Print ancestors of a given binary tree node without recursion

Given a Binary Tree and a key, write a function that prints all the ancestors of the key in the given binary tree.

For example, consider the following Binary Tree

```
           1
       /       \
      2         3
    /   \     /   \
   4     5   6    7
  /       \       /
 8         9     10
```

Following are different input keys and their ancestors in the above tree

```
Input Key    List of Ancestors
------------------------
 1
 2            1
 3            1
 4            2 1
 5            2 1
 6            3 1
 7            3 1
 8            4 2 1
 9            5 2 1
10            7 3 1
```

Recursive solution for this problem is discussed here.

It is clear that we need to use a stack based iterative traversal of the Binary Tree. The idea is to have all ancestors in stack when we reach the node with given key. Once we reach the key, all we have to do is, print contents of stack.

How to get all ancestors in stack when we reach the given node? We can traverse all nodes in Postorder way. If we take a closer look at the recursive postorder traversal, we can easily observe that, when recursive function is called for a node, the recursion call stack contains ancestors of the node. So idea is do iterative Postorder traversal and stop the traversal when we reach the desired node.

Following is C implementation of the above approach.

```
// C program to print all ancestors of a given key
#include <stdio.h>
#include <stdlib.h>

// Maximum stack size
#define MAX_SIZE 100

// Structure for a tree node
struct Node
{
    int data;
    struct Node *left, *right;
};

// Structure for Stack
struct Stack
{
    int size;
    int top;
    struct Node* *array;
};

// A utility function to create a new tree node
struct Node* newNode(int data)
{
```

```
    struct Node* node = (struct Node*) malloc(sizeof(struct Node));
    node->data = data;
    node->left = node->right = NULL;
    return node;
}

// A utility function to create a stack of given size
struct Stack* createStack(int size)
{
    struct Stack* stack = (struct Stack*) malloc(sizeof(struct Stack));
    stack->size = size;
    stack->top = -1;
    stack->array = (struct Node**) malloc(stack->size * sizeof(struct Node*));
    return stack;
}

// BASIC OPERATIONS OF STACK
int isFull(struct Stack* stack)
{
    return ((stack->top + 1) == stack->size);
}
int isEmpty(struct Stack* stack)
{
    return stack->top == -1;
}
void push(struct Stack* stack, struct Node* node)
{
    if (isFull(stack))
        return;
    stack->array[++stack->top] = node;
}
struct Node* pop(struct Stack* stack)
{
    if (isEmpty(stack))
        return NULL;
    return stack->array[stack->top--];
}
struct Node* peek(struct Stack* stack)
{
    if (isEmpty(stack))
        return NULL;
    return stack->array[stack->top];
}

// Iterative Function to print all ancestors of a given key
void printAncestors(struct Node *root, int key)
{
    if (root == NULL) return;

    // Create a stack to hold ancestors
    struct Stack* stack = createStack(MAX_SIZE);

    // Traverse the complete tree in postorder way till we find the key
    while (1)
    {
        // Traverse the left side. While traversing, push the nodes into
        // the stack so that their right subtrees can be traversed later
        while (root && root->data != key)
        {
            push(stack, root);   // push current node
            root = root->left;  // move to next node
        }

        // If the node whose ancestors are to be printed is found,
        // then break the while loop.
        if (root && root->data == key)
            break;

        // Check if right sub-tree exists for the node at top
        // If not then pop that node because we don't need this
        // node any more.
        if (peek(stack)->right == NULL)
        {
```

```
            root = pop(stack);

            // If the popped node is right child of top, then remove the top
            // as well. Left child of the top must have processed before.
            // Consider the following tree for example and key = 3.  If we
            // remove the following loop, the program will go in an
            // infinite loop after reaching 5.
            //          1
            //        /   \
            //       2     3
            //         \
            //           4
            //             \
            //              5
            while (!isEmpty(stack) && peek(stack)->right == root)
               root = pop(stack);
        }

        // if stack is not empty then simply set the root as right child
        // of top and start traversing right sub-tree.
        root = isEmpty(stack)? NULL: peek(stack)->right;
    }

    // If stack is not empty, print contents of stack
    // Here assumption is that the key is there in tree
    while (!isEmpty(stack))
        printf("%d ", pop(stack)->data);
}

// Driver program to test above functions
int main()
{
    // Let us construct a binary tree
    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->right->right = newNode(9);
    root->right->right->left = newNode(10);

    printf("Following are all keys and their ancestors\n");
    for (int key = 1; key <= 10; key++)
    {
        printf("%d: ", key);
        printAncestors(root, key);
        printf("\n");
    }

    getchar();
    return 0;
}
```

Output:

```
Following are all keys and their ancestors
1:
2: 1
3: 1
4: 2 1
5: 2 1
6: 3 1
7: 3 1
8: 4 2 1
9: 5 2 1
10: 7 3 1
```

Exercise

Note that the above solution assumes that the given key is present in the given Binary Tree. It may go in infinite loop if key is not present. Extend the above solution to work even when the key is not present in tree.