subtrees, the tree is termed a *strictly binary tree*.



Level of a Binary Tree Node

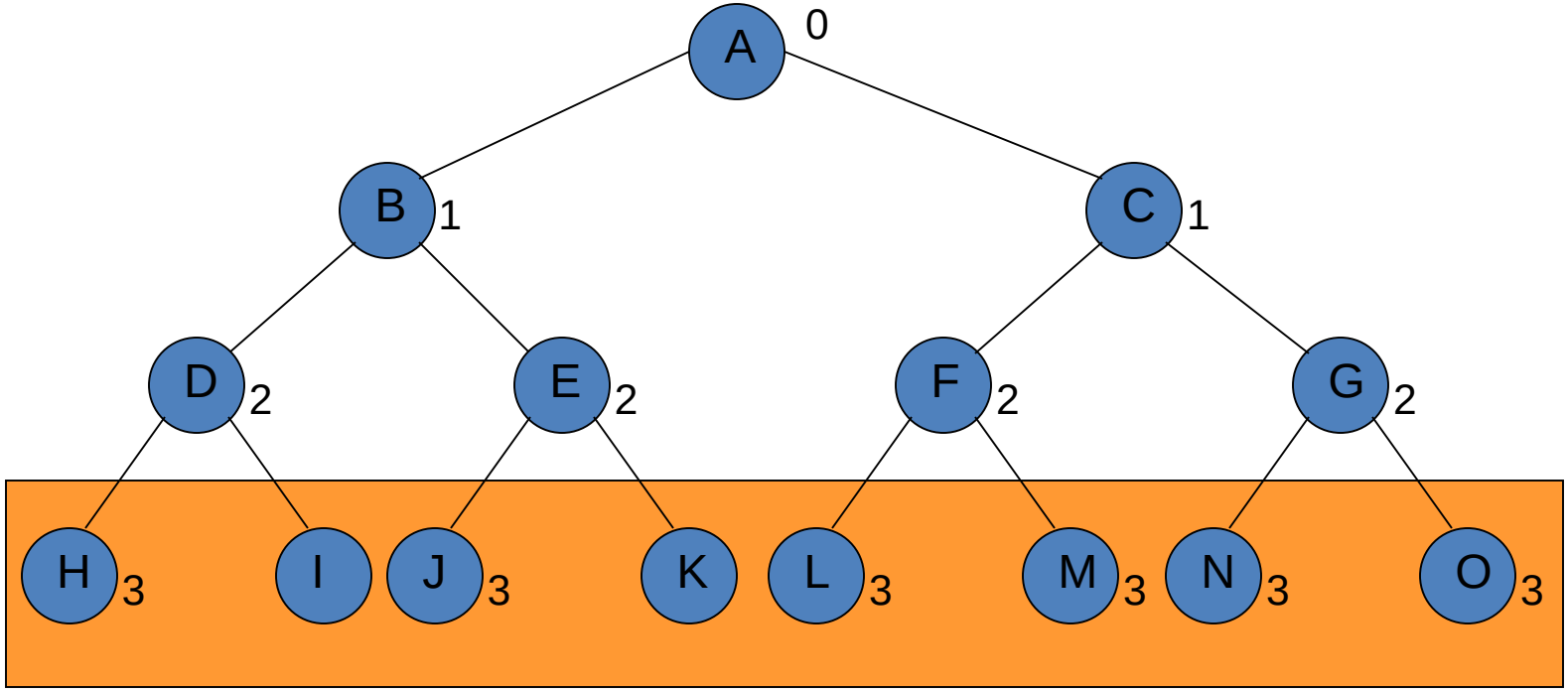
- The *level* of a node in a binary tree is defined as follows:
  - Root has level 0,
  - Level of any other node is one more than the level its parent.
- The *depth* of a binary tree is the maximum level of any leaf in the tree.

Level of a Binary Tree Node

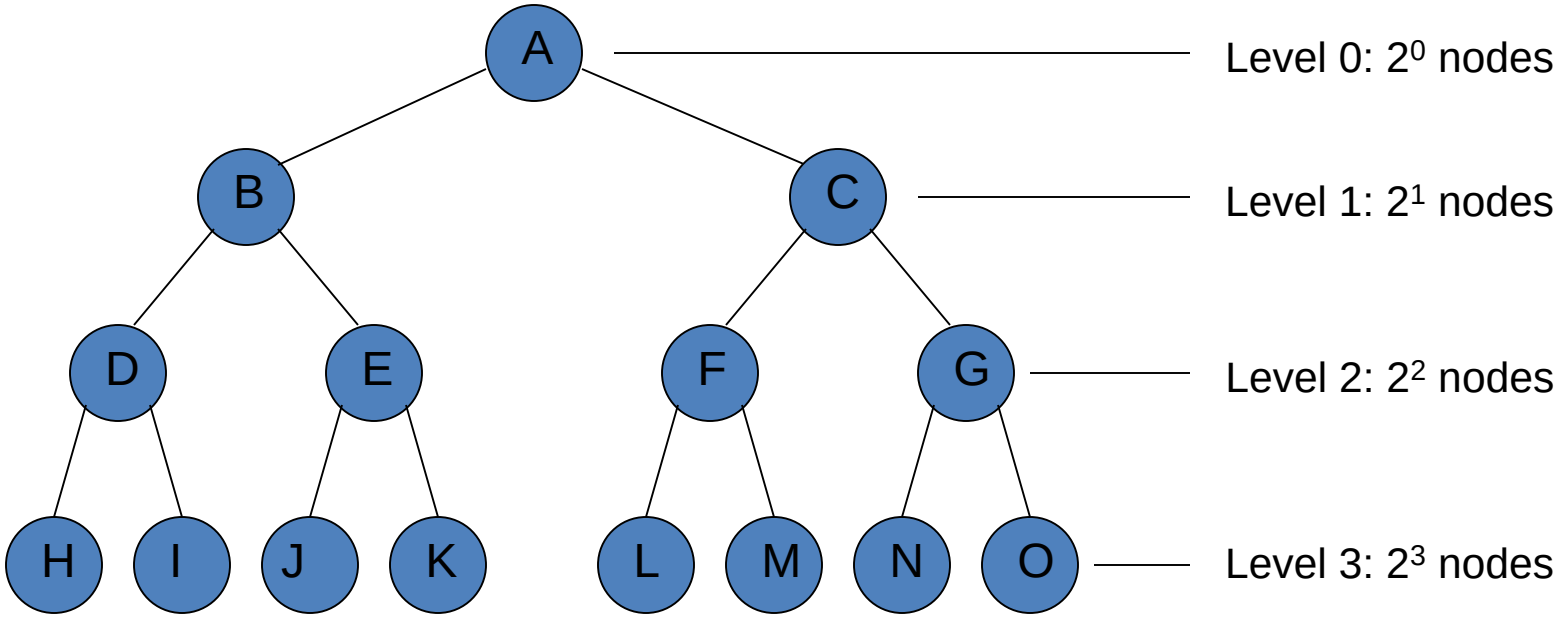


Complete Binary Tree

- A *complete binary tree* of depth *d* is the strictly binary all of whose leaves are at level *d*.



Complete Binary Tree



Complete Binary Tree

- At level k, there are 2^k nodes.
- Total number of nodes in the tree of depth d:

$$2^0 + 2^1 + 2^2 + \ldots\ldots\ldots + 2^d = \sum_{j=0}^{d} 2^j = 2^{d+1} - 1$$

- In a complete binary tree, there are 2^d leaf nodes and $(2^d - 1)$ non-leaf (inner) nodes.

Complete Binary Tree

- If the tree is built out of ‘n’ nodes then

$$n = 2^{d+1} - 1$$

or $log_2(n+1) = d+1$

or $d = log_2(n+1) - 1$

- I.e., the depth of the complete binary tree built using ‘n’ nodes will be $log_2(n+1) - 1$.
- For example, for n=1,000,000, $\log_2(1000001)$ is less than 20; the tree would be 20 levels deep.
- The significance of this shallowness will become evident later.

Operations on Binary Tree

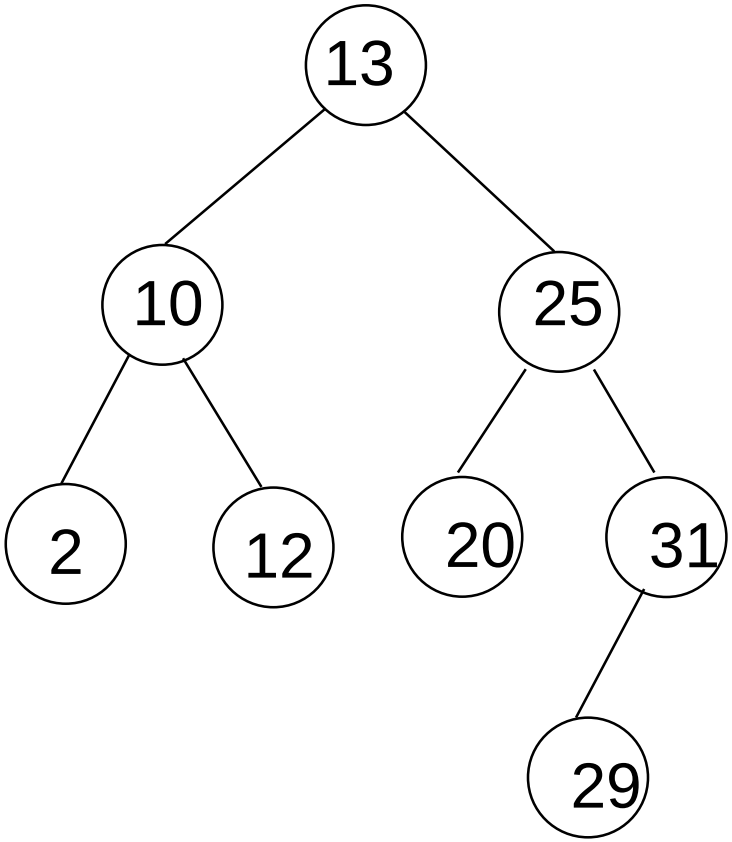
- There are a number of operations that can be defined for a binary tree.
- If *p* is pointing to a node in an existing tree then
  - left(*p*) returns pointer to the left subtree
  - right(*p*) returns pointer to right subtree
  - parent(*p*) returns the parent of *p*
  - sibling(*p*) returns sibling of *p*.
  - info(p) returns content of the node.

Implementation of Binary Tree

- Binary trees can be implemented in at least two ways
  - As arrays
  - As linked structures

Binary Tree as Array

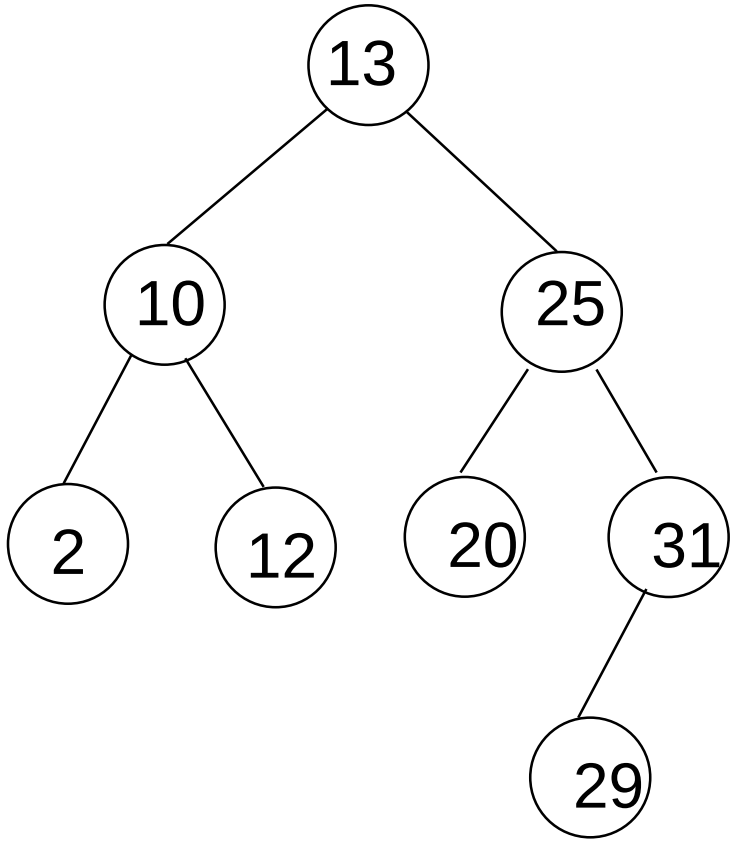
- **0 index based sequencing**
  - The root is always located in the first cell, cell 0, and –1 indicates a null child
  - The key concept is that, if a parent is stored in location k, then its left and right child are located in locations 2k+1 and 2k+2 respectively
- **1 index based sequencing**
  - The root is always located in the second cell, cell 1, and –1 indicates a null child
  - The key concept is that, if a parent is stored in location k, then its left and right child are located in locations 2k and 2k+1 respectively


13
10
25
2
12
20
31
29

Binary Tree: As Array

0 index based sequencing

Index	Info	Left	Right
0	13	1	2
1	10	3	4
2	25	5	6
3	2		
4	12		
5	20		
6	31	13	
7			
8			
9			
10			
11			
12			
13	29		



```
#define size 500
class BinaryTree
{
public:
      int treenodes[size];
      BinaryTree(void);
      bool isleaf(int nodeindex);      // returns true if the node is a leaf
      int parent(int nodeindex);     // returns the index of the parent
      int leftchild(int nodeindex);     // returns the index of the left child
      int rightchild(int nodeindex);   // returns the index of the right child
      bool rootnode(int index, int value);
      bool addleftchild(int value, int parent);
      bool addrightchild(int value,int parent);
};
```

```
BinaryTree::BinaryTree(void) // builds the "constructor"
{

     for(int i = 0; i < 500; i++)
         treenodes[i] = -1; // set all nodes empty
}
bool BinaryTree::isleaf(int nodeindex)
{
        return leftchild(nodeindex)==-1 &&
    rightchild(nodeindex)==-1;
}
int BinaryTree::parent(int nodeindex)
{
      int rv = (nodeindex - 1) / 2;
      return (rv >= size || treenodes[rv] == -1)? -1 : rv;
}
```

```
bool BinaryTree::rootnode(int value)
{
      treenodes[0] =value;
      return true;

}
bool BinaryTree::addleftchild(int value, int parent)
{
      int index =   (2*parent)+1;
      if(index<size)
       {
         treenodes[index] = value;
          return true;

      }
      else
      {
                cout<<“can’t set child”;
                return false;
      }

}
```

```
bool BinaryTree::addrightchild(int value, int parent)
{
    int index =   (2*parent)+2;
     if(index<size)
     {
       treenodes[index] = value;
        return true;


    }
    else
    {
            cout<<“can’t set child”;
            return false;
    }

}
```

```
int BinaryTree::leftchild(int nodeindex) // returns the index
    of the left child
{
        int rv =   (2*nodeindex)+1;
        return (rv >= size || treenodes[rv] == -1)? -1 : rv;
}
int BinaryTree::rightchild(int nodeindex) // returns the
    index of the right child
{
         int rv =  (2*nodeindex)+2;
         return (rv >= size || treenodes[rv] == -1)? -1 : rv;
}
```

IMPLEMENTATION

Binary Tree Implementation Using Pointers

- We use two classes:
  - **TreeNode**
  - **BinaryTree**
- Declare `TreeNode` class for the nodes
  - `info`: `int`-type data in this example

```
class TreeNode {
private:
        int                 info;          // data
        TreeNode*           left;          // pointer to Left
        TreeNode*           right;         // pointer to Right
        TreeNode*           parent;        // pointer to Parent




};
```

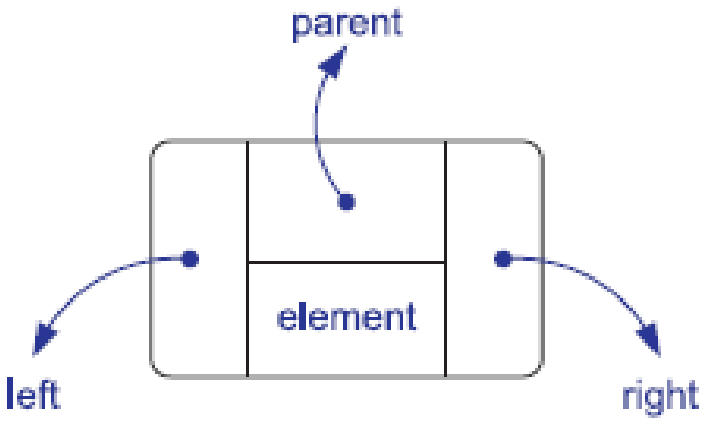


Figure 7.13: A node in a linked data structure for representing a binary tree.

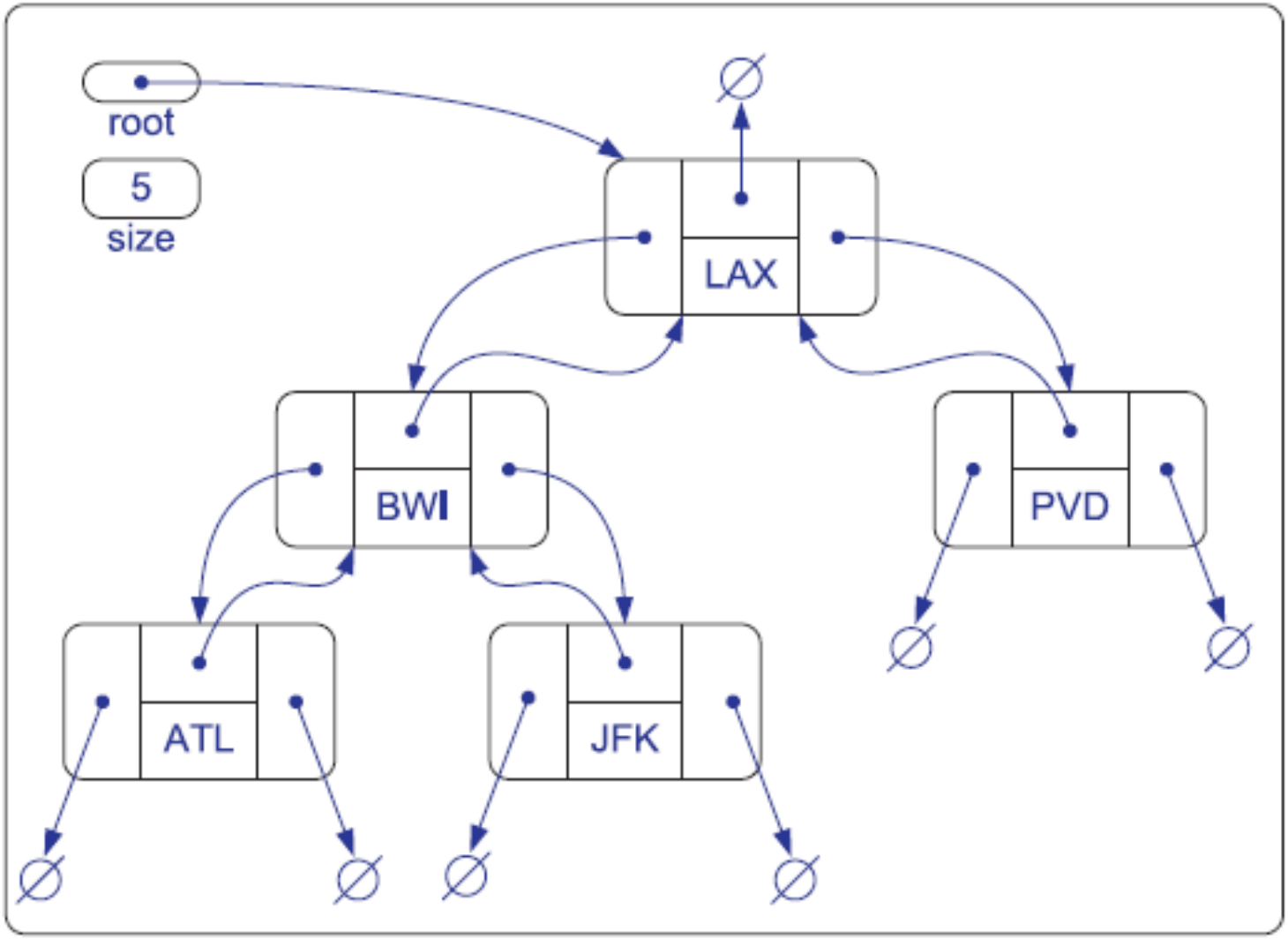


Figure 7.14: An example of a linked data structure for representing a binary tree.

Binary Tree Implementation Using Pointers

```
class TreeNode {
public:
        //constructors
        TreeNode();
        TreeNode( int info);
        int getInfo();
       void setInfo(int info);
        TreeNode* getLeft();
        void setLeft(TreeNode *left);
        TreeNode *getRight();
        void setRight(TreeNode *right);
        int isLeaf( );
        TreeNode* getParent();
        void setParent(TreeNode *parent);



private:
       int    info;          // data
       TreeNode*             left;          // pointer to Left
       TreeNode*             right;         // pointer to Right
       TreeNode*             parent;        // pointer to Parent


};
```

TreeNode Constructors

```
TreeNode::TreeNode()
{
        this->info= -1;
        this->left = this->right = NULL;

}
TreeNode:: TreeNode( int info)
{
        this->info= info;
        this->left = this->right = NULL;

}
```

TreeNode Methods

```
int TreeNode::getInfo()
{
        return this->info;
}
void TreeNode::setInfo(int info)
{
        this->info = info;

}
TreeNode* TreeNode::getLeft()
{
        return this->left;

}
void TreeNode::setLeft (TreeNode *left)
{
        this->left = left;
}
```

TreeNode Methods

```
TreeNode* TreeNode::getRight()
{
        return this->right;

}
void TreeNode::setRight(TreeNode *right)
{

    this->right = right;

}
int TreeNode::isLeaf( )
{

        if( this->left == NULL && this->right == NULL )
                    return 1;
        return 0;
}
TreeNode* TreeNode::getParent()
{
        return this->parent;

}
void TreeNode::setParent(TreeNode *parent)
{
        this->parent = parent;


}
```

```
class BinaryTree{
public:
     void insert(TreeNode* root, int info);
     bool search(int key);
     bool delete(int key);



private:
      int     info;          // data
      TreeNode*              left;        // pointer to Left
      TreeNode*              right;             // pointer to
      Right
      TreeNode*              parent;      // pointer to
      Parent


};
```

Lecture content adapted from Michael T. Goodrich textbook, chapters 7.