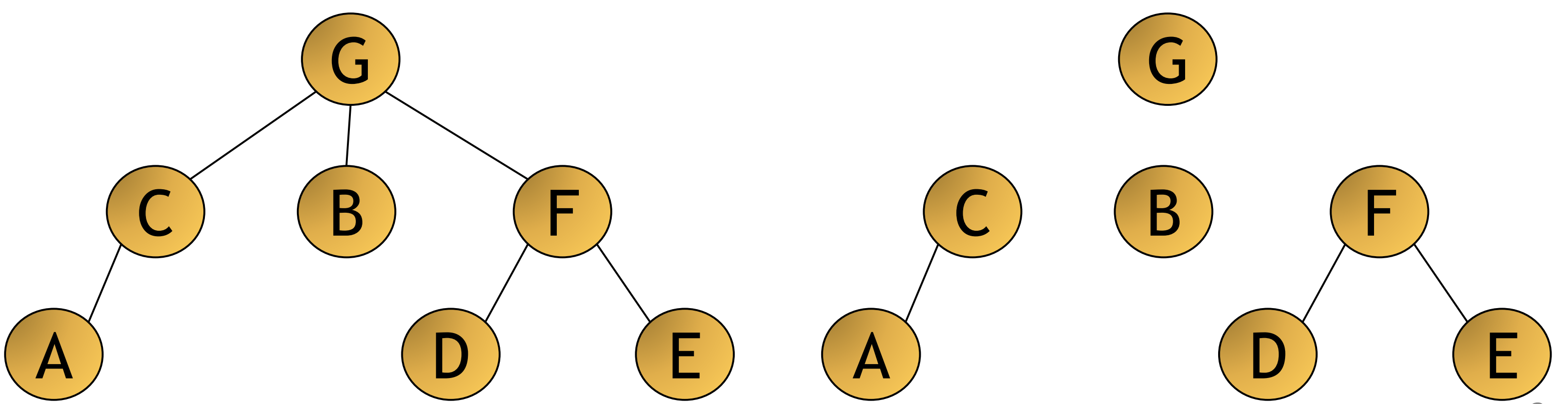
Chap 5. Trees (1)

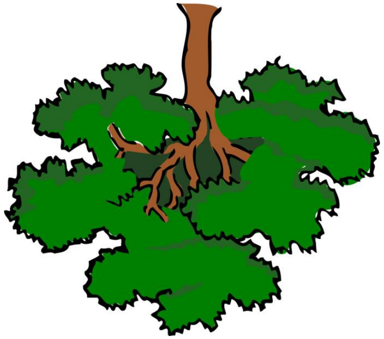
Contents

5.1 Introduction

5.1.1 Terminology

- ***Definition*** **:** A ***Tree*** is a finite set of *one or more nodes* such that
  1. There is a specially designated node called the *root*
  2. The remaining nodes are partitioned into $n \geq 0$ *disjoint sets* $T_1, \ldots, T_n$, where each of these sets is a tree. We call $T_1, \ldots, T_n$ the *subtrees* of the root.





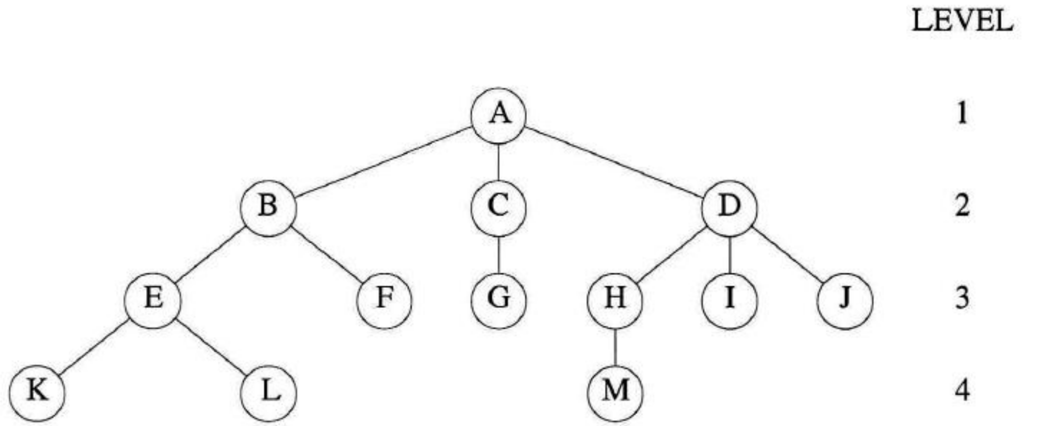
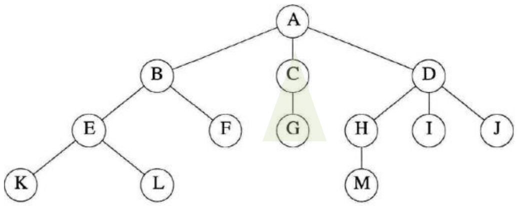


Figure 5.2: A sample tree

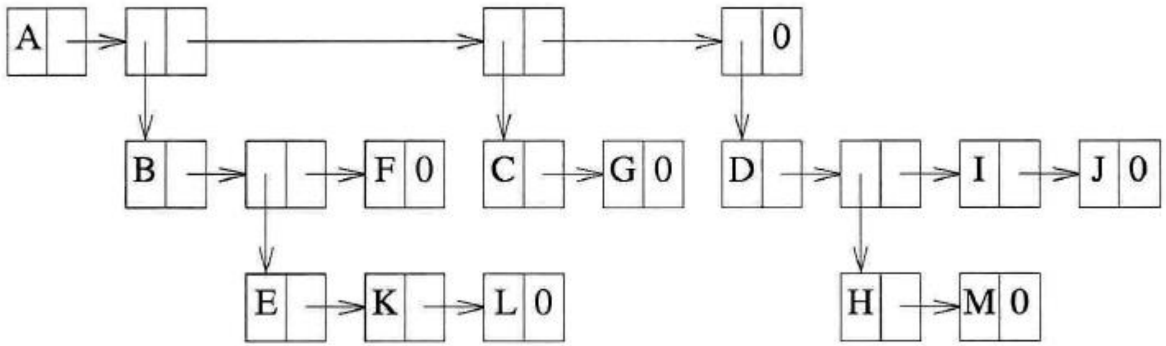
- degree of a node : number of subtrees of the node
- degree of a tree : maximum degree of the nodes in the tree
- leaf (terminal node) : a node with degree zero
- parent, children
- siblings : children of same parent
- grandparent, grandchildren
- ancestors of a node : all the nodes along the path from the root to the node
- descendants of a node : all the nodes that are in its subtrees
- level of a node
- height (depth) of a tree : maximum level of any node in the tree

5.1.2 Representation of Trees

- List Representation



(A (B (E (K, L), F), C (G), D(H (M), I, J)))

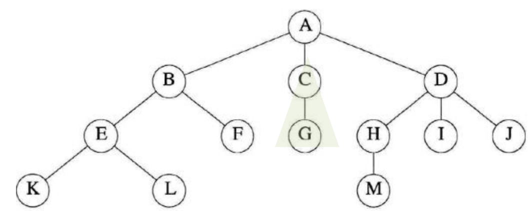


tag fields not shown

Figure 5.3: List representation of the tree of Figure 5.2

5.1.2 Representation of Trees

- A representation that is specialized to tree
  - represent each tree node that has fields for data and pointers to the node's children.



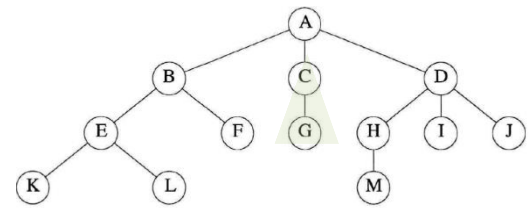
DATA	CHILD 1	CHILD 2	···	CHILD k

Figure 5.4: Possible node structure for a tree of degree k

Lemma 5.1 : If T is a k-ary tree (i.e., a tree of degree k) with n nodes, each having a fixed size as in Figure 5.4, then $n(k-1)+1$ of the nk child fields are 0, $n \geq 1$.

DATA	CHILD 1	CHILD 2	⋯	CHILD k

Figure 5.4: Possible node structure for a tree of degree k

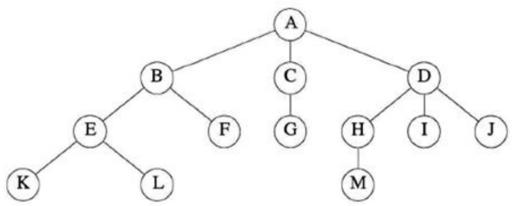


13 nodes
13*3 child fields

the number of non-zero child fields in an *n-node* tree is exactly $n-1$
The total number of child fields in a *k-ary* tree with n nodes is nk.
Hence, the number of zero fields is $nk-(n-1) = n(k-1)+1$.

- Left Child-Right Sibling Representation

Data	
Left Child	Right Sibling



→

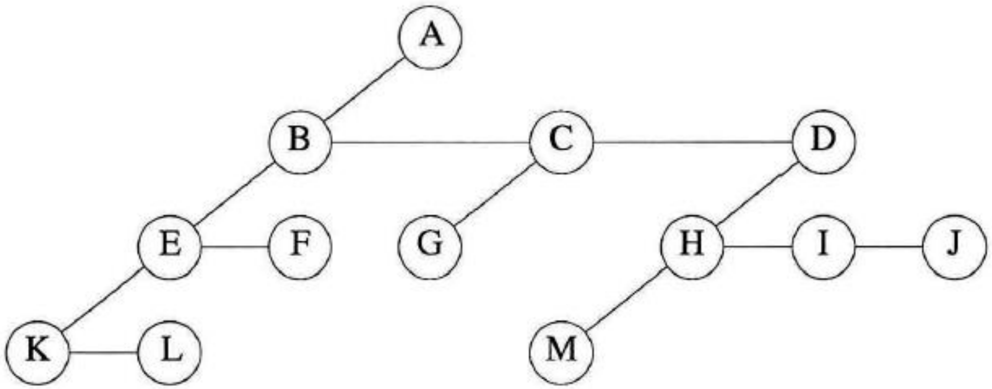
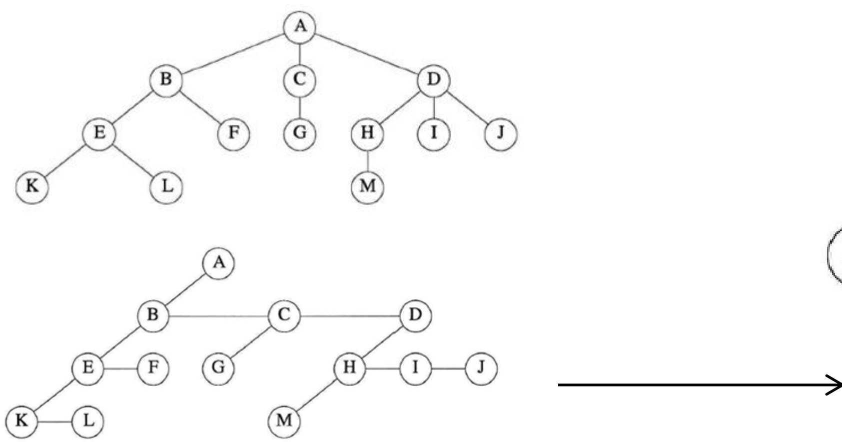


Figure 5.6: Left child-right sibling representation of tree of Figure 5.2

- Representation as a Degree Two Trees

Data	
Left Child	Right Child



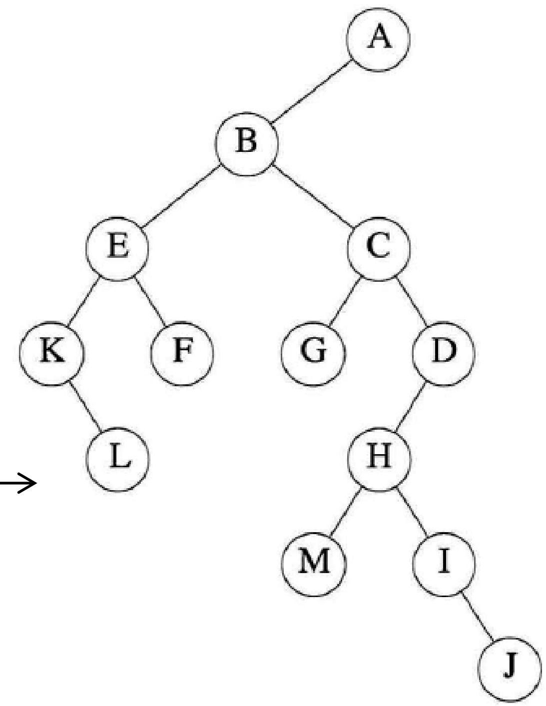


Figure 5.7: Left child-right child tree representation of tree of Figure 5.2

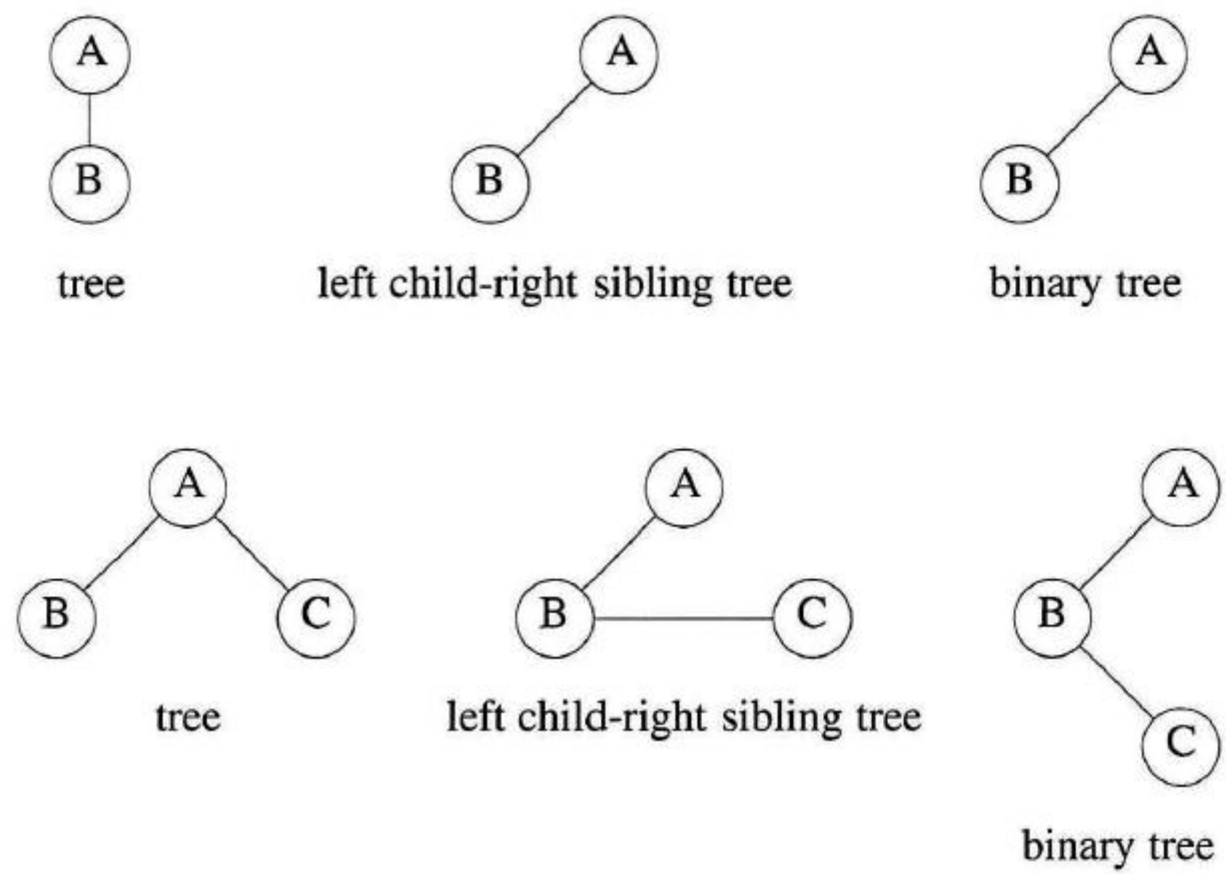


Figure 5.8: Tree representations

5.2 Binary Trees

5.2.1 The Abstract Data Type

Definition :

A ***Binary Tree*** is a finite set of nodes that is either *empty* or consists of a *root* and two disjoint binary trees called the *left subtree* and the *right subtree*.

ADT *Binary_Tree* (abbreviated *BinTree*) is

objects: a finite set of nodes either empty or consisting of a root node, left *Binary_Tree*, and right *Binary_Tree*.

functions:

for all $bt, bt1, bt2 \in BinTree$, $item \in element$

BinTree Create()	::=	creates an empty binary tree
Boolean IsEmpty(*bt*)	::=	**if** (*bt* == empty binary tree) **return** *TRUE* **else return** *FALSE*
BinTree MakeBT(*bt*1, *item*, *bt*2)	::=	**return** a binary tree whose left subtree is *bt*1, whose right subtree is *bt*2, and whose root node contains the data *item*.
BinTree Lchild(*bt*)	::=	**if** (IsEmpty(*bt*)) **return** error **else return** the left subtree of *bt*.
element Data(*bt*)	::=	**if** (IsEmpty(*bt*)) **return** error **else return** the data in the root node of *bt*.
BinTree Rchild(*bt*)	::=	**if** (IsEmpty(*bt*)) **return** error **else return** the right subtree of *bt*.

ADT 5.1: Abstract data type *Binary_Tree*

- Differences between a ***tree*** & a ***binary tree***
  1. There is no tree having zero nodes, but there is an empty binary tree.
  2. In a *binary tree*, we distinguish between *the order of the children* while in a tree we do not.

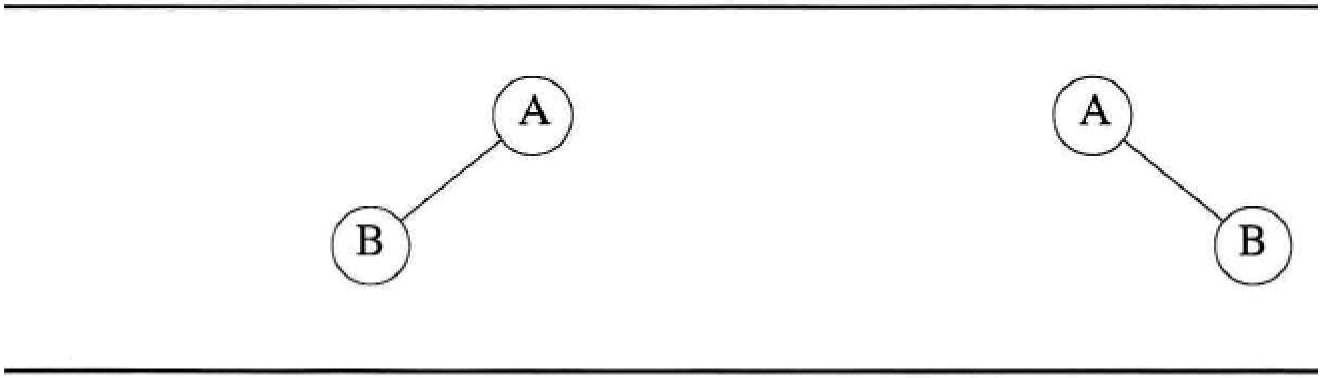


Figure 5.9: Two different binary trees

Viewed as tree, They are same.

5.2.2 Properties of Binary Trees

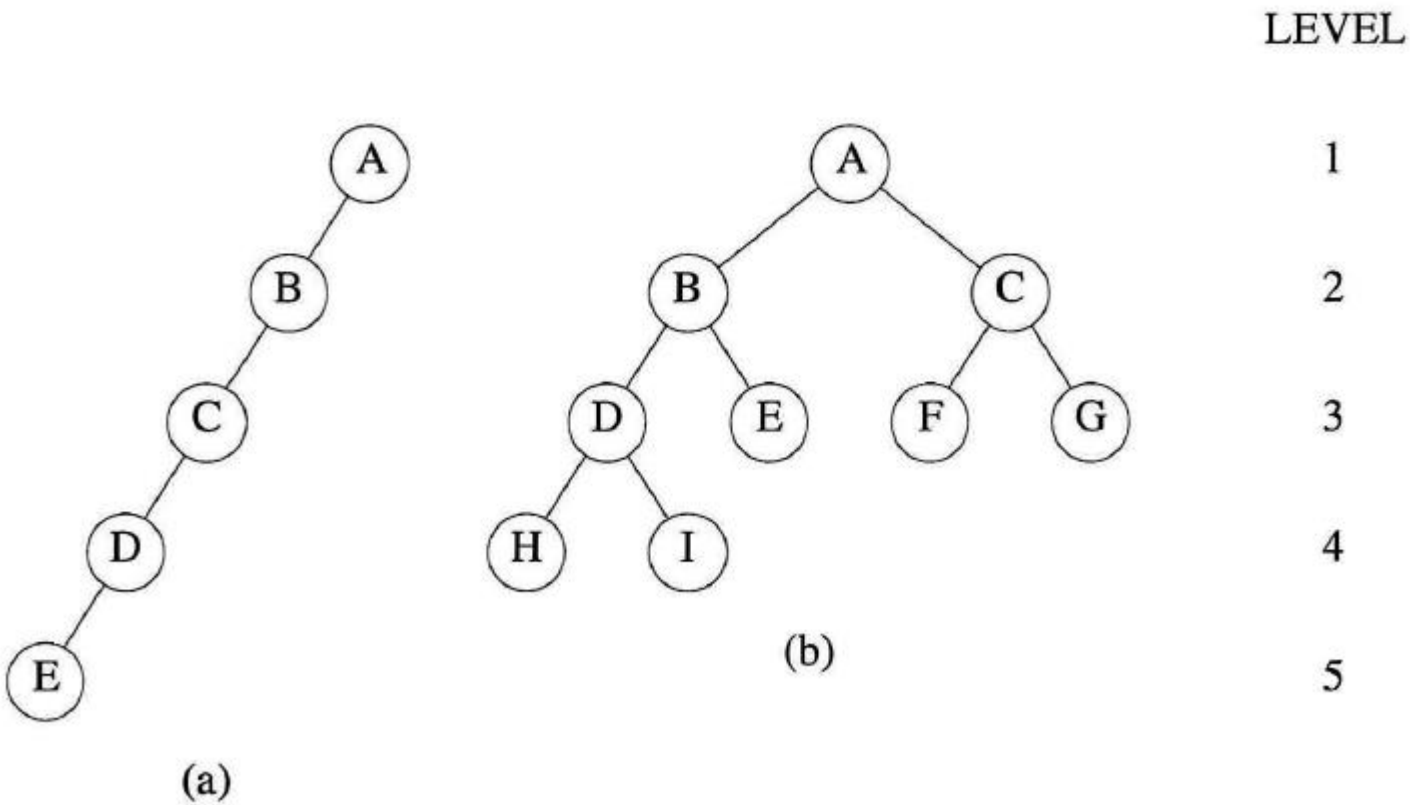


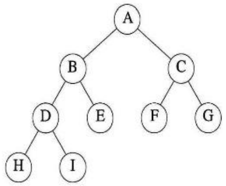
Figure 5.10: Skewed and complete binary trees

Caution

- Some texts start level numbers at 0.
  - Root is at level 0.
  - Its children are at level 1.
  - The grand children of the root are at level 2.
  - And so on.

- *We shall number levels with the root at level 1*.

5.2.2 Properties of Binary Trees



Lemma 5.2 [*Maximum number of nodes*]

1. The maximum number of nodes on level i of a binary tree is $\mathbf{2^{i-1}}$, $i \geq 1$.
2. The maximum number of nodes in a binary tree of depth k is $\mathbf{2^k - 1}$, $k \geq 1$

Proof

1. Induction Base: $i = 1 \Rightarrow$ The max. # of nodes on level 1 is $2^{i-1} = 2^0 = 1$

   Induction Hypothesis: $1 < i \Rightarrow$ The max. # of nodes on level i-1 is 2^{i-2}

   Induction Step: The max. # of nodes at level i
   = (The max. # of nodes at level i-1) $\times 2$
   = $2^{i-2} \times 2 = 2^{i-1}$

2. $$\sum_{i=1}^{k} (\text{maximum number of nodes on level } i) = \sum_{i=1}^{k} 2^{i-1} = 2^k - 1$$

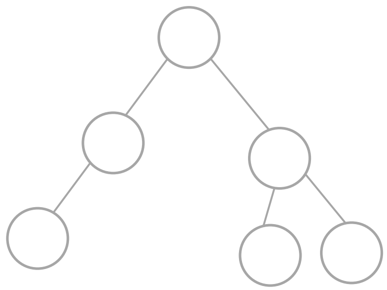
Lemma 5.3 [*Relation between number of leaf nodes and degree-2 nodes*]:

For any nonempty binary tree T, if n_0 is the number of leaf nodes and n_2 the number of nodes of degree 2, then $\boldsymbol{n_0 = n_2+1}$.

Proof

n: the total number of nodes

$$n = n_0 + n_1 + n_2 \qquad ①$$



B: the number of branches

$$n = B + 1, \;\; B = n_1 + 2n_2$$

$$n = B + 1 = n_1 + 2n_2 + 1 \qquad ②$$

$$n_0 = n_2 + 1 \qquad ①\text{-}②$$

Definition [*Full Binary Tree*] :

A *full binary tree* of depth k is a binary tree of depth k having $2^k - 1$ nodes, $k \geq 0$.

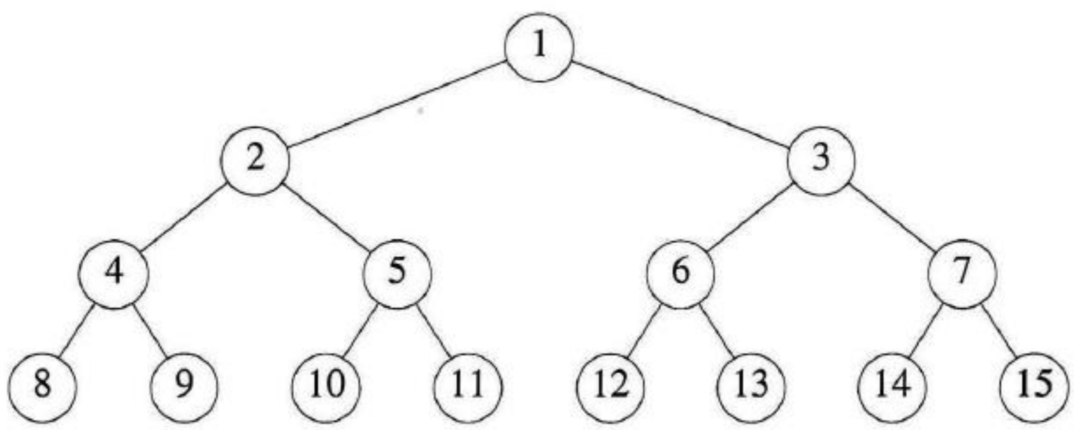
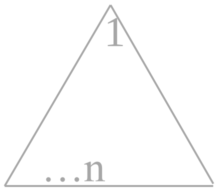
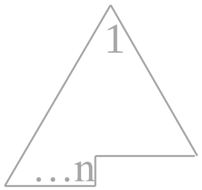


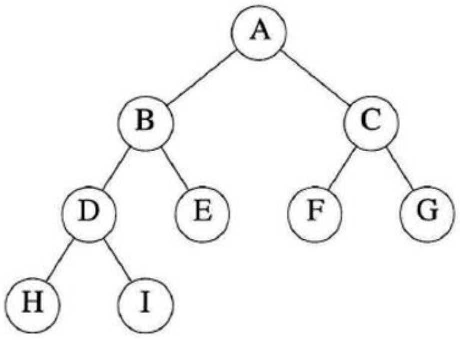
Figure 5.11: Full binary tree of depth 4 with sequential node numbers

Definition [*Complete Binary Tree*] :

A binary tree with n nodes and depth k is *complete* *iff* its nodes correspond to the nodes numbered from 1 to n in the full binary tree of depth k.







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

5.2.3 Binary Tree Representation

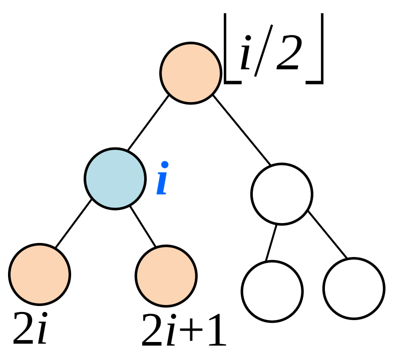
- **Array Representation**

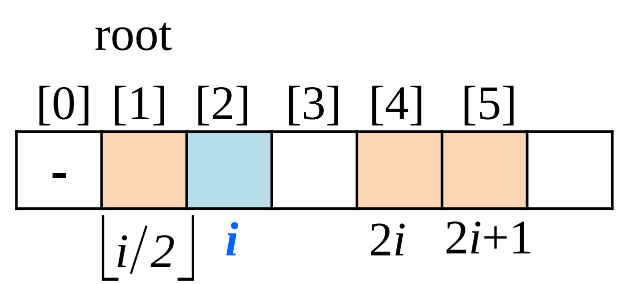
Lemma 5.4: If a complete binary tree with n nodes is represented sequentially, then for any node with index i, $1 \le i \le n$, we have

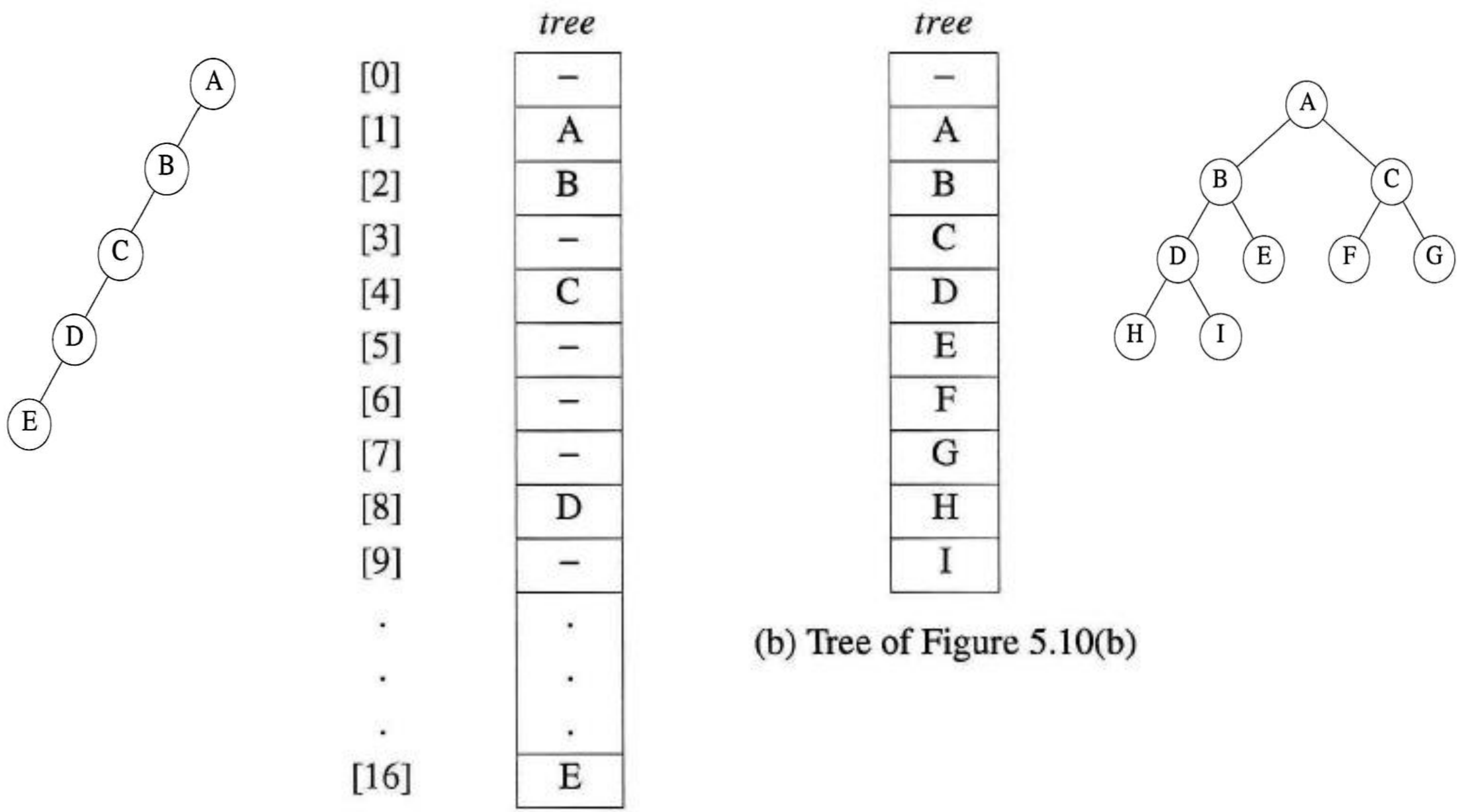
(1) *parent* (i) is at $\lfloor i/2 \rfloor$ if $i \ne 1$. If $i = 1$, i is at the root and has no parent.

(2) *leftChild* (i) is at $2i$ if $2i \le n$. If $2i > n$, then i has no left child.

(3) *rightChild* (i) is at $2i + 1$ if $2i + 1 \le n$. If $2i + 1 > n$, then i has no right child.







(a) Tree of Figure 5.10(a)

(b) Tree of Figure 5.10(b)

Figure 5.12: Array representation of the binary trees of Figure 5.10

- **Linked Representation**

```
typedef struct node *treePointer;
typedef struct node {
        int data;
        treePointer leftChild, rightChild;
        } node;
```

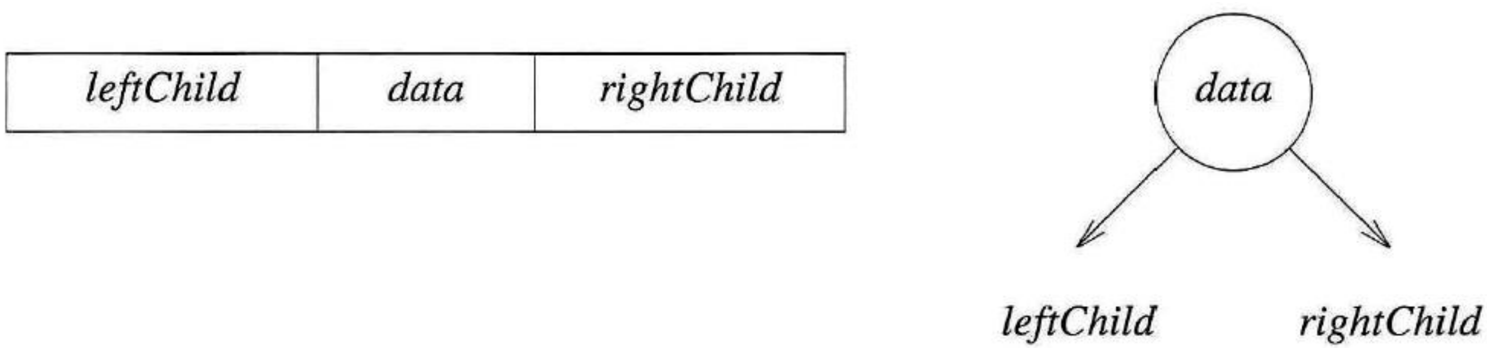


Figure 5.13: Node representations

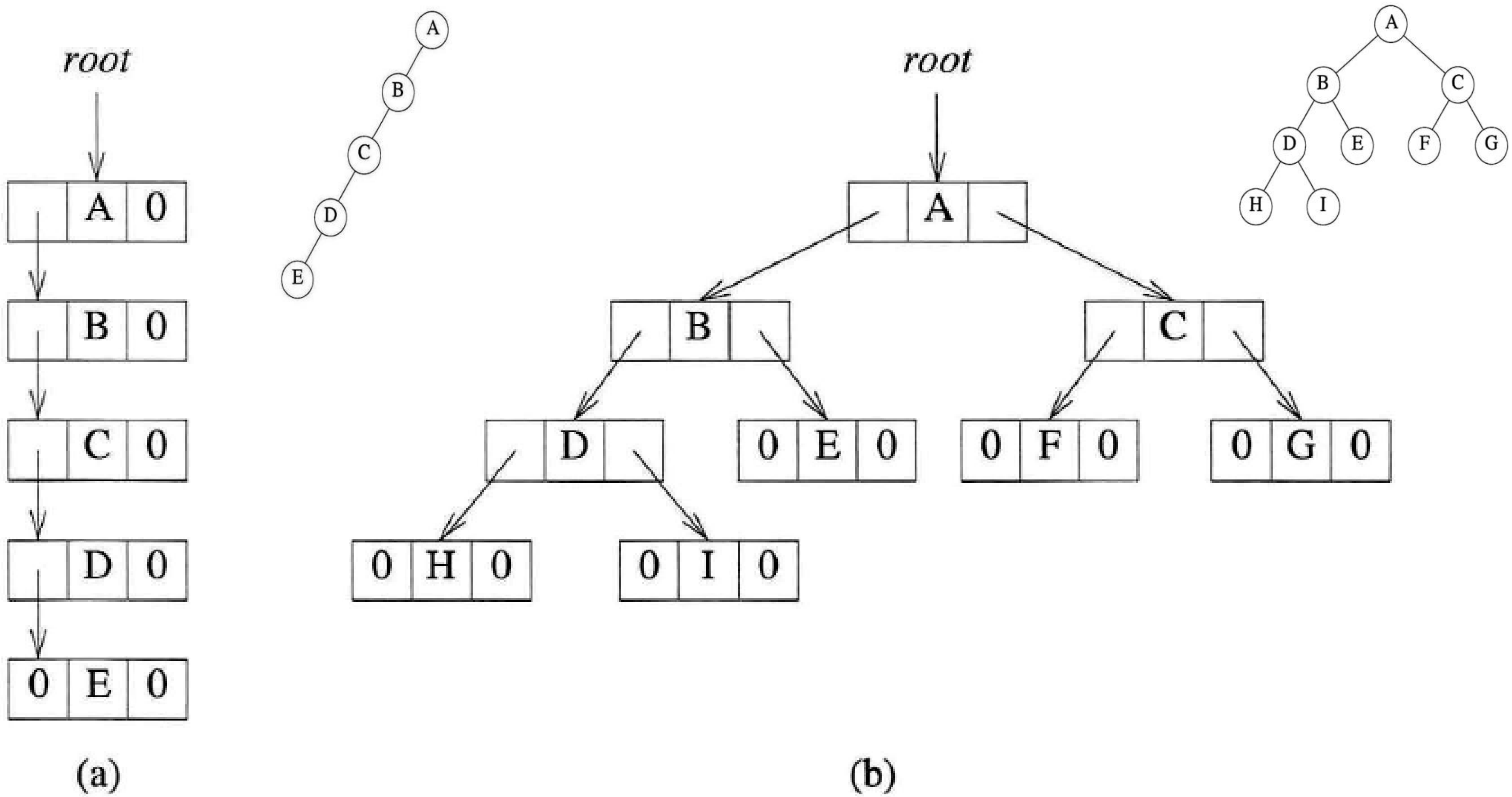


Figure 5.14: Linked representation for the binary trees of Figure 5.10

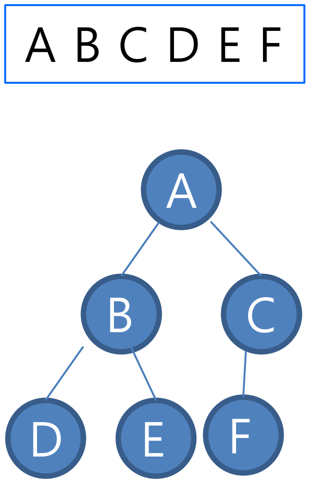
5.3 Binary Tree Traversal

- Traversing a tree
  - Visiting each node in the tree exactly once

- When traversing a binary tree,
  - L, V, R : *moving left*, *visiting the node*, *moving right*
  - Six possible combinations of traversal
    - LVR, LRV, VLR, VRL, RVL, RLV

  - If we traverse left before right, only tree remains
    - LVR: ***inorder***
    - LRV: ***postorder***
    - VLR: ***preorder***

Make a complete binary tree using Queue

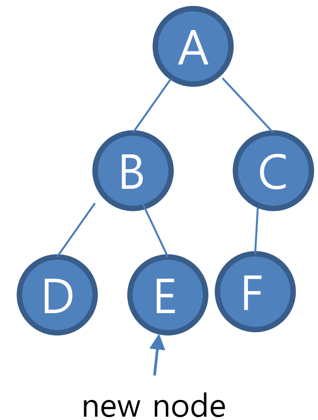


1. createCompBinTree
get item from input file and create a new node
while (!End of file) {
    insert a new node to a tree
    get item from input file and create a new node
}
free(node)

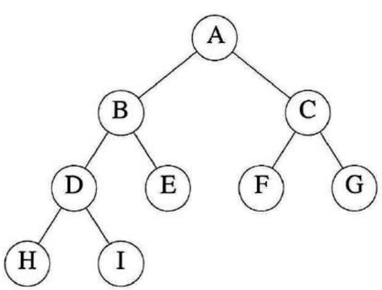
2. insert
1) *If* the tree is empty, initialize the root with *new node*.
2) *Else* {
        get the front node of the queue.
        if the left child of this front node doesn't exist,
            set the left child as the new node.
        else if the right child of this front node doesn't exist,
            set the right child as the new node.
        If the front node has both the left child and right child,
            Dequeue() it.
    }
3) **Enqueue()** the *new node*.

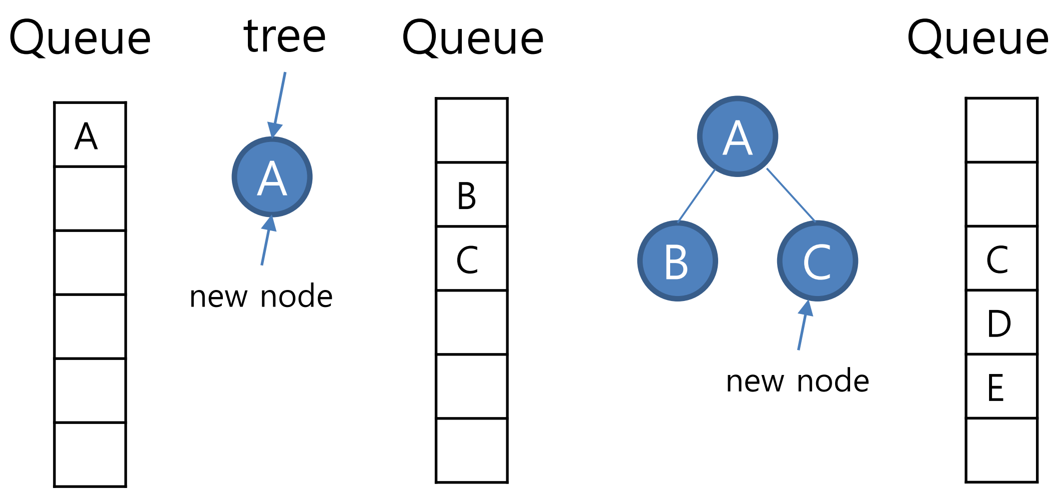
Make a complete binary tree using Queue

insert
1) *If* the tree is empty, initialize the root with *new node*.
2) *Else* {
    get the front node of the queue.
    if the left child of this front node doesn't exist,
        set the left child as the new node.
    else if the right child of this front node doesn't exist,
        set the right child as the new node.
    If the front node has both the left child and right child,
        Dequeue() it.
    }
3) **Enqueue()** the *new node*.



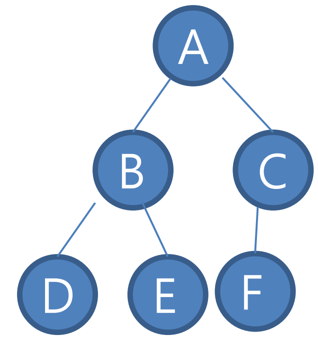
input : ABCDEFGHI





5.3.1 Inorder Traversal

```
void inorder(treePointer ptr)
{/* inorder tree traversal */
    if (ptr) {
        inorder(ptr→leftChild);
        printf("%d",ptr→data);
        inorder(ptr→rightChild);
    }
}
```



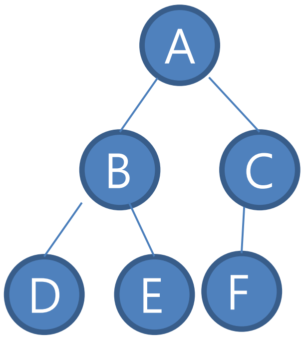
Program 5.1: Inorder traversal of a binary tree

1. Return if the tree is null
2. Inorder traversal of the left subtree
3. Print the value
4. Inorder traversal of the right subtree

5.3.2 Preorder Traversal

```
void preorder(treePointer ptr)
{/* preorder tree traversal */
   if (ptr) {
      printf("%d",ptr→data);
      preorder(ptr→leftChild);
      preorder(ptr→rightChild);
   }
}
```

Program 5.2: Preorder traversal of a binary tree



1. Return if the tree is null
2. Print the value
3. Preorder traversal of the left subtree
4. Preorder traversal of the right subtree

5.3.3 Postorder Traversal

```
void postorder(treePointer ptr)
{/* postorder tree traversal */
   if (ptr) {
      postorder(ptr→leftChild);
      postorder(ptr→rightChild);
      printf("%d",ptr→data);
   }
}
```

Program 5.3: Postorder traversal of a binary tree

1. Return if the tree is null
2. Postorder traversal of the left subtree
3. Postorder traversal of the right subtree
4. Print the value