Trees

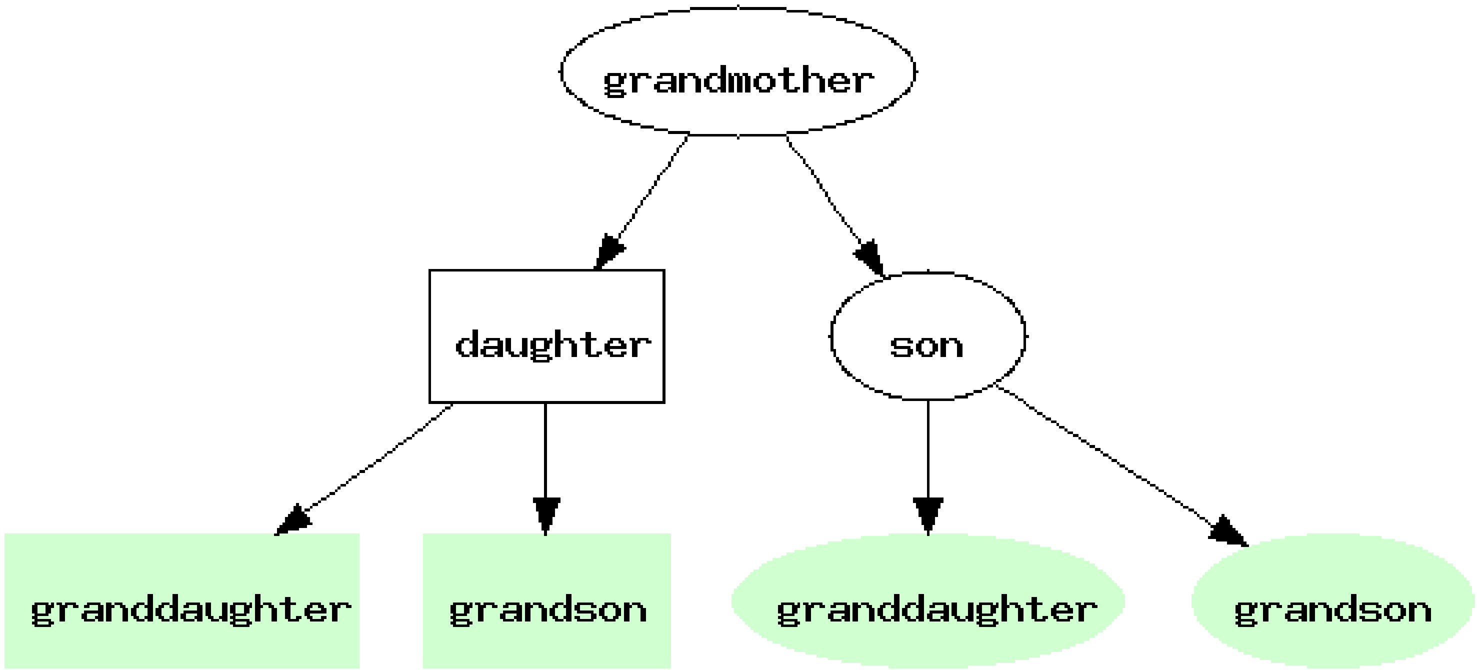
Trees

- Natural structures for representing certain kinds of hierarchical data.(How our files get saved under hierarchical directories)
- Allows us to associate a parent-child relationship between various pieces of data and allows arrange our records, data and files in a hierarchical fashion.
- Have many uses in computing. For example a *parse-tree* can represent the structure of an expression.
- Binary Search Trees help to order the elements in such a way that the searching takes less time as compared to other data structures.(speed advantage over other D.S)

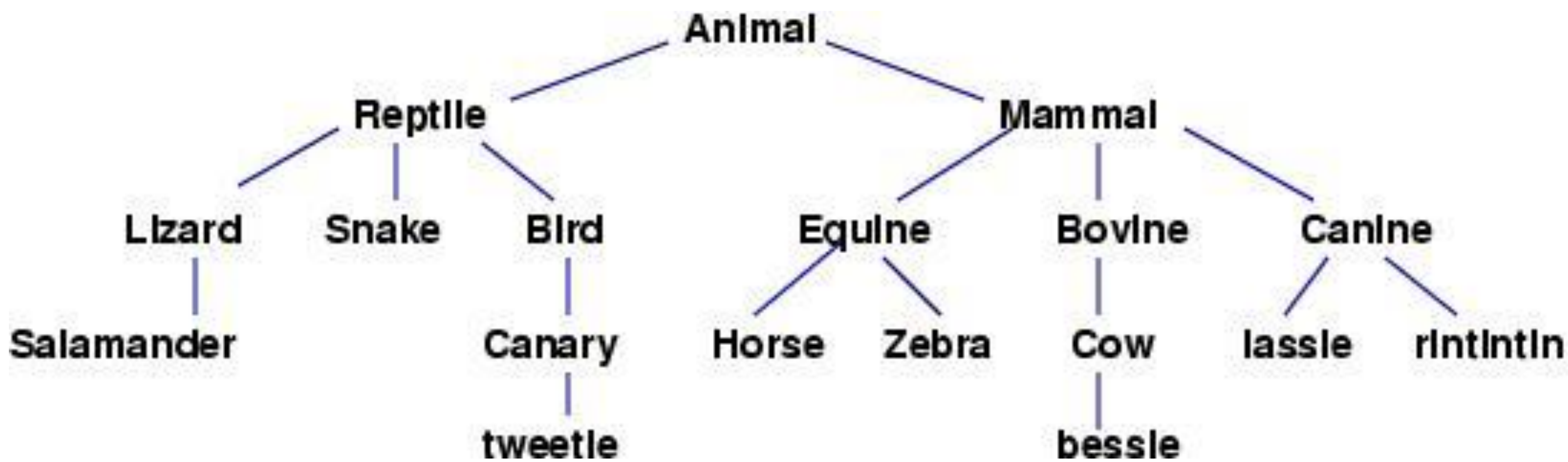
Trees

- Linked list is a linear D.S and for some problems it is not possible to maintain this linear ordering.
- Using non linear D.S such as trees and graphs more complex relations can be expressed.

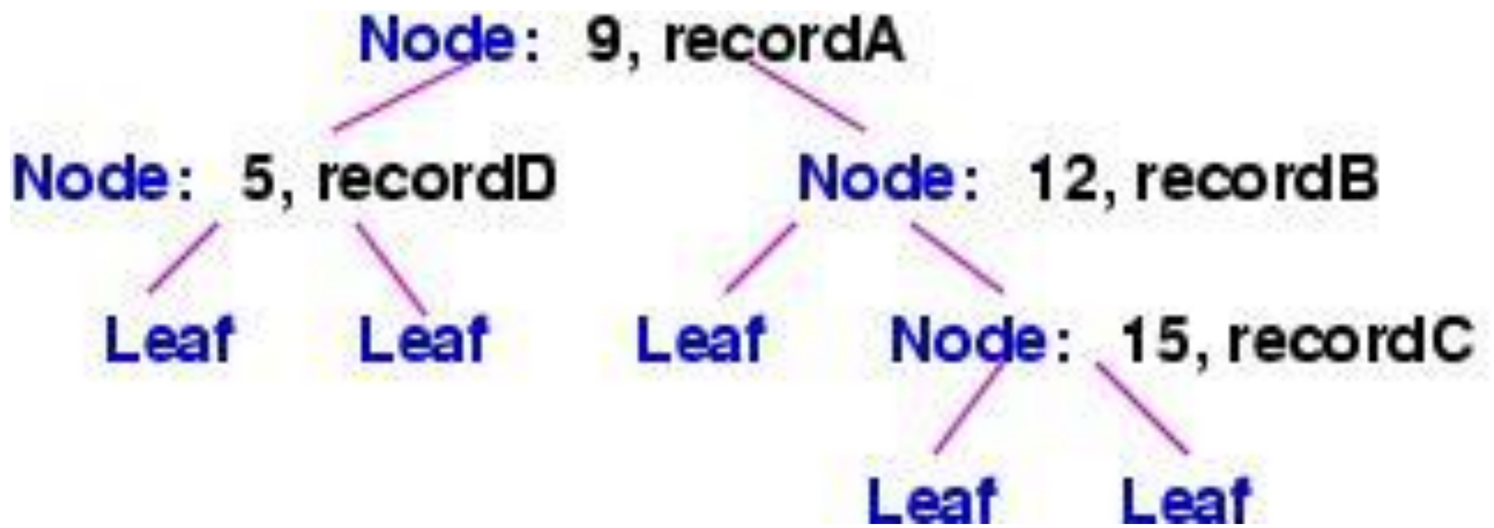
A Family Tree



tree of species, from zoology



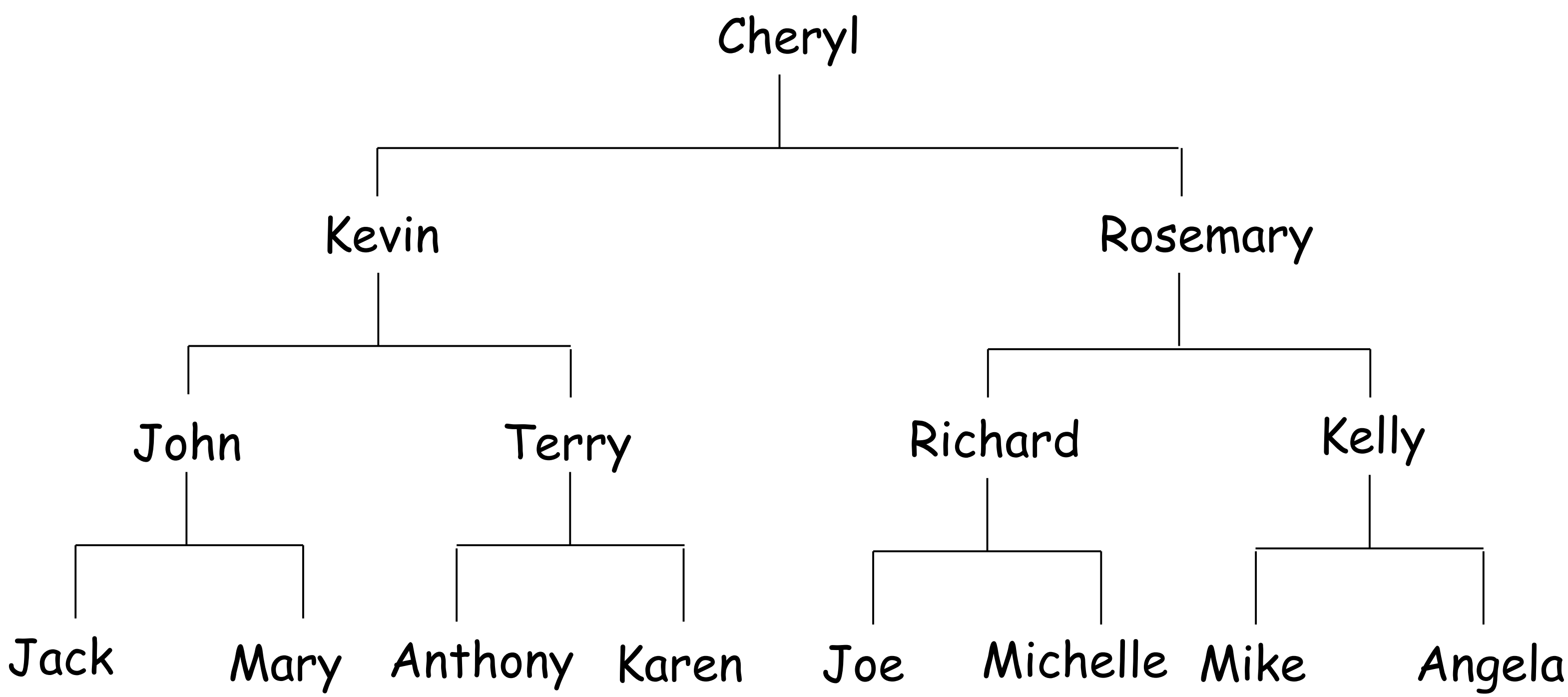
Ordered Tree or Search Tree



Parse Tree

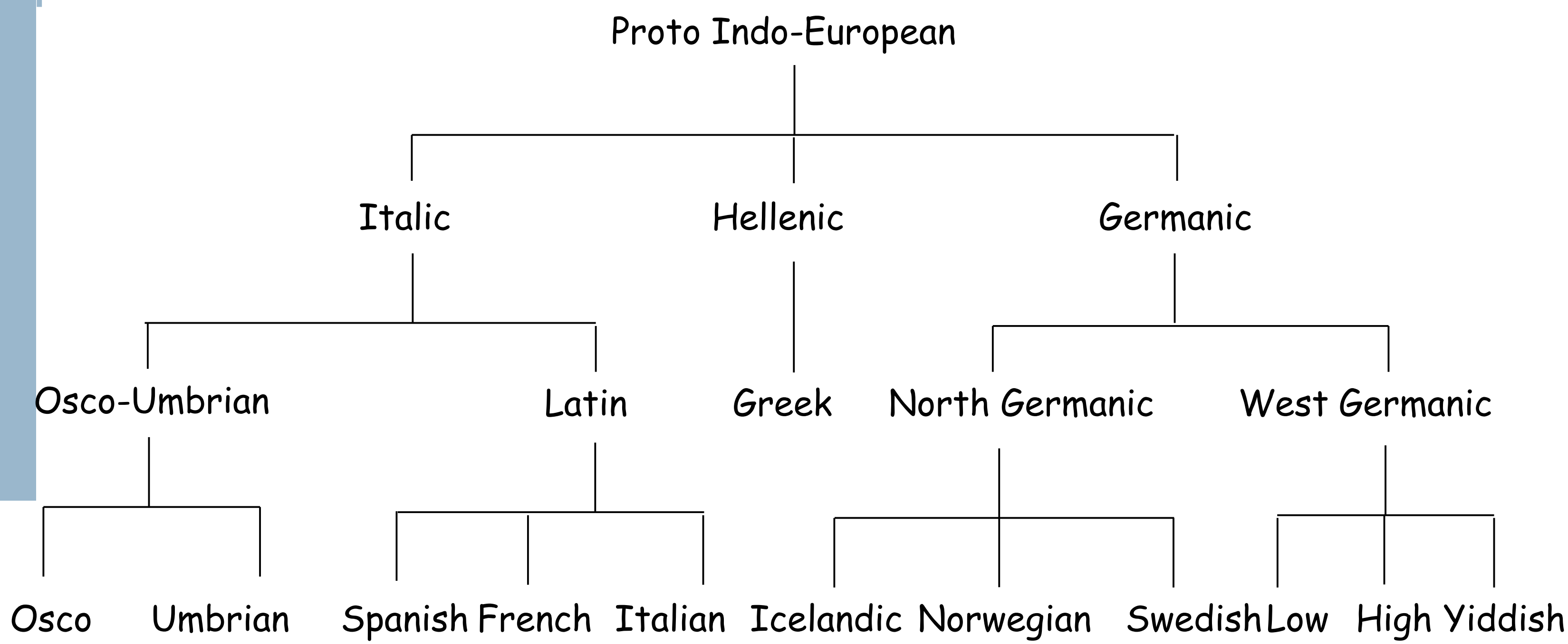
```
int x;
x = 3 + y;


              STATEMENT SEQUENCE
             /                  \
  DECLARATION                ASSIGNMENT
    /        \                /      \
TYPE   VARIABLE          VARIABLE    EXPRESSION
 |         |                 |         /      |    \
int        x                 x      NUMERAL + VARIABLE
                                       |           |
                                       3           y
```

Pedigree Genealogical Chart



Binary Tree

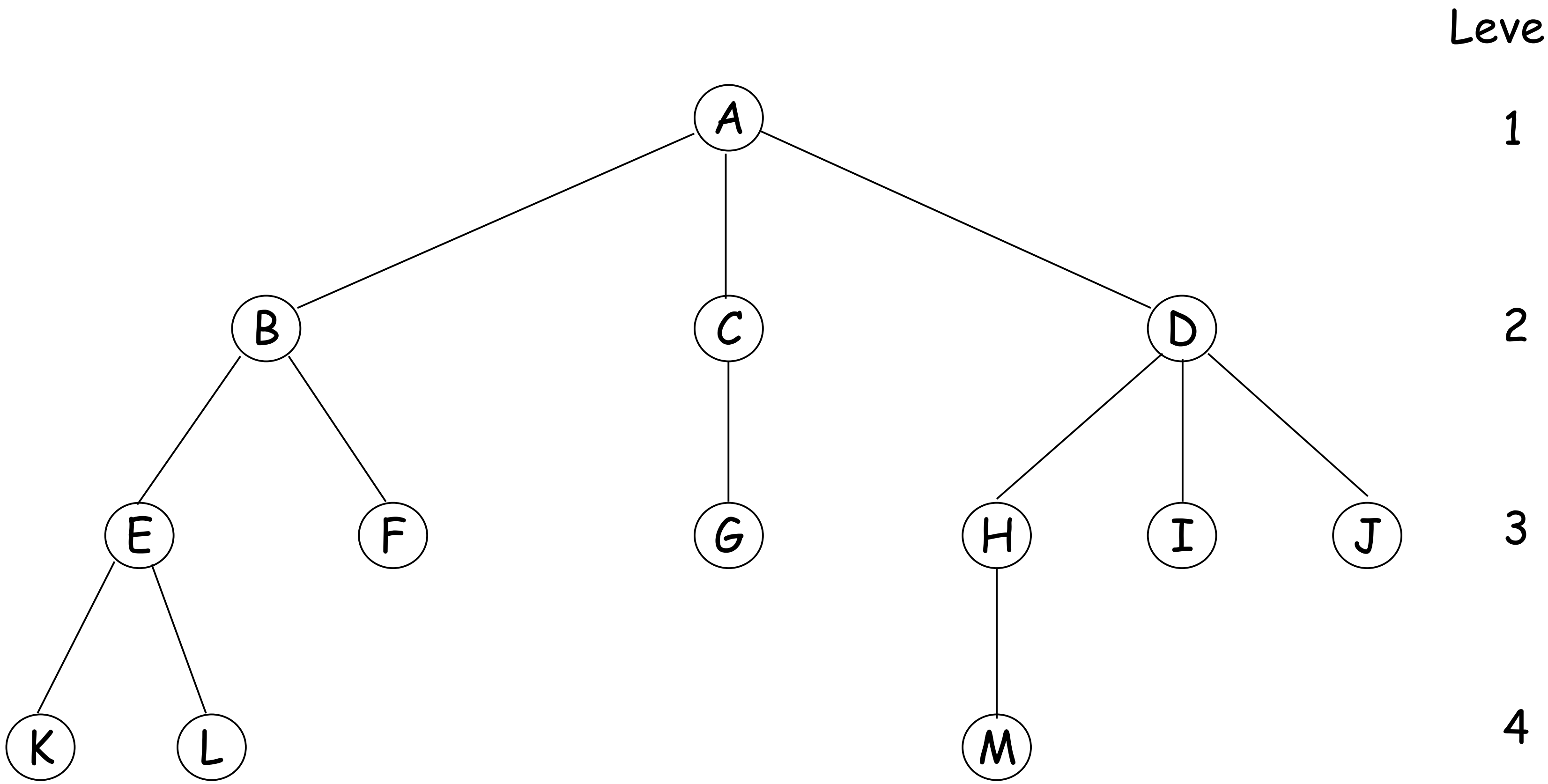
Lineal Genealogical Chart



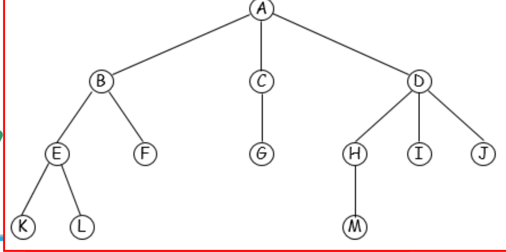
Trees

- Definition: A tree is a finite set of one or more nodes such that:
  - There is a specially designated node called the root.
  - 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.

A Sample Tree



Tree Terminology

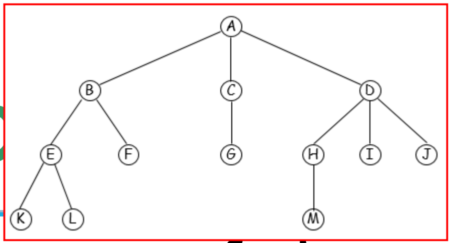


- Normally we draw a tree with the root at the top.
- A node stands for the item of information plus the branches to other nodes.
- The degree of a node is the number of subtrees of the node. [Degree of A=3, C=1, F=0]
- A node with degree zero is a leaf or terminal node.

  [K L F G M I J]
- A node that has subtrees is the parent of the roots of the subtrees, and the roots of the subtrees are the children of the node.

  [Children of B = E and F, parent of B is A]
- Children of the same parents are called siblings. [E and F]

Tree Terminology (Co



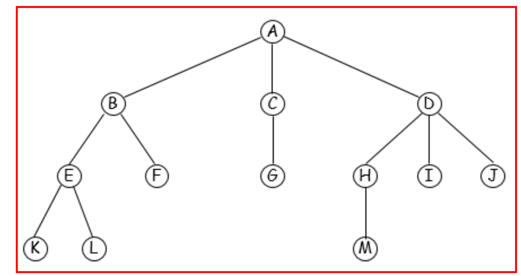
- The degree of a tree is the maximum degree of the nodes in the tree. [Degree of above tree = 3]
- The ancestors of a node are all the nodes along the path from the root to the node.

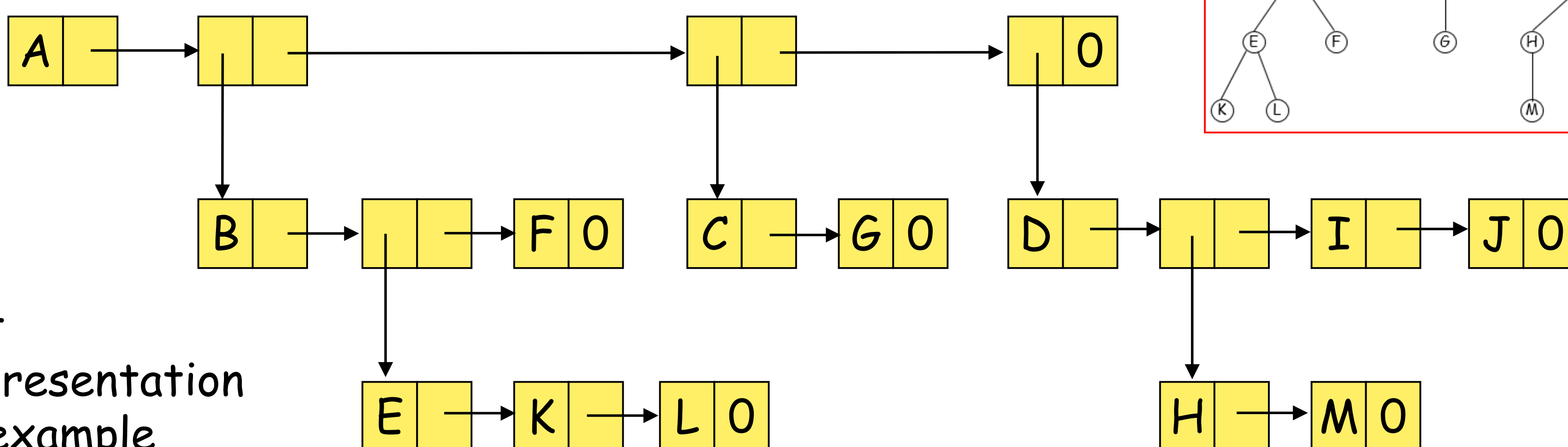
  ancestors of K = A, B and E
- The descendants of a node are all the nodes that are in its subtrees.
- Assume the root is at level 1, then the level of a node is the level of the node's parent plus one.
- The height or the depth of a tree is the maximum level of any node in the tree. [depth of the ex tree = 4]

List Representation of Trees

- Information in root node comes first, followed by a list of the subtrees of that node.
- The example tree could be written as

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





List Representation of example tree

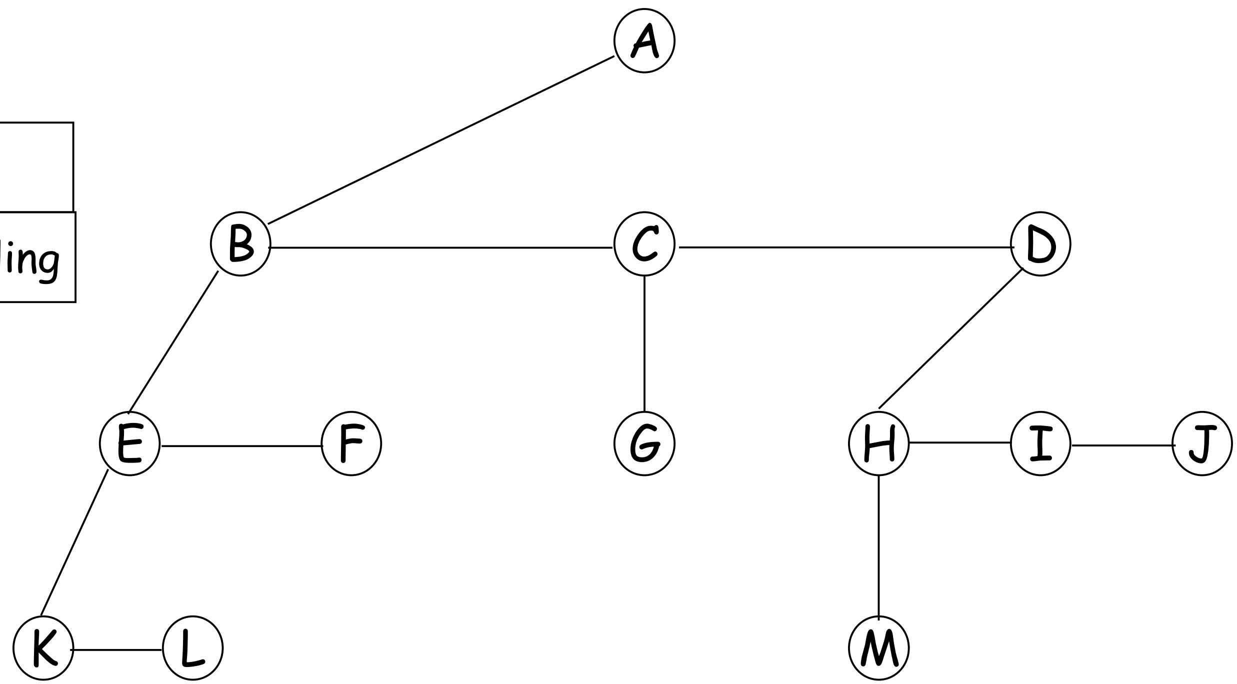
Data	Child1	Chid2	..	Childk

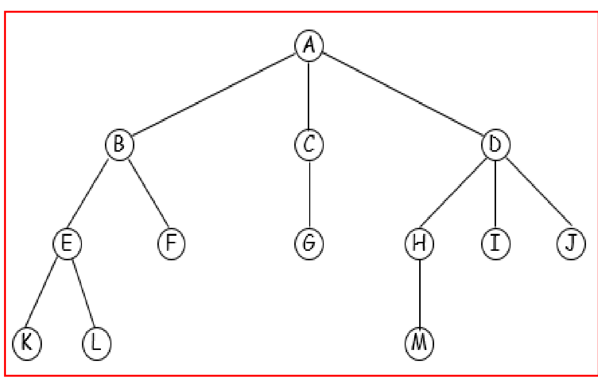
Possible Node Structure for a tree

Representation of Trees

- Left Child-Right Sibling Representation
  - Each node has two links (or pointers).
  - One leftmost child and one closest right sibling.

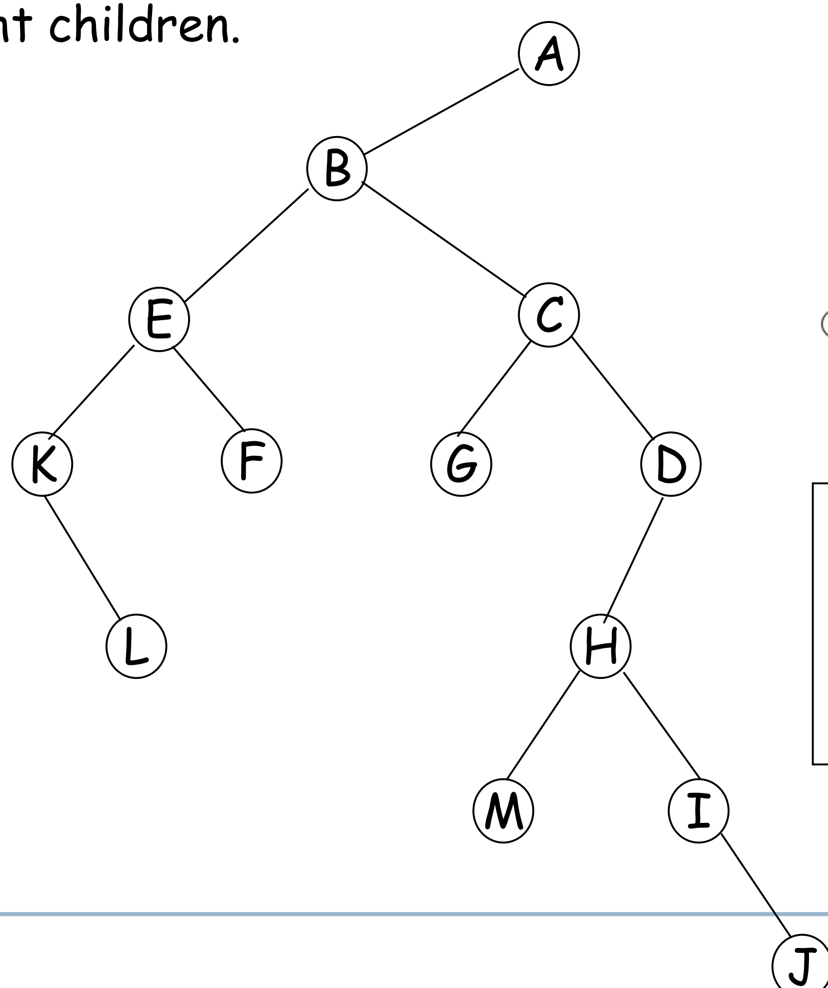
data	
left child	right sibling

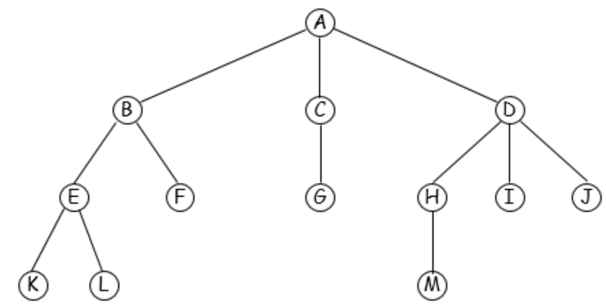




Degree Two Tree Representation

Rotate the right sibling pointers in a left child right sibling tree clockwise by 45 degrees. We refer to the two children of a node as left and right children.

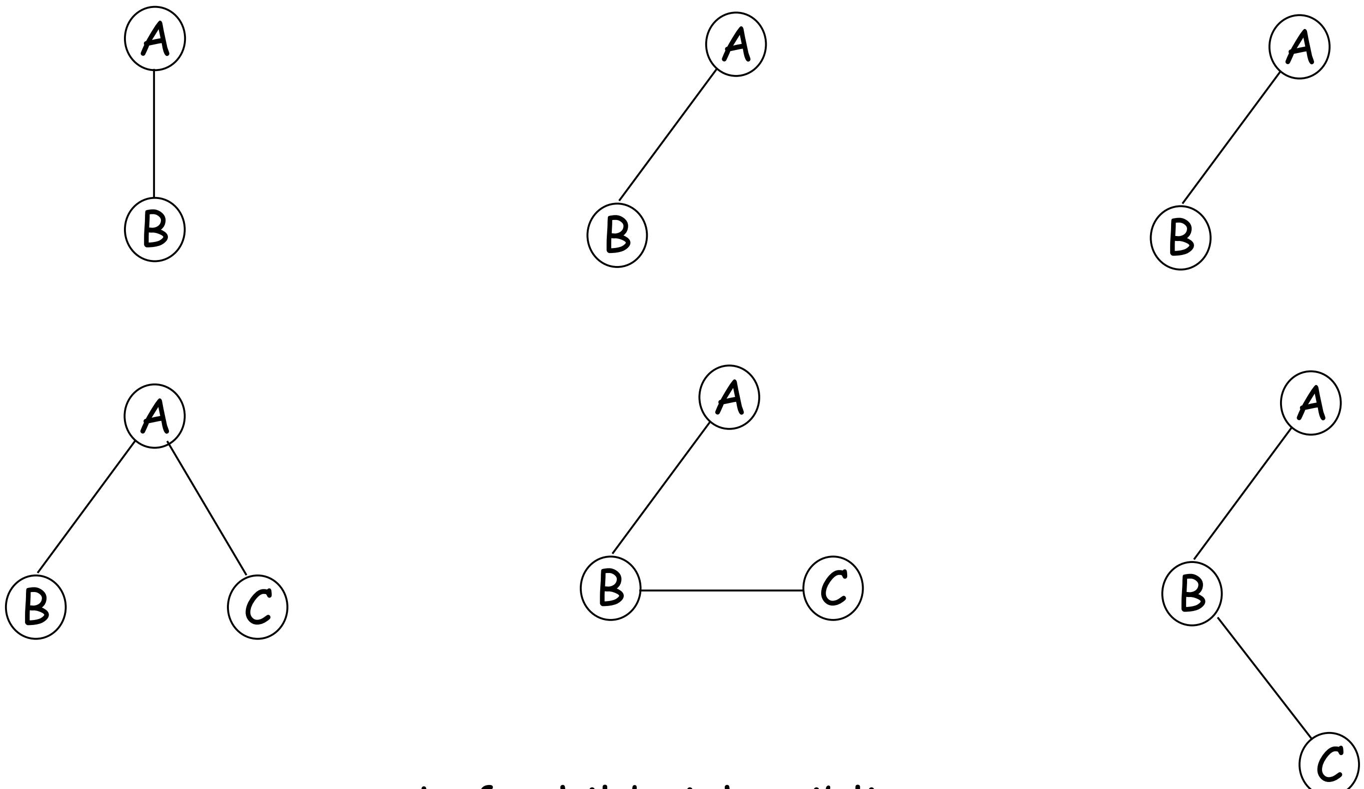




Left Child – Right Child trees are also know as

Binary Trees

Tree Representations



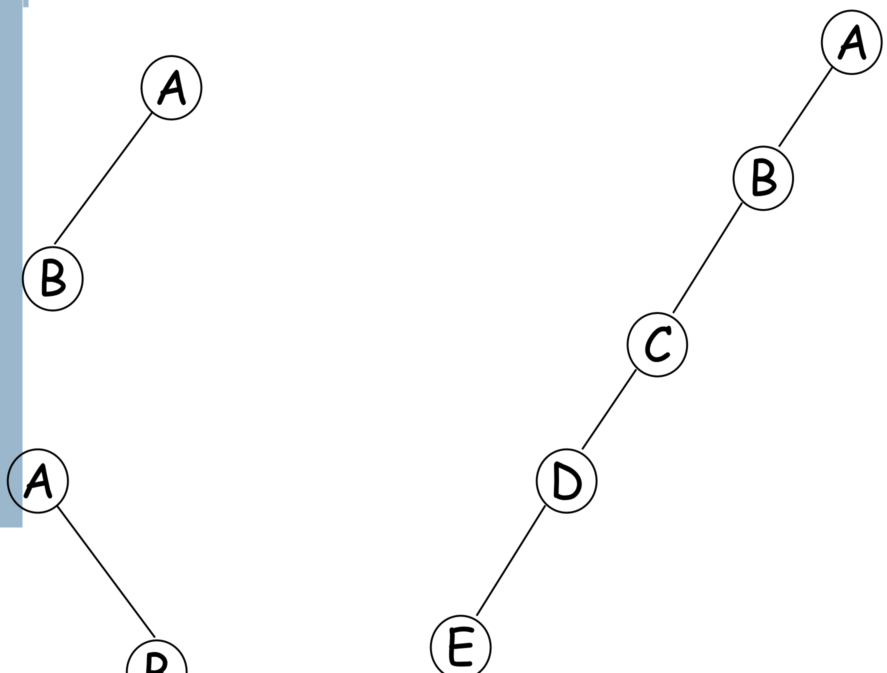
Left child-right sibling

Binary tree

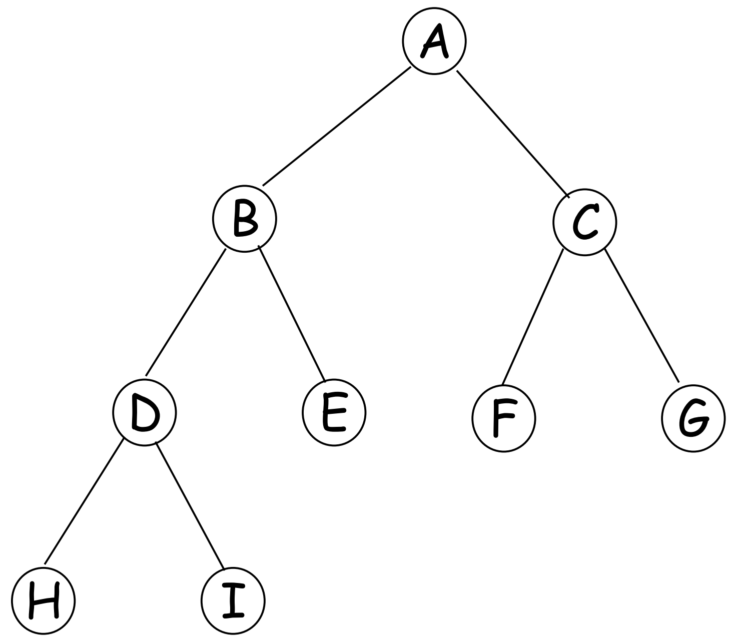
Binary Tree

- 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.
- The degree of any given node in a binary tree must not exceed two.
- Binary tree distinguishes between the order of the children while in a tree we do not.

Binary Tree Examples



Skewed Binary Tree



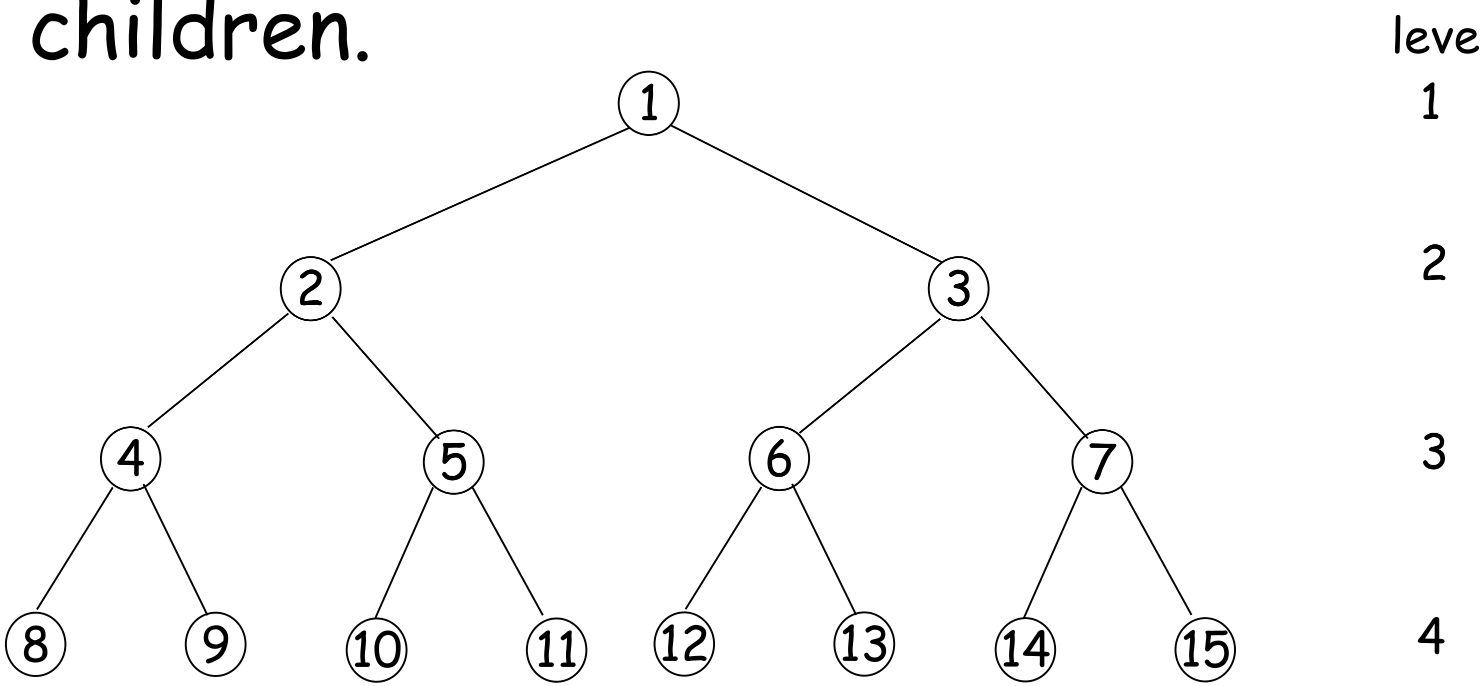
Complete Binary Tree

The 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 2^{i-1}, $i \geq 1$.
  2) The maximum number of nodes in a binary tree of depth k is $2^k - 1$, $k \geq 1$.
- **Lemma 5.3** [Relation between number of leaf nodes and nodes of degree 2]: For any non-empty binary tree, T, if n_0 is the number of leaf nodes and n_2 the number of nodes of degree 2, then $n_0 = n_2 + 1$.

Full binary Tree

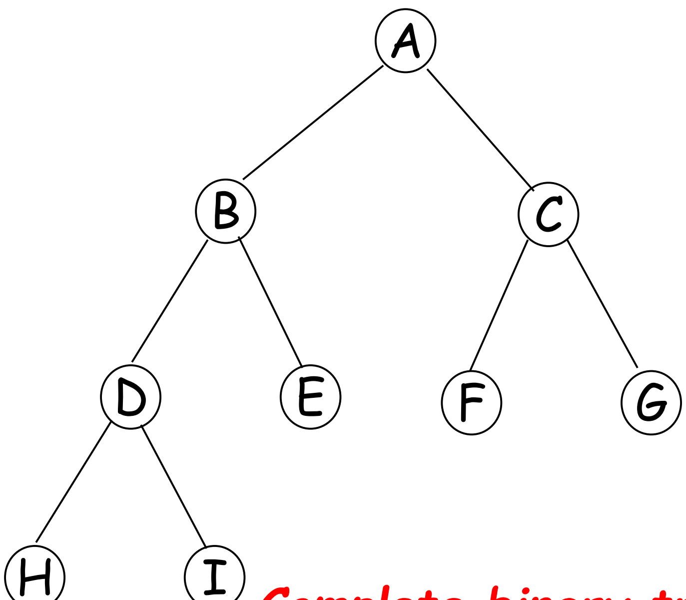
Definition: A **full binary tree** of depth k is a binary tree of depth k having $2^k - 1$ nodes, $k \geq 0$ (i.e having the maximum number of nodes). In this every node other than the leaves has two children.



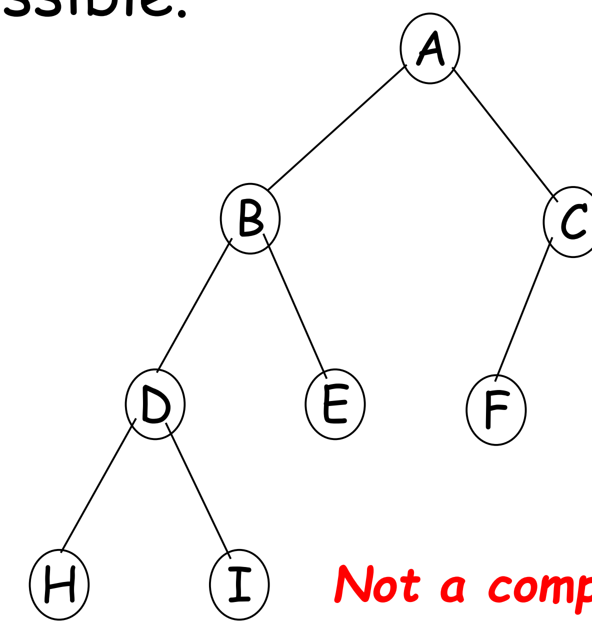
Full Binary Tree of depth 4 with sequential node numbers

Complete binary tree

- **Definition**: 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. In a complete binary tree, every level, except possibly the last, is completely filled, and all nodes are as far left as possible.



Complete binary tree



Not a complete binary tree

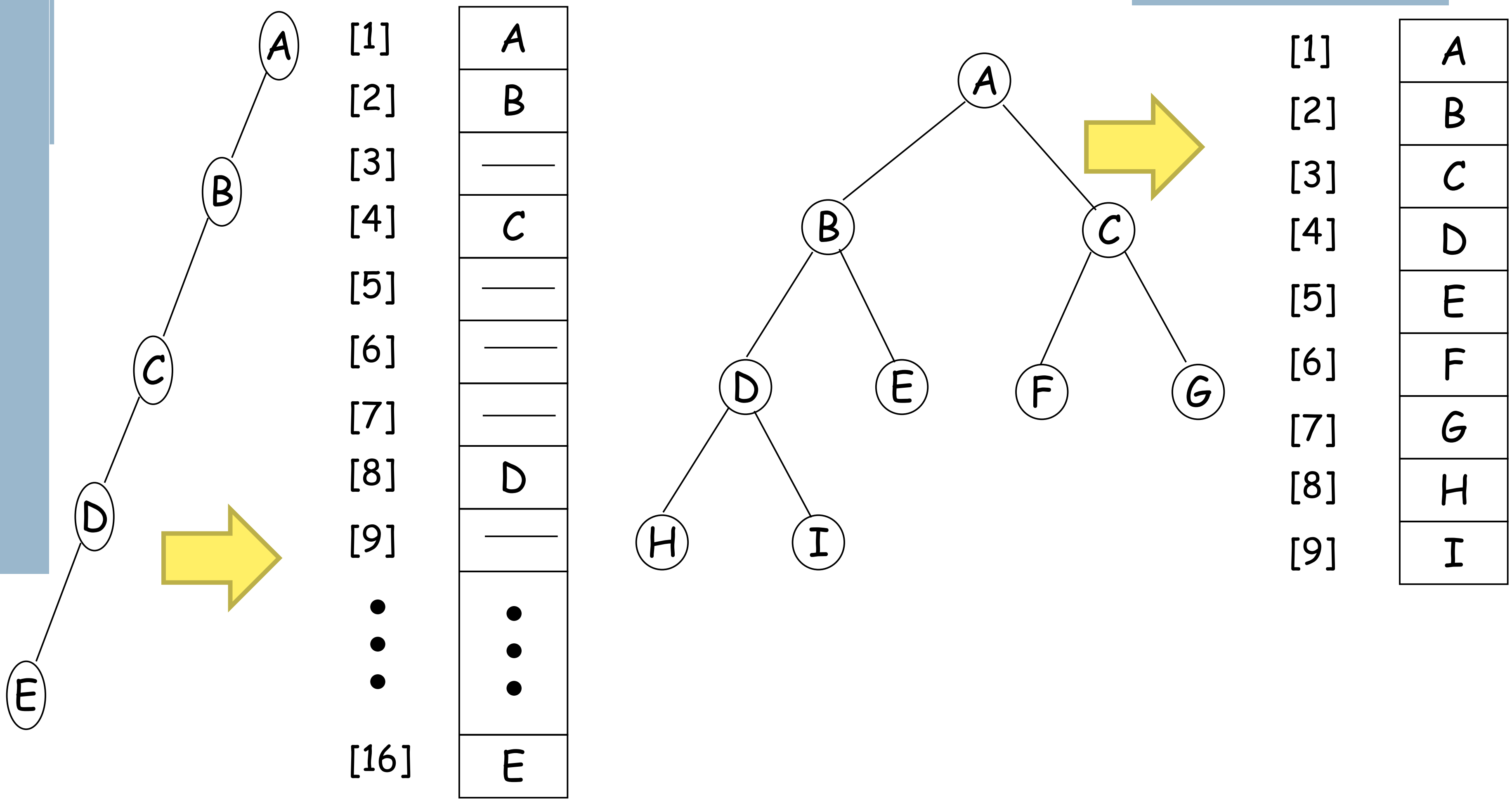
Storage representation of binary trees:

- Trees can be represented using
  - Linear/Sequential (Array) Representation
  - Linked Representation

Array Representation of A Binary Tree

- 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:
  - parent(i) is at $\lfloor i/2 \rfloor$ if $i \neq 1$. If $i = 1$, i is at the root and has no parent.
  - left_child(i) is at $2i$ if $2i \le n$. If 2i > n, then i has no left child.
  - right_child(i) is at $2i + 1$ if $2i + 1 \le n$. If $2i + 1 > n$, then i has no right child.
- Position zero of the array is not used.

Array Representation of Binary Trees



Advantages and disadvantages of Array representation

Advantages:

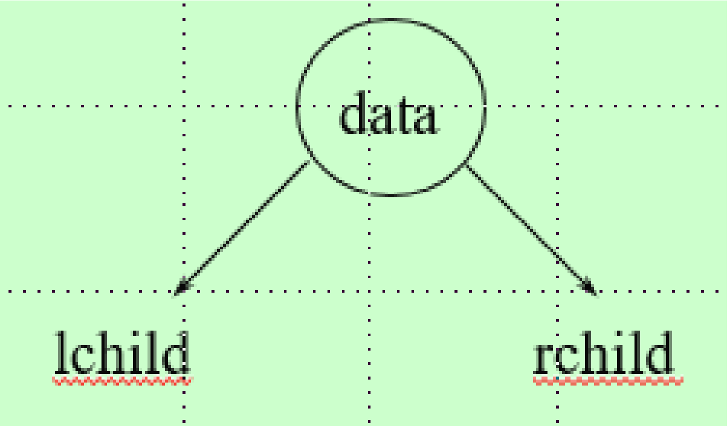
1. This representation is very easy to understand.
2. This is the best representation for full and complete binary tree representation.
3. Programming is very easy.
4. It is very easy to move from a child to its parents and vice versa.

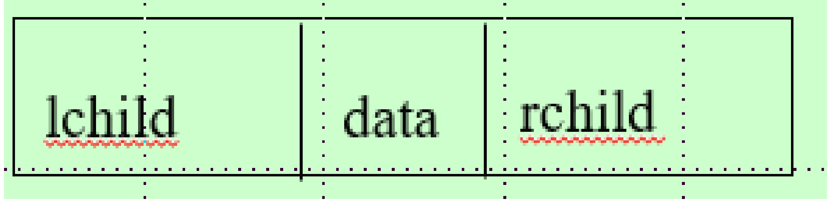
Disadvantages:

1. Lot of memory area wasted.
2. Insertion and deletion of nodes needs lot of data movement.
3. This is not suited for trees other than full and complete tree.

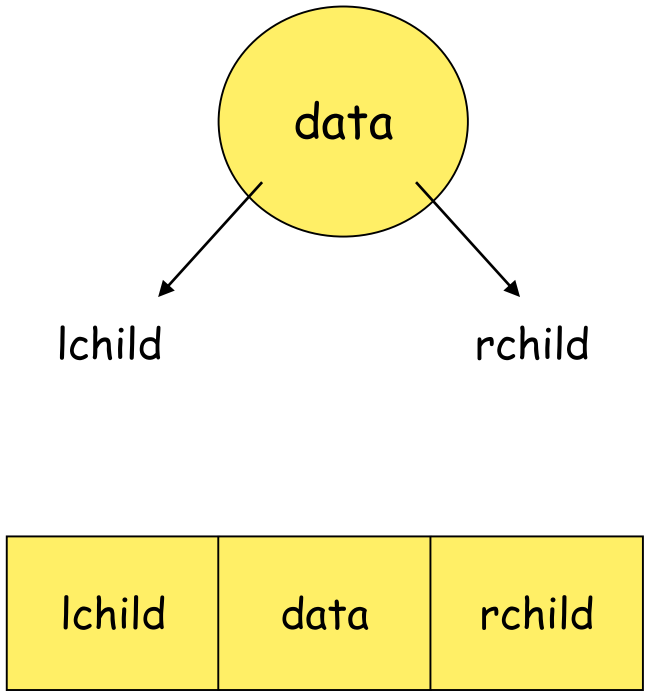
Linked Representation

```
typedef struct node *Nodeptr;

struct node{
    int data;
    Nodeptr rchild;
    Nodeptr lchild;
};
```

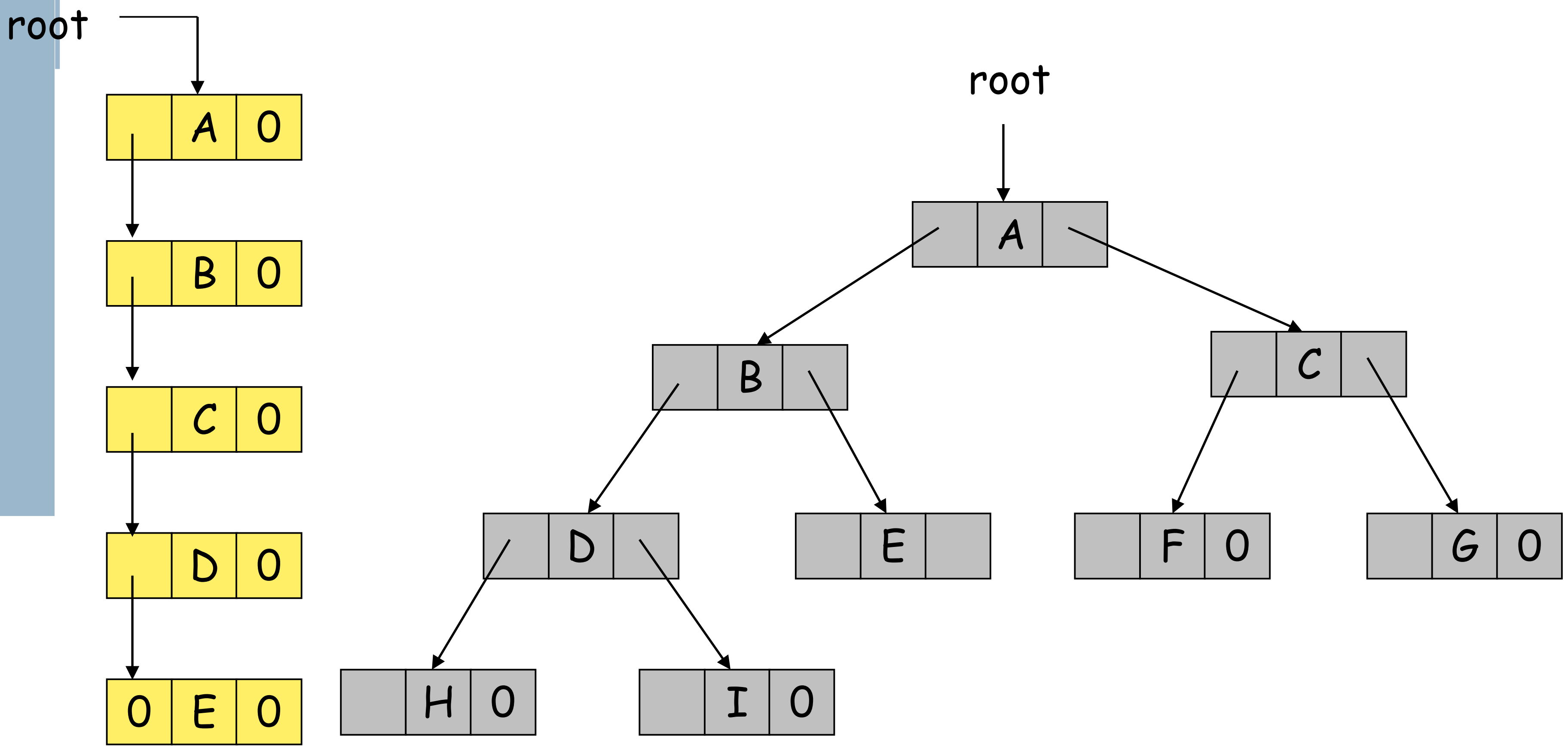




Node Representation



Linked List Representation For The Binary Trees



Advantages and disadvantages of linked representation

Advantages

1. A particular node can be placed at any location in the memory.
2. Insertions and deletions can be made directly without data movements.
3. It is best for any type of trees.
4. It is flexible because the system take care of allocating and freeing of nodes.

Disadvantage

1. It is difficult to understand.
2. Additional memory is needed for storing pointers
3. Accessing a particular node is not easy.

Recursive Function to create a binary tree

```
Nodeptr CreateBinaryTree(int item){
    int x;

    if (item!=-1)   { //until input is not equal to -1
            Nodeptr temp = getnode();
            temp->data = item;

            printf("Enter the lchild of %d :",item);
            scanf("%d",&x);
            temp->lchild = CreateBinaryTree(x);

            printf("Enter the rchild of %d :",item);
            scanf("%d",&x);
            temp->rchild = CreateBinaryTree(x);

            return temp;
    }
    return NULL;
}
```

```
int main()
{
   Nodeptr root = NULL;
    int item;

    printf("Creating the tree : \n");
    printf("Enter the root : ");
    scanf("%d",&item);

    root=CreateBinaryTree(item);
        ...
```

Tree Traversal

- Let L, V, and R stand for moving left, visiting the node, and moving right.
- There are six possible combinations of traversal for a binary tree
  - LVR, LRV, VLR, VRL, RVL, RLV
- Adopt convention that we traverse left before right, only 3 traversals remain
  - LVR, LRV, VLR
  - →inorder, postorder, preorder
- When implementing the traversal, a recursion is perfect for the task.

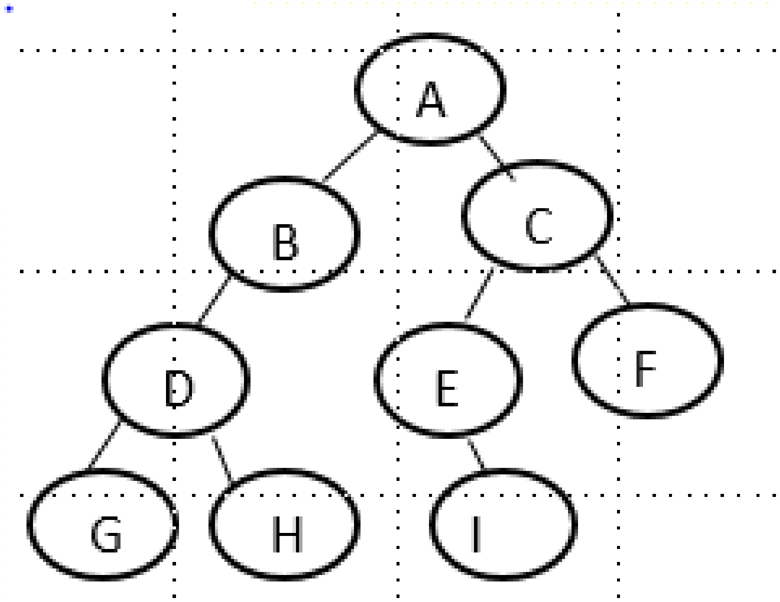
Tree Traversal

Inorder traversal

- It can be recursively defined as follows.
  1. Traverse the left subtree in inorder.
  2. Process the root node.
  3. Traverse the right subtree in inorder.

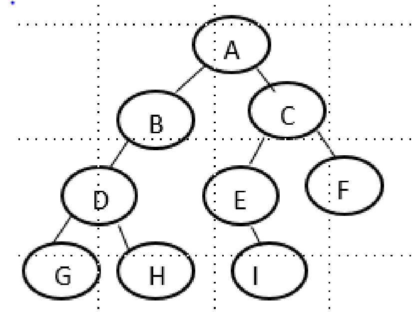
Inorder Traversal

- Move towards the left of the tree(till the leaf node), display that node and then move towards right and repeat the process.
- Since same process is repeated at every stage, recursion will serve the purpose.
- Example:

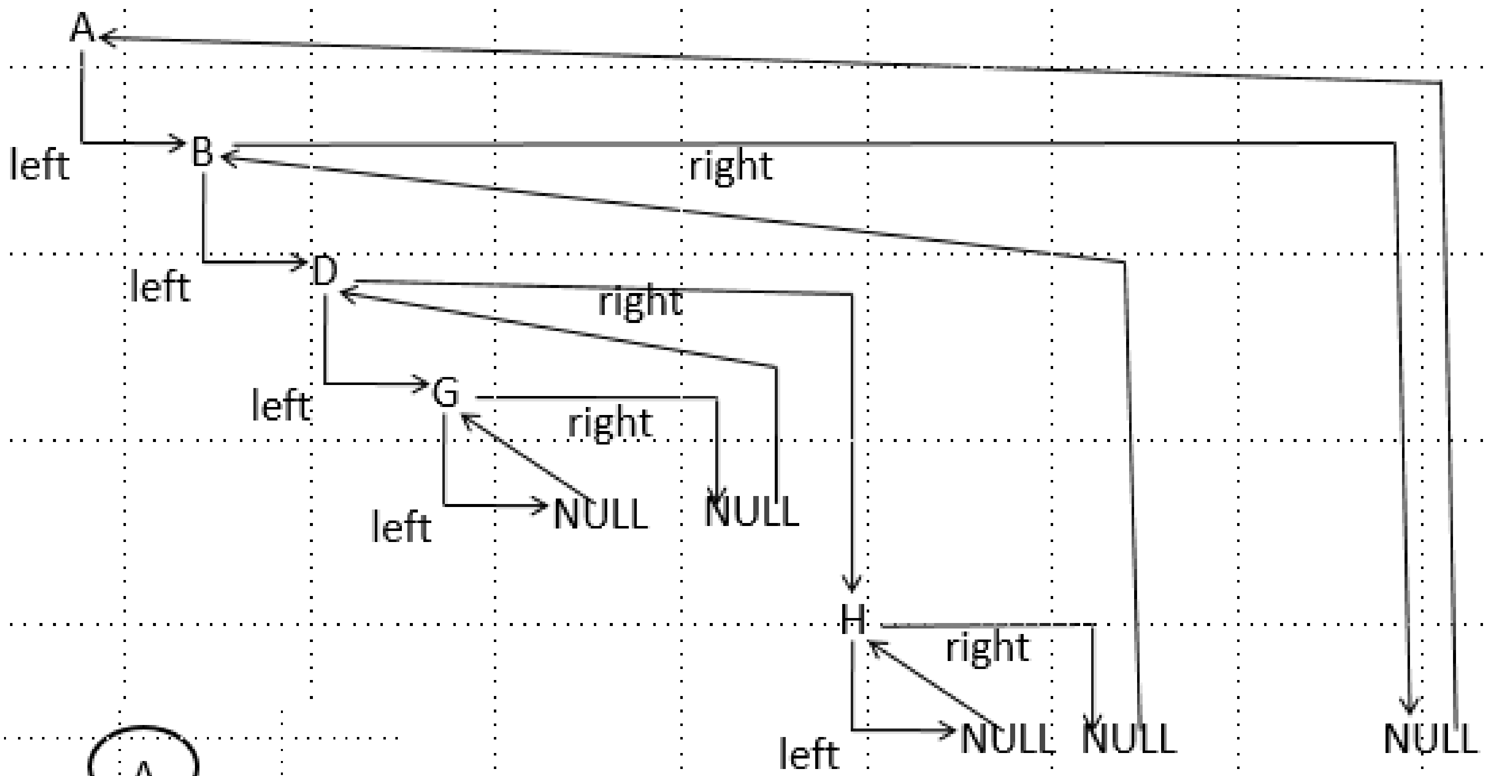


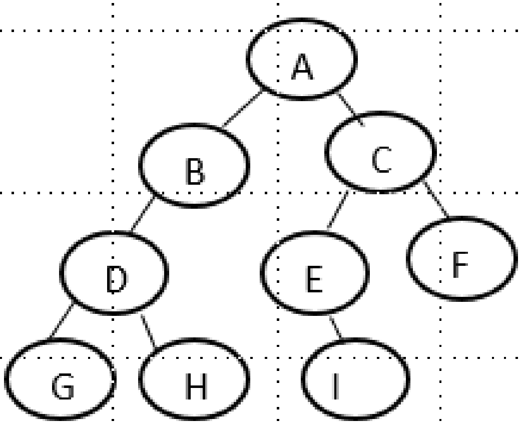
Inorder traversal of above tree gives GDHBAEICF

Inorder Traversal - Example

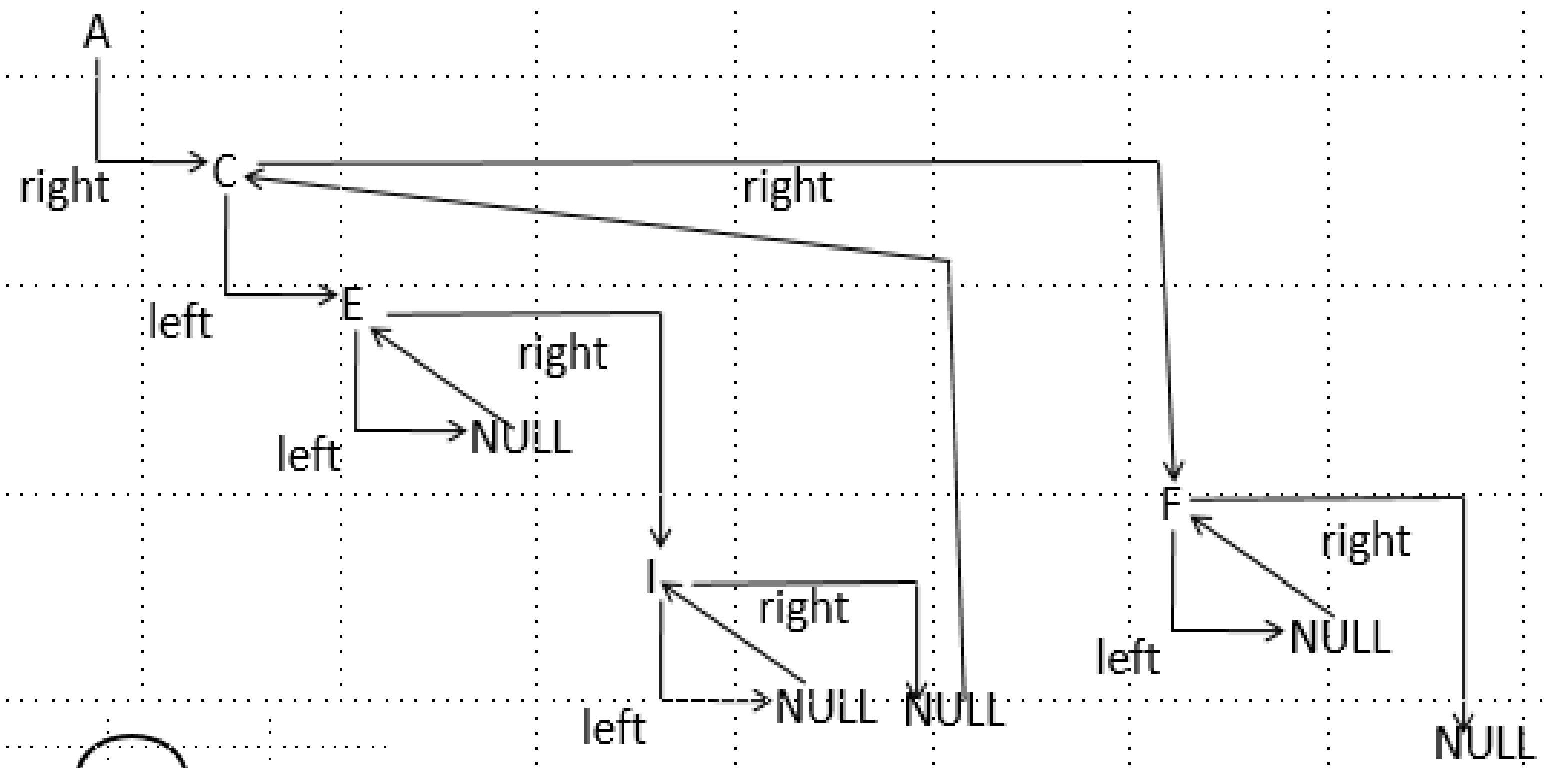


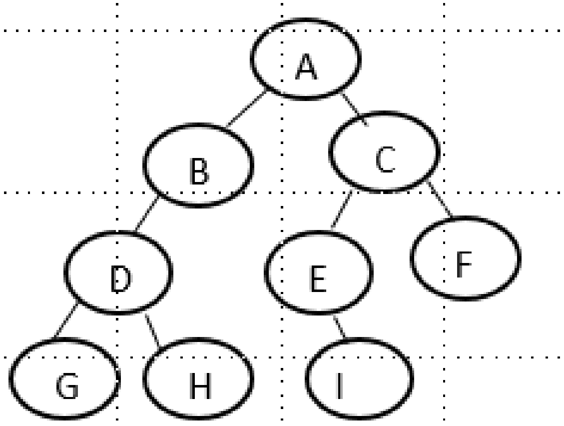
- Move towards left, we end up in G. G does not have a left child. Now display the root node(in this case it is G). Hence G is displayed first.
- Move to the right of G, which is also NULL. Hence go back to root of G and print it. So D is printed next.
- Go to the right of D, which is H. Now another root H is visited.
- Move to the left of H, which is NULL. So go back to root H and print it and go to right of H, which is NULL.
- Go back to the root B and print it and go right of B, which is NULL. So go back to root of B, which is A and print it.
- Traversing of left subtree is finished and so move towards right of it & reach C.
- Move to the left of C and reach E. Again move to left, which is NULL. Print root E and go to right of E to reach I.
- Move to left of I, which is NULL. Hence go back to root I, print it and move to its right, which is NULL.
- Go back to root C, print it and go to its right and reach F.
- Move to left of F, which is NULL. Hence go back to F, print it and go to its right, which is also NULL.


A
left
B
right
left
D
right
left
G
right
left
NULL
NULL
H
right
left
NULL
NULL
NULL


A
B
C
D
E
F
G
H
I

G D H B A


A
right
C
right
left
E
right
left
NULL
I
right
left
NULL
NULL
F
right
left
NULL
NULL

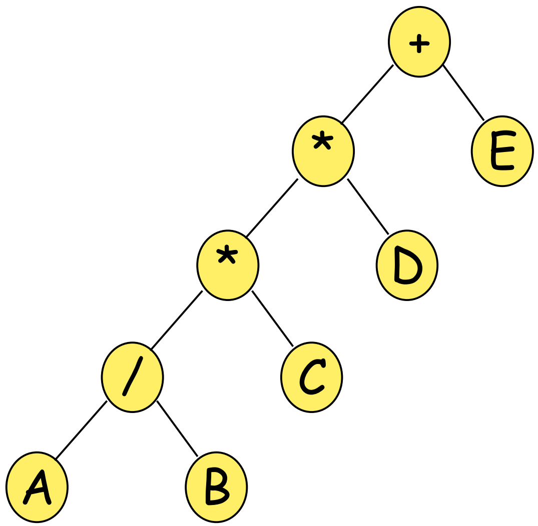

A
B
C
D
E
F
G
H
I

E I C F

Inorder Traversal

```
/*recursive algorithm for inorder traversal*/

void inorder(Nodeptr root)
{
   if (root)
   {
        inorder(root→lchild);
        printf("%d ", root→data);
        inorder(root→rchild);
   }
}
```

Inorder Traversal Example



Binary tree with operators and operands - Expression tree

Call of *inorder*	Value in *root*	Action	*inorder*	in *root*	Value Action
1	+		11	*C*	
2	*		12	*NULL*	
3	*		11	*C*	**printf**
4	/		13	*NULL*	
5	*A*		2	*	**printf**
6	*NULL*		14	*D*	
5	*A*	**printf**	15	*NULL*	
7	*NULL*		14	*D*	**printf**
4	/	**printf**	16	*NULL*	
8	*B*		1	+	**printf**
9	*NULL*		17	*E*	
8	*B*	**printf**	18	*NULL*	
10	*NULL*		17	*E*	**printf**
3	*	**printf**	19	*NULL*	

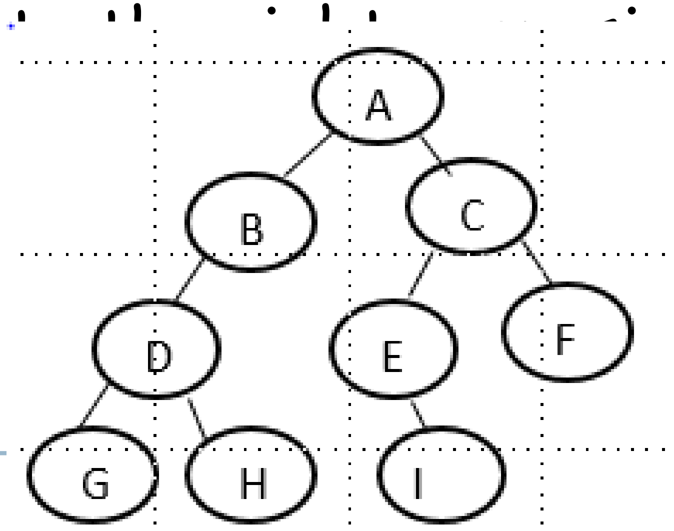
Trace of the program

Preorder Traversal

Preorder traversal is defined as

1. Process the node.
2. Traverse the left subtree in preorder.
3. Traverse the right subtree in preorder.

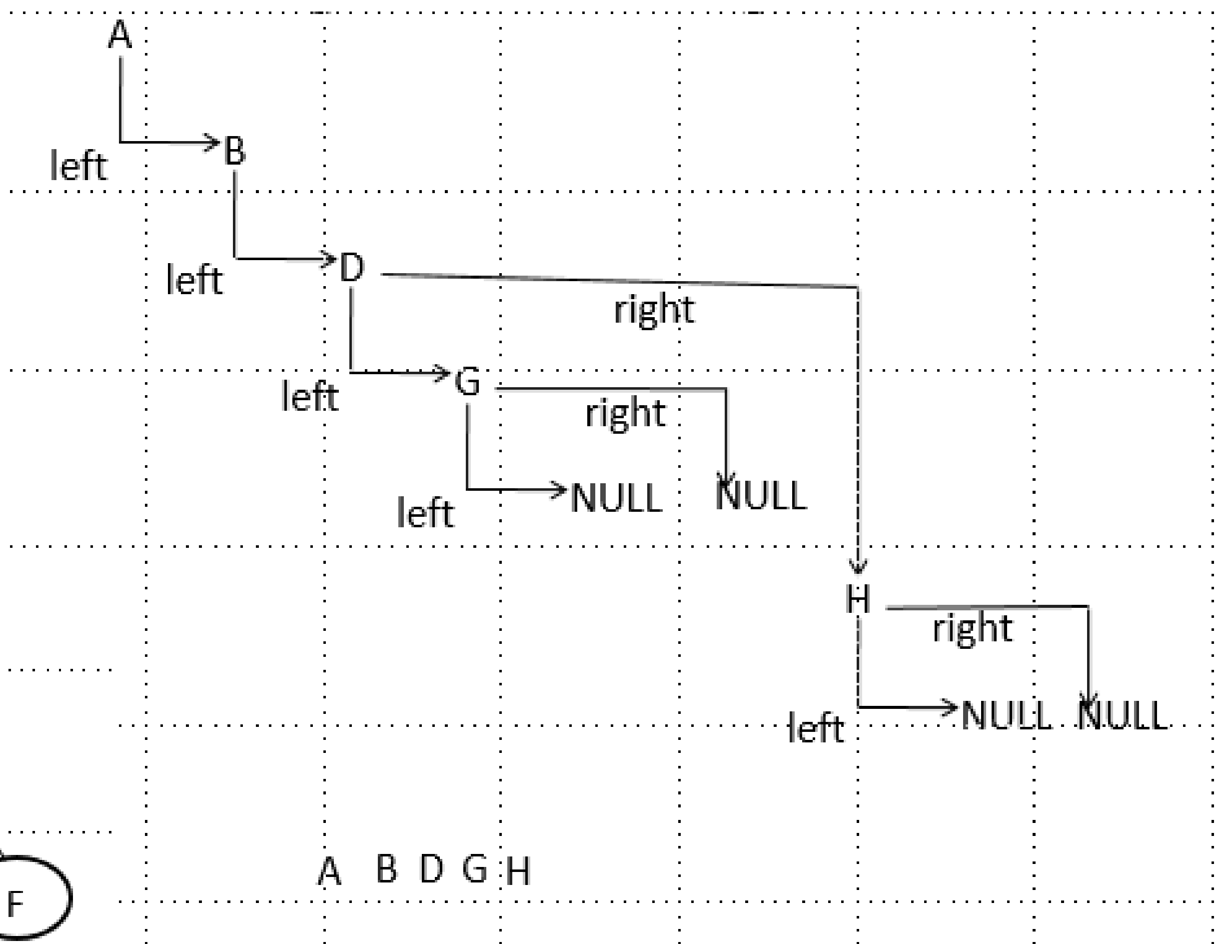
- In preorder, we first visit the node, then move towards left and then [illegible]ely.

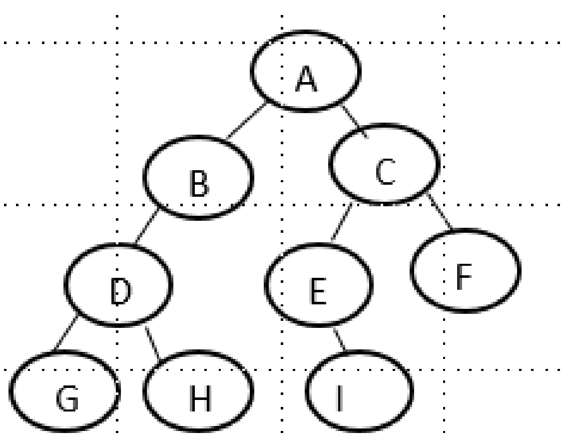


Preorder Traversal

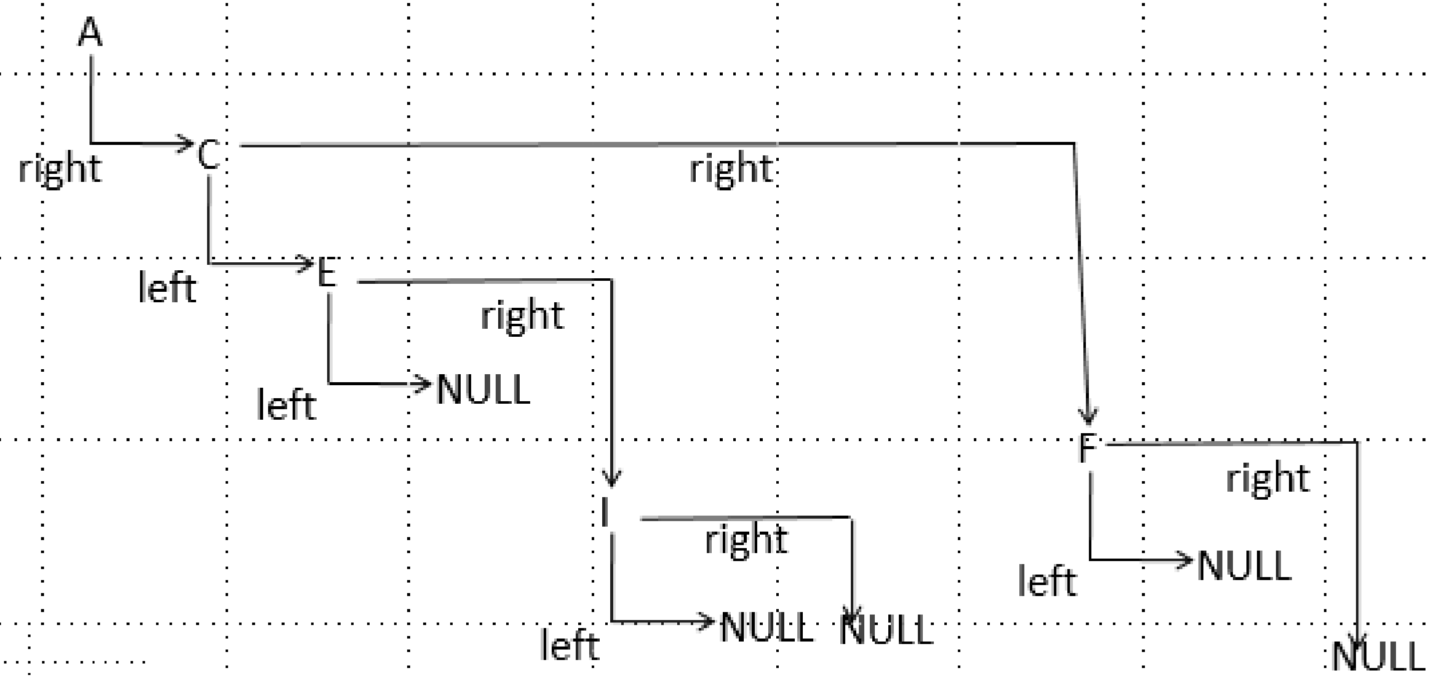
```
void preorder(Nodeptr root)
{

    if(root)
    {
        printf("%d ", root→data);
        preorder(root→lchild);
        preorder(root→rchild);

    }
}
```

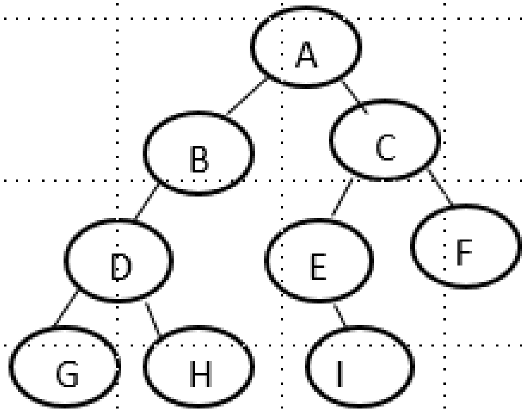
Traversing left sub tree in preorder





Traversing right sub tree in preorder





C E I F

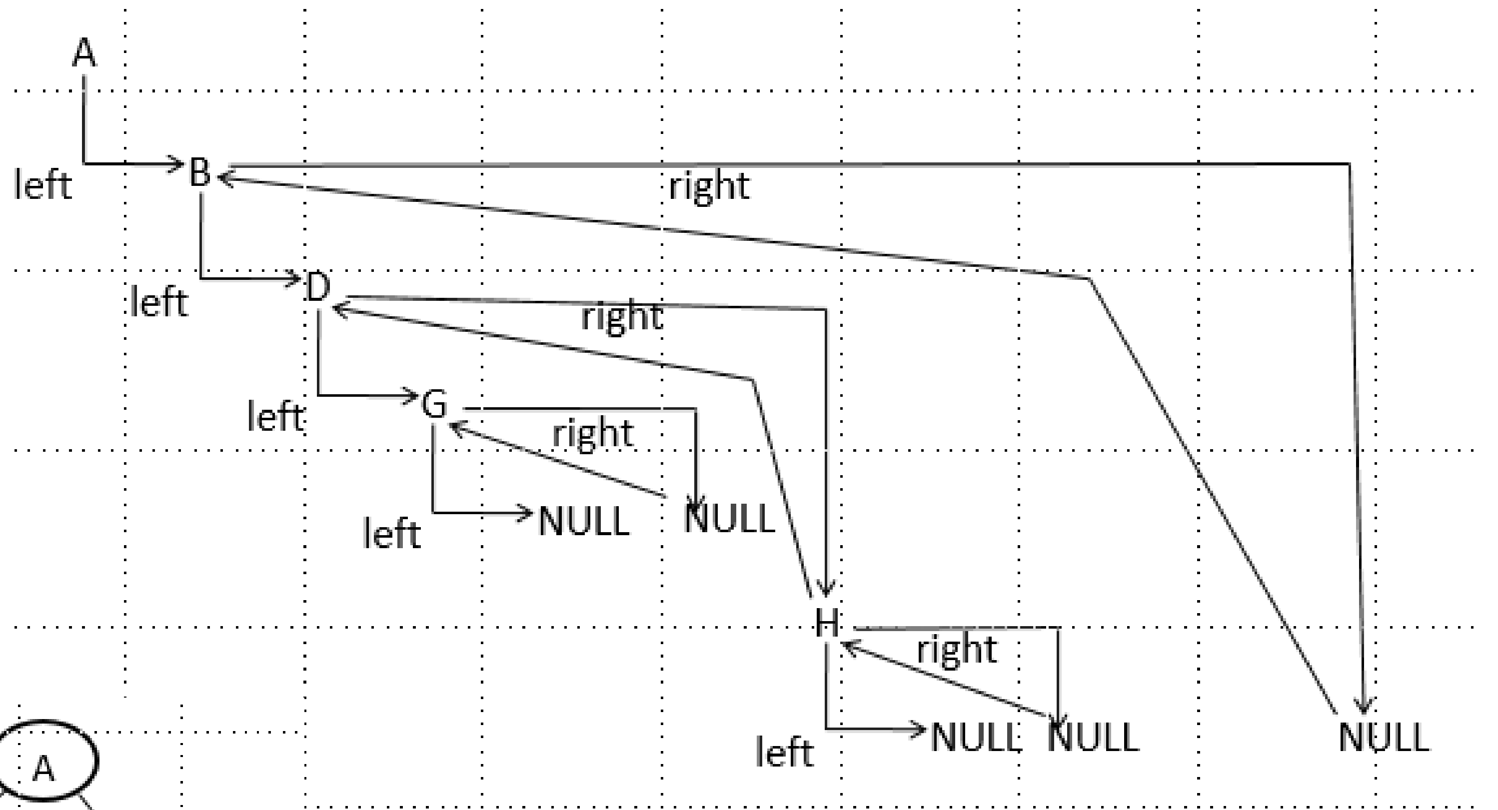
Post order traversal

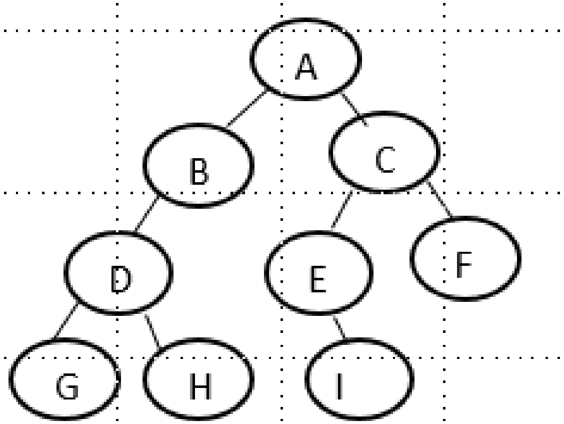
Post order traversal is defined as

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

- In post order traversal, we first traverse towards left, then move to right and then visit the root. This process is repeated recursively.

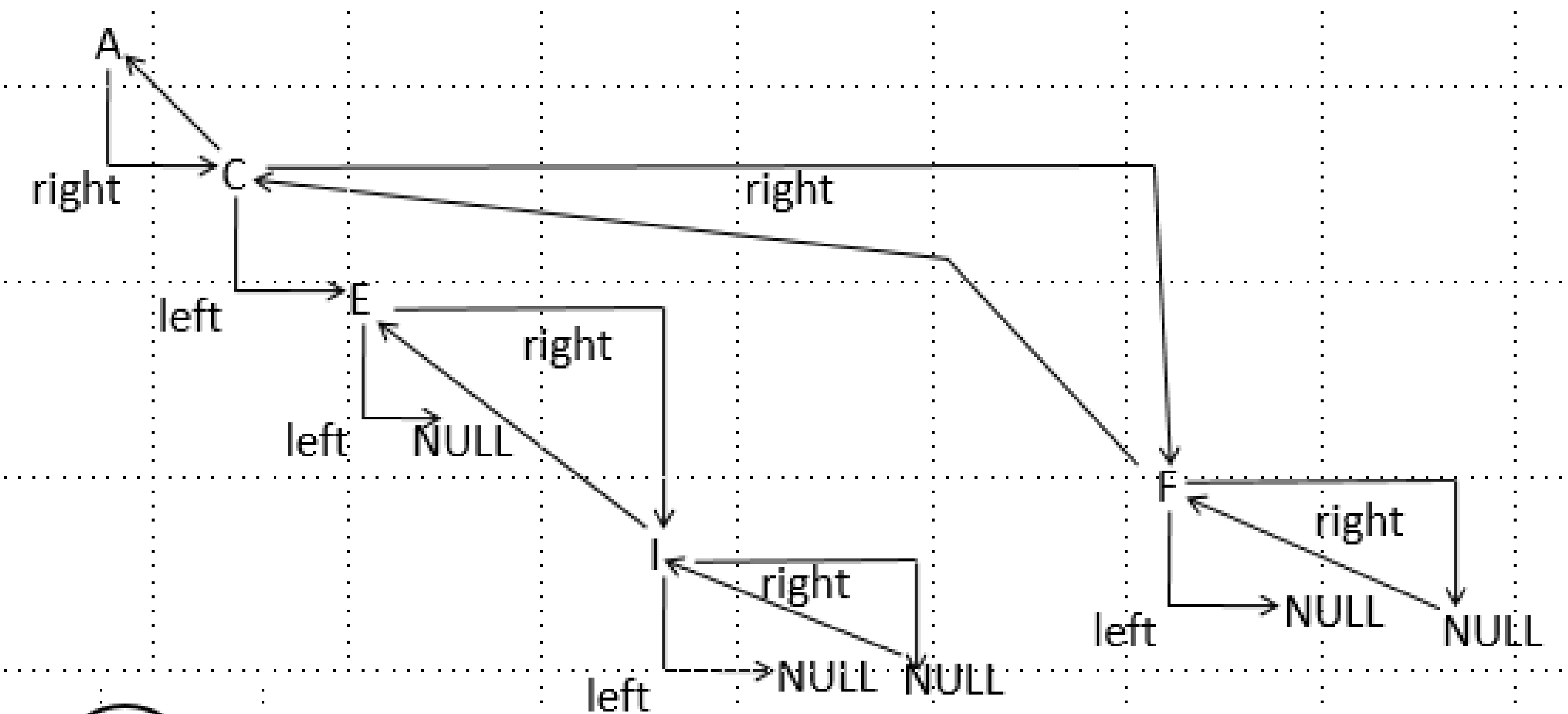
Post order traversal-Example

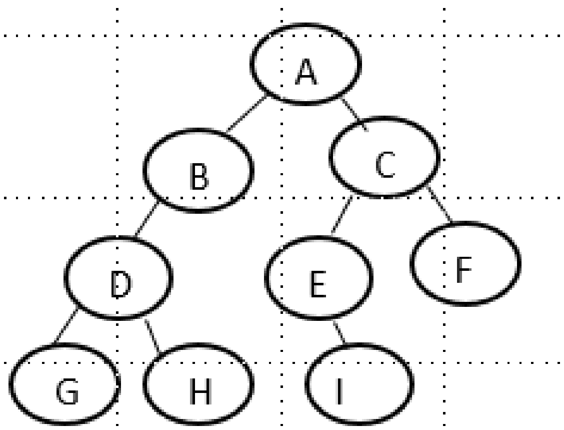




G H D B

Post order traversal - Example



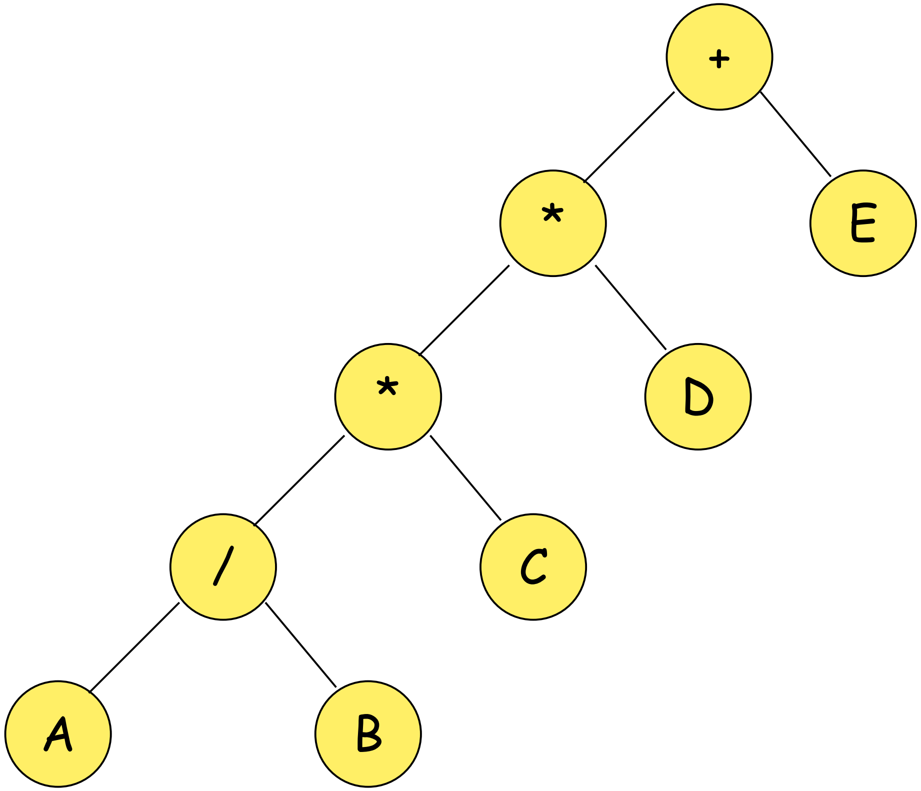


I E F C A

Post order traversal

```
void postorder(Nodeptr root)
{
        if(root)
        {
                postorder(root→lchild);
                postorder(root→rchild);
                printf("%d ",root→data);
        }
}
```

Binary Tree With Arithmetic Expression

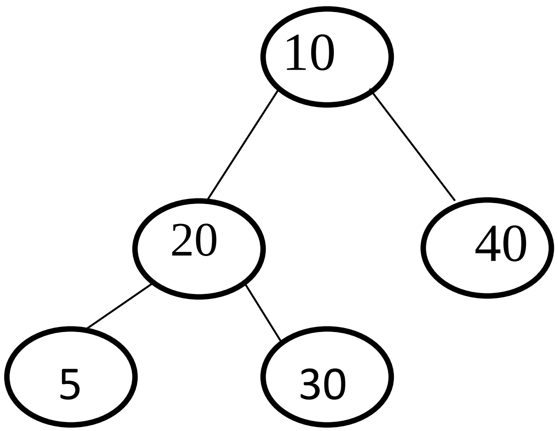


Tree Traversal

- Inorder Traversal: A/B*C*D+E
  => Infix form
- Preorder Traversal: +**/ABCDE
  => Prefix form
- Postorder Traversal: AB/C*D*E+
  => Postfix form

Iterative Inorder Traversal

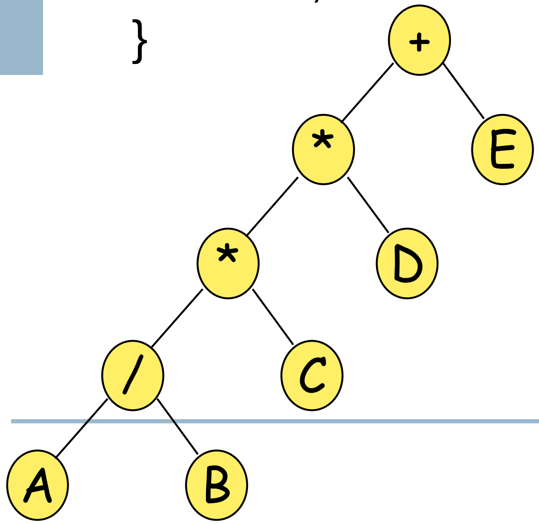
- Every time a node is visited, it is pushed to stack without printing its info and move left.
- After finishing left, pop element from stack, print it and move right.



Here 10, 20 , 5 is pushed to stack. Then pop 5, print it and move right.
Now pop 20, print it and move right and push 30 and move left.
Pop 30, print it and move right.
Pop 10, print it and move right and push 40 and so on

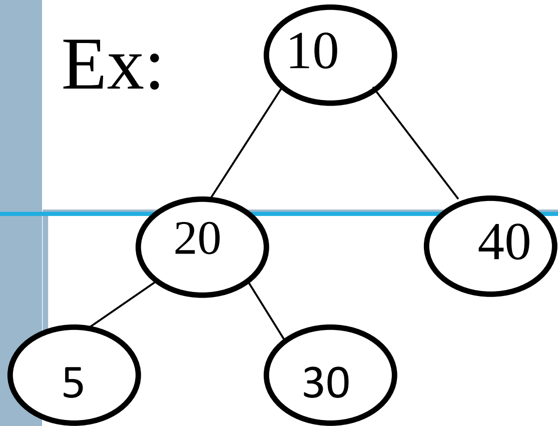
Iterative Inorder Traversal

```
void iterative_inorder(Nodeptr root)
{
    Nodeptr cur;
    int done = false;

    STACK *s, s1;
    s= &s1;
    s->top = -1;
    if(root==NULL){
        printf("Empty Tree\n");
        return;
    }
```




```
    cur=root;
    while(!done){
        while(cur!=NULL){
            Push(s, cur);
            cur=cur->lchild;
        }
        if(!IsEmptyStack(s)){
            cur=Pop(s);
            printf("%d ",cur->data);
            cur=cur->rchild;
        }
        else
            done = true;
    }
}
```

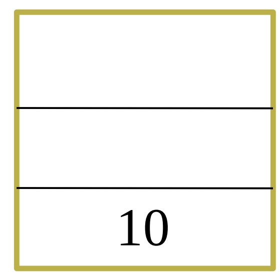
Ex:



1. After 1st iteration of while loop

Node 10 is pushed to stack

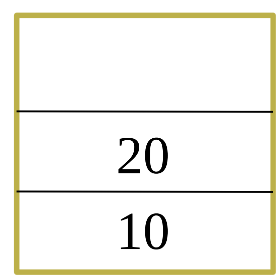
cur=cur→lchild i.e cur=20



2. After 2nd iteration of while loop

Node 20 is pushed to stack

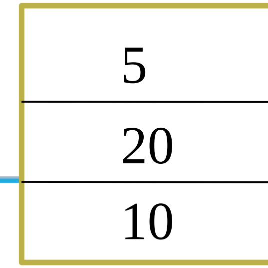
cur=cur→lchild; i.e cur=5



3. After 3rd iteration of while loop

Node 5 is pushed to stack

cur=cur→lchild; i.e cur=NULL



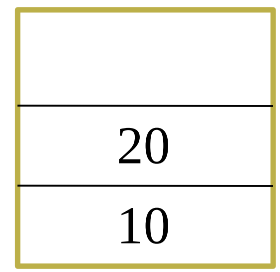
4. While loop terminates since cur==NULL

Stack is not empty

cur=pop( );i.e cur=node 5;

Print 5

cur=cur→rchild; i.e cur=NULL



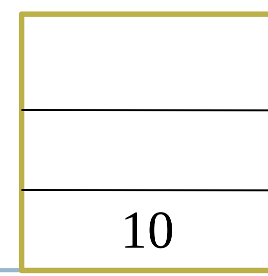
5. cur==NULL, while loop not entered

Stack not empty

cur=pop( ); i.e cur=node 20

Print 20

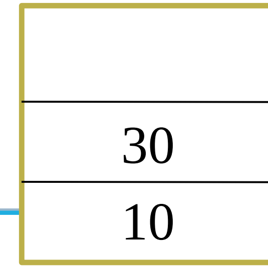
cur=cur→rchild; i.e cur=30



6. While loop is entered

Node 30 is pushed to stack

cur=cur→lchild; i.e cur=NULL



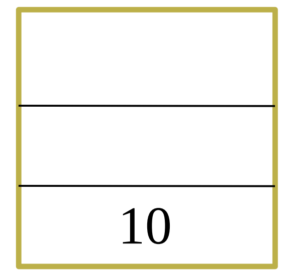
7. While loop terminates since cur==NULL

Stack not empty

cur=pop( );i.e cur=node 30;

Print 30

cur=cur→rchild; i.e cur=NULL



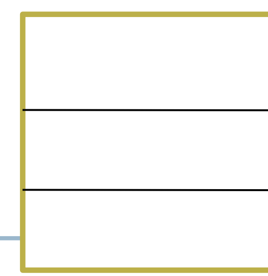
8. cur==NULL, while loop not entered

Stack not empty

cur=pop( ); i.e cur=node 10

Print 10

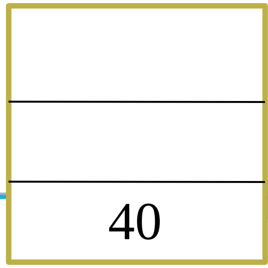
cur=cur→rchild; i.e cur=40



9. While loop is entered

Node 40 is pushed to stack

cur=cur→lchild; i.e cur=NULL



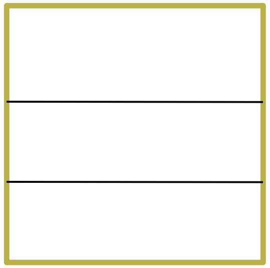
10. While loop terminates since cur==NULL

Stack not empty

cur=pop( );i.e cur=node 40;

Print 40

cur=cur→rchild; i.e cur=NULL



11. cur==NULL and stack empty

return

Hence elements printed are: 5 20 30 10 40

Iterative postorder traversal

- Here a flag variable to keep track of traversing. Flag is associated with each node. Flag== -1 indicates that traversing right subtree of that node is over.

Algorithm:

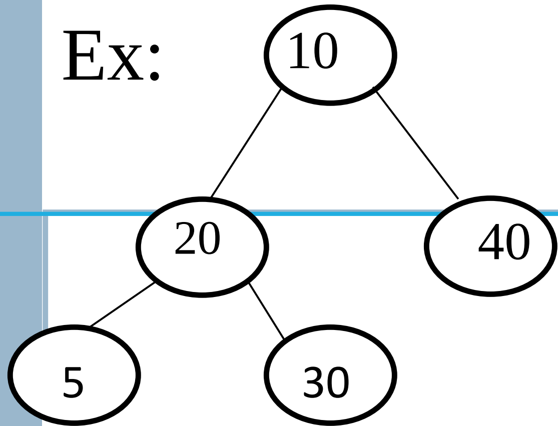
- Traverse left and push the nodes to stack with their flags set to 1, until NULL is reached.
- Then flag of current node is set to -1 and its right subtree is traversed. Flag is set to -1 to indicate that traversing right subtree of that node is over.
- Hence is flag is -1, it means traversing right subtree of that node is over and you can print the item. if flag is not –ve, traversing right is not done, hence traverse right.

Iterative Postorder Traversal

```
/*function for iterative postorder traversal*/
void postorder(Nodeptr root)
{
    struct stack
    {
        Nodeptr node;
        int flag;
    };
    Nodeptr cur;
    struct stack  s[20];
    int top=-1;
    if(root==NULL)
    {
        printf("tree is empty");
        return;
    }
```

```
cur=root;
for(; ;){
    while(cur!=NULL){           //traverse left and push the nodes to the stack and set flag to 1
            s[++top].node=cur;
            s[top].flag = 1;
            cur=cur→llink;
    }
    while(s[top].flag<0){    //if flag is –ve, right subtree is visited and hence node is popped and printed
            cur=s[top--].node;
            printf("%d", cur→info);
            if(stack_empty(top))    //if stack is empty, traversal is complete
                        return;
    }
    cur= s[top].node;   //after left subtree is traversed, move to right and  set its flag to -1 to
                        //indicate  right subtree is traversed*/
    cur=cur→rlink;
    s[top].flag = -1;
}
}
```

Ex:



Initially cur =10;

10	1

1. After 1st iteration of 1st while loop

Node 10 is pushed to stack & its flag set to 1.

cur=cur→lchild; i.e cur=20

20	1
10	1

2. After 2nd iteration of 1st while loop

Node 20 is pushed to stack & its flag set to 1

cur=cur→lchild; i.e cur=5

3. After 3rd iteration of 1st while loop

Node 5 is pushed to stack & its flag set to 1.

cur=cur→lchild; i.e cur=NULL

5	1
20	1
10	1

4. While loop terminates since cur==NULL

Since s[top].flag!=-1, 2nd while not entered.

cur=s[top].node; i.e cur=node 5;

cur=cur→rchild; i.e cur=NULL;

s[top].flag =-1

5	-1
20	1
10	1

5. cur==NULL, 1st while loop not entered

Since s.[top].flag<0, 2nd while is entered.

cur=s[top].node; i.e cur=node 5;

Print 5

Stack is not empty, continue;

20	1
10	1

6. s[top].flag!=-1, 2nd While loop is exited

cur=s[top].node; i.e cur=20;

cur=cur→rchild; i.e cur=30;

s[top].flag =-1

20	-1
10	1

7. 1st while is entered

Node 30 is pushed to stack & its flag set to 1.

cur=cur→lchild; i.e cur=NULL

30	1
20	-1
10	1

8. cur==NULL, 1st while loop exits

Since s[top]!=-1, 2nd while not entered.

cur=s[top].node; i.e cur=30;

cur=cur→rchild; i.e cur=NULL;

s[top].flag =-1

30	-1
20	-1
10	1

9. cur==NULL, 1st while loop not entered

Since s.[top].flag<0, 2nd while is entered.

cur=s[top].node; i.e cur=node 30;

Print 30

Stack is not empty, continue;

20	-1
10	1

10. s[top].flag<0, 2nd While loop continues

cur=s[top].node; i.e cur=node 20;

Print 20

Stack is not empty, continue;

10	1

11. s[top].flag!=-1, 2nd While loop is exited

cur=s[top].node; i.e cur=10;

cur=cur→rchild; i.e cur=40;

s[top].flag =-1

10	-1

12. 1st while is entered

Node 40 is pushed to stack & its flag set to 1.

cur=cur→lchild; i.e cur=NULL

40	1
10	-1

13. cur==NULL, 1st while loop exits

Since s[top]!=-1, 2nd while not entered.

cur=s[top].node; i.e cur=40;

cur=cur→rchild; i.e cur=NULL;

s[top].flag =-1

40	-1
10	-1

14. cur==NULL, 1st while loop not entered

Since s.[top].flag<0, 2nd while is entered.

cur=s[top].node; i.e cur=node 40;

Print 40

Stack is not empty, continue;

10	-1

15. s[top].flag<0, 2nd While loop continues

cur=s[top].node; i.e cur=node 10;

Print 10

Stack is empty, stop;

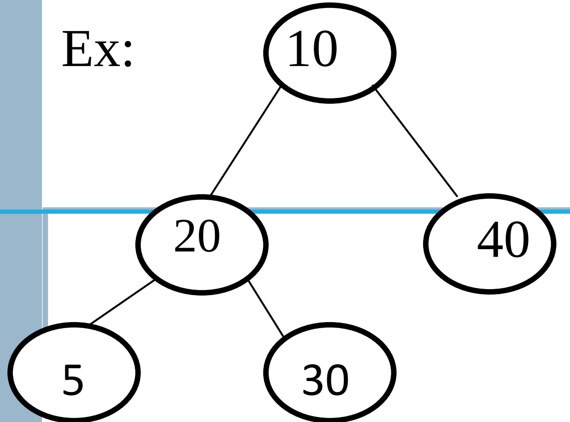
Hence elements printed in postorder are: 5, 30, 20, 40, 10

Iterative Preorder Traversal

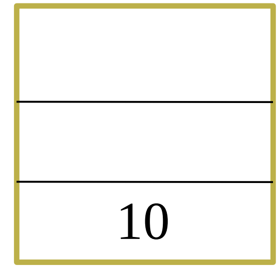
```
void preorder(NODEPTR root)
{
   STACK *s, s1;
    s= &s1;
    s->top = -1;

   Nodeptr cur;
   if(root==NULL){
         printf("tree is empty");
         return;
    }
```

```
Push(s, root);
while(!IsEmpty(s)){
    cur = Pop(s);
    printf("%d ", cur->data);
    if (cur->rchild) Push(s, cur->rchild);

     if (cur->lchild) Push(s, cur->lchild);
}
```

Ex:



Node 10 is pushed to stack



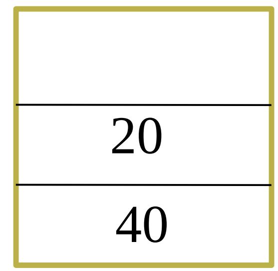
1. <u>After 1st iteration of while loop</u>

cur = Pop(s);

i.e 10 is popped and printed

cur→rchild and cur->lchild are pushed

i.e 40 and 20 are pushed



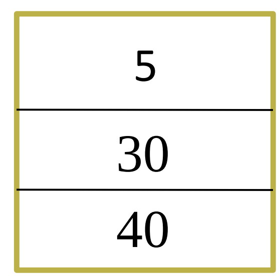
2. <u>After 2nd iteration of while loop</u>

cur = Pop(s);

i.e 20 is popped and printed

cur→rchild and cur->lchild are pushed

i.e 30 and 5 are pushed



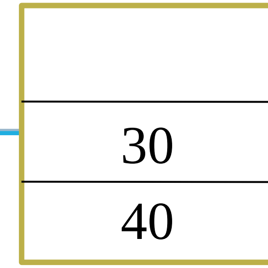
3. After 3^{rd} iteration of while loop

cur = Pop(s);

i.e 5 is popped and printed

cur→rchild and cur->lchild are NULL

Hence No Push



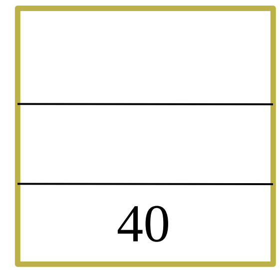
4. After 4^{th} iteration of while loop

cur=pop( );

i.e 30 is popped and printed;

cur→rchild and cur->lchild are NULL

Hence No Push



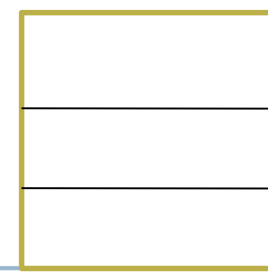
5. After 5^{th} iteration of while loop

cur=pop( );

i.e 40 is popped and printed;

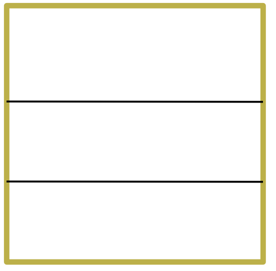
cur→rchild and cur->lchild are NULL

Hence No Push



10. While loop terminates since stack is empty

Hence elements printed are 10, 20, 5, 30, 40

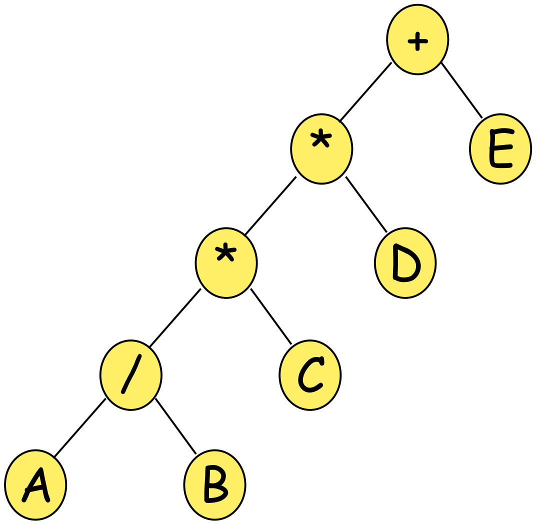


Level-Order Traversal

- All previous mentioned schemes use stacks.
- Level-order traversal uses a queue.
- Level-order scheme visits the root first, then the root's left child, followed by the root's right child.
- All the node at a level are visited before moving down to another level.

Level-Order Traversal of A Binary Tree

```
void Levelorder(Nodeptr root){
    QUEUE *q, q1;
    q = &q1;
    q->Front= -1;
    q->Rear = -1;
    if (root== NULL)    {
        printf("\nEmpty Tree\n");
        return;
    }
    InsertQ(q,root);
    while(!IsEmpty(q)){
        Nodeptr temp= DeleteQ(q);
        printf("%d ", temp->data);
        if (temp->lchild) InsertQ(q,temp->lchild);
        if (temp->rchild) InsertQ(q,temp->rchild);

    }

}
```

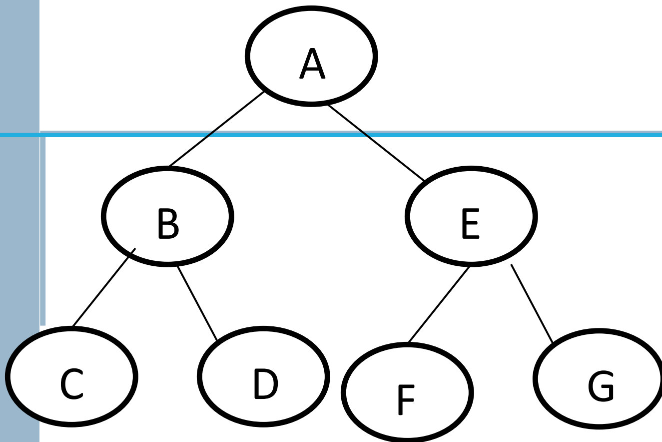


+*E*D/CAB

Insertion into a binary Tree

- A node can be inserted in any position in a tree(unless it is a binary search tree)
- A node cannot be inserted in the already occupied position.
- User has to specify where to insert the item. This can be done by specifying the direction in the form of a string.

For ex: if the direction string is "LLR", it means start from root and go left(L) of it, again go left(L) of present node and finally go right(R) of current node. Insert the new node at this position.



For the above tree if the direction of insertion is “LLR”

- Start from root. i.e A and go left. B is reached.
- Again go left and you will reach C.
- From C, go right and insert the node.

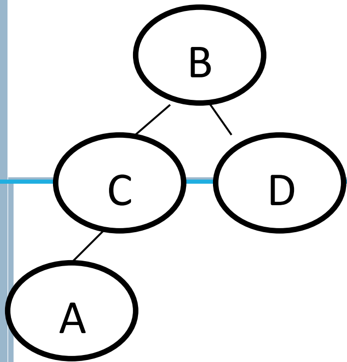
Hence the node is inserted to the right of C.

- To implement this, we make use of 2 pointer variables parent and cur. At any point parent points to the parent node and cur points to the child.

```
void insert(Nodeptr root, char direction[], int ele) { // assume root is already created and tree exists
    int i;
    Nodeptr temp,cur,parent;

    temp= getnode();
    temp->data=ele;
    temp->lchild=temp->rchild=NULL;

    parent = NULL;
    cur=root;
    i=0;
    while (cur && direction[i]) {//traverse down the tree
        parent = cur;
        if(direction[i]=='L' || direction[i]=='l')
            cur=cur->lchild;
        else
            cur=cur->rchild;
        i++;
    }
    if ((cur != NULL) || (direction[i]!='\0')) {   /*node already present at specified pos/incorrect dir string */
       printf("Insertion Not possible\n") ;
       free(temp);
       return;
    }
    /*Based on last character of direction string  insert as a left or right child */
    if(direction[i-1]=='L' || direction[i-1]=='l')
            parent->lchild=temp;
    else
            parent->rchild=temp;
}
```

- Control comes out of while loop, if direction[i] == ‘\0’ or when cur==NULL.
- For insertion to be possible, control should come out when cur==NULL and direction[i] == ‘\0’ both at the same time.

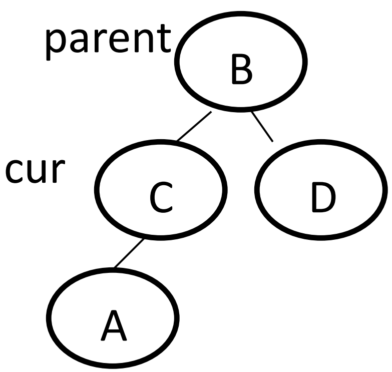
  i.e when we get the position to insert(cur==NULL), the string should be completed.



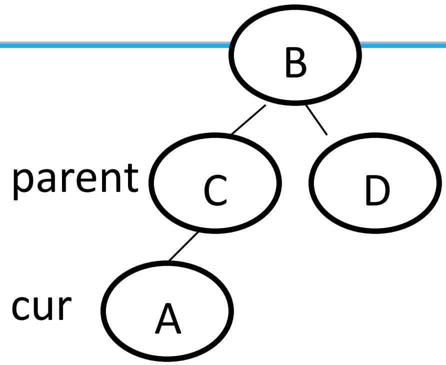
Let the direction string be “LLR”. String length of string is 3.

Initially cur ==root and parent==NULL

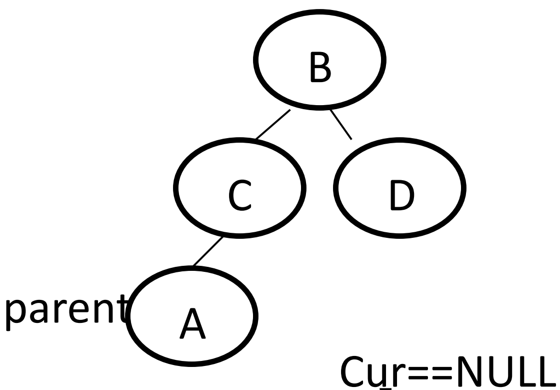
- Loop 1: i=0 ,direction[0]=‘L’



- Loop 2: i=1, direction[1]='L'



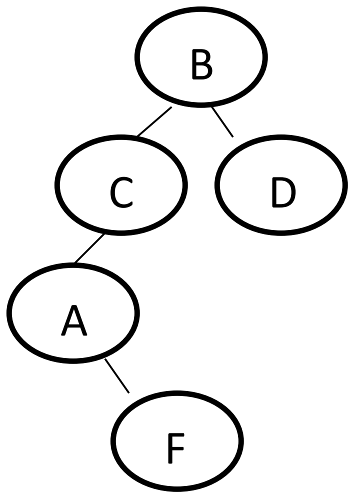
- Loop 3: i=2, directiom[2]='R'



- i=3, Now loop terminates since cur==NULL. Here i==3(i.e direction[i] == '\0') and cur==NULL. Hence insertion is possible

- Now direction[i-1] is direction[3-1] which gives ‘R’.
- Hence parent→rchild is made to point to new node.

After insertion tree looks like



- If direction of insertion is “LLR” for the above tree. Here the loop terminates when direction[i] == ‘\0’. But at this point cur is not equal to NULL. Hence insertion is not possible.

Some Other Binary Tree Functions

- With the inorder, postorder, or preorder mechanisms, we can implement all needed binary tree functions. E.g.,
  - Copying Binary Trees
  - Testing Equality
    - Two binary trees are equal if their topologies are the same and the information in corresponding nodes is identical.

Searching:

- Searching an item in the tree can be done while traversing the tree in inorder, preorder or postorder traversals.
- While visiting each node during traversal, instead of printing the node info, it is checked with the item to be searched.
- If item is found, search is successful.

Searching a binary tree

```
int Search(Nodeptr root,int ele) //uses preorder traversal
{
    static int t=0;
    if(root)
    {
        if(root->data==ele){
            t++;
            return t;
        }
        if (t==0) Search(root->lchild,ele);
        if (t==0) Search(root->rchild,ele);
    }
}
```

Creating a copy of a binary tree

- Getting the exact copy of the given tree.

```
/*recursive function to copy a tree*/
Nodeptr copy (Nodeptr root){
    Nodeptr temp;
    if(root == NULL)
        return NULL;
    temp=getnode();
    temp→data=root→data;
    temp→lchild=copy(root→lchild);
    temp→rchild=copy(root→rchild);
    return temp;
}
```

Finding height of a binary tree

```
/*recursive function to find the height of a tree*/
int height (NODEPTR root)
{
    if(root==NULL)
        return 0;
    return( 1+ max(height (root→lchild), height(root→rchild)));
}
/*max function*/
int max(int a, int b){
    if(a>b)
        return a;
    else
        return b;
}
```

Finding height of a binary tree

Counting the number of nodes in a tree:

- Traverse the tree in any of the 3 techniques and every time a node is visited, count is incremented.

```
void count_nodes( Nodeptr root)
{
    static int count = 0;
    if(root!=NULL)
    {
        count_nodes(root→llink);
        count++;
        count_nodes(root→rlink);
    }
    return count;
}
```

Counting the number of leaf nodes in a binary tree

- Every time a node is visited, check whether the right and left link of that node is NULL. If yes, count is incremented.

```
/*counting number of leaf nodes using inorder technique*/
int count_leafnodes( Nodeptr root){
    static int count = 0;
    if(root!=NULL){
        if(root->lchild==NULL && root->rchild==NULL)
                count++;
        count_leafnodes(root->lchild);

        count_leafnodes(root->rchild);
    }

    return count;
}
```

Equality of 2 binary trees

Returns FALSE if the binary trees root1 and root2 are not equal otherwise returns TRUE

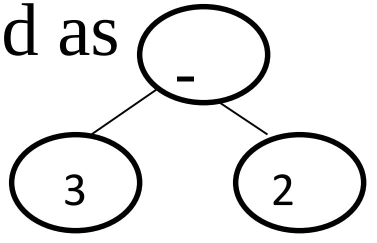
```
int Equal( Nodeptr root1, Nodeptr root2)
{
return ((!root1 && !root2) || (root1 && root2 &&
    (root1→data == root2→data) && Equal( root1-
    >lchild,root2->lchild) && Equal ( root1-
    >rchild,root2->rchild));
}
```

Applications of binary trees

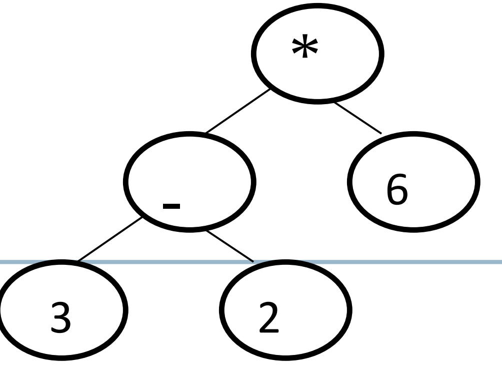
Conversion of expressions:

- An infix expression consisting of operators and operands can be represented using a binary tree with root as operator.
- Non leaf node contains the operator and leaf nodes contain operands.

(3-2) can be represented as



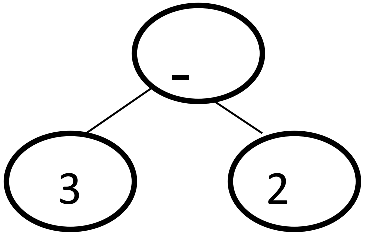
((3-2)*6) can be represented as



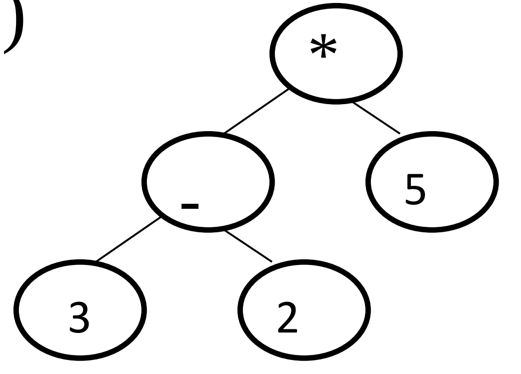
Represent the following expression using binary tree

((6+(3-2)*5)^2)

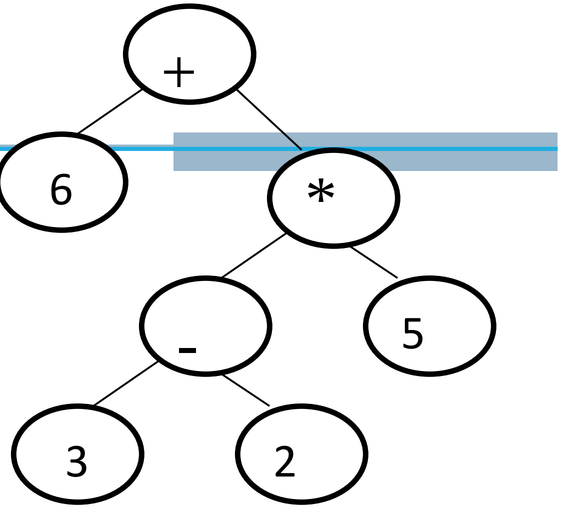
First innermost parenthesis is considered, which is (3-2) and partial tree is drawn for this



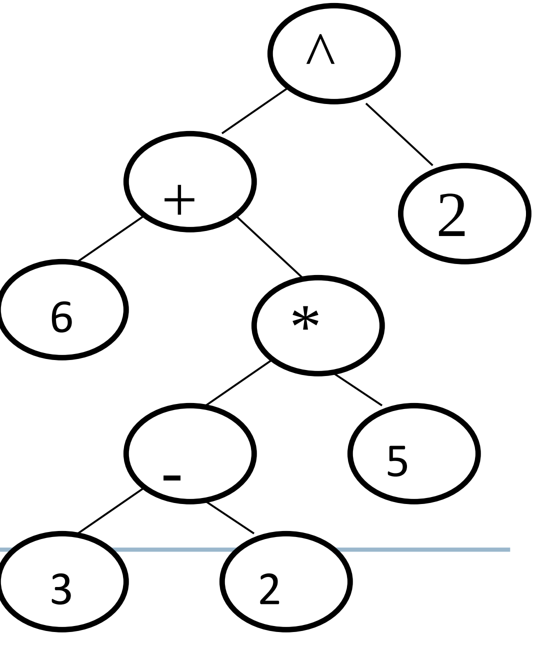
Next tree is extended to include ((3-2)*5)



After considering (6+(3-2)*5)



After (6+(3-2)*5)^2



- When the binary tree for infix expression is traversed in inorder technique, infix expression is obtained.
- Preorder traversal gives the prefix expression and postorder traversal gives the postorder expression.

Inorder traversal of above tree:(6+(3-2)*5)^2

Preorder traversal: ^+6*-3252

Postorder traversal: 632-5*+2^

Creating a binary tree for postfix expression:

Steps:

1. Scan the expression from left to right.
2. Create a node for each symbol encountered.
3. If the symbol is an operand, push the corresponding node on to the stack.
4. If the symbol is an operator, pop top node from stack and attach it to the right of the node with the operator. Next pop present top node and attach it to the left of node with the operator. Push the operator node to the stack.
5. Repeat the process for each symbol in postfix expression. Finally address of root node of expression tree is on top of stack.

Stack of Nodeptr

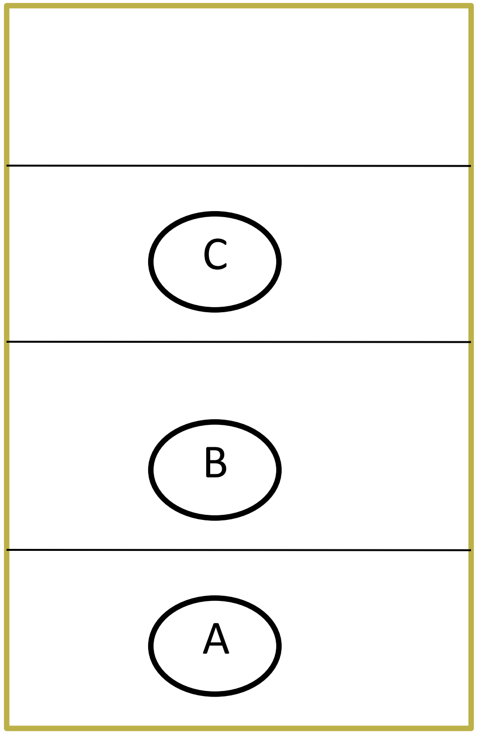
```
#define MaxSize 100
typedef struct node *Nodeptr;

typedef struct{
    Nodeptr Stack[MaxSize];
    int top;
}STACK;

int IsEmptyStack(STACK *s){
    if (s->top==-1)
        return 1;
    else
        return 0;
}
void Push(STACK *s,Nodeptr x){
    if (s->top==MaxSize-1){
        printf("Stack Overflow");
        return;
    }
    s->Stack[++s->top]=x;
}
Nodeptr Pop(STACK *s){
    return (s->Stack[s->top--]);
}
```

```
/* C function to create a binary tree for postfix expression*/
Nodeptr create_tree( char postfix[])
{
    Nodeptr temp;

    STACK *s, s1;
    s = &s1;
    int i=0, k=0;
    char symbol;
    while((symbol=postfix[i++])!='\0')
    {
        temp=getnode();
        temp→info=symbol;
        temp→llink=temp→rlink=NULL;
        if(isalnum(symbol))                 /* if operand push it*/
            Push(s,temp );
        else                                /* else pop element add it to the
        {                                      right of operator node. Pop

            temp→rlink=Pop(s);  next element and add  to

            temp→llink=Pop(s);  left. Push operator node*/
            Push(s, temp)
        }
    }
    return(Pop(s));                         /* return root*/
}
```

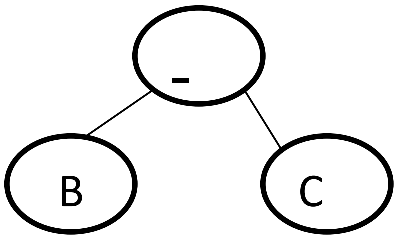
Create a binary tree for the given postfix expression:

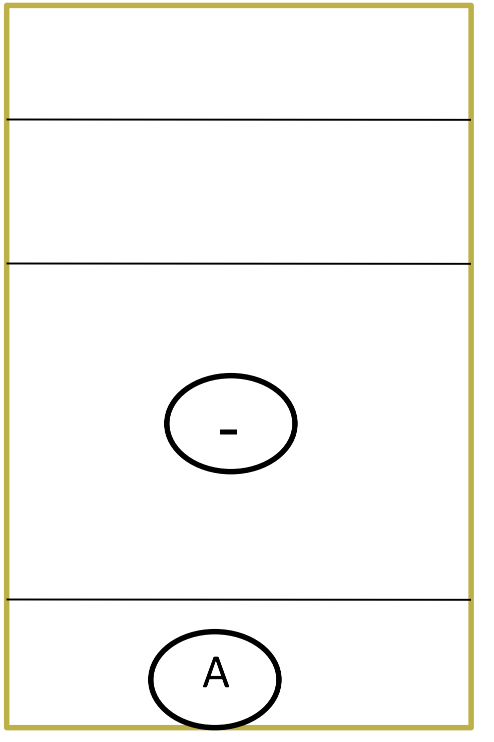
abc-d*+

1. First 3 symbols are operands, hence after pushing these 3 symbols, stack of nodes looks like

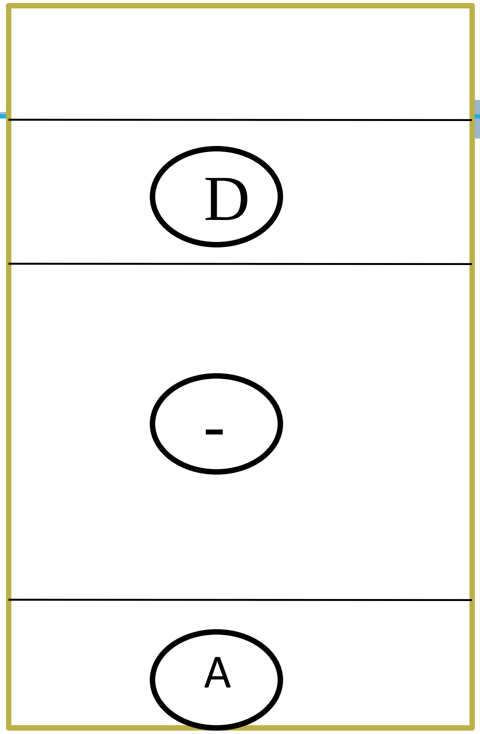


2. Now we get operator ‘–’. Pop top 2 elements and add them to right and left of node with ‘-’respectively and push node with operator ‘-’ to stack.

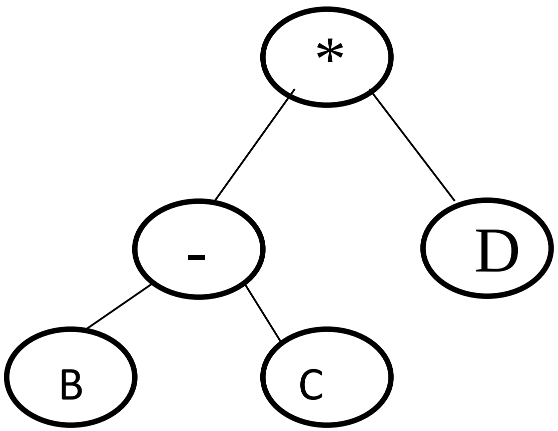


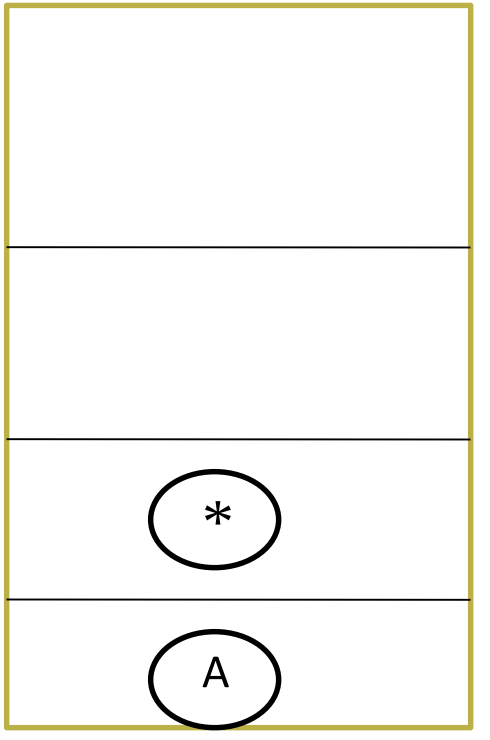


3. Push sumbol ‘D’ to stack

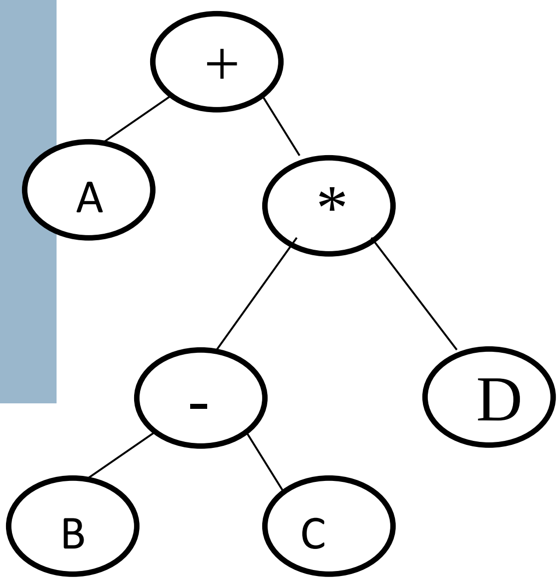


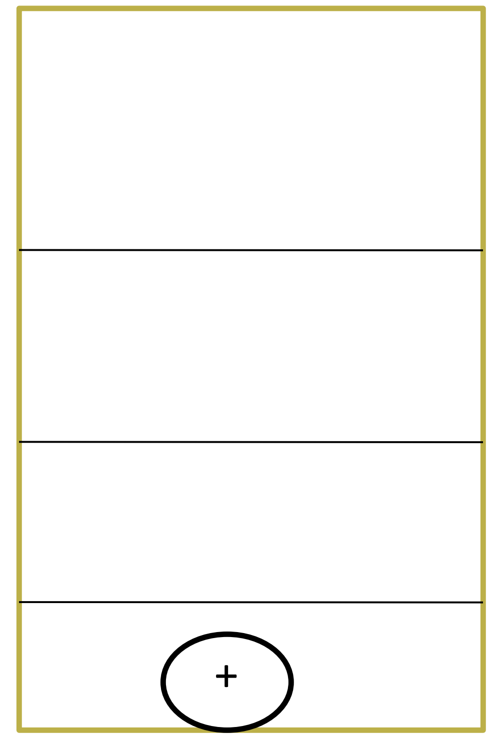
4. Now the operator is '*'. Pop top 2 elements and add it to right and left of node with '*' respectively and push the operator node to stack.





5. Next is the operator ‘+’. Hence after popping and pushing, stack will be

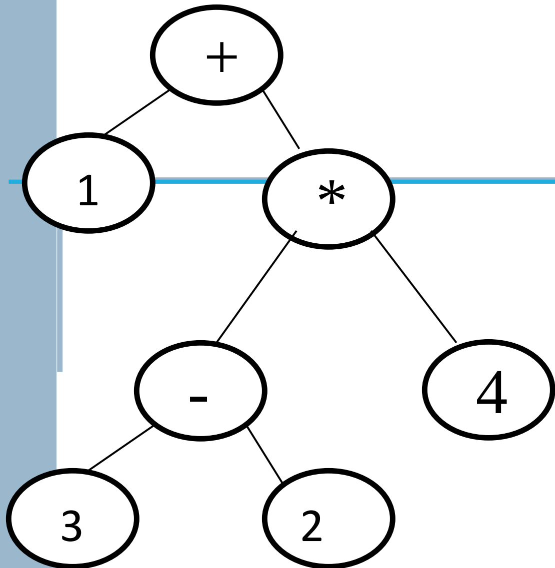


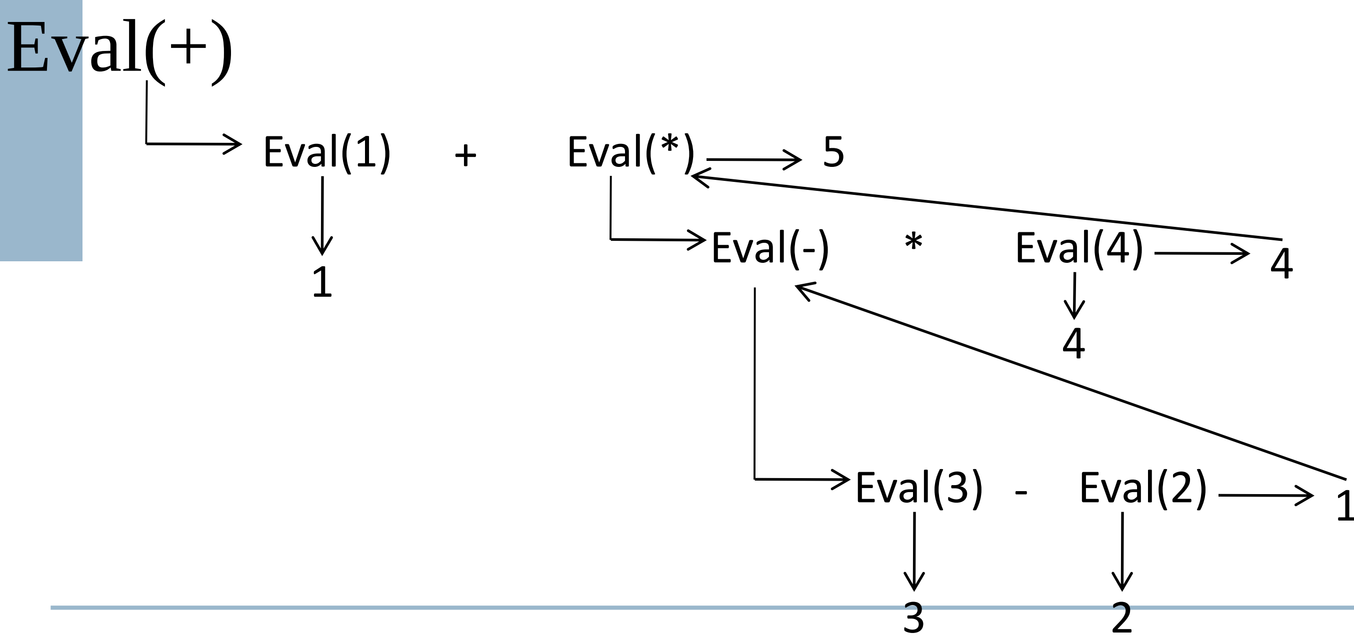


Now stack top will have the root of the final tree.

Evaluating the expression tree using recursion:

```
int eval( Nodeptr root)
{
    float num;
    switch(root→data)
    {
        case '+':return eval(root→lchild)+ eval(root→rchild);
        case '-' :return eval(root→ lchild)- eval(root→rchild);
        case '/':return eval(root→lchild)/ eval(root→rchild);
        case '*' :return eval(root→ lchild)* eval(root→rchild);
        case '$':
        case '^':return pow(eval(root→lchild),
                                    eval(root→rchild));
        default :return(root→data – '0'); //base case
    }
}
```


+
1
*
-
4
3
2


Eval(+)
Eval(1)
+
Eval(*)
5
1
Eval(-)
*
Eval(4)
4
4
Eval(3)
-
Eval(2)
1
3
2