Deletion in Binary Search tree

In a binary search tree, we must delete a node from the tree by keeping in mind that the property of BST is not violated. To delete a node from BST, there are three possible situations occur -

- o The node to be deleted is the leaf node, or,
- o The node to be deleted has only one child, and,
- o The node to be deleted has two children

We will understand the situations listed above in detail.

When the node to be deleted is the leaf node

It is the simplest case to delete a node in BST. Here, we have to replace the leaf node with NULL and simply free the allocated space.

We can see the process to delete a leaf node from BST in the below image. In below image, suppose we have to delete node 90, as the node to be deleted is a leaf node, so it will be replaced with NULL, and the allocated space will free.
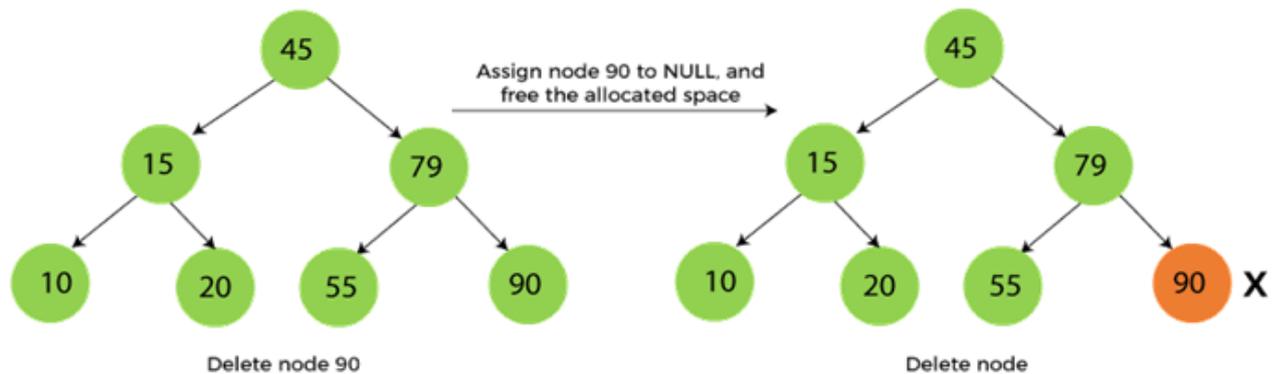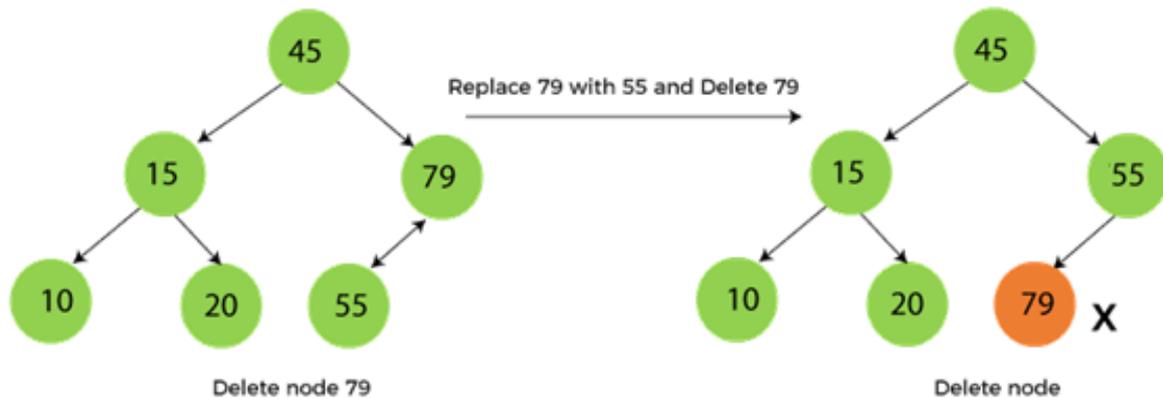

When the node to be deleted has only one child

In this case, we have to replace the target node with its child, and then delete the child node. It means that after replacing the target node with its child node, the child node will now contain the value to be deleted. So, we simply have to replace the child node with NULL and free up the allocated space.

We can see the process of deleting a node with one child from BST in the below image. In the below image, suppose we have to delete the node 79, as the node to be deleted has only one child, so it will be replaced with its child 55.

So, the replaced node 79 will now be a leaf node that can be easily deleted.

Delete node 79 — Replace 79 with 55 and Delete 79 — Delete node

When the node to be deleted has two children

This case of deleting a node in BST is a bit complex among other two cases. In such a case, the steps to be followed are listed as follows -

- o First, find the inorder successor of the node to be deleted.
- o After that, replace that node with the inorder successor until the target node is placed at the leaf of tree.
- o And at last, replace the node with NULL and free up the allocated space.

The inorder successor is required when the right child of the node is not empty. We can obtain the inorder successor by finding the minimum element in the right child of the node.

We can see the process of deleting a node with two children from BST in the below image. In the below image, suppose we have to delete node 45 that is the root node, as the node to be deleted has two children, so it will be replaced with its inorder successor. Now, node 45 will be at the leaf of the tree so that it can be deleted easily.

Delete node 45 — Replace 45 with its inorder successor (i.e., 55) — Delete node
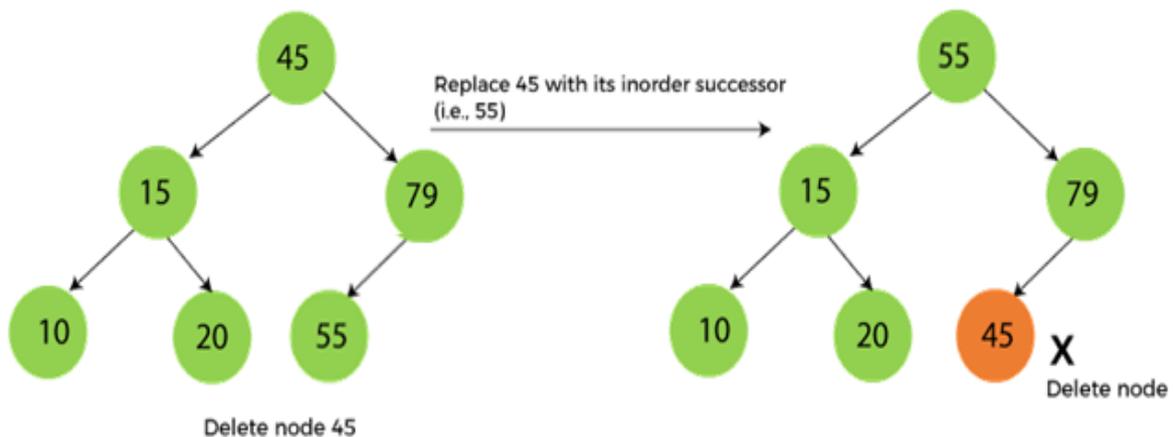
Implementation of Deletion operation in a BST:

```c
// C program to implement optimized delete in BST.
#include <stdio.h>
#include <stdlib.h>

struct Node {
    int key;
    struct Node *left, *right;
};

// A utility function to create a new BST node
struct Node* newNode(int item)
{
    struct Node* temp = (struct Node*)malloc(sizeof(struct Node));
    temp->key = item;
    temp->left = temp->right = NULL;
    return temp;
}

// A utility function to do inorder traversal of BST
void inorder(struct Node* root)
{
    if (root != NULL) {
        inorder(root->left);
        printf("%d ", root->key);
        inorder(root->right);
    }
}

/* A utility function to insert a new node with given key in
 * BST */
struct Node* insert(struct Node* node, int key)
{
    /* If the tree is empty, return a new node */
    if (node == NULL)
        return newNode(key);

    /* Otherwise, recur down the tree */
    if (key < node->key)
        node->left = insert(node->left, key);
    else
        node->right = insert(node->right, key);

    /* return the (unchanged) node pointer */
    return node;
```

```c
}

/* Given a binary search tree and a key, this function
   deletes the key and returns the new root */
struct Node* deleteNode(struct Node* root, int k)
{
    // Base case
    if(root == NULL)
        return root;

    // Recursive calls for ancestors of
    // node to be deleted
    if(root->key > k) {
        root->left = deleteNode(root->left, k);
        return root;
    }
    else if(root->key < k) {
        root->right = deleteNode(root->right, k);
        return root;
    }

    // We reach here when root is the node
    // to be deleted.

    // If one of the children is empty
    if(root->left == NULL) {
        struct Node* temp = root->right;
        free(root);
        return temp;
    }
    else if(root->right == NULL) {
        struct Node* temp = root->left;
        free(root);
        return temp;
    }

    // If both children exist
    else {

        struct Node* succParent = root;

        // Find successor
        struct Node* succ = root->right;
        while(succ->left != NULL) {
            succParent = succ;
            succ = succ->left;
```

```c
        }

        // Delete successor.  Since successor
        // is always left child of its parent
        // we can safely make successor's right
        // right child as left of its parent.
        // If there is no succ, then assign
        // succ->right to succParent->right
        if(succParent != root)
            succParent->left = succ->right;
        else
            succParent->right = succ->right;

        // Copy Successor Data to root
        root->key = succ->key;

        // Delete Successor and return root
        free(succ);
        return root;
    }
}

// Driver Code
int main()
{
    /* Let us create following BST
          50
        /    \
      30     70
     / \    / \
   20  40  60  80 */
    struct Node* root = NULL;
    root = insert(root, 50);
    root = insert(root, 30);
    root = insert(root, 20);
    root = insert(root, 40);
    root = insert(root, 70);
    root = insert(root, 60);

    printf("Original BST: ");
    inorder(root);

    printf("\n\nDelete a Leaf Node: 20\n");
    root = deleteNode(root, 20);
    printf("Modified BST tree after deleting Leaf Node:\n");
    inorder(root);
```

```
    printf("\n\nDelete Node with single child: 70\n");
    root = deleteNode(root, 70);
    printf("Modified BST tree after deleting single child Node:\n");
    inorder(root);

    printf("\n\nDelete Node with both child: 50\n");
    root = deleteNode(root, 50);
    printf("Modified BST tree after deleting both child Node:\n");
    inorder(root);

    return 0;
}
```

Output
Original BST: 20 30 40 50 60 70

Delete a Leaf Node: 20

Modified BST tree after deleting Leaf Node:

30 40 50 60 70

Delete Node with single child: 70

Modified BST tree after deleting single child No...

Time Complexity: O(h), where h is the height of the BST. **Auxiliary Space:** O(n).