Remove all nodes which don't lie in any path with sum>= k

Given a binary tree, a complete path is defined as a path from root to a leaf. The sum of all nodes on that path is defined as the sum of that path. Given a number K, you have to remove (prune the tree) all nodes which don't lie in any path with sum>=k.

Note: A node can be part of multiple paths. So we have to delete it only in case when all paths from it have sum less than K.

```
Consider the following Binary Tree
          1
       /      \
      2        3
    /   \     /  \
   4     5   6    7
  / \    /       /
 8   9  12      10
    / \           \
   13  14         11
       /
      15

For input k = 20, the tree should be changed to following
(Nodes with values 6 and 8 are deleted)
          1
       /      \
      2        3
    /   \        \
   4     5        7
    \    /       /
     9  12      10
    / \           \
   13  14         11
       /
      15

For input k = 45, the tree should be changed to following.
      1
    /
   2
  /
 4
  \
   9
    \
     14
      /
     15
```

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

The idea is to traverse the tree and delete nodes in bottom up manner. While traversing the tree, recursively calculate the sum of nodes from root to leaf node of each path. For each visited node, checks the total calculated sum against given sum "k". If sum is less than k, then free(delete) that node (leaf node) and return the sum back to the previous node. Since the path is from root to leaf and nodes are deleted in bottom up manner, a node is deleted only when all of its descendants are deleted. Therefore, when a node is deleted, it must be a leaf in the current Binary Tree.

Following is C implementation of the above approach.

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

// A utility function to get maximum of two integers
int max(int l, int r) { return (l > r ? l : r); }

// A Binary Tree Node
struct Node
{
    int data;
    struct Node *left, *right;
};

// A utility function to create a new Binary Tree node with given data
struct Node* newNode(int data)
{
    struct Node* node = (struct Node*) malloc(sizeof(struct Node));
    node->data = data;
    node->left = node->right = NULL;
    return node;
}

// print the tree in LVR (Inorder traversal) way.
void print(struct Node *root)
{
    if (root != NULL)
    {
        print(root->left);
        printf("%d ",root->data);
        print(root->right);
    }
}

/* Main function which truncates the binary tree. */
struct Node *pruneUtil(struct Node *root, int k, int *sum)
{
    // Base Case
    if (root == NULL)  return NULL;

    // Initialize left and right sums as sum from root to
    // this node (including this node)
    int lsum = *sum + (root->data);
    int rsum = lsum;

    // Recursively prune left and right subtrees
    root->left = pruneUtil(root->left, k, &lsum);
    root->right = pruneUtil(root->right, k, &rsum);

    // Get the maximum of left and right sums
    *sum = max(lsum, rsum);

    // If maximum is smaller than k, then this node
    // must be deleted
    if (*sum < k)
    {
        free(root);
        root = NULL;
    }

    return root;
}

// A wrapper over pruneUtil()
struct Node *prune(struct Node *root, int k)
{
    int sum = 0;
    return pruneUtil(root, k, &sum);
}

// Driver program to test above function
int main()
{
    int k = 45;
    struct Node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->left = newNode(6);
    root->right->right = newNode(7);
    root->left->left->left = newNode(8);
    root->left->left->right = newNode(9);
    root->left->right->left = newNode(12);
    root->right->right->left = newNode(10);
    root->right->right->left->right = newNode(11);
    root->left->left->right->left = newNode(13);
    root->left->left->right->right = newNode(14);
    root->left->left->right->right->left = newNode(15);

    printf("Tree before truncation\n");
    print(root);

    root = prune(root, k); // k is 45

    printf("\n\nTree after truncation\n");
    print(root);

    return 0;
}
```

Run on IDE

Output:

```
Tree before truncation
8 4 13 9 15 14 2 12 5 1 6 3 10 11 7

Tree after truncation
4 9 15 14 2 1
```

Time Complexity: O(n), the solution does a single traversal of given Binary Tree.

A Simpler Solution:

The above code can be simplified using the fact that nodes are deleted in bottom up manner. The idea is to keep reducing the sum when traversing down. When we reach a leaf and sum is greater than the leaf's data, then we delete the leaf. Note that deleting nodes may convert a non-leaf node to a leaf node and if the data for the converted leaf node is less than the current sum, then the converted leaf should also be deleted.

Thanks to vicky for suggesting this solution in below comments.

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

// A Binary Tree Node
struct Node
{
    int data;
    struct Node *left, *right;
};

// A utility function to create a new Binary
// Tree node with given data
struct Node* newNode(int data)
{
    struct Node* node =
       (struct Node*) malloc(sizeof(struct Node));
    node->data = data;
    node->left = node->right = NULL;
    return node;
}

// print the tree in LVR (Inorder traversal) way.
void print(struct Node *root)
{
    if (root != NULL)
    {
        print(root->left);
        printf("%d ",root->data);
        print(root->right);
    }
}

/* Main function which truncates the binary tree. */
struct Node *prune(struct Node *root, int sum)
{
    // Base Case
    if (root == NULL) return NULL;

    // Recur for left and right subtrees
    root->left = prune(root->left, sum - root->data);
    root->right = prune(root->right, sum - root->data);

    // If we reach leaf whose data is smaller than sum,
    // we delete the leaf.  An important thing to note
    // is a non-leaf node can become leaf when its
    // chilren are deleted.
    if (root->left==NULL && root->right==NULL)
    {
        if (root->data < sum)
        {
            free(root);
            return NULL;
        }
    }

    return root;
}

// Driver program to test above function
int main()
{
    int k = 45;
    struct Node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->left = newNode(6);
    root->right->right = newNode(7);
    root->left->left->left = newNode(8);
    root->left->left->right = newNode(9);
    root->left->right->left = newNode(12);
    root->right->right->left = newNode(10);
    root->right->right->left->right = newNode(11);
    root->left->left->right->left = newNode(13);
    root->left->left->right->right = newNode(14);
    root->left->left->right->right->left = newNode(15);

    printf("Tree before truncation\n");
    print(root);

    root = prune(root, k); // k is 45

    printf("\n\nTree after truncation\n");
    print(root);

    return 0;
}
```

Run on IDE

Output:

```
Tree before truncation
8 4 13 9 15 14 2 12 5 1 6 3 10 11 7

Tree after truncation
4 9 15 14 2 1
```