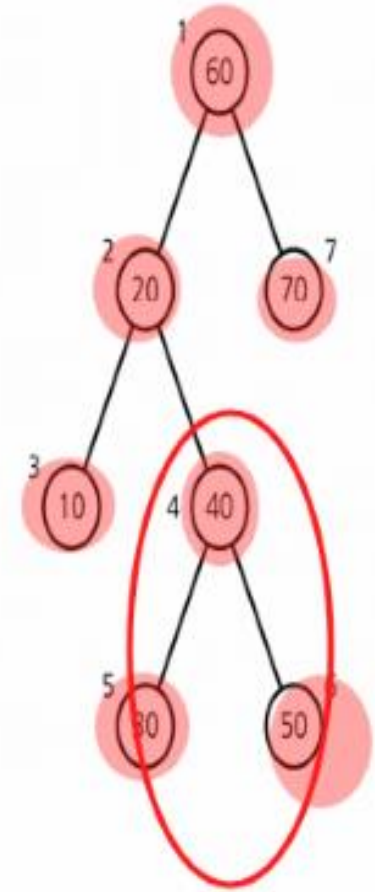
# Algorithms & Data Structure

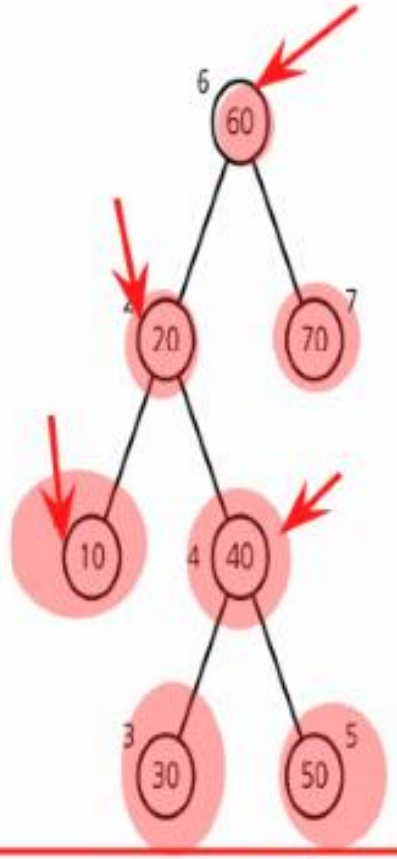
Kiran Waghmare

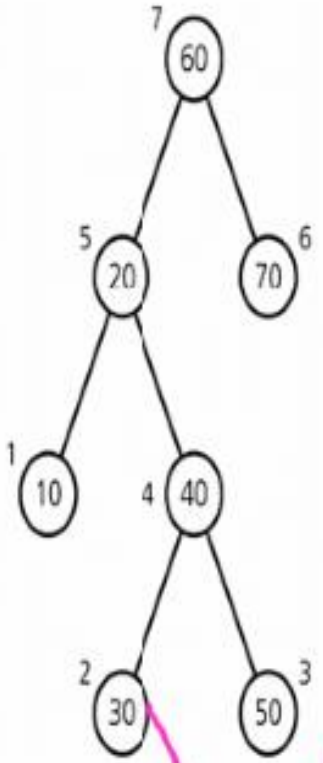
# Binary Tree Traversals

Preorder: 60,20,10,40,30,50,70

Postorder:10,30,50,40,20,70,60



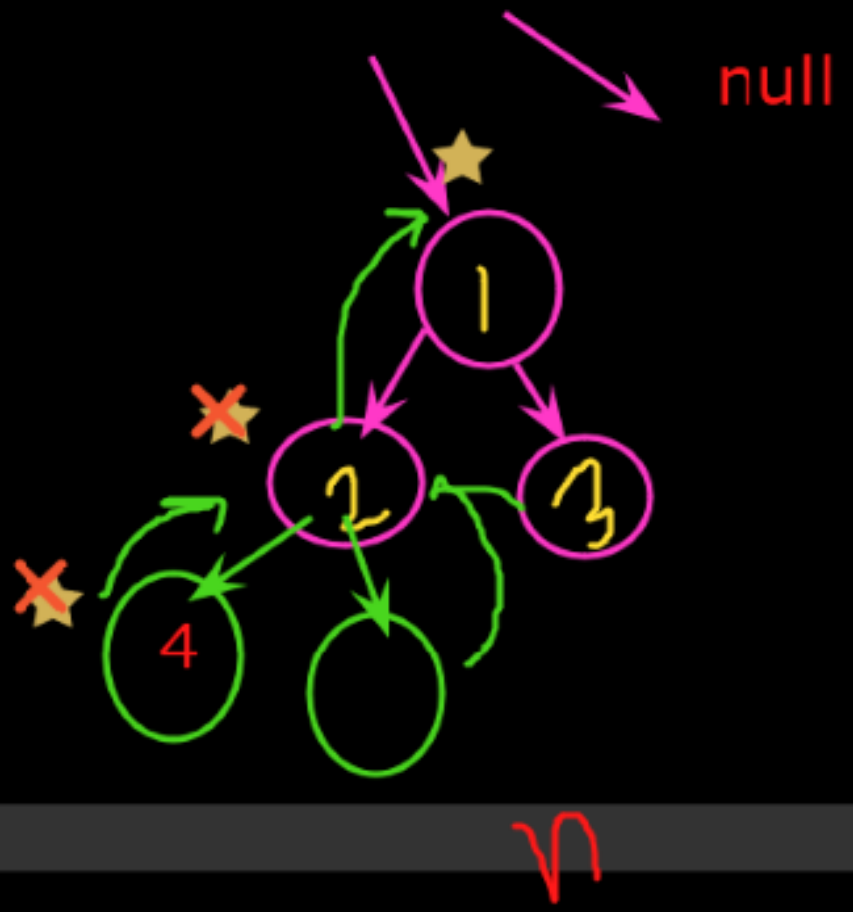




Inorder: 10,20,30,40,50,60,70

```
        return;
    System.out.println(n.data+);
    printPreorder(n.left);
    printPreorder(n.right);
}
```

LC, Root, RC

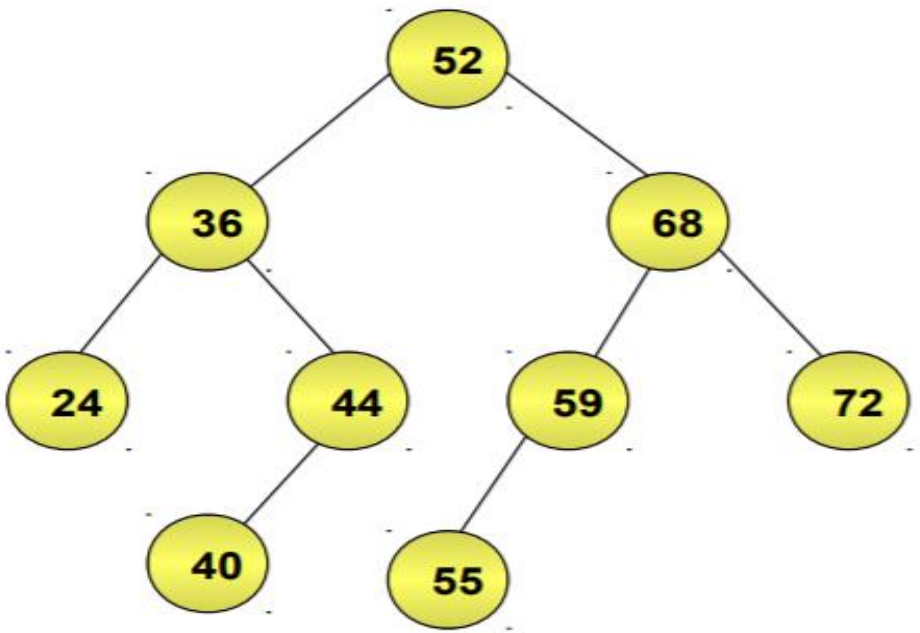
```
void printPostorder(Node n)
{
    if(n == null)
        return;

    printPostorder(n.left);
    printPostorder(n.right);
    System.out.println(n.data+);
}
```




n=null
n.left==null
n.right==null

# Binary Search Tree

- Binary search tree is a binary tree in which every node satisfies the following conditions:
  - All values in the left subtree of a node are less than the value of the node.
  - All values in the right subtree of a node are greater than the value of the node.
- The following is an example of a binary search tree.



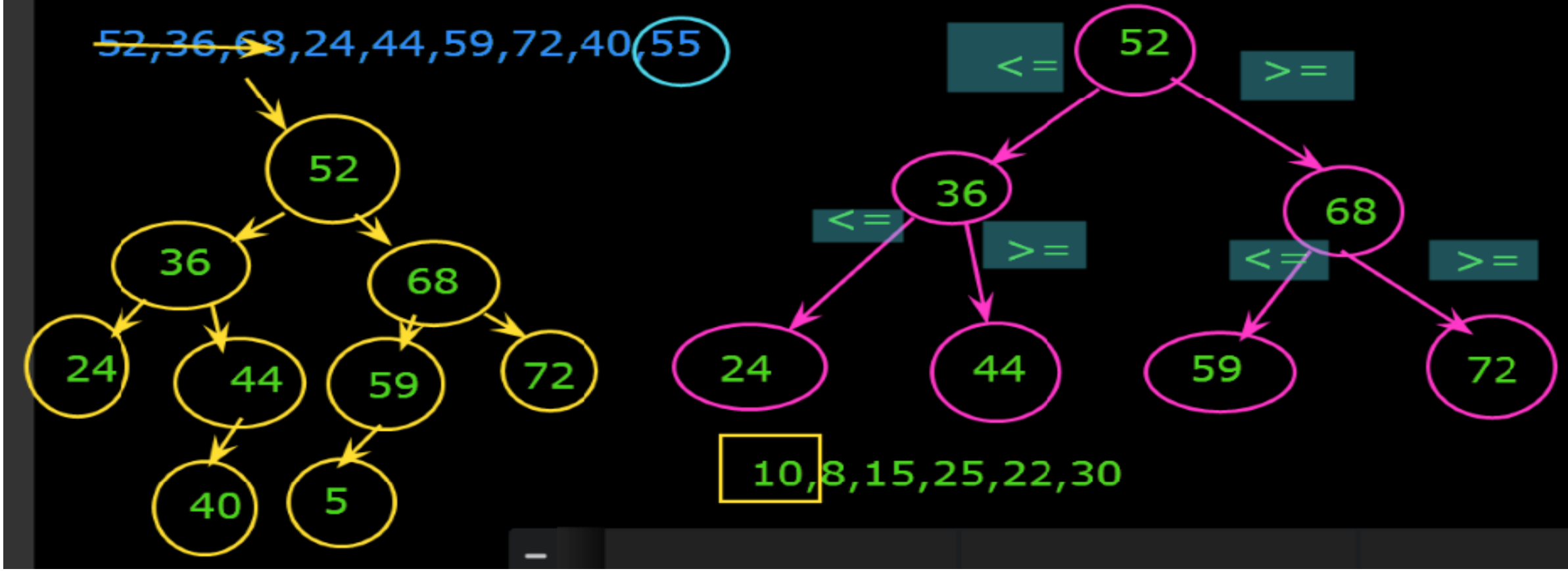
# Binary search Tree:

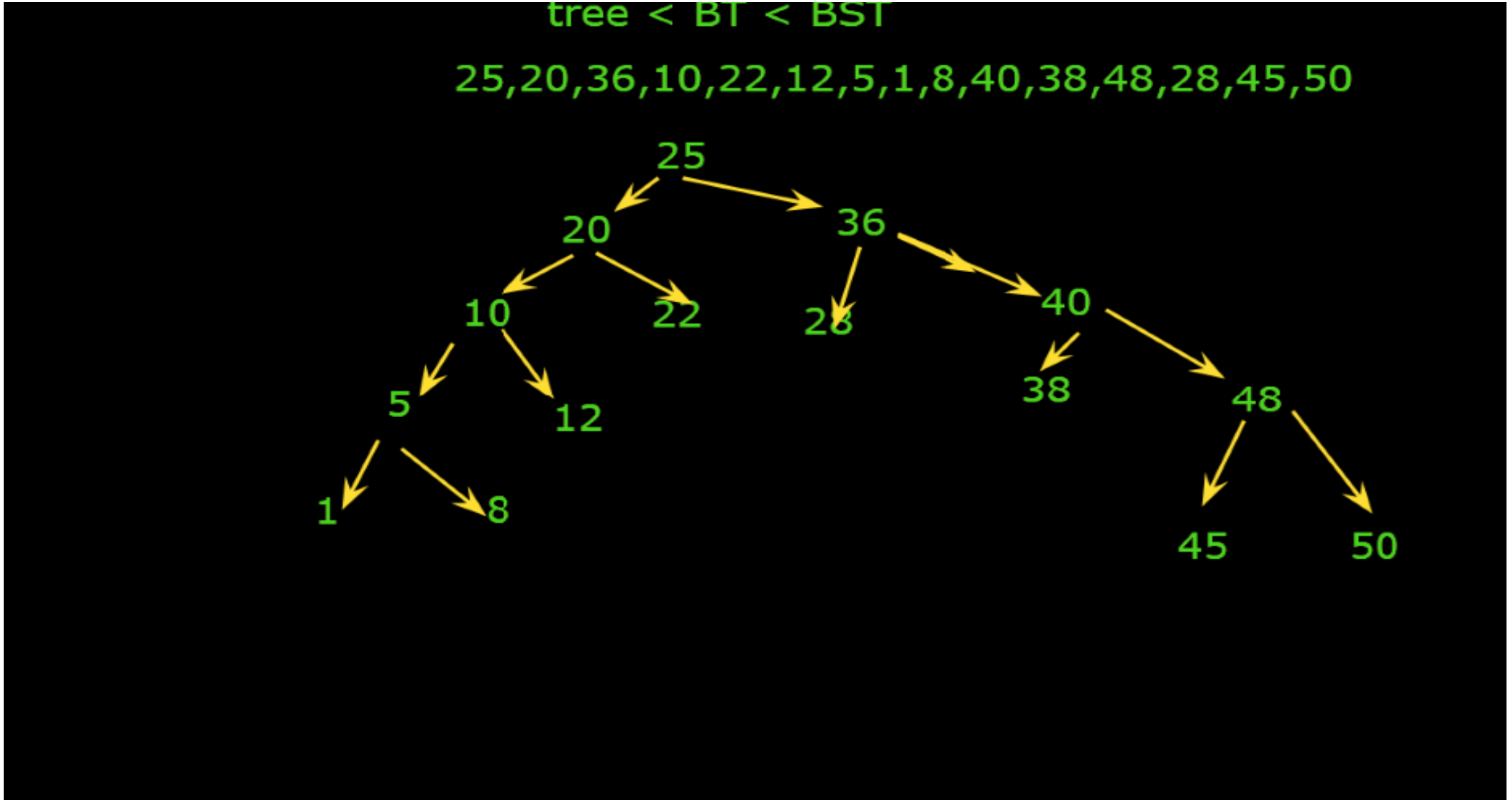
tree < BT < BST

---

-It is a binary tree in which every node satisfies the following conditions:

- -All values in the left subtree of a node are less than the value of the node.
- -All the values in the right subtree of a node are greater than the value of the node.

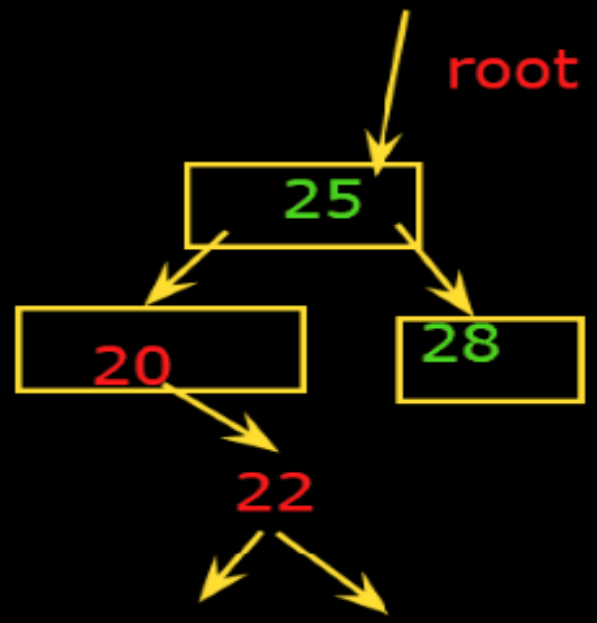



tree < BT < BST
25,20,36,10,22,12,5,1,8,40,38,48,28,45,50
25
20
36
10
22
28
40
5
12
38
48
1
8
45
50

# Operations on a Binary Search Tree

- The following operations are performed on a binary earch tree...
  - Search
  - Insertion
  - Deletion
  - Traversal

# tree < BT < BST




```
Node insertdata(Node root, int key)
{
    if(root == null)
    {
        root = new Node(key);
        return root;
    }
    if(key <= root.data)
        root.left = insertdata(root.left,key);
    else
        root.right = insertdata(root.right,key);
        return root;
}
```

# Insertion of a key in a BST

```
Algorithm:- InsertBST (info, left, right, root, key, LOC)
{
   key is the value to be inserted.
   1. call SearchBST ( info, left, right, root, key, LOC , PAR )    // Find the parent of the new node
   2. If ( LOC != NULL)
   2.1 Print " Node alredy exist"
   2.2 Exit
   3. create a node [ new1 =  ( struct node*) malloc ( sizeof( struct node) ) ]
   4. new1 -> info = key
   5. new1 -> left = NULL , new1 -> right = NULL
   6. If ( PAR = NULL ) Then
   6.1 root = new1
   6.2 exit
     elseif ( new1 -> info < PAR -> info)
   6.1 PAR -> left = new1
   6.2 exit
     else
   6.1 PAR -> right = new1
   6.2 exit
 }
```

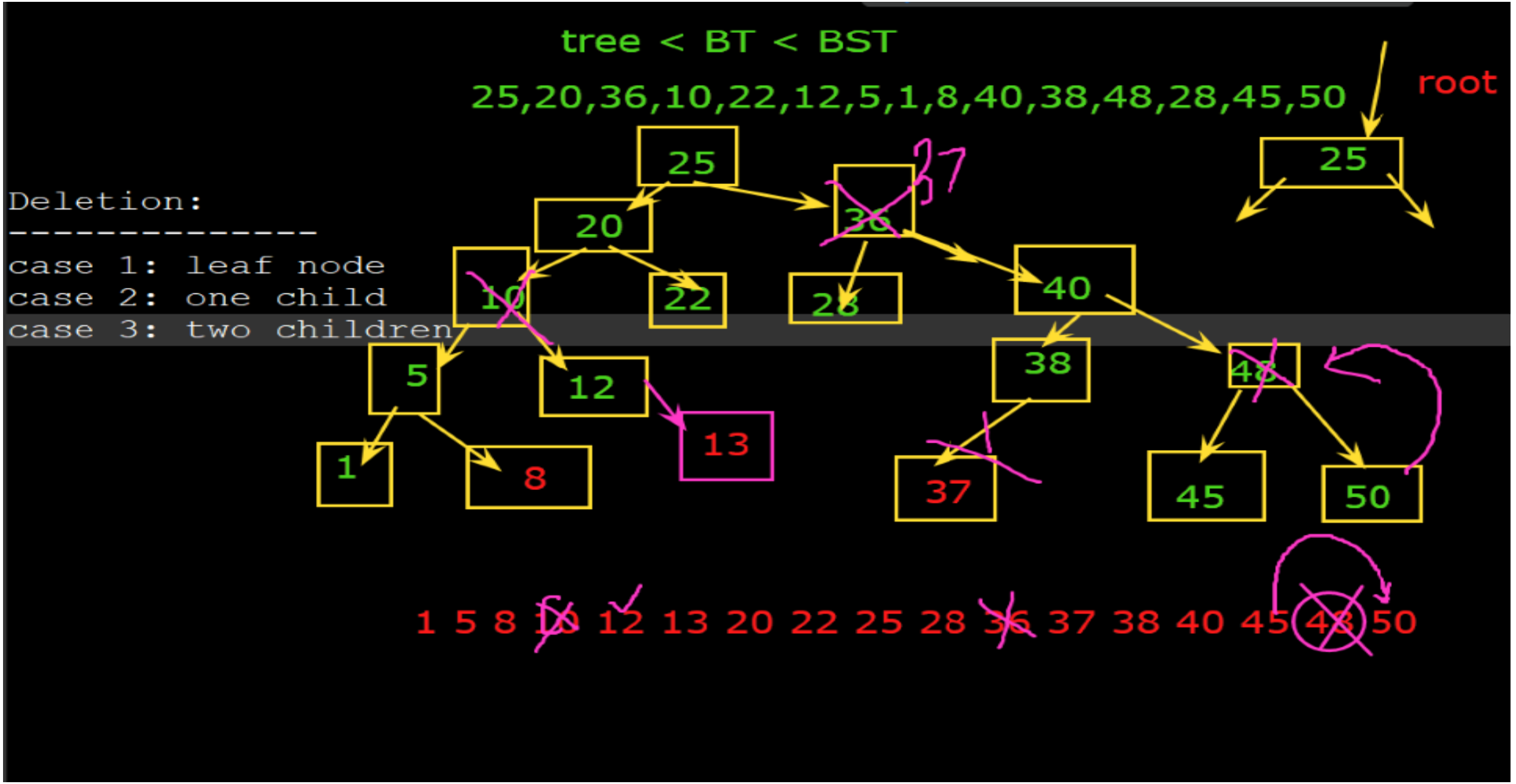
# Deleting Nodes from a Binary Search Tree

- Write an algorithm to locate the position of the node to deleted from a binary search tree.

- Delete operation in a binary search tree refers to the process of deleting the specified node from the tree.
- Before implementing a delete operation, you first need to locate the position of the node to be deleted and its parent.
- To locate the position of the node to be deleted and its parent, you need to implement a search operation.

# Deleting Nodes from a Binary Search Tree (Contd.)

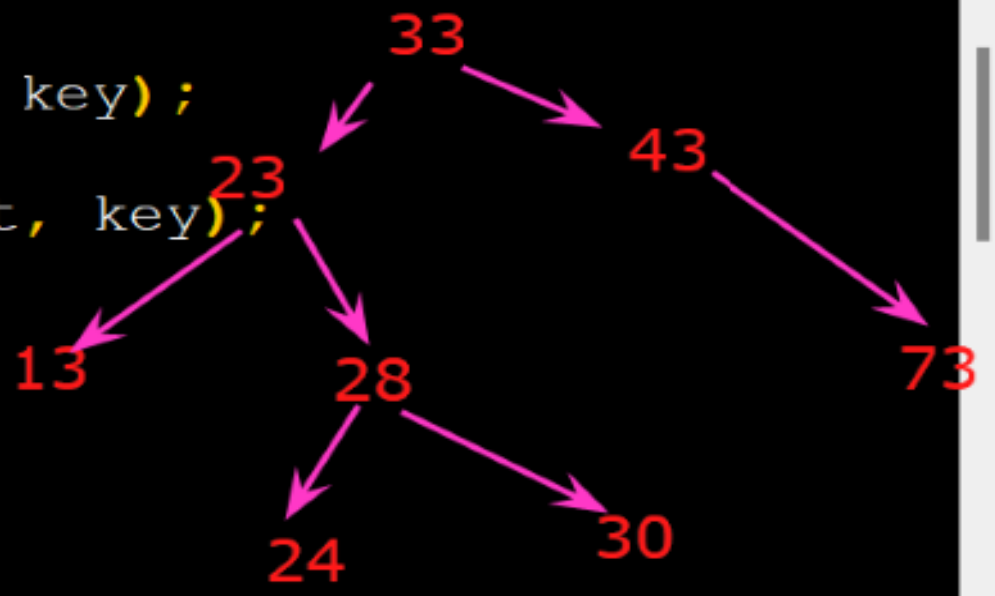
- Once the nodes are located, there can be three cases:
  - **Case I**: Node to be deleted is the leaf node
  - **Case II**: Node to be deleted has one child (left or right)
  - **Case III**: Node to be deleted has two children


tree < BT < BST
25,20,36,10,22,12,5,1,8,40,38,48,28,45,50
root
Deletion:
case 1: leaf node
case 2: one child
case 3: two children
1 5 8 10 12 13 20 22 25 28 36 37 38 40 45 48 50

# Deletion of a key from a BST

```
Algorithm:- Delete1BST (info, left, right, root, LOC, PAR)
            // When leaf node has no child or only one child
{
   1.  if ( ( LOC -> left = NULL)  and  ( LOC -> right = NULL))
          1.1 Child = NULL
       elseIf ( LOC -> left != NULL)
          1.1 Child = LOC -> left
       else
          1.1 Child = LOC -> right
   2. if ( PAR != NULL)
          2.1   if ( LOC = PAR -> left )
                     2.1.1 PAR -> left = Child
          2.1   else
                     2.1.1 PAR -> right = Child
       else
          2.1 root = Child
 }
```

```
void insert(int key)
{
    root = insertdata(root, key);
}
//recursive function
Node deletedata(Node root, int key)
{
    if(root == null)
        return root;
    if(key < root.data)
        root.left= deletedata(root.left, key);
    else if(key > root.data)
        root.right= deletedata(root.right, key);
    else
    {
    //case 1, 2
    if(root.left == null)
        return root.right;
    else if(root.right == null)
        return root.left;

    //case 3
```



# Searching for a key in a BST

**Algorithm:- SearchBST (info, left, right, root, key, LOC, PAR )**

**{**

**key is the value to be searched. This procedure find the location LOC of key and also the location PAR of the parent of the key.**

**1. If ( root = NULL) Then**
**1.1 Print ("Tree does not exist")**
**1.2 LOC = NULL and PAR = NULL**
**1.2 exit**
**2. PAR = NULL, LOC = NULL**
**3. ptr = root**
**4. While ( ptr != NULL )**
**4.1 if ( key = ptr -> info ) then**
**4.1.1 LOC = ptr**
**4.1.2 print PAR AND LOC**
**4.1.3 exit**
**4.1 else if ( key < ptr -> info ) then**
**4.1.1 PAR = ptr**
**4.1.2 ptr = ptr -> left**
**4.1 else**
**4.1.1 PAR = ptr**
**4.1.2 ptr = ptr -> right**
**5. If ( LOC = NULL ) then**
**5.1 Print ("Key not found")**

**}**

```
private boolean search(BSTNode r, int val)
    {
        boolean found = false;
        while ((r != null) && !found)
        {
            int rval = r.getData();
            if (val < rval)
                r = r.getLeft();
            else if (val > rval)
                r = r.getRight();
            else
            {
                found = true;
                break;
            }
            found = search(r, val);
        }
        return found;
    }
```

# Thanks