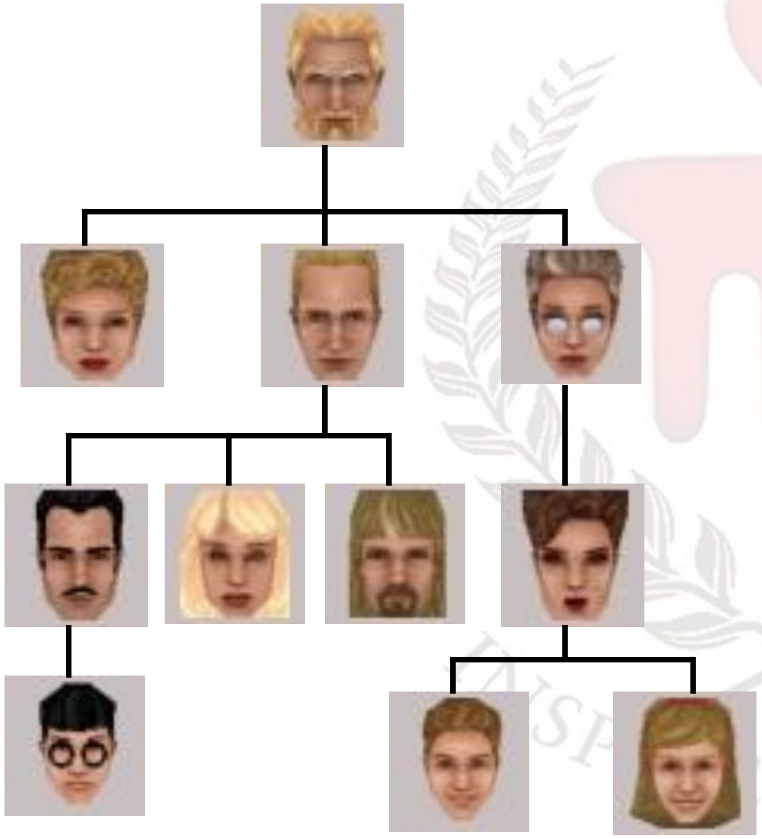
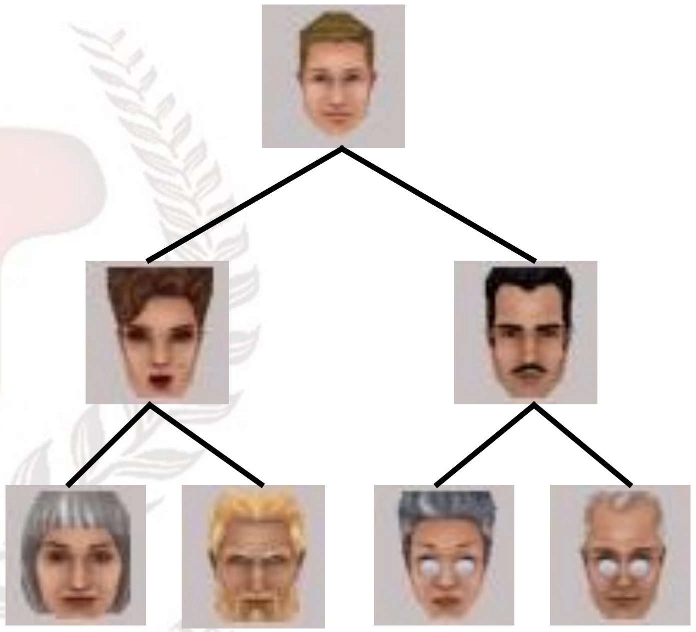
TREES

1 . Preliminaries/Terminology



Lineal Tree



Pedigree Tree
(binary tree)

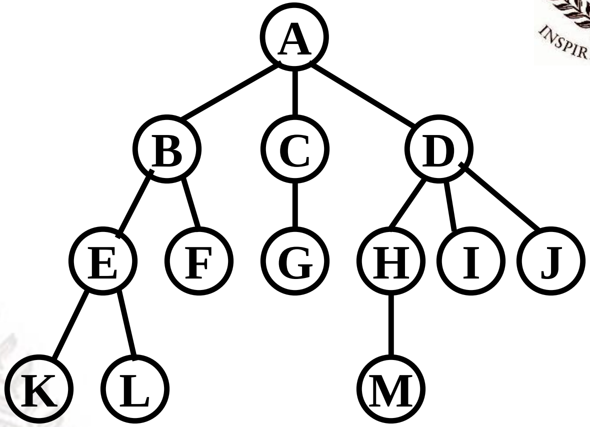
【Definition】A tree is a collection(finite set) of nodes. The collection can be empty; otherwise, a tree consists of

(1) a distinguished node *r*, called the root;

(2) and zero or more nonempty (sub)trees $T_1, \cdots, T_k$, each of whose roots are connected by a directed edge from *r*.

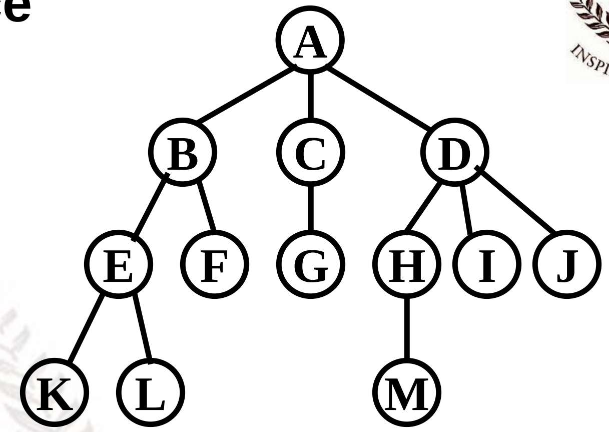
Note:

- Subtrees must not connect together. 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 $1 \le i < k$.



- 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.

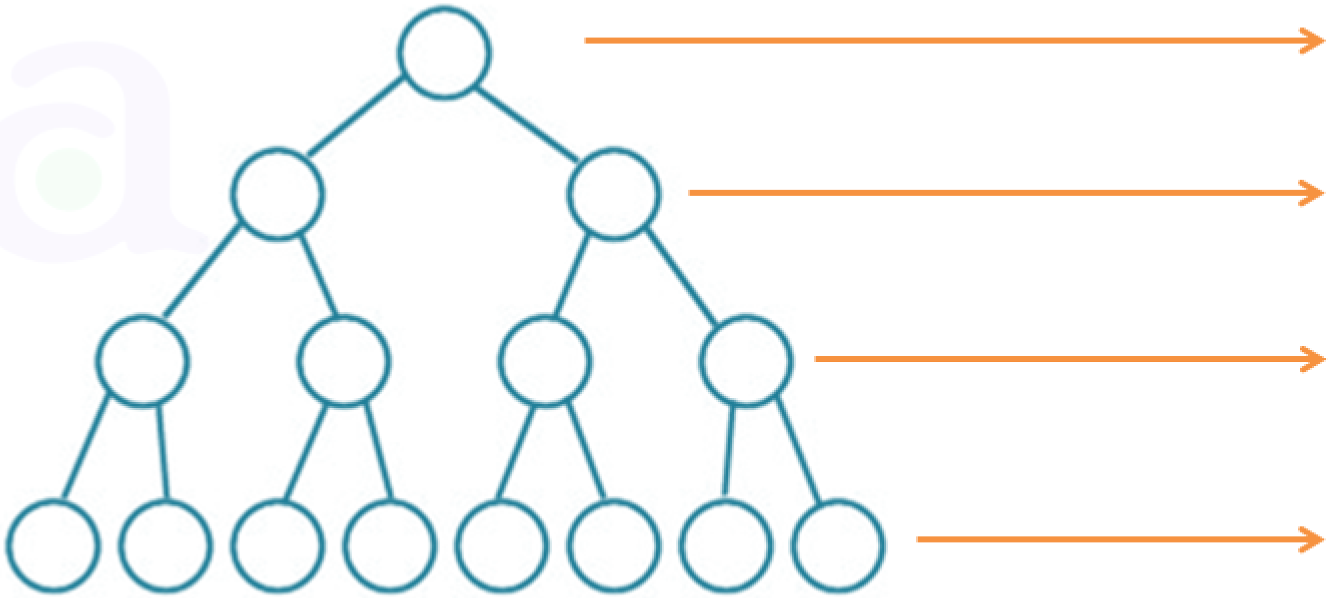
- height of n_i ::= length of the longest path from n_i to a leaf.

- 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.

Depth start with 1 from root node



Depth 1, node = 1

Depth 2, node = 2

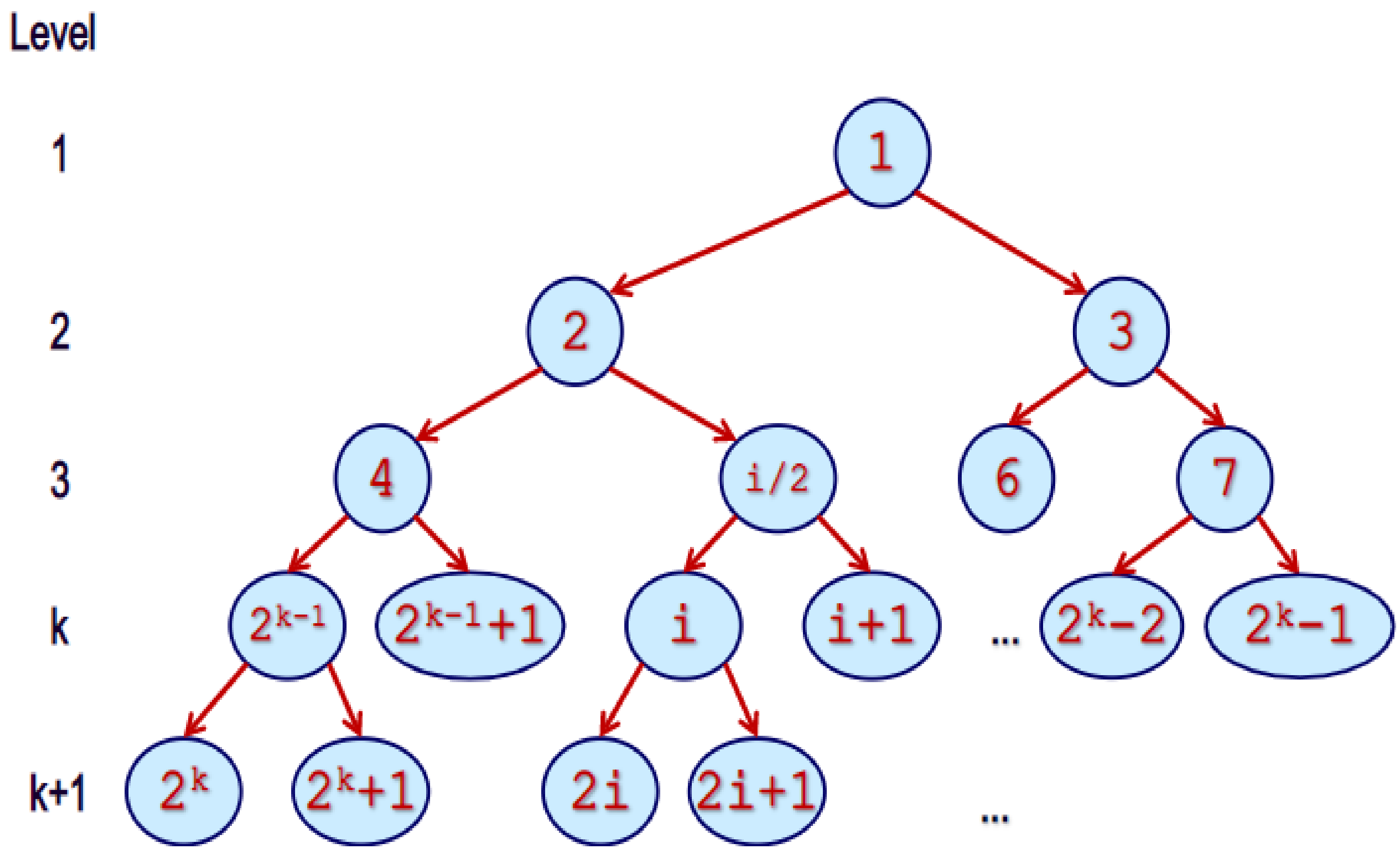
Depth 3, node = 4

Depth 4, node = 8

The maximum number of nodes in a binary tree of depth k is $2^k - 1$ where k >= 1

$2^4 - 1 = 2^4 - 1 = 16 - 1 = 15$

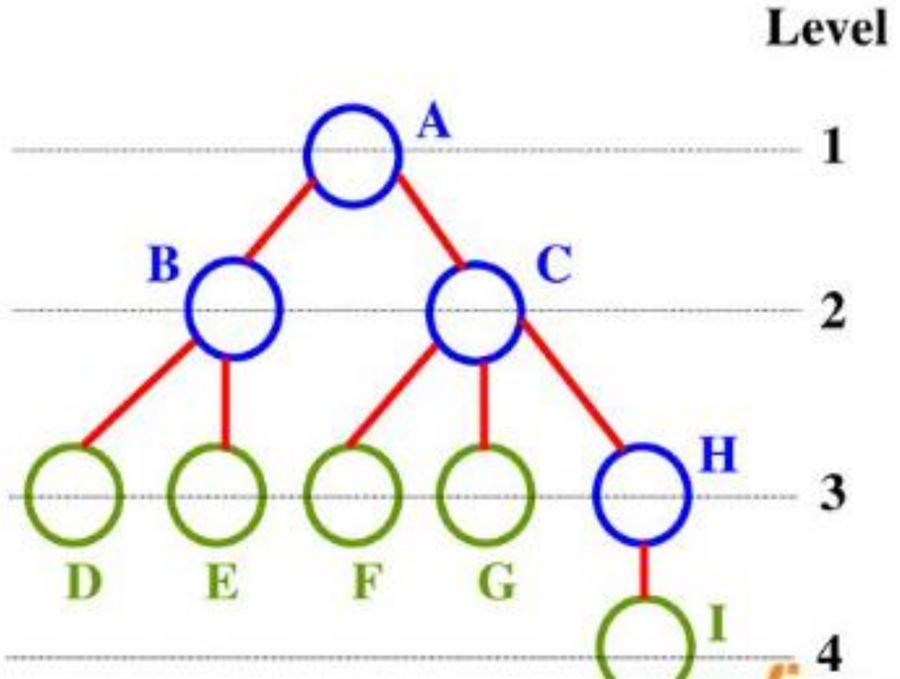
(1+2+4+8 = 15)


Level
1
2
3
k
k+1
1
2
3
4
i/2
6
7
2^k-1
2^k-1+1
i
i+1
...
2^k-2
2^k-1
2^k
2^k+1
2i
2i+1
...

Example

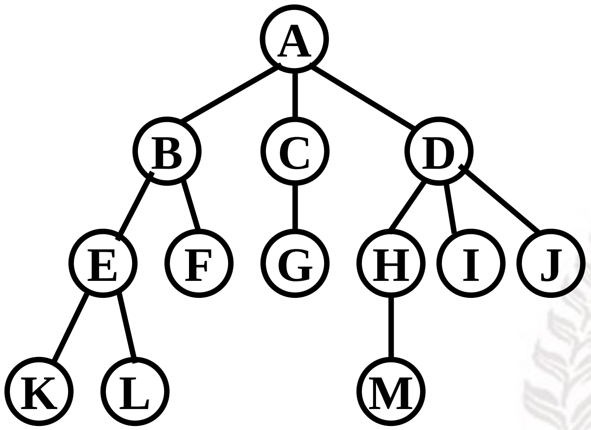
Property: (# edges) = (#nodes) - 1

- A is the *root* node
- B is the *parent* of D and E
- C is the *sibling* of B
- D and E are the *children* of B
- D, E, F, G, I are *external nodes*, or *leaves*
- A, B, C, H are *internal nodes*
- The *level* of E is 3
- The *height (depth)* of the tree is 4
- The *degree* of node B is 2
- The *degree* of the tree is 3
- The *ancestors* of node I is A, C, H
- The *descendants* of node C is F, G, H, I



2. Implementation

❖ List Representation

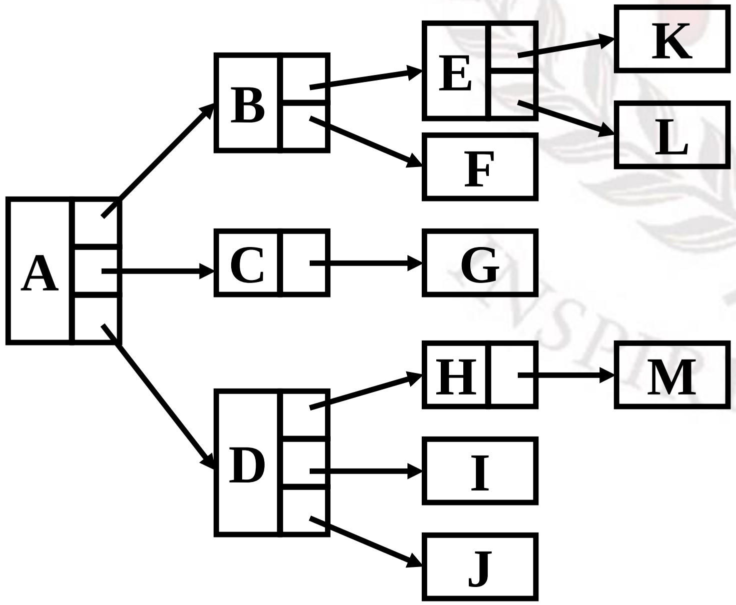


(A)

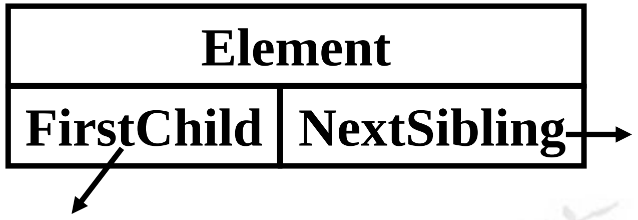
(A (B, C, D))

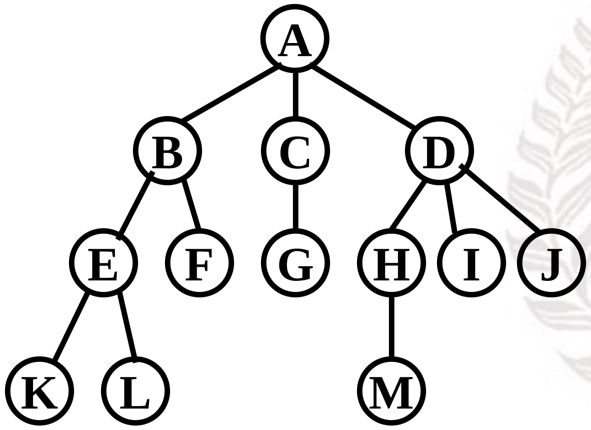
(A (B (E, F), C (G), D (H, I, J)))

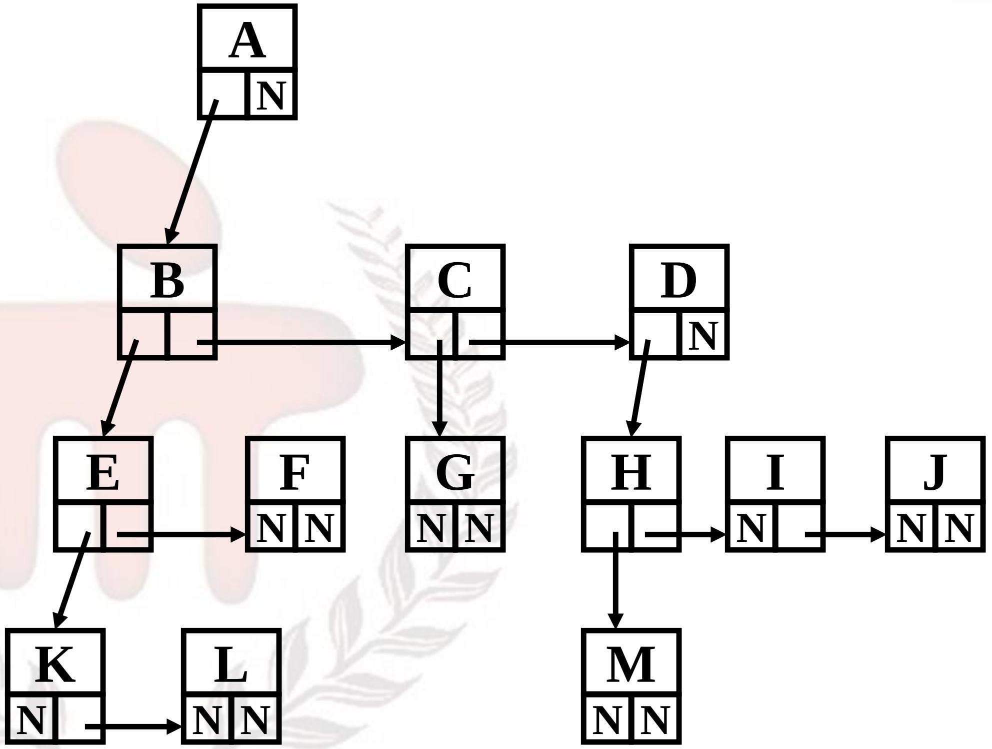
(A (B (E (K, L), F), C (G), D (H (M), I, J)))



❖ FirstChild-NextSibling Representation (Left Child Right Sibling)





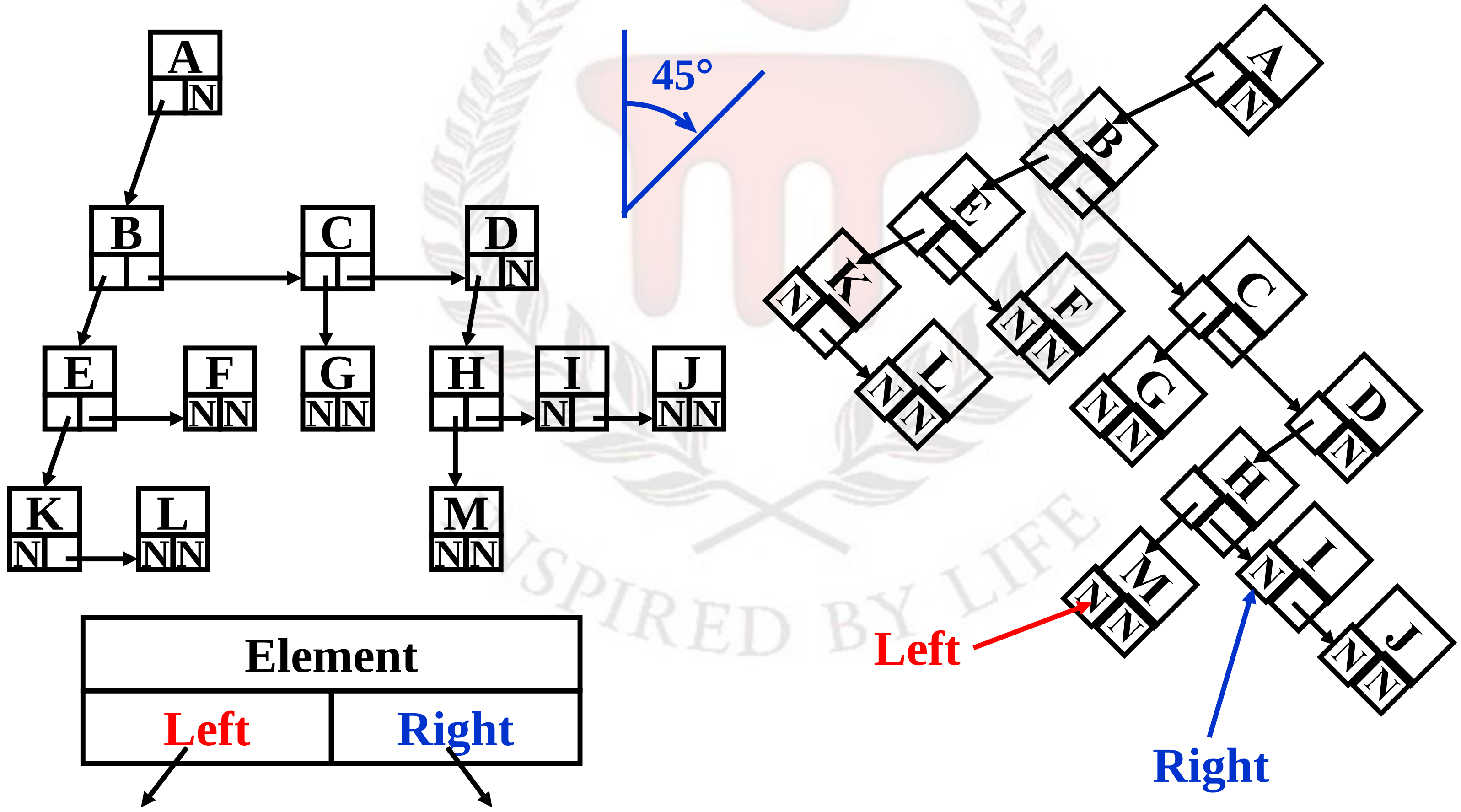


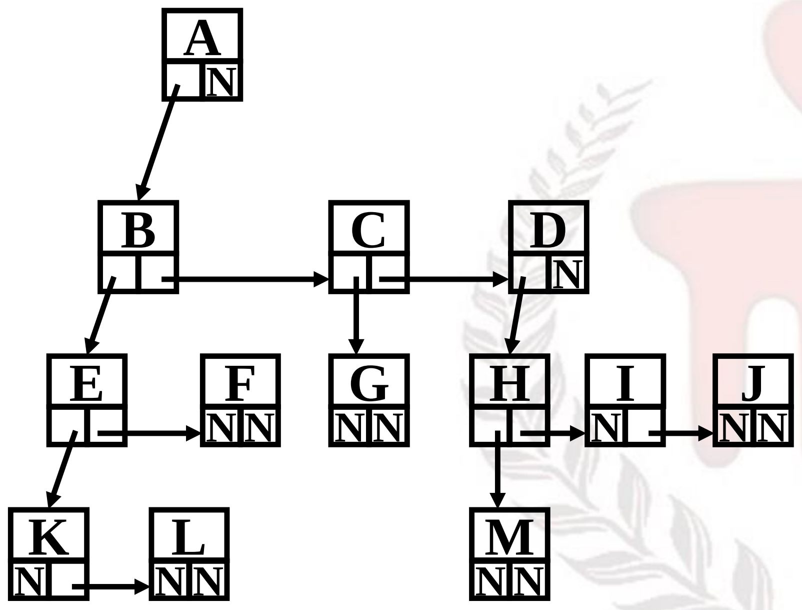
Note: The representation is not unique since the children in a tree can be of any order.

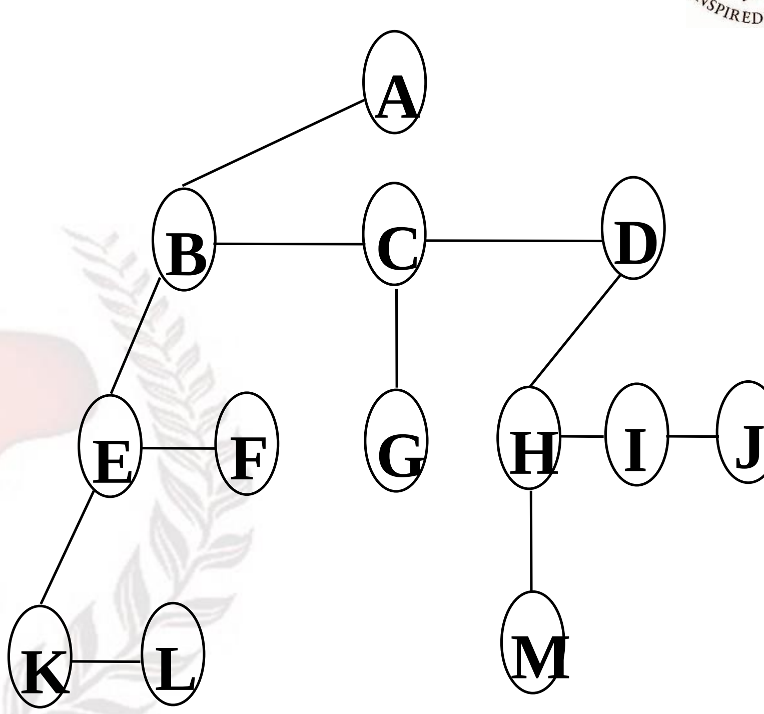
2 Binary Trees

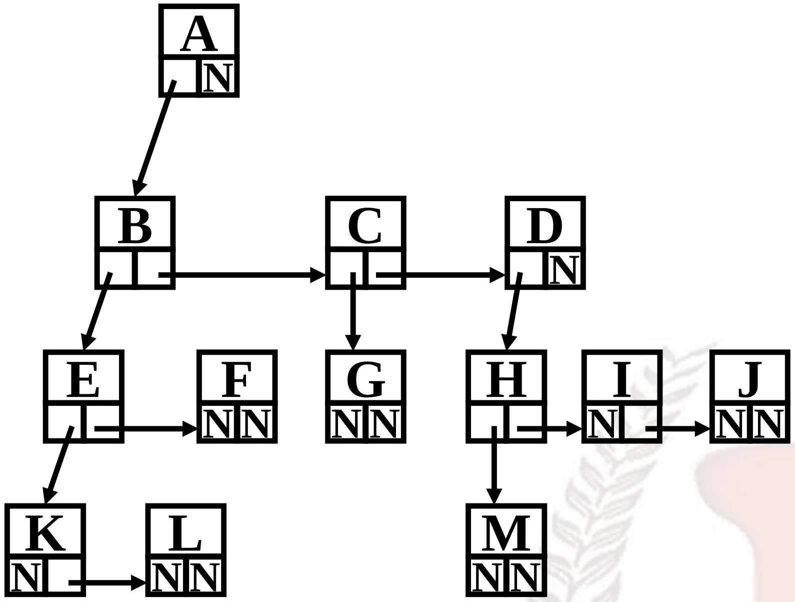
【Definition】A binary tree is a tree in which no node can have more than two children.

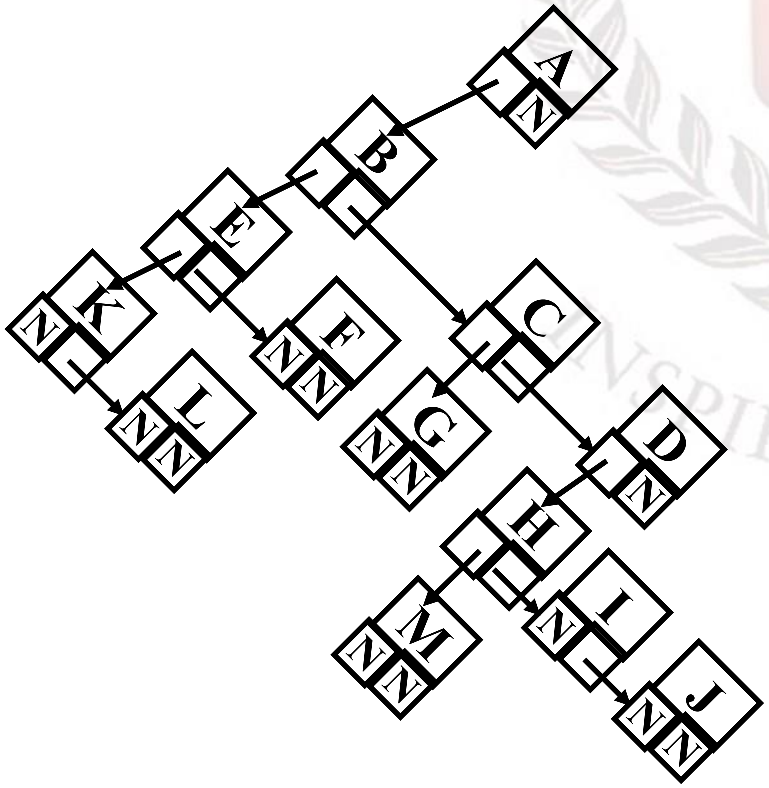
Rotate the FirstChild-NextSibling tree clockwise by 45°.

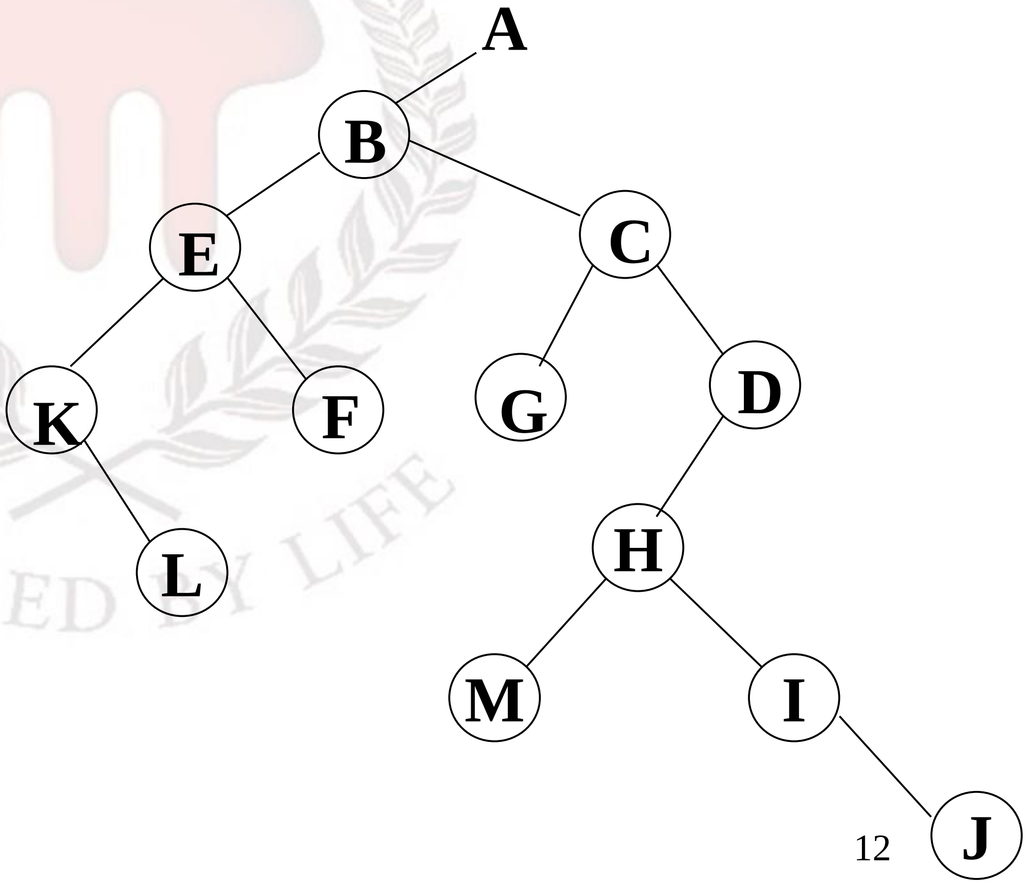



A
N
B
C
D
N
E
F
NN
G
NN
H
I
N
J
NN
K
N
L
NN
M
NN


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


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


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


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

Maximum Number of Nodes in BT

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

Prove by induction.

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

Relations between Number of Leaf Nodes and Nodes of Degree 2

For any nonempty binary tree, T, if $n0$ is the number of leaf nodes and $n2$ the number of nodes of degree 2, then $n0=n2+1$

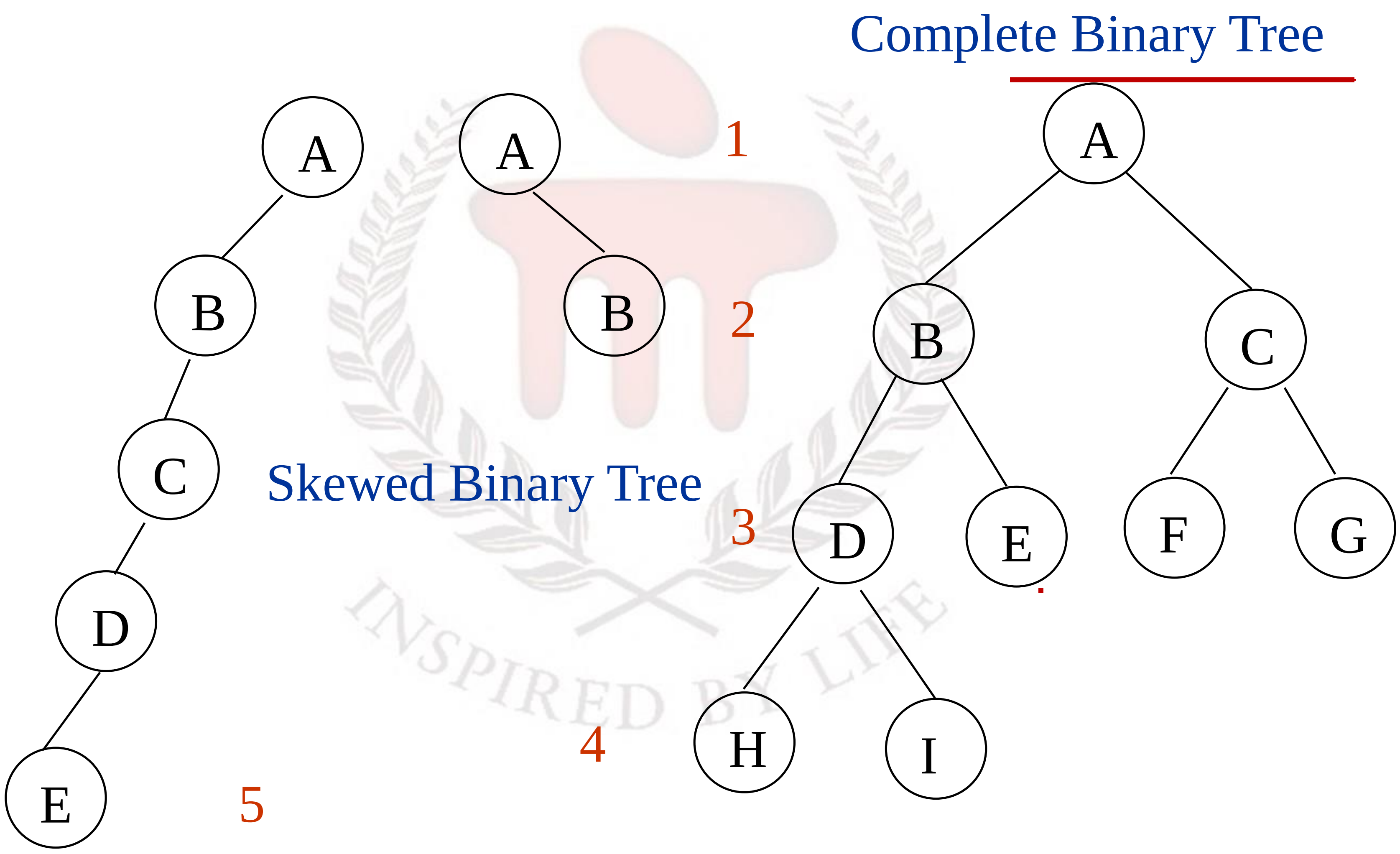
proof:

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

Let n_0, n_1, n_2 represent the nodes with no children, single child, and two children respectively.

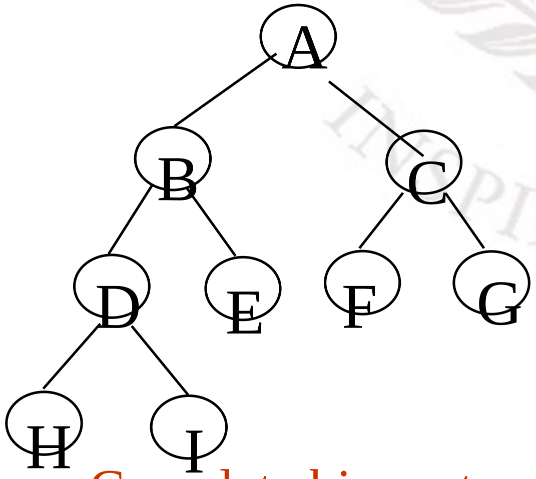
$n= n_0+n_1+n_2$, $B+1=n$, $B=n_1+2n_2$ ==> $n_1+2n_2+1= n$,
$n_1+2n_2+1= n_0+n_1+n_2$ ==> $n_0=n_2+1$

Samples of Trees

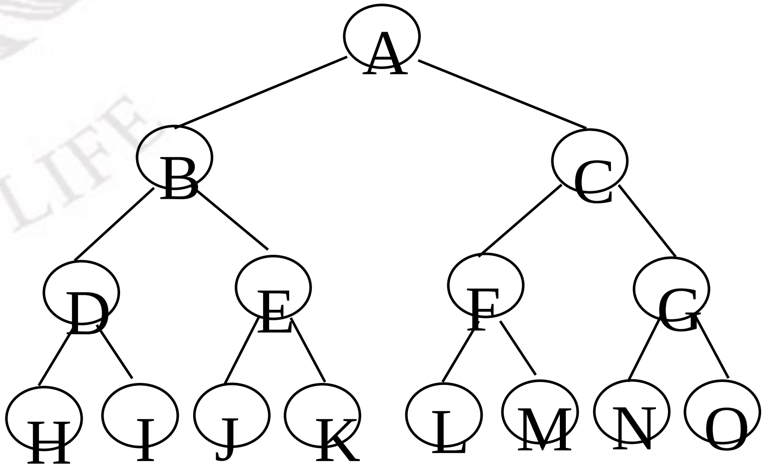


Full BT VS Complete BT

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



Complete binary tree

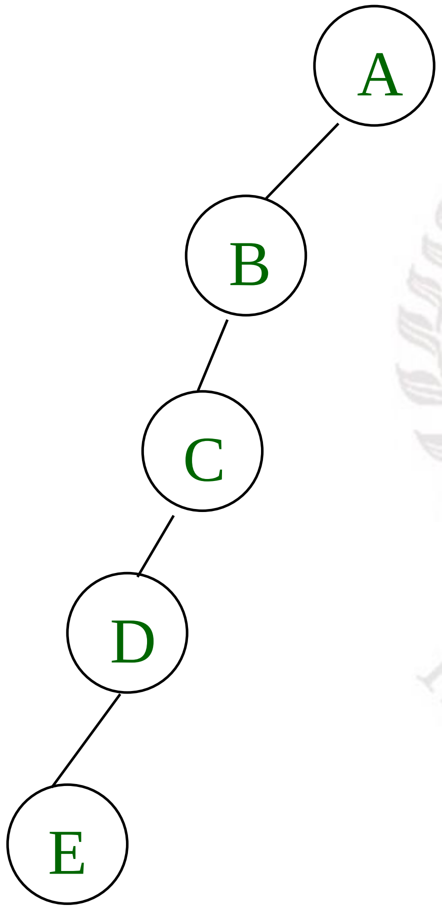


Full binary tree of depth 4

Binary Tree Representations

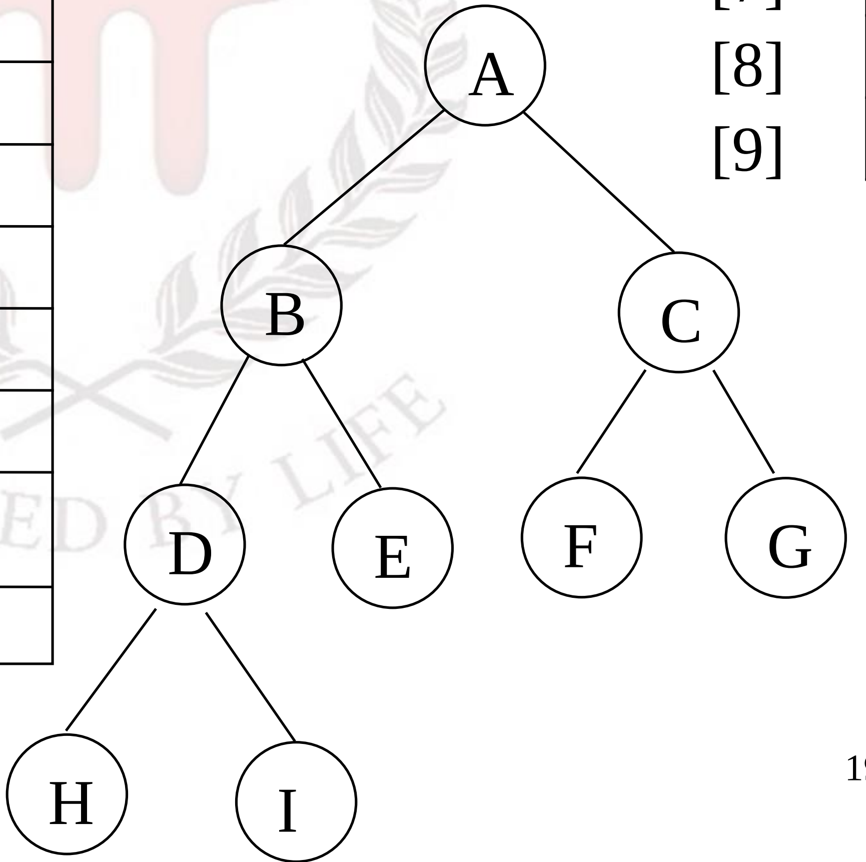
- If a complete binary tree with *n* nodes is represented sequentially, then for any node with index *i*, 1<=*i*<=*n*, we have:
  - *parent*(*i*) is at *i*/2 if *i*!=1. If *i*=1, *i* is at the root and has no parent.
  - *left_child*(*i*) is at 2*i* if 2*i*<=*n*. If 2*i*>n, then *i* has no left child.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

Sequential(Array) Representation



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

(1) waste space
(2) insertion/deletion problem

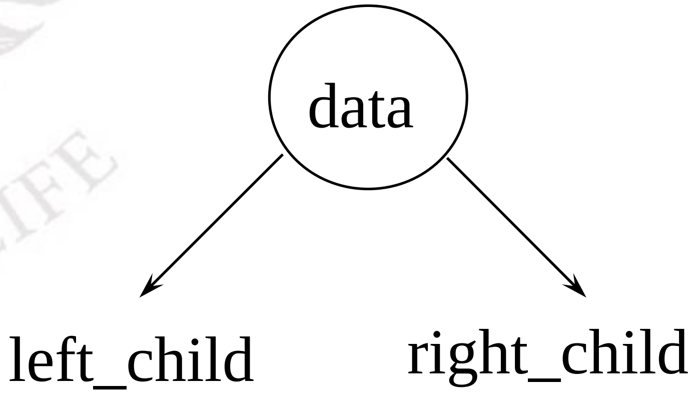


[1]	A
[2]	B
[3]	C
[4]	D
[5]	E
[6]	F
[7]	G
[8]	H
[9]	I

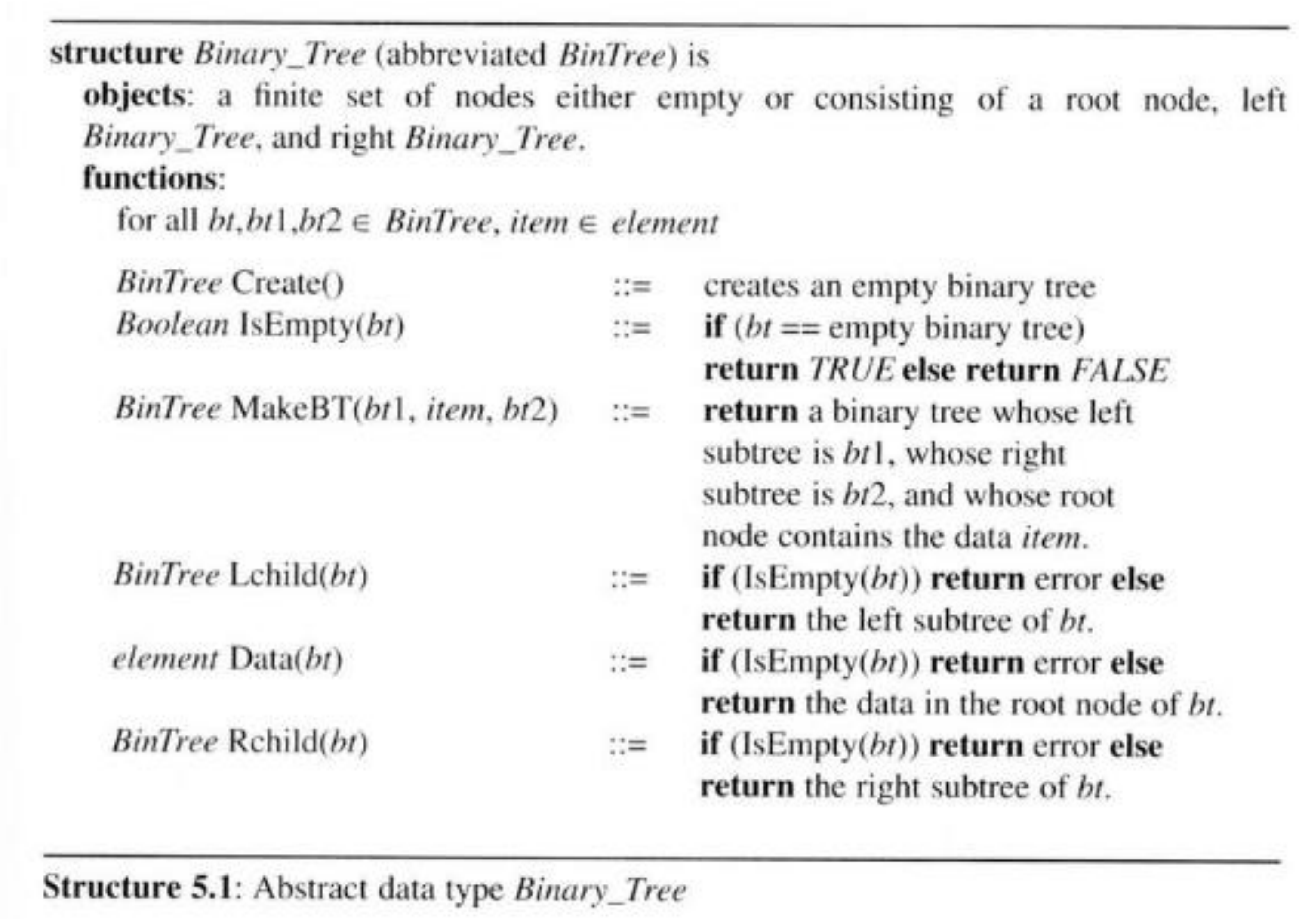
Linked Representation

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

left_child	data	right_child



ADT Binary_Tree



structure *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*,*bt*1,*bt*2 ∈ *BinTree*, *item* ∈ *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*.

Structure 5.1: Abstract data type *Binary_Tree*

Binary Tree Traversals

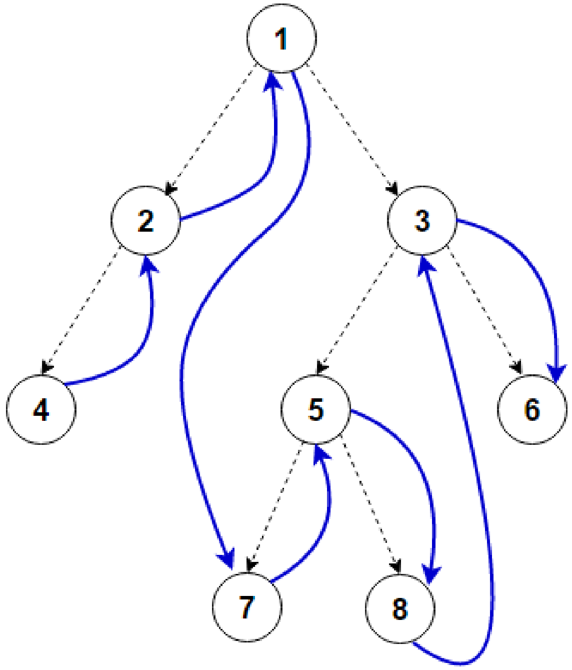
- Let L, N, and R stand for moving left, visiting the node, and moving right.
- There are six possible combinations of traversal
  - LNR, LRN, NLR, NRL, RNL, RLN
- Adopt convention that we traverse left before right, only 3 traversals remain
  - LNR, LRN, NLR
  - inorder, postorder, preorder

For traversing a (non-empty) binary tree in an inorder fashion, we must do these three things for every node n starting from the tree's root:

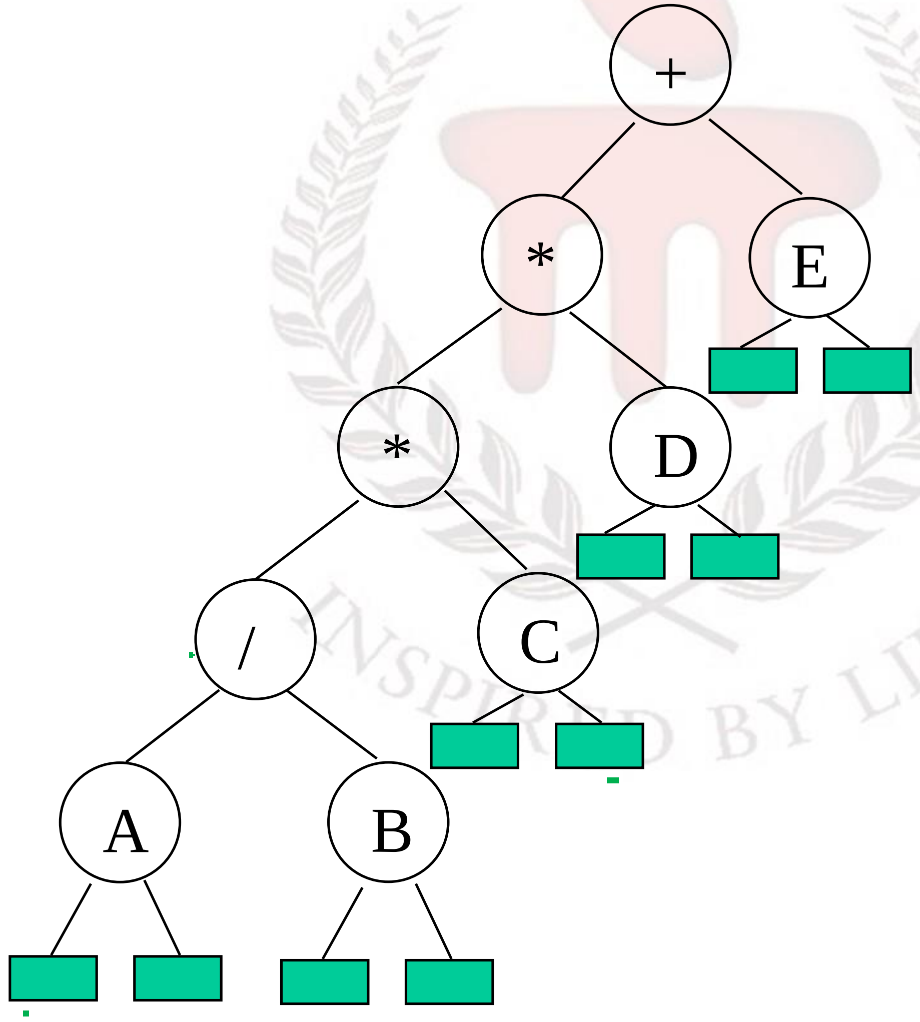
(L) Recursively traverse its left subtree. When this step is finished, we are back at n again.

(N) Process n itself.

(R) Recursively traverse its right subtree. When this step is finished, we are back at n again.



Arithmetic Expression Using BT



inorder traversal
A / B * C * D + E
infix expression
preorder traversal
+ * * / A B C D E
prefix expression
postorder traversal
A B / C * D * E +
postfix expression
level order traversal
+ * E * D / C A B

Inorder Traversal (recursive version)

```
void inorder(node *root)
/* inorder tree traversal */
{
     if (ptr) {
          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 +

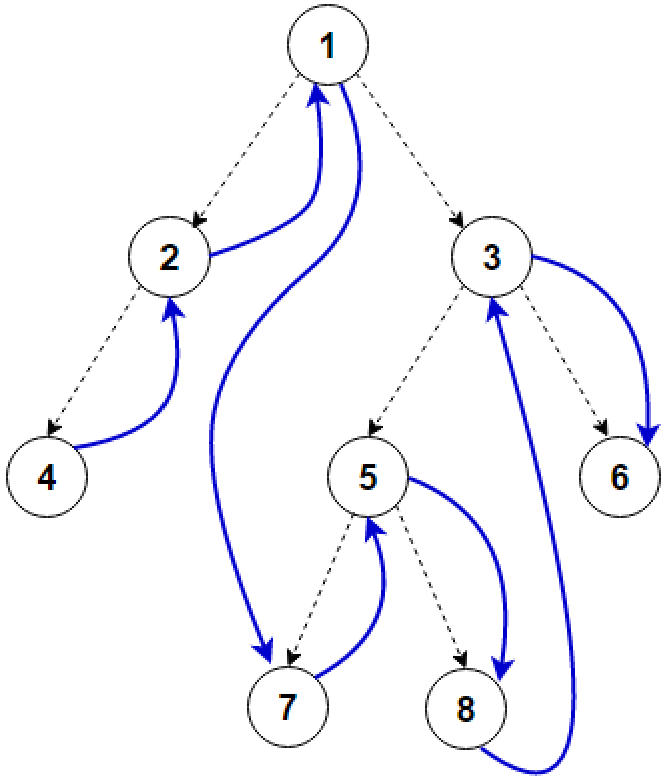
Iterative Inorder Traversal(using stack)

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 4th step.
6. If the right child is NULL and the stack is empty, stop the traversal.

iterativeInorder(node)

```
  s —> empty stack
  while (not s.isEmpty() or node != null)
    if (node != null)
      s.push(node)
      node —> node.left
    else
      node —> s.pop()
      visit(node)
      node —> node.right
```



Inorder: 4, 2, 1, 7, 5, 8, 3, 6

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;
 }
}
```

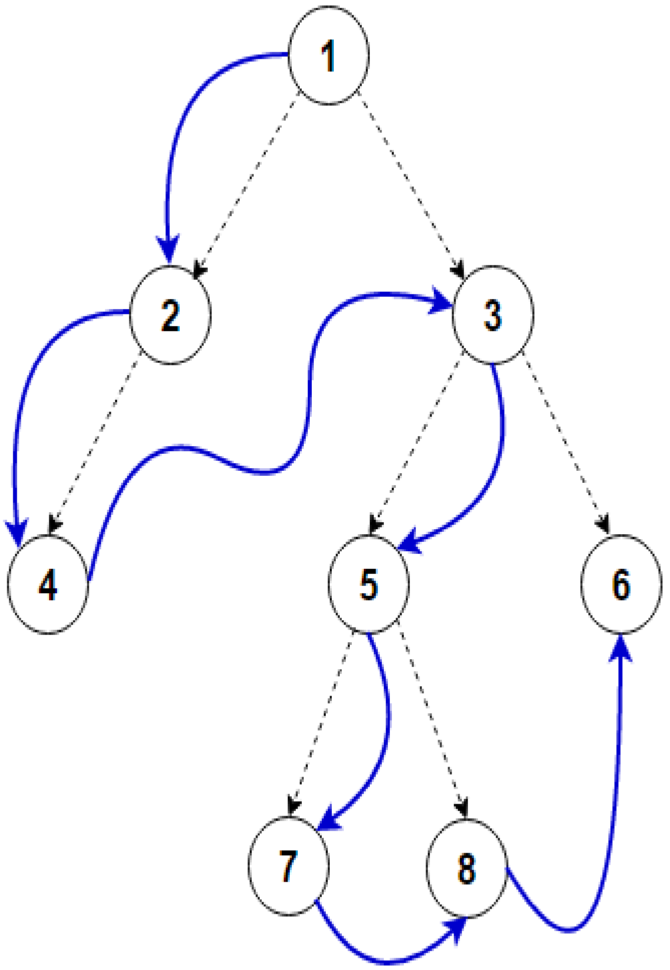
Iterative Preorder Traversal(using stack)

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.
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.

iterativePreorder(node)

```
if (node = null)
  return
s —> empty stack
s.push(node)
while (not s.isEmpty())
  node —> s.pop()
  visit(node)
  if (node.right != null)
    s.push(node.right)
  if (node.left != null)
    s.push(node.left)
```



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.

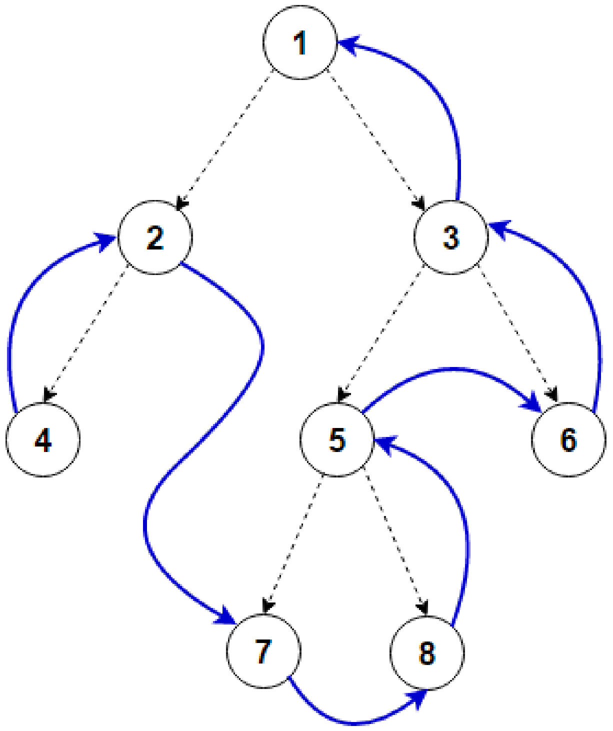
iterativePostorder(node)

```
s —> empty stack
t —> output stack
while (not s.isEmpty())
  node —> s.pop()
  t.push(node)

  if (node.left <> null)
    s.push(node.left)

  if (node.right <> null)
    s.push(node.right)

while (not t.isEmpty())
  node —> t.pop()
  visit(node)
```



Postorder: 4, 2, 7, 8, 5, 6, 3, 1

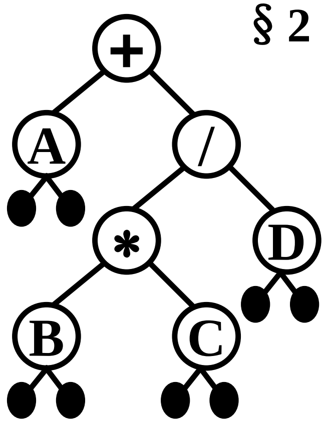
Level Order Traversal

(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〗 $(a + b) * (c * (d + e)) = a\ b + c\ d\ e + * *$