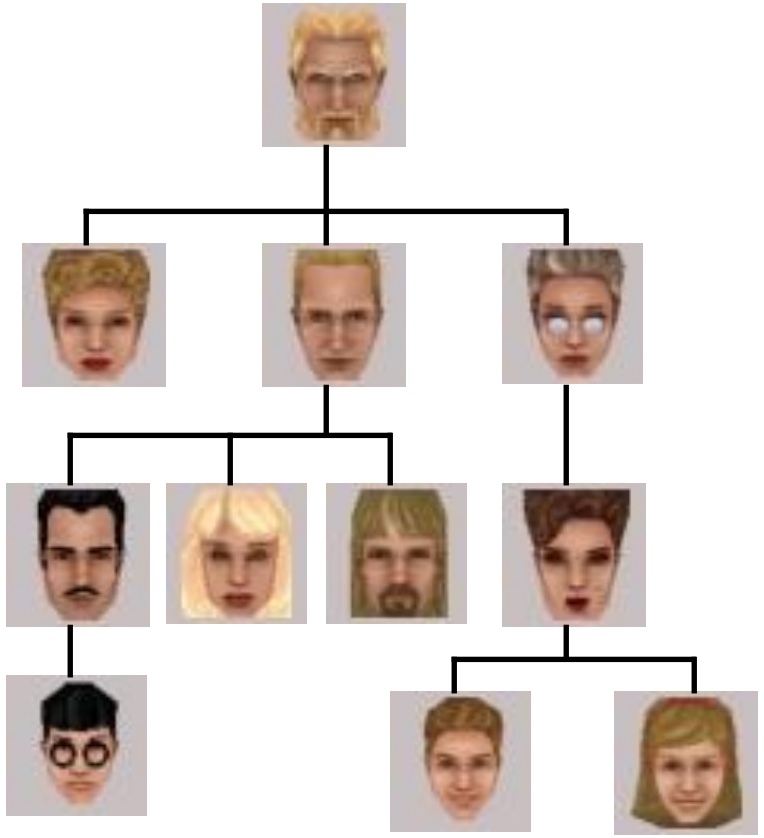
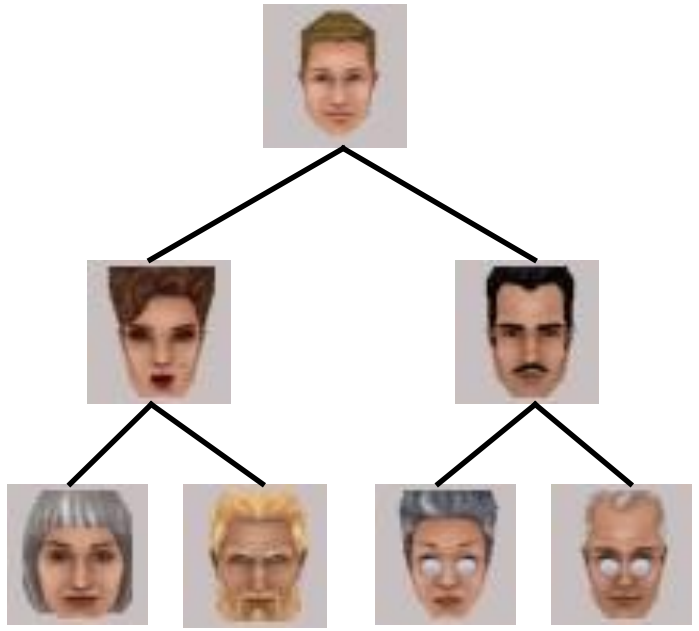
Trees

ICT 4303

Preliminaries



Lineal Tree



Pedigree Tree
(binary tree)

Tree: Definition

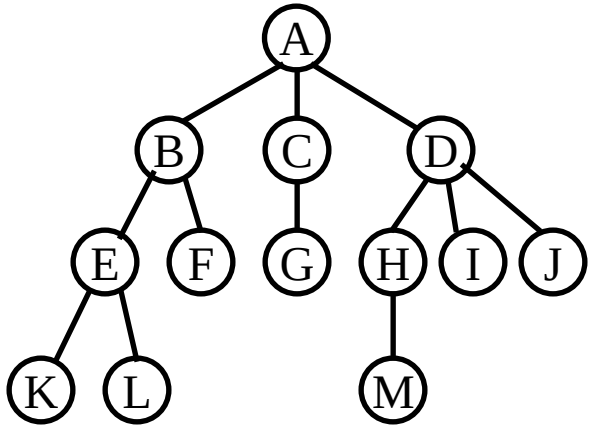
- A **tree** is a finite set of one or more nodes such that

1. There is a specially designated node r, called the **root**.
2. zero or more nonempty disjoint sets or **(sub)trees** $T_1, \cdots, T_k$, each of whose roots are connected by a directed **edge** from *r*.

Note:

- Subtrees must not connect. Therefore, every node in the tree is the root of some subtree.
- There are **N-1** edges in a tree with N nodes.
- Normally the root is drawn at the top.

- degree of a node ::= number of subtrees of the node. For example, degree(A) = 3, degree(F) = 0.



- degree of a tree ::= $\max_{node \in tree} \{degree(node)\}$
  For example, degree of this tree = 3.

- parent ::= a node that has subtrees.

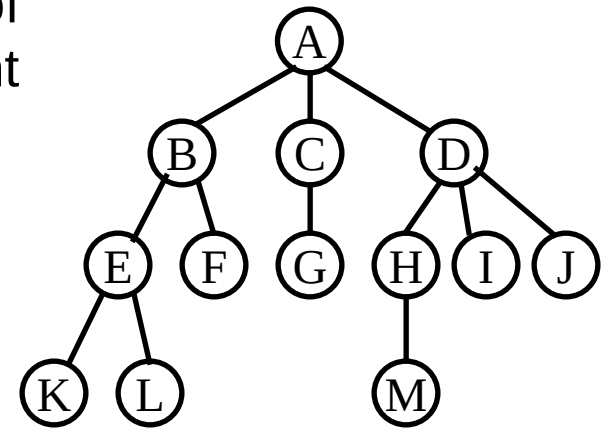
- children ::= the roots of the subtrees of a parent.

- siblings ::= children of the same parent.

- leaf (terminal node) ::= a node with degree 0 (no children).

- level of a node ::= defined by letting the root be at level one. If a node is at level l the its child nodes are at level l+1.

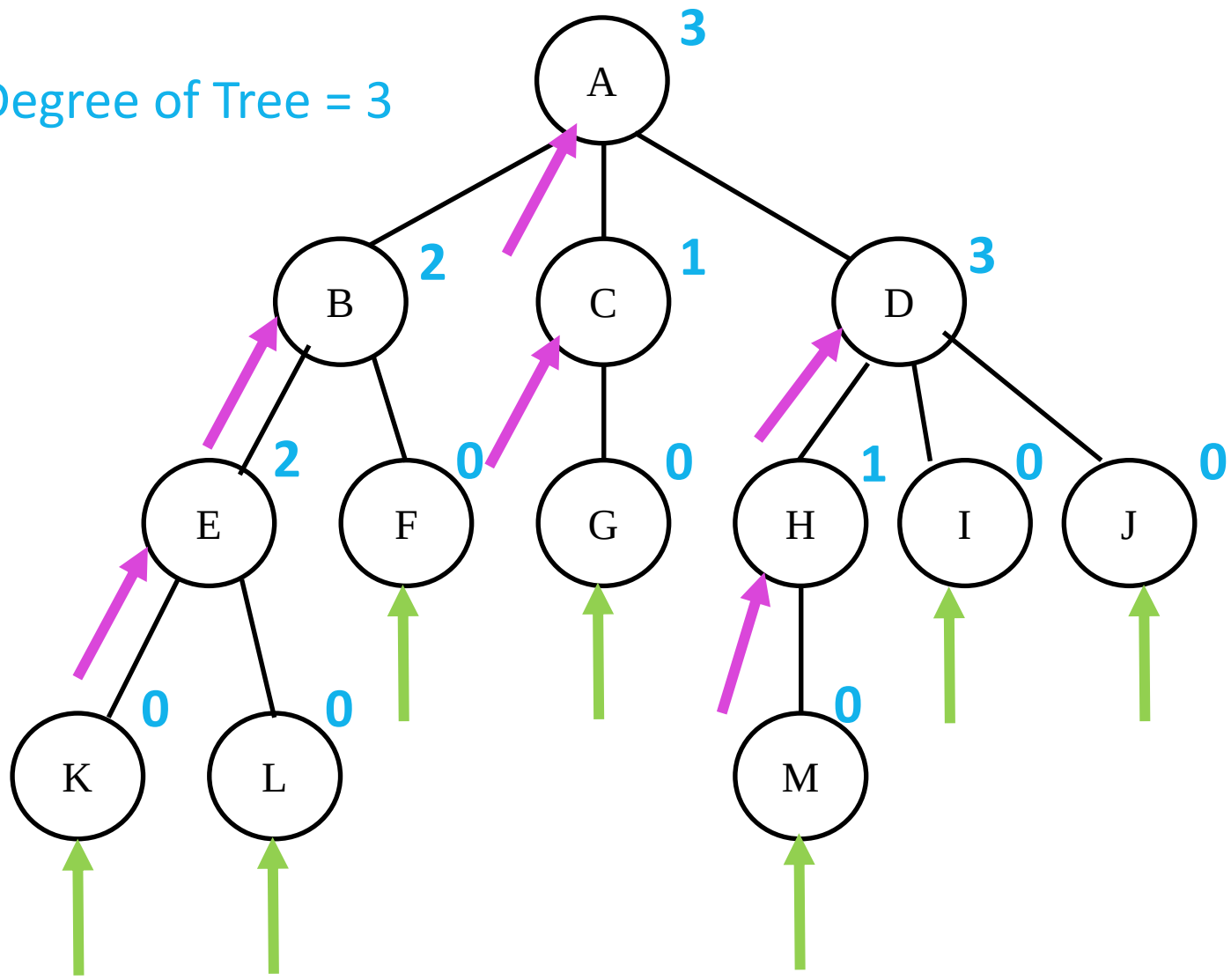
- path from n_1 to n_k ::= a (unique) sequence of nodes $n_1, n_2, \ldots, n_k$ such that n_i is the parent of n_{i+1} for $1 \leq i < k$.



- length of path ::= number of edges on the path.
- depth of n_i ::= length of the unique path from the root to n_i. Depth(root) = 1.
- height of n_i ::= length of the longest path from n_i to a leaf.
- height (depth) of a tree ::= height(root) = depth(deepest leaf).
- ancestors of a node ::= all the nodes along the path from the node up to the root.
- descendants of a node ::= all the nodes in its subtrees.

Degree

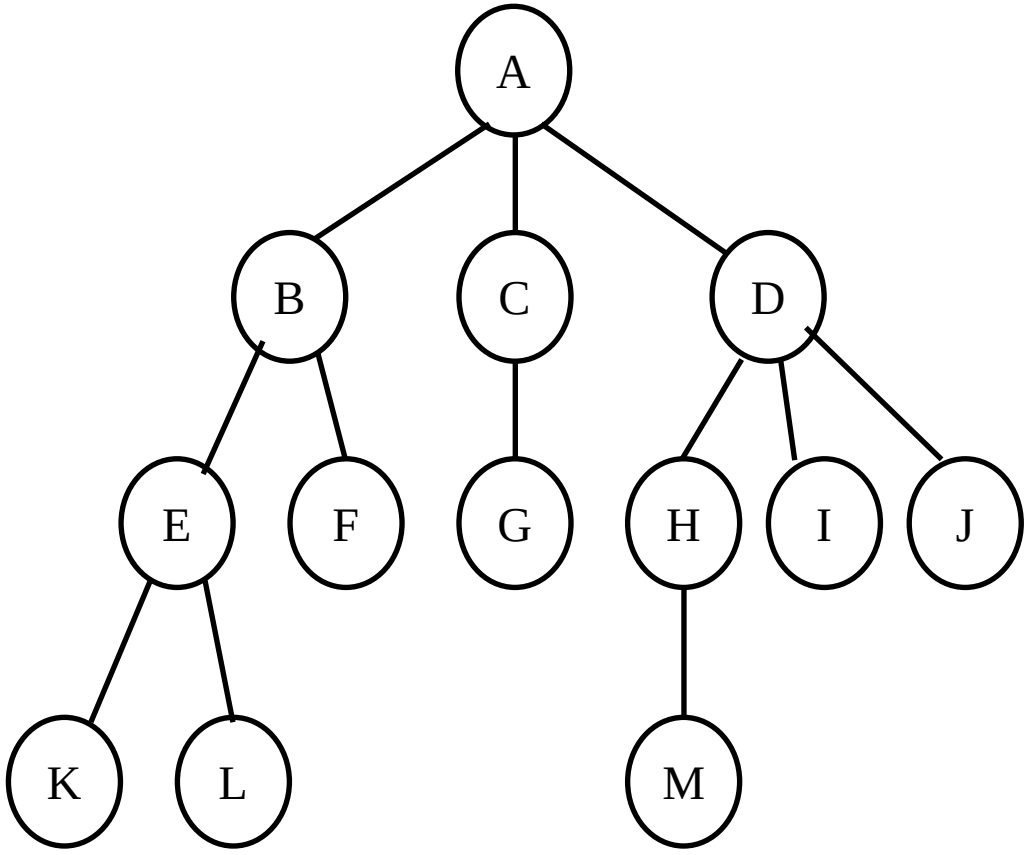
Degree of Tree = 3



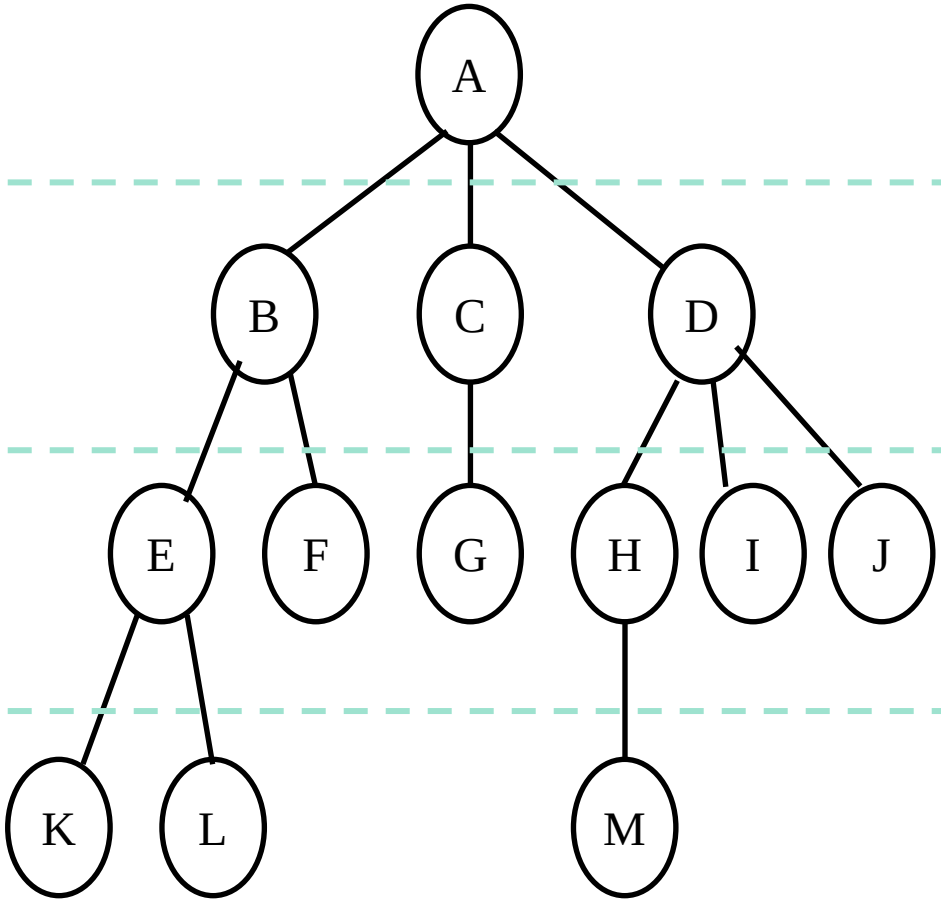
Non-Leaf or Internal or Non-Terminal Nodes

Leaf or External or Terminal Nodes

Degree of Leaf Nodes = 0



Node	Parent	Child(ren)	Ancestor(s)	Descendant(s)	Sibling(s)
A	NULL	B, C, D	NULL	All other nodes	NULL
B	A	E, F	A	E, F, K, L	C, D
C	A	G	A	G	B, D
D	A	H, I, J	A	H, I, J, M	B, C
E	B	K, L	A, B	K, L	F
F	B	NULL	A, B	NULL	E
G	C	NULL	A, C	NULL	NULL
H	D	M	A, D	M	I, J
I	D	NULL	A, D	NULL	H, J
J	D	NULL	A, D	NULL	H, I
K	E	NULL	A, B, E	NULL	L
L	E	NULL	A, B, E	NULL	K
M	H	NULL	A, D, H	NULL	NULL



Level = 1, Depth = 1, Height = 4

Level = 2, Depth = 2, Height = ?

Level = 3, Depth = 3, Height = ?

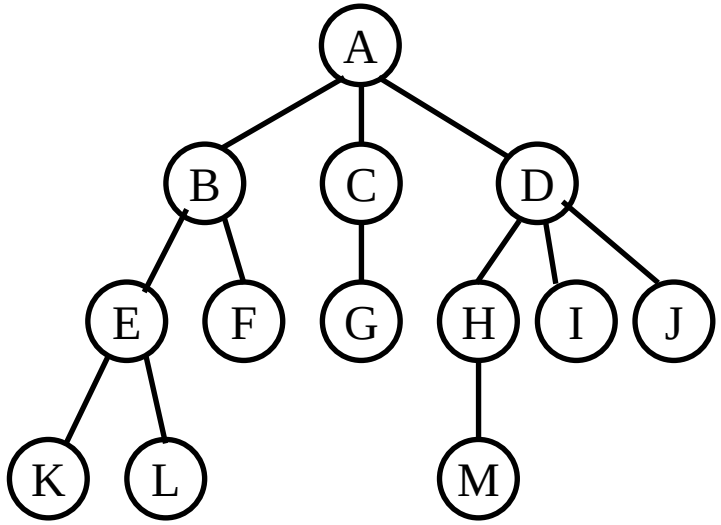
Level = 4, Depth = 4, Height = ?

* As per prescribed textbook.

Representation of Trees

List Representation

- Issues?

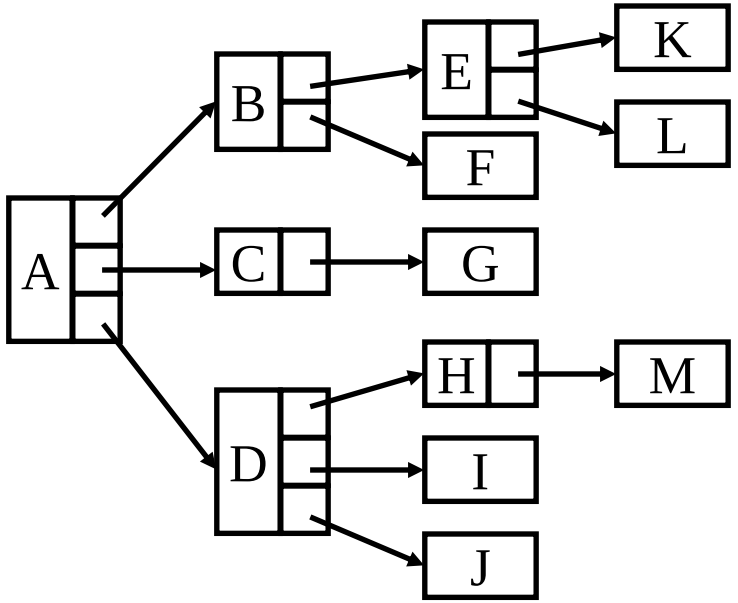


(A)

(A (B, C, D))

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

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

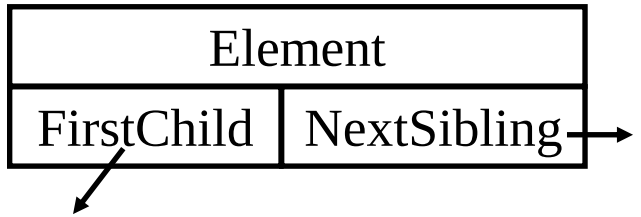


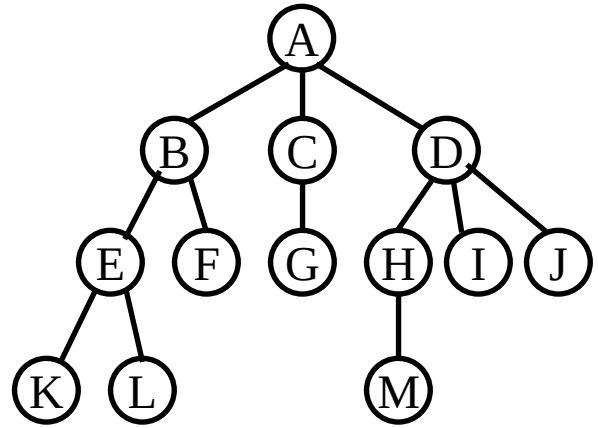
Representation of Trees

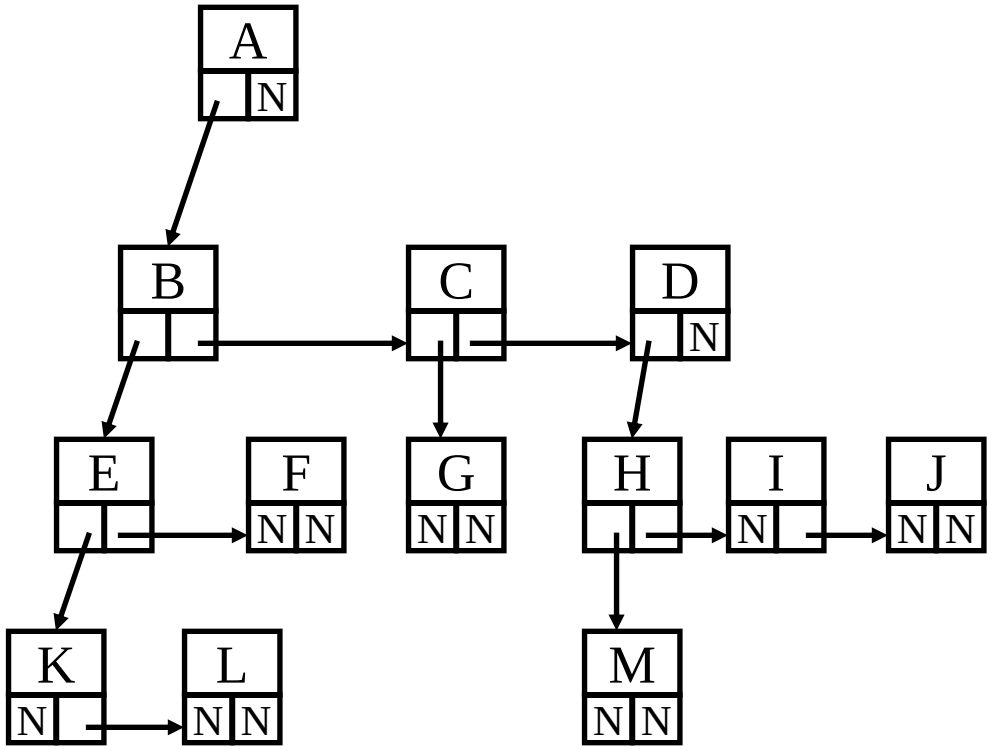
First Child-Next Sibling Representation

(Left Child-Right Sibling Representation)

Note: The representation is not unique since the children in a tree can be of any order.



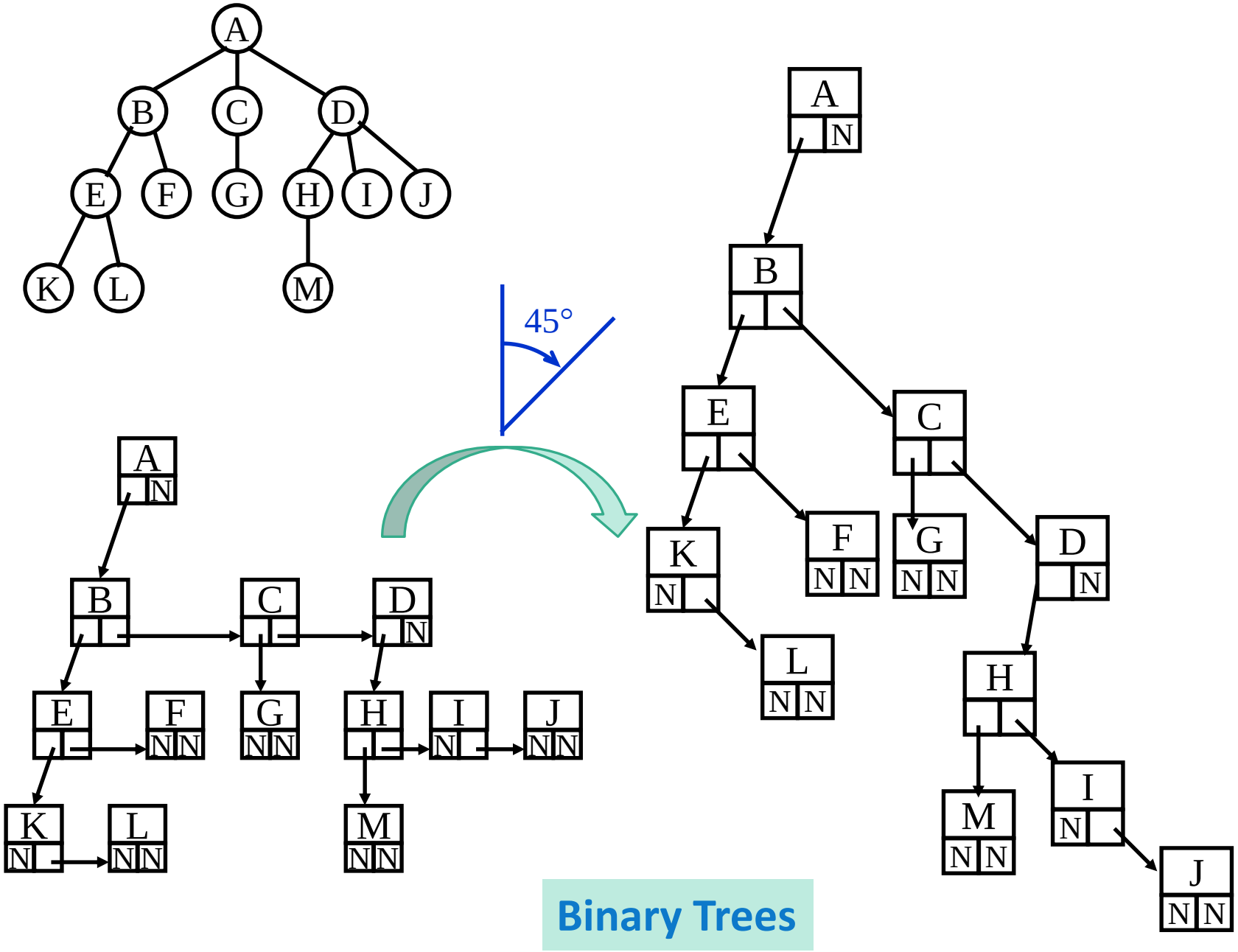




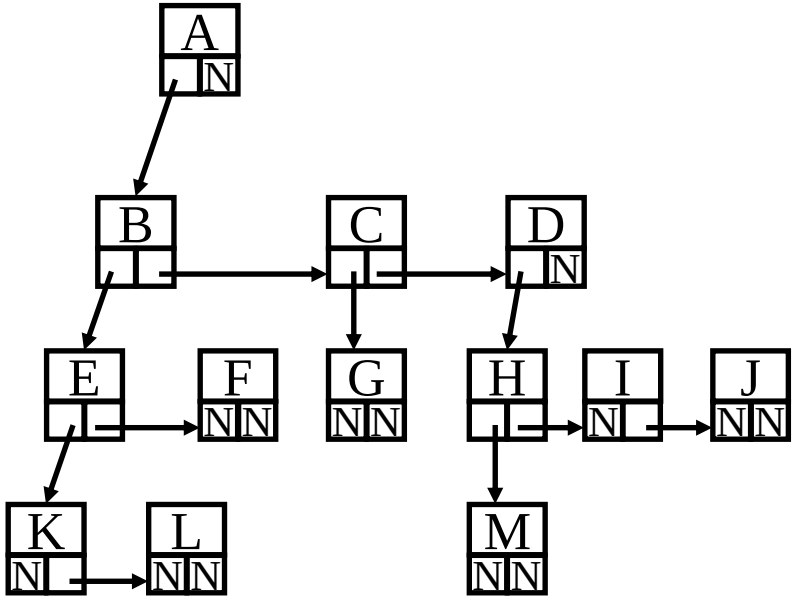
Representation of Trees

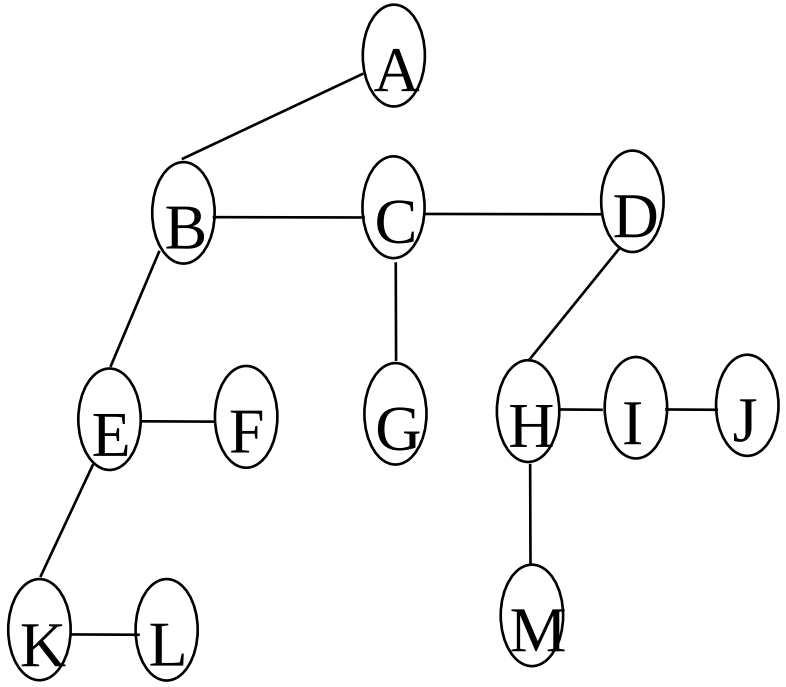
Representation as a Degree-Two Tree

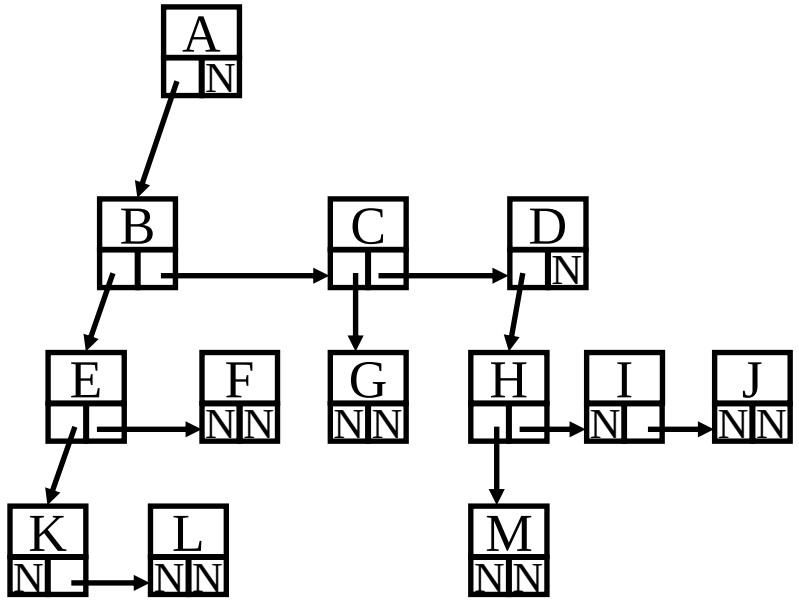
Rotate the Right Sibling Pointer in a Left Child-Right Sibling tree clockwise by 45°.

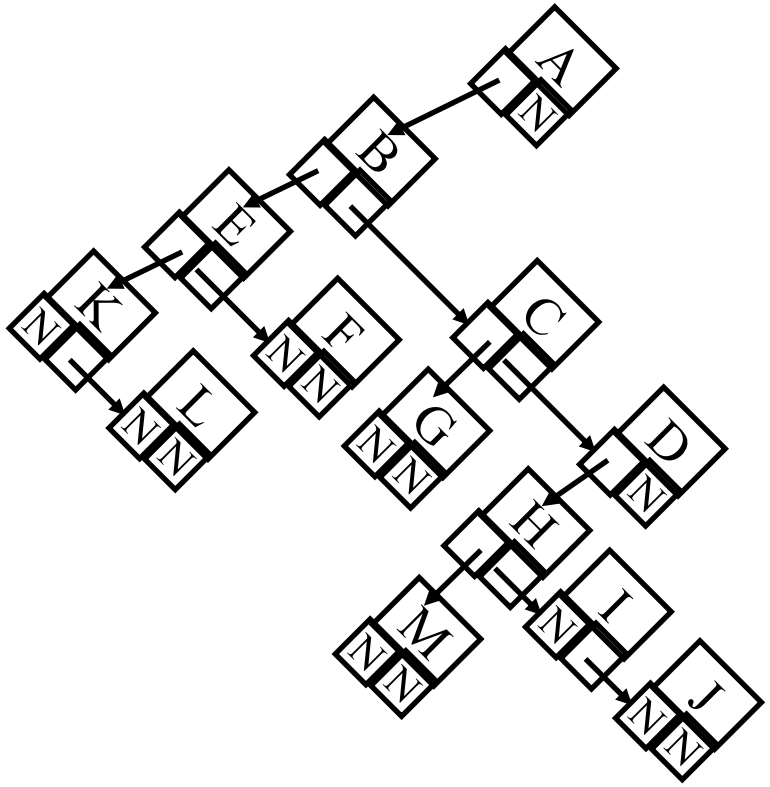


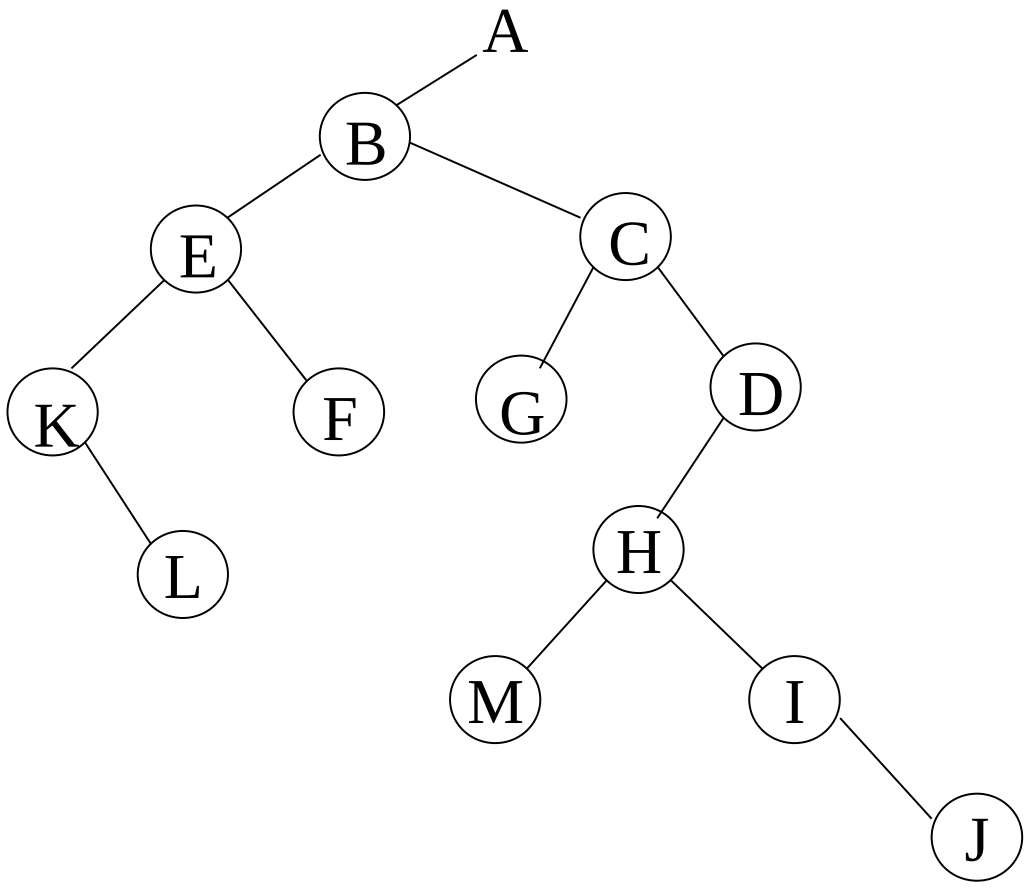
Binary Trees


A
N
B
C
D
N
E
F
N N
G
N N
H
I
N
J
N N
K
N
L
N N
M
N N


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


A
N
B
C
D
N
E
F
N N
G
N N
H
I
N
J
N N
K
N
L
N N
M
N N


A
N
B
E
K
N
L
N N
F
N N
C
G
N N
D
N
H
M
N N
I
N
J
N N


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

Binary Trees

- A binary tree is a tree in which no node can have more than two children.
- Degree of any given node must NOT exceed two.
- The left subtree is distinguished from the right subtree. What does this mean?
- The order of the subtrees is NOT irrelevant.
- 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.

Maximum Number of Nodes in Binary Tree

- The maximum number of nodes on level *i* of a binary tree is 2^{i-1}, I >=1.
- The maximum number of nodes in a binary tree of depth k is $2^{k}-1$, k>=1.
- Proof by induction

$$\sum_{i=1}^{k} 2^{i-1} = 2^{k} - 1$$

- Induction Base: #nodes at level 1=?
- Induction Hypothesis: #nodes at level i-1
- Induction Step: #nodes at level i

Relation between Number of Leaf Nodes and Nodes of Degree 2

- *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 $n_0=n_2+1$*
- Proof:

  Let n and B denote the total number of nodes & branches in T.

  n_0, n_1, n_2 : nodes with no children, 1 child, and 2 children, respectively.

  $n= n_0+n_1+n_2$,

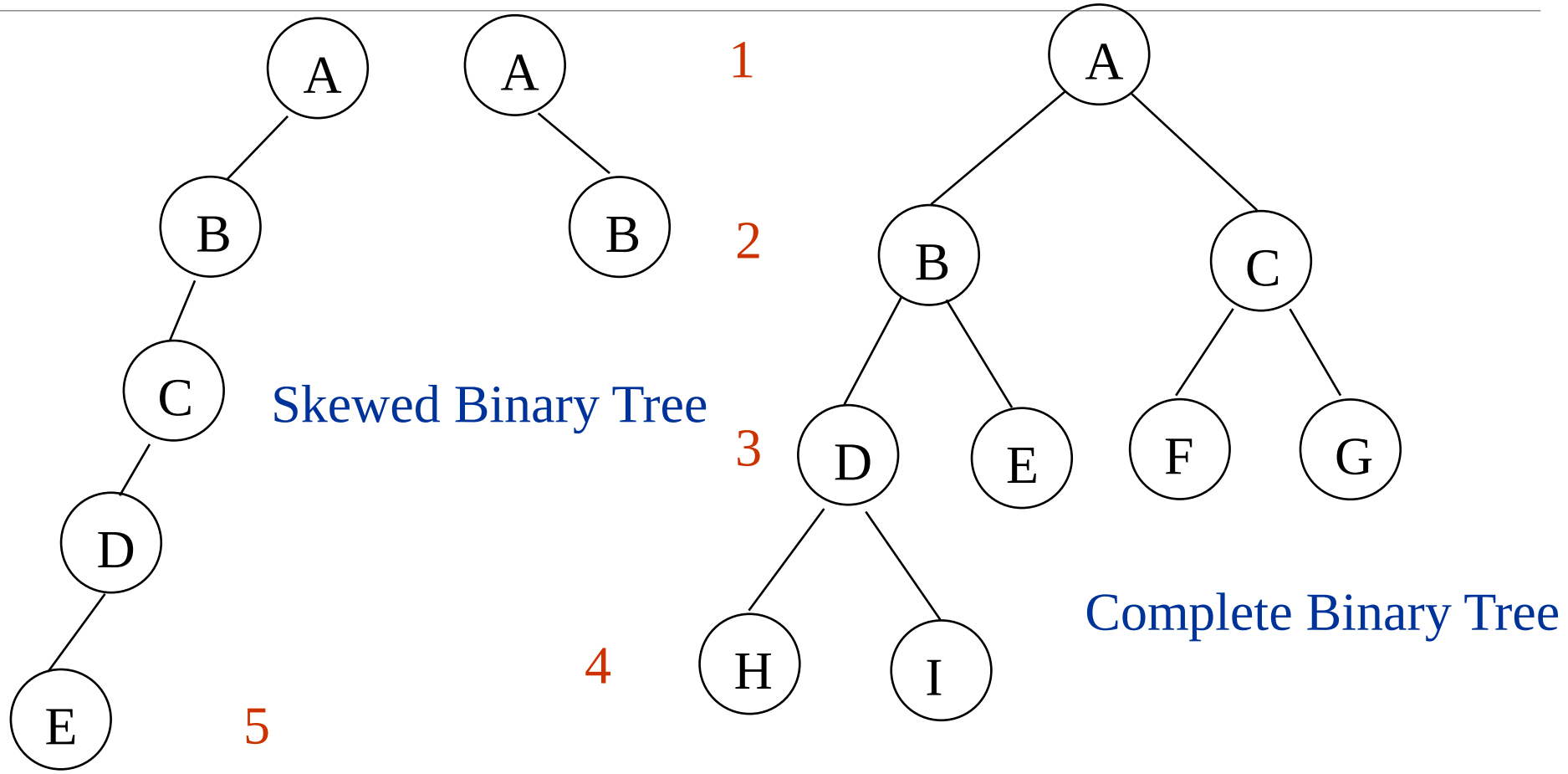
  $B+1=n$,

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

  $n_1+2n_2+1= n_0+n_1+n_2$

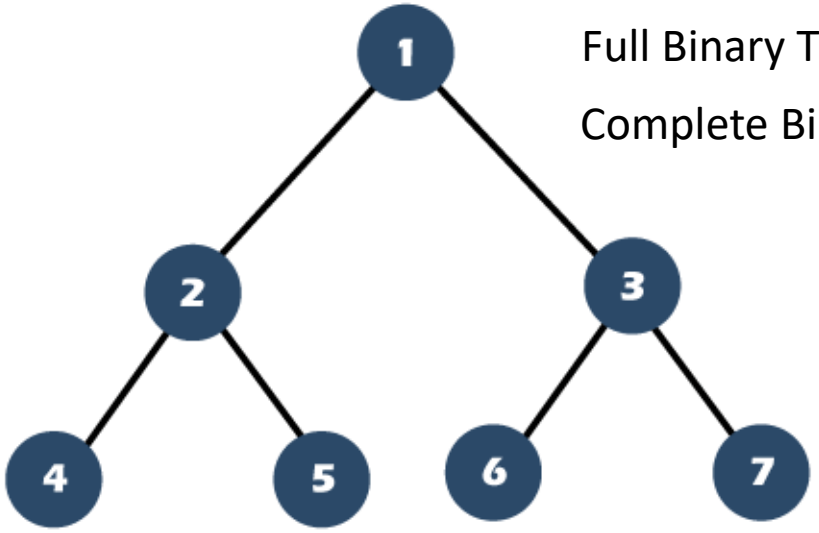
  $\Rightarrow n_0=n_2+1$

Samples of Trees



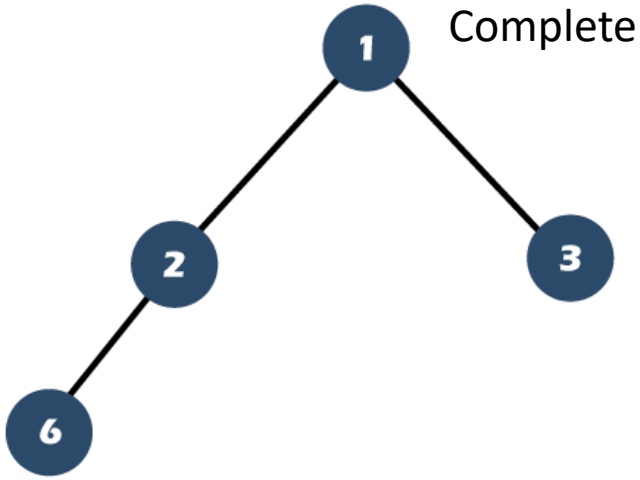
Full Binary Tree versus Complete Binary Tree

- A full binary tree of depth *k* is a binary tree of depth *k* having $2^k - 1$ nodes, *k*>=0.
- Full binary tree: Binary tree where all nodes have zero or two children.
- A binary tree with *n* nodes and depth *k* is **complete** *iff* all of its nodes are filled except possibly the last level, where in the last level the nodes are filled from left to right .
- Height of a complete binary tree with n nodes is ceiling($\log_2(n+1)$)

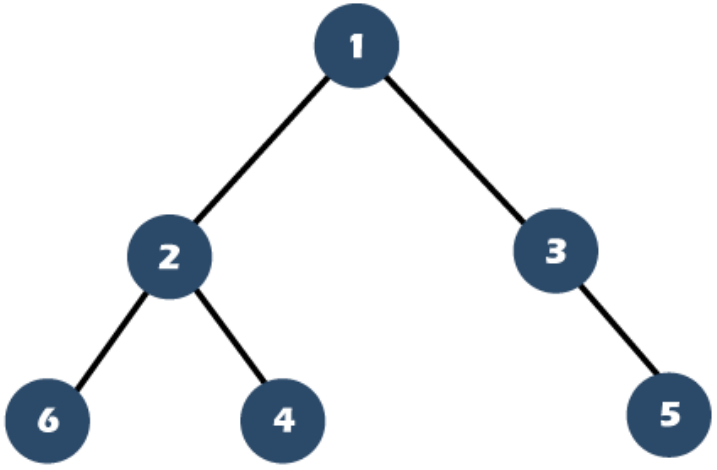


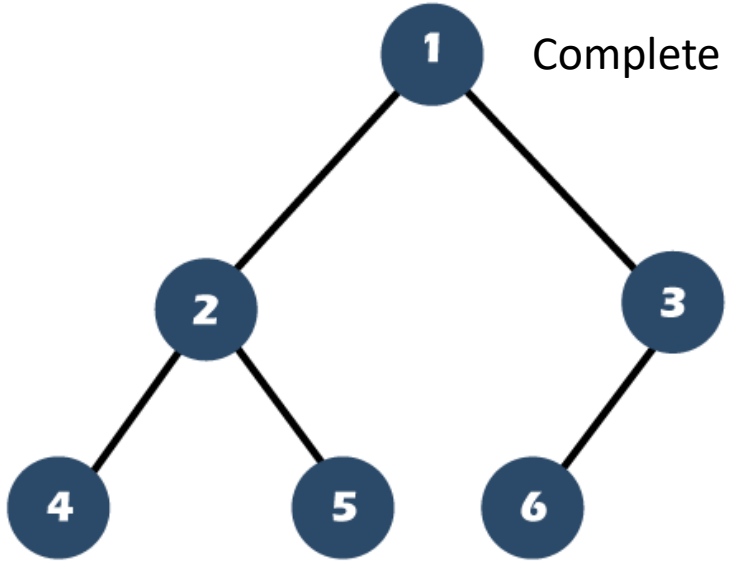
Full Binary Tree

Complete Binary Tree

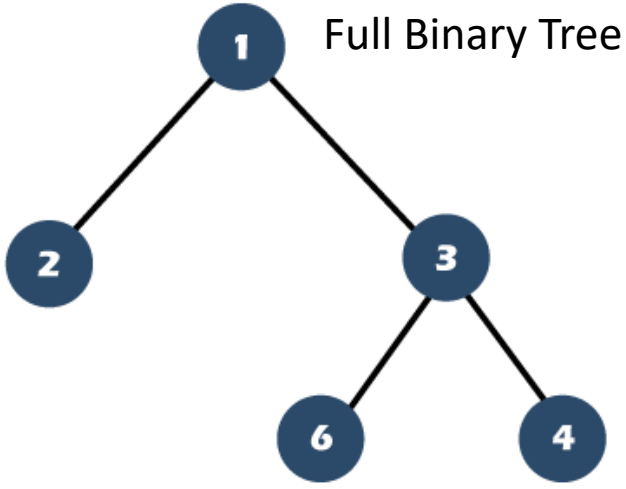


Complete Binary Tree





Complete Binary Tree

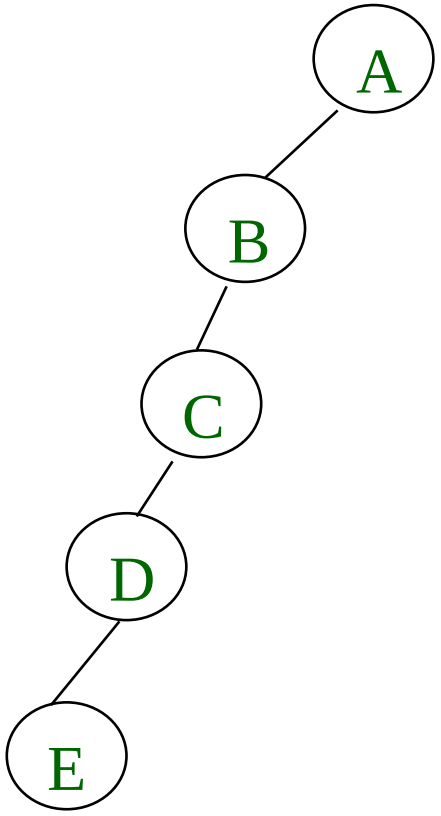


Full Binary Tree

Binary Tree Representations

- If a complete binary tree with '*n*' nodes is represented sequentially, then for any node with index i, 1<=i<=n, we have:
  - parent(i) is at floor(i/2) if i!=1. If i=1, i is at the root and has no parent.
  - left_child(i) is at 2i if 2i<=n. If 2i>n, then i has no left child.
  - right_child(i) is at 2i+1 if 2i +1 <=n. If 2i +1 >n, then i has no right child.

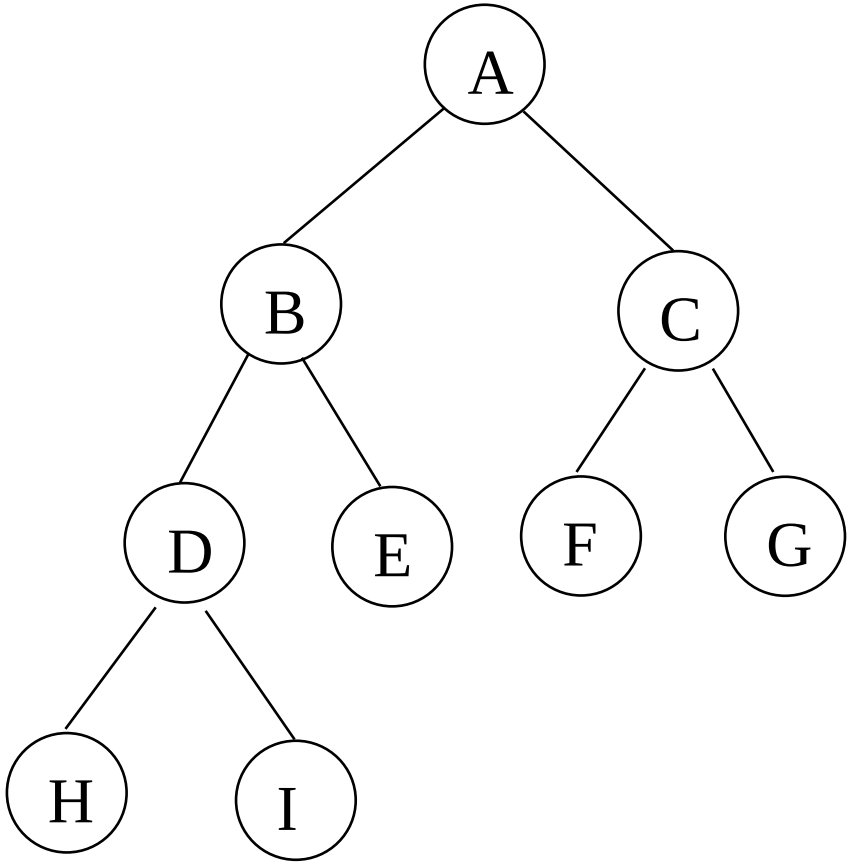
Sequential Representation



[1]	A
[2]	B
[3]	--
[4]	C
[5]	--
[6]	--
[7]	--
[8]	D
[9]	--
.	.
[16]	E

(1) waste space

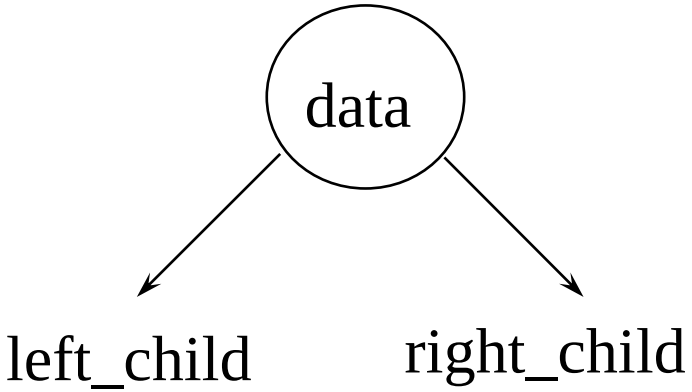
(2) insertion/deletion problem



[1]	A
[2]	B
[3]	C
[4]	D
[5]	E
[6]	F
[7]	G
[8]	H
[9]	I

Linked Representation

```
class node {
 int data;
 node *left_child, *right_child;
};
```



Tree Creation

Tree Creation

```
class Node {
  public:
    int data;
    Node* left;
    Node* right;

    // Val is the key or the value that has to be added to the data part
    Node(int val)
    {
        data = val;

        // Left and right child for node will be initialized to null
        left = NULL;
        right = NULL;
    }
};
```

Tree Creation

```
int main(){
    /*create root*/
    Node* root = new Node(1);
    root->left = new Node(2);
    root->right = new Node(3);
    root->left->left = new Node(4);
    return 0;
}
```



```
int main()
{
    /*create root*/
   Node* root = new Node(1);
   /* following is the tree after above statement
          1
         /\
      NULL   NULL
   */

   root->left = new Node(2);
   root->right = new Node(3);
   /* 2 and 3 become left and right children of 1
            1
           /  \
           2    3
          /\   /\
        NULL NULL NULL NULL
   */
    root->left->left = new Node(4);
return 0;
}
```

Questions

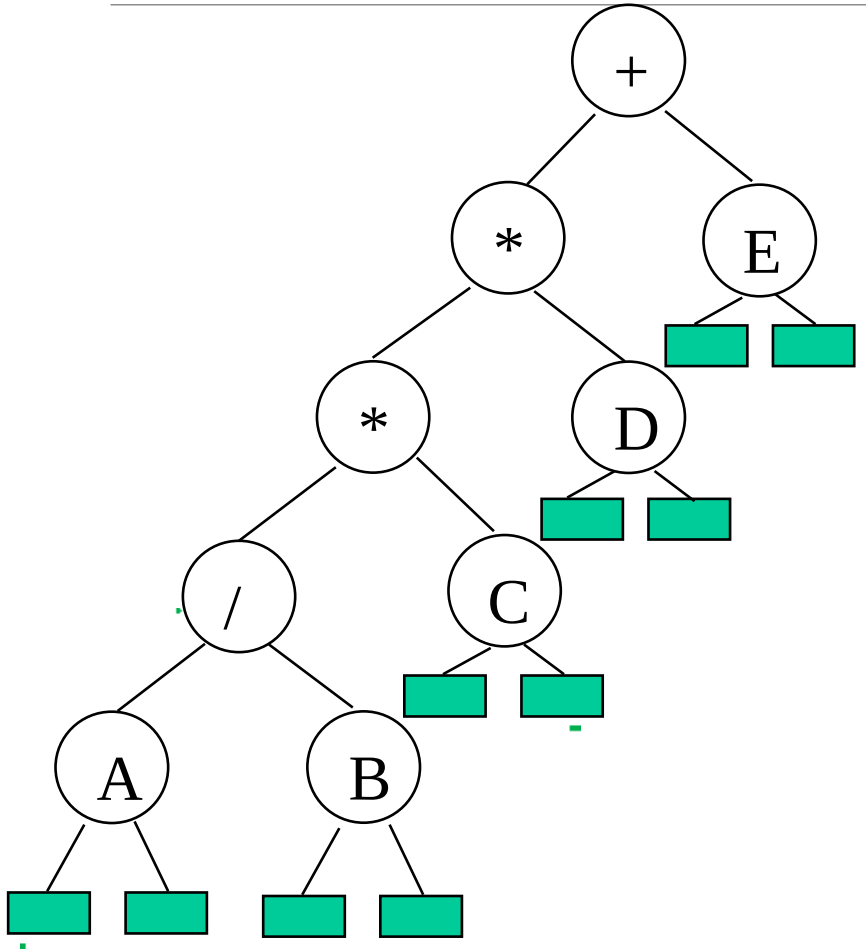
Write a menu-driven dynamic CPP program to:

- Create a Binary Tree.

Binary Tree Traversals

- Let L, N, and R stand for moving left, visiting the node, and moving right.
- There are six possible combinations of traversal: LNR, LRN, NLR, NRL, RNL, RLN
- Adopt convention that we traverse left before right, only 3 traversals remain: LNR, LRN, NLR
- Inorder, Postorder, Preorder

Arithmetic Expression Using Binary Tree



Inorder Traversal
A / B * C * D + E
Infix Expression

Postorder Traversal
A B / C * D * E +
Postfix Expression

Preorder Traversal
+ * * / A B C D E
Prefix Expression

Level Order Traversal
+ * E * D / C A B

Inorder Traversal (Recursive Version)

```
void inorder(node *root)
/* inorder tree traversal */
{
    if (root) {
        inorder(root->left_child);
        cout<<root->data;
        indorder(root->right_child);
    }
}
```

A / B * C * D + E

Preorder Traversal (Recursive Version)

```
void preorder(node *root)
/* preorder tree traversal */
{
    if (root) {
        cout<<root->data;
        preorder(root->left_child);
        predorder(root->right_child);
    }
}
```

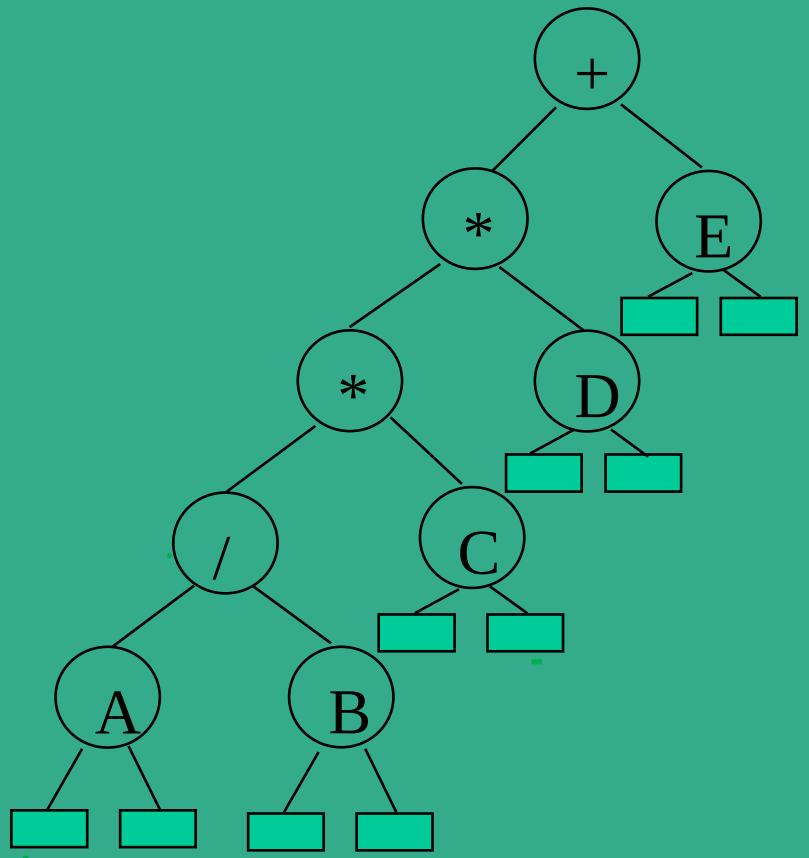
+ * * / A B C D E

Postorder Traversal (Recursive Version)

```
void postorder(node *root)
/* postorder tree traversal */
{
    if (root) {
        postorder(root->left_child);
        postdorder(root->right_child);
        cout<<root->data;
    }
}
```

A B / C * D * E +

Inorder Traversal: Trace



Call of inorder	Value in root	Action
1	+	
2	*	
3	*	
4	/	
5	A	
6	NULL	
5	A	Print
7	NULL	
4	/	Print
8	B	
9	NULL	
8	B	Print
10	NULL	
3	*	Print

Call of inorder	Value in root	Action
11	C	
12	NULL	
11	C	Print
13	NULL	
2	*	Print
14	D	
15	NULL	
14	D	Print
16	NULL	
1	+	Print
17	E	
18	NULL	
17	E	Print
19	NULL	

Tree Iterative Traversals

TREES

Iterative Inorder Traversal (using stack)

The iterative method to find the inorder traversal is as follows:

1. Prepare the stack and the binary tree.
2. Push the node into the stack and find the left child. If the left child is not NULL, push it into stack and find its left. Continue this process until you find the NULL left child.
3. If the left child is NULL, pop a node from the stack.
4. Print the node and find its right child. Then go to step 2 to repeat the process.
5. If the right child is NULL, pop a node from the stack and go to 4^{th} step.
6. If the right child is NULL and the stack is empty, stop the traversal.

Iterative Inorder Traversal (using stack)

```
void iter_inorder(node *root)
{
  int top= -1; /* initialize stack */
  node stack[MAX_STACK_SIZE];
  for (;;) {
   for (; root; root=root->left_child)
     push(root);/* add to stack */
   root= pop();
                  /* delete from stack */
   if (!root) break; /* empty stack */
   cout<<root->data;
   root = root->right_child;
  }
}
```

Iterative Preorder Traversal (using stack)

The iterative method for preorder traversal is given below:

1. Prepare the stack and the tree.
2. Print the root information and push the root into the stack and move towards the left of the root until the root becomes NULL.
3. When the root reaches NULL, pop an element from the stack and check its right. If the right child node is existing, consider that as the root, then go to step 2.

Iterative Preorder Traversal (using stack)

4. If the right child is also NULL, pop an element from the stack.
5. Check the right child of the popped element. If it is present then go to step 2. If not present then pop an element from the stack.
6. Continue these steps until root reaches NULL and the stack is empty.

Iterative Preorder Traversal (using stack)

```
void itpre(node *r){
 for(;;) {
  for(;r;r=r->lcl) {
  cout<<r->info<<" ";
  s1.push(r);
}
r=s1.pop();
if(!r)break;
r=r->rcl;
}}
```

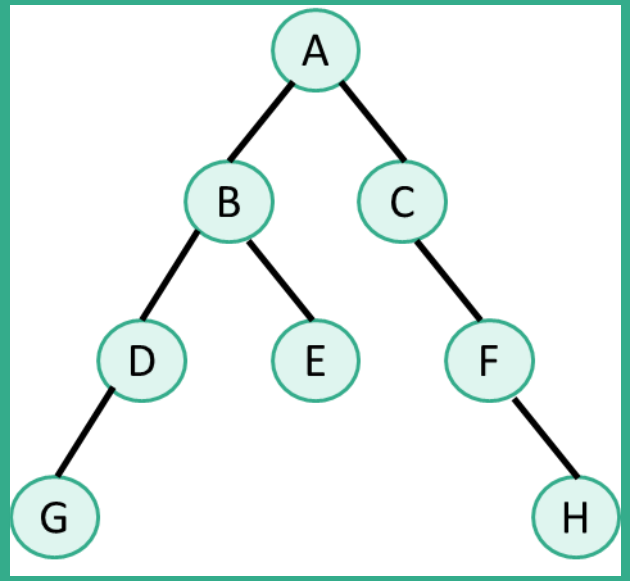
Iterative Postorder Traversal (using stack)

1. Prepare the stack and binary tree
2. Start from the root, check if its right child is present or not.
3. If the right child is present, then push the right child into stack and then push the root to stack.
4. If the right child is not present for the root, push the root to stack.
5. Then change the root to root's left go to step 2.
6. If the root becomes NULL, pop an element from the stack and call it root.

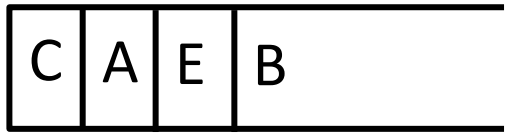
Iterative Postorder Traversal (using stack)

7. Check the right child of the root. If present, check if it is equal to the top of the stack. If it is equal to top of the stack, pop the stack top and push the root back to stack and then change the root to root's right.
8. If the root's right is NULL or the root's right is not equal to the top of the stack, print the root information and assign the root to NULL.
9. Repeat from step 2 till the stack is empty.

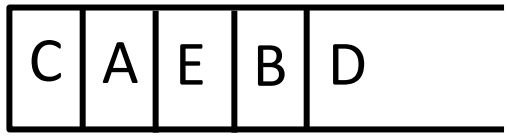
Iterative Postorder Traversal (using stack)



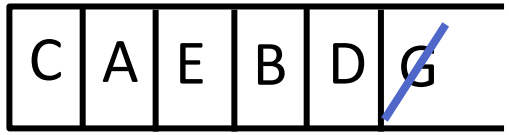
Root = LC (A) = B



Root = LC (B) = D



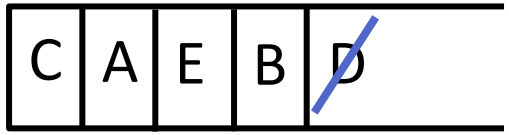
Root = LC (D) = G



Root = LC (G) = NULL

Pop G Root = G

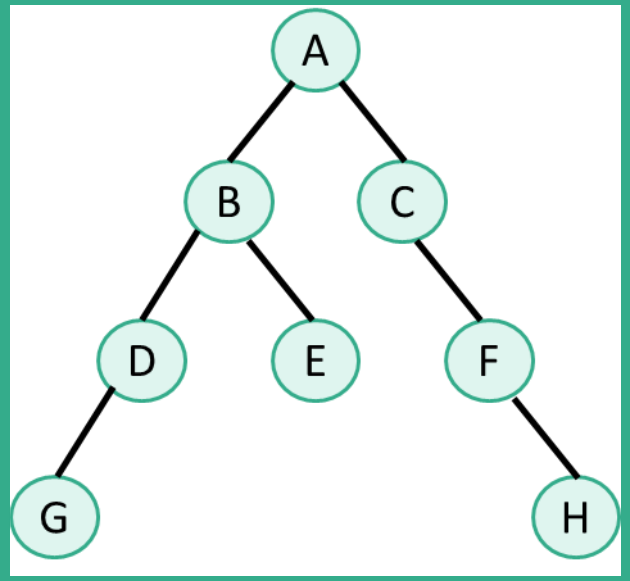
RC (G) = NULL => **Print (G)**
Root = NULL

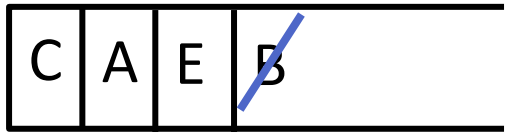


Pop D Root = D

RC (D) = NULL => **Print (D)**
Root = NULL

Iterative Postorder Traversal (using stack)

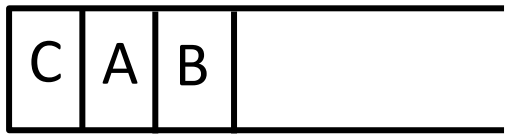




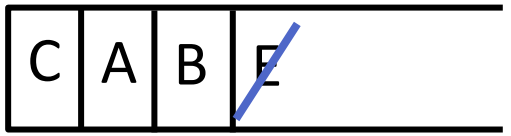
Pop B Root = B

RC (B) = E = Top of Stack

Pop E, Push B, Root = E



RC (E) = NULL

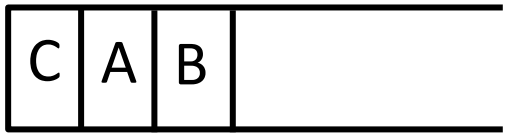


Root = LC (E) = NULL

Pop E Root = E

RC (E) = NULL => **Print (E)**

Root = NULL



Pop B Root = B

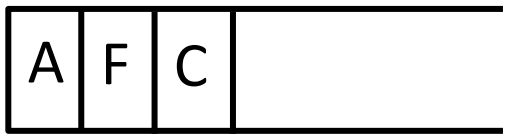
RC (B) != Top of Stack=> **Print (B)**

Root = NULL

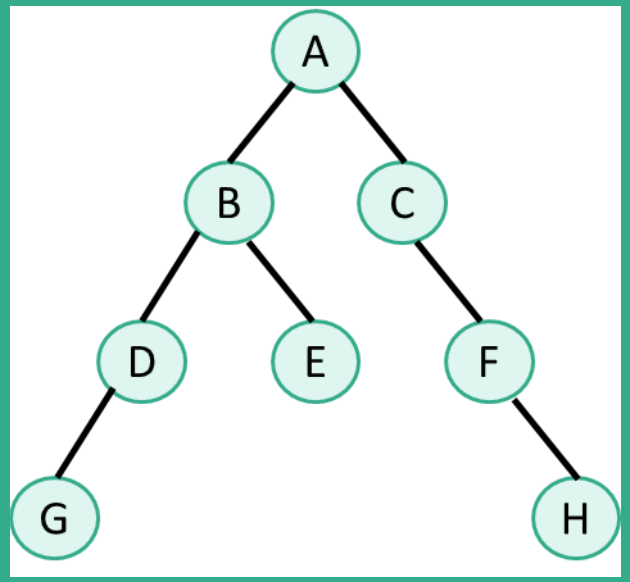
Pop A Root = A

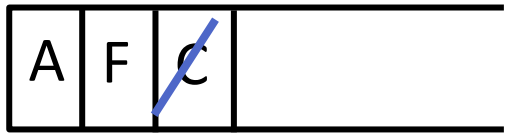
RC (A) = C = Top of Stack

Pop C, Push A, Root = C



Iterative Postorder Traversal (using stack)

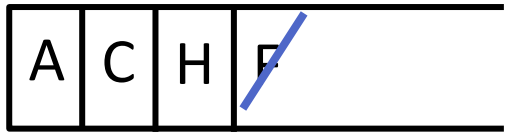




Root = LC (C) = NULL

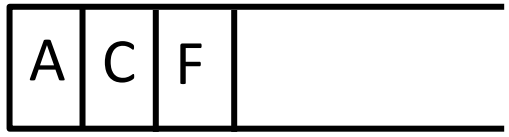
Pop C Root = C

RC (C) = F= Top of Stack
Pop F , Push C, Root = F



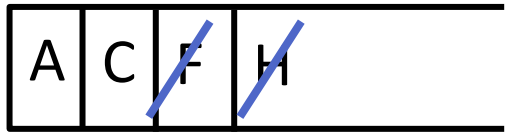
Root = LC (F) = NULL

Pop F Root = F
RC (F) = H= Top of Stack
Pop H , Push F, Root = H



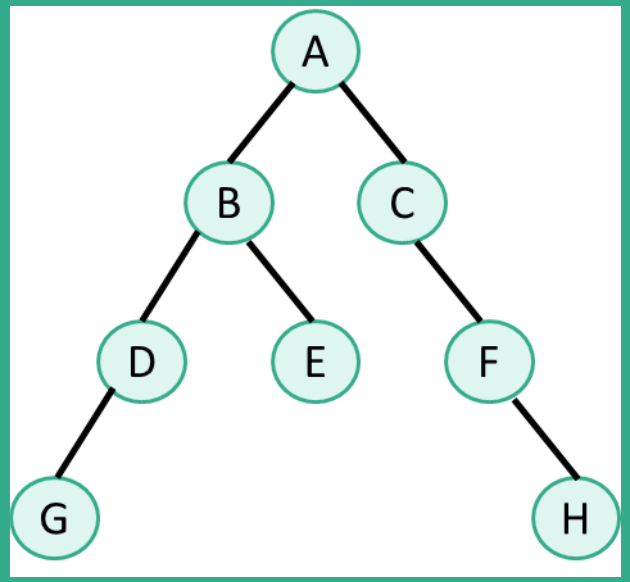
Root = LC (H) = NULL

Pop H Root = H
RC (H) = NULL => **Print (H)**
Root = NULL



Pop F Root = F
RC (F) != Top of Stack=> **Print (F)**
Root = NULL

Iterative Postorder Traversal (using stack)



Pop C

RC (C) != Top of Stack=> **Print (C)**
Root = NULL

Pop A

RC (A) != Top of Stack=> **Print (A)**
Root = NULL

G D E B H F C A

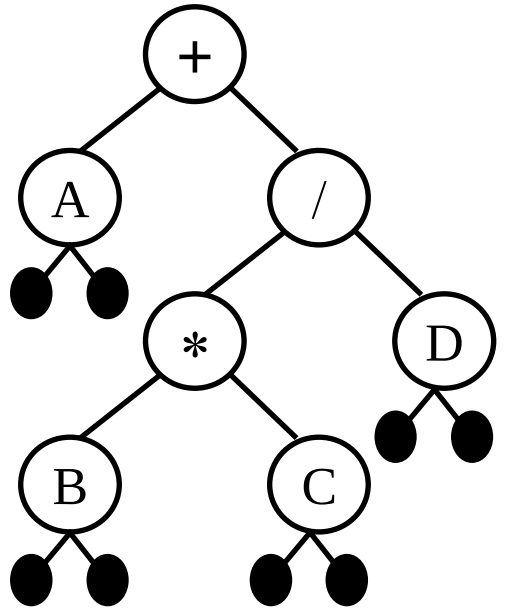
Level Order Traversal (using queue)

```
void levelorder(node *root)
{
 node *q[MAX], *cur;
 int f=0, r=-1;
 q[++r]=root;
 while(f<=r)
 {
  cur=q[f++];
  cout<<cur->info;
  if(cur->lcl!=NULL)
       q[++r]=cur->lcl;
  if(cur->rcl!=NULL)
       q[++r]=cur->rcl;
  }
 cout<<endl;
}
```

Expression Trees (syntax trees)

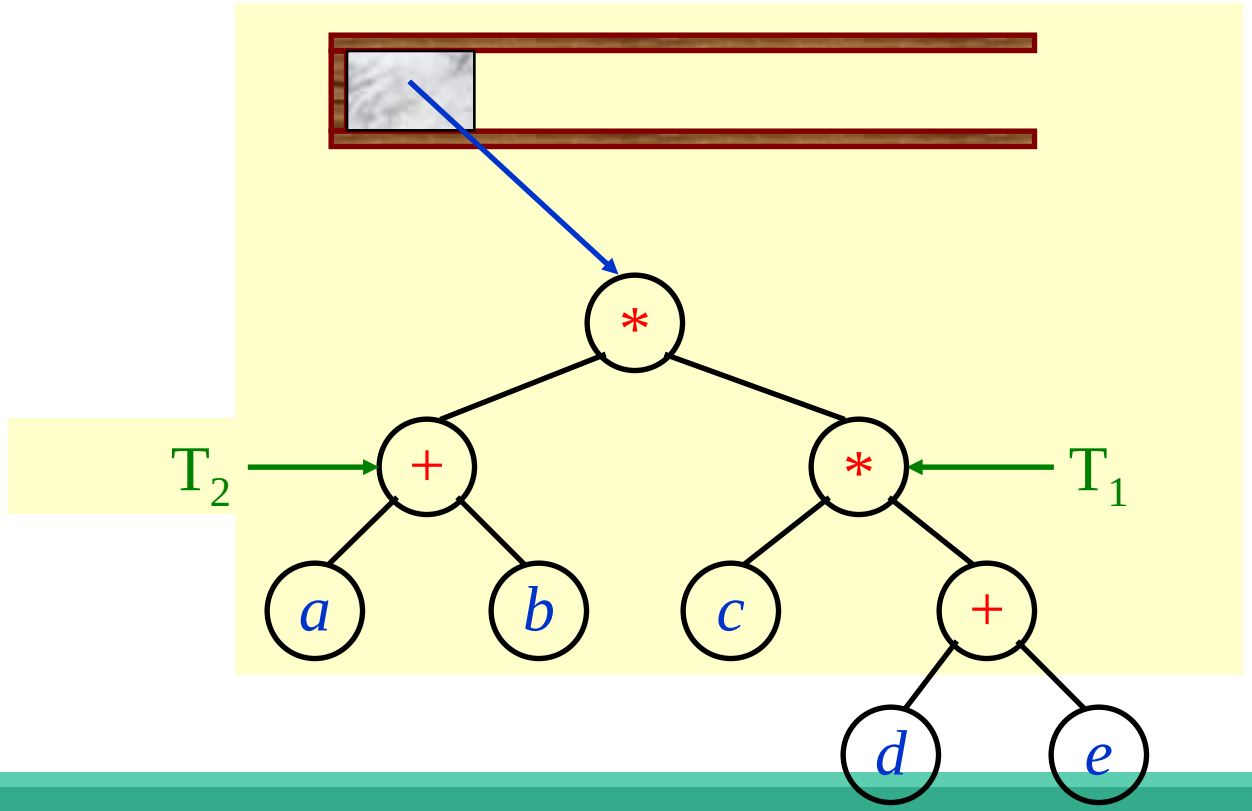
〖Example〗 Given an infix expression:

$A + B * C / D$



Constructing an Expression Tree (from **postfix** expression)

〖Example〗 (*a* + *b*) * (*c* * (*d* + *e*)) = *a b* + *c d e* + * *



Books

- Ellis Horowitz, Sartaj Sahni, Susan Anderson-Freed, Fundamentals of Data structures in C (2e), Silicon Press, 2008.
- Ellis Horowitz, Sartaj Sahni, Dinesh Mehta, Fundamentals of Data Structures in C++ (2e), Galgotia Publications, 2008.