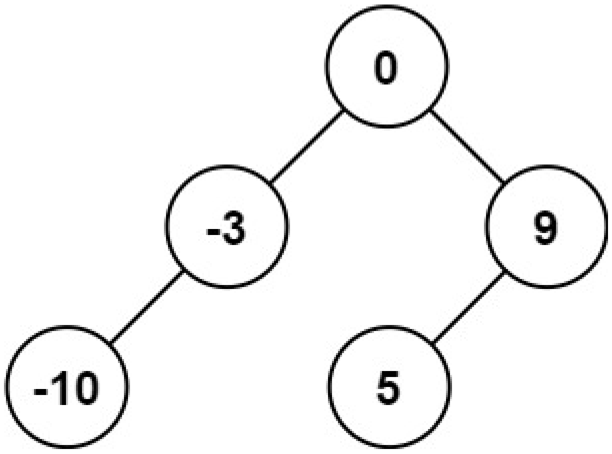
NAME-PRERNA TYAGI	**UID-22BCS16183**
BRANCH-BE-CSE	**SECTION/GROUP-IOT-638-A**
SEMESTER-6	**DATE OF PERFORMANCE-26/02/2023**
SUBJECT NAME-AP-LAB II	**SUBJECT CODE-22CSP-351**

EXPERIMENT -6

Problem-1: Convert Sorted Array to Binary Search Tree

Given an integer array nums where the elements are sorted in ascending order, convert it to a height-balanced binary search tree.

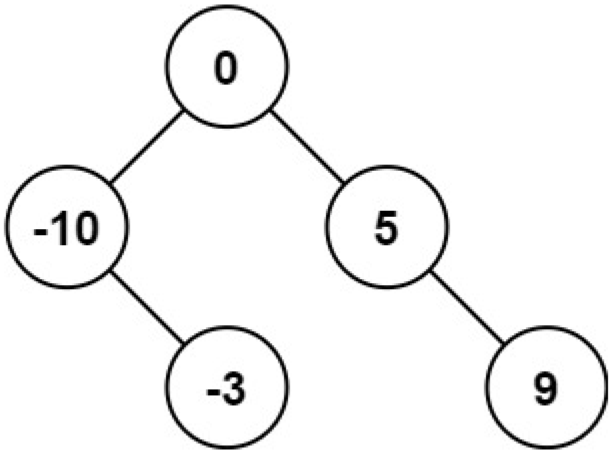


Example 1:

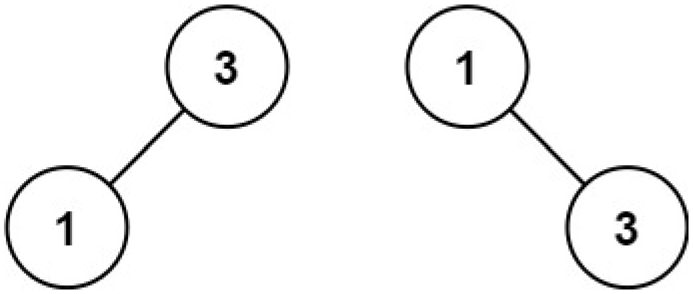
Input: nums = [-10,-3,0,5,9]

Output: [0,-3,9,-10,null,5]

Explanation: [0,-10,5,null,-3,null,9] is also accepted:



Example 2:



Input: nums = [1,3]

Output: [3,1]

Explanation: [1,null,3] and [3,1] are both height-balanced BSTs.

Constraints:

1 <= nums.length <= 104

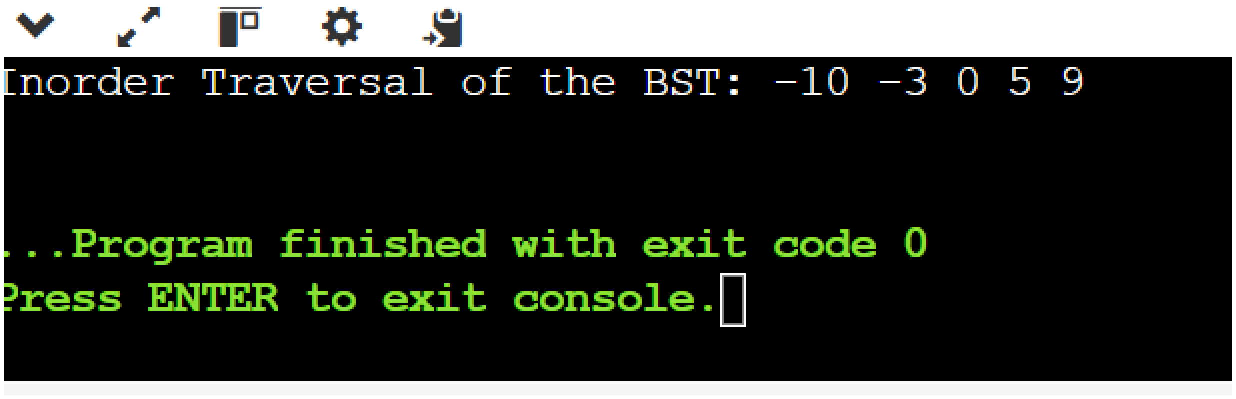
-104 <= nums[i] <= 104

nums is sorted in a strictly increasing order.

2.CODE:

```
 #include <iostream>
#include <vector>

// Definition for a binary tree node.
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode(int x) : val(x), left(NULL), right(NULL) {}
};

class Solution {
public:
    TreeNode* sortedArrayToBST(std::vector<int>& nums) {
        return constructBST(nums, 0, nums.size() - 1);
    }

private:
    TreeNode* constructBST(std::vector<int>& nums, int left, int right) {
        if (left > right) return nullptr;

        int mid = left + (right - left) / 2;
        TreeNode* root = new TreeNode(nums[mid]);
        root->left = constructBST(nums, left, mid - 1);
        root->right = constructBST(nums, mid + 1, right);
        return root;
    }
};

// Function to print inorder traversal of the BST
void inorderTraversal(TreeNode* root) {
    if (root == nullptr) return;
    inorderTraversal(root->left);
    std::cout << root->val << " ";
    inorderTraversal(root->right);
}

int main() {
    Solution solution;
    std::vector<int> nums = {-10, -3, 0, 5, 9};
    TreeNode* root = solution.sortedArrayToBST(nums);

    std::cout << "Inorder Traversal of the BST: ";
    inorderTraversal(root);
    std::cout << std::endl;

    return 0;
}
```

OUTPUT:



```
Inorder Traversal of the BST: -10 -3 0 5 9


...Program finished with exit code 0
Press ENTER to exit console.
```

LEARNING OUTCOMES:

 Understanding Binary Search Trees (BSTs)

- This code demonstrates how to construct a height-balanced BST from a sorted array, ensuring optimal search efficiency (O(log N) for balanced trees).

 Recursive Tree Construction

- The use of recursion to build the tree by selecting the middle element as the root and dividing the array into left and right subarrays helps in understanding divide and conquer algorithms.

 Memory Allocation and Tree Traversal

- The implementation highlights dynamic memory allocation using new for creating tree nodes and demonstrates inorder traversal, which helps in understanding tree traversal techniques.

 Optimized Midpoint Selection

- The method mid = left + (right - left) / 2 prevents integer overflow when calculating the middle index, teaching an important coding best practice for handling large numbers efficiently.

PROBLEM 2

Problem-2: Maximum Depth of Binary Tree

Given the root of a binary tree, return its maximum depth.

A binary tree's maximum depth is the number of nodes along the longest path from the root node down to the farthest leaf node.

Example 1:

Input: root = [3,9,20,null,null,15,7]

Output: 3

Example 2:

Input: root = [1,null,2]

Output: 2

Constraints:

The number of nodes in the tree is in the range [0, 104].

-100 <= Node.val <= 100

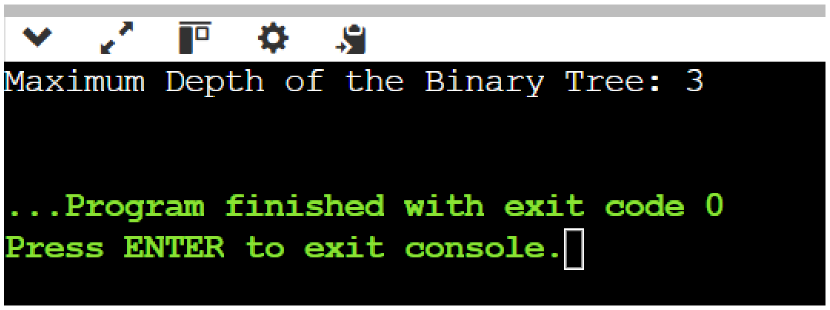
CODE:

OUTPUT:

```
#include <iostream>

// Definition for a binary tree node.
struct TreeNode {
   int val;
   TreeNode *left;
   TreeNode *right;
   TreeNode(int x) : val(x), left(NULL), right(NULL) {}
};

class Solution {
public:
   int maxDepth(TreeNode* root) {
      if (root == nullptr) return 0;
      int leftDepth = maxDepth(root->left);
      int rightDepth = maxDepth(root->right);
      return std::max(leftDepth, rightDepth) + 1;
   }
};
```

```
// Helper function to create a simple binary tree for testing
TreeNode* createSampleTree() {
    TreeNode* root = new TreeNode(3);
    root->left = new TreeNode(9);
    root->right = new TreeNode(20);
    root->right->left = new TreeNode(15);
    root->right->right = new TreeNode(7);
    return root;
}

int main() {
    Solution solution;
    TreeNode* root = createSampleTree();
    return 0;
}
```

OUTPUT:



```
Maximum Depth of the Binary Tree: 3


...Program finished with exit code 0
Press ENTER to exit console.
```

LEARNING OUTCOMES:

 Understanding Recursive Tree Traversal

- The code uses recursion to explore both left and right subtrees, reinforcing the concept of depth-first search (DFS).

 Calculating Maximum Depth Efficiently

- By using divide and conquer, the function determines the depth in O(N) time complexity, where N is the number of nodes in the tree.

 Handling Edge Cases in Binary Trees

- The code correctly handles cases like an empty tree (nullptr root) and unbalanced trees, ensuring robustness.

 Practical Implementation of Tree Structures

- This example reinforces how to define a binary tree, dynamically allocate nodes, and use recursion to solve tree-based problems.