Binary Tree Deletion

```
#include <iostream>
using namespace std;

class Node {
    public:
    int data;
    Node* left;
    Node* right;

    Node(int val) {
        data = val;
        left = right = NULL;
    }
};

// Inorder Traversal
void inorder(Node* root) {
    if (root == NULL) return;
    inorder(root->left);
    cout << root->data << " ";
    inorder(root->right);
}

// Find the deepest node in the tree
Node* findDeepest(Node* root) {
    if (root == NULL) return NULL;

    // If it's a leaf node, return it
    if (root->left == NULL && root->right == NULL) {
        return root;
    }

    // Prefer rightmost first, else left
    if (root->right) return findDeepest(root->right);
    return findDeepest(root->left);
}
```

```
// Delete the deepest node from the tree
void deleteDeepest(Node* root, Node* dNode) {
    if (root == NULL) return;

    if (root->left) {
        if (root->left == dNode) {
            delete root->left;
            root->left = NULL;
            return;
        } else {
            deleteDeepest(root->left, dNode);
        }
    }

    if (root->right) {
        if (root->right == dNode) {
            delete root->right;
            root->right = NULL;
            return;
        } else {
            deleteDeepest(root->right, dNode);
        }
    }
}

// Delete node by key
Node* deleteNode(Node* root, int key) {
    if (root == NULL) return NULL;

    // Case: only one node in tree
    if (root->left == NULL && root->right == NULL) {
        if (root->data == key) {
            delete root;
            return NULL;
        }
        return root;
    }

    Node* keyNode = NULL;
```

```
    // If root itself is the node to be deleted
    if (root->data == key) keyNode = root;

    // Recursively search for keyNode
    if (!keyNode && root->left) root->left = deleteNode(root->left, key);
    if (!keyNode && root->right) root->right = deleteNode(root->right, key);

    // If keyNode found
    if (keyNode != NULL) {
        Node* deepest = findDeepest(root); // find deepest node
        keyNode->data = deepest->data;     // copy deepest node's value
        deleteDeepest(root, deepest);      // delete deepest node
    }

    return root;
}

// Insert node
Node* insert(Node* root, int key) {
    if (root == NULL) {
        return new Node(key);
    }
    if (root->left == NULL) {
        root->left = new Node(key);
    }
    else if (root->right == NULL) {
        root->right = new Node(key);
    }
    else {
        root->left = insert(root->left, key);
    }
    return root;
}

int main() {
    Node* root = NULL;

    // Build tree
    root = insert(root, 1);
    root = insert(root, 2);
```

```
    root = insert(root, 3);
    root = insert(root, 4);
    root = insert(root, 5);

    cout << "Inorder before deletion: ";
    inorder(root);

    root = deleteNode(root, 2);

    cout << "\nInorder after deleting 2: ";
    inorder(root);

    return 0;
}
```

Output-

Inorder before deletion: 4 2 5 1 3
Inorder after deleting 2: 4 5 1 3