BST:-

Problem Statement

You are given a binary tree, and your task is to determine whether it is a binary search tree (BST) or not.

A binary search tree is defined as follows:

1. The left subtree of a node contains only nodes with keys less than the node's key.
2. The right subtree of a node contains only nodes with keys greater than the node's key.
3. Both the left and right subtrees must also be binary search trees.

Write a program to check if the given binary tree is a BST or not based on these criteria.

Input format :

The first line of input consists of the root node's value as an integer.

For each non-null node, there will be two inputs: the value of the left child (if exists) or -1 if there is no left child, the value of the right child (if exists) or -1 if there is no right child.

Output format :

Print "The given binary tree is a BST" if the input binary tree satisfies the BST properties. Otherwise, print "The given binary tree is not a BST".

Sample test cases :

Input 1 :

```
4 2 1 -1 -1 3 -1 -1 5 -1 -1
```

Output 1 :

```
The given binary tree is a BST
```

Input 2 :

```
3 2 1 -1 -1 4 -1 -1 5 -1 -1
```

Output 2 :

```
The given binary tree is not a BST
```

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

using namespace std;


struct Node {

   int data;

   Node* left;

   Node* right;



   Node(int data) {

      this->data = data;

      this->left = NULL;

      this->right = NULL;
```

```
    }
};

int isBSTUtil(Node* node, int min, int max);


int isBST(Node* node) {
    return (isBSTUtil(node, INT_MIN, INT_MAX));
}


int isBSTUtil(Node* node, int min, int max) {
    if (node == NULL)
        return 1;


    if (node->data < min || node->data > max)
        return 0;


    return isBSTUtil(node->left, min, node->data - 1)
        && // Allow only distinct values
        isBSTUtil(node->right, node->data + 1,
            max); // Allow only distinct values
}

Node* buildTree() {
    int data;
    cin >> data;

    if (data == -1)
```

```
        return NULL;

    Node* root = new Node(data);

    root->left = buildTree();
    root->right = buildTree();

    return root;
}

int main() {
    Node* root = buildTree();

    if (isBST(root))
        cout << "The given binary tree is a BST" << endl;
    else
        cout << "The given binary tree is not a BST" << endl;

    return 0;
}
```

Problem Statement

You are developing software for a car rental agency that manages its vehicle fleet across two locations, City A and City B. Each location's vehicle fleet is represented as a Binary Search Tree (BST) containing unique vehicle IDs.

Your program needs to determine whether the vehicle fleets in both cities are identical or not.

Input format :

The first line of input consists of the space-separated vehicle IDs for City A, terminated by -1.
The second line consists of the space-separated vehicle IDs for City B, terminated by -1.

Output format :

Print "Both vehicle fleets are identical" if the vehicle fleets in both cities are identical.
Otherwise, print "Vehicle fleets are not identical"

Sample test cases :

Input 1 :

```
10 5 15 -1
10 5 15 -1
```

Output 1 :

```
Both vehicle fleets are identical
```

Input 2 :

```
30 25 15-1
25 30 -1
```

Output 2 :

```
Vehicle fleets are not identical
```

```
#include <iostream>

using namespace std;

// BST node
struct Node {
    int data;
    Node* left;
    Node* right;
};

// Utility function to create a new Node
Node* newNode(int data) {
    Node* node = new Node();
    node->data = data;
    node->left = nullptr;
    node->right = nullptr;
    return node;
}

// Function to insert a node into a BST
Node* insert(Node* root, int data) {
    if (root == nullptr)
        return newNode(data);

    if (data < root->data)
```

```
        root->left = insert(root->left, data);
    else if (data > root->data)
        root->right = insert(root->right, data);

    return root;
}

// Function to check if two BSTs are identical
bool areFleetsIdentical(Node* fleet1, Node* fleet2) {
    if (fleet1 == nullptr && fleet2 == nullptr)
        return true;
    else if (fleet1 == nullptr || fleet2 == nullptr)
        return false;
    else {
        if (fleet1->data == fleet2->data && areFleetsIdentical(fleet1->left, fleet2->left) &&
            areFleetsIdentical(fleet1->right, fleet2->right))
            return true;
        else
            return false;
    }
}

int main() {
    Node* cityAFleet = nullptr;
    Node* cityBFleet = nullptr;
    int vehicleID;

    while (1) {
        cin >> vehicleID;
        if (vehicleID == -1)
            break;
```

```
        cityAFleet = insert(cityAFleet, vehicleID);
    }

    while (1) {
        cin >> vehicleID;
        if (vehicleID == -1)
            break;
        cityBFleet = insert(cityBFleet, vehicleID);
    }

    if (areFleetsIdentical(cityAFleet, cityBFleet))
        cout << "Both vehicle fleets are identical" << endl;
    else
        cout << "Vehicle fleets are not identical" << endl;

    return 0;
}
```

Problem Statement

You are working on a critical customer database system for a large e-commerce platform.

The database stores customer information, like their unique customer IDs, in a binary tree structure. Ensuring data integrity is paramount, as duplicate customer IDs can lead to serious data inconsistencies and operational issues.

Your task is to develop a program that checks whether the binary tree containing customer IDs has any duplicate values. Detecting duplicates is essential to maintain the accuracy of customer records.

Input format :

The first line of input consists of an integer representing the value of the root node.
For each node in the tree, there are two integers,
Left child data: an integer representing the value of the left child node. Use -1 to indicate no left child.
Right child data: an integer representing the value of the right child node. Use -1 to indicate no right child.

Output format :

Print "Yes" if there are any duplicate customer IDs in the binary tree.
Otherwise, print "No".

Code constraints :

1 <= Number of nodes in the binary tree <= 100
1 <= Customer ID (node value) <= 100

Sample test cases :

Input 1 :

```
1
1
-1
-1
-1
-1
```

Output 1 :

```
Yes
```

Input 2 :

```
5
3
7
2
8
-1
-1
-1
-1
-1
-1
```

Output 2 :

```
No
```

```
#include <iostream>

// Define the structure for a binary tree node
struct Node {
    int data;
    Node* left;
    Node* right;
};

// Function to count the number of nodes in the binary tree
int nodeCount(Node* root) {
    if (root == nullptr) {
        return 0;
    }
    return 1 + nodeCount(root->left) + nodeCount(root->right);
}
```

```
// Function to create a new binary tree node
Node* createNode(int data) {
    Node* newNode = new Node();
    newNode->data = data;
    newNode->left = newNode->right = nullptr;
    return newNode;
}

// Function to build a binary tree from user input
Node* buildBinaryTree() {
    int data;
    std::cin >> data;

    if (data == -1) {
        return nullptr; // Return nullptr for an empty node
    }

    Node* root = createNode(data);

    root->left = buildBinaryTree();
    root->right = buildBinaryTree();

    return root;
}

// Function to check if a binary tree has duplicate values
bool hasDuplicateValuesUtil(Node* root, int* prev_values, int* prev_index) {
    if (root == nullptr) {
        return false;
    }
```

```
    // Check if the current node's data matches any previous value.
    for (int i = 0; i < *prev_index; i++) {
        if (prev_values[i] == root->data) {
            return true;
        }
    }

    // Store the current node's data in the array of previous values.
    prev_values[(*prev_index)++] = root->data;

    // Recursively check for duplicate values in left and right subtrees.
    return hasDuplicateValuesUtil(root->left, prev_values, prev_index) ||
           hasDuplicateValuesUtil(root->right, prev_values, prev_index);
}

bool hasDuplicateValues(Node* root) {
    int prev_index = 0;
    int numNodes = nodeCount(root);

    int* prev_values = new int[numNodes]; // Dynamically allocate memory

    bool result = hasDuplicateValuesUtil(root, prev_values, &prev_index);

    delete[] prev_values; // Free dynamically allocated memory

    return result;
}

int main() {
    Node* root = nullptr;
```

```
    root = buildBinaryTree();

    if (hasDuplicateValues(root)) {
        std::cout << "Yes" << std::endl;
    } else {
        std::cout << "No" << std::endl;
    }

    return 0;
}
```

Problem Statement

Imagine you are working on a coding competition platform where participants are required to solve various algorithmic problems.

One of the challenges involves checking whether a given array of integers represents a valid in-order traversal of a Binary Search Tree (BST).

Your task is to develop a program that can validate their solutions. Your program should take the input array and determine if it's a valid BST in order traversal.

Input format :

The first line of input consists of an integer **N**, representing the number of elements in the array.
The second line consists of **N** space-separated integers, representing the elements of the array.

Output format :

Print "Yes" if the array represents an inorder traversal of a BST.
Otherwise, print "No".

Sample test cases :

Input 1 :

```
5
2 4 6 8 9
```

Output 1 :

```
Yes
```

Input 2 :

```
5
7 4 1 2 5
```

Output 2 :

```
No
```

```
#include <iostream>

#include <climits>
```

```
// Function to check if an array represents an inorder traversal of a BST
bool isValidInorder(int arr[], int n) {
    // If the array is empty, it is a valid BST
    if (n == 0) {
        return true;
    }

    // Initialize the minimum possible value for the root
    int min_val = INT_MIN;

    // Traverse the array and check if it is in ascending order
    for (int i = 0; i < n; i++) {
        // If the current element is less than the minimum value, it's not a BST
        if (arr[i] < min_val) {
            return false;
        }
        // Update the minimum value
        min_val = arr[i];
    }

    // If the array is in ascending order, it represents an inorder traversal of a BST
    return true;
}

int main() {
    int n;

    // Read the size of the array
    std::cin >> n;
```

```
    int arr[n];

    // Read the array elements
    for (int i = 0; i < n; i++) {
        std::cin >> arr[i];
    }

    // Check if the array represents an inorder traversal of a BST
    if (isValidInorder(arr, n)) {
        std::cout << "Yes" << std::endl;
    } else {
        std::cout << "No" << std::endl;
    }

    return 0;
}
```

Problem Statement

Imagine you are building a program to assist a librarian in managing a library's book inventory. The library uses a binary search tree (BST) to organize books based on their identification numbers.

Each book's identification number is unique and corresponds to a node in the BST. The librarian needs your help to find the predecessor of a given book's identification number through an **in-order** traversal of the tree.

Write a program to accomplish the above task.

Input format :

The first line of input consists of an integer **N**, representing the number of books to be inserted into the BST.

The second line consists of **N** space-separated integers, representing the identification numbers of the books to be inserted.

The last line consists of an integer **M**, representing the identification number of the book for which you need to find the predecessor.

Output format :

If the predecessor exists, print "Inorder Predecessor: " followed by the predecessor book's identification number.

If the predecessor doesn't exist, the output prints "Doesn't exist."

Sample test cases :

Input 1 :

```
4
1 2 3 4
```

```
4
```

Output 1 :

```
Inorder Predecessor : 3
```

Input 2 :

```
4
1 2 3 4
1
```

Output 2 :

```
Doesn't exist
```

Input 3 :

```
5
66 25 4 15 87
25
```

Output 3 :

```
Inorder Predecessor : 15
```

```
#include <iostream>

struct Node {
   int data;
   Node* left;
   Node* right;
};

int predecessor;

void findPredecessor(Node* root, int val) {
   if (root != nullptr) {
      if (root->data == val) {
         if (root->left != nullptr) {
            Node* t = root->left;
            while (t->right != nullptr) {
               t = t->right;
            }
            predecessor = t->data;
         }
      } else if (root->data > val) {
```

```
            findPredecessor(root->left, val);
        } else if (root->data < val) {
            predecessor = root->data;
            findPredecessor(root->right, val);
        }
    }
}

Node* newNode(int data) {
    Node* node = new Node;
    node->data = data;
    node->left = nullptr;
    node->right = nullptr;
    return node;
}

Node* insert(Node* root, int data) {
    if (root == nullptr) {
        return newNode(data);
    } else {
        if (data <= root->data) {
            root->left = insert(root->left, data);
        } else {
            root->right = insert(root->right, data);
        }
        return root;
    }
}

int main() {
    Node* root = nullptr;
```

```
    int n, data;
    std::cin >> n;

    for (int i = 0; i < n; ++i) {
        std::cin >> data;
        root = insert(root, data);
    }

    int targetData;
    std::cin >> targetData;

    predecessor = -1;
    findPredecessor(root, targetData);

    if (predecessor != -1) {
        std::cout << "Inorder Predecessor : " << predecessor << std::endl;
    } else {
        std::cout << "Doesn't exist";
    }

    delete root;

    return 0;
}
```

Problem Statement

You find yourself in a mysterious binary forest where each tree node is associated with a treasure chest containing a certain number of gold coins.

Your task is to navigate through the forest, construct a binary tree based on the clues you find, and determine the maximum amount of gold coins you can collect using **preorder** traversal.

Note: If multiple nodes (gold coins) in the tree have the same maximum value, your function should print the value of the first node encountered during the preorder traversal.

Input format :

The first line of input consists of an integer **N**, representing the number of treasure chests you discover in the forest.

The second line consists of **N** space-separated integers, each representing the number of gold coins in a treasure chest encountered in the order of your journey.

Output format :

The output prints a single integer, which represents the maximum amount of gold coins you can collect by navigating through the binary tree constructed from the given clues.

Sample test cases :

Input 1 :

```
7
10 5 15 3 8 12 18
```

Output 1 :

```
18
```

Input 2 :

```
6
10 5 20 8 15 20
```

Output 2 :

```
20
```

```
#include <iostream>

#include <climits> // Include this header for INT_MIN and INT_MAX

using namespace std;


struct Node {

   int data;

   struct Node* left;

   struct Node* right;

};


struct Node* createNode(int data) {

   struct Node* newNode = new struct Node;

   newNode->data = data;

   newNode->left = NULL;

   newNode->right = NULL;

   return newNode;

}
```

```
struct Node* insertNode(struct Node* root, int data) {
    if (root == NULL) {
        return createNode(data);
    }
    if (data < root->data) {
        root->left = insertNode(root->left, data);
    }
    else {
        root->right = insertNode(root->right, data);
    }
    return root;
}

void findMaxNodePreorder(struct Node* root, int& maxVal) {
    if (root == NULL)
        return;

    if (root->data > maxVal)
        maxVal = root->data;

    findMaxNodePreorder(root->left, maxVal);
    findMaxNodePreorder(root->right, maxVal);
}

int main() {
    struct Node* root = NULL;
    int n, data;
    cin >> n;
    cin >> data;
    root = insertNode(root, data);
```

```
    for (int i = 1; i < n; i++) {
        cin >> data;
        insertNode(root, data);
    }

    int maxVal = INT_MIN; // Include <climits> to use INT_MIN and INT_MAX
    findMaxNodePreorder(root, maxVal);
    cout << maxVal << endl;

    return 0;
}
```

Problem Statement

Rahul is working on a program for a tree manipulation application. Your task is to help Rahul implement a program that converts a given binary tree into its mirror image using **preorder** traversal.

Your program should traverse the binary tree in a preordered manner and swap the left and right subtrees of each node to achieve the mirror image.

Write a function that performs the conversion to the mirror image using a preorder traversal approach. The function should take the root of the binary tree as input and modify the tree to obtain its mirror image.

Input format :

The first line of input consists of an integer **N,** representing the number of nodes in the binary tree.

The second line consists of **N** space-separated integers representing the values of the nodes.

Output format :

The first line of output should display the preorder traversal of the original binary tree.

The second line should display the preorder traversal of the binary tree after converting it into its mirror image.

Code constraints :

1 <= node values <= 200

Sample test cases :

Input 1 :

```
5
4 2 6 1 3
```

Output 1 :

```
Original tree: 4 2 1 3 6
Mirror Image: 4 6 2 3 1
```

Input 2 :

```
8
41 52 36 51 24 64 12 25
```

Output 2 :

```
Original tree: 41 36 24 12 25 52 51 64
Mirror Image: 41 52 64 51 36 24 25 12
```

```
#include <iostream>
using namespace std;

struct Node {
   int data;
   struct Node* left;
   struct Node* right;
};

struct Node* createNode(int data) {
   struct Node* newNode = new struct Node;
   newNode->data = data;
   newNode->left = NULL;
   newNode->right = NULL;
   return newNode;
}

struct Node* insertNode(struct Node* root, int data) {
   if (root == NULL) {
      return createNode(data);
   }
   if (data < root->data) {
      root->left = insertNode(root->left, data);
   }
   else {
      root->right = insertNode(root->right, data);
   }
   return root;
}

void preorderTraversal(struct Node* root) {
```

```
    if (root != NULL) {
        cout << root->data << " ";
        preorderTraversal(root->left);
        preorderTraversal(root->right);
    }
}

void mirrorImage(struct Node* root) {
    if (root == NULL) {
        return;
    }

    // Swap left and right subtrees
    struct Node* temp = root->left;
    root->left = root->right;
    root->right = temp;

    mirrorImage(root->left);
    mirrorImage(root->right);
}

int main() {
    struct Node* root = NULL;
    int n, data;
    cin >> n;
    for (int i = 0; i < n; i++) {
        cin >> data;
        root = insertNode(root, data);
    }

    cout << "Original tree: ";
```

```
    preorderTraversal(root);
    cout << endl;

    mirrorImage(root);

    cout << "Mirror Image: ";
    preorderTraversal(root);
    cout << endl;

    return 0;
}
```

Problem Statement

Amla is the librarian at a local library. She needs a tool to manage the library's catalog efficiently.

She decides to use a program that can organize the book IDs in ascending order to help her locate books more easily.

Write a program that allows Sarah to input the book IDs one by one. The program should then construct a binary search tree with the book IDs and display the book IDs in ascending order using an **in-order** traversal.

Input format :
The first line of input consists of an integer **N**, the number of books in the library.
The next line consists of **N** positive integers separated by space, representing the unique book IDs.

Output format :
The output prints the in-order traversal of the book IDs.

Code constraints :
$1 \leq$ book ID $\leq 10^5$

Sample test cases :

Input 1 :

```
7
53 36 74 23 41 68 87
```

Output 1 :

```
23 36 41 53 68 74 87
```

Input 2 :

```
4
145 632 541 987
```

Output 2 :

```
145 541 632 987
```

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

```
#define MAX_NODES 1000

struct Node {
   int key;
   struct Node* left;
   struct Node* right;
};

struct Node nodes[MAX_NODES];
int nodeCount = 0;

struct Node* createNode(int item) {
   struct Node* newNode = &nodes[nodeCount++];
   newNode->key = item;
   newNode->left = newNode->right = NULL;
   return newNode;
}

void insert(struct Node* root, int key) {
   while (1) {
      if (key < root->key) {
         if (root->left == NULL) {
            root->left = createNode(key);
            return;
         }
         root = root->left;
      } else if (key > root->key) {
         if (root->right == NULL) {
            root->right = createNode(key);
            return;
```

```
            }
            root = root->right;
        }
    }
}

void inorder(struct Node* root) {
    if (root != NULL) {
        inorder(root->left);
        printf("%d ", root->key);
        inorder(root->right);
    }
}

int main() {
    struct Node* root = NULL;
    int n;
    scanf("%d", &n);

    for (int i = 0; i < n; i++) {
        int key;
        scanf("%d", &key);
        if (root == NULL) {
            root = createNode(key);
        } else {
            insert(root, key);
        }
    }

    inorder(root);
```

```
    return 0;
}
```

Problem Statement

Megna is implementing a program to construct a binary tree and perform a **post-order** traversal on it. The binary tree will be built according to a sequence of integers provided as input. Each integer represents a node in the binary tree.

Your task is to help her construct the tree and then perform a post-order traversal to print the values of the nodes in the specified order.

Input format :

The first line of input consists of an integer, n, representing the number of integers in the sequence.

The second line of input consists of n space-separated integers, where each integer represents a node's value to be inserted into the binary tree.

Output format :

The output prints the values of the nodes in the binary tree in post-order traversal, separated by spaces.

Refer to the sample output for formatting specifications.

Code constraints :

1 <= n <= 20

1 <= elements <= 1000

Sample test cases :

Input 1 :

```
5
1 3 4 5 2
```

Output 1 :

```
5 2 3 4 1
```

Input 2 :

```
5
1 7 9 5 6
```

Output 2 :

```
5 6 7 9 1
```

```
#include <iostream>

using namespace std;


struct Node {

    int data;

    Node* left;

    Node* right;


    Node(int val) : data(val), left(nullptr), right(nullptr) {}

};
```

```
Node* insert(Node* root, int data) {
    if (root == nullptr) {
        return new Node(data);
    } else {
        Node* cur = new Node(data);

        if (root->left == nullptr) {
            root->left = cur;
        } else if (root->right == nullptr) {
            root->right = cur;
        } else {
            // If both left and right children are already present,
            // you can choose one side to insert the new node, e.g., left.
            root->left = insert(root->left, data);
        }

        return root;
    }
}

void postOrder(Node* root) {
    if (root != nullptr) {
        postOrder(root->left);
        postOrder(root->right);
        cout << root->data << " ";
    }
}

int main() {
    Node* root = nullptr;
```

```
    int n;
    int data;

    cin >> n;

    while (n-- > 0) {
        cin >> data;
        root = insert(root, data);
    }

    postOrder(root);
    return 0;
}
```

Problem Statement

Harish is learning about binary trees and postorder traversals in his computer science class. He is given an assignment to write a program that constructs the postorder traversal of a binary tree from an array of integers. Can you help him solve this problem?

Note:

- The binary tree can have a maximum of 100 nodes.
- All node values in the binary tree are unique.
- For a node at index i in the input array: Its left child is at index 2 * i + 1, and its right child is at index 2 * i + 2.
- The array represents the binary tree's structure, with each index corresponding to a node and its value.

Input format :
The first line contains an integer n, representing the number of elements in the array.
The second line contains n space-separated integers, arr [i], representing the values of the nodes in the binary tree. The elements are given in the order of a binary tree, where the i-th element is the value of the i-th node.
Output format :
The output displays a single line containing n space-separated integers, which represent the postorder traversal of the binary tree constructed from the input array.

Refer to the sample output for formatting specifications.
Code constraints :
$1 \leq n \leq 15$
$1 \leq arr[i] \leq 100$
Sample test cases :
Input 1 :

```
3
1 2 3
```

Output 1 :

```
2 3 1
```

Input 2 :

```
5
1 2 3 4 5
```

Output 2 :

```
4 5 2 3 1
```

```
#include <iostream>
#include <cstring>

using namespace std;

const int MAX_NODES = 100;

struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;
    TreeNode() : val(0), left(nullptr), right(nullptr) {}
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
    TreeNode(int x, TreeNode* left, TreeNode* right) : val(x), left(left), right(right) {}
};

int ans[MAX_NODES];
int idx = 0;

void postorder(TreeNode* root) {
    if (root == nullptr)
        return;
    postorder(root->left);
    postorder(root->right);
    ans[idx++] = root->val;
```

```
}

void fillArray(TreeNode* root) {
    postorder(root);
}

int* postorderTraversal(TreeNode* root) {
    fillArray(root);
    return ans;
}

TreeNode* createTree(int arr[], int i, int n) {
    TreeNode* newNode = nullptr;
    if (i < n) {
        newNode = new TreeNode(arr[i]);
        newNode->left = createTree(arr, 2 * i + 1, n);
        newNode->right = createTree(arr, 2 * i + 2, n);
    }
    return newNode;
}

int main() {
    int n;
    cin >> n;

    int arr[MAX_NODES];
    for (int i = 0; i < n; ++i) {
        cin >> arr[i];
    }

    TreeNode* root = createTree(arr, 0, n);
```

```
    int* postorderResult = postorderTraversal(root);

    for (int i = 0; i < n; ++i) {
        cout << postorderResult[i] << " ";
    }

    return 0;
}
```

Problem Statement

Ragu is studying binary search trees (BSTs) and postorder traversals in his computer science class. He has been given an assignment to write a program that constructs a BST from a sequence of positive integers and then performs a postorder traversal of the tree. Can you help him complete this task?

Note:
The Binary Search Tree may not be balanced.
The binary search tree is not necessarily balanced.
The binary search tree can have a maximum of 1000 nodes.

Input format :
The input consists of a sequence of positive integers, each on a separate line, where each integer d represents a node's value. The sequence ends when Ragu enters -1, indicating the end of the input.

Output format :
The output displays the postorder traversal of the constructed binary search tree.

Refer to the sample output for the formatting specifications.

Code constraints :
1 <= input elements <= 100
The input will be terminated by entering '-1'.

Sample test cases :

Input 1 :

```
6
3
1
4
2
-1
```

Output 1 :

```
Post order Traversal:
2 1 4 3 6
```

Input 2 :

```
1
7
9
5
6
```

```
-1
```

Output 2 :

```
Post order Traversal:
6 5 9 7 1
```

Input 3 :

```
100
20
200
10
30
150
300
-1
```

Output 3 :

```
Post order Traversal:
10 30 20 150 300 200 100
```

```
#include <iostream>
#include <cstdlib>
using namespace std;
struct node
{
 int data;
 struct node*left;
 struct node*right;
};
struct node*root;
void append(int d)
{
 struct node*newnode = (struct node*)malloc(sizeof(struct node));
 struct node*temp = root;
 newnode->data = d;
 newnode->left = NULL;
 newnode->right = NULL;
 if(root == NULL)
 {
   root = newnode;
```

```
 }
 else
 {
  while(true)
  {
   if(d < temp->data)
   {
    if(temp->left != NULL)
    {
     temp = temp->left;
    }
    else
    {
     temp->left = newnode;
     break;
    }
   }
   else
   {
    if(temp->right != NULL)
    {
     temp = temp->right;
    }else
    {
     temp->right = newnode;
     break;
    }
   }
  }
 }
}
```

```
void postorder(struct node*root)
{
 if(root != NULL)
 {

   postorder(root->left);
   postorder(root->right);
   cout<<root->data<<" ";
 }
}
int main()
{
 int d;
 do
 {
   cin>>d;
   if(d > 0)
     append(d);
 }while(d != -1);
 cout<<"Post order Traversal:"<<endl;
 postorder(root);
 return 0;
}
```

Problem Statement

Rohith is a computer science student who is fascinated by data structures and algorithms. He recently learned about binary search trees and their traversal techniques. He wants to implement a program to create a binary search tree and perform post-order traversal on it, along with calculating the height of the tree.

Note:
The height of the tree is computed by finding the maximum height between the left and right subtrees of the root node and then adding 1 to that maximum height.

Input format :
The first line contains an integer, n, representing the number of elements to be inserted into the binary search tree.

The second line contains n space-separated integers, each representing an element to be inserted into the binary search tree.

Output format :

The first line should display the post-order traversal of the binary search tree, with elements separated by spaces.

The second line should display the height of the binary search tree.

Refer to the sample output for formatting specifications.

Code constraints :

1 <= n <= 20

1 <= elements <= 1000

Sample test cases :

Input 1 :

```
7
22 12 30 8 20 25 40
```

Output 1 :

```
Post-order traversal: 8 20 12 25 40 30 22
Height of the tree: 2
```

```
#include <stdio.h>

#include <stdlib.h>


struct Node {

   int data;

   struct Node* left;

   struct Node* right;

};


struct Node* createNode(int data) {

   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));

   newNode->data = data;

   newNode->left = NULL;

   newNode->right = NULL;

   return newNode;

}


struct Node* insert(struct Node* root, int data) {

   if (root == NULL) {
```

```
        return createNode(data);
    } else {
        struct Node* cur;
        if (data <= root->data) {
            cur = insert(root->left, data);
            root->left = cur;
        } else {
            cur = insert(root->right, data);
            root->right = cur;
        }
        return root;
    }
}

void postOrder(struct Node* root) {
    if (root == NULL)
        return;

    postOrder(root->left);
    postOrder(root->right);
    printf("%d ", root->data);
}

int getHeight(struct Node* root) {
    if (root == NULL)
        return -1; // Height of an empty tree is -1

    int leftHeight = getHeight(root->left);
    int rightHeight = getHeight(root->right);

    return (leftHeight > rightHeight ? leftHeight : rightHeight) + 1;
```

```
}

int main() {
    struct Node* root = NULL;

    int n;
    int data;

    scanf("%d", &n);

    while (n-- > 0) {
        scanf("%d", &data);
        root = insert(root, data);
    }

    printf("Post-order traversal: ");
    postOrder(root);
    printf("\nHeight of the tree: %d\n", getHeight(root));

    return 0;
}
```

Problem Statement

Ragul wants to build a binary search tree (BST) and perform a key search operation on it. He needs your help to accomplish this. Write a program that helps Ragul create a BST and search for a specific key within it.

Note: This kind of question will help in clearing Wipro recruitment.
Input format :
The first line of input consists of the number of nodes **n**.
The second line of input consists of **n** unique values for nodes, separated by a space.
The third line of input consists of the key to be searched.
Output format :
The output displays one of the following messages based on whether the key is found in the binary search tree or not in the following format:

1. If the key is found in the binary search tree, print **"The key <<key value>> is found in the binary search tree"**
2. If the key is not found in the binary search tree, print **"The key <<key value>> is not found in the binary search tree"**

Refer to the sample output for the exact format.

Code constraints :

1 ≤ numNodes ≤ 10

Each node value is a unique integer.

1 <= key <= 1000

Sample test cases :

Input 1 :

```
7
101 102 103 105 106 108 110
102
```

Output 1 :

```
The key 102 is found in the binary search tree
```

Input 2 :

```
7
101 102 103 105 106 108 110
115
```

Output 2 :

```
The key 115 is not found in the binary search tree
```

```
#include <iostream>

using namespace std;


struct Node {

   int data;

   Node* left;

   Node* right;

};


Node* createNode(int value) {

   Node* newNode = new Node();

   if (!newNode) {

      return NULL;

   }

   newNode->data = value;

   newNode->left = newNode->right = NULL;

   return newNode;
```

```
}

Node* insertNode(Node* root, int value) {
    if (root == NULL) {
        return createNode(value);
    }

    if (value < root->data) {
        root->left = insertNode(root->left, value);
    } else if (value > root->data) {
        root->right = insertNode(root->right, value);
    }

    return root;
}

bool searchKey(Node* root, int key) {
    while (root != NULL) {
        if (key == root->data) {
            return true;
        } else if (key < root->data) {
            root = root->left;
        } else {
            root = root->right;
        }
    }
    return false;
}

int main() {
    Node* root = NULL;
```

```
    int numNodes, value, key;

    cin >> numNodes;

    for (int i = 0; i < numNodes; i++) {
        cin >> value;
        root = insertNode(root, value);
    }

    cin >> key;

    bool found = searchKey(root, key);
    if (found) {
        cout << "The key " << key << " is found in the binary search tree";
    } else {
        cout << "The key " << key << " is not found in the binary search tree";
    }

    return 0;
}
```

Problem Statement

Kathir is learning about binary search trees (BSTs) in his computer science class. He has just implemented a simple program to create a binary search tree and calculate the sum of all its leaf nodes. Can you help him validate his code and solve some scenarios to test it?

For example, consider a tree with the following structure:

```
   5
  / \
 3   7
/ \   \
2  4   9
```

In this tree, the leaf nodes are 2, 4, and 9. The program should compute the sum of these leaf node elements, which is 2 + 4 + 9 = 15.

Input format :

The input consists of a series of positive integers as input, each on a separate line.
The input terminates when Kathir enters -1. This signifies the end of input.

You can assume that the input integers are unique.

Output format :

The output displays the sum of all the leaf nodes in the binary search tree in the following format:
"Sum of all leaf nodes are <<value>>"

Refer to the sample output for the formatting specifications.

Code constraints :

All input integers will be positive integers (greater than 0).
The number of nodes in the binary tree is at most 100.
The value of each node is at most 100.

Sample test cases :

Input 1 :

```
5
3
7
2
4
9
-1
```

Output 1 :

```
Sum of all leaf nodes are 15
```

Input 2 :

```
6
3
1
4
2
-1
```

Output 2 :

```
Sum of all leaf nodes are 6
```

```
#include<iostream>

#include<cstdlib>

using namespace std;

struct node

{

 int data;

 struct node* left;

 struct node* right;

};

struct node* root;


struct node* createNode(int d)

{
```

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

void append(int d)
{
 struct node* newnode = createNode(d);
 struct node* temp = root;
 if (root == NULL)
 {
  root = newnode;
 }
 else
 {
  while (true)
  {
   if (d < temp->data)
   {
    if (temp->left != NULL)
    {
     temp = temp->left;
    }
    else
    {
     temp->left = newnode;
     break;
    }
   }
```

```
      else
      {
       if (temp->right != NULL)
       {
         temp = temp->right;
       }
       else
       {
         temp->right = newnode;
         break;
       }
      }
    }
  }
}

void leafsum(node* root, int* sum)
{
  if (!root)
    return;
  if (!root->left && !root->right)
    *sum += root->data;
  leafsum(root->left, sum);
  leafsum(root->right, sum);
}

int main()
{
  int d;
  do
  {
```

```
    cin >> d;
    if (d > 0)
      append(d);
  } while (d != -1);
  int sum = 0;
  leafsum(root, &sum);
  cout << "Sum of all leaf nodes are " << sum;
  return 0;
}
```

Problem Statement

Arun is developing a program that uses a Binary Search Tree (BST) to manage a collection of integers. He has written the code to create a BST and search for elements within it, but he needs your assistance in testing its functionality.

You are tasked with helping Arun by designing a program that performs the following actions:

1. Build a Binary Search Tree (BST) by adding integers to it one by one.
2. Search for a specific integer in the BST to determine if it is present.

Input format :
The first n lines of input consist of a series of positive integers (greater than zero) separated by space.
The input ends when -1 is entered.
The last line of input consists of an integer representing the elements to be inserted into the binary search tree.
Output format :
The output displays one of the following messages:

If the key is present in the binary search tree, print <key> is present in the BST.
If the key is not present in the binary search tree, print <key> is not present in the BST.

Refer to the sample output for the formatting specifications.
Code constraints :
The input integers will be positive and greater than zero.
The number of elements in the binary search tree will be at most 100.
Sample test cases :
Input 1 :

```
50
30
70
20
40
60
-1
30
```

Output 1 :

```
30 is present in the BST
```

Input 2 :

```
6
3
1
2
4
-1
10
```

Output 2 :

```
10 is not present in the BST
```

```
#include<iostream>
#include<cstdlib>
using namespace std;
struct node
{
 int data;
 struct node*left;
 struct node*right;
};
struct node*root;
void append(int d)//3
{
 struct node*newnode = (struct node*)malloc(sizeof(struct node));
 struct node*temp = root;//6
 newnode->data = d;//3
 newnode->left = NULL;
 newnode->right = NULL;
 if(root == NULL)//6
 {
  root = newnode;//6
 }
 else
 {
  while(true)
```

```
    {
      if(d < temp->data)
      {
        if(temp->left != NULL)
        {
          temp = temp->left;
        }
        else
        {
          temp->left = newnode;//6->3
          break;
        }
      }else
      {
        if(temp->right != NULL)
        {
          temp = temp->right;
        }else
        {
          temp->right = newnode;
          break;
        }
      }
    }
  }
}
bool iterativesearch(struct node*root,int key)
{
  while(root != NULL)//10
  {
    if(key > root->data)
```

```
    {
      root = root->right;//10
    }
    else if(key < root->data)
    {
      root = root->left;
    }
    else
      return true;
  }
  return false;
}
int main()
{
  int d,search;
  do
  {
    cin>>d;//3
    if(d > 0)
      append(d);
  }while(d != -1);//-1
  cin>>search;
  if(iterativesearch(root,search))
    cout<<search<<" is present in the BST";
  else
    cout<<search<<" is not present in the BST";
  return 0;
}
```