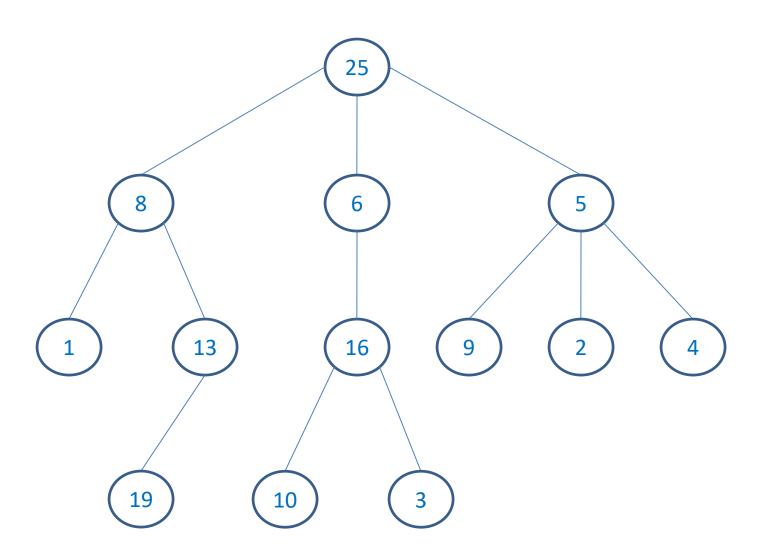
Tree: A Non-Linear Data Structure

Joy Mukherjee

Example of a Tree



Tree

- ☐ A tree with n nodes has n-1 edges
- ☐ A tree does not have a cycle
- ☐ A tree is connected
- ☐ There exists one and only one path between any two

nodes

Definitions

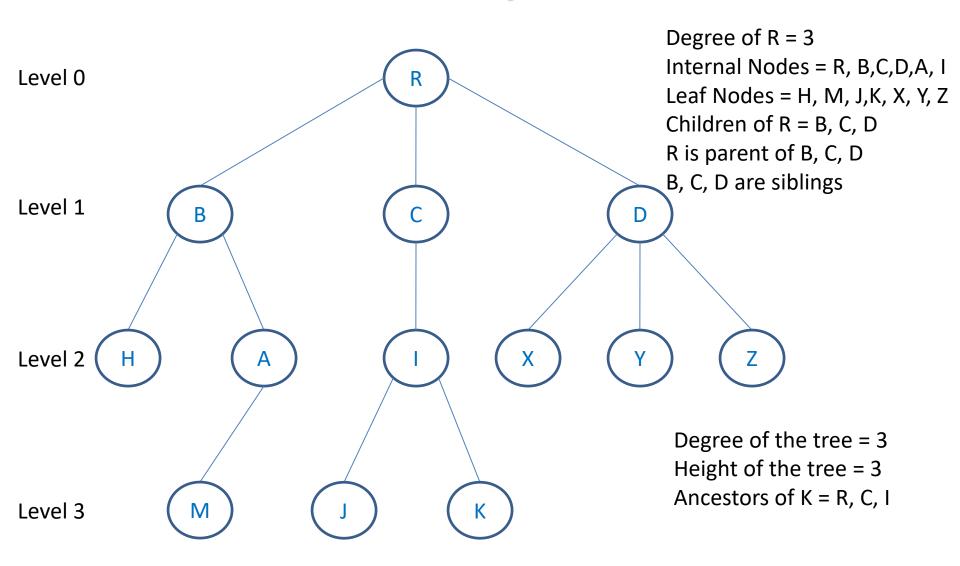
- Degree of a node: No of subtrees of the node
- ☐ Leaf Node: Node having degree zero
- ☐ Internal Node: Node having degree at least one
- ☐ Children of a node: Roots of the subtrees of the node
- ☐ Siblings: Children of the same parent.
- □ Degree of a tree: Maximum degree of any node in the

tree

Definitions

- ☐ Ancestors of a node: All the nodes along the path from the root to that node.
- \Box Level of root = 0.
- ☐ Level of a child = 1+Level of its parent
- ☐ Height/Depth of a tree: Maximum level of any node in the tree

Example

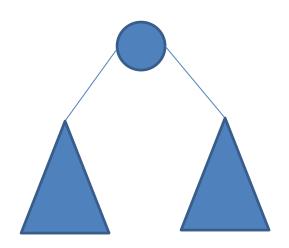


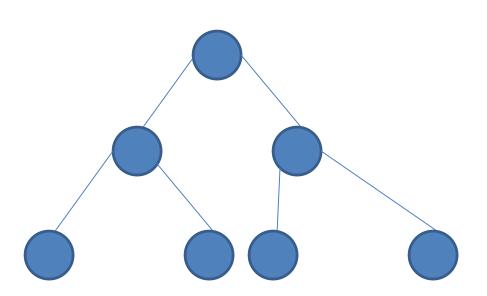
Binary Tree

- ☐ A binary tree is a finite set of nodes that is either empty or consists of a root and two disjoint binary trees called the left subtree and the right subtree.
- ☐ Every node has at most two children.
- \square Maximum no of nodes at level $x = 2^x$

Binary Tree: Recursive Definition

- ☐ An empty node is a binary tree
- ☐ A single node, called root, is a binary tree
- ☐ It has at most two disjoint subtrees which are also binary trees. This disjoint property ensures that the tree is acyclic and there is a unique path between any two nodes of the tree.





Binary Tree

- ☐ Each node has either 0, 1 or 2 children.
- ☐ Maximum no of nodes in a binary tree of height h is n = $2^0 + 2^1 + 2^2 + ... + 2^h = 2^{h+1} 1$
- ☐ Given a binary tree with n nodes, the minimum height of the binary tree $h_{min} = log_2(n+1) 1$ (Each node has 2 children)
- \Box Given a binary tree with n nodes, the maximum height of the binary tree $h_{max} = n 1$ (Each node has a single child)

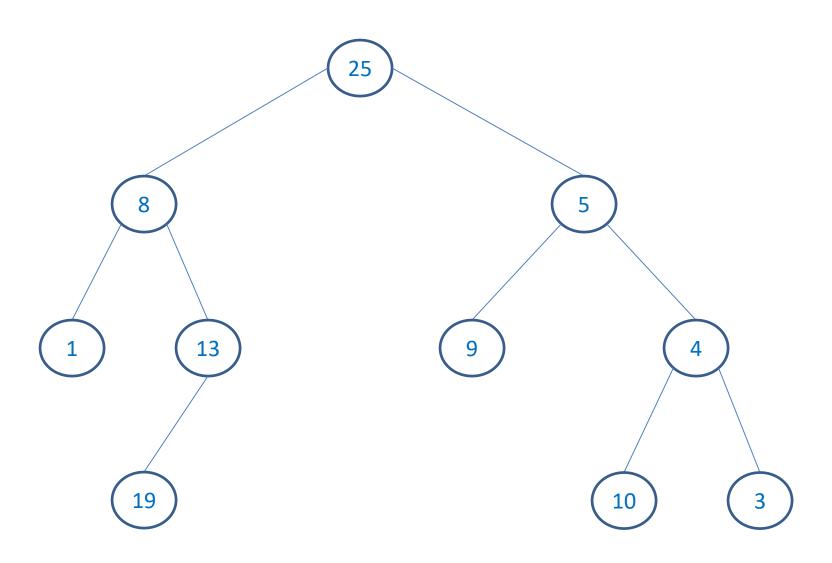
Binary Tree

□ For any non-empty binary tree, if N₀ be the no of leaf nodes, and N₂ be the no of nodes having degree 2, then N₀ = N₂ + 1.

Binary Tree Data Structure

```
typedef struct node {
typedef struct node {
                                 int data;
 int data;
                                 struct node *next;
 struct node *left;
                               }node;
 struct node *right;
                               void linearTraversal(node *h)
}node;
                                    while(h!= NULL) {
void preorder(node *);
                                     printf("%d", h->data);
                                     h = h->next;
void inorder(node *);
void postorder(node *);
```

Example



Binary Tree Traversals

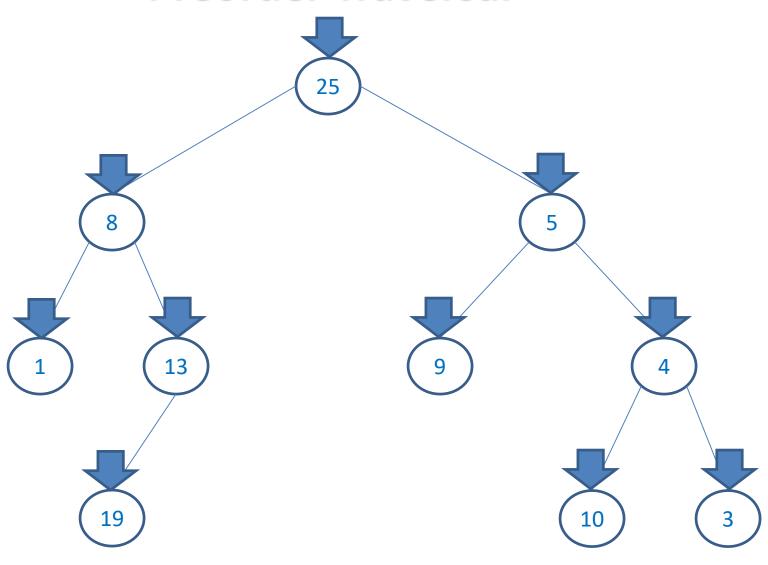
- ☐ Preorder: Print Root -> Left Subtree -> Right Subtree
- ☐ Inorder: Left Subtree -> Print Root -> Right Subtree
- ☐ Postorder: Left Subtree -> Right Subtree-> Print Root
- ☐ In all the traversals, Left subtree is traversed before

right subtree. (Left -> Right)

Preorder Traversal

```
void preorder(node *root)
  if(root != NULL) {
    printf("%4d", root->data);
    preorder(root->left);
    preorder(root->right);
```

Preorder Traversal

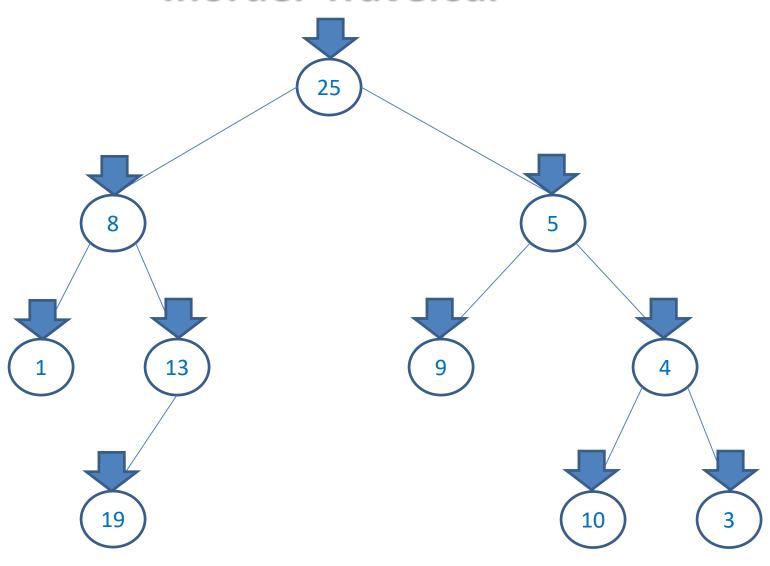


25, 8, 1, 13, 19, 5, 9, 4, 10, 3

Inorder Traversal

```
void inorder(node *root)
  if(root != NULL) {
    inorder(root->left);
    printf("%4d", root->data);
     inorder(root->right);
```

Inorder Traversal

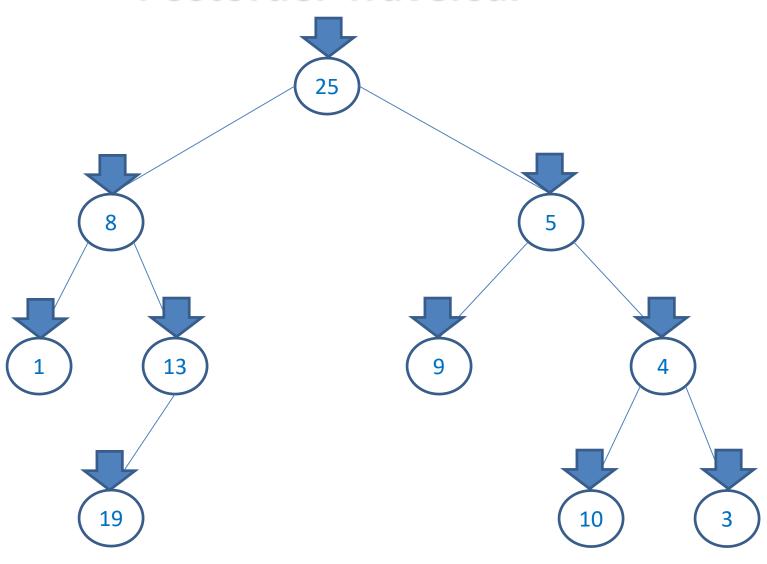


1, 8, 19, 13, 25, 9, 5, 10, 4,3

Postorder Traversal

```
void postorder(node *root)
  if(root != NULL) {
    postorder(root->left);
    postorder(root->right);
    printf("%4d", root->data);
```

Postorder Traversal



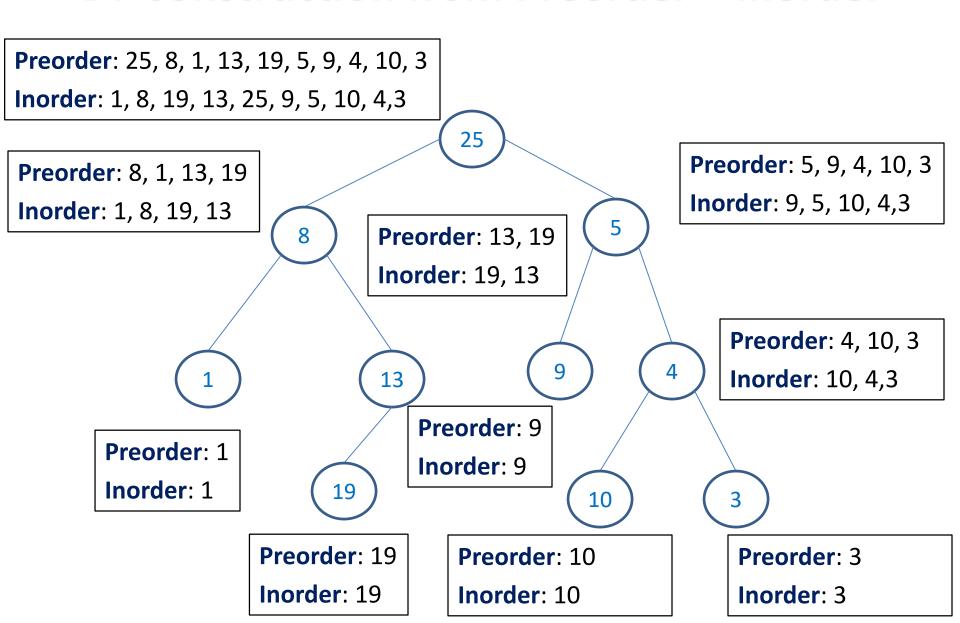
1, 19, 13, 8, 9, 10, 3, 4, 5, 25

Binary Tree Construction

- ☐ Inorder + Preorder -> Unique Binary Tree
- ☐ Inorder + Postorder -> Unique Binary Tree
- ☐ Preorder + Postorder -> Unique Binary Tree construction is not possible, in general.

- ☐ Preorder: Root -> Left -> Right
- ☐ Inorder: Left -> Root-> Right

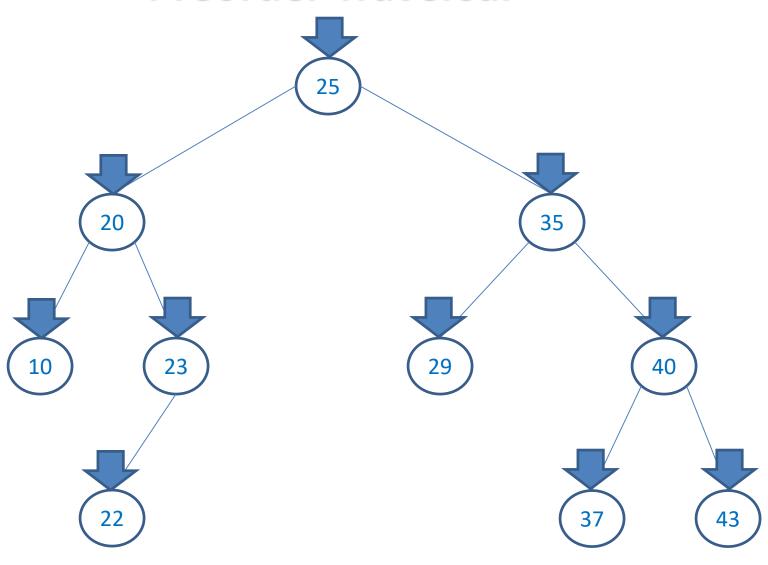
BT Construction from Preorder + Inorder



Binary Search Tree

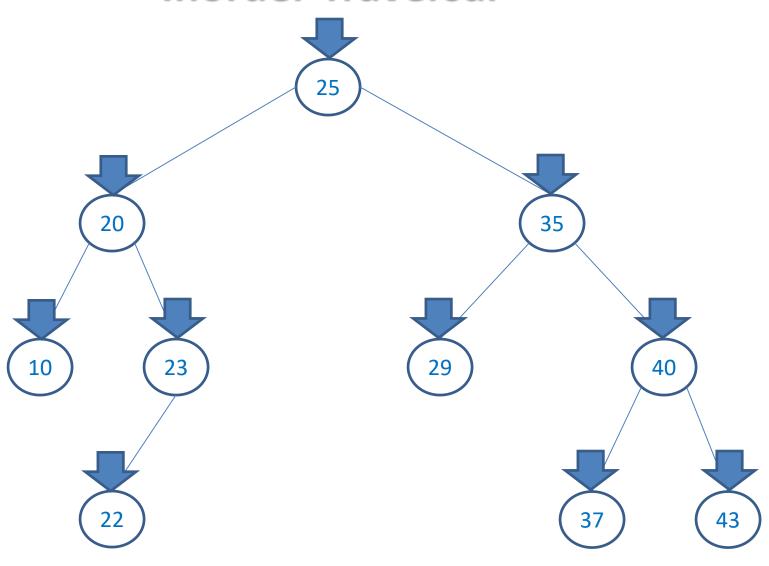
- ☐ A Binary Search Tree (BST) is a binary tree. It may be empty. If it is not empty, then it satisfies the following properties:
- i. The root has a key.
- ii. The keys (if any) in the left subtree are smaller than the key in the root.
- iii. The keys (if any) in the right subtree are larger than the key in the root.
- iv. The left and right subtrees are also BSTs.

Preorder Traversal



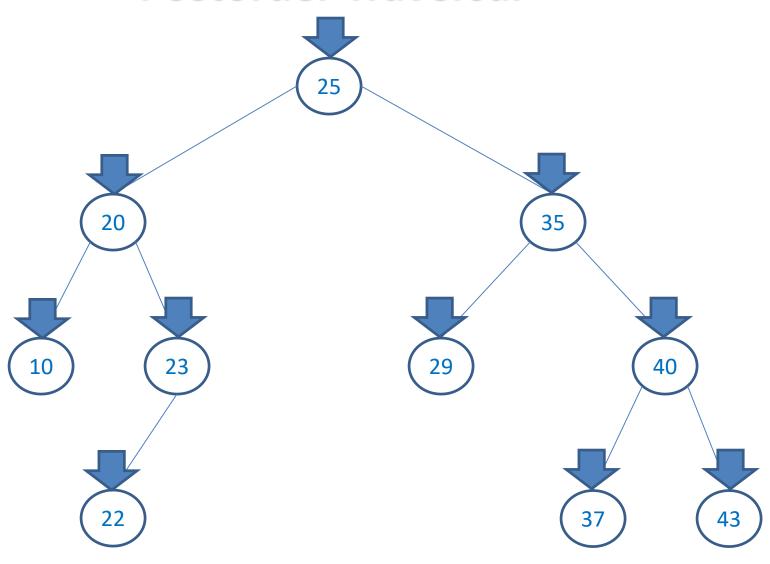
25, 20, 10, 23, 22, 35, 29, 40, 37, 43

Inorder Traversal



10, 20, 22, 23, 25, 29, 35, 37, 40, 43

Postorder Traversal

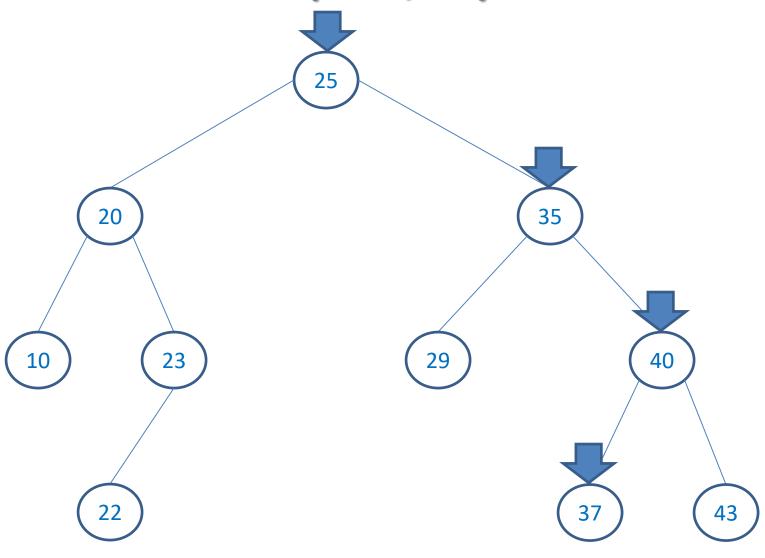


10, 22, 23, 20, 29, 37, 43, 40, 35, 25

Search in BST

- 1. If the BST is empty, then search is unsuccessful
- 2. If search_key is in the root node, then return root
- If search_key < Key in the root node, then search_key is searched in root->left
- 4. If search_key > Key in the root node, then search_key is searched in root->right

Search (root, 37)



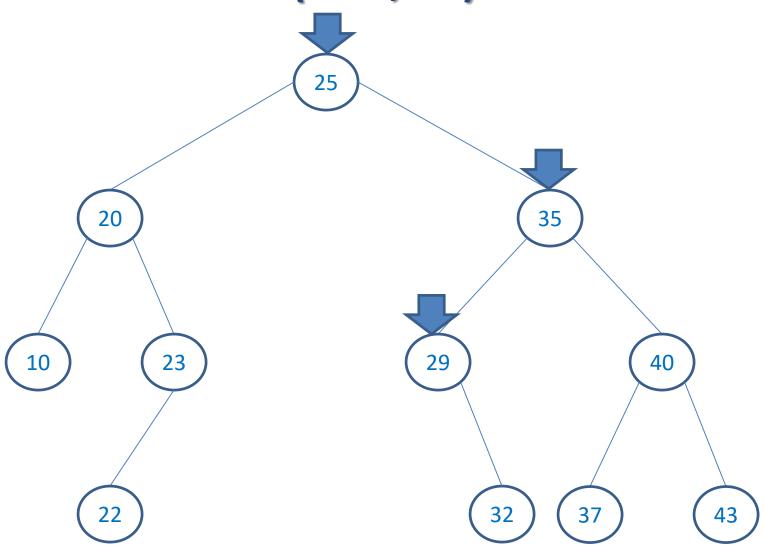
Search in BST

```
node *search(node *root, int data)
  if(root == NULL | | root->data == data)
             return root;
  else if(data < root->data)
             return search(root->left, data);
  else
             return search(root->right, data);
```

Insertion in BST

- 1. If the BST is empty, then create a node and it will be the root
- 2. If insert_key < Key in the root node, then insert_key will be inserted in root->left
- 3. If insert_key > Key in the root node, then insert_key will be inserted in root->right

Insert(root, 32)



Insertion in BST

```
node *createNode(int data)
  node *n = (node *)malloc(sizeof(node));
  n->data = data;
  n->left = NULL;
  n->right = NULL;
                             &left_subtree Data &right_subtree
  return n;
```

Insertion in BST

```
node *insert(node *root, int data)
  if(root == NULL) root = createNode(data);
  else if(data < root->data) root->left = insert(root-> left, data);
  else if(data > root->data) root->right = insert(root->right, data);
  return root;
From main():
for(i = 0; i < n; i++)
   root = insert(root, a[i]);
```

Deletion in BST

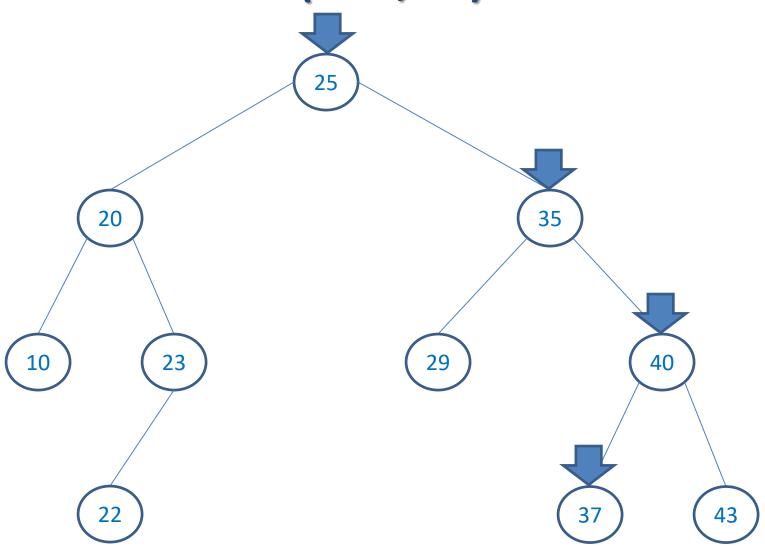
```
node *delete(node *root, int data)
  if(root == NULL) return NULL;
  else if(data < root->data) root->left = delete(root->left, data);
  else if(data > root->data) root->right = delete(root->right,
  data);
  else { // data == root->data
      Actual Code for Deletion
  return root;
```

Deletion in BST

Case 1: Node is a leaf node:

- 1. Temp = leaf node
- If temp is the left child of its parent, then set the left child of the parent NULL
- 3. If temp is the right child of its parent, then set the right child of the parent NULL
- 4. Delete temp

Delete(root, 37)

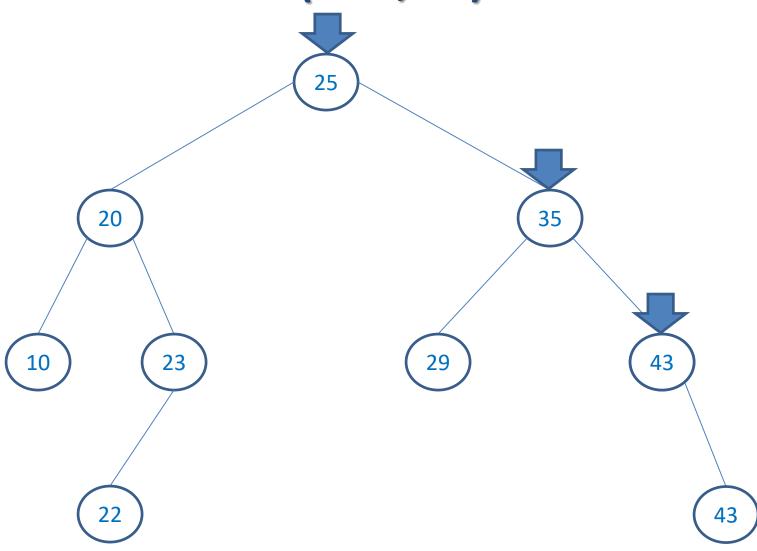


Deletion in BST

Case 2: Node has a left/right child C:

- 1. Temp = node to be deleted
- If temp is the left child of its parent, then set the left child of the parent to C
- 3. If temp is the right child of its parent, then set the right child of the parent to C
- 4. Delete temp

Delete(root, 40)

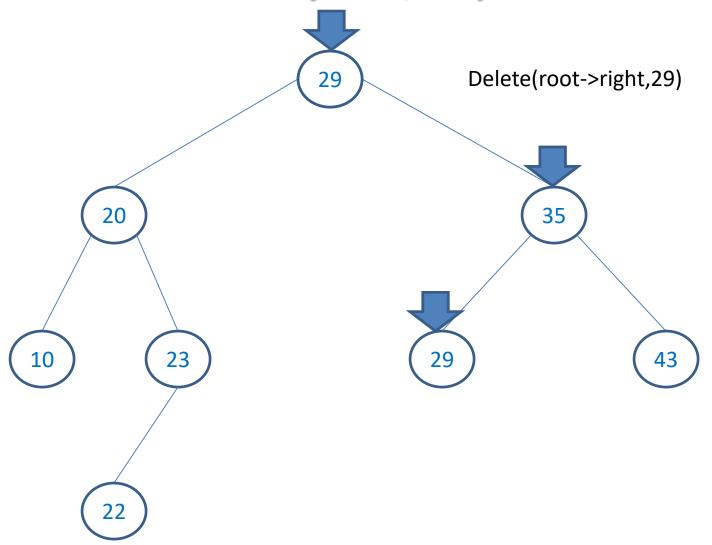


Deletion in BST

Case 3: Node have a left child L and a right child R:

- 1. Temp = node to be deleted
- 2. Find the minimum key in temp->right subtree
- 3. Copy that minimum key to temp->key
- 4. Delete the node with the minimum key in its right subtree

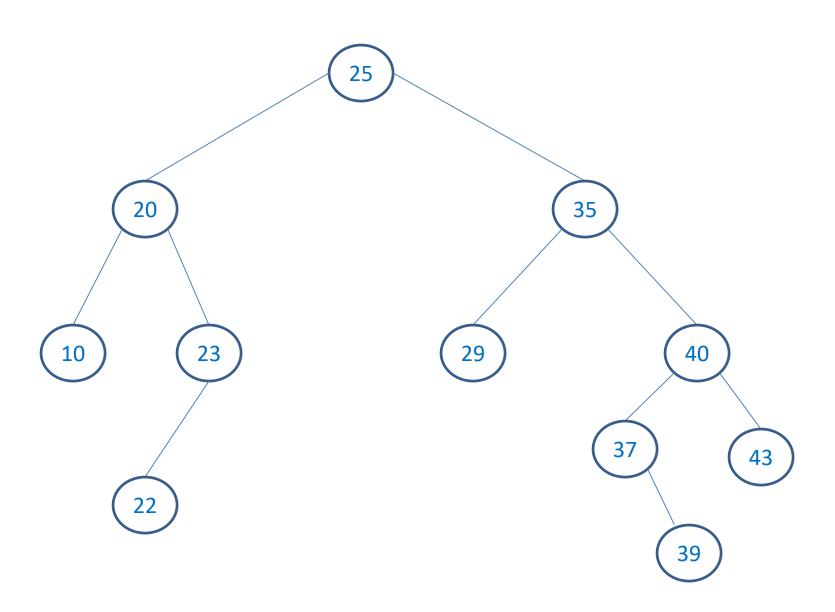
Delete(root, 25)



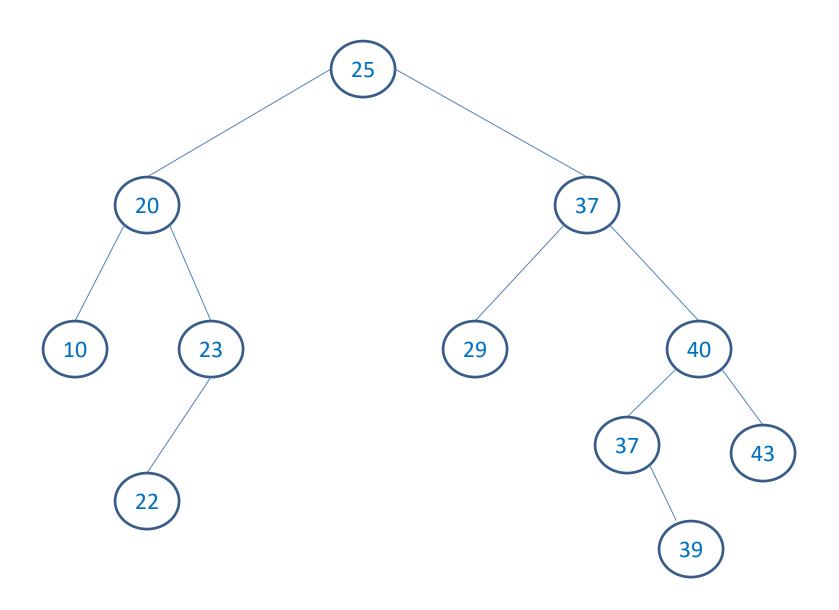
Deletion in BST

```
node *temp = NULL;
if(root->left != NULL && root->right != NULL) {
    temp = root->right;
    while(temp->left != NULL)
               temp = temp->left;
    root->data = temp->data;
    root->right = delete(root->right, temp->data);
    return root;
temp = (root->left != NULL) ? root->left : root->right;
free(root);
root = NULL;
return temp;
```

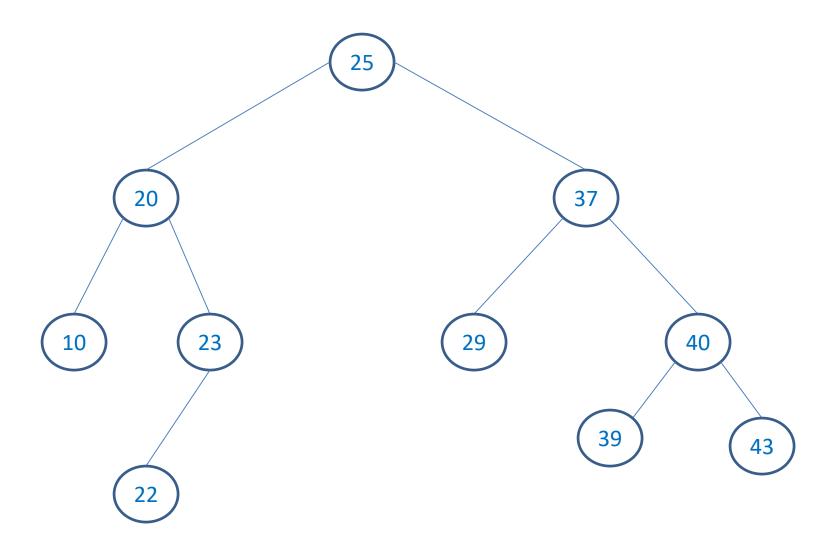
Delete(root, 35)



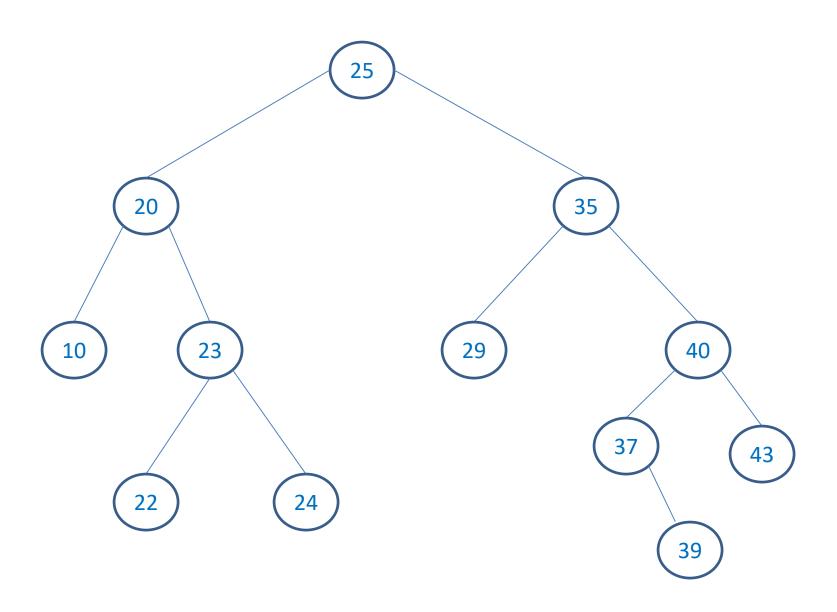
Delete(root->right, 37)



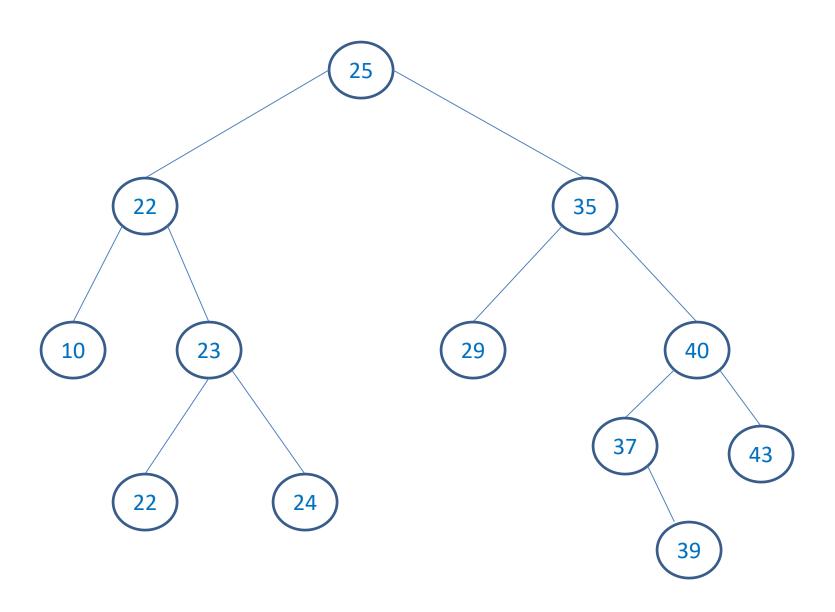
Delete(root->right, 37)



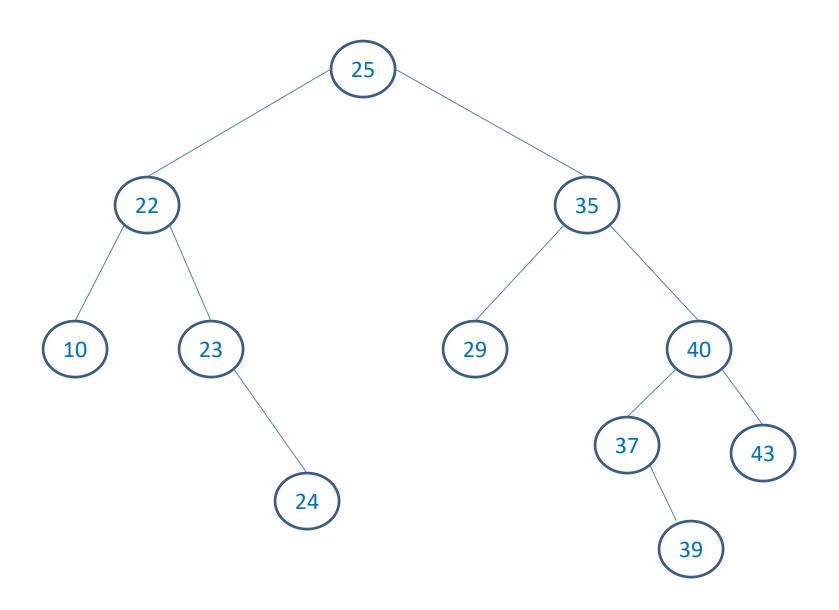
Delete(root, 20)



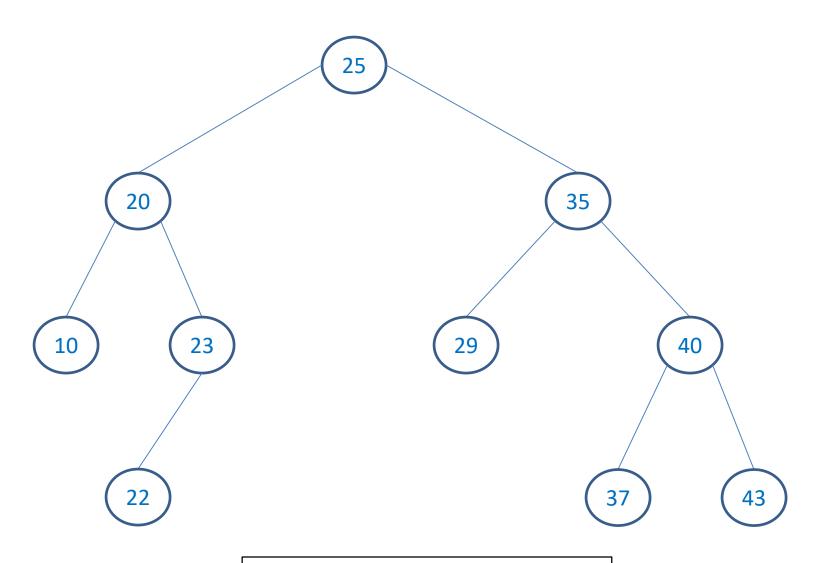
Delete(root->right, 22)



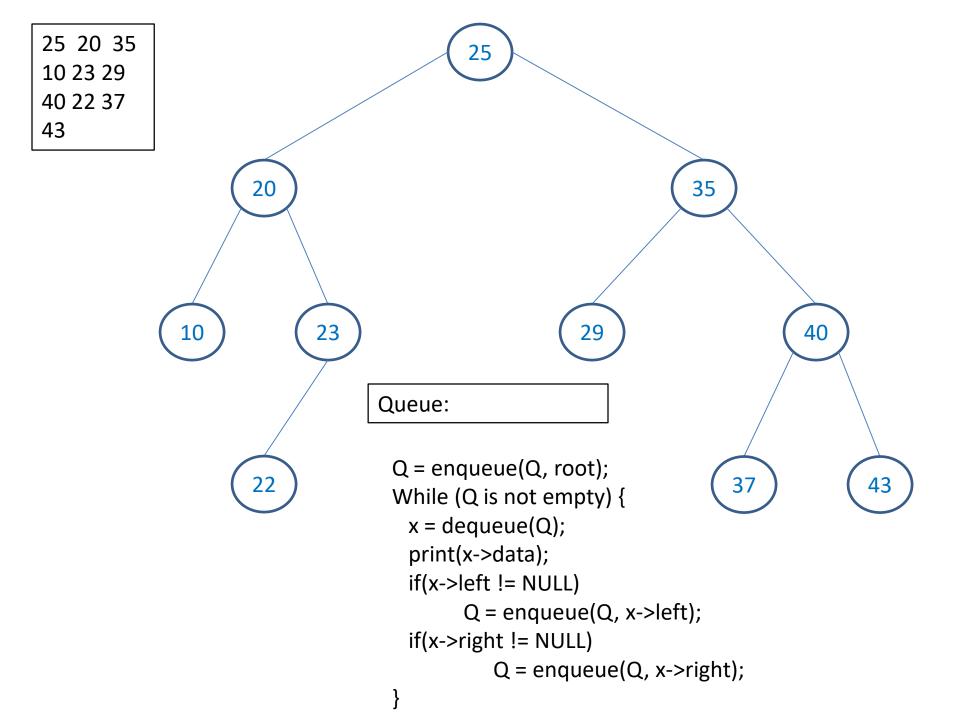
Delete(root->right, 22)



Application of Queue: Levelorder Traversal



25, 20, 35, 10, 23, 29, 40, 22, 37, 43



Find Lowest Common Ancestor

