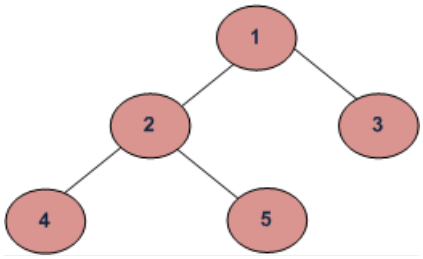
Write a program to Delete a Tree.

To delete a tree we must traverse all the nodes of the tree and delete them one by one. So which traversal we should use – Inorder or Preorder or Postorder. Answer is simple – Postorder, because before deleting the parent node we should delete its children nodes first

We can delete tree with other traversals also with extra space complexity but why should we go for other traversals if we have Postorder available which does the work without storing anything in same time complexity.

For the following tree nodes are deleted in order – 4, 5, 2, 3, 1



Example Tree

Program

C

```
#include<stdio.h>
#include<stdlib.h>

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
                        malloc(sizeof(struct node));

    node->data = data;
    node->left = NULL;
    node->right = NULL;
    return(node);
}

/*  This function traverses tree in post order to
    to delete each and every node of the tree */
void deleteTree(struct node* node)
{
    if (node == NULL) return;

    /* first delete both subtrees */
    deleteTree(node->left);
    deleteTree(node->right);

    /* then delete the node */
    printf("\n Deleting node: %d", node->data);
    free(node);
}


/* Driver program to test deleteTree function*/
int main()
{
    struct node *root = newNode(1);
    root->left            = newNode(2);
    root->right          = newNode(3);
    root->left->left     = newNode(4);
    root->left->right   = newNode(5);

    deleteTree(root);
    root = NULL;

    printf("\n Tree deleted ");

    getchar();
    return 0;
}
```

Java

```
// Java program to delete a tree

// A binary tree node
class Node
{
    int data;
    Node left, right;

    Node(int item)
    {
        data = item;
        left = right = null;
    }
}

class BinaryTree
{
    Node root;

    /*  This function traverses tree in post order to
        to delete each and every node of the tree */
    void deleteTree(Node node)
    {
        if (node == null)
            return;

        /* first delete both subtrees */
        deleteTree(node.left);
        deleteTree(node.right);

        /* then delete the node */
        System.out.println("The deleted node is " + node.data);
        node = null;
    }

    /* Driver program to test above functions */
    public static void main(String[] args)
    {
        BinaryTree tree = new BinaryTree();

        tree.root = new Node(1);
        tree.root.left = new Node(2);
        tree.root.right = new Node(3);
        tree.root.left.left = new Node(4);
        tree.root.left.right = new Node(5);

        /* Print all root-to-leaf paths of the input tree */
        tree.deleteTree(tree.root);
        tree.root = null;
        System.out.println("Tree deleted");

    }
}
```

Output:

```
 Deleting node: 4
 Deleting node: 5
 Deleting node: 2
 Deleting node: 3
 Deleting node: 1
 Tree deleted
```

The above deleteTree() function deletes the tree, but doesn't change root to NULL which may cause problems if the user of deleteTree() doesn't change root to NULL and tires to access values using root pointer. We can modify the deleteTree() function to take reference to the root node so that this problem doesn't occur. See the following code.

C

```
#include<stdio.h>
#include<stdlib.h>

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
                        malloc(sizeof(struct node));

    node->data = data;
    node->left = NULL;
    node->right = NULL;
    return(node);
}

/*  This function is same as deleteTree() in the previous program */
void _deleteTree(struct node* node)
{
    if (node == NULL) return;

    /* first delete both subtrees */
    _deleteTree(node->left);
    _deleteTree(node->right);

    /* then delete the node */
    printf("\n Deleting node: %d", node->data);
    free(node);
}

/* Deletes a tree and sets the root as NULL */
void deleteTree(struct node** node_ref)
{
  _deleteTree(*node_ref);
  *node_ref = NULL;
}

/* Driver program to test deleteTree function*/
int main()
{
    struct node *root = newNode(1);
    root->left            = newNode(2);
    root->right          = newNode(3);
    root->left->left     = newNode(4);
    root->left->right   = newNode(5);

    // Note that we pass the address of root here
    deleteTree(&root);
    printf("\n Tree deleted ");

    getchar();
    return 0;
}
```

Java

```
// Java program to delete a tree

/* A binary tree node has data, pointer to left child
   and pointer to right child */
class Node
{
    int data;
    Node left, right;

    Node(int d)
    {
        data = d;
        left = right = null;
    }
}

class BinaryTree
{

    static Node root;

    /*  This function is same as deleteTree() in the previous program */
    void deleteTree(Node node)
    {
        if (node == null)
        {
            return;
        }

        /* first delete both subtrees */
        deleteTree(node.left);
        deleteTree(node.right);

        /* then delete the node */
        System.out.println("The deleted node is " + node.data);
        node = null;
    }

    /* Wrapper function that deletes the tree and
       sets root node as null  */
    void deleteTreeRef(Node nodeRef)
    {
        deleteTree(nodeRef);
        nodeRef=null;
    }

    /* Driver program to test deleteTree function */
    public static void main(String[] args)
    {

        BinaryTree tree = new BinaryTree();

        tree.root = new Node(1);
        tree.root.left = new Node(2);
        tree.root.right = new Node(3);
        tree.root.left.left = new Node(4);
        tree.root.left.right = new Node(5);

        /* Note that we pass root node here */
        tree.deleteTreeRef(root);
        System.out.println("Tree deleted");

    }
}

// This code has been contributed by Mayank Jaiswal(mayank_24)
```

```
Deleting node: 4
Deleting node: 5
Deleting node: 2
Deleting node: 3
Deleting node: 1
Tree deleted
```

Time Complexity: O(n)

Space Complexity: If we don't consider size of stack for function calls then O(1) otherwise O(n)