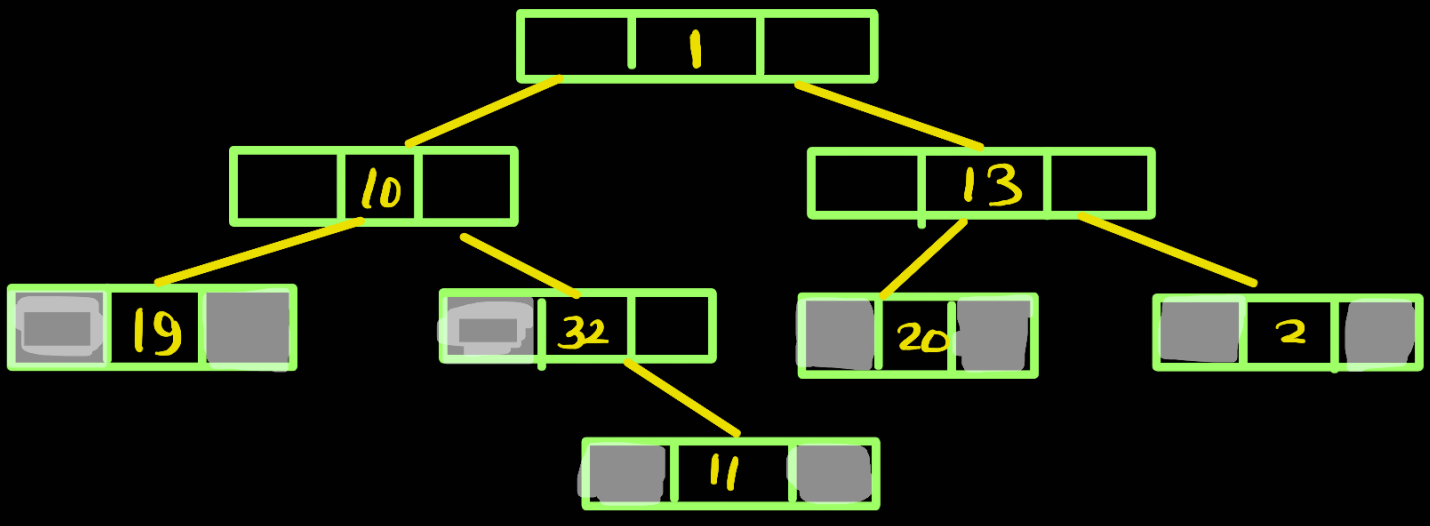
Threaded Binary Tree.

Need of Threaded Binary Tree.



↳ Majority (n+1) Pointers are NULL.

↳ In above tree we have 8 nodes and 9 empty pointers.

↳ Also we need large storage such as stack or Queue for traversal.

So, to overcome these problems we have threaded Binary Tree.

Advantages of Threaded Binary Tree.

1. Store Meaningful information in these NULL Pointers.

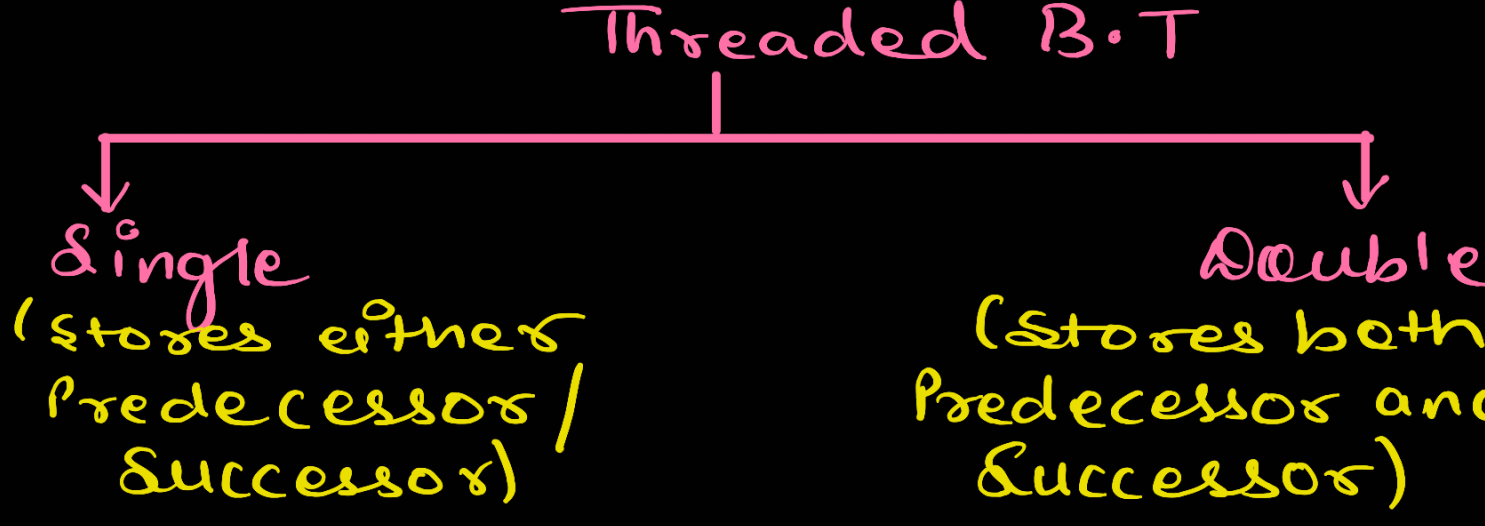
2. Eliminate use of stack and queue for traversal.

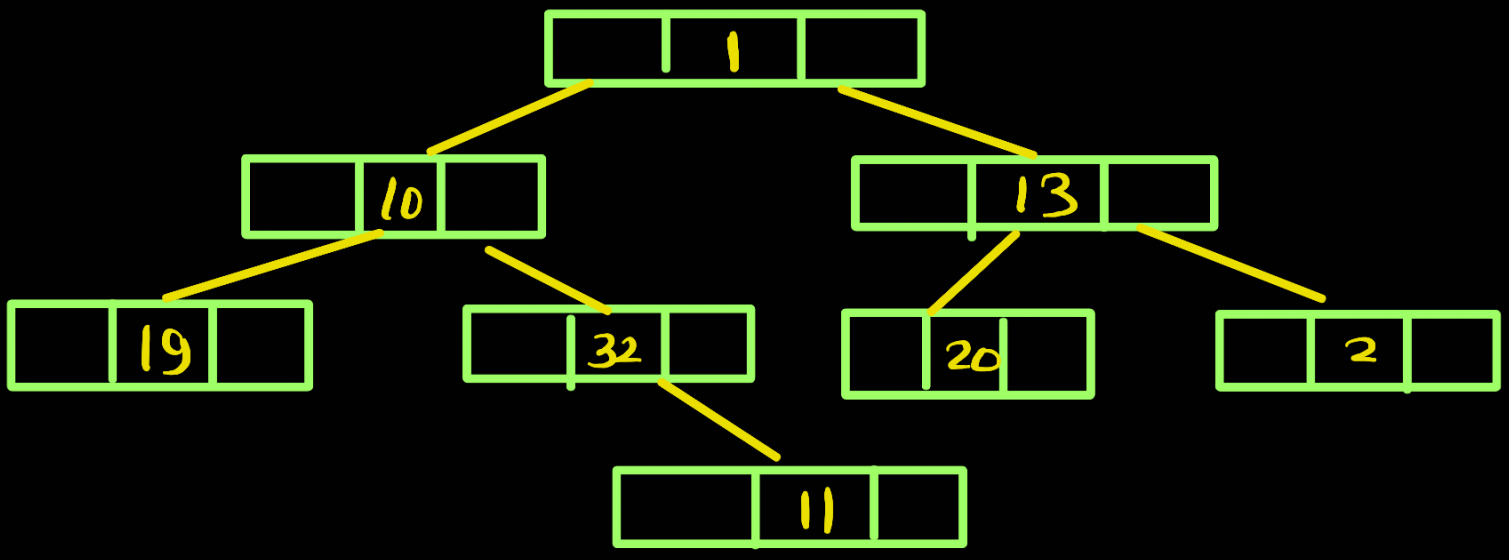
Now, the question is what information does these NULL Pointers store?

↳ Predecessor / Successor information.

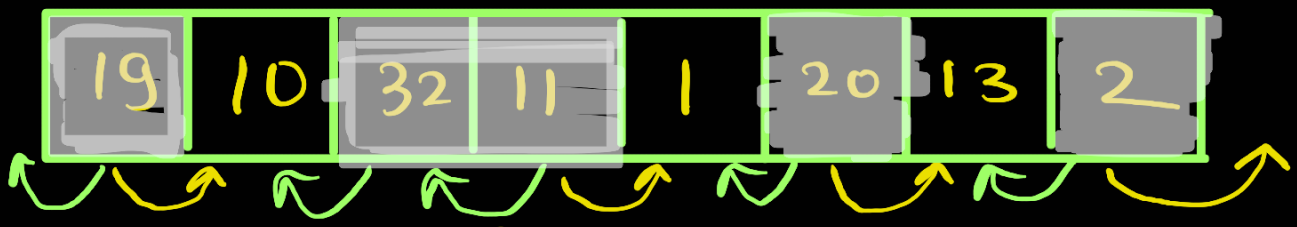
Left Null pointer points to predecessor.

Right NULL pointer points to successor.



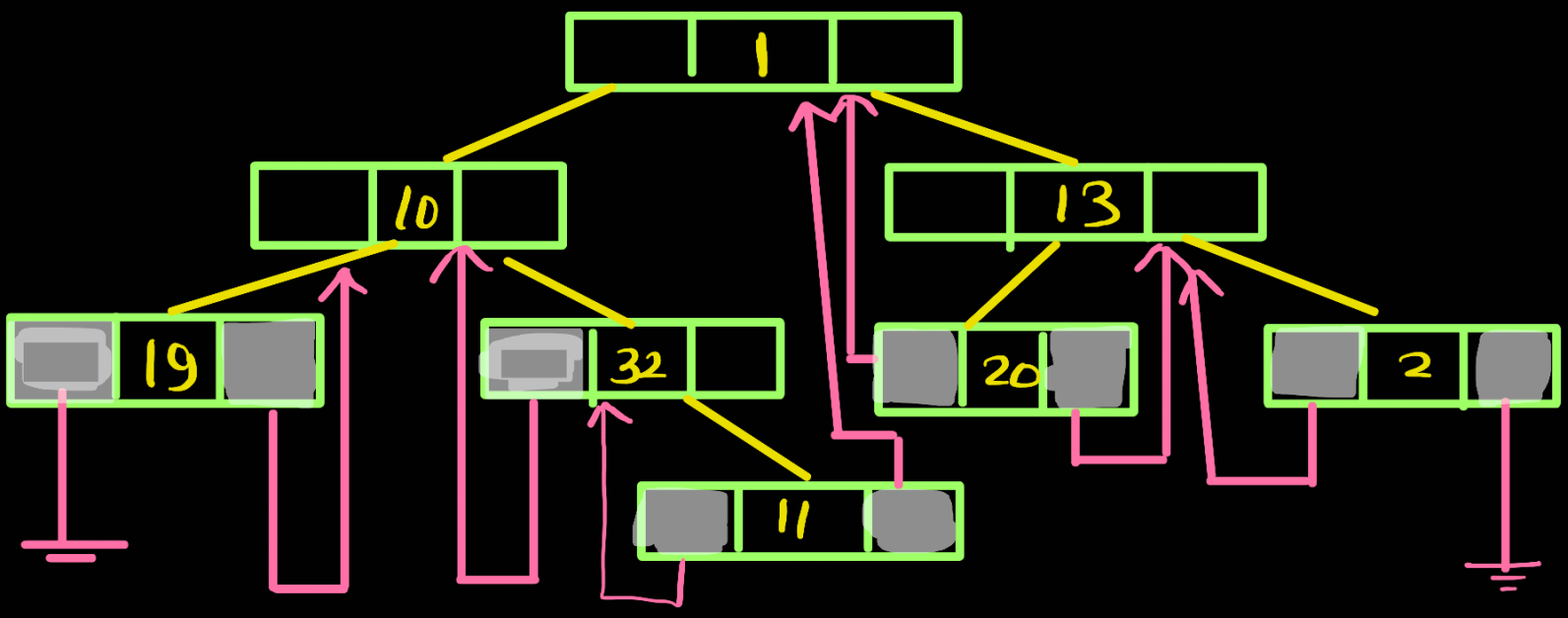


Inorder Traversal :→



The highlighted nodes have 1 or 2 empty pointers. For these nodes we will create threaded binary tree

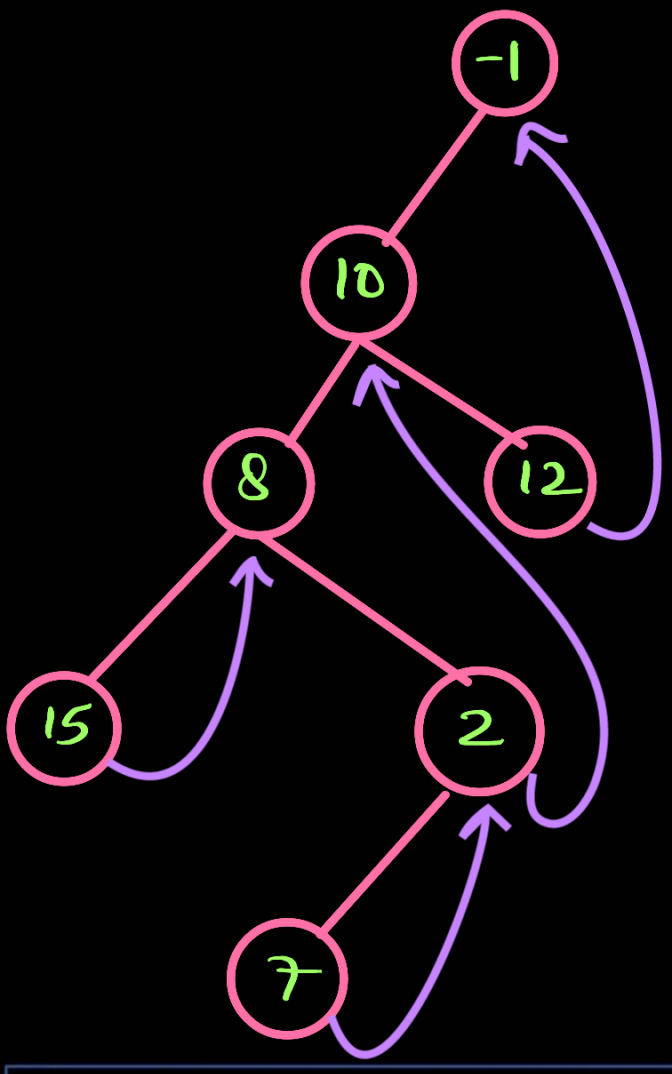
For the empty pointer we create a thread or link of left pointer to inorder predeccessor and right pointer to inorder right predecessor.



Structure :->

```
class TreeNode{
    int data;
    TreeNode left;
    TreeNode right;
    boolean leftThread; // Denotes if left pointer points to predecessor
    boolean rightThread; //Denotes if right pointer points to successor
}
```

Single Threaded Binary Tree :→



Disadvantages :→

1. Tree traversal algo becomes complex.
2. Extra space needed to store boolean value.
3. Insertion & deletion becomes more difficult.

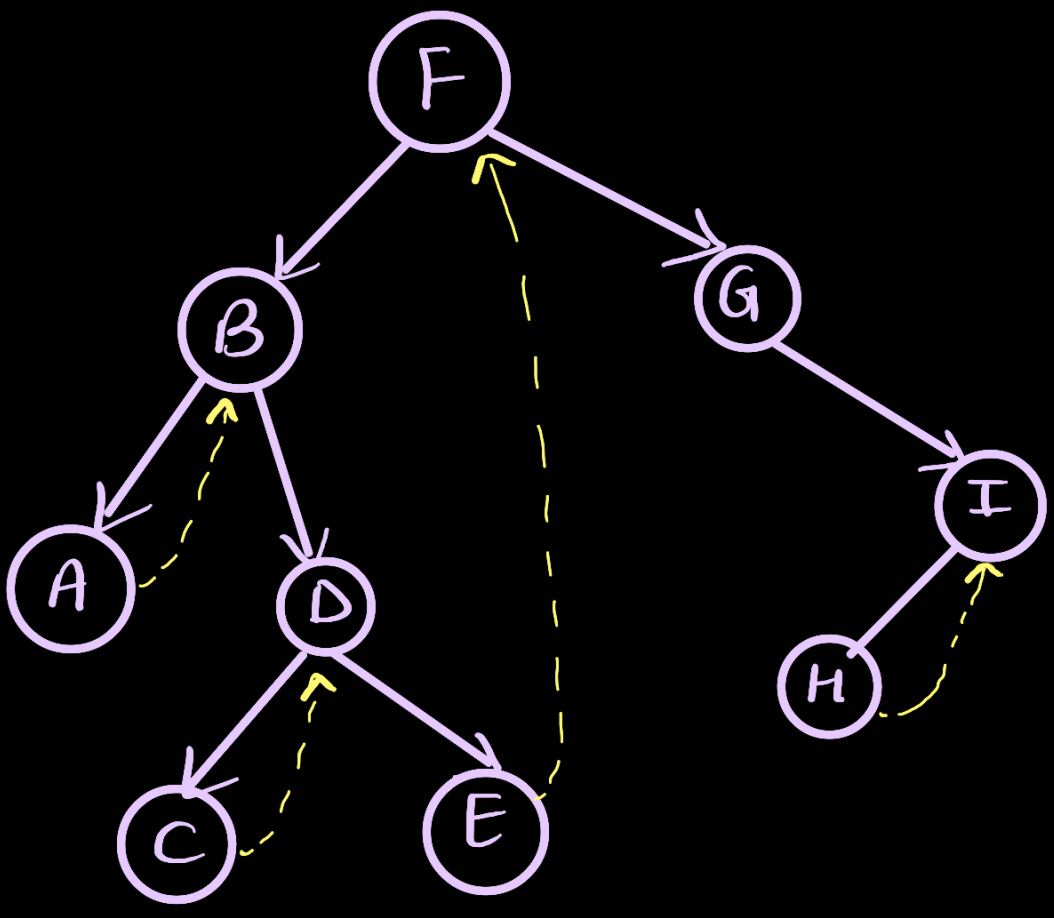
1. Initialize current --> leftmost(node)
2. Loop while current NOT_NULL && current != DUMMY_NODE
   - process Node
   - if current has Thread
     - current --> current.right
   - else
     - current --> leftmost(current.right)

Applications :->

1. In scenarios where the trees are created once with very less insertion and deletion operations but more traversal operations.

2. When stack space is limited.

Morden Morris Traversal :->



Morden :-> A B C D E F G H I

Important Point :->

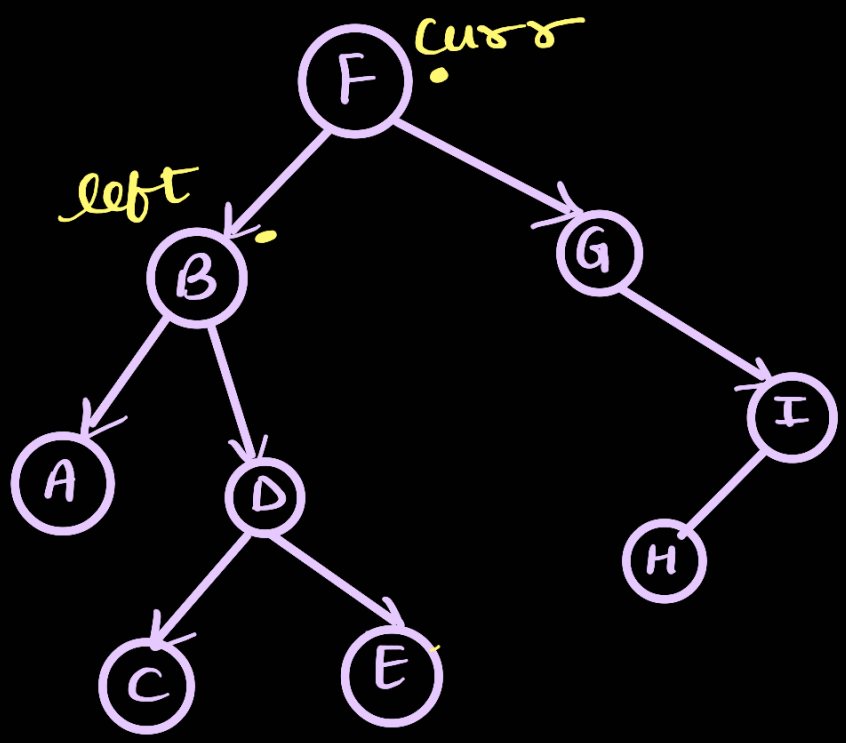
either

(1). Move to right^ when left does not exist or left subtree is completely traversed.

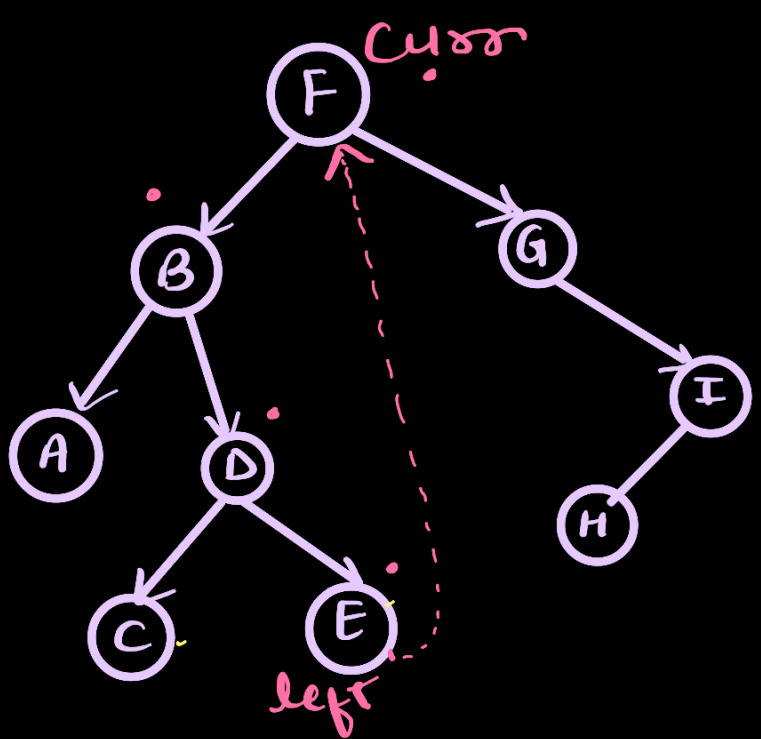
(2). Mark something so that we get to know whether

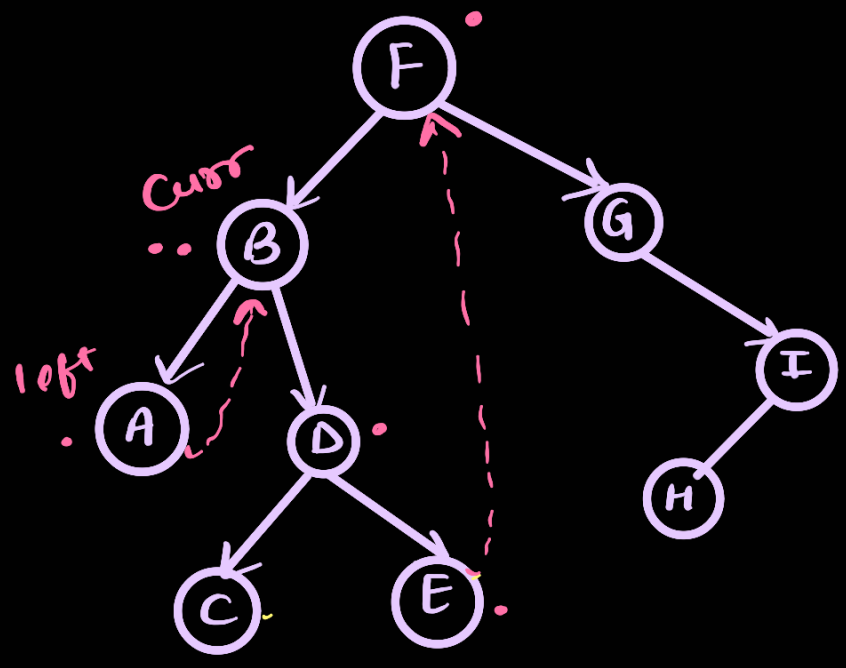
left subtree is completely processed.

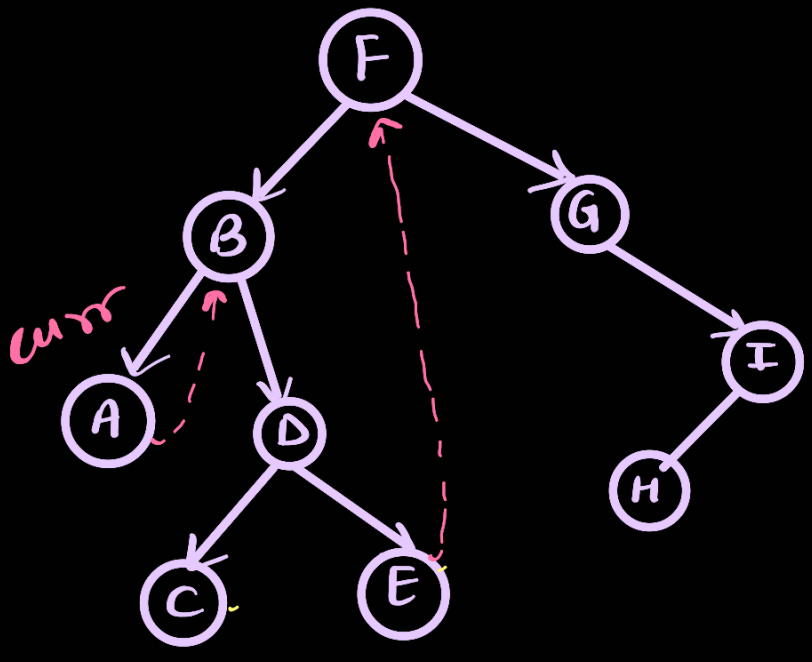
Start Morris Traversal :->



Move left to the rightmost node of left subtree to create a thread b/w rightmost node and root.







Now, when curr is at A and left points to null. It means we will print A and move to its right.

Print :->

1. If left is null
2. If thread is cut down print curr node.

* How we know these exist a thread?

When node.right point to parent it could only be possible in case of threaded tree

How do we know that left subtree is completely processed
↳ If thread already exist.

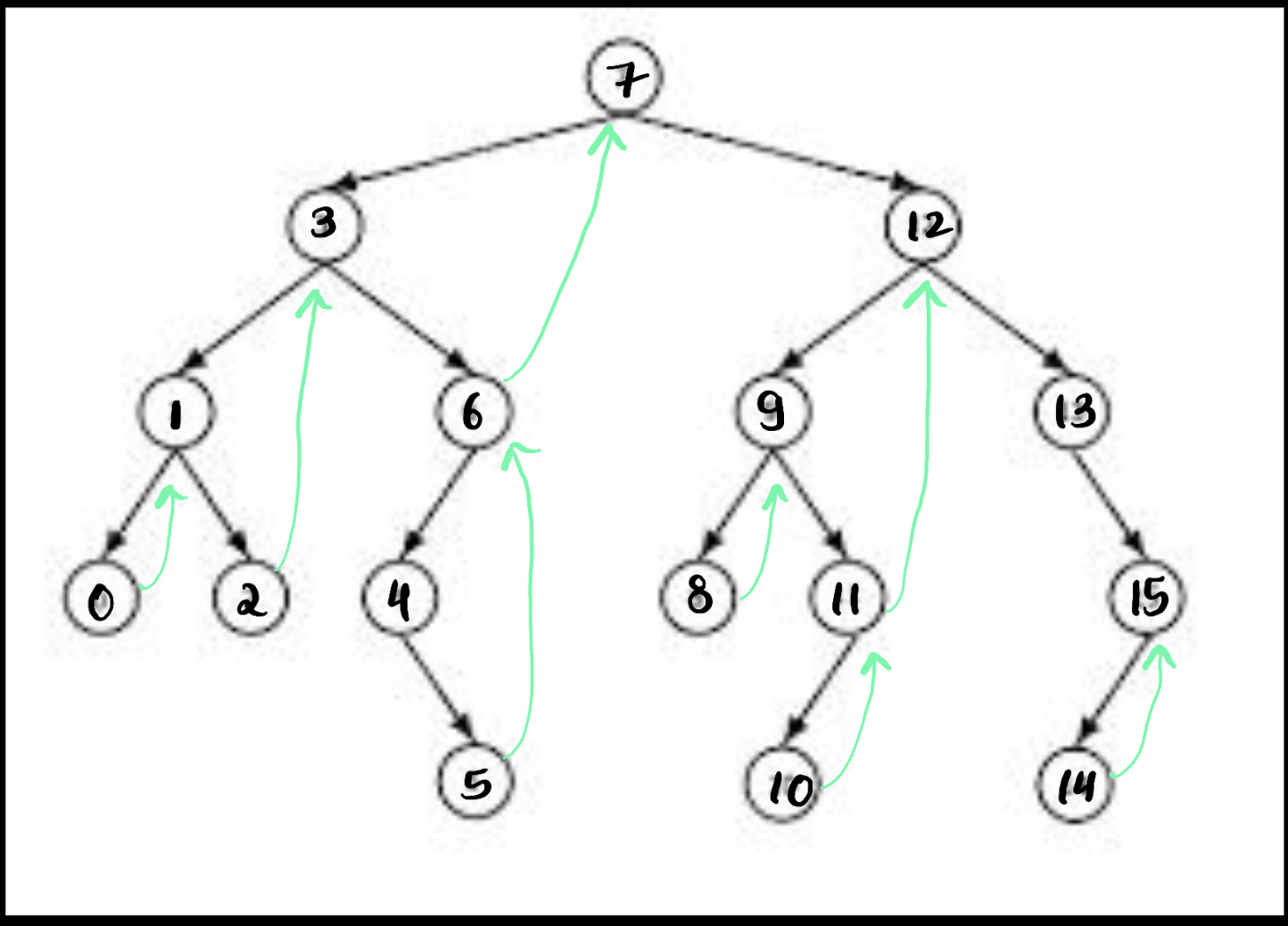
Code :-

```
public static TreeNode rightMostNode(TreeNode node, TreeNode curr) {
  while (node.right != null && node.right != curr) {
    node = node.right;
  }
  return node;
}

public static ArrayList<Integer> morrisInTraversal(TreeNode Treenode) {
  ArrayList<Integer> ans = new ArrayList<>();
  TreeNode curr = Treenode;
  while (curr != null) {
    TreeNode leftTreeNode = curr.left;
    if (leftTreeNode == null) { // left null
      ans.add(curr.val);
      curr = curr.right;
    } else {
      TreeNode rmost = rightMostNode(leftTreeNode, curr);
      if (rmost.right == null) { // thread Creation
        rmost.right = curr;
        curr = curr.left;
      } else { // thread Break
        rmost.right = null;
        ans.add(curr.val);
        curr = curr.right;
      }
    }
  }

  return ans;
}
```

Morris Preorder Traversal.



Preorder :→

7 3 1 0 2 6 4 5 12 9 8 11 10 13 15 14

Cases :→

① . Print :→

a) As soon as you visit print its value.

b) .

② . How to figure out if left subtree is processed or not ?

③ . When we should move to right subtree ?

④ . When your left subtree is null go to right .

Pseudocode :→

Point :→

① . When left child is null then print curr node .

② . As soon as we create thread print curr node .

Right :→

① . When left child is null then go to right .

② . When we cut down the thread then go to right child .

Code :->

```java
public static TreeNode getRightMostNode(TreeNode leftNode, TreeNode curr){
    while(leftNode.right != null && leftNode.right != curr){
        leftNode = leftNode.right;
    }
    return leftNode;
}

public static ArrayList<Integer> morrisPreTraversal(TreeNode curr) {
  ArrayList<Integer> ans = new ArrayList<>();
  while(curr != null){
      TreeNode leftNode = curr.left;
      if(leftNode == null){
          ans.add(curr.val);
          curr = curr.right;
      }else{
          TreeNode rightMostNode = getRightMostNode(leftNode, curr);
          if(rightMostNode.right == null){
              rightMostNode.right = curr;
              ans.add(curr.val);
              curr = curr.left;
          }else{
              rightMostNode.right = null;

              curr = curr.right;
          }
      }
  }
  return ans;
}
```

PostOrder Traversal Morris:→

```
public static HashSet<Node> postorder(Node head)
{
    Node temp = head;
    HashSet<Node> visited = new HashSet<>();
    while ((temp != null  && !visited.contains(temp)))
    {

        // Visited left subtree
        if (temp.left != null &&
         !visited.contains(temp.left))
            temp = temp.left;

        // Visited right subtree
        else if (temp.right != null &&
        !visited.contains(temp.right))
            temp = temp.right;

        // Print node
        else
        {
            System.out.printf("%d ", temp.data);
            visited.add(temp);
            temp = head;
        }
    }

    return visited;
}
```