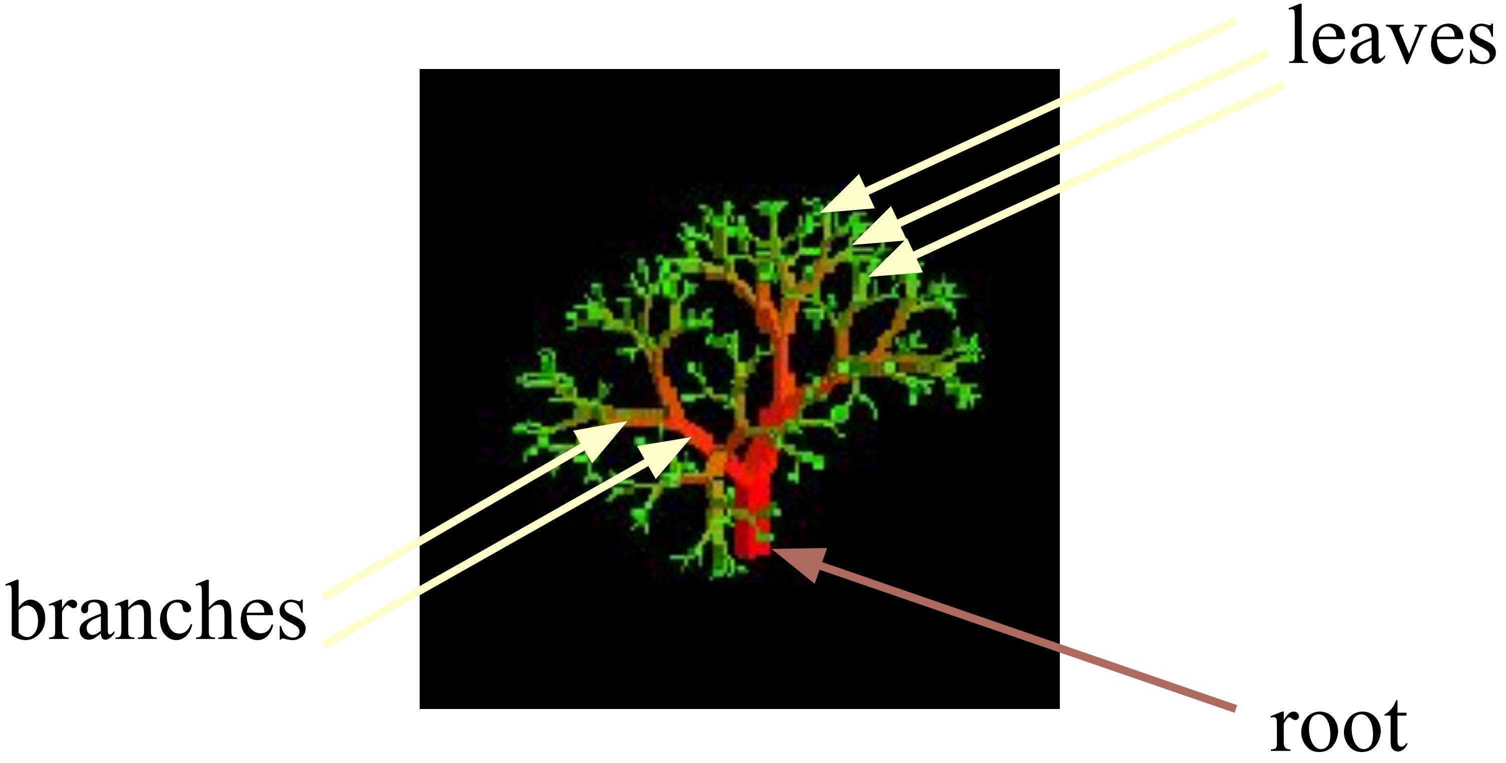
Unit IV

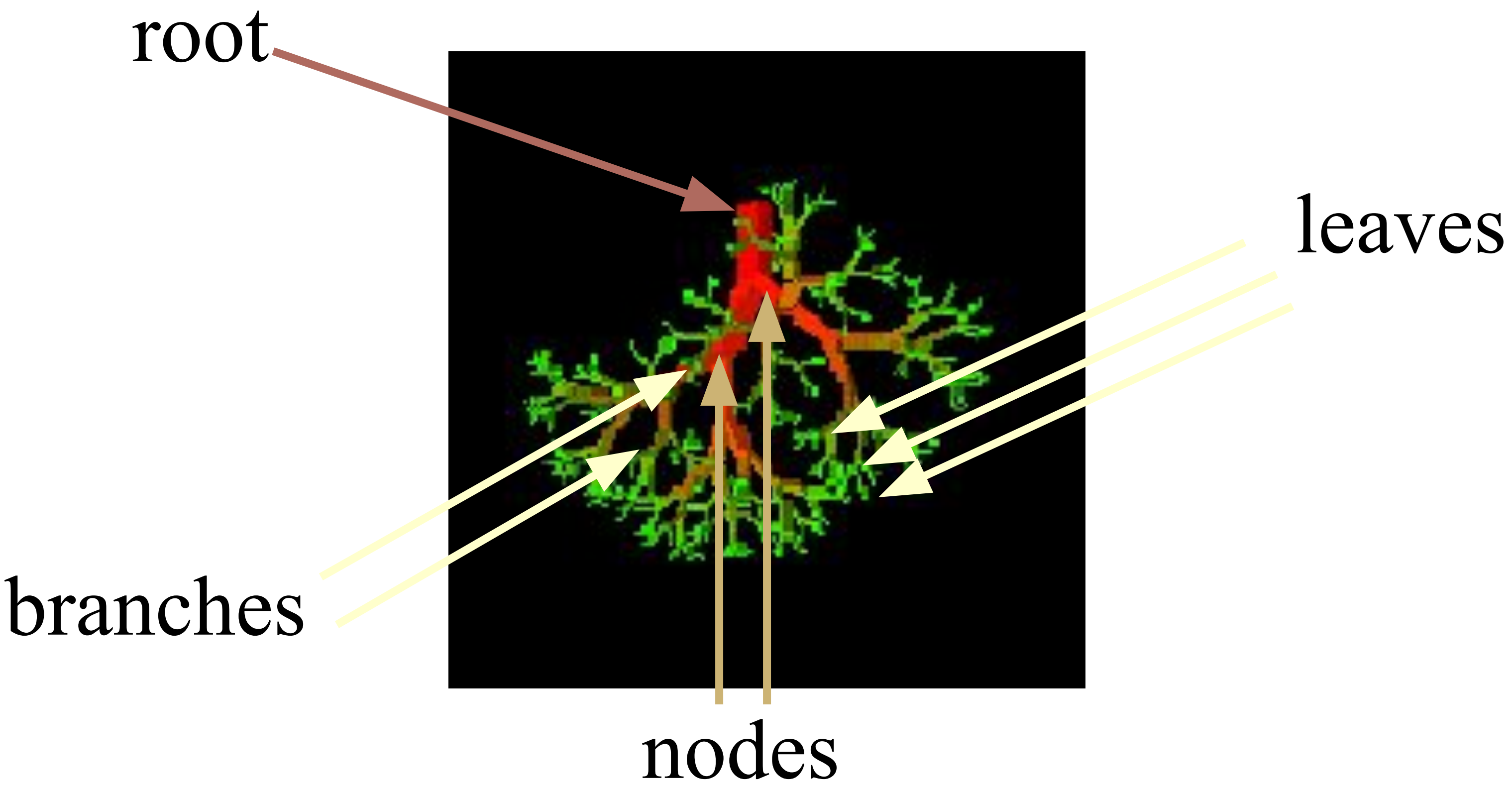
Non-Linear Data Structure

(Tree and Graph)

Nature View of a Tree

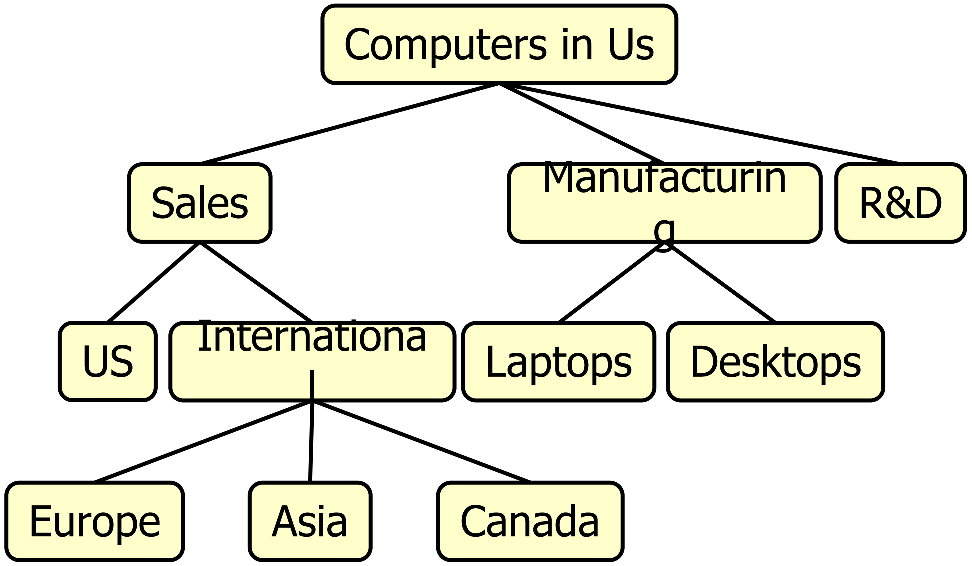


Computer Scientist's View



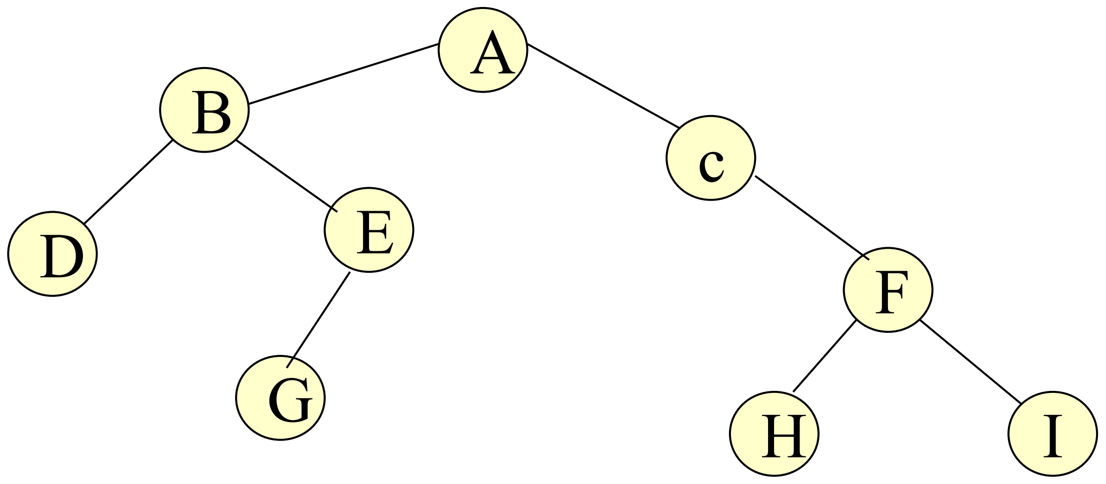
What is a Tree

- A tree is a finite nonempty set of elements.
- It is an abstract model of a hierarchical structure.
- It consists of nodes with a parent-child relation.
- Applications:
  - Organization charts
  - File systems



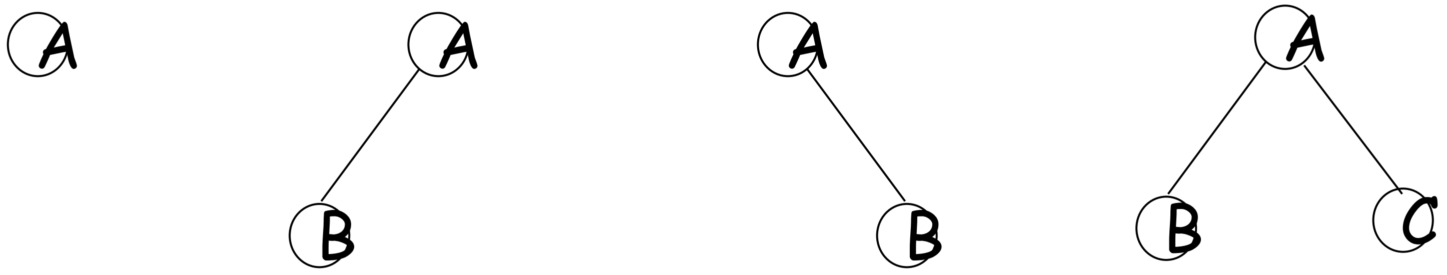
Tree

Each element of a tree is called a node of the tree.



Binary Trees

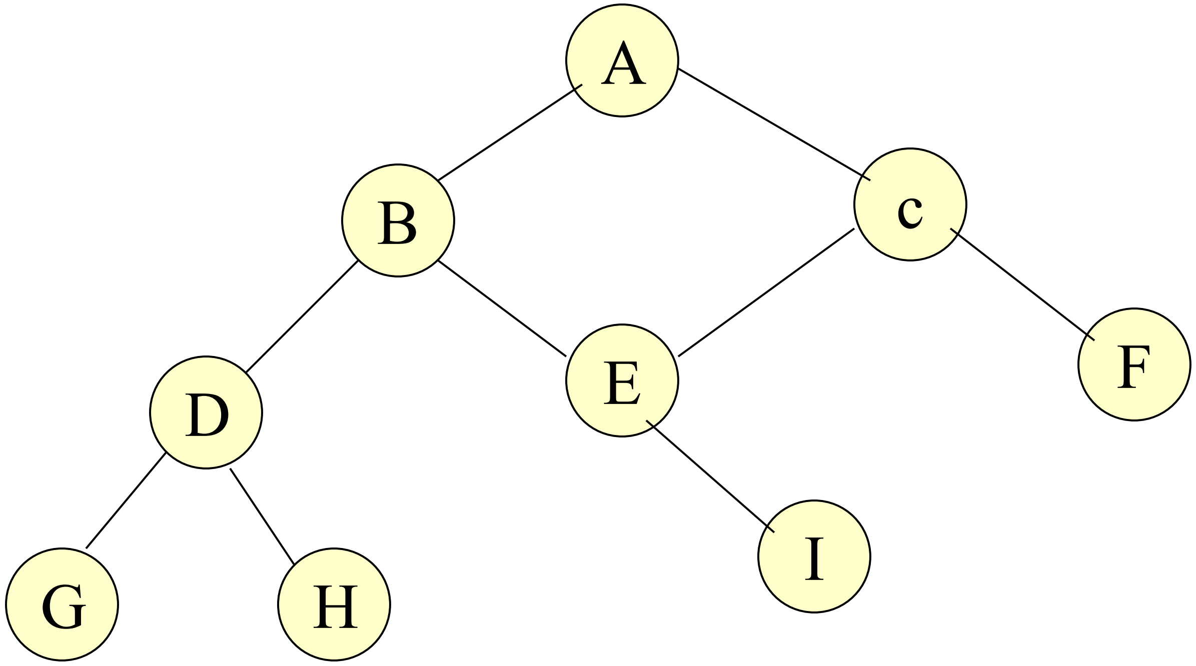
- Binary trees are characterized by the fact that any node can have at most two branches
- **Definition** (recursive):
  - A *binary tree* is a finite set of nodes that is either empty or consists of a root and two disjoint binary trees called the left subtree and the right subtree
- Thus the left subtree and the right subtree are distinguished



- Any tree can be transformed into binary tree
  - by left child-right sibling representation

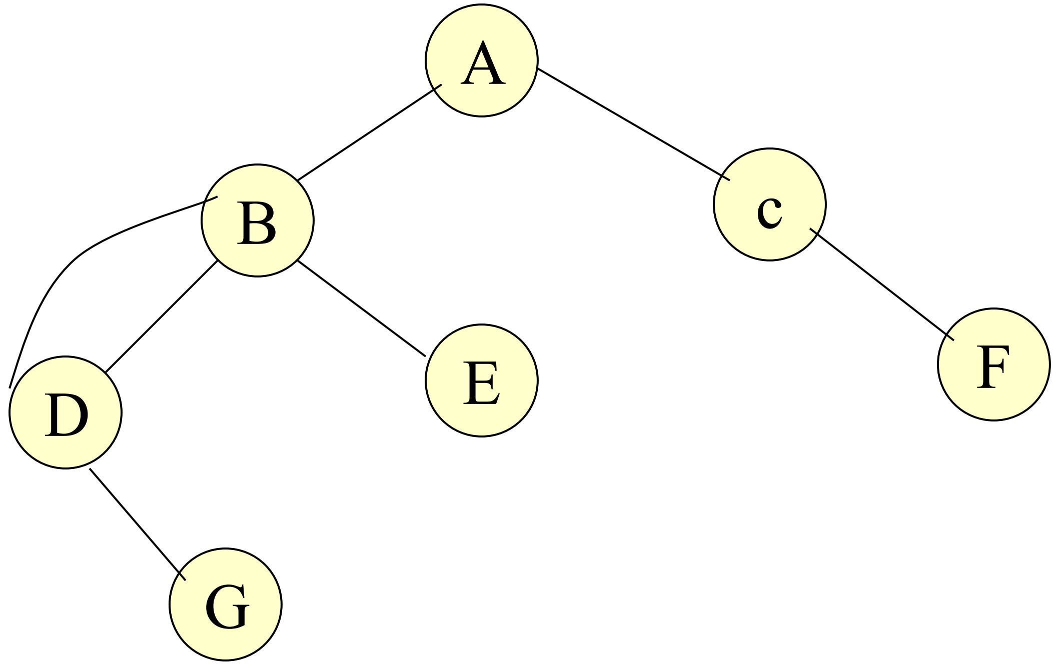
Binary Tree

Structures that are not binary trees



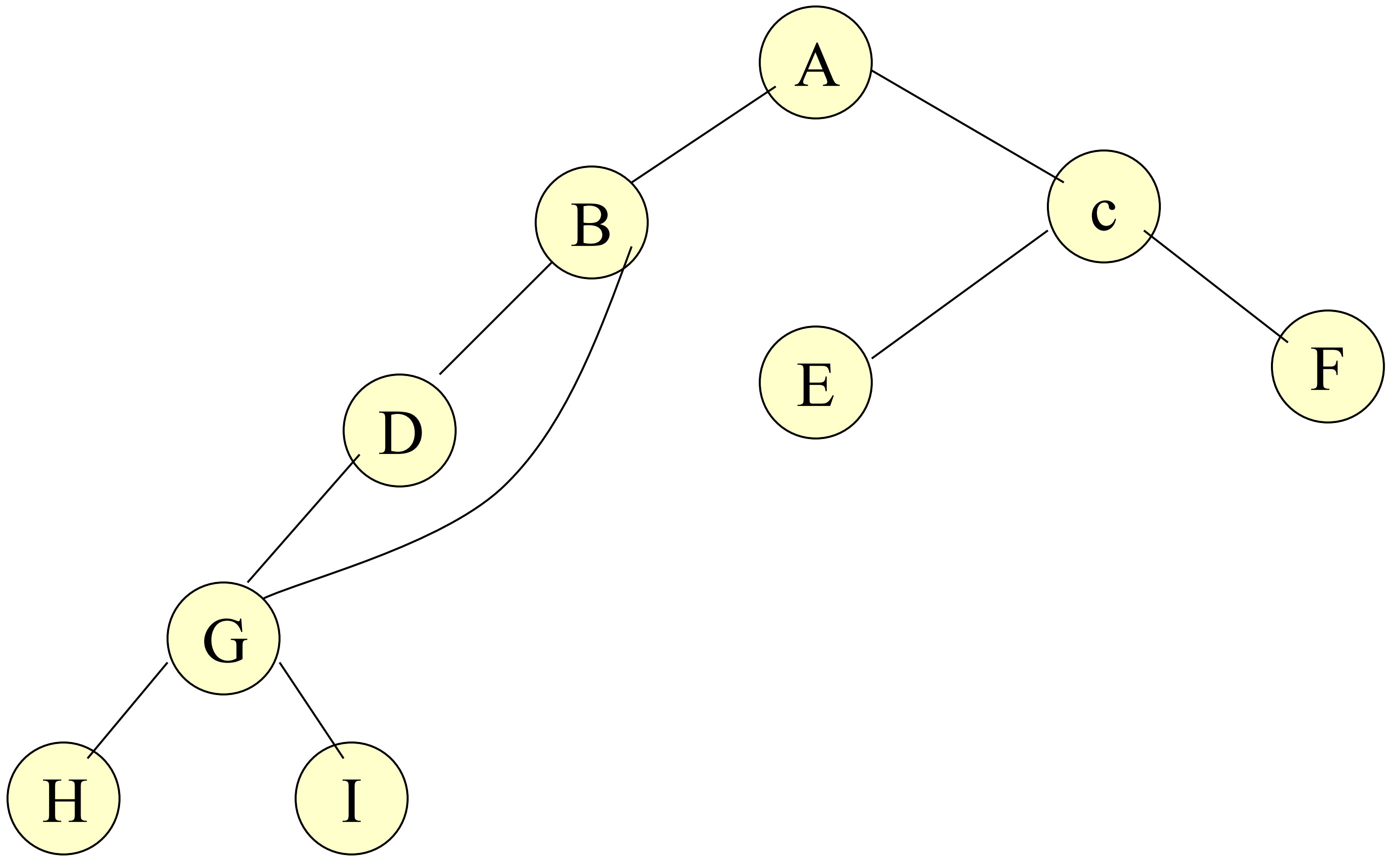
Binary Tree

Structures that are not binary trees



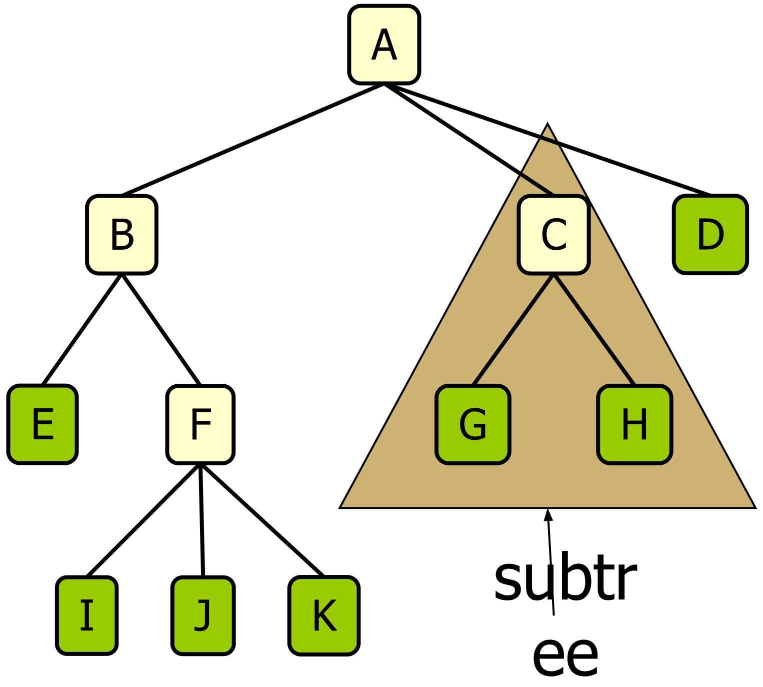
Binary Tree

Structures that are not binary trees



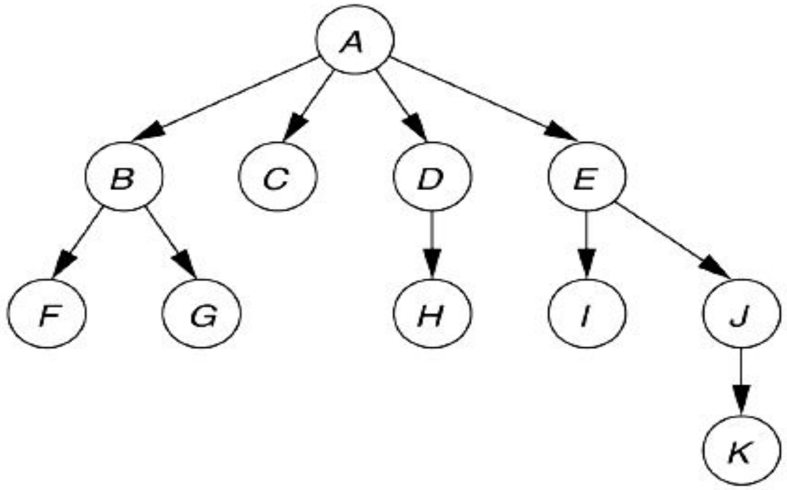
Tree Terminology

- **Root**: node without parent (A)
- **Parent node:** Immediate predecessor of any node (Node A is parent node of B,C,D)
- **Child node:** Immediate successor of any node
- **Siblings**: Child nodes of the same parent
- **Internal node**: node with at least one child (A, B, C, F)
- **Leaf Node/External node**: node without children (E, I, J, K, G, H, D)
- **Ancestors** of a node: parent, grandparent, grand-grandparent, etc.
- **Descendant of a node** : child, grandchild, grand-grandchild, etc.
- **Depth/Level of a node** : number of ancestors (Depth/Level of a node E is 2)
- **Height/Depth of a tree:** maximum depth of any node (3)
- **Degree of a node**: the number of node connected to particular node (Degree of node F is 4)
- **Degree of a tree**: the maximum number of its node.
- **Subtree**: tree consisting of a node and its descendants



Tree Terminology

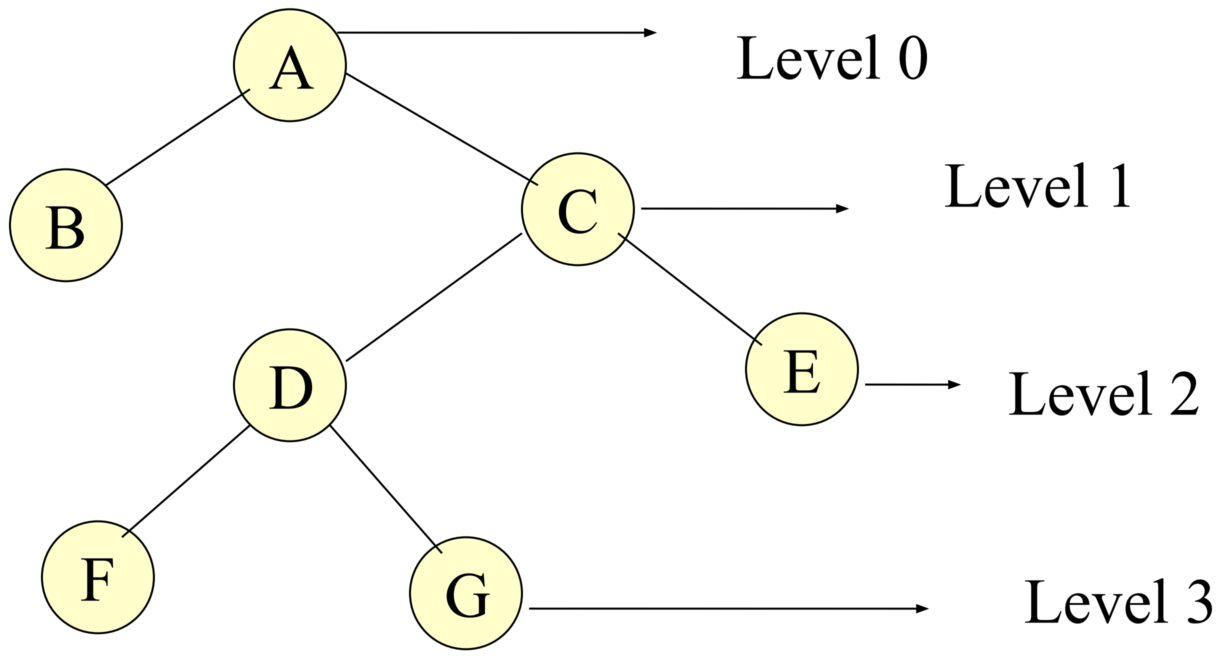
- **Height of node**: Longest path from the node to leaf.
- **Depth of node**: The length of the path from the root to the node.



Node	Height	Depth
A	3	0
B	1	1
C	0	1
D	1	1
E	2	1
F	0	2
G	0	2
H	0	2
I	0	2
J	1	2
K	0	3

Level of a binary Tree

Level of binary tree:
The root of the tree has level 0. And the level of any other node is one more than the level of its father.

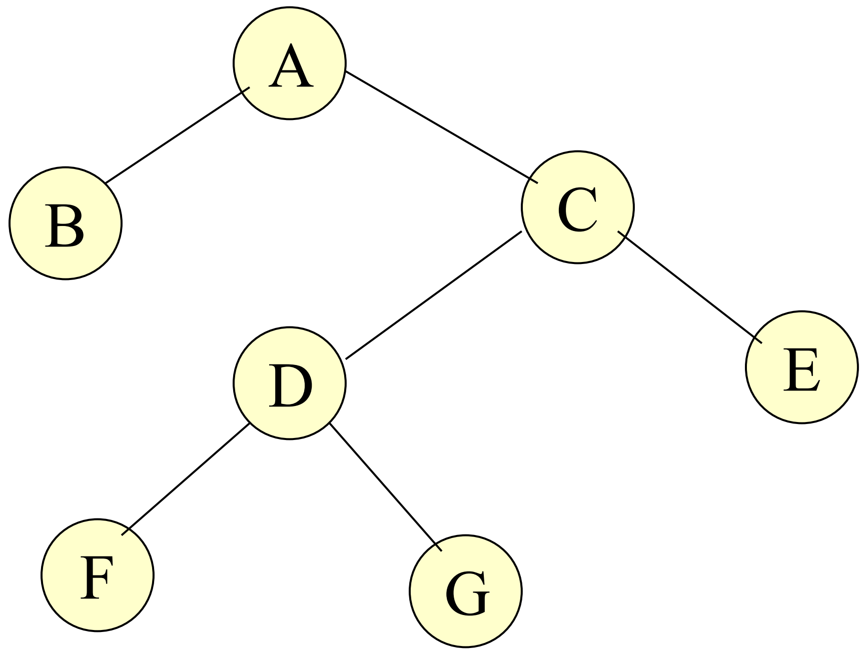


Depth is 3.

Binary Tree

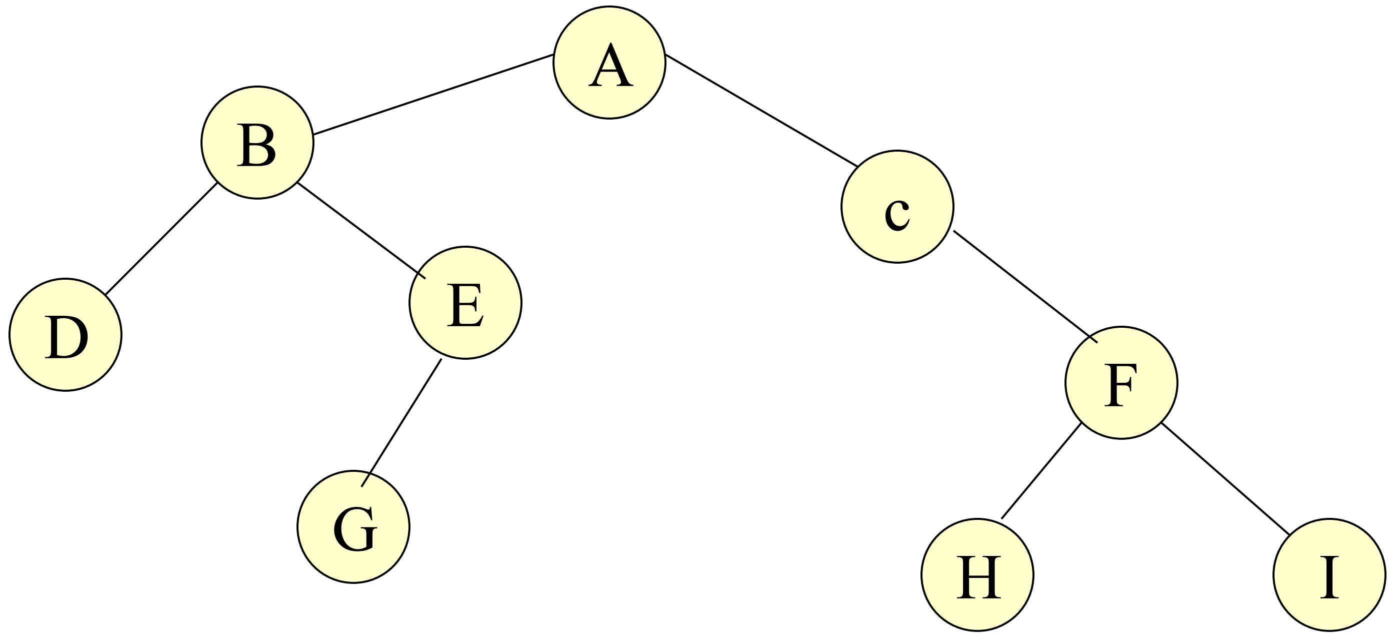
Strictly binary trees:

- If every non-leaf node in a binary tree has nonempty left and right subtrees, the tree is called a strictly binary tree.
- A strictly binary tree with n leaves always contains 2n -1 nodes.



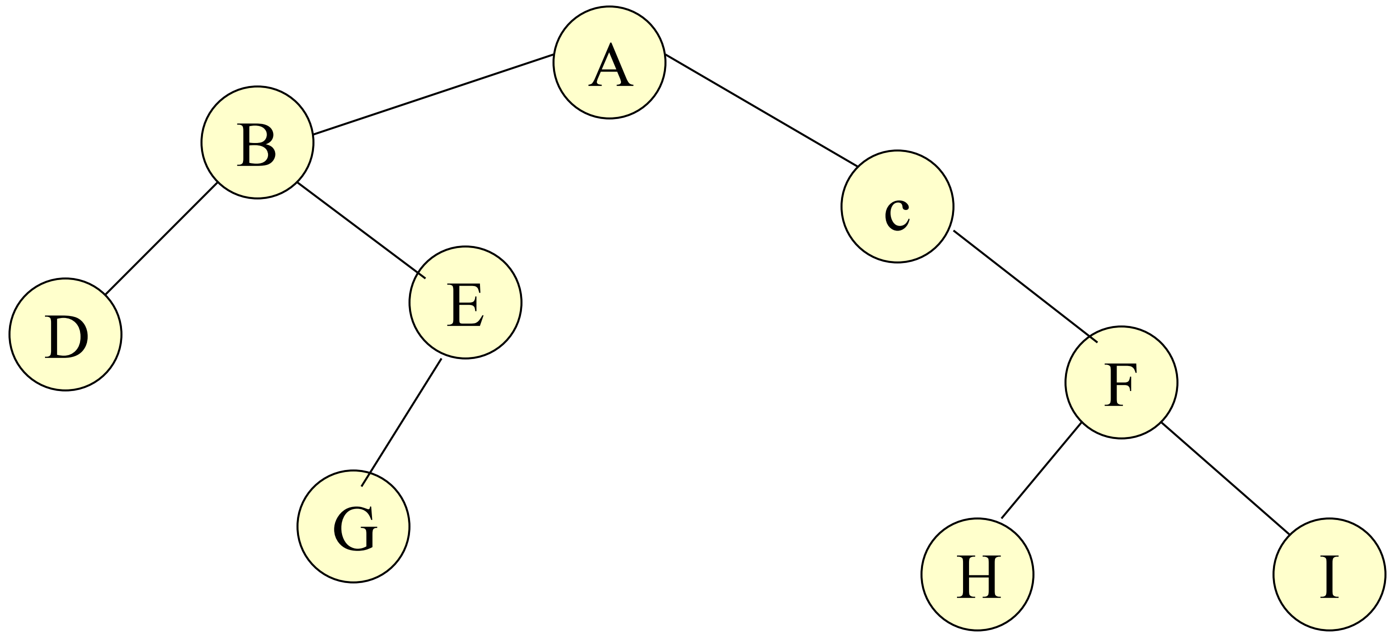
Binary tree

Is it a strictly binary tree?



Binary tree

Structure that is not a strictly binary tree, because nodes C and E have one son each.



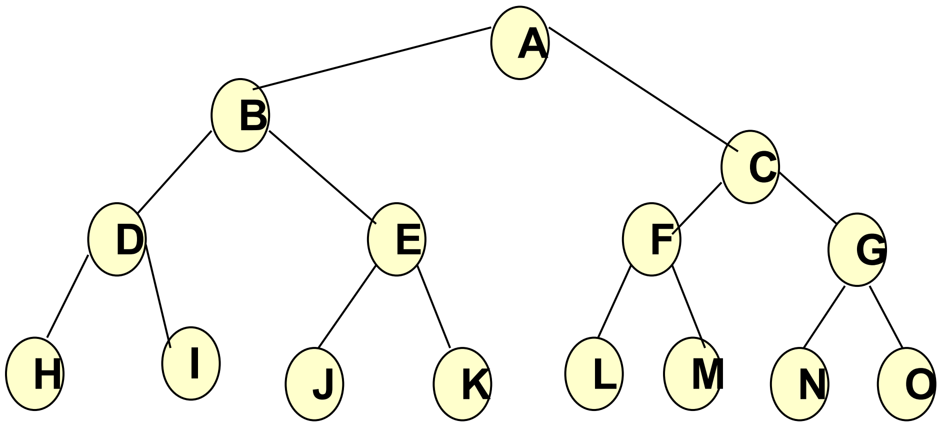
A complete binary tree

- Complete binary tree of depth d is the strictly binary tree all of whose leaves are at level d.
- A complete binary tree of depth d is the binary tree of depth d that contains exactly 2^k nodes at each level k between 0 and d.

The total number of nodes = the sum of the number of nodes at each level between 0 and d.

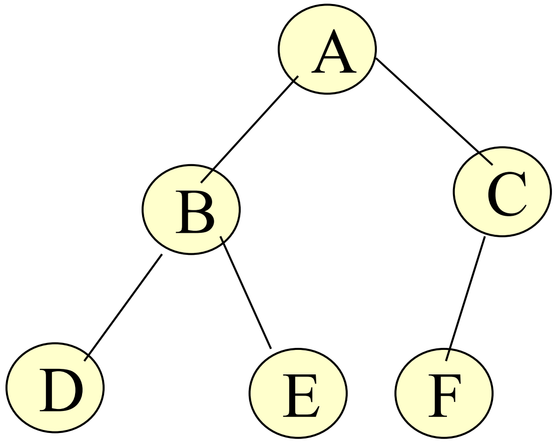
$= 2^{d+1} - 1$

A complete tree is a tree in which every level is completely filled

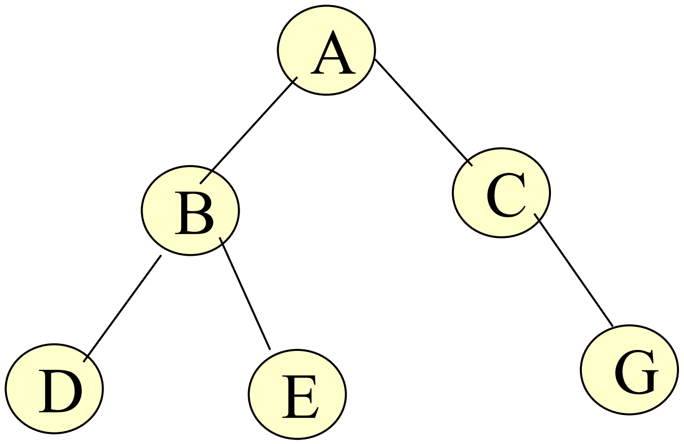


Almost complete binary tree

- **Almost complete tree** is a tree in which if last level is not completely filled.
- An almost complete binary tree is a special kind of binary tree where insertion takes place level by level and from left to right order at each level and the last level is not filled fully always.

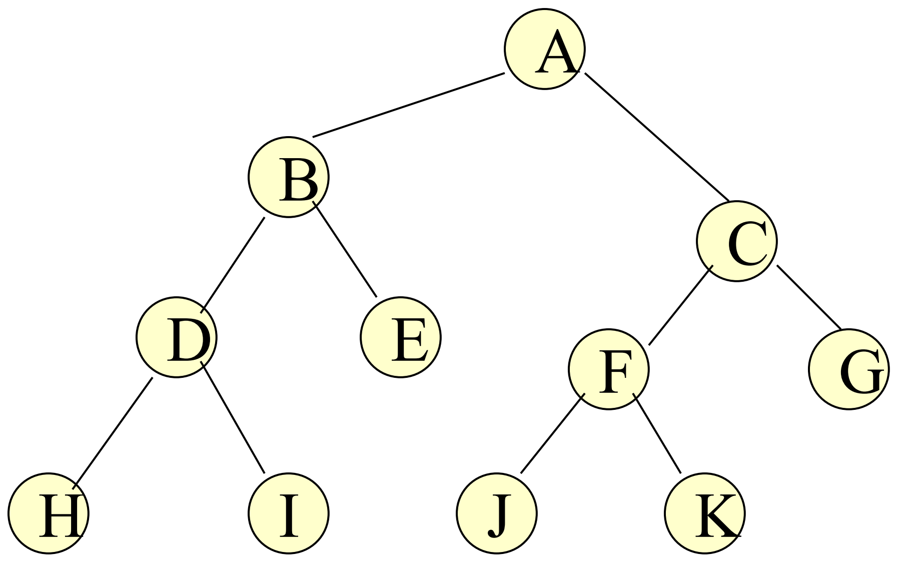


Almost complete tree



Not Almost complete tree

Almost complete binary tree



Not Almost complete tree

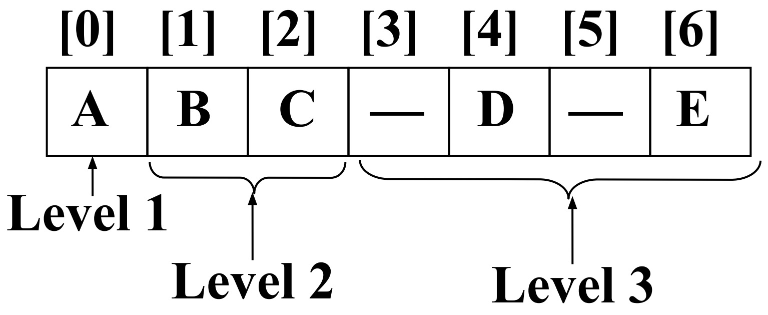
The strictly binary tree is not almost complete, since A has a right descendant at level 3 (J) but also has a left descendant that is a leaf at level 2 (E)

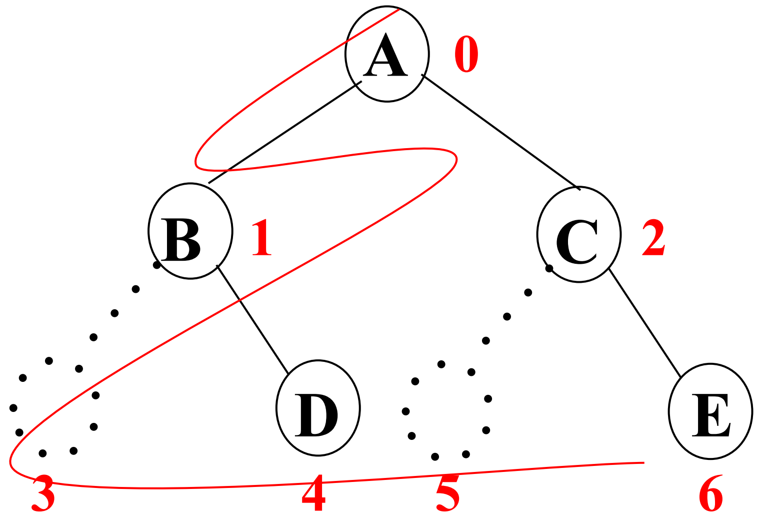
Binary Tree representations

- Binary tree representations using array

  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

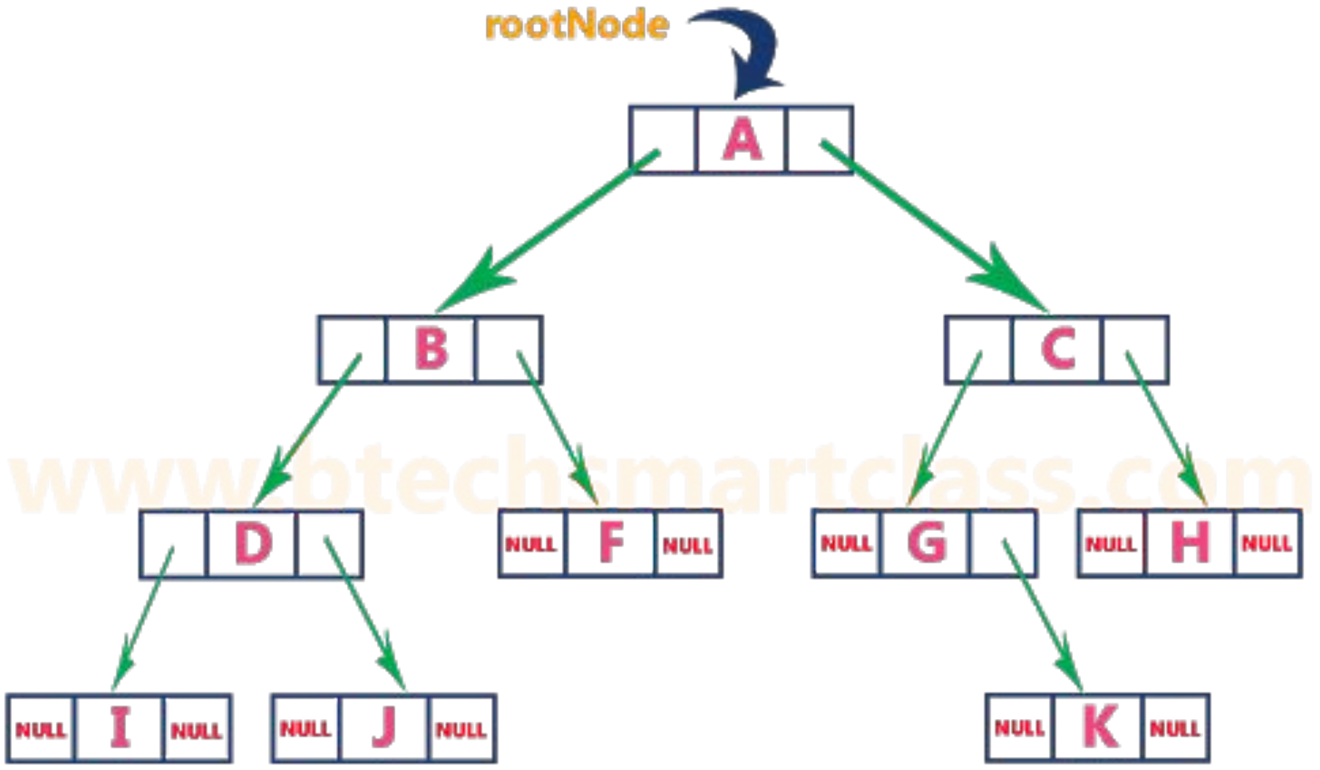
  - *node n* is at *i*.
    If i = 0, i is at the root and has no parent.
  - *LeftChild(i)* is at $2i+1$.
  - *RightChild(i)* is at $2i+2$.





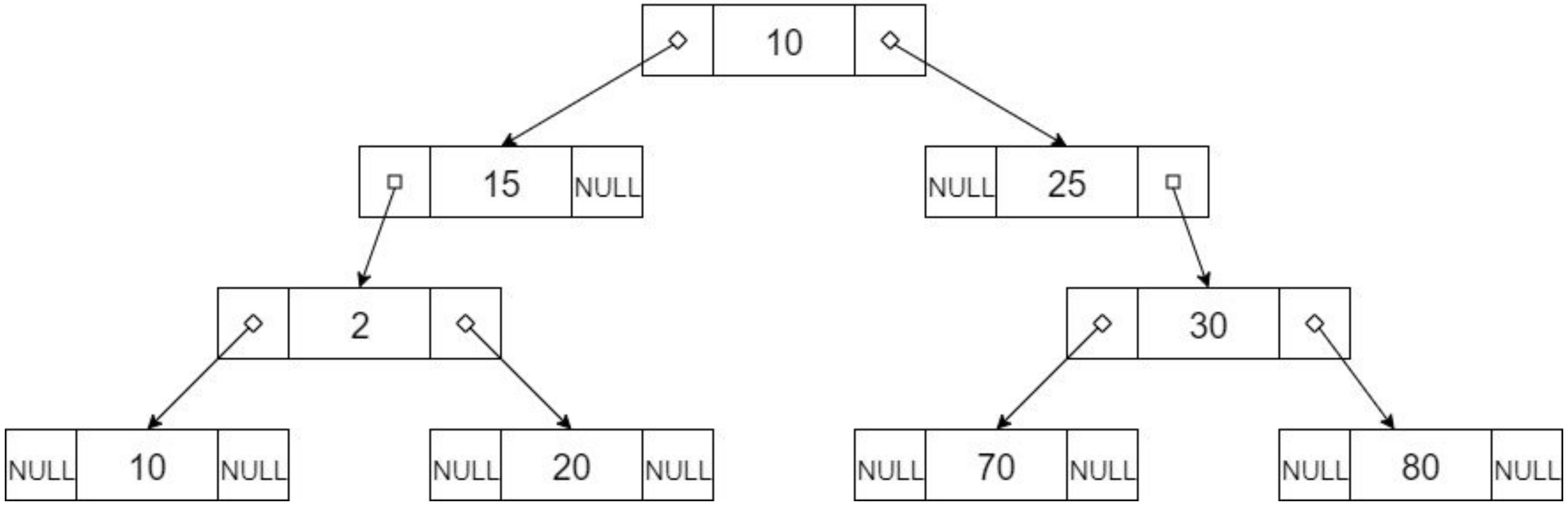
Binary Tree representations

- Binary tree representations using link



Binary Tree representations

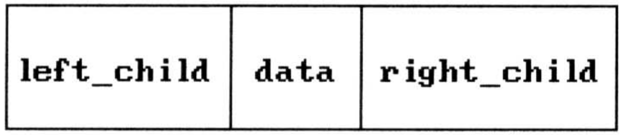
- Binary tree representations using linked list

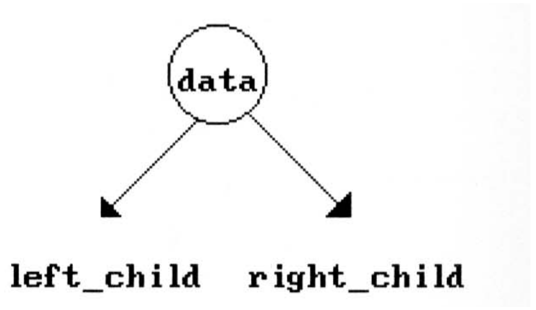


Binary Tree representations

- Binary tree Node structure

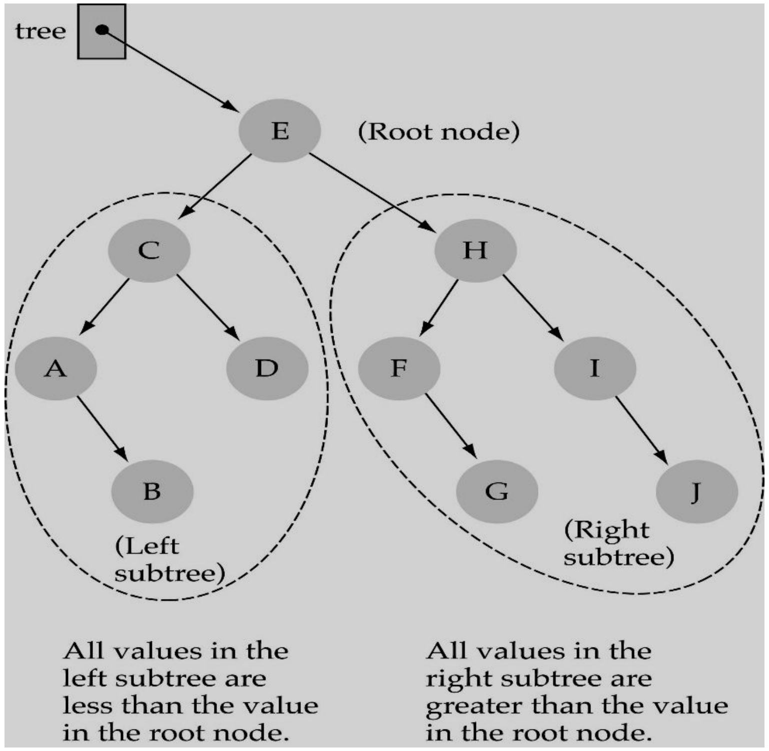
```
typedef struct node
{
    int data;
    struct node *left_child;
    struct node *right_child;
}node;
```





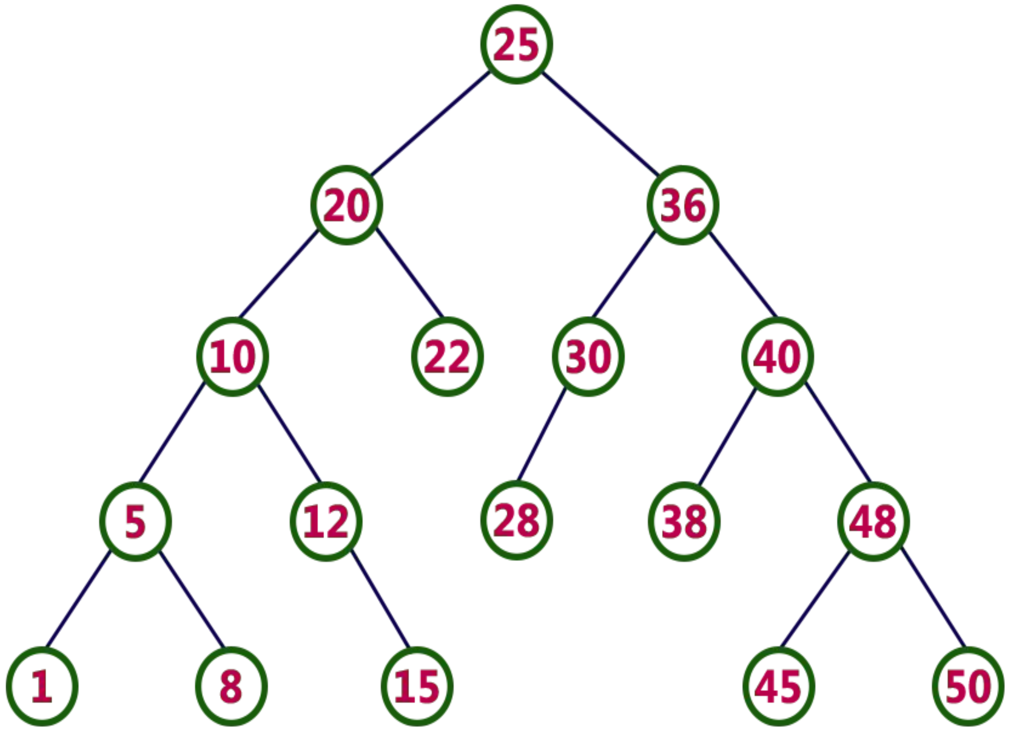
Binary Search Trees (BSTs)

- **Binary Search Tree Property**:
  The value stored at a node is *greater* than the value stored at its left child and *less* than the value stored at its right child


tree
E
(Root node)
C
H
A
D
F
I
B
G
J
(Left subtree)
(Right subtree)
All values in the left subtree are less than the value in the root node.
All values in the right subtree are greater than the value in the root node.

Binary Search Trees (BSTs)

The data in each node is greater than the data in its left subtree, and less than or equal to the data in its right subtrees.

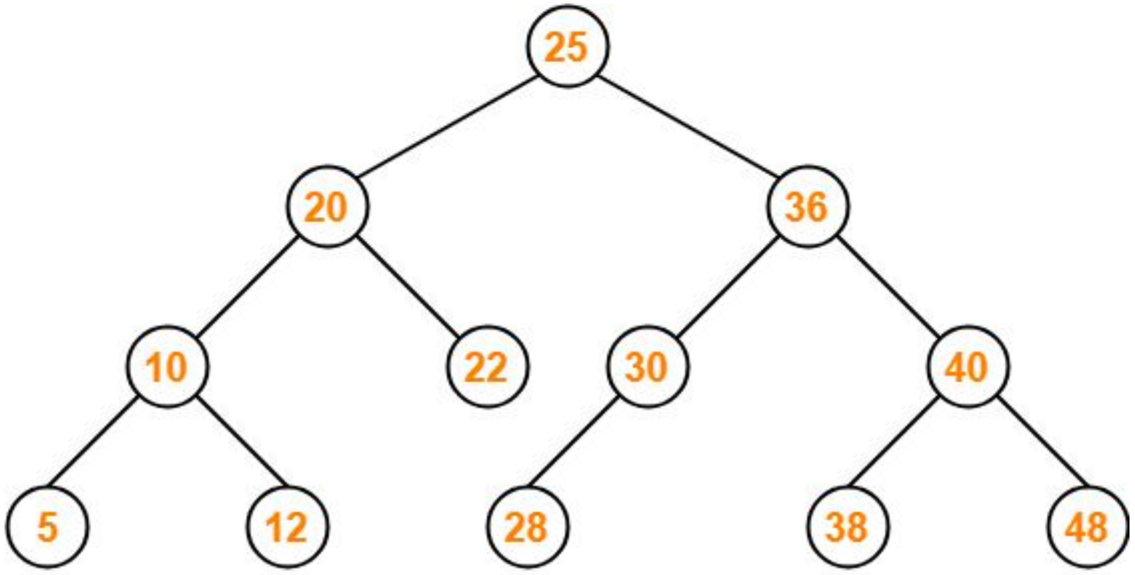


Binary Search Trees (BSTs)

Binary Search Trees (BSTs)

- 20 10 22 36 40 12 30 38 28 48 5

  5 10 12 20 22 25 28 30 36 38 40 48



Binary Search Tree

Function to Create Binary Search Trees (BSTs)

```
bst *create(bst * root)
{
char ch;
bst *p,*temp;
do {
p=(bst*)malloc(sizeof(bst));
printf("\n enter data:");
_flushall();
scanf("%d",&p->data);
p->left=NULL;
p->right=NULL;
if(root==NULL)
{
root=p;
}
```

Function to Create Binary Search Trees (BSTs)

```
else
{
temp=root;
while(1) {
if(p->data<temp->data)
{
if(temp->left==NULL)
{
temp->left=p;
break;
}
else
temp=temp->left;
}
```

Function to Create Binary Search Trees (BSTs)

```
else {
if(temp->right==NULL)
{
temp->right=p;
break;
}
else
temp=temp->right;
} } }
printf(“do you want to add new node?”);
scanf(“%c” &ch);
} while(ch==‘y’ || ch==‘Y’);
return root;
}
```

Function to insert node in Binary Search Trees (BSTs)

```
void create(bst * root)
{
char ch;
bst *p,*temp;
p=(bst*)malloc(sizeof(bst));
printf("\n enter data:");
_flushall();
scanf("%d",&p->data);
p->left=NULL;
p->right=NULL;
else {
temp=root;
while(1) {
if(p->data<temp->data)
{
if(temp->left==NULL)
{
```

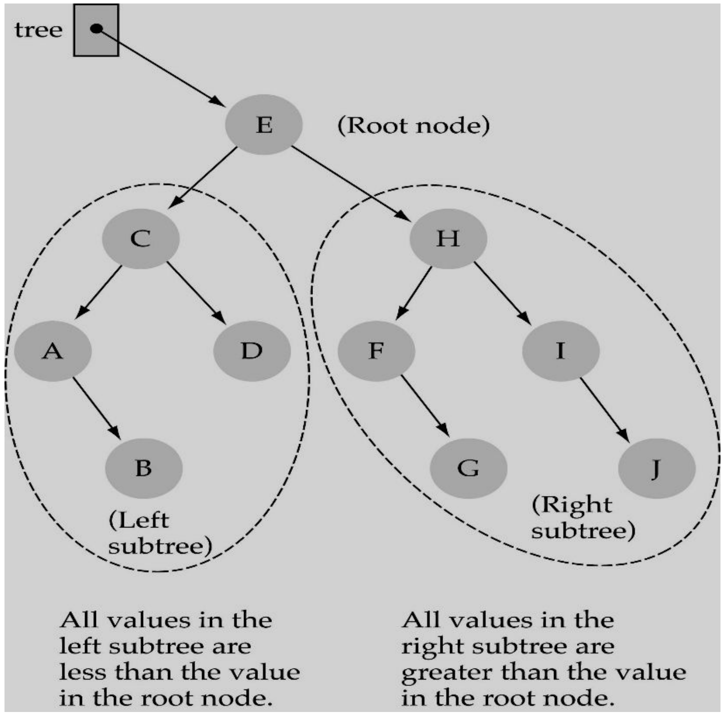
Function to insert node in Binary Search Trees (BSTs)

```
temp->left=p;
break;
}
else
temp=temp->left;
}
else {
if(temp->right==NULL)
{
temp->right=p;
break;
}
else
temp=temp->right;
} } }
}
```

Function to Search an element in Binary Search Trees (BSTs)

```
void search(bst*root,int key)
{
bst *temp;
temp=root;
while(temp!=NULL)
{
If(key==temp->data)
printf("\n Data %d present at %u ",temp->data,temp);
else if(key<temp->data)
temp=temp->left;
else
temp=temp->right;
}
If(temp==NULL)
printf("\n Data Not found!");
}
```

How to search a binary search tree?


tree
E
(Root node)
C
H
A
D
F
I
B
G
J
(Left subtree)
(Right subtree)
All values in the left subtree are less than the value in the root node.
All values in the right subtree are greater than the value in the root node.

(1) Start at the root

(2) Compare the value of the item you are searching for with the value stored at the root

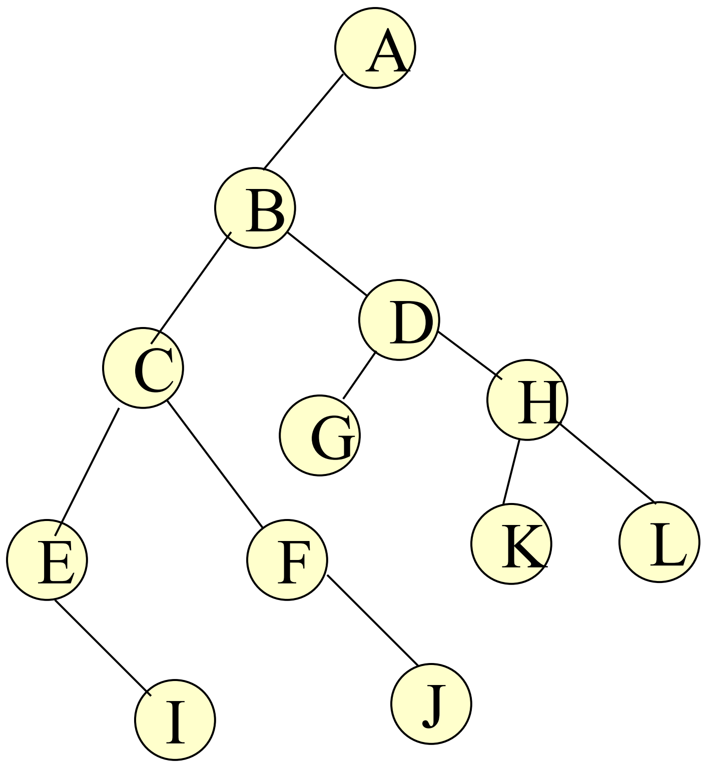
(3) If the values are equal, then *item found*; otherwise, if it is a leaf node, then *not found*

(4) If it is less than the value stored at the root, then search the left subtree

(5) If it is greater than the value stored at the root, then search the right subtree

(6) Repeat steps 2-6 for the root of the subtree chosen in the previous step 4 or 5

Traversing a binary tree



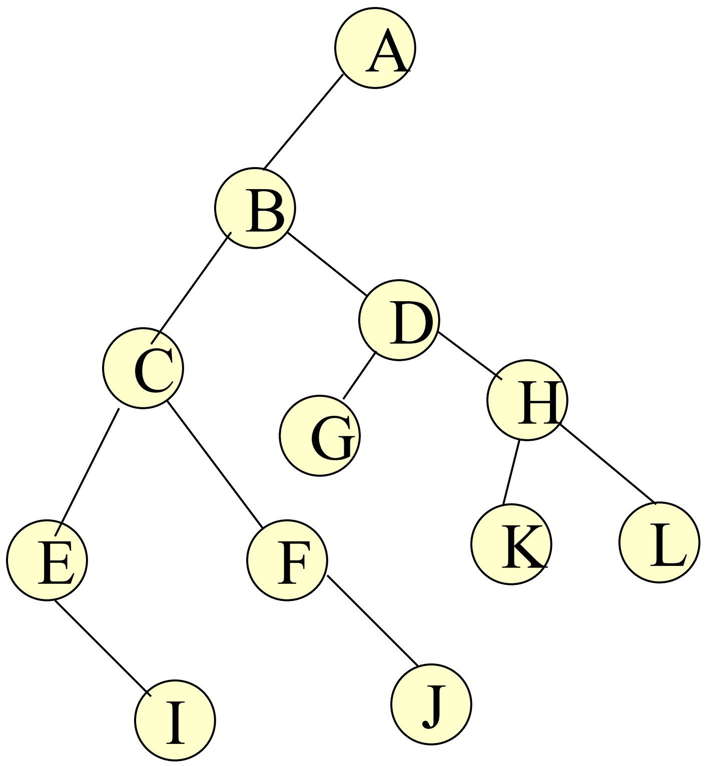
<u>Inorder</u>

1. Traverse the left subtree in inorder
2. Visit the root
3. Traverse the right subtree in inorder

inorder: EICFJBGDKHLA

Left Root Right

Traversing a binary tree

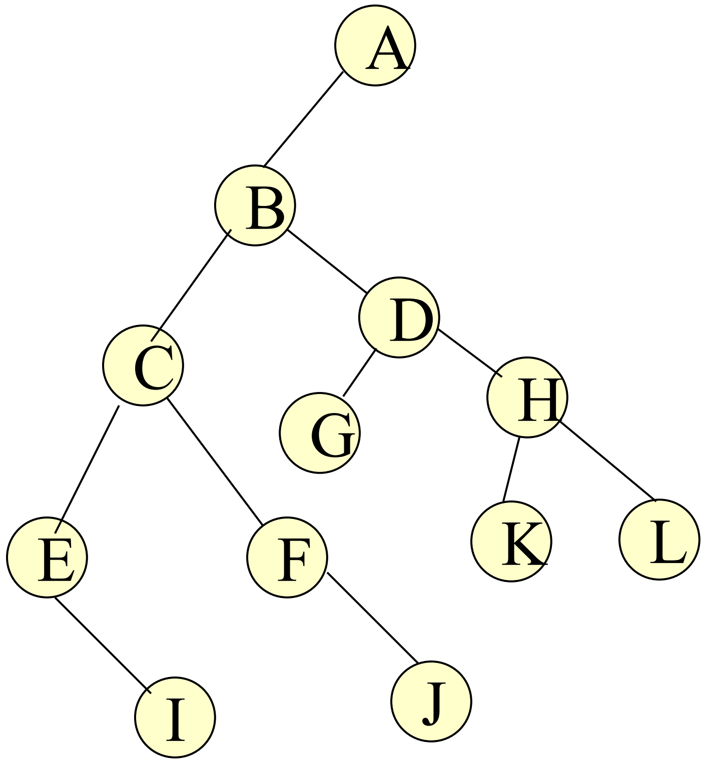


<u>Preorder</u>

1. Visit the root
2. Traverse the left subtree in preorder
3. Traverse the right subtree in preorder

preorder: ABCEIFJDGHKL

Traversing a binary tree

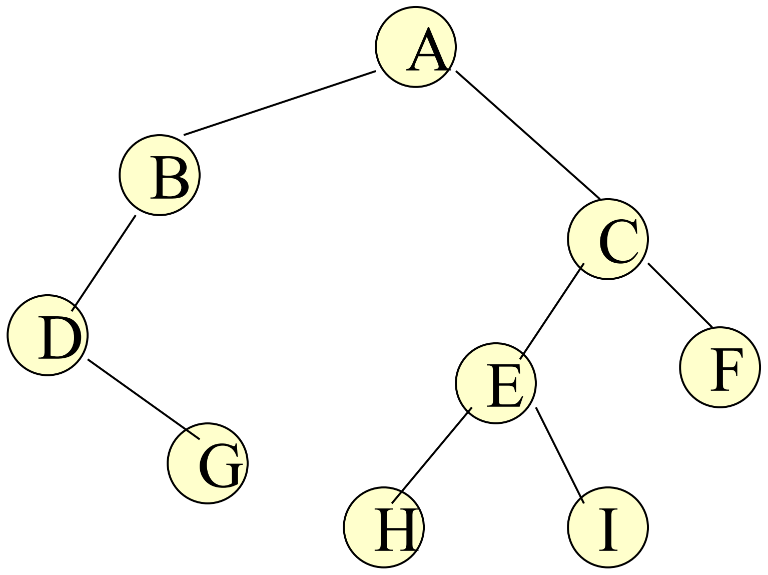


<u>Postorder</u>

1. Traverse the left subtree in postorder
2. Traverse the right subtree in postorder
3. Visit the root

postorder: IEJFCGKLHDBA

Traversing a binary tree



Preorder
1. Visit the root
2. Traverse the left subtree in preorder
3. Traverse the right subtree in preorder

preorder: ABDGCEHIF

Inorder
1. Traverse the left subtree in inorder
2. Visit the root
3. Traverse the right subtree in inorder

inorder: DGBAHEICF

Postorder
1. Traverse the left subtree in postorder
2. Traverse the right subtree in postorder
3. Visit the root

postorder: GDBHIEFCA

Recursive Function for Inorder traversing

```
void Inorder(bst*root)
{
if(root!=NULL)
{
Inorder(root->left);
printf("%d",root->data);
Inorder(root->right);
}
}
```

Recursive Function for Preorder traversing

```
void preorder(bst*root)
{
if(root!=NULL)
{
printf("%d",root->data);
preorder(root->left);
preorder(root->right);
}
}
```

Recursive Function for Postorder traversing

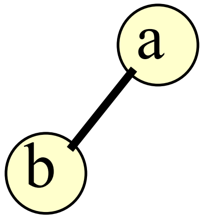
```
void postorder(bst*root)
{
if(root!=NULL)
{
postorder(root->left);
postorder(root->right);
printf("%d",root->data);
}
}
```

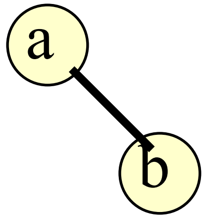
Binary Tree Reconstruction

- Suppose that the elements in a binary tree are distinct.
- Can you construct the binary tree from which a given traversal sequence came?
- When a traversal sequence has more than one element, the binary tree is not uniquely defined.
- Therefore, the tree from which the sequence was obtained cannot be reconstructed uniquely.

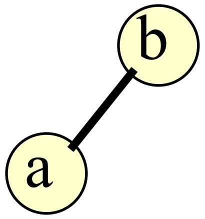
Some Examples

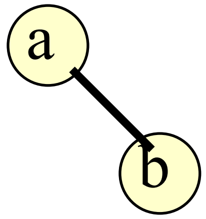
preorder = ab



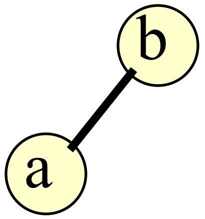


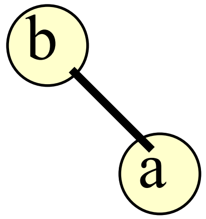
inorder = ab





postorder = ab

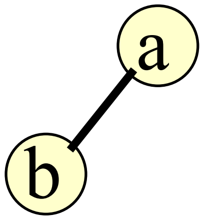




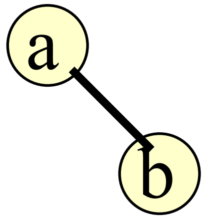
Binary Tree Construction

Preorder And Postorder

preorder = ab



postorder = ba

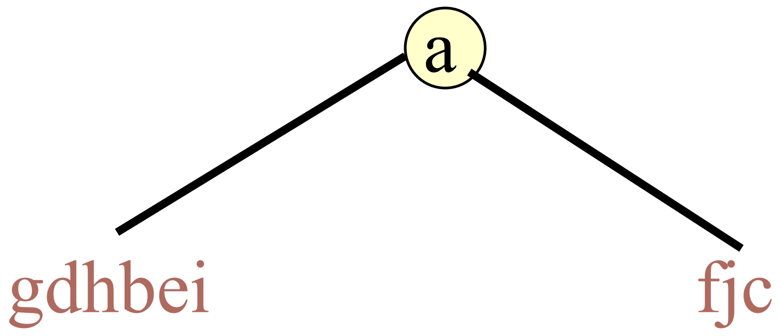


- Preorder and postorder do not uniquely define a binary tree.
- Can you construct the binary tree, given two traversal sequences?
- Depends on which two sequences are given.

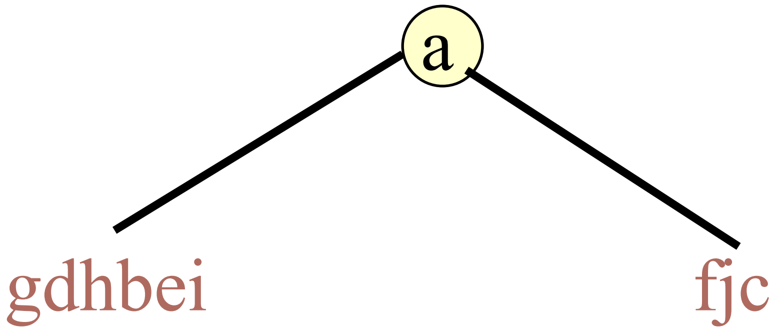
Inorder And Preorder

inorder = g d h b e i a f j c
preorder = a b d g h e i c f j

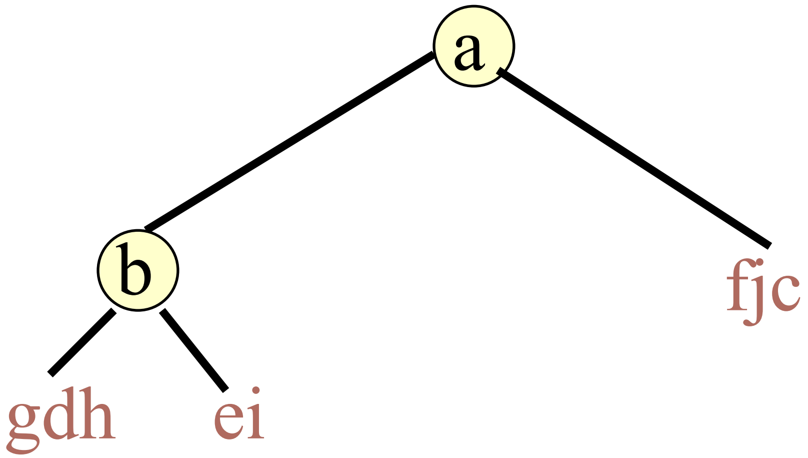
- Scan the preorder left to right using the inorder to separate left and right subtrees.
- a is the root of the tree; gdhbei are in the left subtree; fjc are in the right subtree.



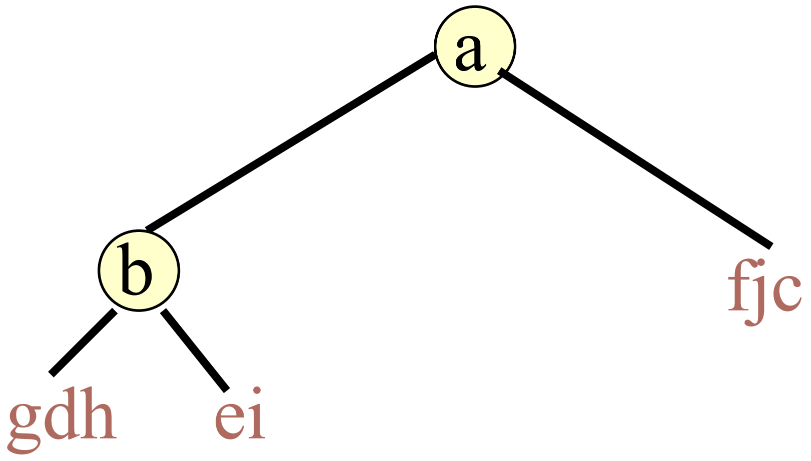
Inorder And Preorder



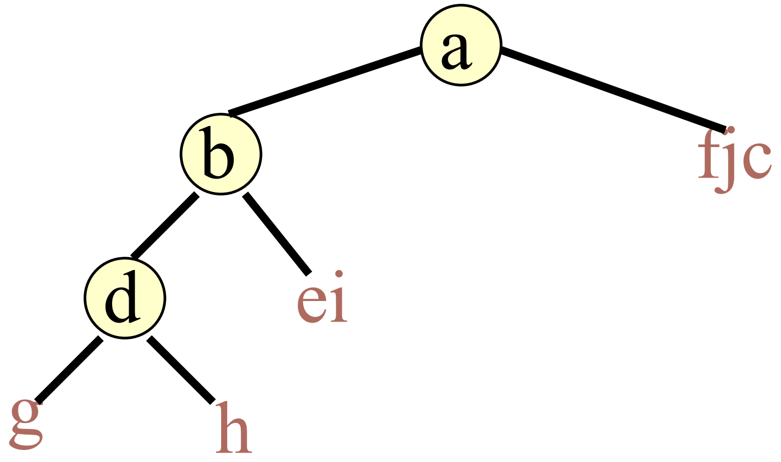
- preorder = b d g h e i c f j
- b is the next root; gdh are in the left subtree; ei are in the right subtree.



Inorder And Preorder



- preorder = d g h e i c f j
- d is the next root; g is in the left subtree; h is in the right subtree.



Inorder And Postorder

- Scan postorder from right to left using inorder to separate left and right subtrees.

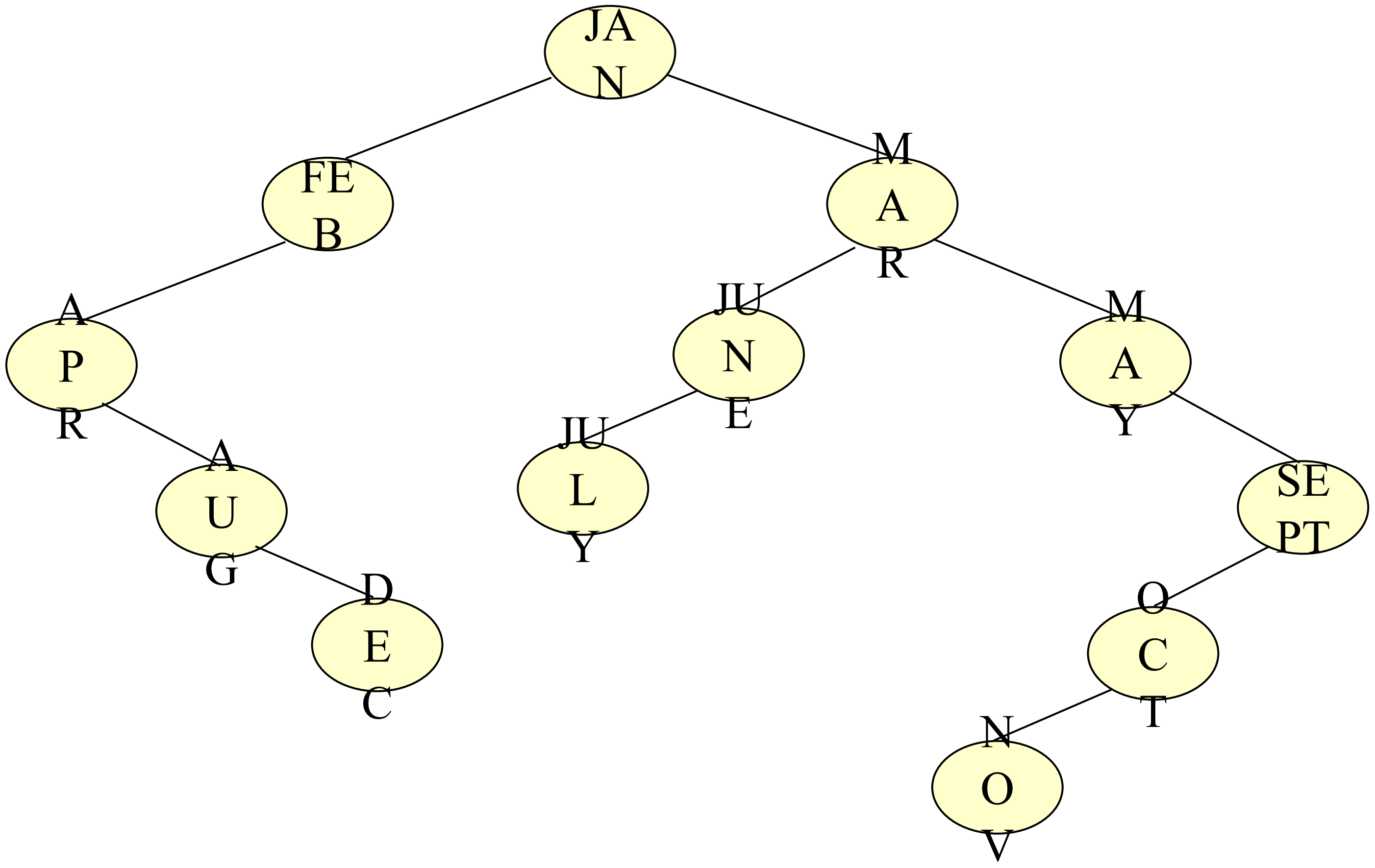
  inorder = g d h b e i a f j c

  postorder = g h d i e b j f c a

- Tree root is a; gdhbei are in left subtree; fjc are in right subtree.

Binary Search Tree for The Months of The Year

Input Sequence: JAN, FEB, MAR, APR, MAY, JUNE, JULY, AUG, SEPT, OCT, NOV, DEC



Applications of Trees

1. Store the data in Hierarchical Data.

2. Trees are used to organize the data and store the data in a logical manner.

3. Spanning trees are also used to find the minimum/shortest path.

4. To count the number of nodes.

5. To count the number of leaf nodes.

6. Minimum spanning tree.

There are much more applications of tree, apart from the above listed.

Counting Number of Nodes

```
int count(node *P)
{
    int c = 1;

    if (P == NULL)
        return 0;
    else
    {
        c = c + count(P->left);
        c = c + count(P->right);
        return c;
     }
}
```

Counting Leaf Nodes

```
int countLeafNode(struct node *root)
{
        if(root == NULL)
        return 0;

// Check for leaf node
        if(root->left == NULL && root->right == NULL)
        return 1;

 // For internal nodes, return the sum of leaf nodes in left and right
sub-tree
    return (countLeafNode(root->left) + countLeafNode(root->right));
}
```