



Faculty of Engineering and Tecnology

Computer Science Department

Trees_2

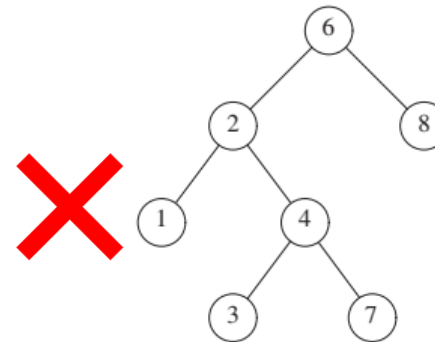
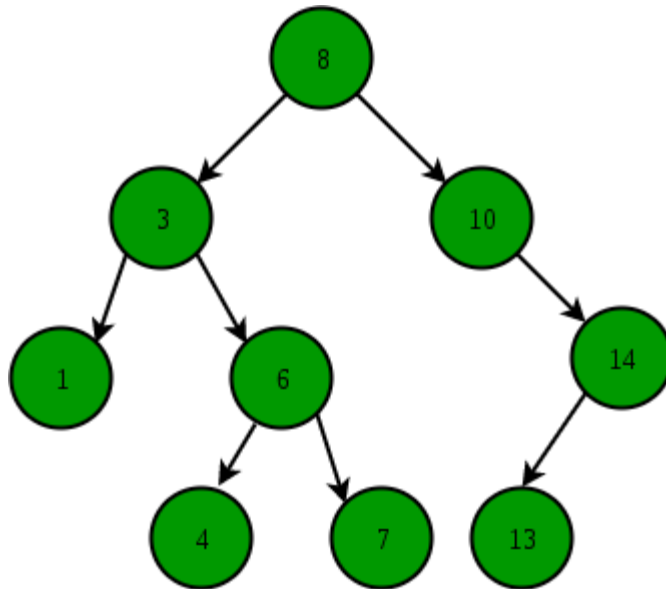
Binary Search Tree

Binary Search Trees (BST)

- BST: is a binary tree that satisfies the following properties:
 - The left subtree of any node contains only nodes with keys (values) less than the node's key.
 - The right subtree of any node contains only nodes with keys greater than the node's key.
 - The left and right subtree each must also be a binary search tree.

BST

the average depth of a binary search tree turns out to be $O(\log N)$



Operations on BST

- Creation
- Insertion
- Deletion
- Searching
- Traversing

BST: Implementation

```
typedef struct tree_node *tree_ptr;  
struct tree_node  
{  
    element_type element;  
    tree_ptr left;  
    tree_ptr right;  
};  
typedef tree_ptr BST;
```

- Routine to make an empty tree

```
BST Make_null ( void)  
{  
    return NULL;  
}
```

BST: Find

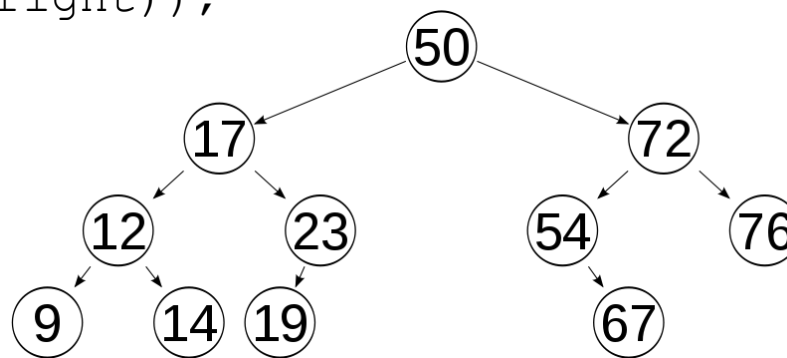
```

tree_ptr find( element_type x, BST T)
{
    if ( T == NULL)
return NULL;

if ( x < T->element) //greater than x move to right
    return ( find (x, T->left));
else //Less than x move to left
    if ( x > T->element )
return ( find ( x, T->right));
    else

return T;
}

```



BST: Traversal

```
//inorder
void traversal(BST T)
{
    if ( T == NULL)
        return ;
    traversal(T->left);
    printf("%d ",T->element );
    traversal(T->right);
}
```

```
//Preorder
void traversal(BST T)
{
    if ( T == NULL)
        return ;
    printf("%d ",T->element );
    traversal(T->left);
    traversal(T->right);
}
```

```
//Postorder
void traversal(BST T)
{
    if ( T == NULL)
        return ;
    traversal(T->left);
    traversal(T->right);
    printf("%d ",T->element );
}
```

None- traversal (post order) :

```

struct Node {
    int data;
    struct Node *left, *right;
    bool visited;
};

void postorder(struct Node* root)
{
    struct Node* temp = root; // Save head in temp tree

    while (temp && temp->visited == false) {

        // Visited left subtree
        if (temp->left && temp->left->visited == false)
            temp = temp->left;

        // Visited right subtree
        else if (temp->right && temp->right->visited == false)
            temp = temp->right;

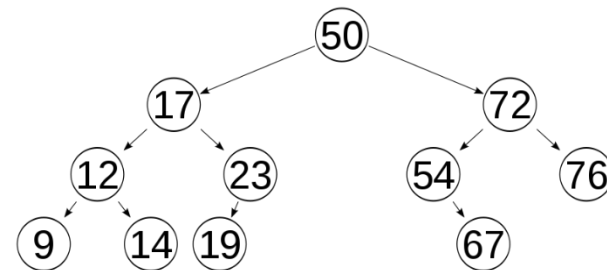
        // Print node
        else {
            printf("%d ", temp->data);
            temp->visited = true;
            temp = root;
        }
    }
}

```

```

struct Node* newNode(int data)
{
    struct Node* node = new
    Node;
    node->data = data;
    node->left = NULL;
    node->right = NULL;
    node->visited = false;
    return (node);
}

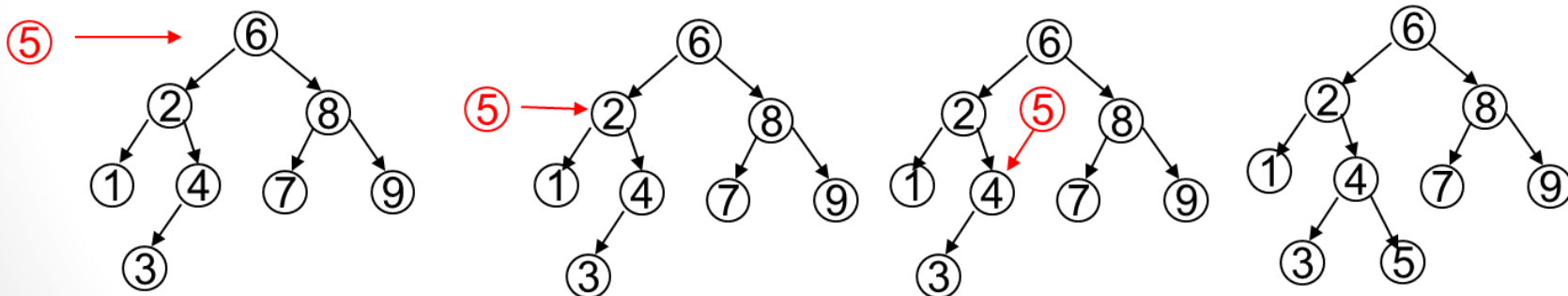
```



BST: Insertion

The idea is to do iterative level order traversal of the given tree.

- **If we find a node whose left child is empty**
we make new key **as left child** of the node.
- **Else if we find a node whose right child is empty**
we make new key **as right child**.
- We keep traversing the tree until we find a node whose either left or right is empty.



BST: Insertion

```

BST insert ( BST T, element_type x)
{
    //Tree empty, insert first element

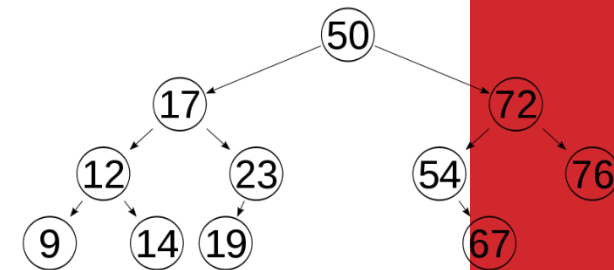
    if ( T == NULL ) {
        T = ( BST) malloc (sizeof(struct tree_node));

        if ( T == NULL )
            printf (" Out of space!!!");
        else
        {
            T->element = x;
            T->left = T->right = NULL;
        }
    }

    else //Tree not empty, check to insert to left or right.
        if ( x < T->element)
            T->left = insert ( T->left , x);
        else
            if ( x > T->element)
                T->right = insert(T->right, x);
    return T;
}

```

Example: Insert 8,28,52



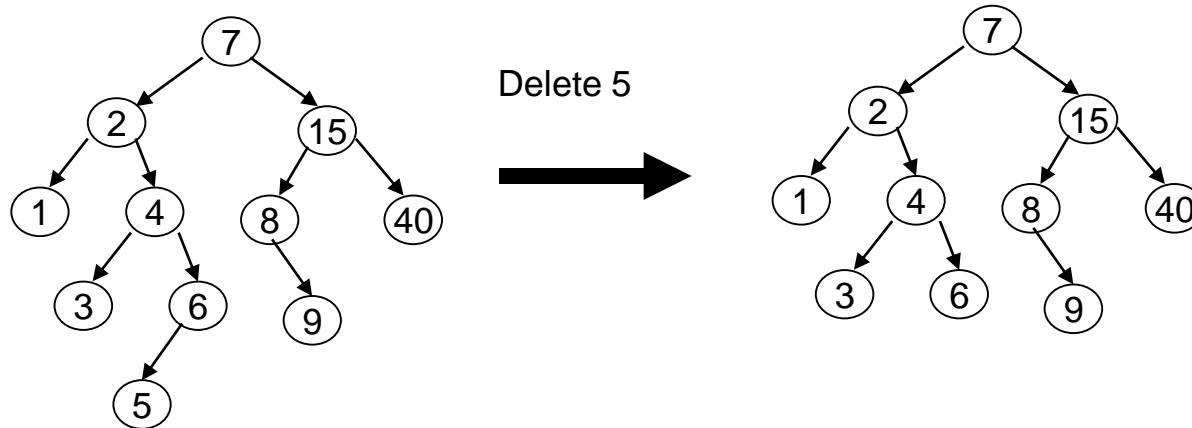
BST: Deletion

When we delete a node, three possibilities arise.

- 1) **Node to be deleted is leaf**: Simply remove from the tree.
- 2) **Node to be deleted has only one child**: Copy the child to the node and delete the child
- 3) **Node to be deleted has two children**: Find **in order successor of the node**. Copy contents of the **in order** successor to the node and delete the **in order successor**. Note that in order predecessor can also be used.

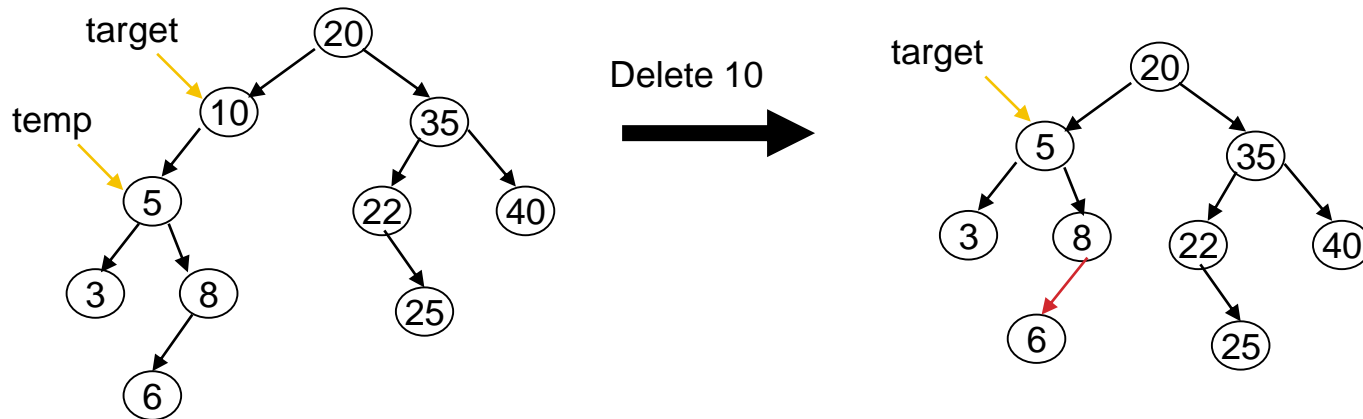
BST: Deletion a leaf

- Example: Delete 5 in the tree below:



BST: Deleting a one-child node

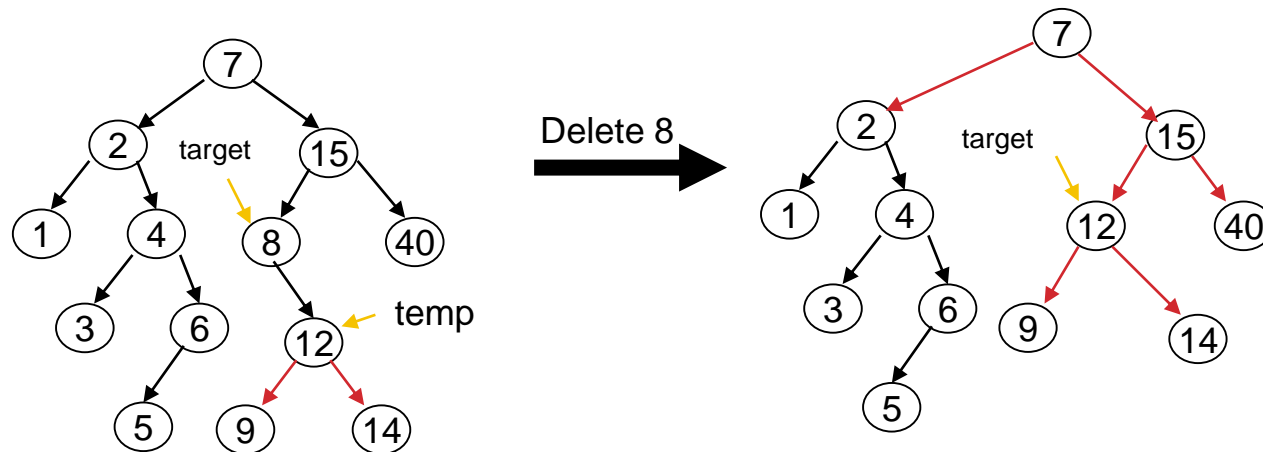
- CASE 2: THE NODE TO BE DELETED HAS ONE NON-EMPTY CHILD
 (a) The right subtree of the node x to be deleted is empty.
- Example:



BST: Deleting a one-child node

(b) The left subtree of the node x to be deleted is empty.

Example:

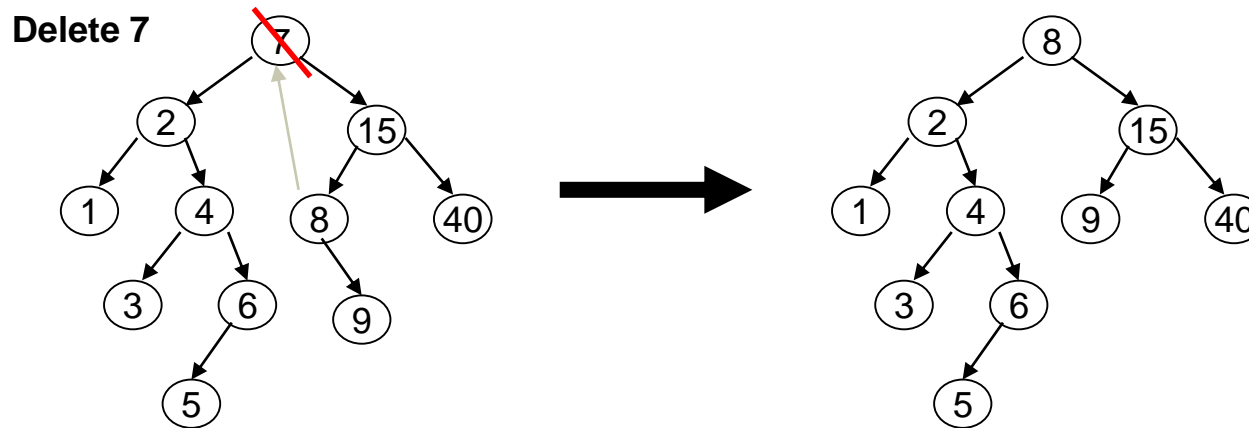


BST: Deleting a two-child node

METHOD#1: DELETION BY COPYING the minimum:

Copy the **minimum** key in the **right** subtree of x to the node x, then delete the one-child or leaf-node with this minimum key.

- Example:

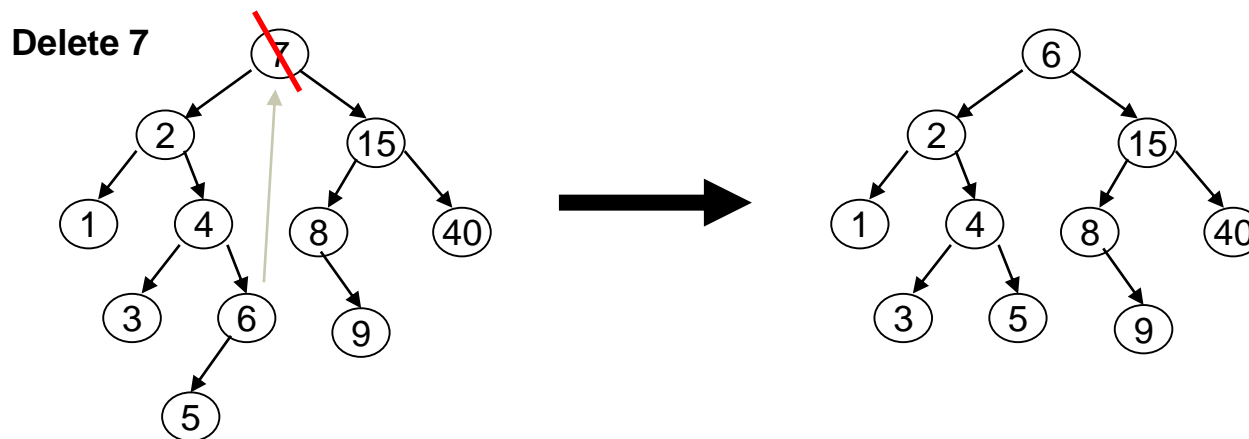


BST: Deleting a two-child node

METHOD#2: DELETION BY COPYING the maximum

Copy the **maximum** key in the **left** subtree of x to the node x, then delete the one-child or leaf-node with this **maximum** key.

- Example:



BST: Finding the minimum

Recursive implementation of find_min for binary search trees

```
tree_ptr find_min( BST T)
{
    if ( T == NULL )           //empty tree
        return NULL;
    else
        if ( T->left == NULL ) //node itself
            return ( T );
        else
            return ( find_min ( T->left )); //find min recursive
}
```

Nonrecursive implementation of find_max for binary search trees

```
tree_ptr find_max( BST T)
{
    if ( T != NULL )

        while ( T->right != NULL )
            T = T->right;
        return T;
}
```

Home Work: Rebuild two above functions in alternative way

BST: Delete function

```
tree_ptr delete ( BST T, int x)
```

```
{
```

```
tree_ptr tmp_cell, child;
```

```
if ( T == NULL )
```

```
    printf(“ Element not found” );
```

```
else if ( x < T->element)
```

```
    T->left = delete(T->left, x);
```

```
else if ( x > T->element)
```

```
    T->right = delete(T->right, x);
```

```
else if ( T->left && T->right ) //found element and has (right ,left) elements
```

```
{
```

```
    tmp_cell = find_min(T->right);
```

```
    T->element = tmp_cell->element;
```

```
    T->right = delete(T->right, T->element);
```

```
}
```

```
else
```

```
{
```

```
    tmp_cell = T;
```

```
    if ( T->left == NULL)
```

```
        child = T->right;
```

```
    if ( T->right == NULL)
```

```
        child = T->left;
```

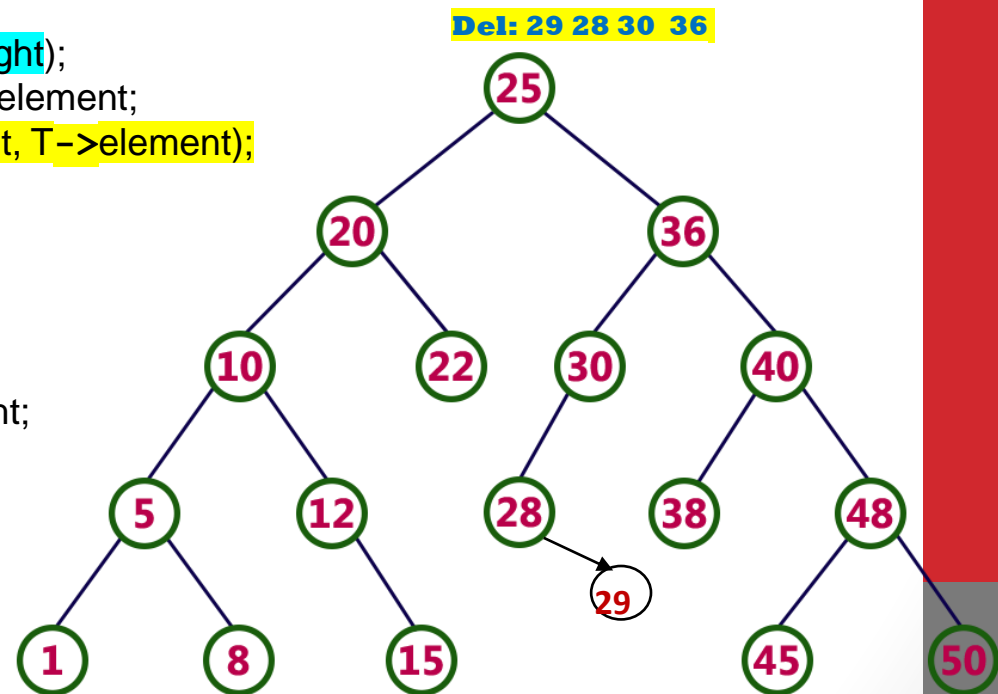
```
    free ( tmp_cell);
```

```
    return child;
```

```
}
```

```
return T;
```

```
}
```



Exercise :Constructing binary tree from in order and post order traversals

Post-order : 17 40 38 48 47 45 53 58 65 62 55 **50**

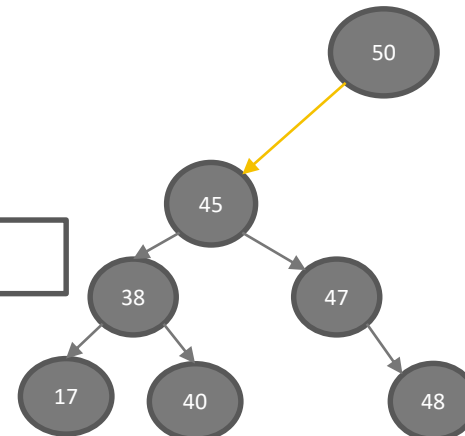
In-order : 17 38 40 45 47 48 **50** 53 55 58 62 65

Left Subtree

Post-order : 17 40 38 48 47 **45**

In-order : 17 38 40 **45** 47 48

Post-order : 17 40 **38**



Exercise: Constructing binary tree from in order and post order traversals

Post-order : 17 40 38 48 47 45 53 58 65 62 55 **50**

In-order : 17 38 40 45 47 48 **50** 53 55 58 62 65

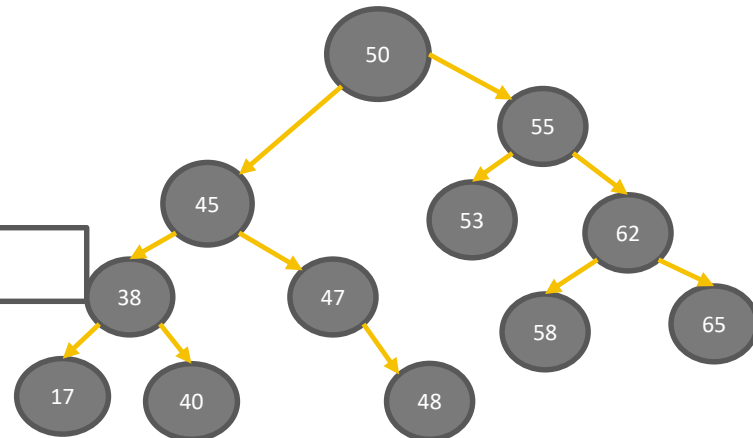
Right Subtree

Post-order 53 58 65 62 **55**

In-order : 53 **55** 58 62 55

Post-order 58 65 **62**

In order : 58 **62** 55



THANK YOU
