CAPSTONE PROJECT

BINARY SEARCH TREES

CSA0695- DESIGN AND ANALYSIS OF ALGORITHMS

SAVEETHA SCHOOL OF ENGINEERING

SIMATS ENGINEERING

Supervisor

Dr. R. Dhanalakshmi

Done
by
A.HARSHAVARDHAN KARTHEEK REDDY
(192210042)

BINARY SEARCH TREES

PROBLEM STATEMENT:

Given an array nums that represents a permutation of integers from 1 to n. We are going to construct a binary search tree (BST) by inserting the elements of nums in order into an initially empty BST. Find the number of different ways to reorder numbers so that the constructed BST is identical to that formed from the original array nums.

For example, given nums = [2,1,3], we will have 2 as the root, 1 as a left child, and 3 as a right child. The array [2,3,1] also yields the same BST but [3,2,1] yields a different BST.
Return the number of ways to reorder numbers such that the BST formed is identical to the original BST formed from nums.

Example 1:
Input: nums = [2,1,3]
Output: 1
Explanation:
We can reorder nums to be [2,3,1] which will yield the same BST. There are no other ways to reorder numbers which will yield the same BST.

ABSTRACT:

A Binary Search Tree (BST) is a data structure that facilitates efficient searching, insertion, and deletion operations on a set of elements. In a BST, each node contains a key and two child nodes, with the left child node holding a key less than its parent node, and the right child node holding a key greater than its parent node. This property ensures that the BST maintains a sorted order of elements, which allows for fast retrieval operations, typically with a time complexity of $O(\log n)$ for balanced trees, where n is the number of nodes.

INTRODUCTION:

A Binary Search Tree (BST) is a specialised form of binary tree used for efficient data storage and retrieval. It is a fundamental data structure in computer science, commonly utilised for tasks that require dynamic data manipulation, such as searching, inserting, and deleting data.

In a BST, each node has at most two children referred to as the left and right children. The key property that defines a BST is:

For each node: The key of the left child (and all its descendants) is less than the key of the node, and the key of the right child (and all its descendants) is greater than the key of the node.

This ordering property ensures that BSTs are efficient for operations that involve sorting and searching.

1. Search: Determines whether a key exists in the tree. The search operation begins at the root and traverses the tree based on comparisons, moving left or right depending on the key value relative to the current node's key.

2. Insertion: Adds a new key to the tree while preserving the BST property. The new key is compared with the current node's key and inserted into the appropriate subtree, ensuring the binary search property is maintained.
3. Deletion: Removes a key from the tree and restructures the tree to maintain the BST properties. This operation involves three possible scenarios: deleting a leaf node, deleting a node with one child, or deleting a node with two children, where additional steps are required to maintain the tree's order.
4. Traversal: Visits all nodes in a specified order, such as in-order (ascending order of keys), pre-order, or post-order. In-order traversal is particularly useful for producing a sorted sequence of elements.

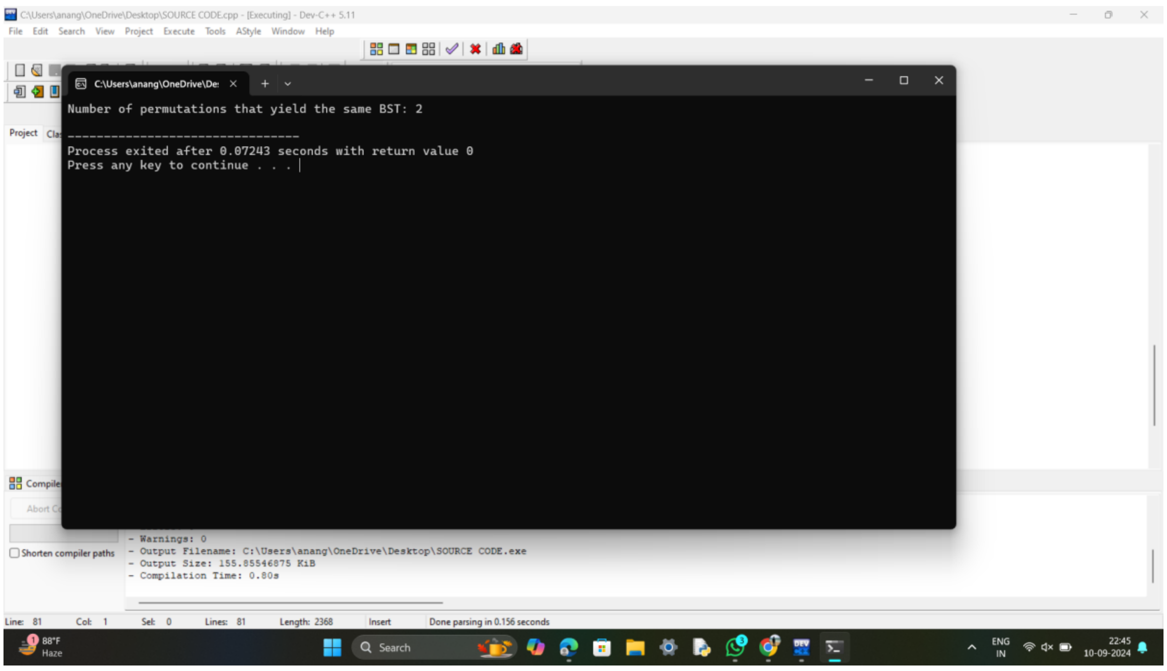
SOURCE CODE:

```
#include <stdio.h>

#include <stdlib.h>

typedef struct TreeNode

{

   int value;

   struct TreeNode *left, *right;

}

TreeNode;

TreeNode* createNode(int value)

{

   TreeNode* newNode = (TreeNode*)malloc(sizeof(TreeNode));

   newNode->value = value;

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

   return newNode;

}

TreeNode* insert(TreeNode* root, int value)

{

   if (root == NULL) return createNode(value);

   if (value < root->value) root->left = insert(root->left, value);

   else root->right = insert(root->right, value);

   return root;

}
```

```
int countPermutations(TreeNode* root)
{
    if (root == NULL) return 1;
    int leftSubtreeSize = 0, rightSubtreeSize = 0;
    TreeNode* temp = root->left;
    while (temp)
            {
        leftSubtreeSize++;
        temp = temp->left ? temp->left : temp->right;
    }
    temp = root->right;
    while (temp)
            {
        rightSubtreeSize++;
        temp = temp->left ? temp->left : temp->right;
    }
    int leftPermutations = countPermutations(root->left);
    int rightPermutations = countPermutations(root->right);
    int binomialCoefficient = 1;
    int n = leftSubtreeSize + rightSubtreeSize;
    int k = leftSubtreeSize;
    for (int i = 0; i < k; i++)
            {
        binomialCoefficient = binomialCoefficient * (n - i) / (i + 1);
    }
    return binomialCoefficient * leftPermutations * rightPermutations;
}
TreeNode* buildBST(int* nums, int size)
```

```
{
    TreeNode* root = NULL;
    for (int i = 0; i < size; i++)
            {
        root = insert(root, nums[i]);
    }
    return root;
}
int main()
{
    int nums[] = {2, 1, 3};
    int size = sizeof(nums) / sizeof(nums[0]);
    TreeNode* root = buildBST(nums, size);
    int result = countPermutations(root);
    printf("Number of permutations that yield the same BST: %d\n", result);
    return 0;
}
```

OUTPUT:


C:\Users\anang\OneDrive\Desktop\SOURCE CODE.cpp - [Executing] - Dev-C++ 5.11
File Edit Search View Project Execute Tools AStyle Window Help
C:\Users\anang\OneDrive\De:
Project
Compile
Abort C
Shorten compiler paths
Line: 81 Col: 1 Sel: 0 Lines: 81 Length: 2368 Insert Done parsing in 0.156 seconds
88°F Haze
Search
ENG IN
22:45 10-09-2024

```
Number of permutations that yield the same BST: 2

--------------------------------
Process exited after 0.07243 seconds with return value 0
Press any key to continue . . .
```

```
- Warnings: 0
- Output Filename: C:\Users\anang\OneDrive\Desktop\SOURCE CODE.exe
- Output Size: 155.85546875 KiB
- Compilation Time: 0.80s
```

COMPLEXITY ANALYSIS:

TIME COMPLEXITY:

The algorithm has a time complexity of O(n^2). This arises because for each node count from 2 to n, the algorithm considers each node as a potential root and calculates the number of unique left and right subtrees

Best Case: O(1)

The best case occurs when the target value is the root of the BST, requiring constant time to find it.

Average Case: O(log n)

For a balanced BST, the average case time complexity is logarithmic due to the balanced height of the tree. Each level of the tree is traversed in logarithmic time on average.

Worst Case: O(n)

In an unbalanced BST that degenerates into a linked list (e.g., when elements are inserted in sorted order), the search operation may require traversing all n nodes, resulting in linear time complexity.

SPACE COMPLEXITY:

The space complexity of a Binary Search Tree (BST) can vary depending on whether it is implemented recursively or iteratively, and how balanced the tree.

FUTURE SCOPE:

Binary Search Trees (BST) lies in various improvements, specialized variants, and new applications that build upon its basic principles. As computational needs evolve, so do the requirements for efficient data structures. BST lies in both theoretical and practical improvements to handle more complex, dynamic, and large-scale data efficiently. Research in areas like self-balancing, concurrency, parallelism, and integration with modern computing paradigms (IoT, big data, AI) ensures that BSTs will continue to evolve and find new applications across different domains.

CONCLUSION:

The Binary Search Tree (BST) is a fundamental data structure widely used for its simplicity and efficiency in searching, inserting, and deleting data. Its average-case performance for these operations is O(log n), making it an ideal choice for many applications where sorted data and fast lookups are needed. However, in its basic form, a BST can degrade to O(n) time complexity in the worst case, particularly when the tree becomes skewed, which emphasises the importance of using self-balancing variants like AVL trees or Red-Black trees for more consistent performance.

BSTs have numerous real-world applications, including databases, compilers, search engines, and networking systems, where their efficiency in managing hierarchical or sorted data is critical. The structure continues to evolve with new research focusing on parallelism, concurrency, and application-specific optimizations