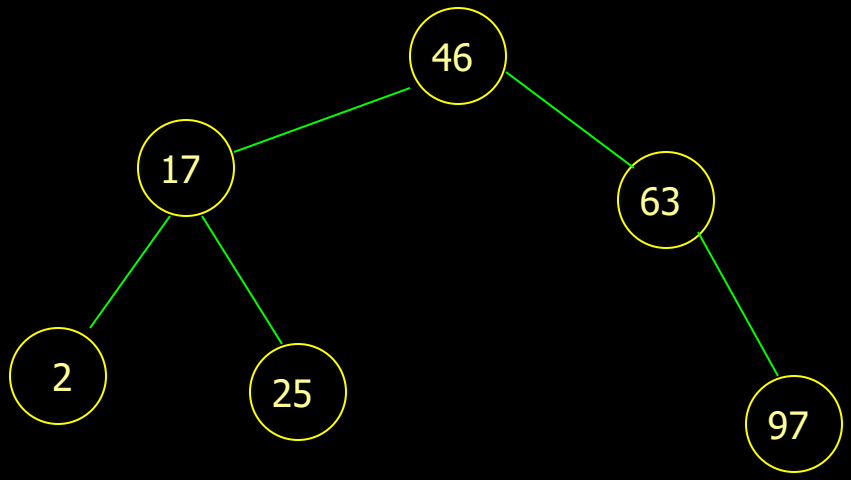
Lecture # 12

Binary Search Trees

- Consider the following Tree



- The value in each node is greater than the value in its left child and less than the value in its right child (if it exists). A binary tree having this property is called a binary search tree (BST).

Binary Search Trees

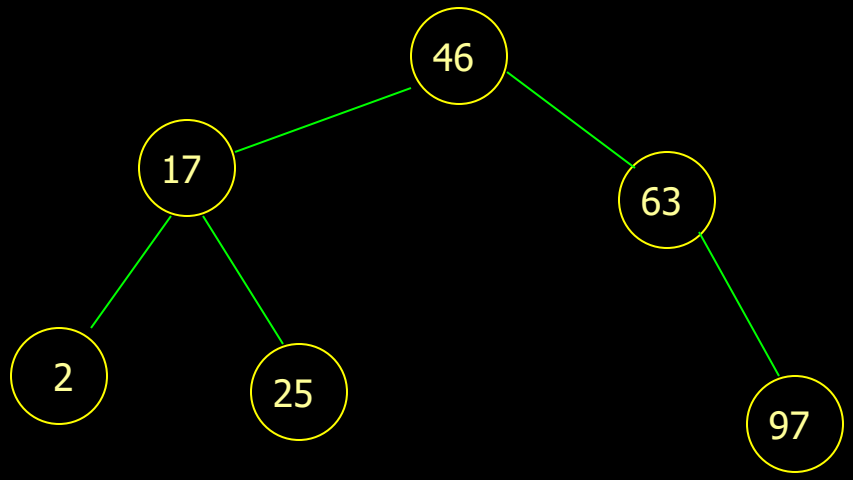
- The tree we built for searching for duplicate numbers was a binary search tree.
- BST and its variations play an important role in searching algorithms.

Basic Operations

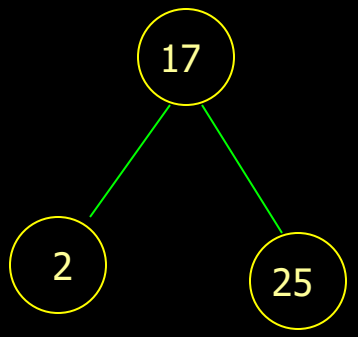
- Construct an empty BST.
- Determine if the BST is empty.
- Search the BST for a given item.
- Insert a new item in the BST and maintain the BST property.
- Delete the item from BST and maintain the BST property.
- Traverse the BST visiting each node exactly once using any traversal method.

Operations
of
Binary Search Tree (BST)

Searching a BST



- Suppose we wish to search above BST for 25, we begin at root and since 25 is less than the value 46 in this root, we know that the desired number is in left sub tree which is rooted at 17

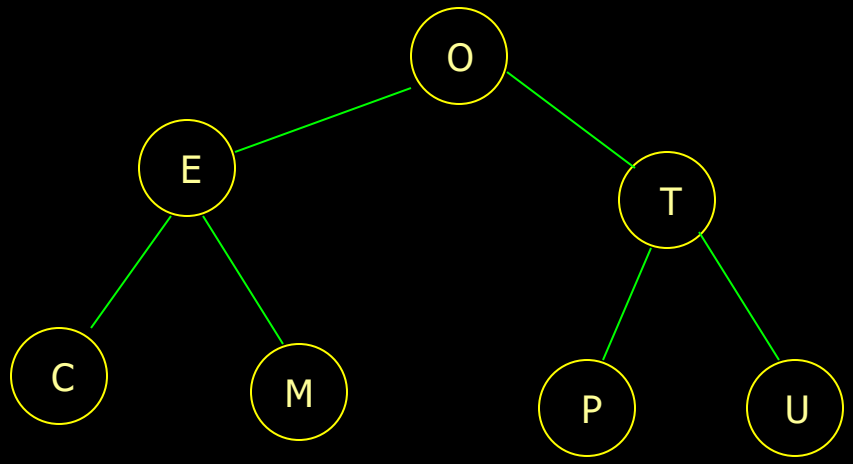


- Now we compare 25 with 17 and it is greater than 17, so we know it will exist in right sub tree of 17. so examining there we found that 25 exists so we found that 25 is present in the BST.

- Similarly to search 55 which is less than 63 so it should be in left sub tree of 63 but it is empty so we found that 55 is not present in BST

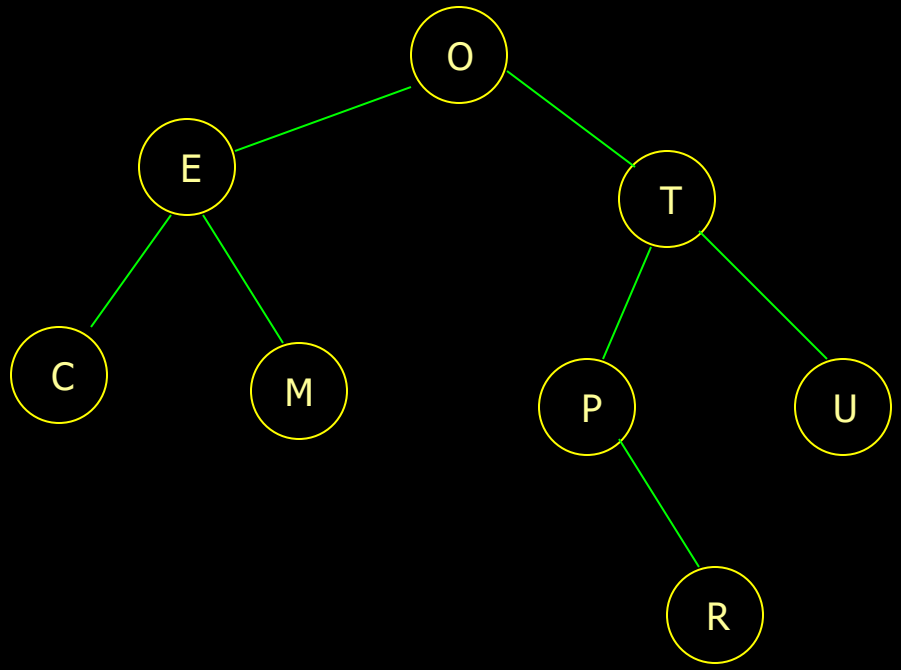
Inserting into a BST

- The method used to determine where to insert an element is similar to that used to search the tree. In fact, we need to modify the **search( )** algorithm to maintain a pointer to the parent of the node currently being examined as we descend the tree, looking for a place to insert an item. To illustrate that assume that following tree has already been generated.



- And we wish to insert R. we begin at root and compare ‘R’ with letter there. Since R is greater (>) than O, we descend to the right sub tree. After comparing R with T stored into the root of this sub tree, we descend to left sub tree since R < T.

- Now when we reach P we find that R > P. so we move to the right subtree of P but we find the fact that the right sub tree is empty, which shows that R was not present previously in the tree and should be inserted at this place i.e. as a right child of P in the BST as follows.



Problem of Lopsidedness (irregularity/unevenness)

- The order in which items are inserted into a BST determines the shape of the tree. For example, inserting the letters O, E, T, C, U, M, P into a BST of characters in this order gives the nicely balanced tree.

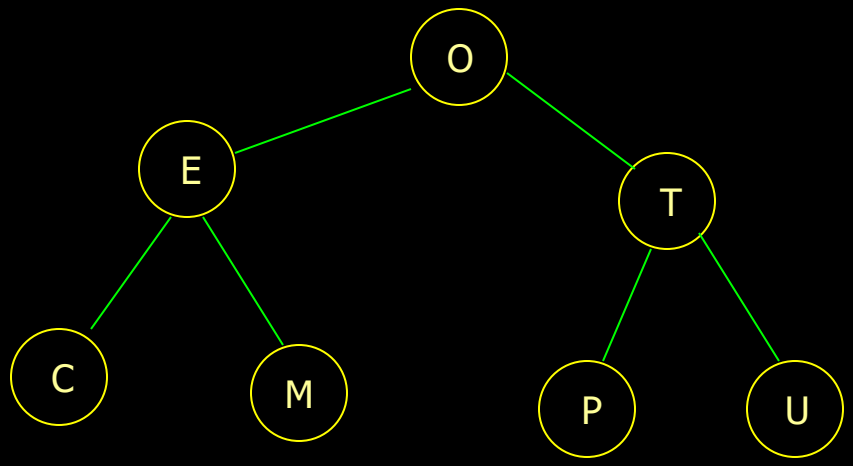


Figure 1

- But inserting them in the order C, O, M, P, U, T, E yields the unbalanced tree

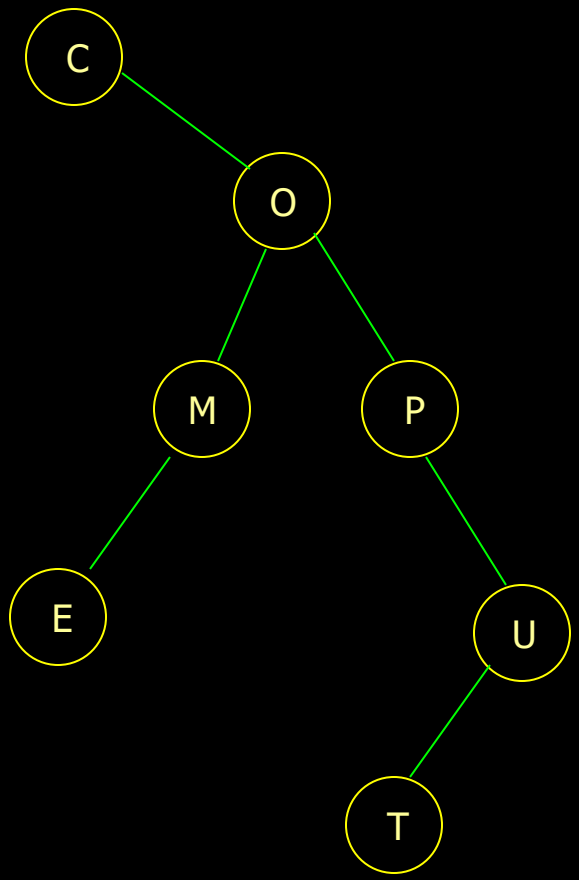


Figure 2

- And inserting characters in the order C, E, M, O, P, T, U will generate the following BST in worst unbalanced fashion.

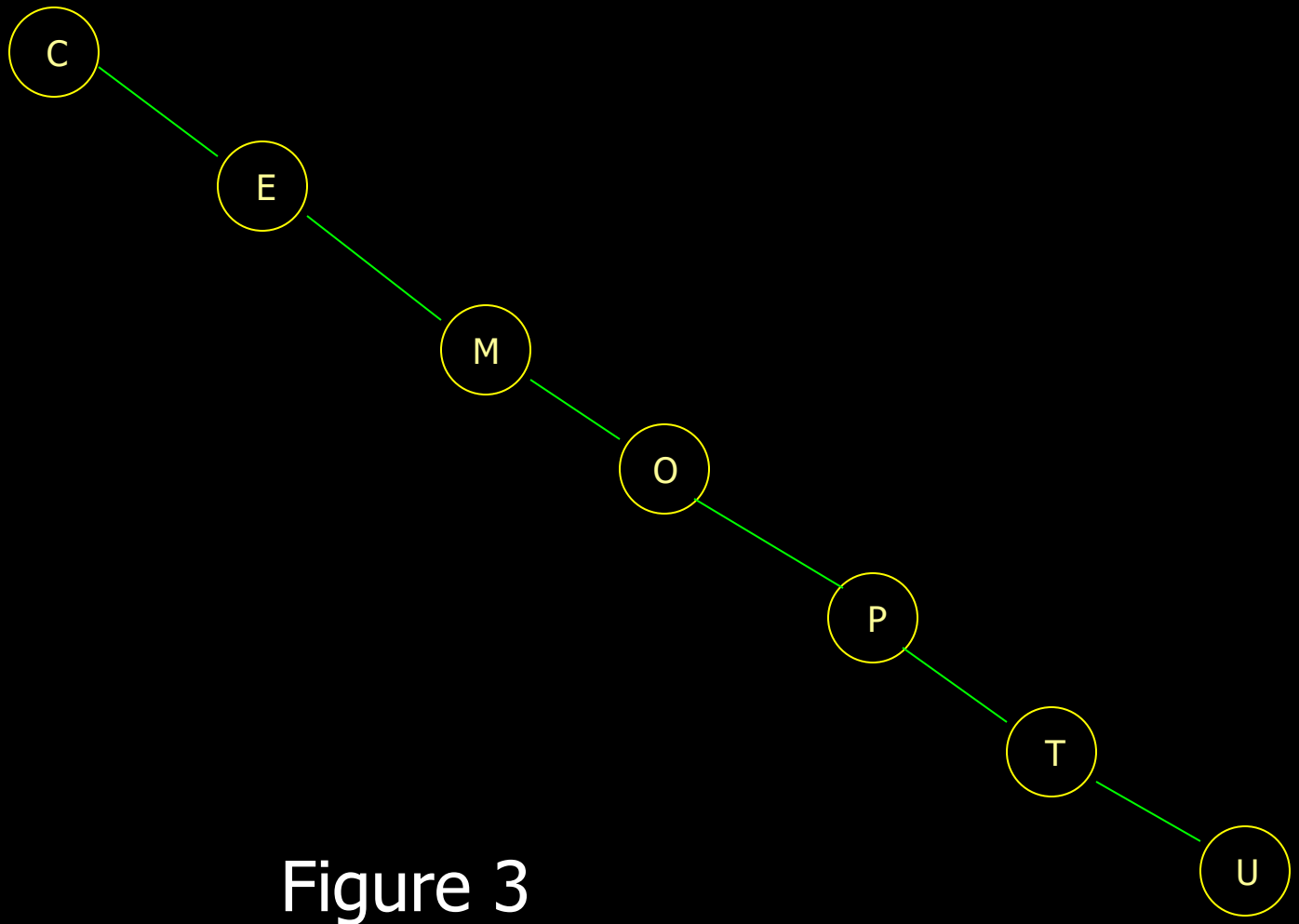


Figure 3

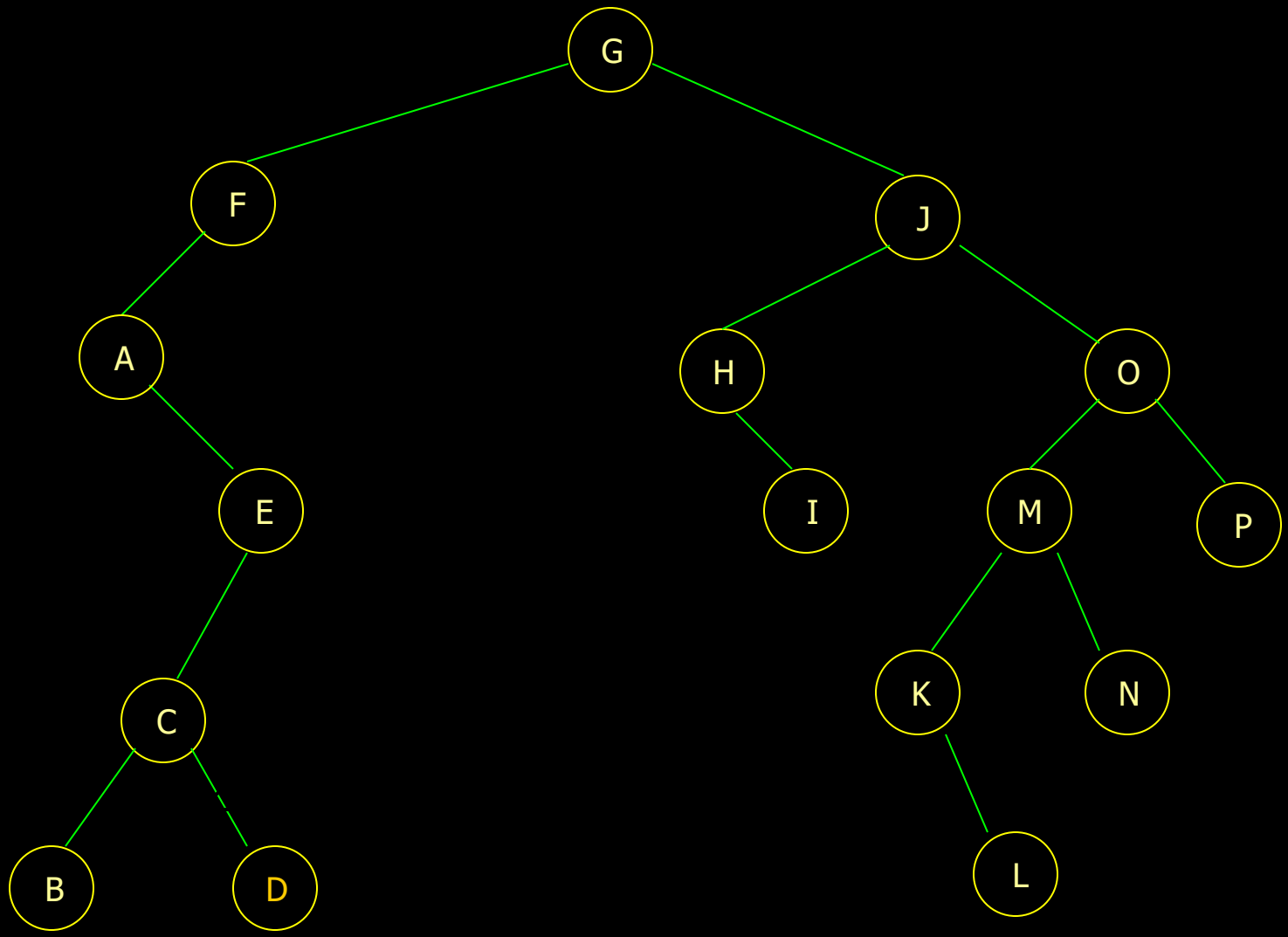
Deletion from a BST

- To delete a node X from BST we consider three cases

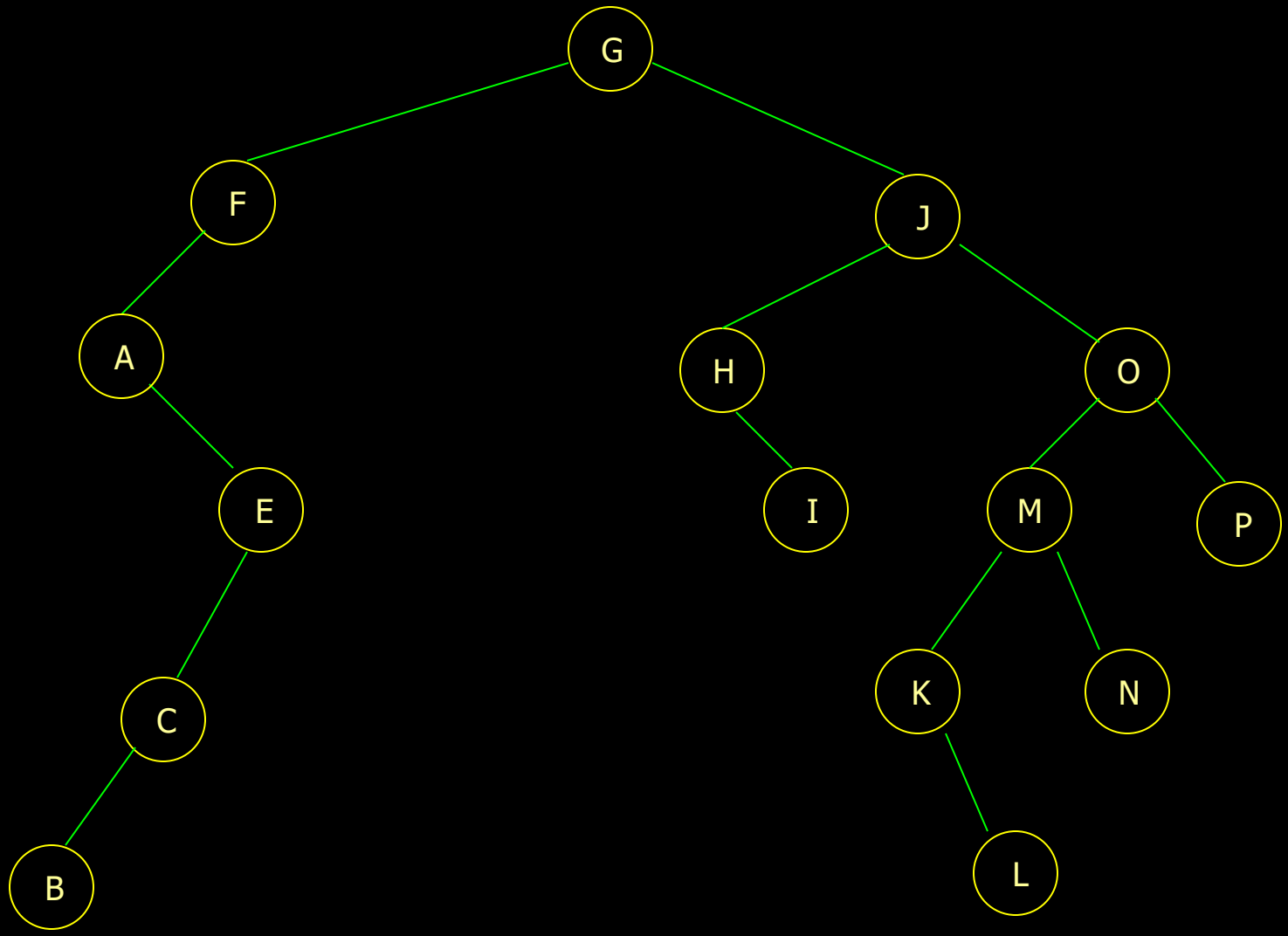
1. X is a leaf
2. X has only one child
3. X has two children

- The First case is very easy. We simply make the appropriate pointer in X's parent null pointer i.e. left or right pointer according to the shape or situation of the tree.

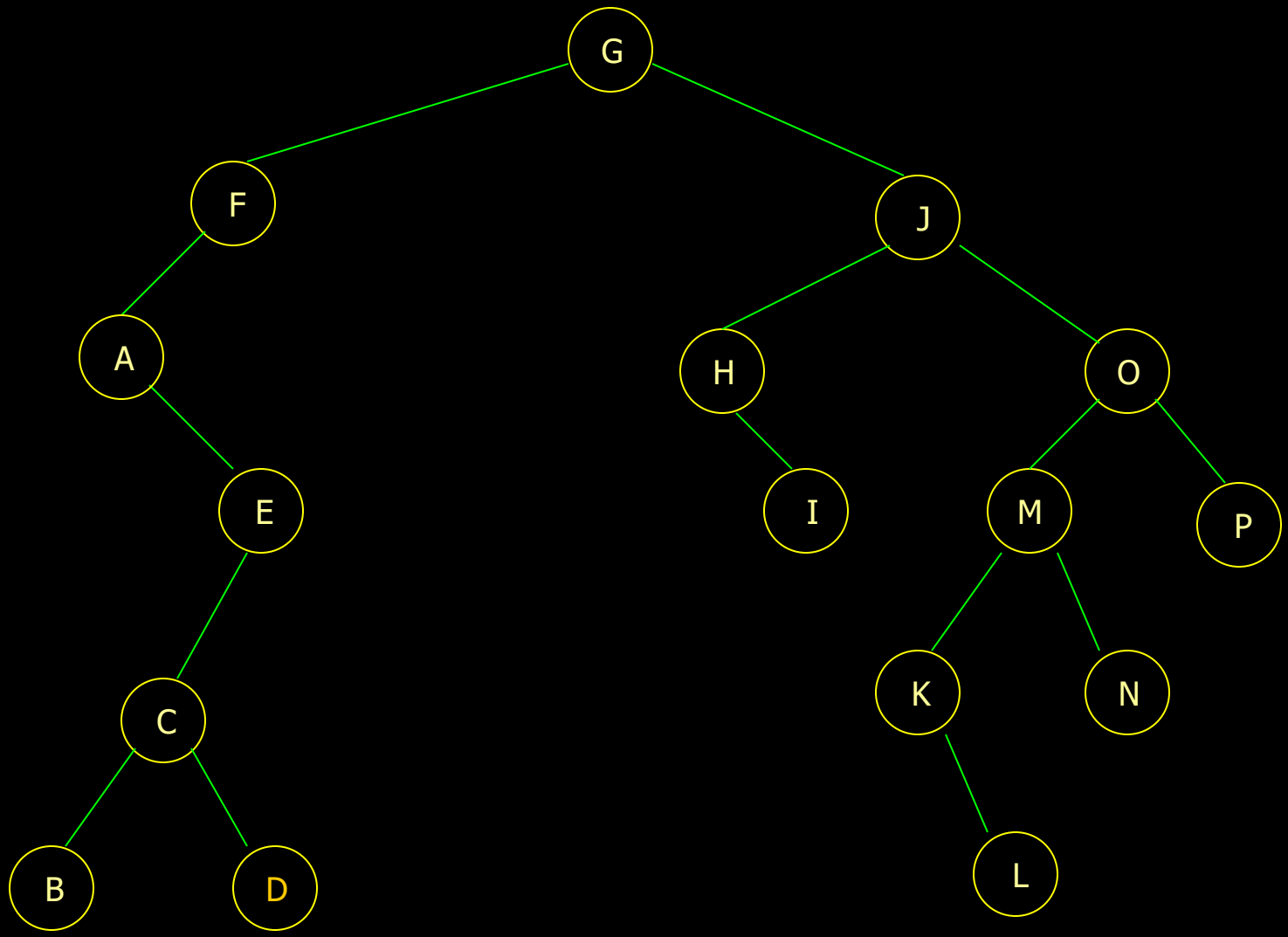
- Consider the Example……..

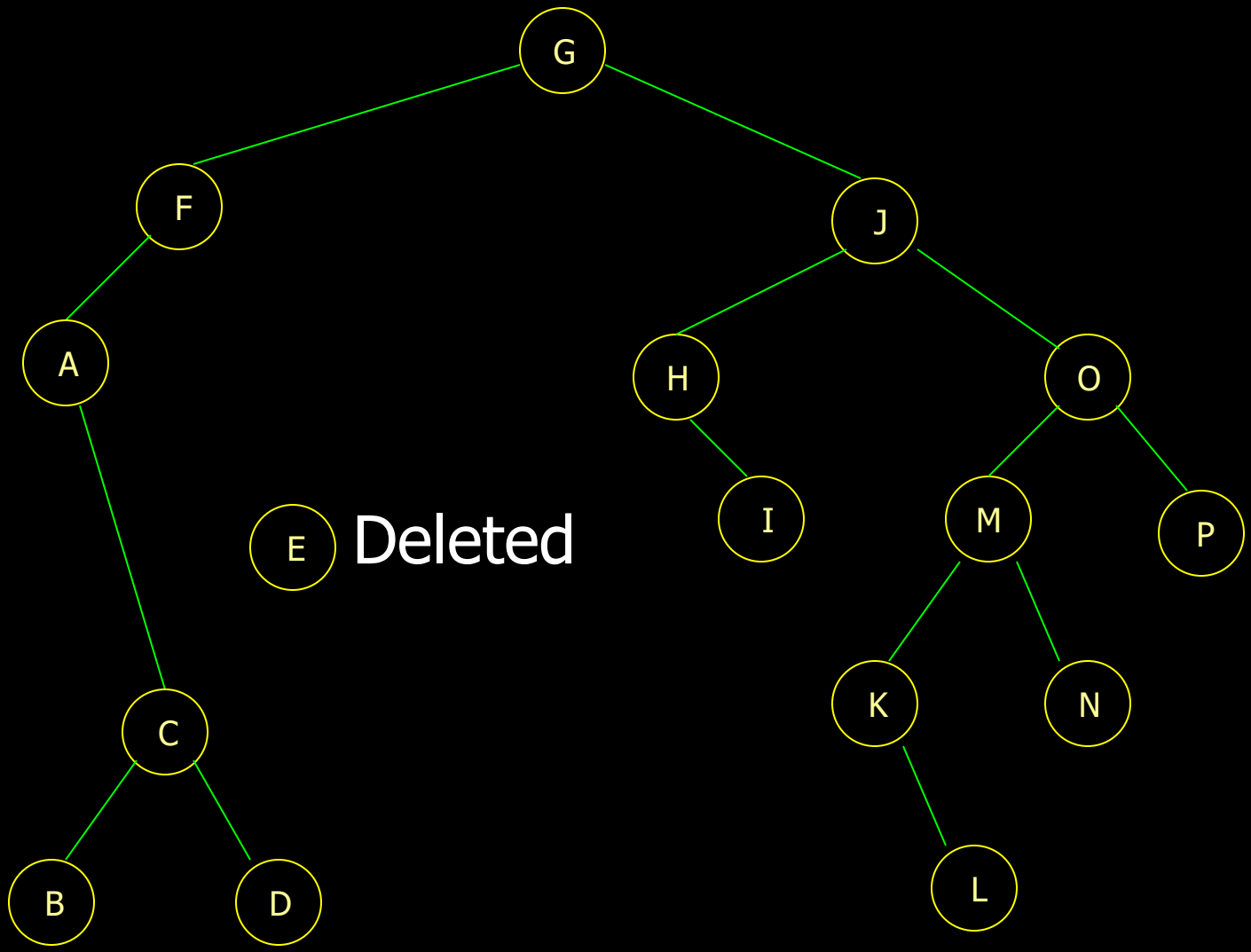

G
F
J
A
H
O
E
I
M
P
C
K
N
B
D
L

- For example to delete D in the above BST we can simply make the right pointer in its parent C a NULL.



- The Second case, where the node X has exactly one child. Here we need to only set the appropriate pointer in X's parent to point to child. For example we can delete the node E in BST by simply setting the right pointer of its parent A to point to the node C and then delete X as follows.

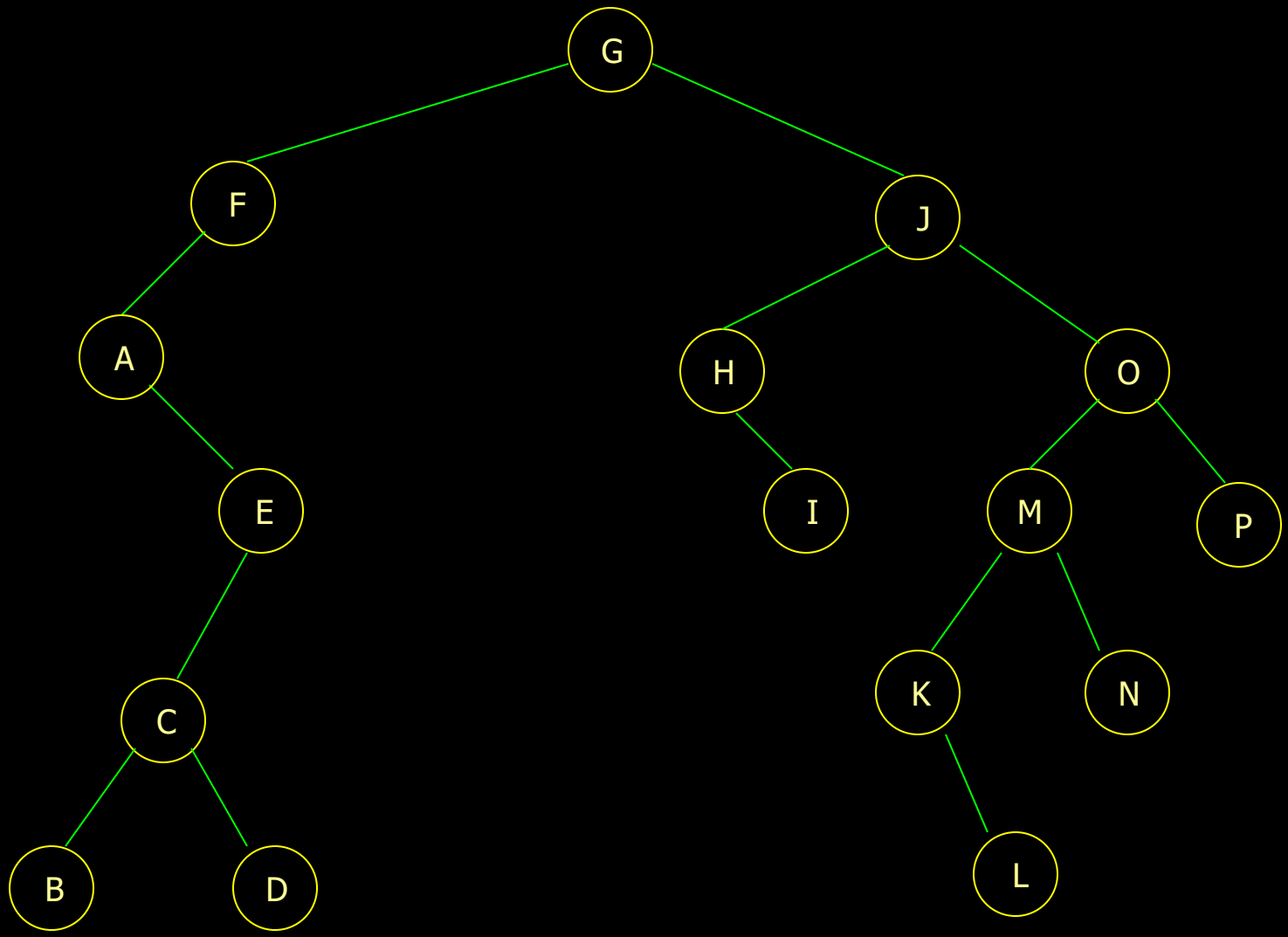

G
F
J
A
H
O
E
I
M
P
C
K
N
B
D
L


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

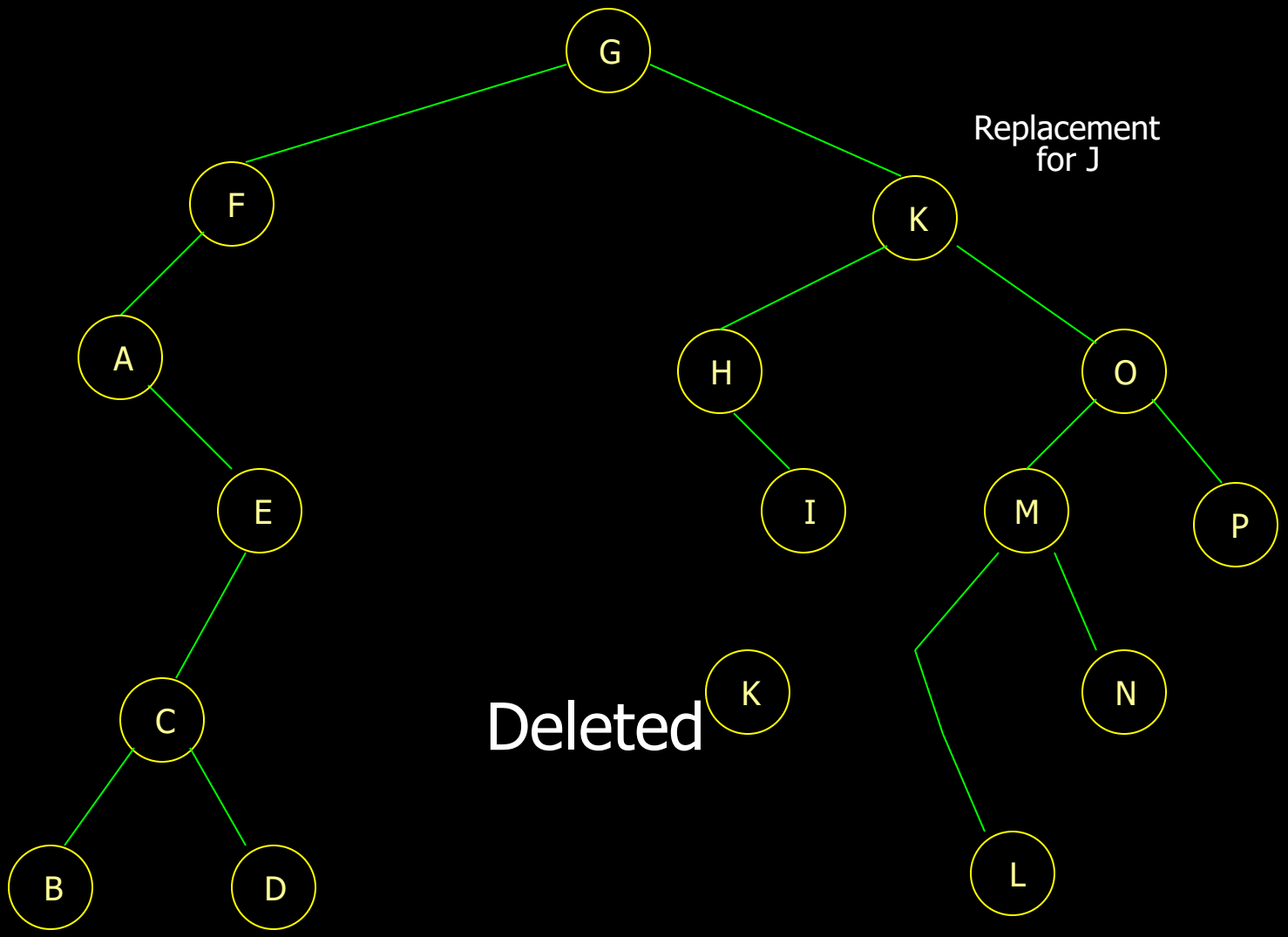
- In third case, in which X has 2 children we can replace the value stored in node X by its inorder successor or predecessor and then delete that successor or predecessor.

- To consider this we consider the following BST and we want to delete the node J.

- So we can replace it with its immediate inorder successor and we can locate it by starting from right child of J. and in our example the immediate inorder successor of J is K.


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

After deletion we have....



Implementation of BST (Binary Search Tree)

```
/*          Implementation of Binary Search Tree
                        Only Insertion                          */
#include <iostream.h>
#include <conio.h>
#include <stdlib.h>

struct node
{
  int info;
  node *left,*right;
};
class bin_search_tree
{
   private :
         node *temp;
   public  :
         node *root;
         int number;
         bin_search_tree();
         void options();
         void b_search_tree(node *);
         void in_order(node *);
};
```

```
void main()
{
  clrscr();
  char ch;
  bin_search_tree obj;
  while( 4 )
  {
    obj.options();
    ch=getch();
    switch(ch)
     {
       case '1':
            clrscr();
            cout<<"\n Enter number to add in a tree... \n";
            cin>>obj.number;
            obj.b_search_tree( obj.root );
            break;
```

```
        case '2':
             clrscr();
             obj.in_order(obj.root);
             break;
        case '3':
             exit(0);
             break;
        default :
             exit(0);
             break;
      } // end of switch.
   } // end of while.
}//---------------------------------------

bin_search_tree :: bin_search_tree()
{
  root=temp=NULL;
}
//-------------------------------------------------------
```

```
void bin_search_tree :: b_search_tree(node *temp) //b_search_tree(int val)
{
  if( root==NULL )
   {
     temp=new node;      //root = new node(val);
     temp->info = number;
     temp->left=NULL;
     temp->right=NULL;
     root=temp;
     return;
   }

  if( temp->info==number )
   {
      cout<<" \n Given number is already present in tree.\n";
      return;
    }
```

```
if(temp->info > number)
 {
    if( temp->left!=NULL )
     {
          b_search_tree(temp->left);
          return;
     }
    else
     {
        temp->left=new node;    //temp->left = new node(val);
        temp->left->info = number;
        temp->left->left=NULL;
        temp->left->right=NULL;
        return;
     }
 }
```

```
if(temp->info < number)
   {
      if( temp->right!=NULL )
       {
            b_search_tree( temp->right );
            return;
       }
      else
       {
          temp->right=new node; //temp->right = new node(val);
          temp->right->info = number;
          temp->right->left=NULL;
          temp->right->right=NULL;
          return;
       }
   }

} //--------------------    Insertion Function  Ends    ------------------------
```

```
void bin_search_tree :: options()
{
        cout<<"\n\n ************** Select Option ******************.\n";
        cout<<"\n Enter any of choices.\n";
        cout<<"\n 1 : Adding (inserting) node in BST.\n";
        cout<<"\n 2 : Print the whole BST .\n";
        cout<<"\n 3 : Quitting the Program.\n";
}//-------------------------------------------------------
void bin_search_tree :: in_order(node *temp)
{
        if(root==NULL)
          {
          cout<<" Tree is empty.\n";
          return;
          }
        if( temp->left!=NULL )
              in_order(temp->left);
        cout<<temp->info<<"  ";
        if( temp->right!=NULL )
              in_order(temp->right);
        return;
}//--------------------------     PROGRAM ENDS HERE  ------------------------------------
```

Thank You.....