How to implement decrease key or change key in Binary Search Tree?

Given a Binary Search Tree, write a function that takes following three as arguments:

1) Root of tree

2) Old key value

3) New Key Value

The function should change old key value to new key value. The function may assume that old key value always exists in Binary Search Tree.

Example:

```
Input: Root of below tree
              50
           /     \
          30      70
         /  \    /  \
       20   40  60   80
     Old key value:  40
     New key value:  10

Output: BST should be modified to following
              50
           /     \
          30      70
         /       /  \
       20      60   80
       /
     10
```

We strongly recommend you to minimize your browser and try this yourself first

The idea is to call delete for old key value, then call insert for new key value. Below is C++ implementation of the idea.

```
// C program to demonstrate decrease  key operation on binary search tree
#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)
{
```

```c
    /* 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;
}

/* Given a non-empty binary search tree, return the node with minimum
   key value found in that tree. Note that the entire tree does not
   need to be searched. */
struct node * minValueNode(struct node* node)
{
    struct node* current = node;

    /* loop down to find the leftmost leaf */
    while (current->left != NULL)
        current = current->left;

    return current;
}

/* 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 key)
{
    // base case
    if (root == NULL) return root;

    // If the key to be deleted is smaller than the root's key,
    // then it lies in left subtree
    if (key < root->key)
        root->left = deleteNode(root->left, key);

    // If the key to be deleted is greater than the root's key,
    // then it lies in right subtree
    else if (key > root->key)
        root->right = deleteNode(root->right, key);

    // if key is same as root's key, then This is the node
    // to be deleted
    else
    {
        // node with only one child or no child
        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;
        }

        // node with two children: Get the inorder successor (smallest
        // in the right subtree)
        struct node* temp = minValueNode(root->right);

        // Copy the inorder successor's content to this node
        root->key = temp->key;

        // Delete the inorder successor
        root->right = deleteNode(root->right, temp->key);
    }
    return root;
    return root;
}

// Function to decrease a key value in Binary Search Tree
struct node *changeKey(struct node *root, int oldVal, int newVal)
{
    //  First delete old key value
    root = deleteNode(root, oldVal);

    // Then insert new key value
    root = insert(root, newVal);

    // Return new root
    return root;
}

// Driver Program to test above functions
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);
    root = insert(root, 80);


    printf("Inorder traversal of the given tree \n");
    inorder(root);

    root = changeKey(root, 40, 10);

    /* BST is modified to
              50
           /     \
          30      70
         /       /  \
       20      60   80
       /
     10     */
    printf("\nInorder traversal of the modified tree \n");
    inorder(root);

    return 0;
}
```

Output:

```
Inorder traversal of the given tree
20 30 40 50 60 70 80
Inorder traversal of the modified tree
10 20 30 50 60 70 80
```

Time complexity of above changeKey() is O(h) where h is height of BST.