Traversal of BT

- Tree traversal is one of the most common operations performed in tree data structure.
- In a traversal of a binary tree, each element of
- The binary tree is visited exactly once in a systematic manner.
  - Preorder
  - Inorder
  - Postorder

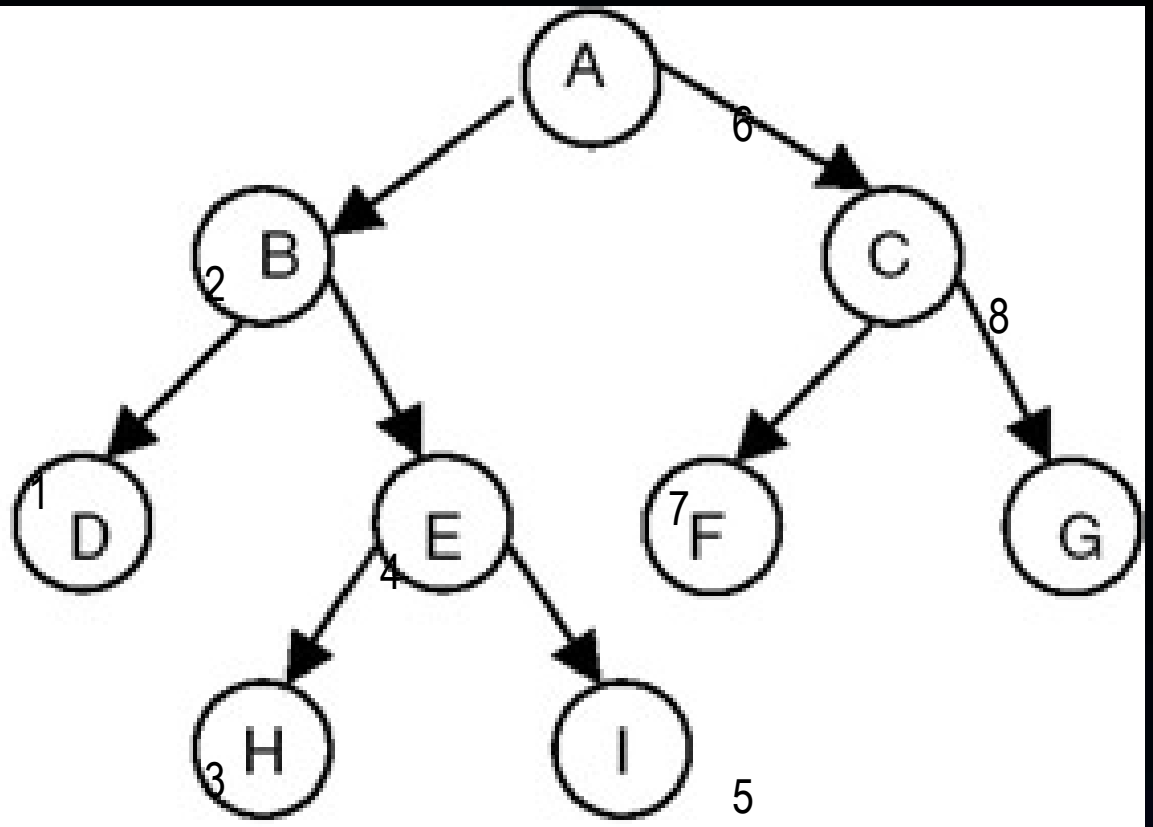
In order traversal (LNR)

The inorder traversal of a non empty binary tree is defined as:

1. Traverse the left subtree in inorder (L)
2. Visit the root node (n)
3. Traverse the right subtree in in order (R)

```
void inorder( struct node *ptr)
{
if(ptr!=NULL)
    inorder(ptr->lchild);
    cout<<ptr->info;
    inorder(ptr->rchild);
 }
```

In order traversal



Inorder : DBHEIAFCG

Preorder : ABDEHICFG

Postorder : DHIEBFGCA

- In an inorder traversal , left subtree is traversed recursively in inorder before visiting the root node.
- After visiting the root node , the right subtree is taken up and it is traversed recursively again in inorder.

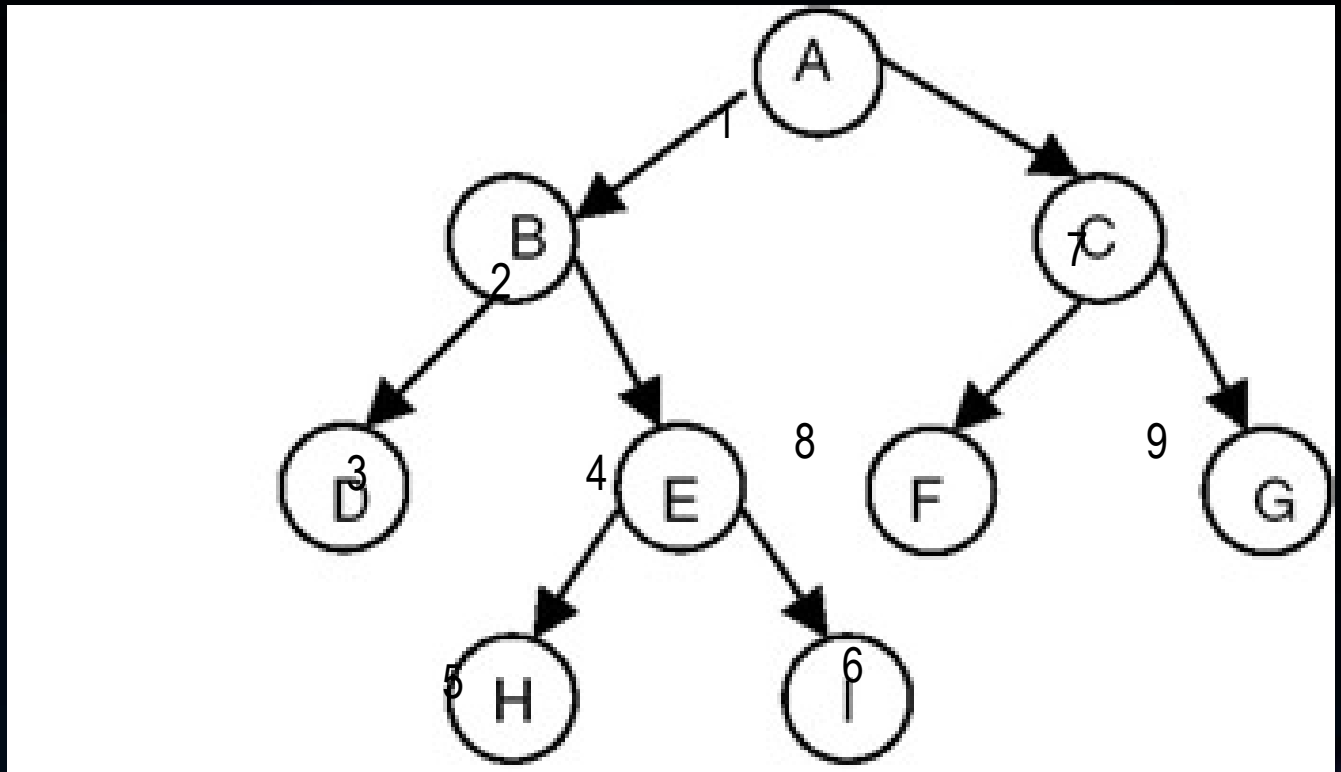
Preorder traversal (NLR)

The preorder traversal of a non empty binary tree is defined as:

1. Visit the root node (n)
2. Traverse the left subtree in preorder (L)
3. Traverse the right subtree in pre order (R)

```
Void preorder( struct node *ptr)
 {
 if(ptr!=NULL)
   {
      cout<<ptr->info;
      preorder(ptr->lchild);
      preorder(ptr->rchild);
   }
}
```

Preorder traversal



Inorder : DBHEIAFCG

Preorder : ABDEHICFG

Postorder : DHIEBFGCA

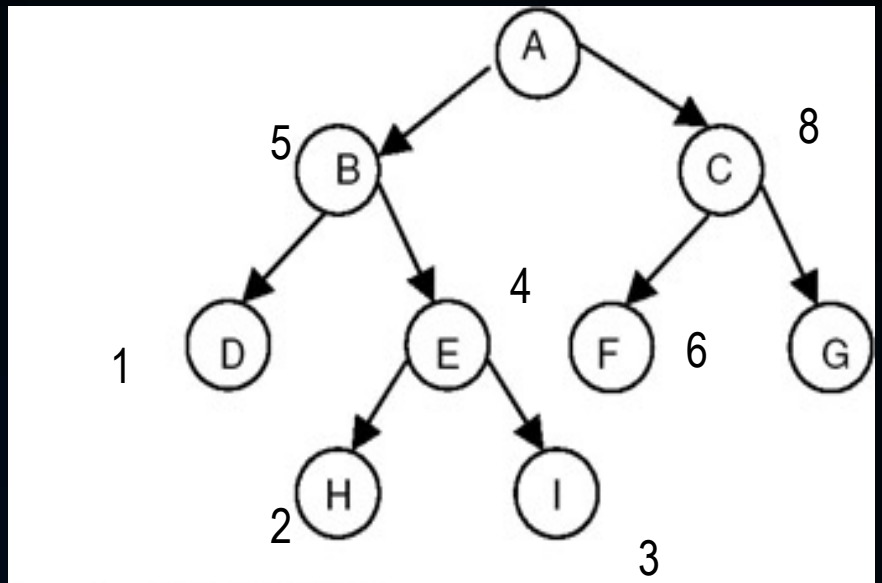
- In preorder traversal the root node is visited before traversing the left and right subtrees.
- The preorder notation recursive in nature , in left subtree and right subtree.

Post order traversal(LRN)

- The post order traversal of non empty binary tree is :-
- 1. traverse left subtree in post order.(L)
- 2. Traverse the right subtree in post order.(R)
- 3. visit the root node.(N)

```
void postorder( struct node *ptr)
  {
  if (ptr!=NULL)
   {
      postorder(ptr->left);
      postorder(ptr->right);
      cout<< ptr->num;
  }
}
```

Post order traversal



Inorder : DBHEIAFCG

Preorder : ABDEHICFG

Postorder : DHIEBFGCA


Binary Search
Tree

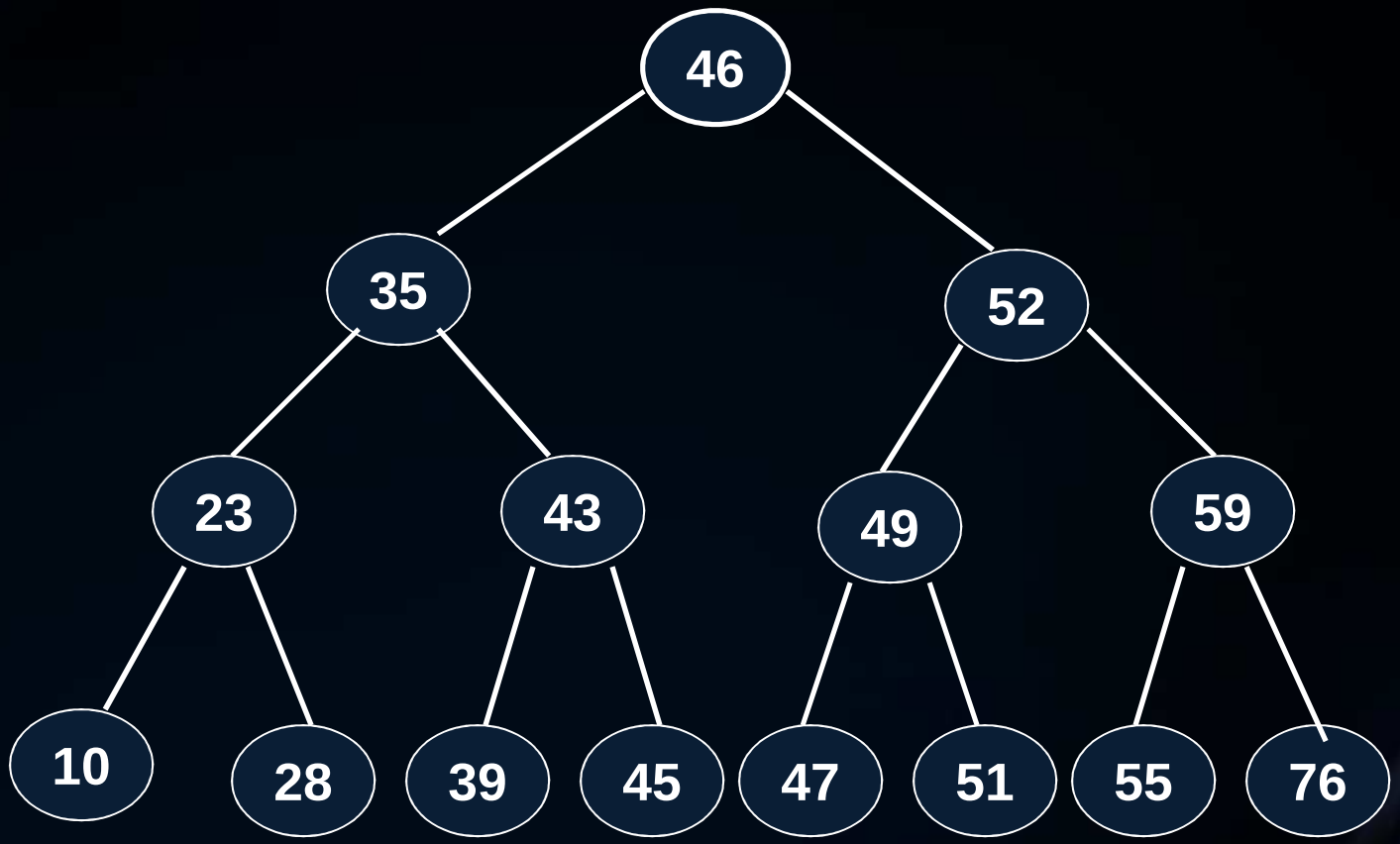
- A binary search tree is a binary tree which is either empty or satisfies the following rules.
- 1. the value of the key in the left child or left subtree is less than the value of the root.

2. The value of the key in the right or right subtree is more than or equal to the value of the root.
3. All the subtree of the left and right children observe the above two rules.

Binary Search Tree

- A binary tree T is termed as binary search tree (or binary sorted tree) if each node N of T satisfies the following property:
  - The value N is greater than every value in the left sub-tree of N and is less than every value in the right sub-tree of N.

A Sample Binary Search Tree

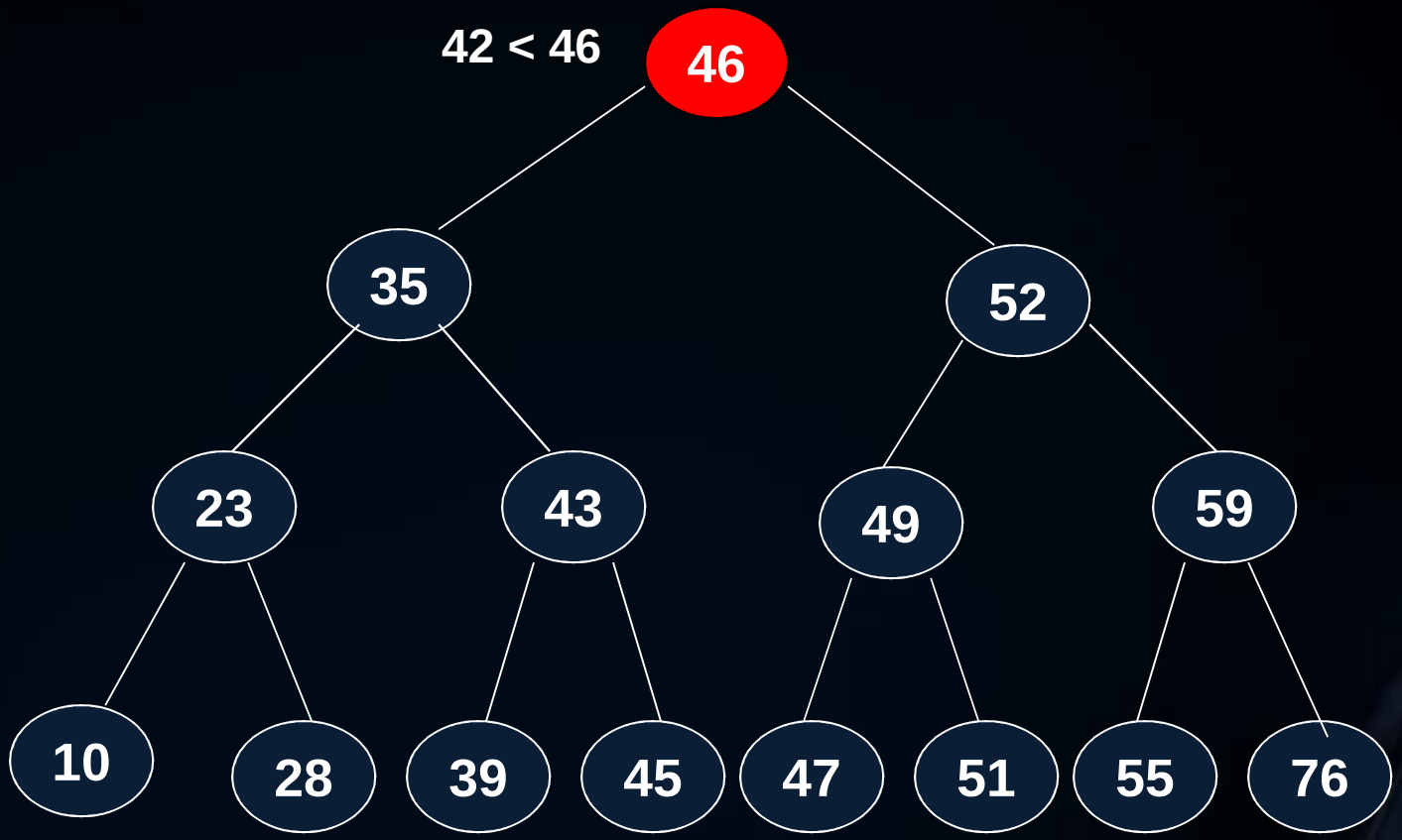


Operations on a Binary Search Tree

- Searching
- Insertion
- Deletion
- Traversal

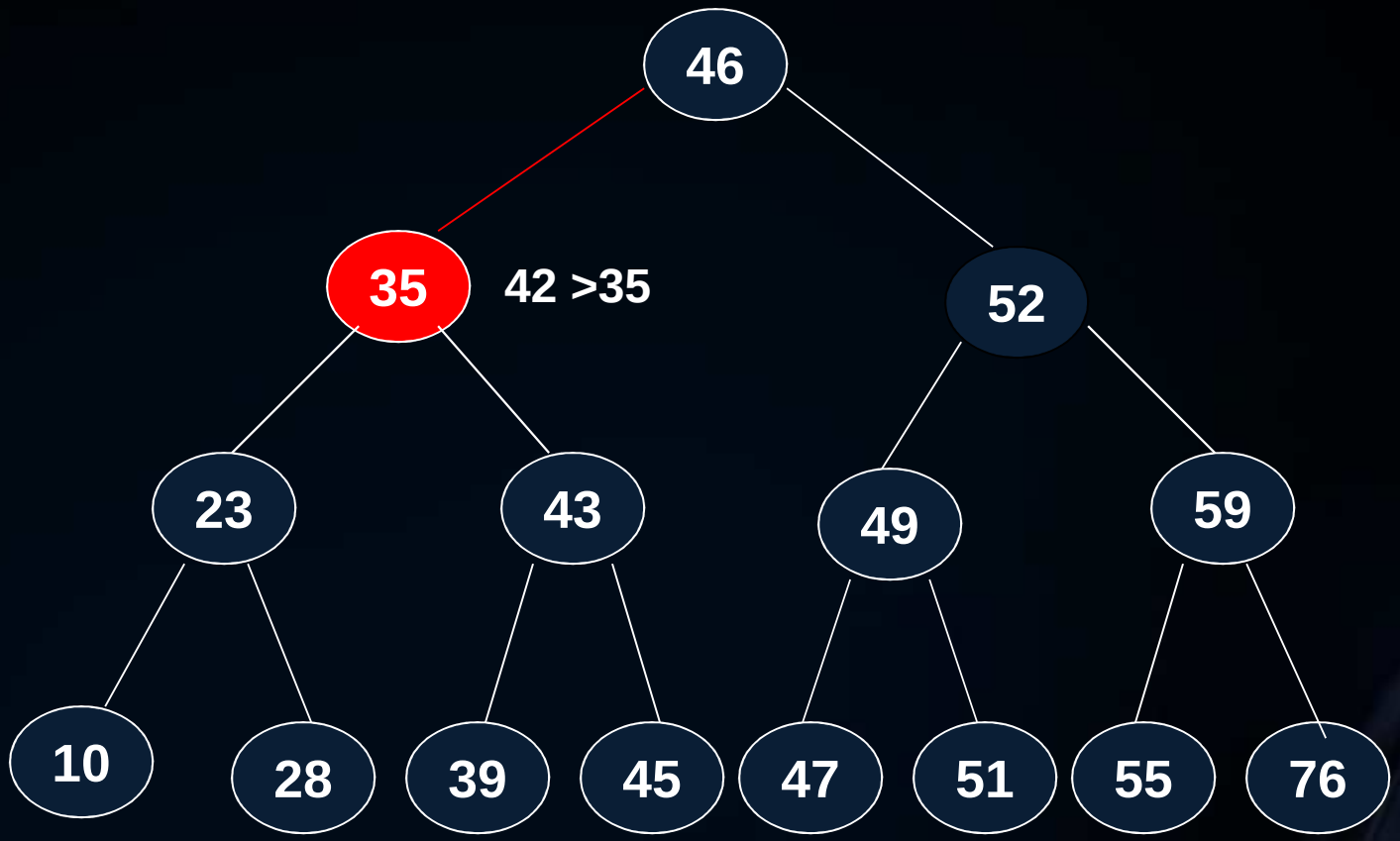
Insertion

NEW KEY -> 42



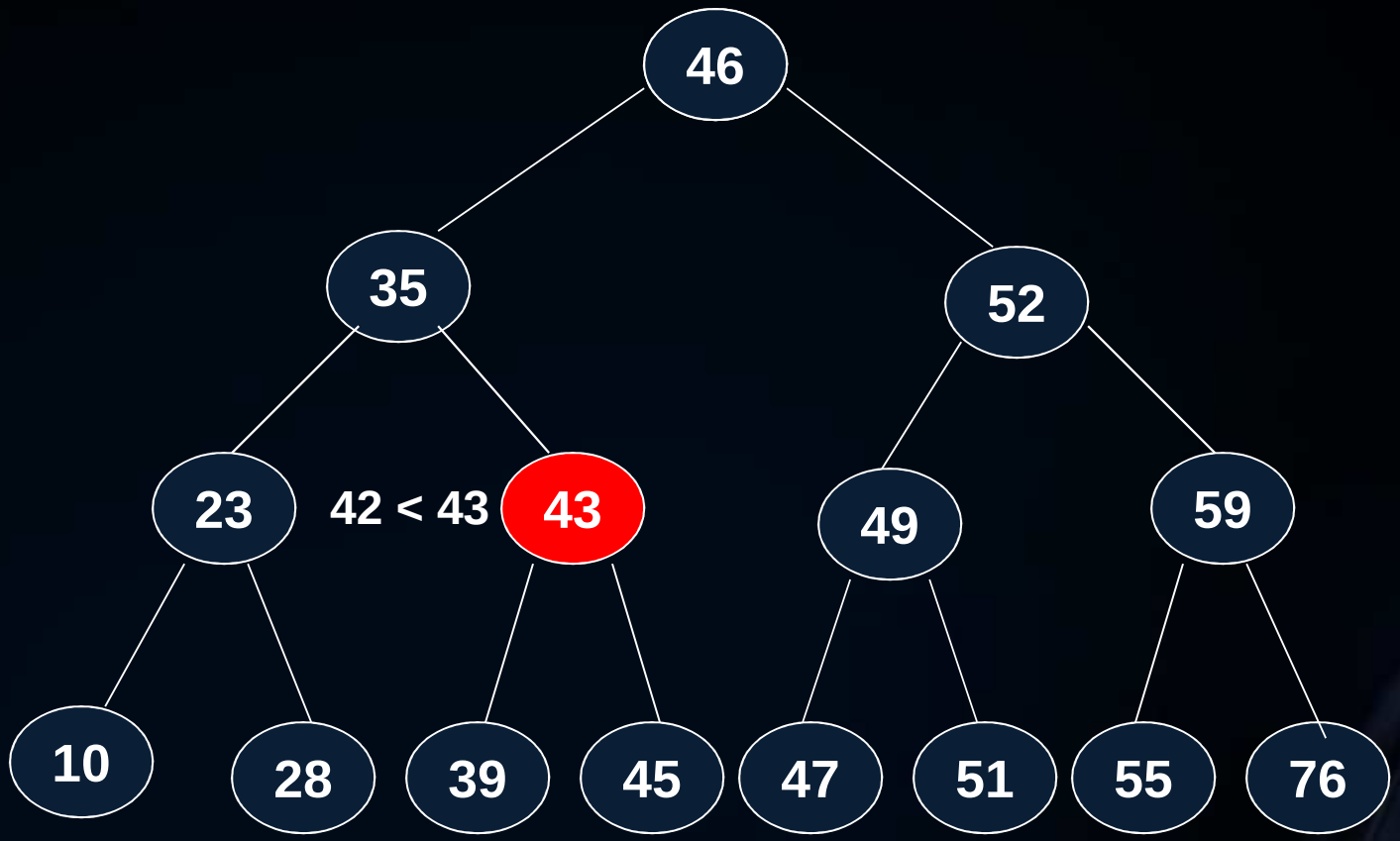
Insertion

NEW KEY -> 42



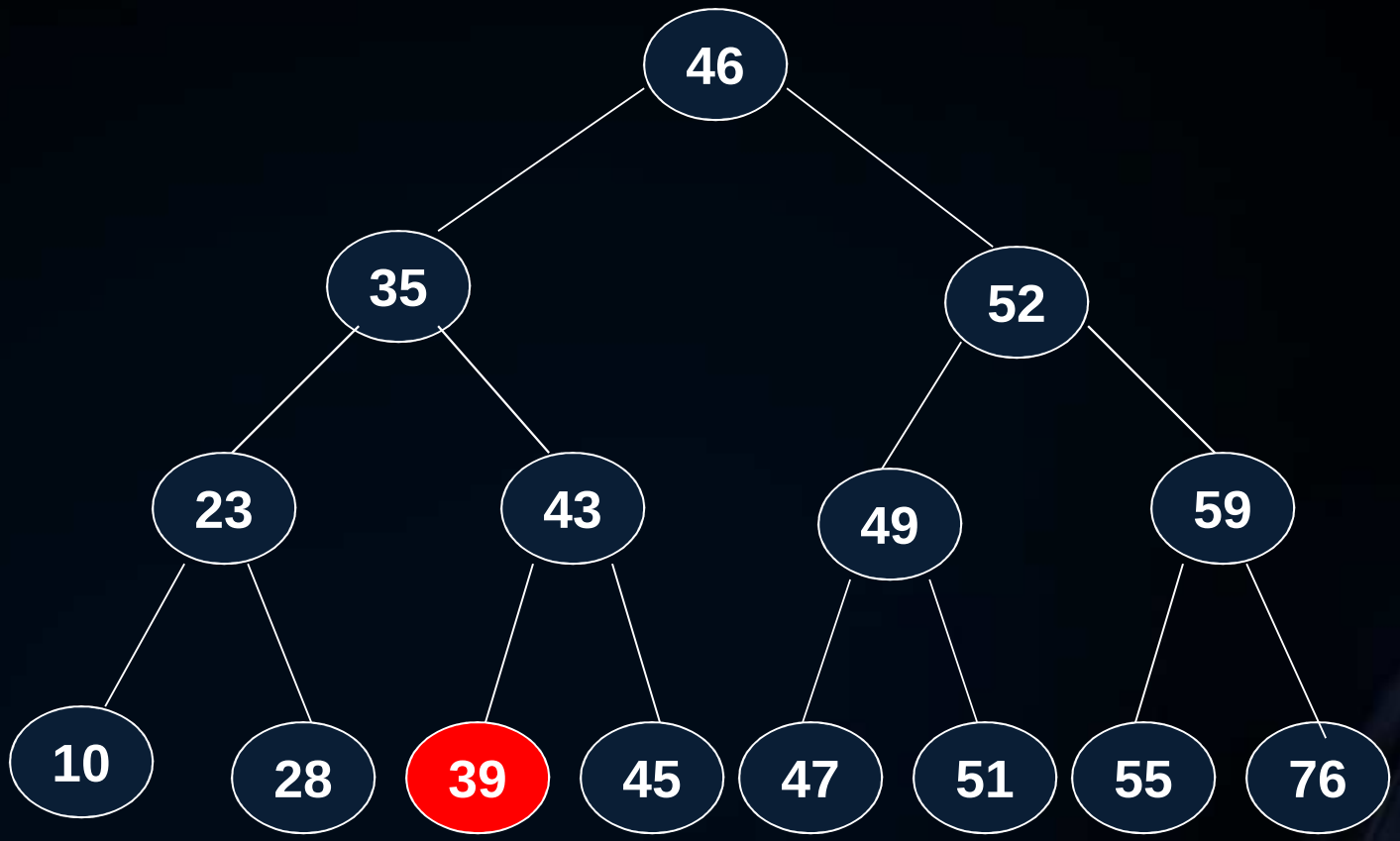
Insertion

NEW KEY -> 42



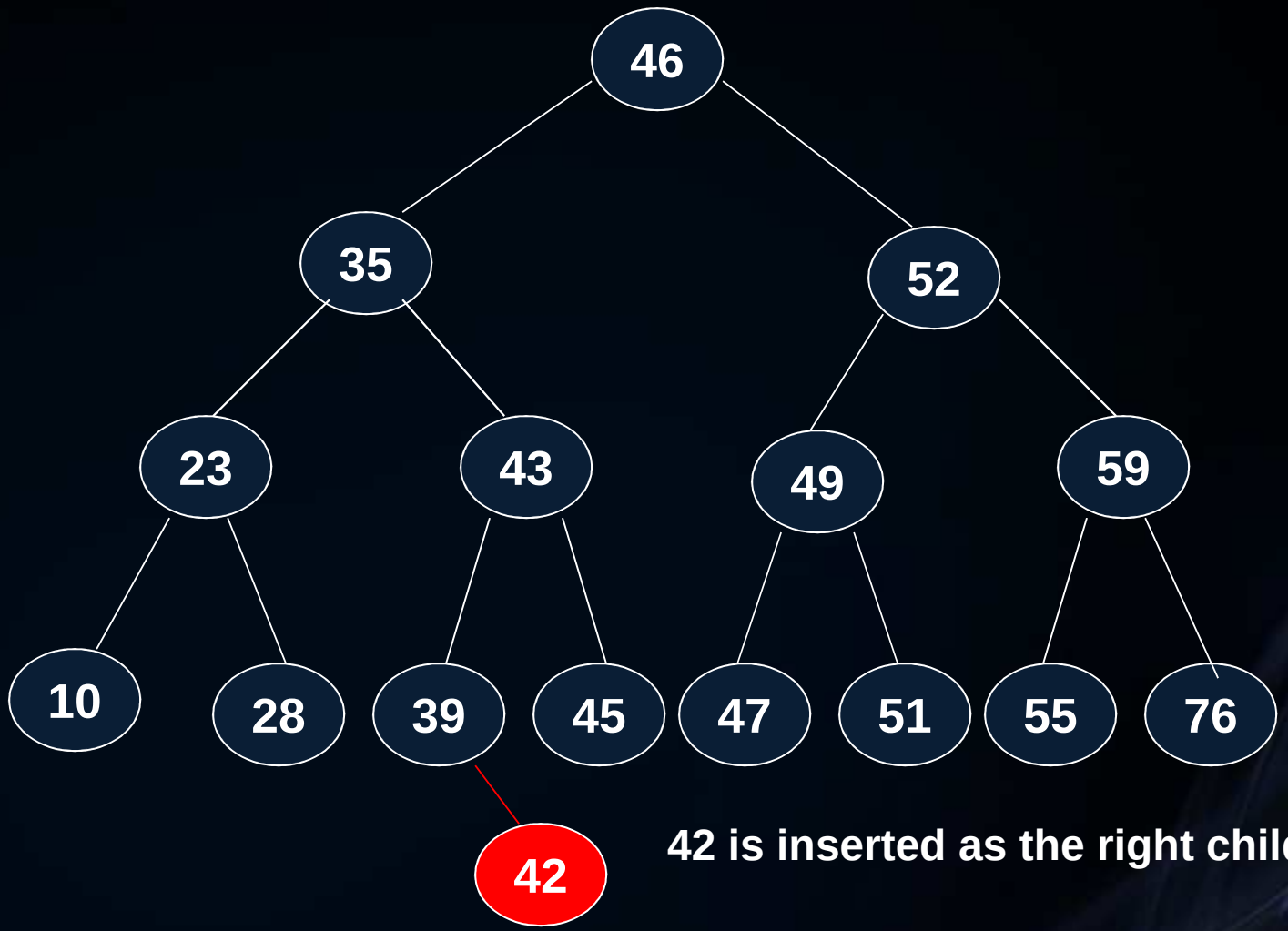
Insertion

NEW KEY -> 42



Insertion

NEW KEY -> 42



42 is inserted as the right child

Insertion of nodes on BST

```
tree *insert(tree *root, int digit)
{
 if (root==NULL)
  {
 root= new tree;
 root->lchild=root->rchild=NULL;
 root->data=digit;
}
```

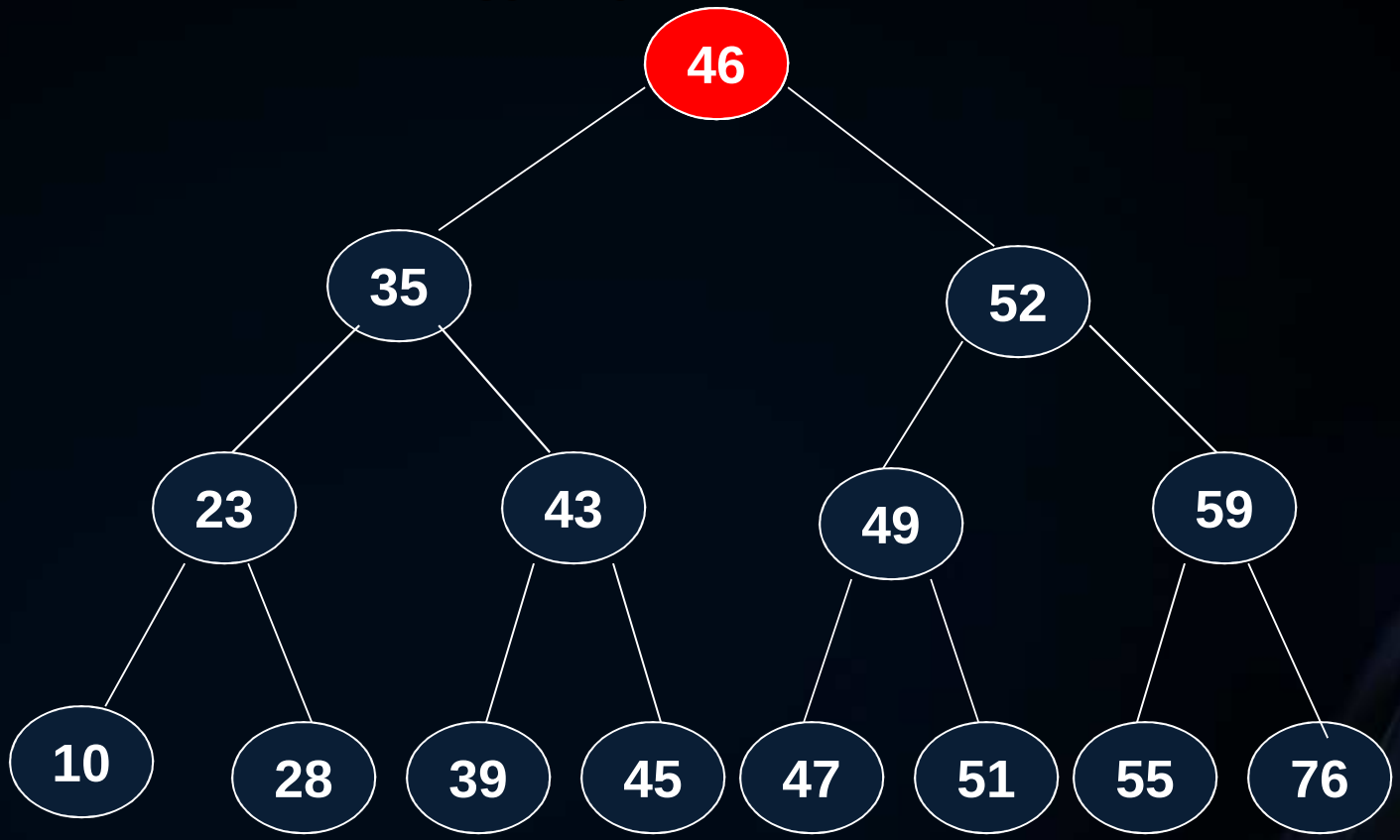
```
 else
{
  if(digit<root->data)
      root->lchild=insert(root->lchild, digit);
 else
    if(digit>root->data)
      root->rchild=insert(root->rchild, digit);
   else
    if(digit==root->data)
   {  puts("duplicate  node : program exited);
          exit(0);
  }
  return root;
}
```

Searching

- We start from the root node R.
- If value in root node is equal to ITEM, search successfully completed.
- If ITEM is less than the value in the root node R, we proceed to its left child
- If ITEM is greater than the value in the node R, we proceed to its right child.
- Continue until we find ITEM or reach the dead end that is leaf node.

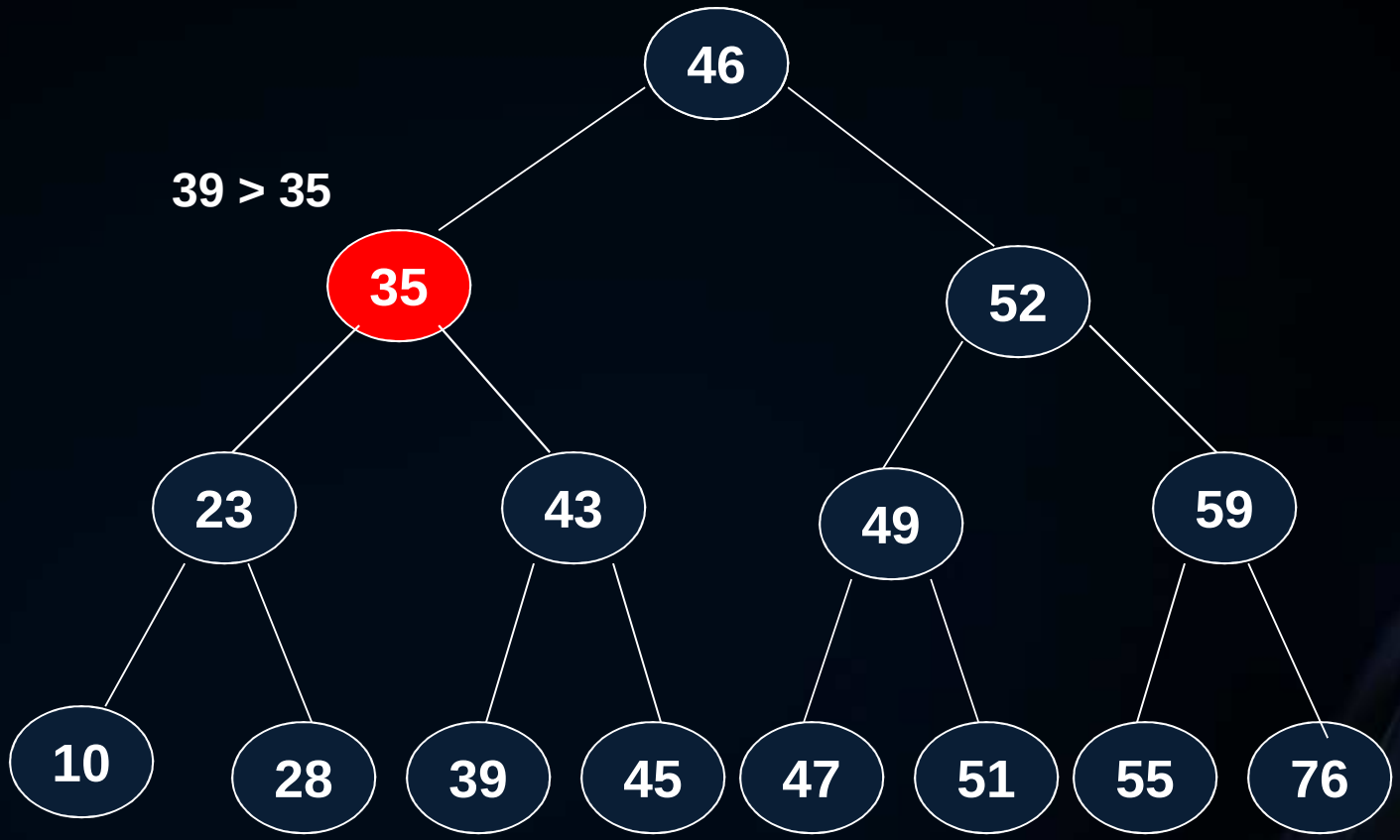
Searching

SEARCH KEY -> 39



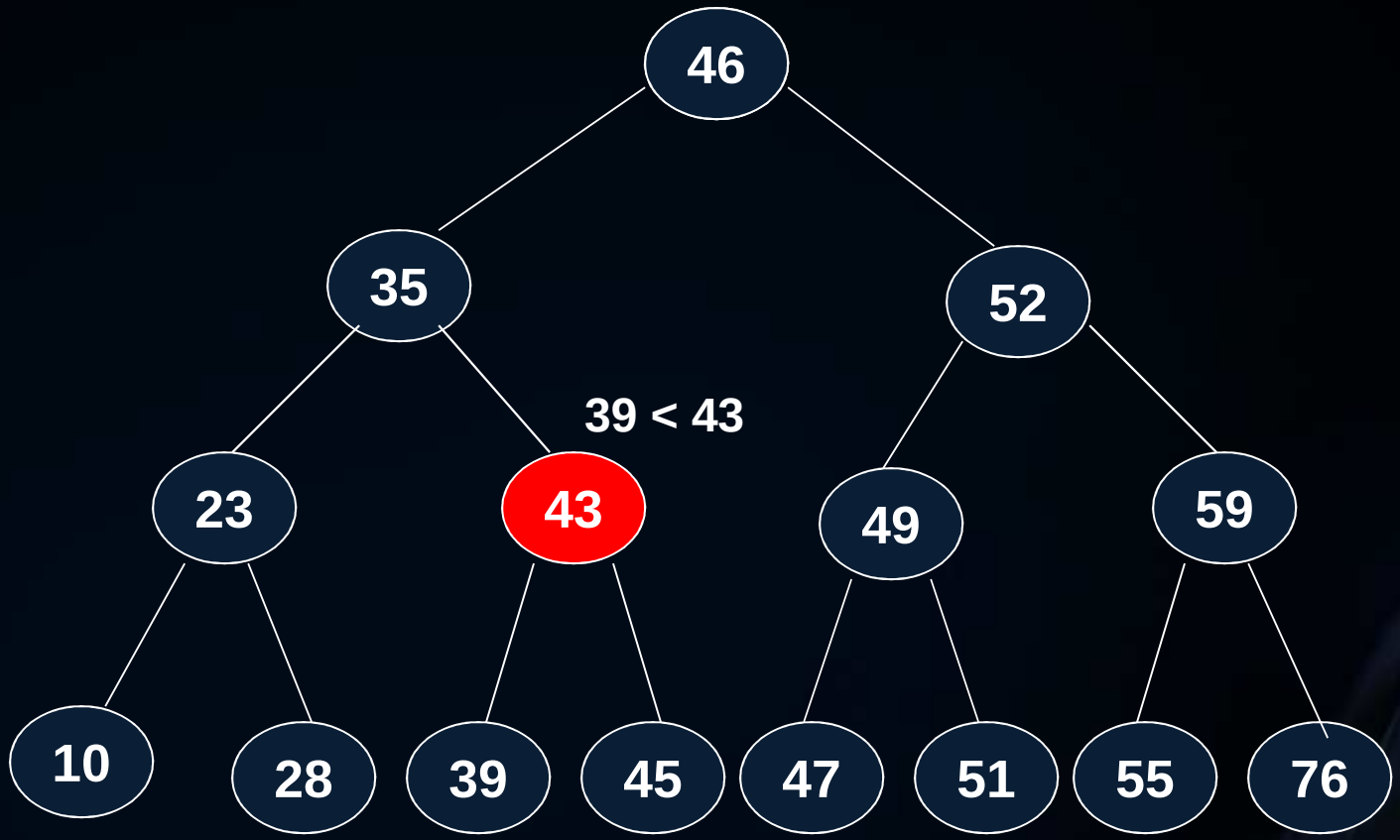
Searching

SEARCH KEY -> 39



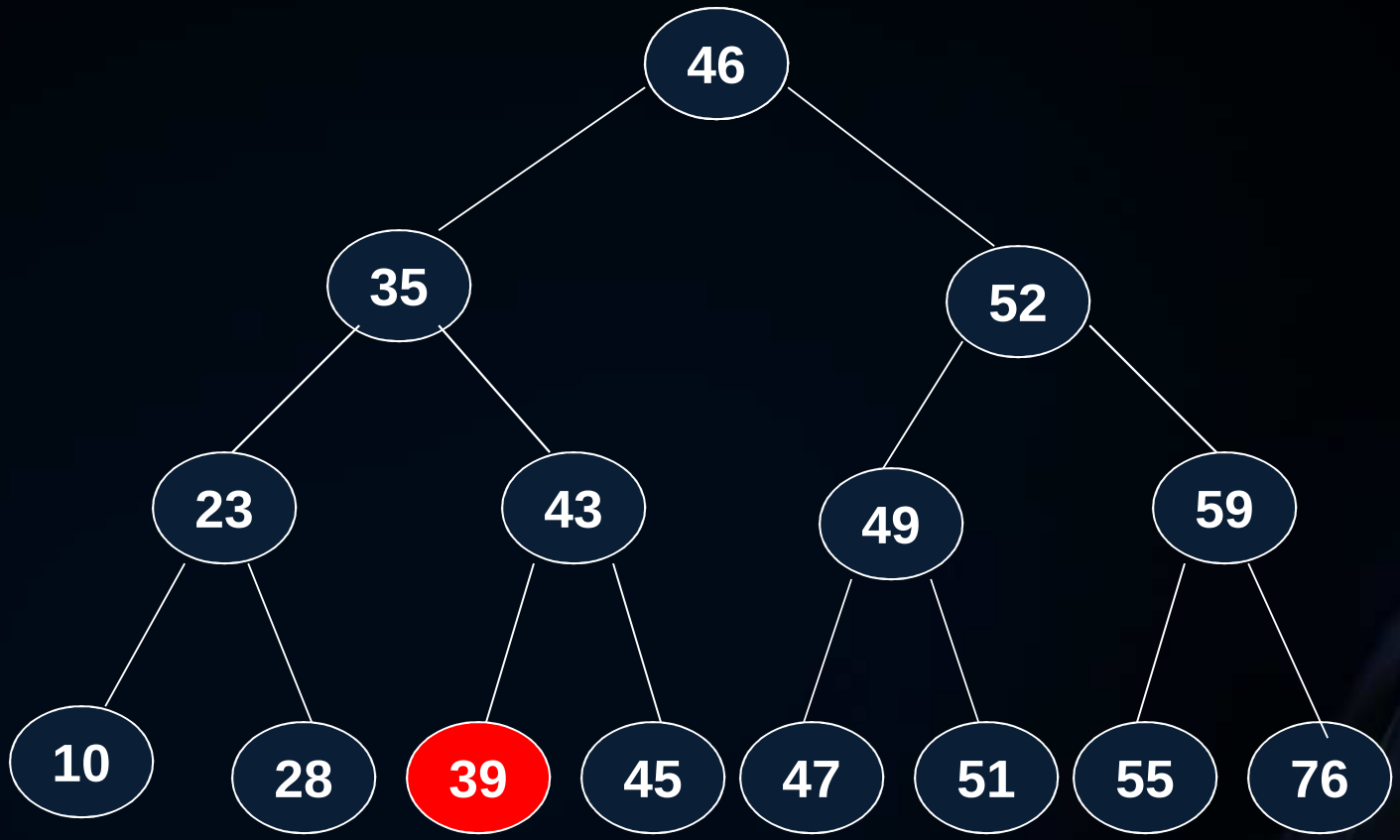
Searching

SEARCH KEY -> 39



Searching

SEARCH KEY -> 39



FOUND!!!!!!!!!

Binary search for a node

```
1. Void search (tree * root, int item)
2. if (root==NULL) step 3
3. puts("the  number does not exist");
4  else if(item ==root->data) step 5
5    cout<<item;
6. else if(item<root->data) step 7
7.    search(root->lchild,item);
8  else search(root->rchild,item);
9     end.
```

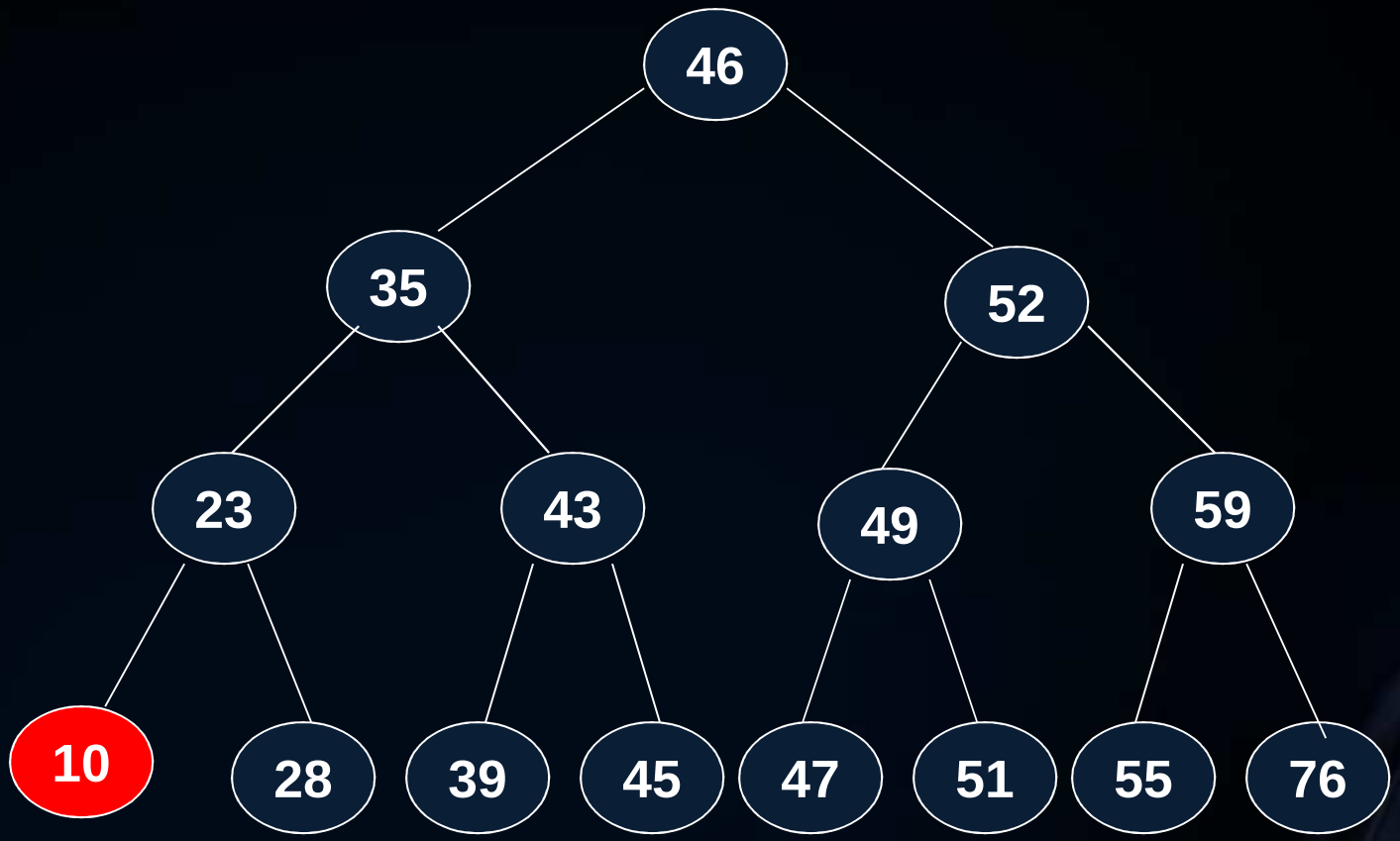
```
Void search(tree *root, int digit)
 {
 if (root==NULL)
  puts("the number does not exist");
 else
        if(digit==root->data)
       cout<<digit;
 else
       if(digit<root->data)
       search(root->lchild,digit);
  else
       search(digit->rchild,digit);
}
```

Deletion

- The tree is searched starting from the root node.
- If node N contains the information to be deleted; PARENT (N) is the parent of N and SUCC(N) denotes the inorder successor of N.
- Deletion of node N depends on the number of child nodes; three cases are:
  - ***N is the leaf node:*** N is deleted by simply setting the pointer of N in the parent node PARENT (N) by NULL value.
  - ***N has exactly one child:*** N is deleted by simply replacing the pointer of N in PARENT (N) by the pointer of the only child of N.
  - ***N has two children:*** N is deleted by first deleting SUCC (N) from T and then replaces the data content in node N by the data content in node SUCC (N).

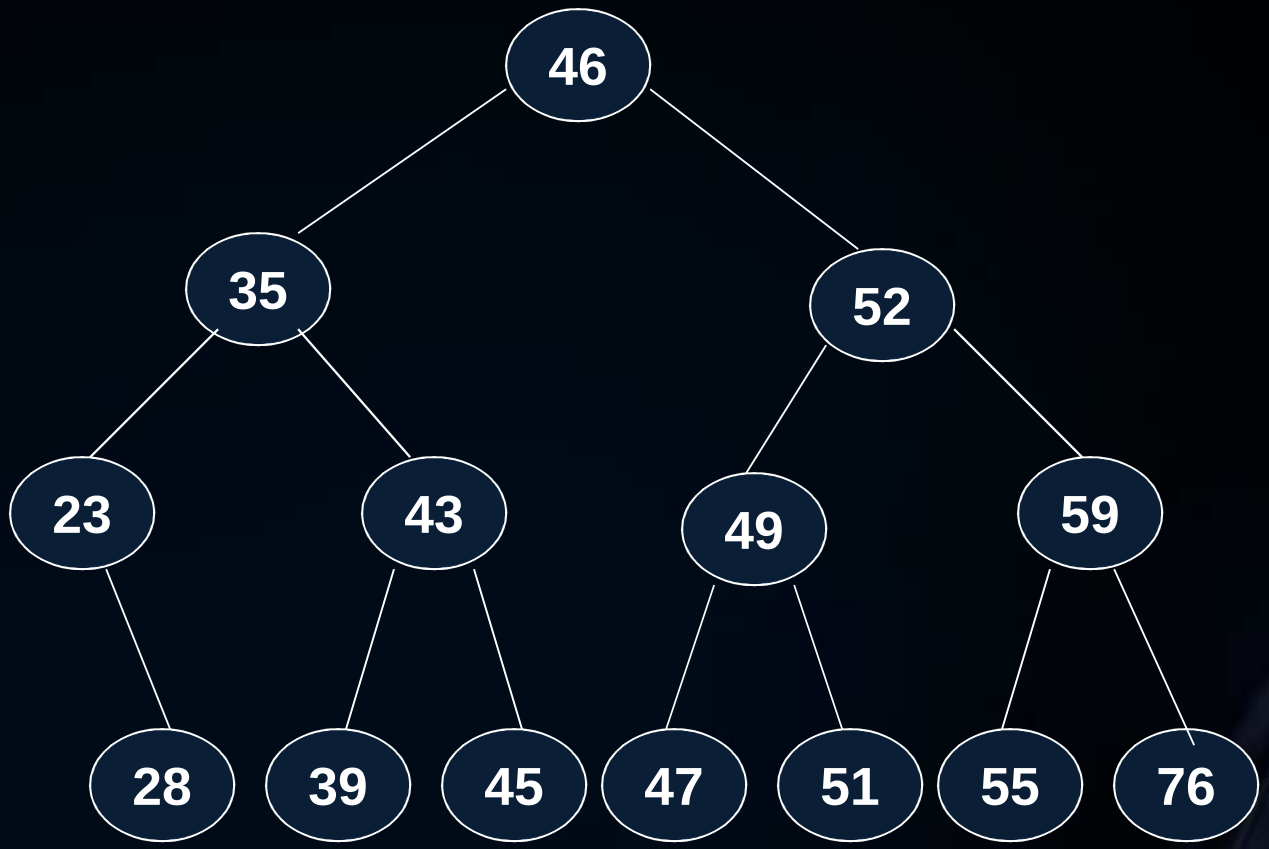
Deletion

KEY -> 10



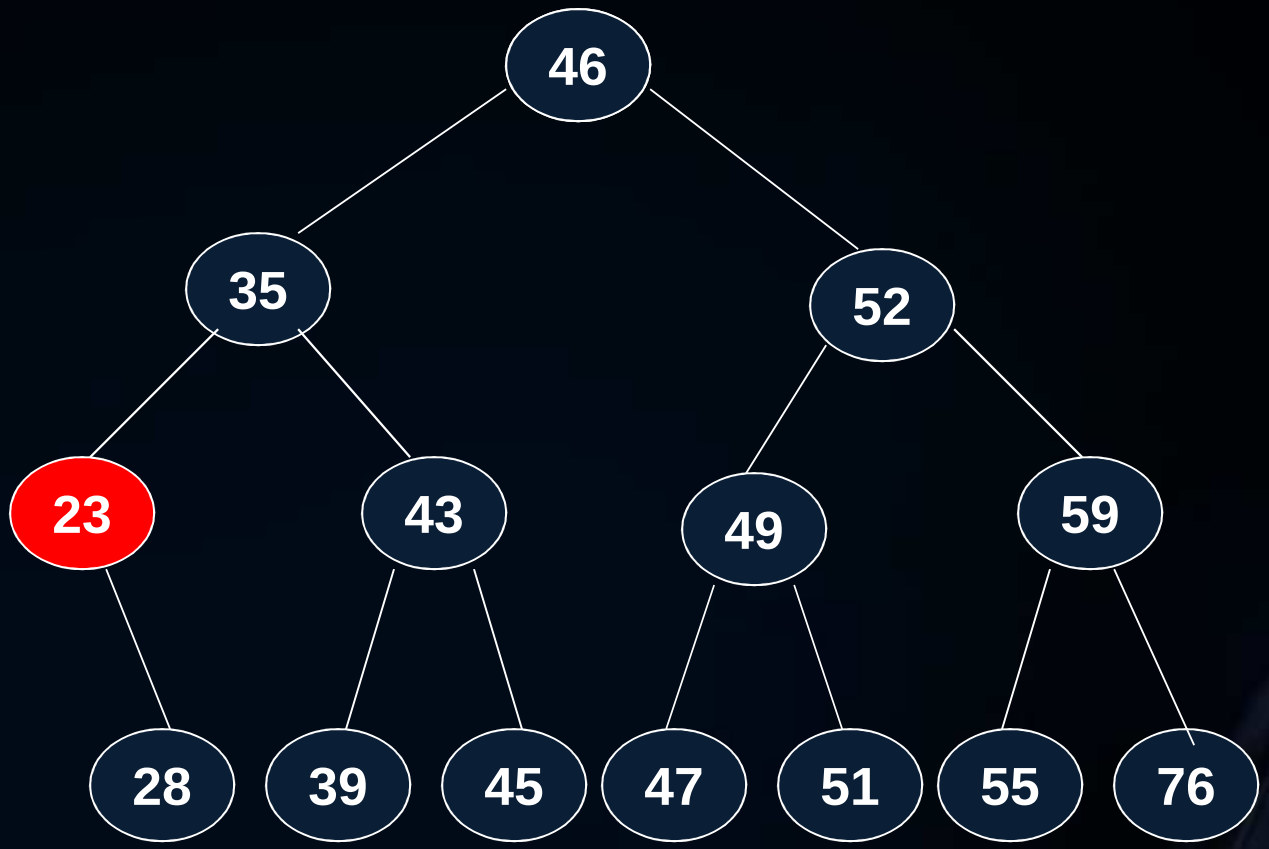
Deletion

KEY -> 10



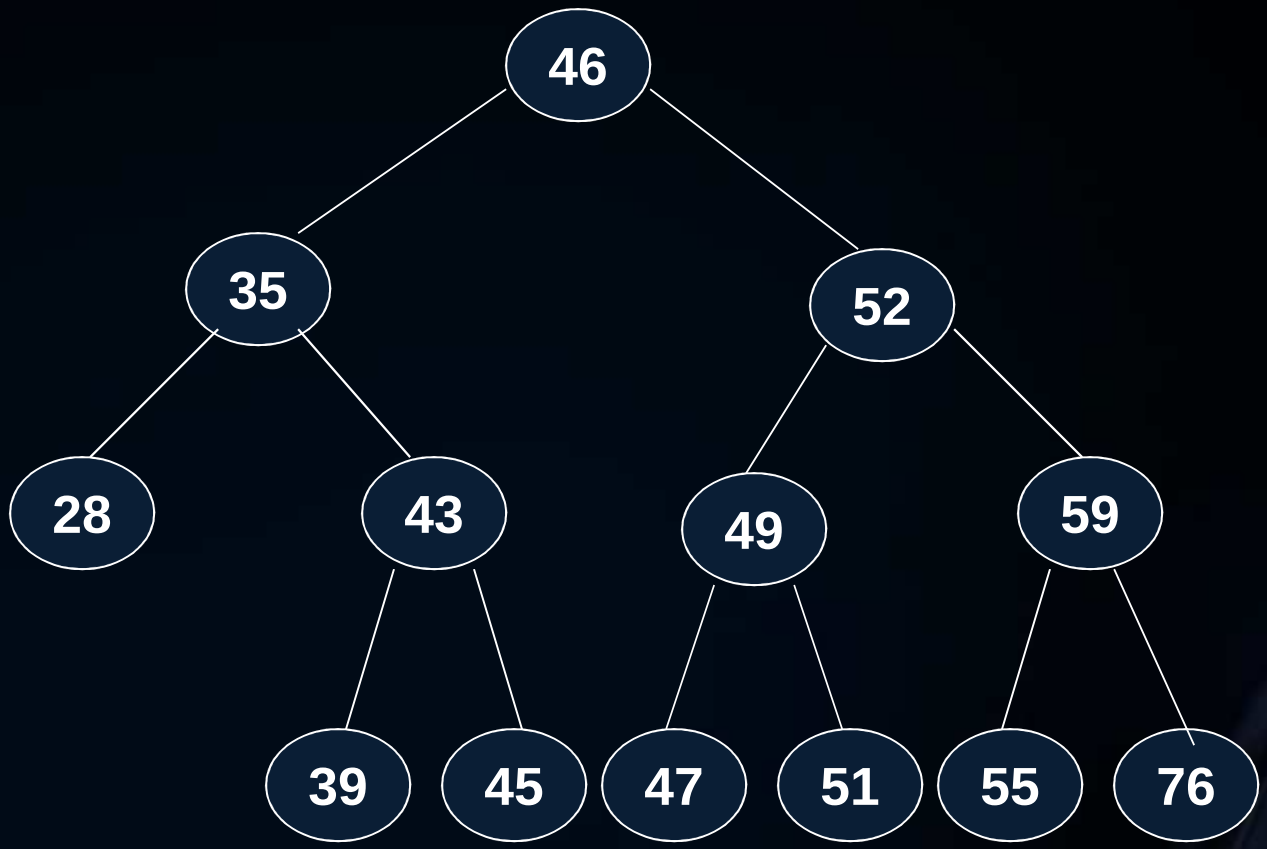
Deletion

KEY -> 23



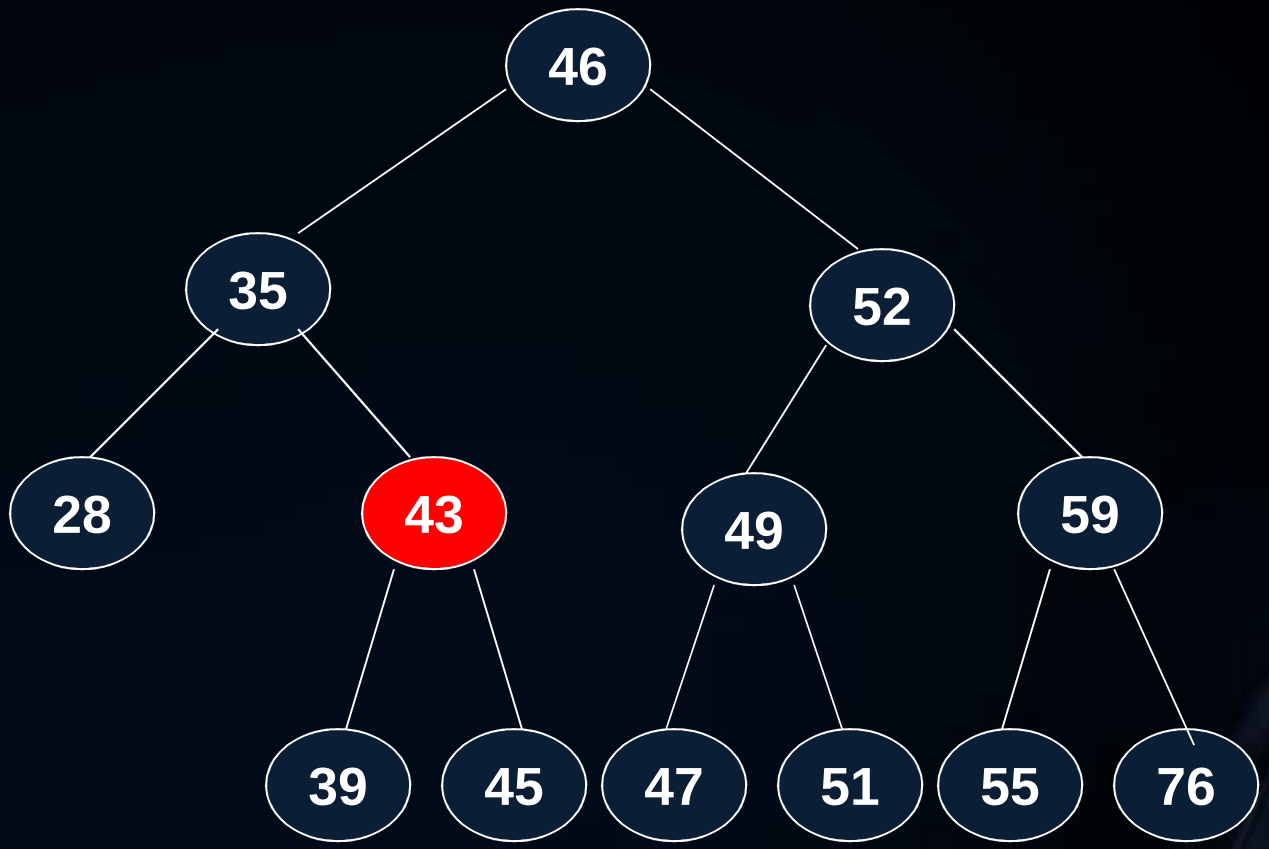
Deletion

KEY -> 23



Deletion

KEY -> 43



Deletion

KEY -> 43