Inorder predecessor and successor for a given key in BST

You need to find the inorder successor and predecessor of a given key. In case the given key is not found in BST, then return the two values within which this key will lie.

Following is the algorithm to reach the desired result. It is a recursive method:

```
Input: root node, key
output: predecessor node, successor node


1. If root is NULL
      then return
2. if key is found then
    a. If its left subtree is not null
        Then predecessor will be the right most
        child of left subtree or left child itself.
    b. If its right subtree is not null
        The successor will be the left most child
        of right subtree or right child itself.
    return
3. If key is smaller than root node
        set the successor as root
        search recursively into left subtree
    else
        set the predecessor as root
        search recursively into right subtree
```

Following is the implementation of the above algorithm:

```
// C++ program to find predecessor and successor in a BST

#include <iostream>

using namespace std;
```

```
// BST Node
struct Node
{
        int key;
        struct Node *left, *right;
};

// This function finds predecessor and successor of key in BST.
// It sets pre and suc as predecessor and successor respectively
void findPreSuc(Node* root, Node*& pre, Node*& suc, int key)
{
        // Base case
        if (root == NULL) return ;

        // If key is present at root
        if (root->key == key)
        {
                // the maximum value in left subtree is predecessor
                if (root->left != NULL)
                {
                        Node* tmp = root->left;
                        while (tmp->right)
                                tmp = tmp->right;
                        pre = tmp ;
                }

                // the minimum value in right subtree is successor
                if (root->right != NULL)
        {
            Node* tmp = root->right ;
            while (tmp->left)
                tmp = tmp->left ;
            suc = tmp ;
        }
        return ;
    }

    // If key is smaller than root's key, go to left subtree
    if (root->key > key)
    {
        suc = root ;
        findPreSuc(root->left, pre, suc, key) ;
    }
    else // go to right subtree
    {
        pre = root ;
        findPreSuc(root->right, pre, suc, key) ;
    }
}

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

```
}

/* A utility function to insert a new node with given key in BST */
Node* insert(Node* node, int key)
{
        if (node == NULL) return newNode(key);
        if (key < node->key)
                node->left = insert(node->left, key);
        else
                node->right = insert(node->right, key);
        return node;
}

// Driver program to test above function
int main()
{
        int key = 65; //Key to be searched in BST

/* Let us create following BST
                        50
                /       \
                30      70
                / \ / \
        20 40 60 80 */
        Node *root = NULL;
        root = insert(root, 50);
        insert(root, 30);
        insert(root, 20);
        insert(root, 40);
```

```
    insert(root, 70);
    insert(root, 60);
    insert(root, 80);


    Node* pre = NULL, *suc = NULL;

    findPreSuc(root, pre, suc, key);
    if (pre != NULL)
    cout << "Predecessor is " << pre->key << endl;
    else
    cout << "No Predecessor";

    if (suc != NULL)
    cout << "Successor is " << suc->key;
    else
    cout << "No Successor";
    return 0;
}
```

Output

```
Predecessor is 60

Successor is 70
```

Complexity Analysis:
Time Complexity: O(h), where h is the height of the tree. In the worst case as explained above we travel the whole height of the tree.
Auxiliary Space: O(1), since no extra space has been taken.

Another Approach:
We can also find the inorder successor and inorder predecessor using inorder traversal. Check if the current node is smaller than the given key for the

predecessor and for a successor, check if it is greater than the given key. If it is greater than the given key then, check if it is smaller than the already stored value in the successor then, update it. At last, get the predecessor and successor stored in q(successor) and p(predecessor).

C++

```
// CPP code for inorder successor

// and predecessor of tree

#include<iostream>

#include<stdlib.h>


using namespace std;


struct Node

{

    int data;

    Node* left,*right;

};


// Function to return data

Node* getnode(int info)

{

    Node* p = (Node*)malloc(sizeof(Node));

    p->data = info;

    p->right = NULL;

    p->left = NULL;

    return p;

}
```

```
/*
since inorder traversal results in
ascending order visit to node , we
can store the values of the largest
no which is smaller than a (predecessor)
and smallest no which is large than
a (successor) using inorder traversal
*/
void find_p_s(Node* root,int a,
              Node** p, Node** q)
{
    // If root is null return
    if(!root)
        return ;

    // traverse the left subtree
    find_p_s(root->left, a, p, q);

    // root data is greater than a
    if(root&&root->data > a)
    {

        // q stores the node whose data is greater
        // than a and is smaller than the previously
        // stored data in *q which is successor
        if((!*q) || (*q) && (*q)->data > root->data)
                *q = root;
    }

    // if the root data is smaller than
    // store it in p which is predecessor
    else if(root && root->data < a)
    {
        *p = root;
    }

    // traverse the right subtree
    find_p_s(root->right, a, p, q);
}

// Driver code
int main()
{
    Node* root1 = getnode(50);
    root1->left = getnode(20);
    root1->right = getnode(60);
    root1->left->left = getnode(10);
    root1->left->right = getnode(30);
    root1->right->left = getnode(55);
    root1->right->right = getnode(70);
    Node* p = NULL, *q = NULL;

    find_p_s(root1, 55, &p, &q);
    if(p)

        cout << p->data;

    if(q)

        cout << " " << q->data;

    return 0;

}
```

Output

```
50 60
```

Complexity Analysis:
Time Complexity: O(n), where n is the total number of nodes in the tree. In the worst case as explained above we travel the whole tree.
Auxiliary Space: O(n).
Iterative method:
Input: root node, key
output: predecessor node, successor node

1. *set suc and pre as NULL initially.*
2. *Create a Node temp1 and set it to root node, temp1 will give the successor while traversing*
3. *In first while loop, if temp1->key>key, then temp1->key may be a successor of the key and go to the left of temp.*
4. *else, go to the right.*
5. *Create a Node temp2 and set it to root node, temp2 will give the predecessor while traversing*
6. *In second while loop, if temp2->key<key, then temp1->key may be a predecessor of the key and go to the right of temp.*
7. *else, go to the left.*

Following is the implementation of the above algorithm:

- C++
- Java
- Python3
- C#

- Javascript

```
// C++ program to find predecessor and successor in a BST
#include <iostream>
using namespace std;

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

// This function finds predecessor and successor of key in
// BST. It sets pre and suc as predecessor and successor
// respectively
void findPreSuc(Node* root, Node*& pre, Node*& suc, int key)
{
    // set pre and suc as NULL initially
    pre = NULL;
    suc = NULL;

    // set temp node as root
    Node* temp1 = root;
    while (temp1) {
        // the maximum value in left subtree is successor
        if (temp1->key > key) {
            suc = temp1;
            temp1 = temp1->left;
        }
        else
            temp1 = temp1->right;
    }
    Node* temp2 = root;
    while (temp2) {
        // the minimum value in right subtree is predecessor
        if (temp2->key < key) {
            pre = temp2;
            temp2 = temp2->right;
        }
        else
            temp2 = temp2->left;
    }
    return;
```

```
}

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

/* A utility function to insert a new node with given key in
 * BST */
Node* insert(Node* node, int key)
{
    if (node == NULL)
        return newNode(key);
    if (key < node->key)
        node->left = insert(node->left, key);
    else
        node->right = insert(node->right, key);
    return node;
}

// Driver program to test above function
int main()
{
    int key = 65; // Key to be searched in BST

    /* Let us create following BST
                            50
                    /      \
                    30      70
                    / \ / \
            20 40 60 80 */
    Node* root = NULL;
    root = insert(root, 50);
    insert(root, 30);
    insert(root, 20);
    insert(root, 40);
    insert(root, 70);
    insert(root, 60);
    insert(root, 80);
    Node *pre = NULL, *suc = NULL;

    findPreSuc(root, pre, suc, key);
    if (pre != NULL)
        cout << "Predecessor is " << pre->key << endl;
    else
        cout << "No Predecessor";

    if (suc != NULL)
        cout << "Successor is " << suc->key;
    else
        cout << "No Successor";
    return 0;
}
```

Output

```
Predecessor is 60

Successor is 70
```

Complexity Analysis:

Time Complexity: O(n), where n is the total number of nodes in the tree. In the worst case as explained above we travel the whole tree.

Auxiliary Space: O(1).