

A decorative graphic on the left side of the slide, consisting of a network of light blue lines and small circles, resembling a circuit board or a tree structure, set against a blue gradient background.

# BINARY TREES

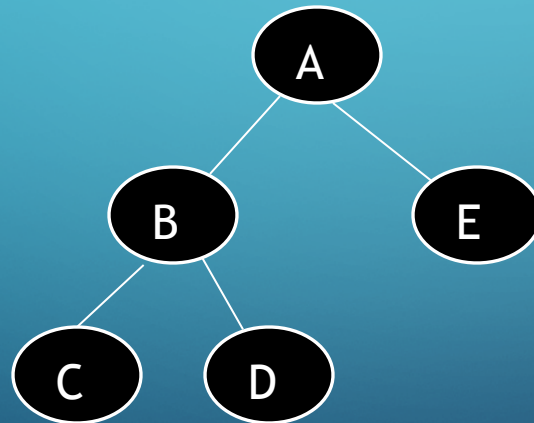
# Trees

- Trees are natural structures for representing certain kinds of hierarchical data.(How our files get saved under hierarchical directories)
- Tree is a data structure which allows us to associate a parent-child relationship between various pieces of data and thus allows us to arrange our records, data and files in a hierarchical fashion.
- Trees 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)

- 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.

#### INDEGREE AND OUTDEGREE OF A NODE:

- INDEGREE OF A NODE IS THE NUMBER OF EDGES INCIDENT ON A NODE AND OUTDEGREE IS NUMBER OF EDGES LEAVING A NODE.



- INDEGREE OF B, C, D AND E IS 1 AND THAT OF A IS 0.
- OUTDEGREE OF A AND B IS 2 AND THAT OF C, D AND E IS 0.



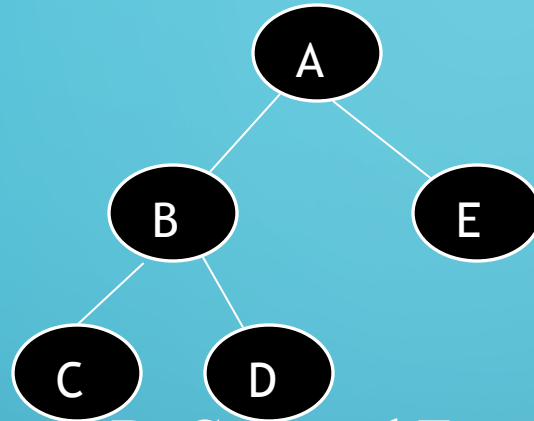
## Directed tree:

- Is a tree which has only one node with indegree 0 and all other nodes have indegree 1.
- The node with indegree 0 is called the root and all other nodes are reachable from root.

Ex: The tree in the above diagram is a directed tree with A as the root and B, C, D and E are reachable from the root.

## Ancestors and descendants of a node:

- In a tree, all the nodes that are reachable from a particular node are called the descendants of that node.



Descendants of A are B, C, D and E.

Descendants of B are C and D.

- Nodes from which a particular node is reachable starting from root are called ancestors of that node.

Ancestors of D and C are B and A.

Ancestors of E and B is A.

## Children of a node:

- The nodes which are reachable from a particular node using only a single edge are called children of that node and this node is called the father of those nodes.

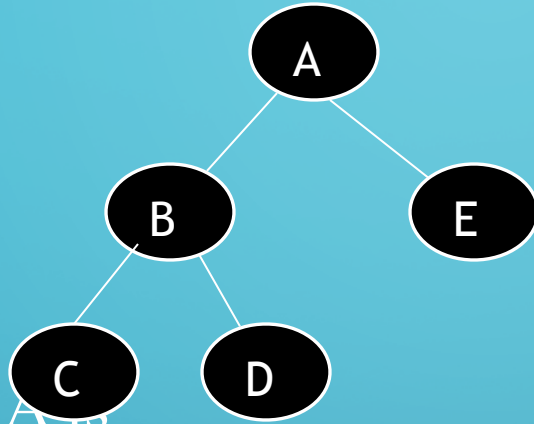
Children of A are B and E.

Children of B are C and D.

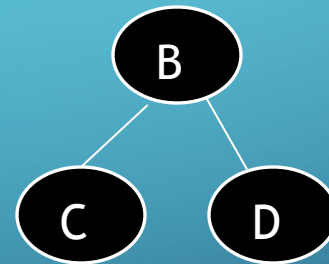


## Left subtree and right subtree of a node:

- All nodes that are all left descendants of a node form the left subtree of that node.



left subtree of A is



left subtree of B is



- All nodes that are right descendants of a node form the right subtree of that node.

right subtree of A is

E

right subtree of B is

D

right subtree of E is empty tree.



### Terminal node or leaf node:

- All nodes in a tree with outdegree zero are called the terminal nodes, leaf nodes or external nodes of the tree.
- All other nodes other than leaf nodes are called non leaf nodes or internal nodes of the tree.

In the previous tree, C, D and E are leaf nodes and A, B are non leaf nodes.

### Levels of a tree:

- Level of a node is the number of edges in the path from root node.

Level of A is 0.

Level of B and E is 1.

Level of C and D is 2.

### Height of a tree:

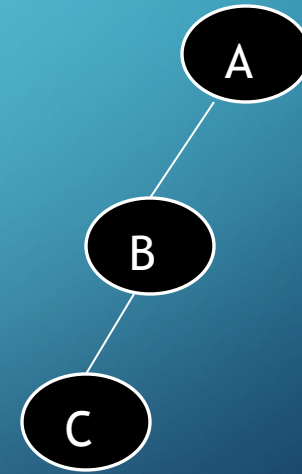
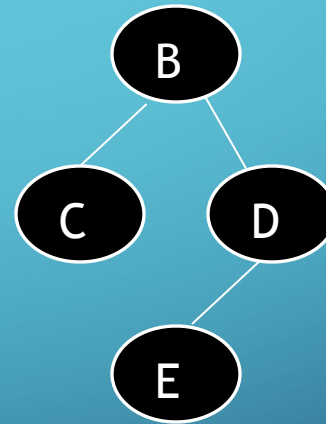
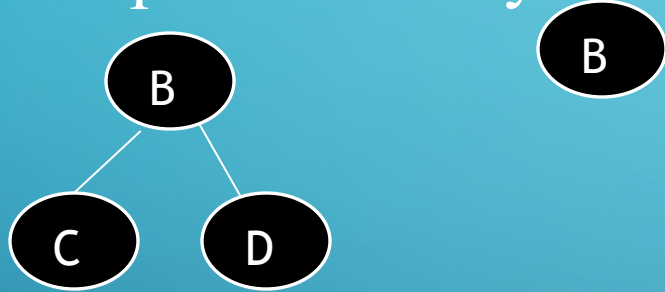
- Height of a tree is one more than the maximum level in the tree.

In the previous tree , maximum level is 2 and height is 3

## Binary trees:

- A binary tree is a directed tree in which outdegree of each node is less than or equal to 2. i.e each node can have 0, 1 or 2 children.

### Examples of binary tree

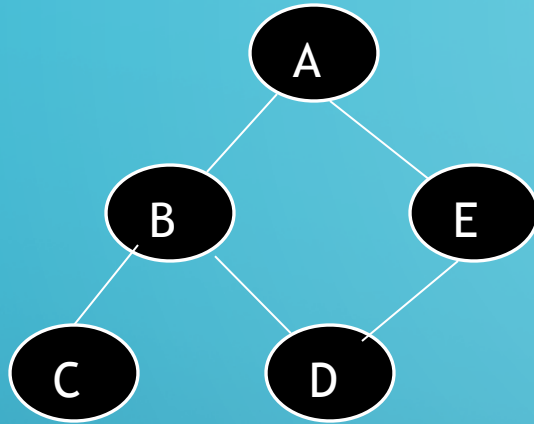


# BINARY TREE

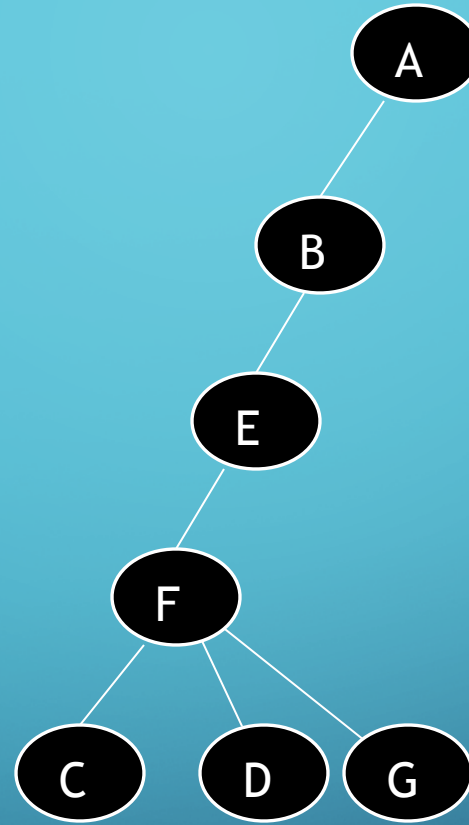
- Definition: A binary tree is a finite set of elements that is either empty or partitioned into 3 disjoint subsets. The first subset contains root of the tree and other two subsets themselves are binary trees called left and right subtree. Each element of a binary tree is called a NODE of the tree.



## Examples of non binary trees:

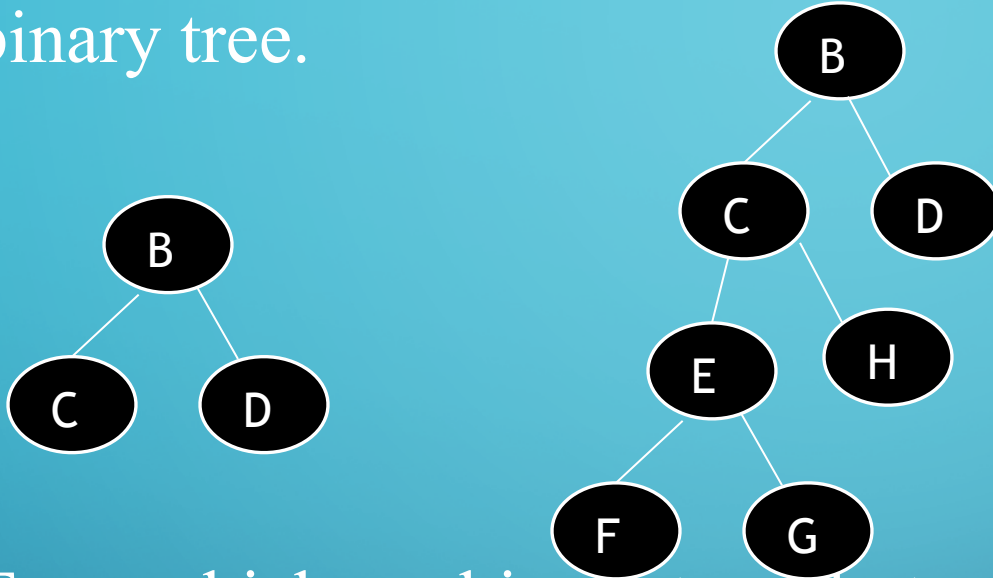


D has indegree 2,  
hence not directed  
tree

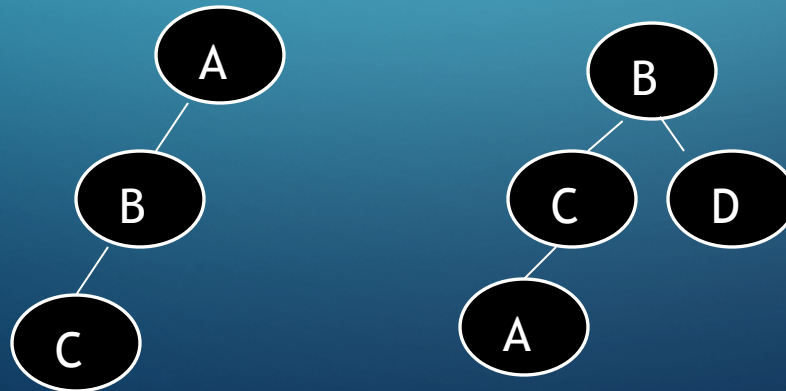


F has outdegree 3.

Strictly binary tree: If the out degree of every node in a tree is either 0 or 2 (1 not allowed), then the tree is strictly binary tree.



Trees which are binary trees but not strictly binary:

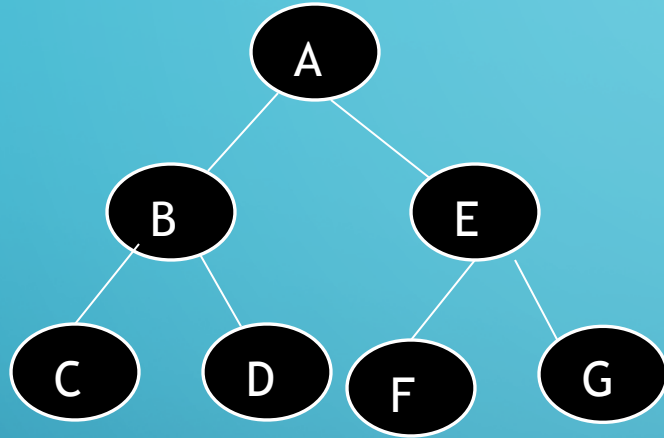


- If every non leaf node in a BT has non empty left and right subtrees ,the tree is called strictly binary tree.
- Properties
- If a SBT has  $n$  leaves then it contains  $2n-1$  nodes.
- Depth: it is the maximum level of any leaf in the tree.



## Complete binary tree:

- Is a strictly binary tree in which the number of nodes at any level 'i' is  $\text{pow}(2,i)$ .



Number of nodes at level 0(root level) is  $\text{pow}(2,0) \rightarrow 1$

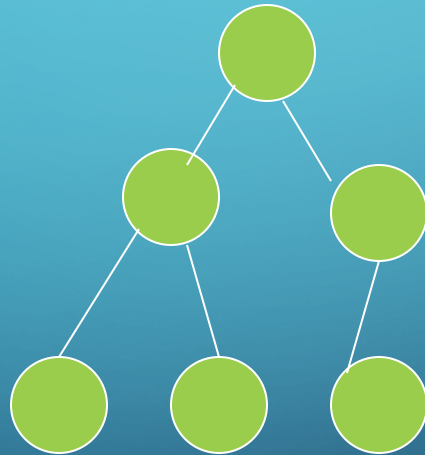
Number of nodes at level 1(B and E) is  $\text{pow}(2,1) \rightarrow 2$

Number of nodes at level 2(C,D,F and G) is  $\text{pow}(2,2) \rightarrow 4$

- A complete binary tree of depth  $d$  is the strictly binary tree all of whose leaves are at level  $d$ .
- The total number of nodes at each level between 0 and  $d$  equals the sum of nodes at each level which is equal to  $2^{d+1} - 1$
- No of non leaf nodes in that tree  $= 2^d - 1$
- No of leaf nodes  $= 2^d$

# ALMOST COMPLETE BT

- All levels are complete except the lowest
- In the last level empty spaces are towards the right.





## Storage representation of binary trees:

- Trees can be represented using
  - Linear (Array) Representation
  - Linked Representation
- In 2<sup>nd</sup> technique, node has 3 fields
  1. Info : which contains actual information.
  2. Llink :contains address of left subtree.
  3. Rlink :contains address of right subtree.

```
class Node{  
    int info;  
    Node *llink;  
    Node *rlink;  
public:  
    Node(int x=0){info = x; llink=NULL; rlink=NULL;}  
    ...  
};  
typedef Node *NODEPTR;
```

- A pointer variable root is used to point to the root node.
- Initially root is NULL, which means the tree is empty.

# ADVANTAGES AND DISADVANTAGES OF LINKED REPRESENTATION

- 

## **Advantages**

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

## **Disadvantages**

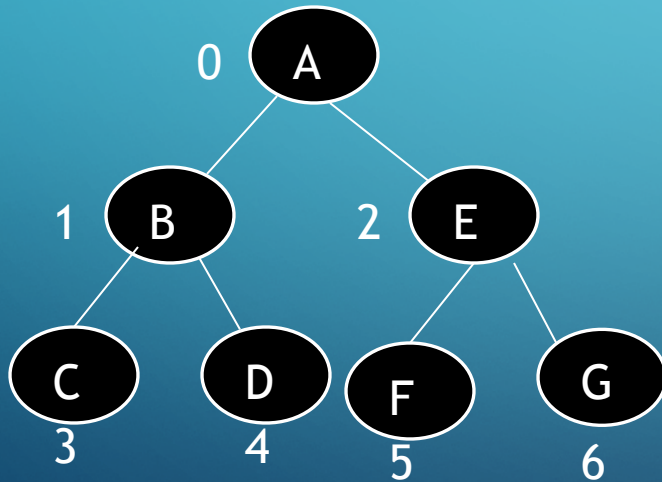
- I. It is difficult to understand.
  - II. Additional memory is needed for storing pointers
  - III. Accessing a particular node is not easy.
- 5) Define binary tree traversal and explain any one traversal with example.

## Array representation of binary tree:

A single array can be used to represent a binary tree.

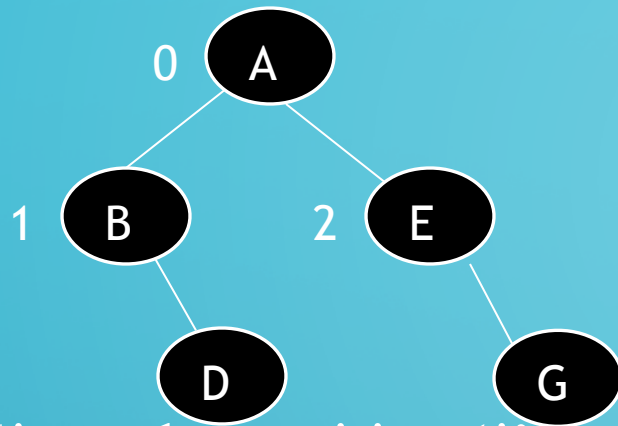
Nodes are numbered / indexed according to a scheme giving 0 to root. Then all the nodes are numbered from left to right level by level from top to bottom. Empty nodes are also numbered. Then each node having an index  $i$  is put into the array as its  $i$ th element.

In the figure shown below the nodes of binary tree are numbered according to the given scheme.



0	1	2	3	4	5	6
A	B	E	C	D	F	G





0	1	2	3	4	5	6
A	B	E		D		G

- Given the position 'i' of any node,  $2i+1$  gives the position of left child and  $2i+2$  gives the position of right child.

In the above diagram, B's position is 1.  $2*1+2$  gives 4, which is the position of its right child D.

- Given the position 'i' of any node,  $(i-1)/2$  gives the position of its father.

Position of E is 2.  $(2-1)/2$  gives 0, which is the position of its father A.

# ADVANTAGES AND DISADVANTAGES OF ARRAY REPRESENTATION

- 

## Advantages:

- I. This representation is very easy to understand.
- II. This is the best representation for complete and almost complete binary tree representation.
- III. Programming is very easy.
- IV. It is very easy to move from a child to its parents and vice versa.

## Disadvantages:

- I. Lot of memory area wasted.
- II. Insertion and deletion of nodes needs lot of data movement.
- III. Execution time high.
- IV. This is not suited for trees other than full and complete tree.

## Operations on binary trees:

Insertion: inserting an item into the tree.

Traversal: visiting the nodes of the tree one by one.

Search: search for the specified item in the tree.

Copy: to obtain exact copy of given tree.

Deletion: delete a node from the tree.

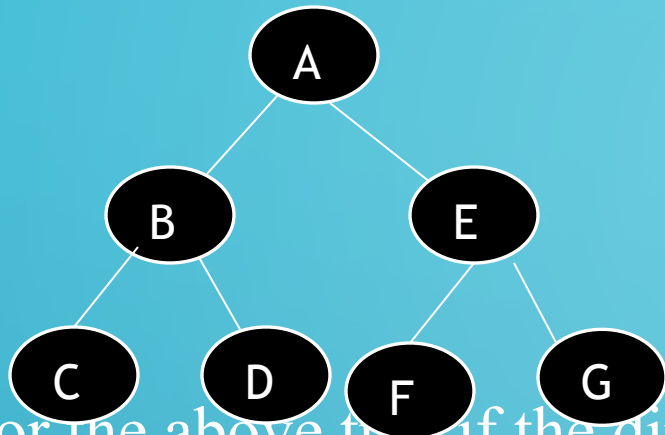
```
class Bintree{  
    NODEPTR root;  
public:  
    Bintree(){root = NULL;}  
    NODEPTR Create(int);  
    void CreateBinTree();  
    void Inorder(NODEPTR );  
    void Preorder(NODEPTR );  
    void Postorder(NODEPTR );  
    void Levelorder();  
    NODEPTR insert(int);  
    ...  
};
```



## Insertion:

- 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 prev and cur. At any point prev points to the parent node and cur points to the child.



```
/*function to insert item into tree*/
```

```
NODEPTR insert(int item)
```

```
{
```

```
    NODEPTR cur, prev;
```

```
    char direction[10];
```

```
    int i;
```

```
    NODEPTR temp= new NODE(item);
```

```
    if(root==NULL)
```

```
        return temp;
```

```
    cout<<“enter the direction of insertion”;
```

```
    cin>>direction;
```

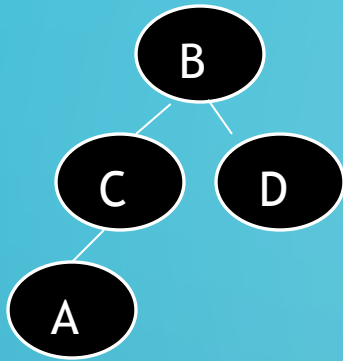




```
prev=NULL;
cur=root;
for(i=0;i<strlen(direction)&&cur!=NULL; i++)
{
    prev=cur;                                /*keep track of parent*/
    if(direction[i]=='L')                    /*if direction is L, move left of
        cur=cur->llink;                      current node*/
    else                                    /*if direction is R, move right of
        cur=cur->rlink;                      current node*/
}
If(cur!=NULL || i!=strlen(direction))
{
    cout<<"insertion not possible";
    delete temp;
    return root;
}
```

```
if (direction[i-1]=='L')      /* if last direction is 'L', point the
    prev→llink=temp;         llink of parent to new node*/
else                          /* else point rlink of parent to new
    prev→rlink=temp;         node*/
return root;
}
```

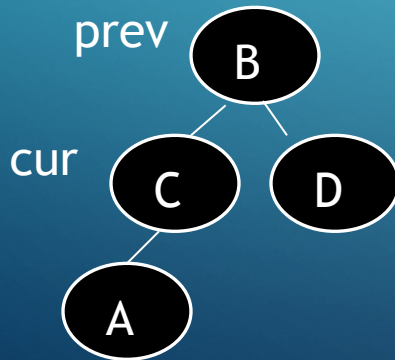
- Control comes out of for loop, if  $i = \text{strlen}(\text{direction})$  or when  $\text{cur} == \text{NULL}$ .
- For insertion to be possible, control should come out when  $\text{cur} == \text{NULL}$  and  $i = \text{strlen}(\text{direction})$  both at the same time.  
i.e when we get the position to insert ( $\text{cur} == \text{NULL}$ ), the string should be completed.



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

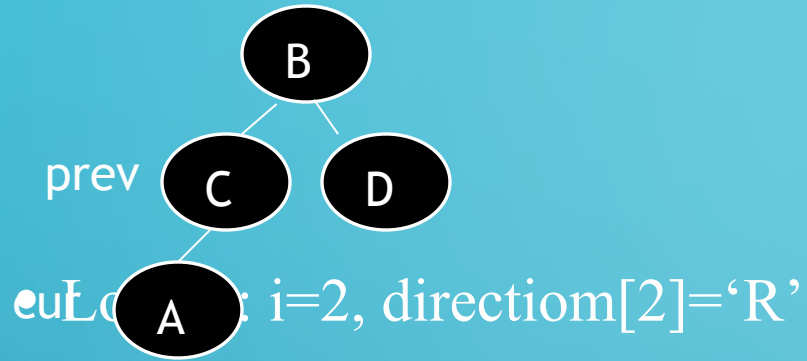
Initially  $cur == root$  and  $prev == NULL$

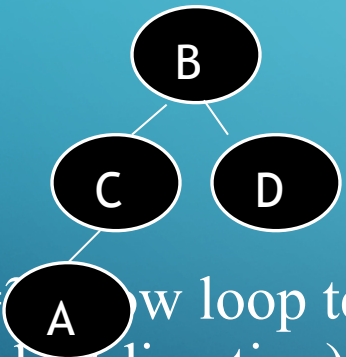
- Loop 1:  $i=0$  ,  $direction[0]='L'$





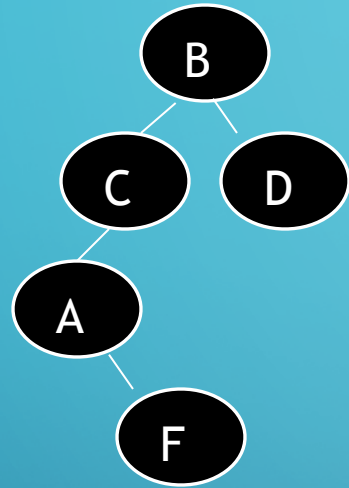
- Loop 2:  $i=1$ ,  $\text{direction}[1]='L'$



-  Now loop terminates since  $\text{cur}==\text{NULL}$ . Here  $i==3$  (i.e.  $\text{strlen}(\text{direction})$ ) and  $\text{cur}==\text{NULL}$ . Hence insertion is possible

- Now  $\text{direction}[i-1]$  is  $\text{direction}[3-1]$  which gives 'R'.
- Hence  $\text{prev} \rightarrow \text{rlink}$  is made to point to new node.

After insertion tree looks like



- If direction of insertion is “LLR” for the above tree. Here for loop terminates when  $i == \text{strlen}(\text{direction})$  i.e 3. But at this point cur is not equal to NULL. Hence insertion is not possible.

## Traversals:

- Is visiting each node exactly once systematically one after the another.

There are 3 main traversals techniques

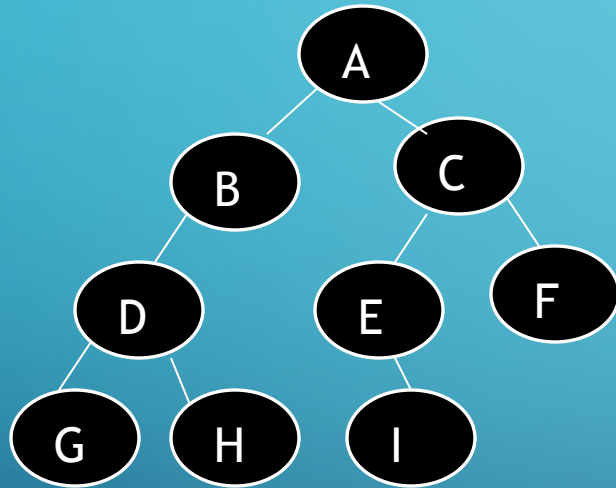
- Inorder traversal
- Preorder traversal
- Postorder traversal

### 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.



- In 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.



Here when we 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.

- Now 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.
- Next go to the right of D, which is H. Now another root H is visited.
- Now 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.
- Next 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.
- Now traversing of left is finished and so move towards right of it and 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.

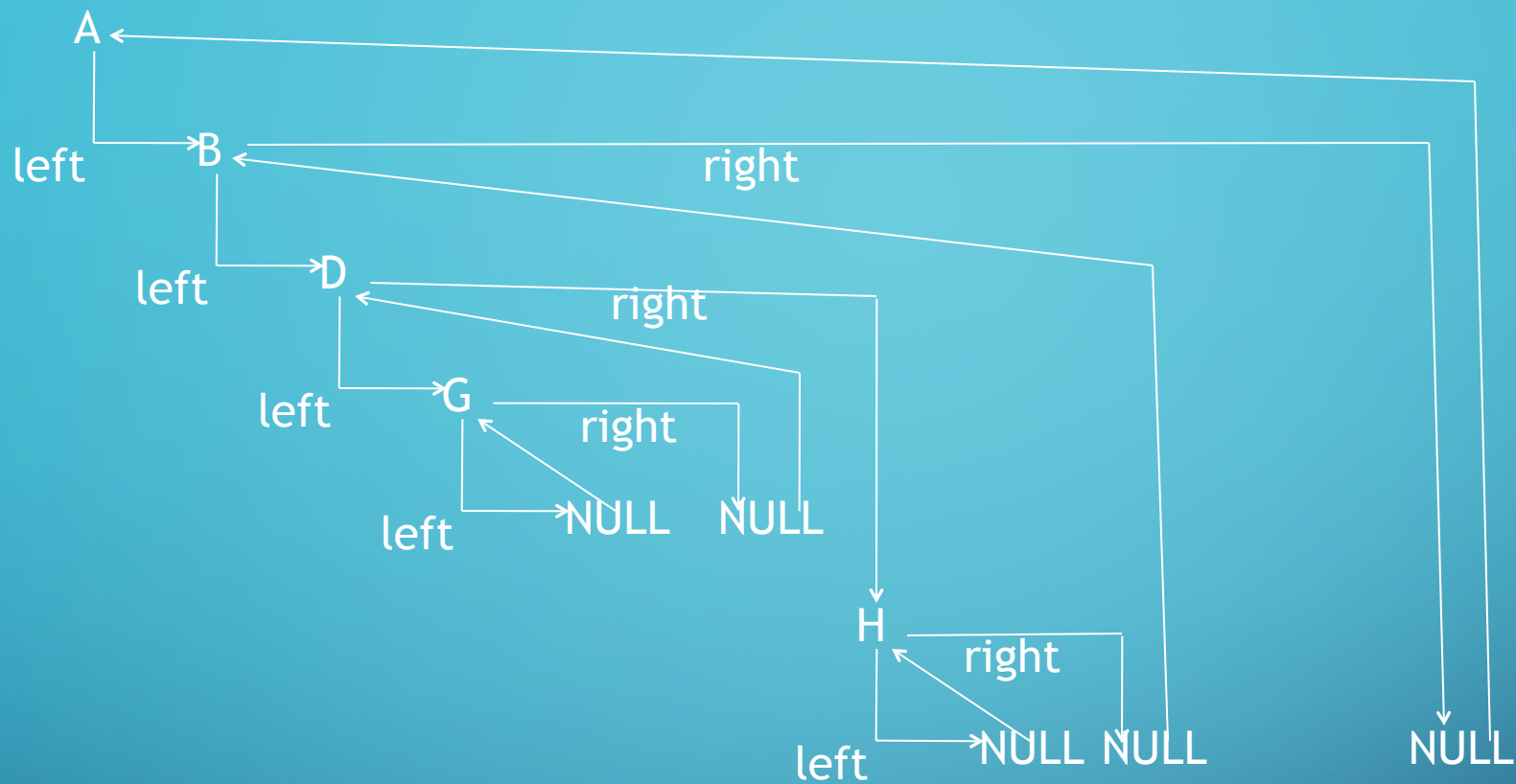
- Move to left of I, which is NULL. Hence go back to root I, print it and move to its right, which is NULL.
- Go back to root C, print it and go to its right and reach F.
- Move to left of F, which is NULL. Hence go back to F, print it and go to its right, which is also NULL.
- Traversal ends at this point.

Hence inorder traversal of above tree gives GDHBAEICF

*/\*recursive algorithm for inorder traversal\*/*

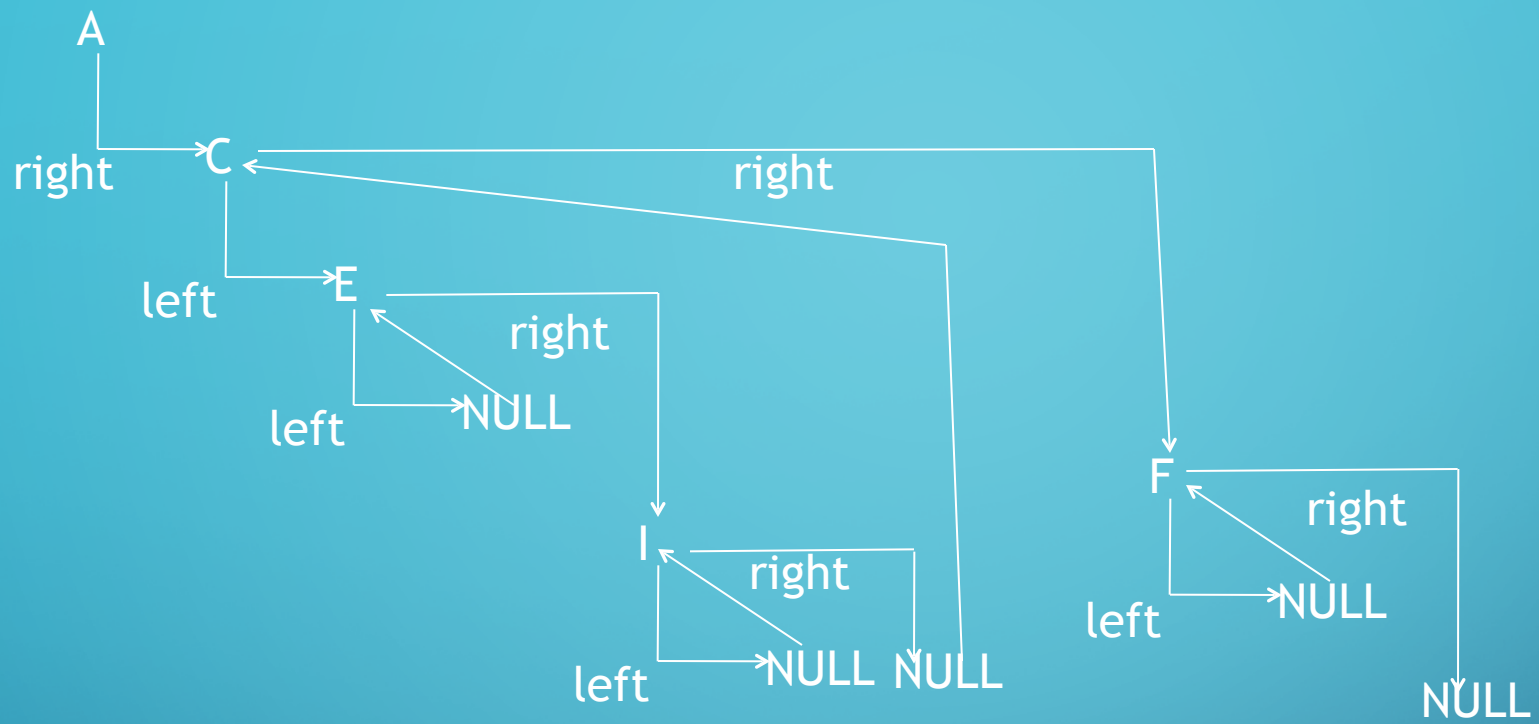
```
void inorder(NODEPTR root)
{
    if(root!=NULL)
    {
        inorder(root->llink);
        cout<<root->info;
        inorder(root->rlink);
    }
}
```





G D H B A

Traversing left subtree of A inorder



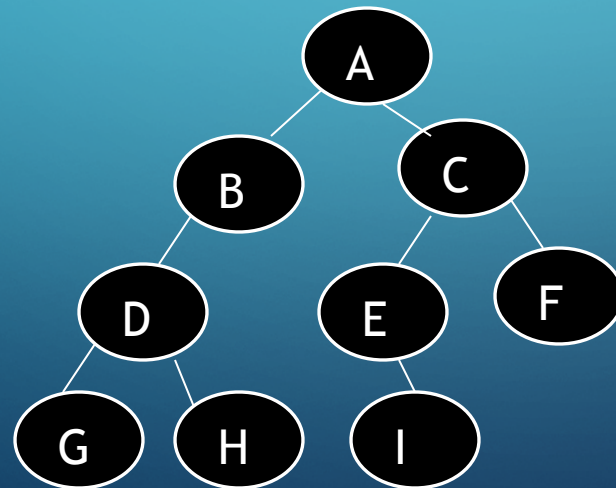
E I C F

Traversing right subtree of A inorder

## Preorder traversal:

Preorder traversal is defined as

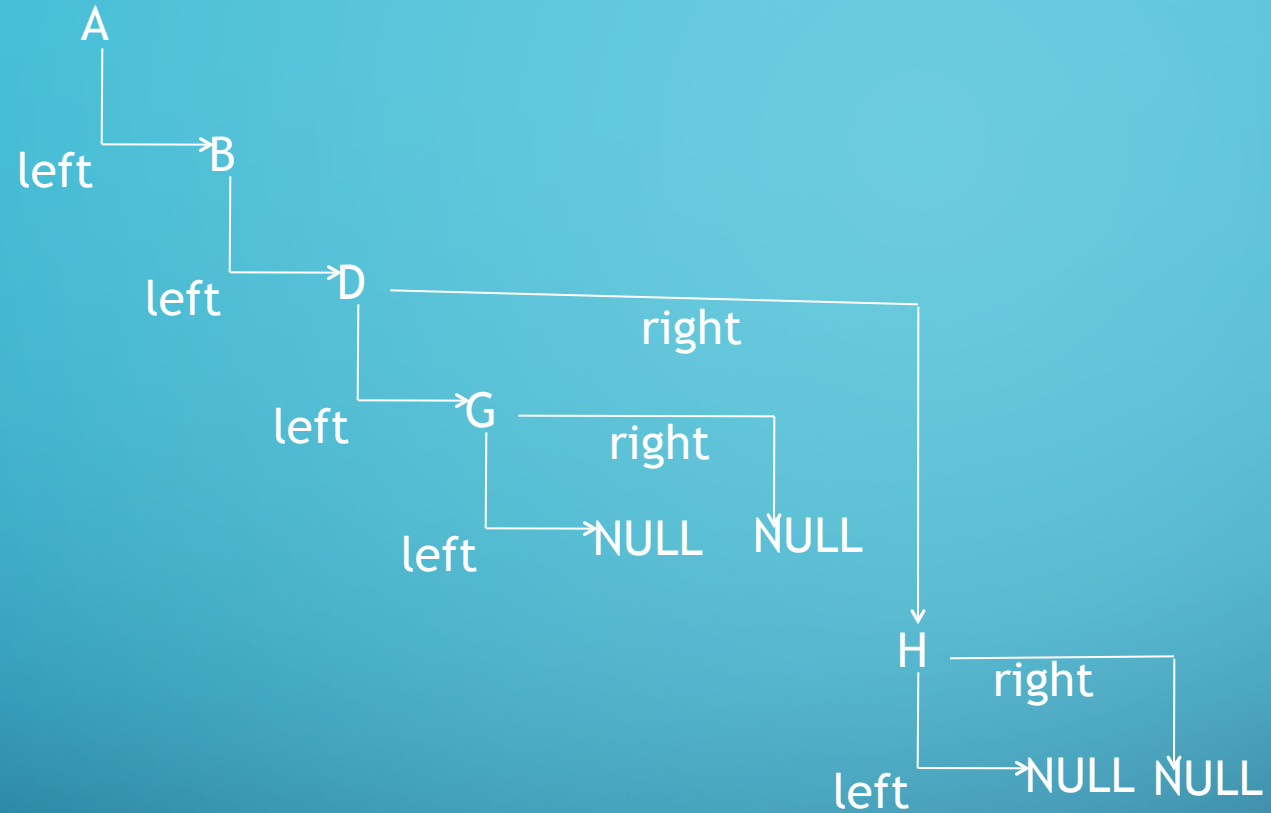
1. Process the node.
  2. Traverse the left subtree in preorder.
  3. Traverse the right subtree in preorder.
- In preorder, we first visit the node, then move towards left and then to the right recursively.



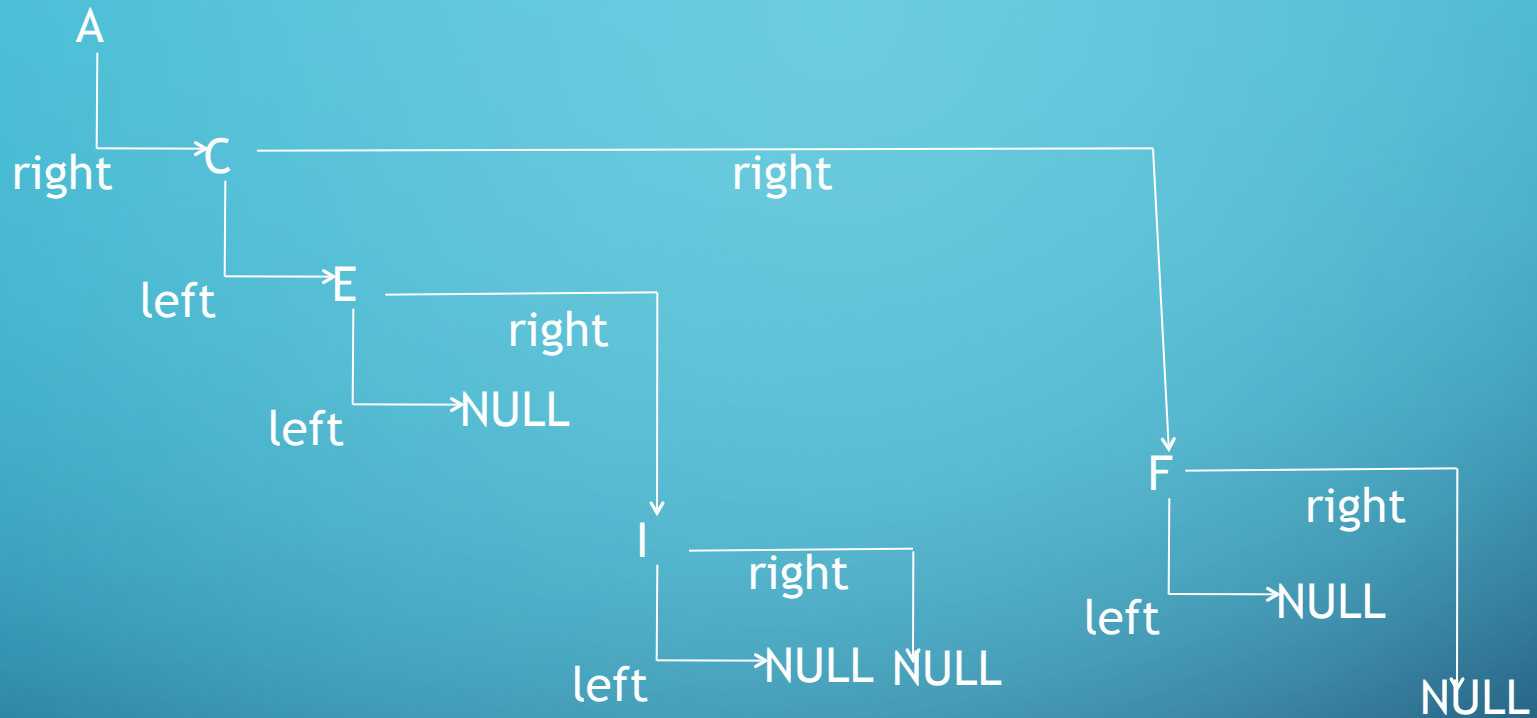


```
void preorder(NODEPTR root)
{
    if(root!=NULL)
    {
        cout<<root->info;
        preorder(root->llink);
        preorder(root->rlink);
    }
}
```

## Traversing left subtree in preorder



## Traversing right sub tree in preorder



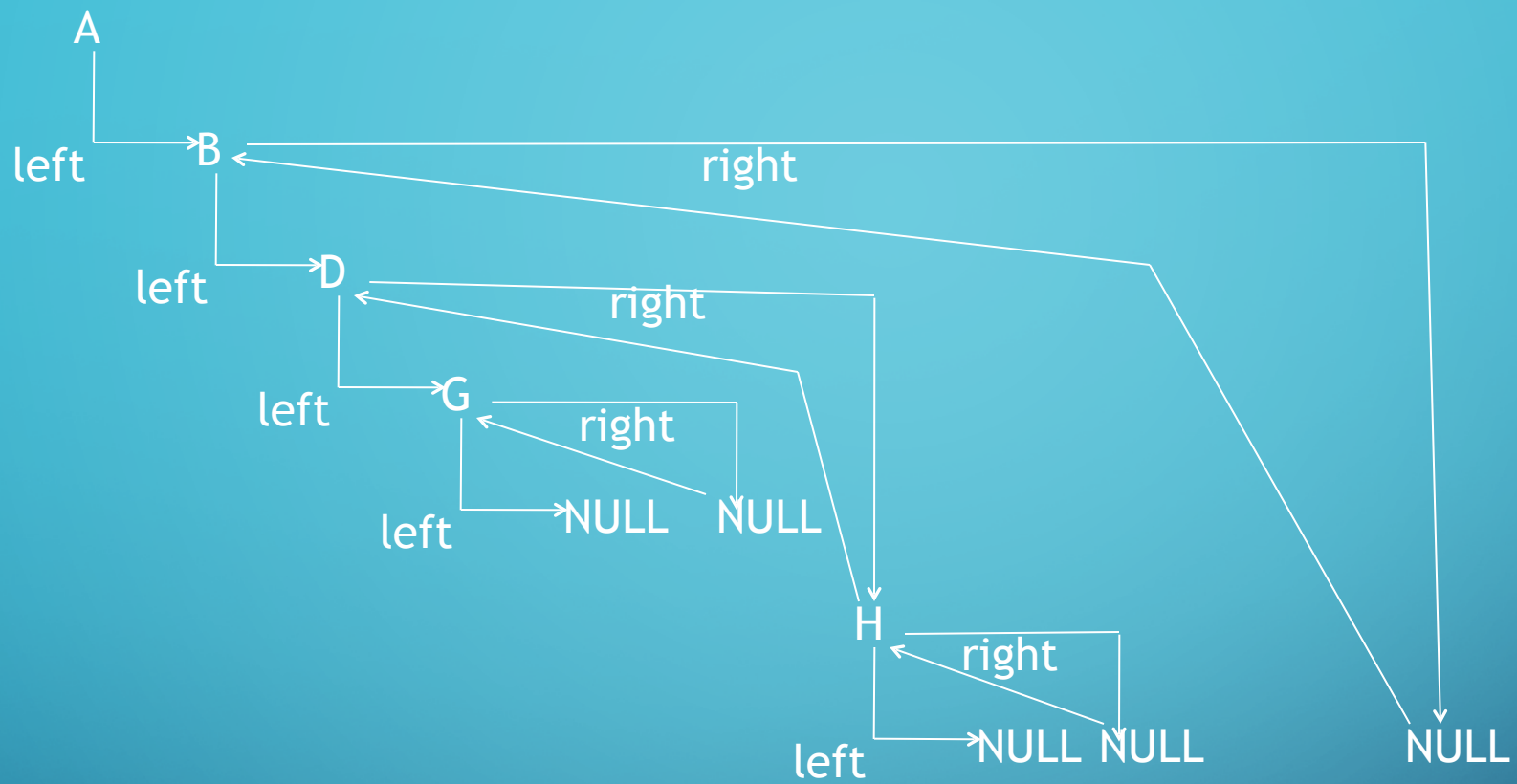


## Post order traversal

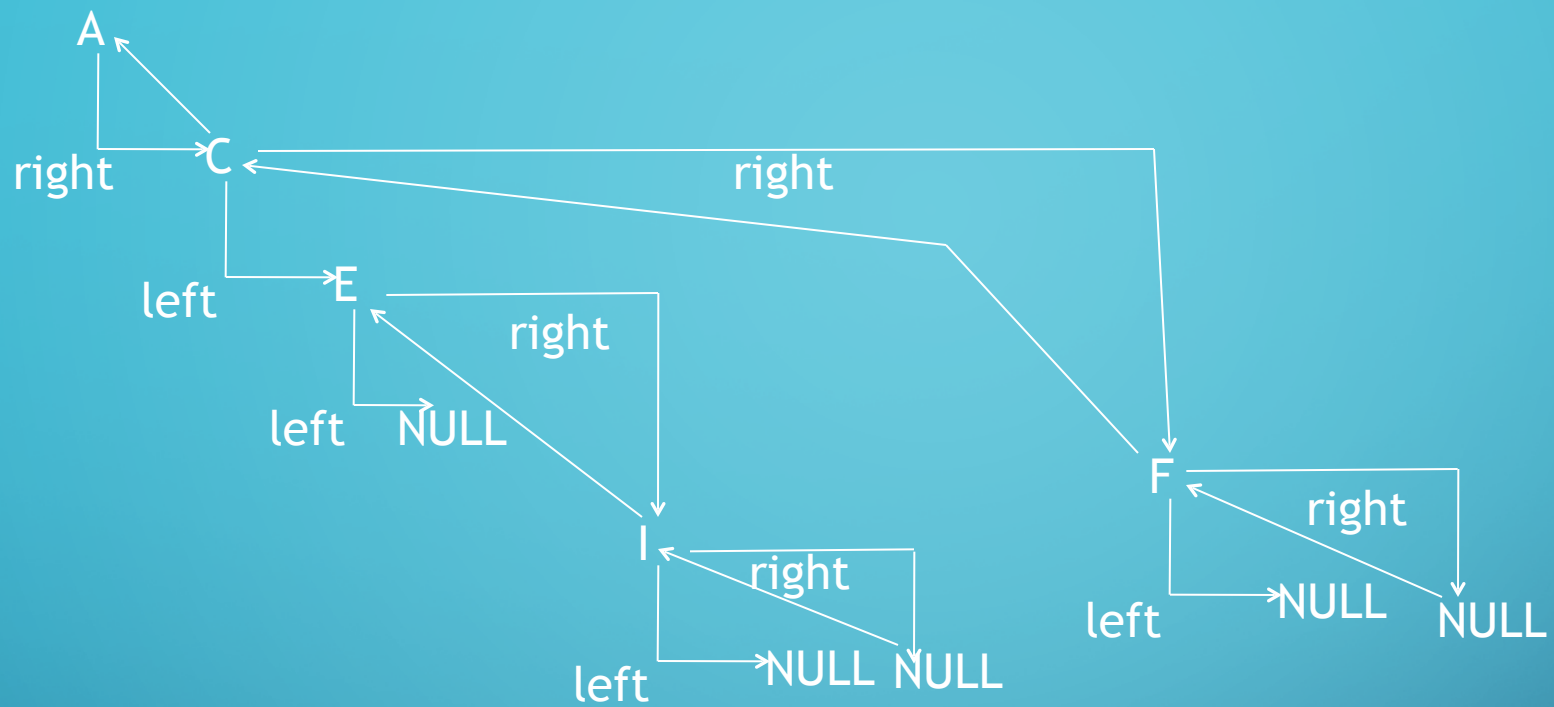
Post order traversal is defined as

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.

```
void postorder(NODEPTR root)
{
    if(root!=NULL)
    {
        postorder(root->llink);
        postorder(root->rlink);
        cout<<root->info;
    }
}
```



Traversing left subtree of A postorder



I E F C A

Traversing right subtree of A in postorder



## 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.
- flag is a reference variable and set to 0 initially (before call in main)

```
void search(int item, NODEPTR root, int &flag){
```

```
    if(root!=NULL)
```

```
    {
```

```
        if(item==root→info)
```

```
        {
```

```
            flag=1
```

```
            return;
```

```
        }
```

```
        search(item, root→llink, flag);
```

```
        if (!(flag)) search(item, root→rlink, flag);
```

```
    }
```

```
}
```

## Copying a tree:

- Getting the exact copy of the given tree.

/\*recursive function to copy a tree\*/

NODEPTR copy (NODEPTR root)

{

if(root == NULL)

return NULL;

NODEPTR temp = new NODE(root->info);

temp->llink=copy(root->llink);

temp->rlink=copy(root->rlink);

return temp;

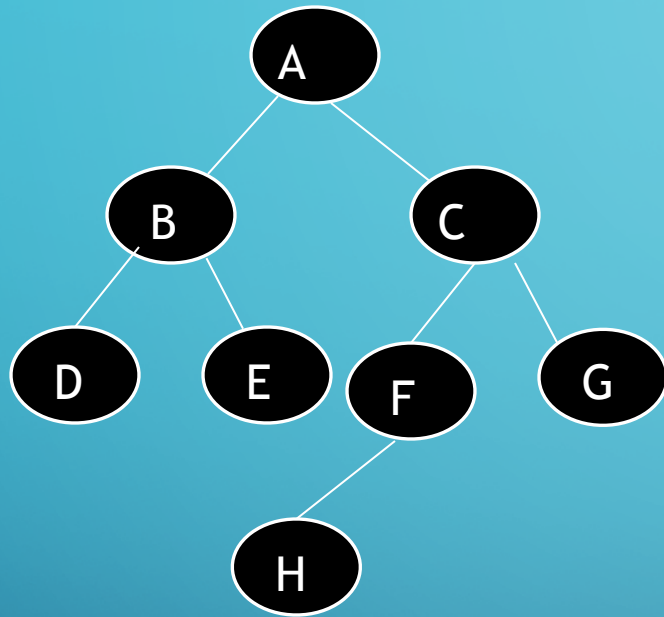
}

/\*recursive function to find the height of a tree\*/

```
Int height (NODEPTR root)
{
    if(root==NULL)
        return 0;
    return( 1+ max(height (root→llink), height(root→rlink)));
}
```



## Finding the height of the tree using recursion

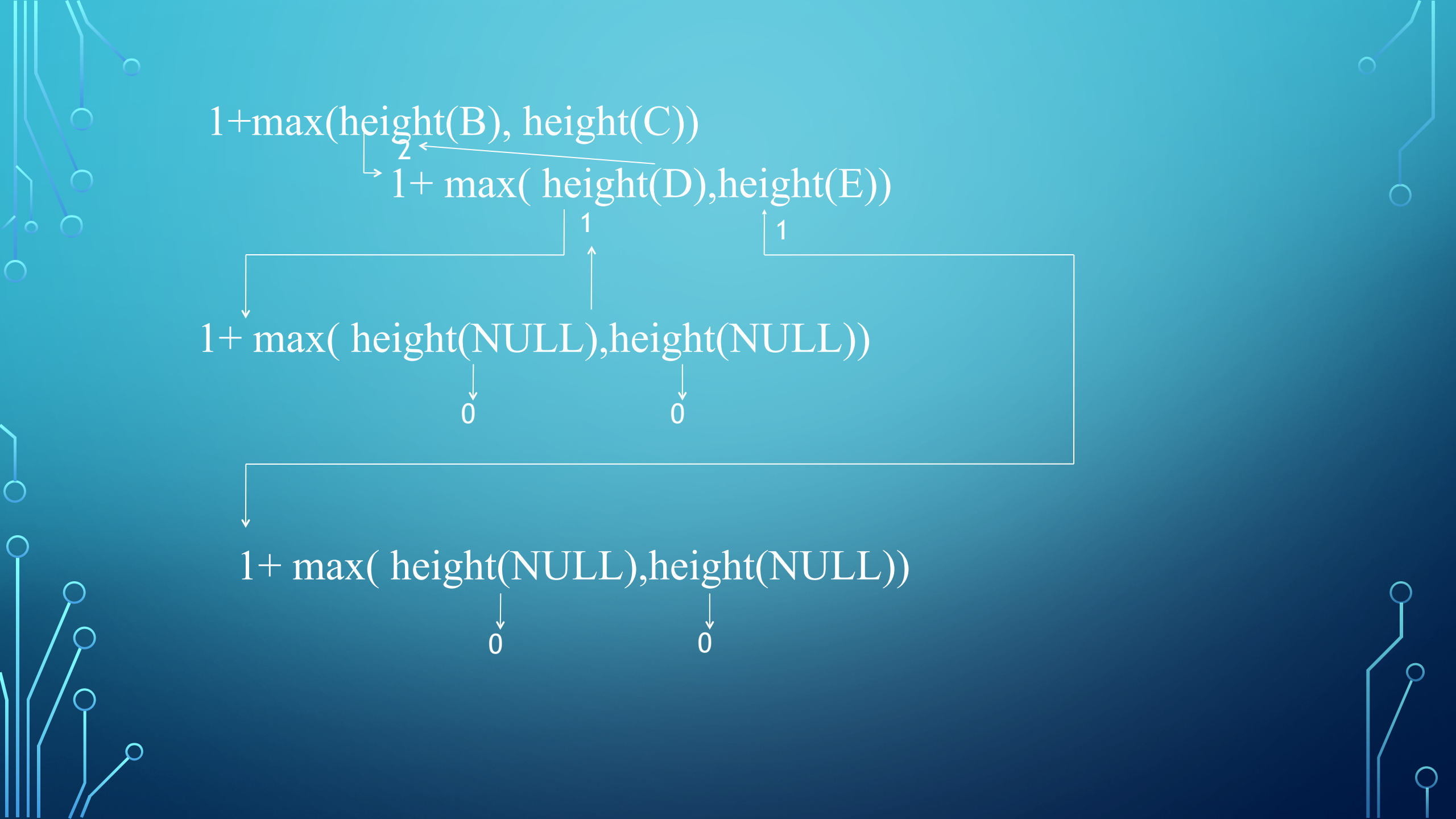


$1 + \max(\text{height}(B), \text{height}(C))$

$1 + \max(\text{height}(D), \text{height}(E))$

$1 + \max(\text{height}(\text{NULL}), \text{height}(\text{NULL}))$

$1 + \max(\text{height}(\text{NULL}), \text{height}(\text{NULL}))$



Height(C)

$$3 \leftarrow 1 + \max(\text{height}(F), \text{height}(G))$$

$$1 + \max(\text{height}(H), \text{height}(\text{NULL}))$$

1

0

$$1 + \max(\text{height}(\text{NULL}), \text{height}(\text{NULL}))$$

Finally,

$1 + \max(\text{height}(B), \text{height}(C)) \rightarrow 1 + \max(2, 3) \rightarrow 4$ , which is the height of the 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.

*/\*counting number of nodes using inorder technique\*/*

```
Void count_nodes( NODEPTR root)
```

```
{  
    if(root!=NULL)  
    {  
        count_nodes(root→llink);  
        count++;  
        count_nodes(root→rlink);  
    }  
}
```

### Counting the number of leaf nodes in a tree:

- Traverse the tree in any of the 3 techniques and 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\*/*

```
Void count_leafnodes( NODEPTR root)
```

```
{
```

```
    if(root!=NULL)
```

```
    {
```

```
        count_leafnodes(root→llink);
```

```
        if(root→llink==NULL && root→rlink==NULL)
```

```
            count++;
```

```
        count_leafnodes(root→rlink);
```

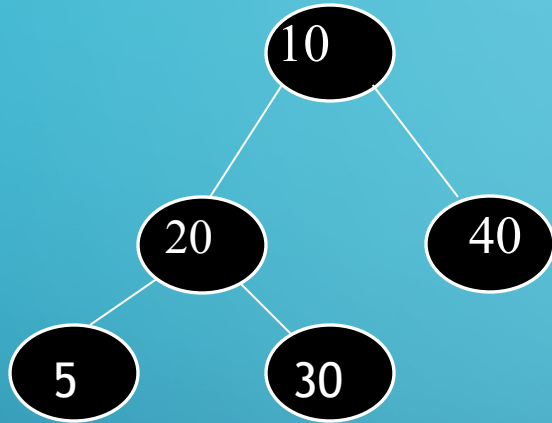
```
    }
```

```
}
```

## Iterative traversals of binary tree

### Iterative preorder traversal:

- Every time a node is visited, its info is printed and node is pushed to stack, since the address of node is needed to traverse to its right later.



Here 10 is printed and this node is pushed onto stack and then move left to 20. This is done because we need to go right of 10 later.

Similarly 20, 5, 30 and 40 are moved to stack



- Once traversing the left side is over, pop the most pushed node and go to its right.

In the previous tree, after 5 is printed, recently pushed node 20 is popped and move right to 30.



```
/*function for iterative preorder traversal*/
```

```
Void preorder(NODEPTR root)
```

```
{
```

```
    NODEPTR cur;
```

```
    stack<NODEPTR>      s; //stack of NODEPTR
```

```
    if(root==NULL)
```

```
    {
```

```
        cout<<"tree is empty";
```

```
        return;
```

```
    }
```





```
cur=root;
```

```
for(;;)
```

```
{
```

```
while(cur!=NULL)
```

```
{
```

```
    cout<<cur->info<<endl;
```

```
    s.push(cur)           /*push the node to stack*/
```

```
    cur=cur->llink;
```

```
}
```

```
if(!s.IsEmpty())          /*more nodes existing*/
```

```
{
```

```
    cur=s.pop();           /* pop most recent node*/
```

```
    cur=cur->rlink;        /*traverse right*/
```

```
}
```

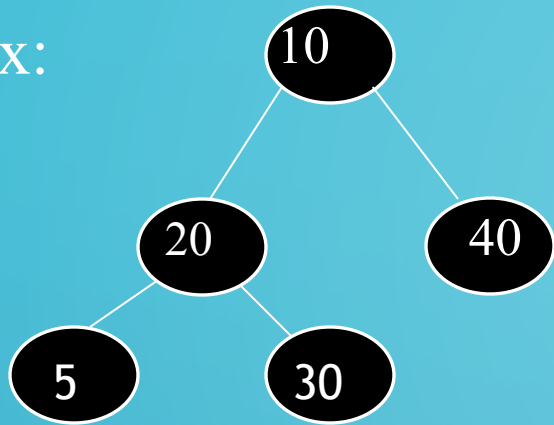
```
else return;
```

```
}}
```





Ex:

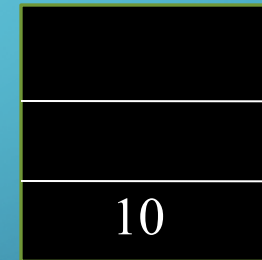


1. After 1<sup>st</sup> iteration of while loop

10 is printed

Node 10 is pushed to stack

Cur=cur→llink; i.e Cur=20

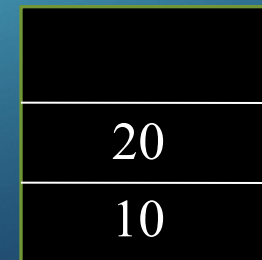


2. After 2<sup>nd</sup> iteration of while loop

20 is printed

Node 20 is pushed to stack

Cur=cur→llink; i.e Cur=5



3. After 3<sup>rd</sup> iteration of while loop

5 is printed

Node 5 is pushed to stack

Cur=cur→llink; i.e Cur=NULL

5
20
10

4. While loop terminates since cur==NULL

Stack is not empty

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

cur=cur→rlink; i.e cur=NULL

20
10

5. cur==NULL, while loop not entered

Stack not empty

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

cur=cur→rlink; i.e cur=30

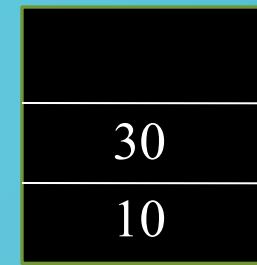
10

6. While loop is entered

30 is printed

Node 30 is pushed to stack

Cur=cur→llink; i.e cur=NULL

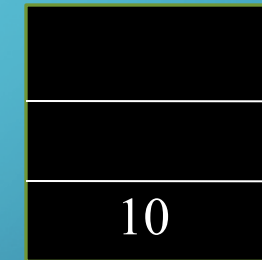


7. While loop terminates since cur==NULL

Stack not empty

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

cur=cur→rlink; i.e cur=NULL

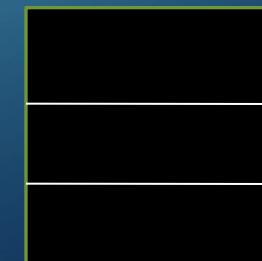


8. cur==NULL, while loop not entered

Stack not empty

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

cur=cur→rlink; i.e cur=40



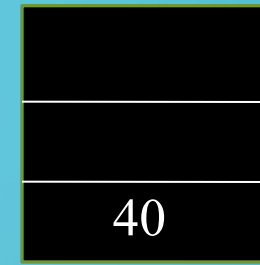


9. While loop is entered

40 is printed

Node 40 is pushed to stack

cur=cur→llink; i.e cur=NULL

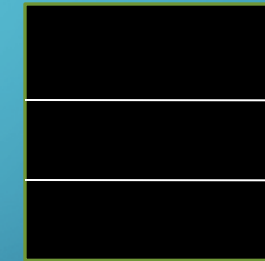


10. While loop terminates since cur==NULL

Stack not empty

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

cur=cur→rlink; i.e cur=NULL



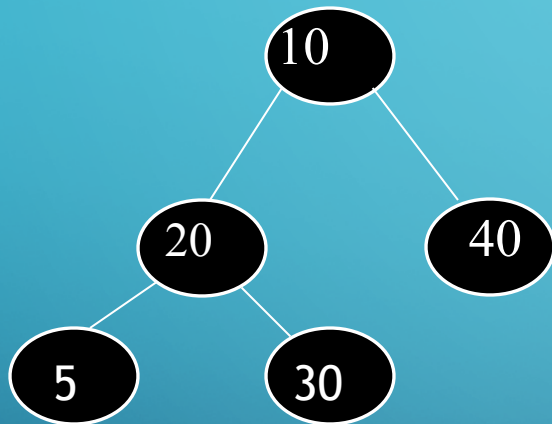
11. cur==NULL and stack empty

return

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

### 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



```
/*function for iterative inorder traversal*/
```

```
Void inorder(NODEPTR root)
```

```
{
```

```
    NODEPTR cur;
```

```
    stack<NODEPTR> s;
```

```
    if(root==NULL)
```

```
    {
```


```
        cout<<"tree is empty";
```

```
        return;
```


```
    }
```



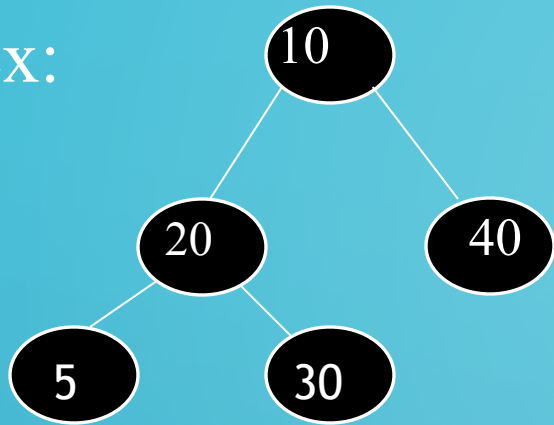




```
cur=root;
for(; ;)
{
    while(cur!=NULL)
    {
        s.push(cur);           /*push the node to stack*/
        cur=cur->llink;
    }
    if(!s.IsEmpty())           /*more nodes existing*/
    {
        cur=s.pop();           /* pop most recent node*/
        cout<<cur->info<<endl;
        cur=cur->rlink;         /*traverse right*/
    }
    else return;
}}
```



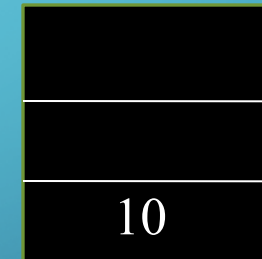
Ex:



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

Node 10 is pushed to stack

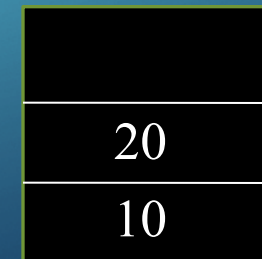
Cur=cur→llink; i.e Cur=20



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

Node 20 is pushed to stack

Cur=cur→llink; i.e Cur=5



3. After 3<sup>rd</sup> iteration of while loop

Node 5 is pushed to stack

Cur=cur→llink; i.e Cur=NULL

5
20
10

4. While loop terminates since cur==NULL

Stack is not empty

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

Print 5

cur=cur→rlink; i.e cur=NULL

20
10

5. cur==NULL, while loop not entered

Stack not empty

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

Print 20

cur=cur→rlink; i.e cur=30

10



## 6. While loop is entered

Node 30 is pushed to stack

Cur=cur→llink; i.e cur=NULL

30
10

## 7. While loop terminates since cur==NULL

Stack not empty

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

Print 30

cur=cur→rlink; i.e cur=NULL

10

## 8. cur==NULL, while loop not entered

Stack not empty

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

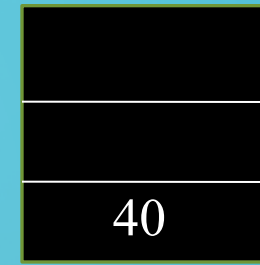
Print 10

cur=cur→rlink; i.e cur=40


9. While loop is entered

Node 40 is pushed to stack

cur=cur→llink; i.e cur=NULL



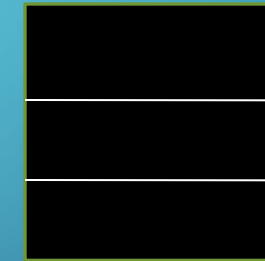
10. While loop terminates since cur==NULL

Stack not empty

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

Print 40

cur=cur→rlink; 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 if 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.





```
/*function for iterative postorder traversal*/
```

```
Void postorder(NODEPTR root)
```

```
{
```

```
    struct record
```

```
    {
```

```
        NODEPTR node;
```

```
        int flag;
```

```
    };
```

```
    record r;
```

```
    NODEPTR cur;
```

```
    stack<record> s;
```

```
    if(root==NULL)
```

```
    {
```

```
        cout<<"tree is empty";
```

```
        return;
```

```
    }
```

```
cur=root;
for(; ;)
{
    while(cur!=NULL)
    {
        r.node = cur;
        r.flag=1;
        s.push(r);
        cur=cur->llink;
    }
    while(s.checkflag()<0)
    {
        r =s.pop();
        cur=r.node;
        cout<<cur->info<<endl;
        if(s.IsEmpty())
            return;
    }
}
```

/\*traverse left of tree and push  
the nodes to the stack and  
s[top].flag=1;  
set flag to 1\*/

// s[top].flag<0

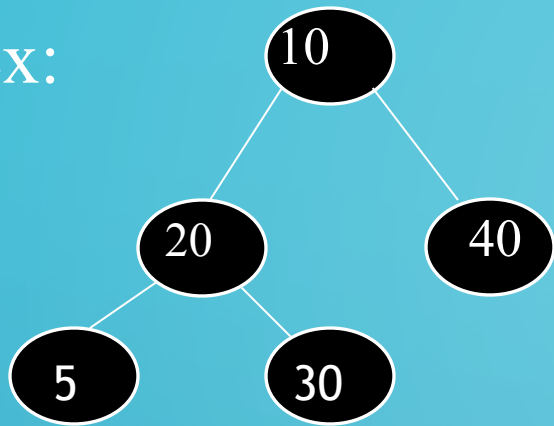
/\*if flag is -ve, right subtree  
is visited and hence node  
is popped and printed\*/

/\*if stack is empty, traversal  
is complete\*/

```
cur= s.topnode(); // s[top].node;    /*after left subtree is  
cur=cur→rlink; // traversed, move to right and set its flag  
           //to -1 to indicate right subtree is traversed  
s.setflag(); //s[top].flag=-1;  
  
}  
  
}
```



Ex:



Initially cur = 10;

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

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

Cur=cur→llink; i.e Cur=20

10	1

2. After 2<sup>nd</sup> iteration of 1<sup>st</sup> while loop

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

Cur=cur→llink; i.e Cur=5

20	1
10	1

3. After 3<sup>rd</sup> iteration of 1<sup>st</sup> while loop  
Node 5 is pushed to stack & its flag set to 1.  
Cur=cur→llink; i.e Cur=NULL

5	1
20	1
10	1

4. While loop terminates since cur==NULL  
Since s.checkflag() !=-1, 2<sup>nd</sup> while not entered.  
cur= s.topnode() ;//s[top].node; i.e cur=node 5;  
Cur=cur→rlink; i.e cur=NULL;  
s.setflag(); //s[top].flag =-1

5	-1
20	1
10	1

5. cur==NULL, 1<sup>st</sup> while loop not entered  
Since s.checkflag()<0, 2<sup>nd</sup> while is entered.  
r=s.pop();  
cur=r.node; i.e cur=node 5;  
Print 5  
Stack is not empty, continue;

20	1
10	1

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

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

Cur=cur→rlink; i.e cur=30;

s[top].flag = -1

20	-1
10	1

7. 1<sup>st</sup> while is entered

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

Cur=cur→llink; i.e Cur=NULL

30	1
20	-1
10	1

8. cur==NULL, 1<sup>st</sup> while loop exits

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

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

cur=cur→rlink; i.e cur=NULL;

s[top].flag = -1

30	-1
20	-1
10	1



9. cur==NULL, 1<sup>st</sup> while loop not entered

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

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

Print 30

Stack is not empty, continue;

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

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

Print 20

Stack is not empty, continue;

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

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

Cur=cur→rlink; i.e cur=40;

s[top].flag =-1

20	-1
10	1

10	1

10	-1

12. 1<sup>st</sup> while is entered

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

Cur=cur→llink; i.e Cur=NULL

13. cur==NULL, 1<sup>st</sup> while loop exits

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

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

cur=cur→rlink; i.e cur=NULL;

s[top].flag =-1

14. cur==NULL, 1<sup>st</sup> while loop not entered

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

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

Print 40

Stack is not empty, continue;

40	1
10	-1

40	-1
10	-1

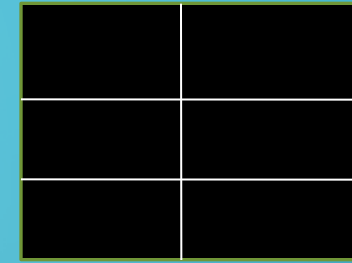
10	-1

15.  $s[\text{top}].\text{flag} < 0$ , 2<sup>nd</sup> While loop continues

$\text{cur} = s[\text{top}].\text{node}$ ; i.e  $\text{cur} = \text{node } 10$ ;

Print 10

Stack is empty, stop;



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