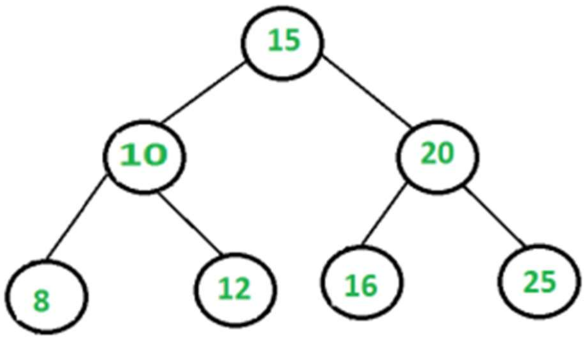
Find a pair with given sum in a Balanced BST

Given a Balanced Binary Search Tree and a target sum, write a function that returns true if there is a pair with sum equals to target sum, otherwise return false. Expected time complexity is O(n) and only O(Logn) extra space can be used. Any modification to Binary Search Tree is not allowed. Note that height of a Balanced BST is always O(Logn).



This problem is mainly extension of the previous post. Here we are not allowed to modify the BST.
The **Brute Force Solution** is to consider each node in the BST and search for (target – node->val) in the BST.
Below is the implementation of the above approach:

C++

```
// C++ code to find a pair with given sum

// in a Balanced BST


#include <bits/stdc++.h>

using namespace std;

// A binary tree node

class Node {

public:

    int data;

    Node* left;
    Node* right;
    Node(int d)
    {
        data = d;
        left = NULL;
        right = NULL;
    }
};

class BinarySearchTree {

    // Root of BST
public:
    Node* root;

    // Constructor
    BinarySearchTree() { root = NULL; }
    /* A recursive function to insert a new key in BST */
    Node* insertRec(Node* root, int data)
    {

        /* If the tree is empty, return a new node */
        if (root == NULL) {
            root = new Node(data);
            return root;
        }
    /* Otherwise, recur down the tree */
    if (data < root->data)
        root->left = insertRec(root->left, data);
    else if (data > root->data)
        root->right = insertRec(root->right, data);

    return root;
}

// This method mainly calls insertRec()
void insert(int key) { root = insertRec(root, key); }

bool isPairPresent(Node* root, Node* temp, int target)
{
    if (temp == NULL)
        return false;

    return search(root, temp, target - temp->data)
           || isPairPresent(root, temp->left, target)
           || isPairPresent(root, temp->right, target);
}

bool search(Node* root, Node* temp, int k)
{

    if (root == NULL)
        return false;
        Node* c = root;
        bool flag = false;
        while (c != NULL && flag != true) {
            if (c->data == k && temp != c) {
                flag = true;
                cout << "Pair Found: " << c->data << " + "
                     << temp->data;
                return true;
            }
            else if (k < c->data)
                c = c->left;
            else
                c = c->right;
        }

        return false;
    }
};

// Driver function
int main()
{
    BinarySearchTree* tree = new BinarySearchTree();
    /*
            15
            /     \
        10      20

        / \      / \

        8 12 16 25 */

    tree->insert(15);

    tree->insert(10);

    tree->insert(20);

    tree->insert(8);

    tree->insert(12);

    tree->insert(16);

    tree->insert(25);


    bool test

        = tree->isPairPresent(tree->root, tree->root, 35);

    if (!test)

        cout << "No such values are found!";

}
```

Output

```
Pair Found: 20 + 15
```

Time Complexity: O(N^2logN), where N is the number of nodes in the given tree.
Auxiliary Space: O(logN), for recursive stack space.

A **Better Solution** is to create an auxiliary array and store the Inorder traversal of BST in the array. The array will be sorted as Inorder traversal of BST always produces sorted data. Once we have the Inorder traversal, we can pair in O(n) time (See this for details). This solution works in O(n) time but requires O(n) auxiliary space.
Implementation:

C++

```
#include<bits/stdc++.h>
using namespace std;

class Node{
    public:
      int data;
      Node* left;
      Node* right;
      Node(int d)
      {
          data=d;
          left=NULL;
          right=NULL;
      }
};

class BinarySearchTree {

    // Root of BST
    public:
    Node *root;

    // Constructor
    BinarySearchTree()
    {
        root = NULL;
    }
```

```
// Utility function for inorder traversal of the tree
void inorderUtil(Node* node)
{
    if (node == NULL)
        return;

    inorderUtil(node->left);
    cout << node->data << " ";
    inorderUtil(node->right);
}

// Inorder traversal of the tree
void inorder()
{
    inorderUtil(this->root);
}

/* A recursive function to insert a new key in BST */
Node* insertRec(Node* root, int data)
{

    /* If the tree is empty, return a new node */
    if (root == NULL) {
        root = new Node(data);
        return root;
    }
    /* Otherwise, recur down the tree */
    if (data < root->data)
        root->left = insertRec(root->left, data);
    else if (data > root->data)
        root->right = insertRec(root->right, data);

    return root;
}

// This method mainly calls insertRec()
void insert(int key)
{
    root = insertRec(root, key);
}



// Method that adds values of given BST into vector
// and hence returns the vector
vector<int> treeToList(Node* node, vector<int>& list)
{
    // Base Case
    if (node == NULL)
        return list;

    treeToList(node->left, list);
    list.push_back(node->data);
```

```
        treeToList(node->right, list);

        return list;
    }

    // method that checks if there is a pair present
    bool isPairPresent(Node* node, int target)
    {
        // This list a1 is passed as an argument
        // in treeToList method
        // which is later on filled by the values of BST
        vector<int> a1;

        // a2 list contains all the values of BST
        // returned by treeToList method
        vector<int> a2 = treeToList(node, a1);

        int start = 0; // Starting index of a2

        int end = (int)a2.size() - 1; // Ending index of a2

        while (start < end) {

            if (a2[start] + a2[end] == target) // Target Found!
            {
                cout << "Pair Found: " << a2[start] << " + " << a2[end] << " = " << targ
<< '\n';
                return true;
            }

            if (a2[start] + a2[end]  > target) // decrements end
            {
                end--;
            }

            if (a2[start] + a2[end]  < target) // increments start
            {
                start++;
            }
        }

        cout << "No such values are found!\n";
        return false;
    }
};

// Driver function
int main()
{
    BinarySearchTree *tree = new BinarySearchTree();
    /*
            15
            /     \
        10      20
        / \      / \
        8 12 16 25 */
    tree->insert(15);
    tree->insert(10);
    tree->insert(20);
    tree->insert(8);
    tree->insert(12);
    tree->insert(16);
    tree->insert(25);

    tree->isPairPresent(tree->root, 33);
}
```

Output

```
Pair Found: 8 + 25 = 33
```

Complexity Analysis:

- **Time Complexity:** O(n).
  Inorder Traversal of BST takes linear time.
- **Auxiliary Space:** O(n).
  Use of array for storing the Inorder Traversal.

A **space optimized solution** is discussed in previous post. The idea was to first in-place convert BST to Doubly Linked List (DLL), then find pair in sorted DLL in O(n) time. This solution takes O(n) time and O(Logn) extra space, but it modifies the given BST.

The **solution discussed below takes O(n) time, O(Logn) space and doesn't modify BST**. The idea is same as finding the pair in sorted array (See method 1 of this for details). We traverse BST in Normal Inorder and Reverse Inorder simultaneously. In reverse inorder, we start from the rightmost node which is the maximum value node. In normal inorder, we start from the left most node which is minimum value node. We add sum of current nodes in both traversals and compare this sum with given target sum. If the sum is same as target sum, we return true. If the sum is more than target sum, we move to next node in

reverse inorder traversal, otherwise we move to next node in normal inorder traversal. If any of the traversals is finished without finding a pair, we return false.
Following is the implementation of this approach.

C++

```
/* In a balanced binary search tree
isPairPresent two element which sums to
a given value time O(n) space O(logn) */
#include <bits/stdc++.h>
using namespace std;
#define MAX_SIZE 100

// A BST node
class node {
public:
    int val;
    node *left, *right;
};

// Stack type
class Stack {
public:
    int size;
    int top;
    node** array;
};

// A utility function to create a stack of given size
```

```
Stack* createStack(int size)

{

    Stack* stack = new Stack();

    stack->size = size;

    stack->top = -1;

    stack->array = new node*[(stack->size * sizeof(node*))];

    return stack;

}


// BASIC OPERATIONS OF STACK

int isFull(Stack* stack)

{

    return stack->top - 1 == stack->size;

}


int isEmpty(Stack* stack)

{

    return stack->top == -1;

}


void push(Stack* stack, node* node)

{

    if (isFull(stack))

        return;

    stack->array[++stack->top] = node;

}
```

```
node* pop(Stack* stack)
{
    if (isEmpty(stack))
        return NULL;
    return stack->array[stack->top--];
}

// Returns true if a pair with target
// sum exists in BST, otherwise false
bool isPairPresent(node* root, int target)
{
    // Create two stacks. s1 is used for
    // normal inorder traversal and s2 is
    // used for reverse inorder traversal
    Stack* s1 = createStack(MAX_SIZE);
    Stack* s2 = createStack(MAX_SIZE);

    // Note the sizes of stacks is MAX_SIZE,
    // we can find the tree size and fix stack size
    // as O(Logn) for balanced trees like AVL and Red Black
    // tree. We have used MAX_SIZE to keep the code simple

    // done1, val1 and curr1 are used for
    // normal inorder traversal using s1
    // done2, val2 and curr2 are used for
    // reverse inorder traversal using s2
    bool done1 = false, done2 = false;
```

```
int val1 = 0, val2 = 0;
node *curr1 = root, *curr2 = root;

// The loop will break when we either find a pair or one of the two
// traversals is complete
while (1) {
    // Find next node in normal Inorder
    // traversal. See following post
    // https:// www.geeksforgeeks.org/inorder-tree-traversal-without-recursion/
    while (done1 == false) {
        if (curr1 != NULL) {
            push(s1, curr1);
            curr1 = curr1->left;
        }
        else {
            if (isEmpty(s1))
                done1 = 1;
            else {
                curr1 = pop(s1);
                val1 = curr1->val;
                curr1 = curr1->right;
                done1 = 1;
            }
        }
    }

    // Find next node in REVERSE Inorder traversal. The only
    // difference between above and below loop is, in below loop
    // right subtree is traversed before left subtree
    while (done2 == false) {
        if (curr2 != NULL) {
            push(s2, curr2);
            curr2 = curr2->right;
        }
        else {
            if (isEmpty(s2))
                done2 = 1;
            else {
                curr2 = pop(s2);
                val2 = curr2->val;
                curr2 = curr2->left;
                done2 = 1;
            }
        }
    }

    // If we find a pair, then print the pair and return. The first
    // condition makes sure that two same values are not added
    if ((val1 != val2) && (val1 + val2) == target) {
        cout << "Pair Found: " << val1 << "+ " << val2 << " = " << target << endl;
        return true;
    }

    // If sum of current values is smaller,
            // then move to next node in
            // normal inorder traversal
            else if ((val1 + val2) < target)
                done1 = false;

            // If sum of current values is greater,
            // then move to next node in
            // reverse inorder traversal
            else if ((val1 + val2) > target)
                done2 = false;

            // If any of the inorder traversals is
            // over, then there is no pair
            // so return false
            if (val1 >= val2)
                return false;
        }
    }

    // A utility function to create BST node
    node* NewNode(int val)
    {
        node* tmp = new node();
        tmp->val = val;
        tmp->right = tmp->left = NULL;
        return tmp;
    }
```

```
// Driver program to test above functions
int main()
{
    /*
                15
                / \
            10 20
            / \ / \
            8 12 16 25 */
    node* root = NewNode(15);
    root->left = NewNode(10);
    root->right = NewNode(20);
    root->left->left = NewNode(8);
    root->left->right = NewNode(12);
    root->right->left = NewNode(16);
    root->right->right = NewNode(25);

    int target = 33;
    if (isPairPresent(root, target) == false)
        cout << "\nNo such values are found\n";
    return 0;
}
```

Output

```
Pair Found: 8+ 25 = 33
```

Complexity Analysis:

- **Time Complexity:** O(n).
  Inorder Traversal of BST takes linear time.

- **Auxiliary Space:** O(logn).
  The stack holds **log N** values as at a single time