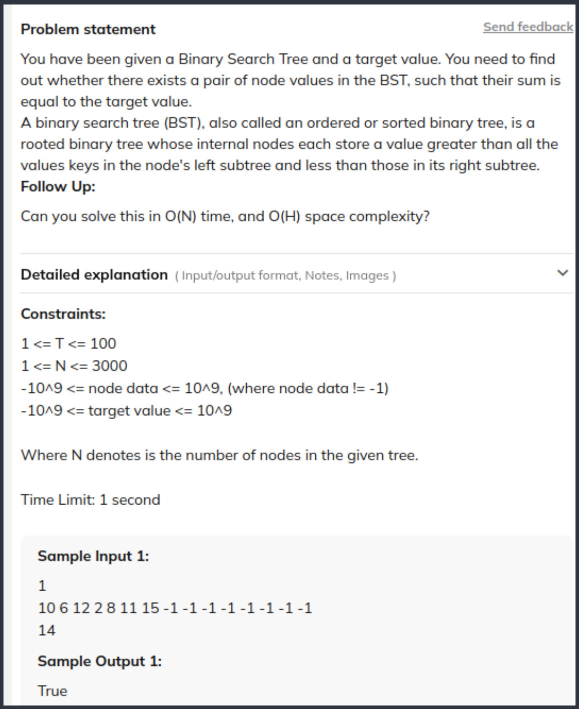
Two Sum IV - Input in BST

→ Brute force:- Traverse BST in level order and let's denote the value of current node as A. For each node check wheather (target - A) is present in the BST or not. We can use BST feature to check if (target - A) is present/not.



Problem statement

Send feedback

You have been given a Binary Search Tree and a target value. You need to find out whether there exists a pair of node values in the BST, such that their sum is equal to the target value.

A binary search tree (BST), also called an ordered or sorted binary tree, is a rooted binary tree whose internal nodes each store a value greater than all the values keys in the node's left subtree and less than those in its right subtree.

Follow Up:

Can you solve this in O(N) time, and O(H) space complexity?

Detailed explanation (Input/output format, Notes, Images)

Constraints:

1 <= T <= 100
1 <= N <= 3000
-10^9 <= node data <= 10^9, (where node data != -1)
-10^9 <= target value <= 10^9

Where N denotes is the number of nodes in the given tree.

Time Limit: 1 second

Sample Input 1:

```
1
10 6 12 2 8 11 15 -1 -1 -1 -1 -1 -1 -1 -1
14
```

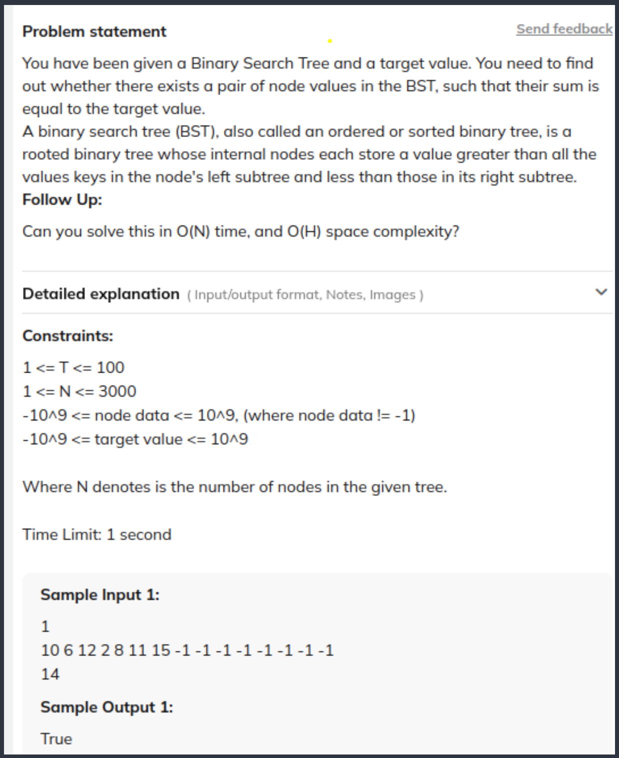
Sample Output 1:

```
True
```

```cpp
bool search(BinaryTreeNode<int>* root, int key){

    // Base case
    if (root == NULL){
        return false;
    }

    return ((root->data == key) ||
        (root->data < key) && search(root->right, key) ||
        (root->data > key) && search(root->left, key));
}

bool twoSumInBST(BinaryTreeNode<int>* root, int target) {

    queue<BinaryTreeNode<int>*> q;

    //Push the root to the queue
    q.push(root);

    while(!q.empty()){

        //Pop the first element from the queue.
        BinaryTreeNode<int>* curr = q.front();
        q.pop();

        //If the value of the current node is not exactly half of the target value, then search for the node with value (target - curr->val)
        if(curr->data * 2 != target){

            if(search(root, target - curr->data) == true){
                return true;
            }
        }

        //Push the left and the right child of the current node to the queue if they are not null.
        if(curr->left){
            q.push(curr->left);
        }
        if(curr->right){
            q.push(curr->right);
        }
    }

    //Finally return false.
    return false;
}
```

Time Complexity : $O(N * H)$

Space Complexity : $O(N)$



Problem statement

Send feedback

You have been given a Binary Search Tree and a target value. You need to find out whether there exists a pair of node values in the BST, such that their sum is equal to the target value.
A binary search tree (BST), also called an ordered or sorted binary tree, is a rooted binary tree whose internal nodes each store a value greater than all the values keys in the node's left subtree and less than those in its right subtree.
Follow Up:
Can you solve this in O(N) time, and O(H) space complexity?

Detailed explanation (Input/output format, Notes, Images)

Constraints:

1 <= T <= 100
1 <= N <= 3000
-10^9 <= node data <= 10^9, (where node data != -1)
-10^9 <= target value <= 10^9

Where N denotes is the number of nodes in the given tree.

Time Limit: 1 second

Sample Input 1:

1
10 6 12 2 8 11 15 -1 -1 -1 -1 -1 -1 -1 -1
14

Sample Output 1:

True

→ Approach;

↳ Use morris traversal to traverse through the array.

↳ Use a set to store the complement of the nodes data.

↳ If a nodes data is found in set return true.

↳ Else false.

```
#include <bits/stdc++.h>

BinaryTreeNode<int> *getSuccessor(BinaryTreeNode<int> *curr) {
  BinaryTreeNode<int> *node = curr;
  if (curr->left) {
    curr = curr->left;
    while (curr->right != NULL && curr->right != node)
      curr = curr->right;
  }
  return curr;
}

bool twoSumInBST(BinaryTreeNode<int> *root, int target) {
  unordered_set<int> mpp;

  BinaryTreeNode<int> *curr = root;

  while (curr != NULL) {
    if (mpp.find(curr->data) != mpp.end())
      return true;
    else {
      int required = target - curr->data;
      mpp.insert(required);
    }
    if (curr->left) {
      BinaryTreeNode<int> *successor = getSuccessor(curr);
      if (successor->right) {
        successor->right = NULL;
        curr = curr->right;
      } else {
        successor->right = curr;
        curr = curr->left;
      }
    } else {
      curr = curr->right;
    }
  }
  return false;
}
```

Time complexity : O(N)
Space Complexity : O(N)

```
bool solve(BinaryTreeNode<int>* root, int target, unordered_set<int> &st){
    if(root == NULL){
        return false;
    }
    if(st.find(target - root->data) != st.end()){
        return true;
    }
    st.insert(root->data);
    return solve(root->left, target, st) || solve(root->right, target, st);
}
bool twoSumInBST(BinaryTreeNode<int>* root, int target) {
    unordered_set<int> st;
    return solve(root, target, st);
}
```

→ Approach :-

↳ Since inorder traversal of BST is in sorted array.

↳ Once we have the sorted array, we can use two pointer approach to get the required result.

```
void inorder(BinaryTreeNode<int>* root,vector<int> &in){

    if(root== NULL){
        return ;
    }
    inorder(root->left,in);
    in.push_back(root->data);
    inorder(root->right,in);
}

bool twoSumInBST(BinaryTreeNode<int>* root, int target) {
    vector<int>inordervalue;
        inorder(root,inordervalue);
    int i=0,j=inordervalue.size()-1;
    while(i<j){
        int sum=inordervalue[i]+inordervalue[j];
        if(sum == target){
           return true;
        }
        else if(sum>target)
           j--;
           else
            i++;
    }
    return false;
}
```

Time Complexity : O(N)
Space Complexity : O(N)

→ Two pointers on BST :-

The idea is the same as the previous approach with space optimization, by applying the two-pointer technique on the given BST. In this approach, we will maintain a forward and a backward iterator that will iterate the BST in the order of in-order and reverse in-order traversal respectively using the two stacks, one maintaining the leftmost value of the BST, start, and one maintaining the rightmost value of BST, end. Now, using the two-pointer technique, we check if the sum of values at the start and end nodes are equal to the target. If they are equal to the target value, then return true. If the values are lesser than the target value, we make forward iterator point to the next element using stack, else if the values are greater than the target value, we make backward iterator point to the previous element using the second stack.
At last, if we find no such two elements, then return false.

```cpp
#include<stack>

bool twoSumInBST(BinaryTreeNode<int>* root, int target) {

    // It stores the iterators for starting and ending indices
    stack<BinaryTreeNode<int> *> start, end;

    // It initialises the starting operator
    BinaryTreeNode<int> *currNode = root;

    while (currNode != NULL){
        start.push(currNode);
        currNode = currNode->left;
    }

    // It initialises the ending operator
    currNode = root;

    while (currNode != NULL){
        end.push(currNode);
        currNode = currNode->right;
    }

    // Using the two-pointer technique.
    while (start.top() != end.top()){

        // It stores the values at starting and end node data
        int val1 = start.top()->data;
        int val2 = end.top()->data;

        // If the sum of values = target, return true
        if (val1 + val2 == target)
        {
            return true;
        }

        // If the sum of values < target, then move to the next greatest closer value.
        if (val1 + val2 < target)
        {
            currNode = start.top()->right;
            start.pop();
            while (currNode != NULL)
            {
                start.push(currNode);
                currNode = currNode->left;
            }
        }

        // If the sum of values > target value, then move to the next smallest closer value .
        else
        {
            currNode = end.top()->left;
            end.pop();
            while (currNode != NULL)
            {
                end.push(currNode);
                currNode = currNode->right;
            }
        }
    }

    // If no two nodes is found, return false
    return false;
}
```

Time Complexity : O(N)

Space Complexity : O(H)