

# Number of Ways to Reorder Array to Get Same BST

Created by

Harshavardhan Kartheek Reddy A  
(192210042)

CSA0695-

Design and Analysis of  
Algorithms for Open  
Addressing

Faculty :

Dr R Dhanalakshmi

# 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 `nums` so that the constructed BST is identical to that formed from the original array `nums`.  
ted by

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 `nums` such that the BST formed is identical to the original BST formed from `nums`.

Input: `nums = [2,1,3]`

Output: 1

=

# 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.
- In a BST, each node has at most two children referred to as the left and right children.
- The primary strength of BSTs lies in their ability to maintain ordered data, supporting operations like in-order traversal to retrieve elements in a sorted manner.

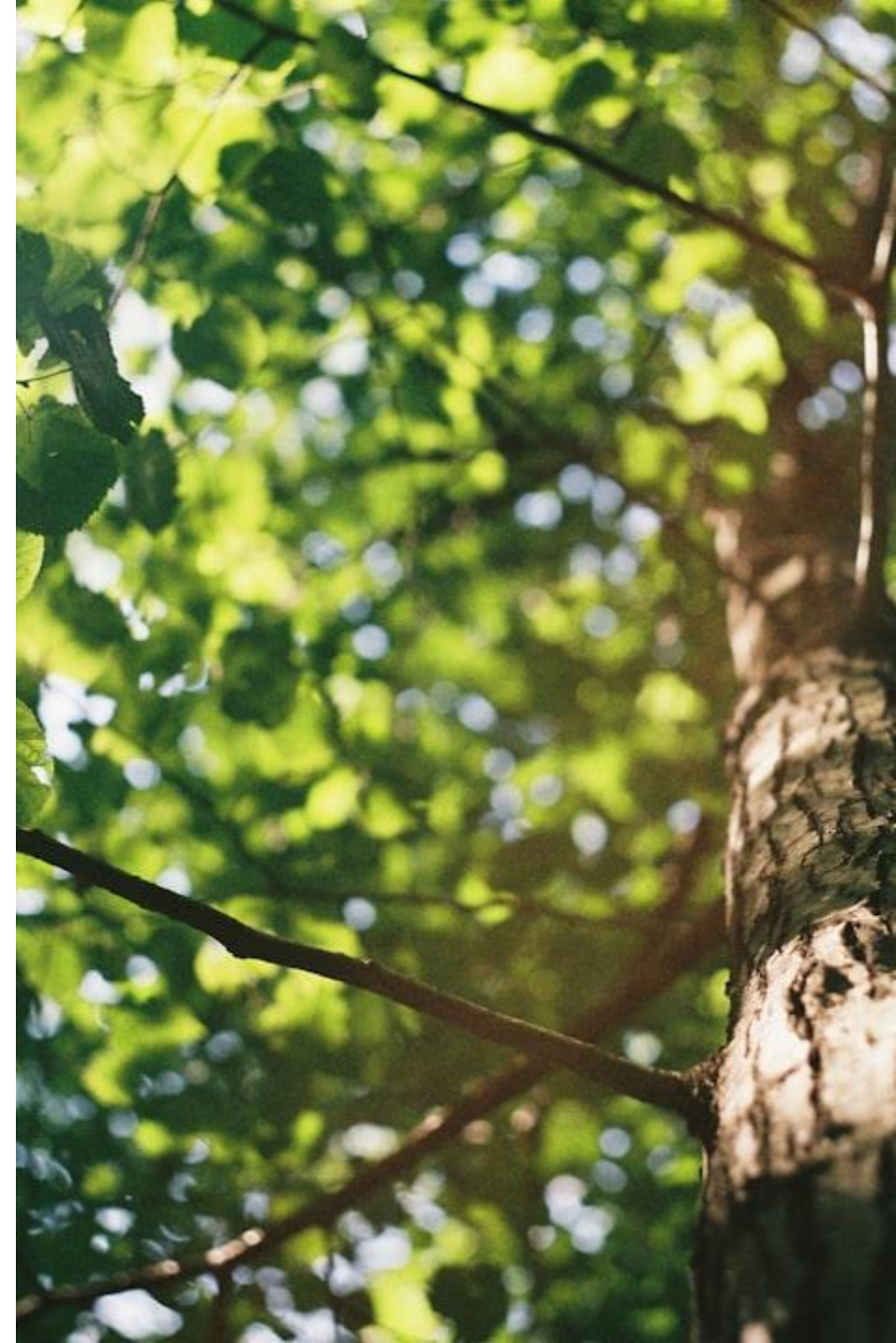




=

# Introduction

- It is a binary tree. It may be empty or not an empty, it satisfies the following properties:-Every element has a key, and are unique.
- The keys in a nonempty left subtree must be smaller, right subtree must be larger.
- The left and right subtrees are also binary search trees.
- It is a fundamental data structure in computer science, commonly utilized for tasks that require dynamic data manipulation, such as searching, inserting, and deleting data.
- 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.



# Examples and Cases



## Unique Values

For an array of unique values, any permutation results in the same BST structure. The total permutations equal the factorial of the number of elements, illustrating complete flexibility in reorderings.

## With Duplicates

When duplicates appear, the number of valid reorderings reduces, as some arrangements will yield identical BST configurations. It's crucial to analyze how these duplicates can affect overall reordering possibilities.

## Visual Representations

Creating visual representations of BSTs for various insertion orders aids in understanding the impact of reordering. Diagrams help clarify how different element arrangements can lead to the same BST structure.

=

# 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.



# Source code

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  typedef struct TreeNode
4  {
5  }
6  int value;
7  struct TreeNode *left, *right;
8  TreeNode;
9  TreeNode* createNode(int value)
10 {
11 }
12 TreeNode* newNode = (TreeNode*)malloc(sizeof(TreeNode));
13 newNode->value = value;
14 newNode->left = newNode->right = NULL;
15 return newNode;
16 TreeNode* insert(TreeNode* root, int value)
17 {
18     if (root == NULL) return createNode(value);
19     if (value < root->value) root->left = insert(root->left, value);
20     else root->right = insert(root->right, value);
21     return root;
22 }
23 int countPermutations(TreeNode* root)
24 {
25 }
26 if (root == NULL) return 1;
27 int leftSubtreeSize = 0, rightSubtreeSize = 0;
28 TreeNode* temp = root->left;
29 while (temp)
30 {
31     leftSubtreeSize++;
32     temp = temp->left ? temp->left : temp->right;
33 }
```

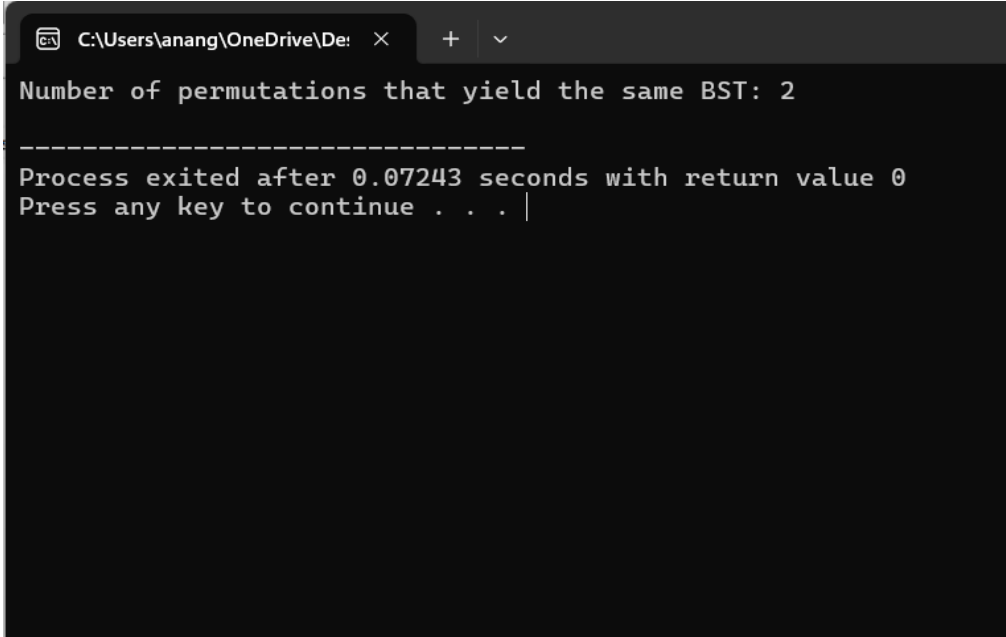
```
6 {
7     rightSubtreeSize++;
8     temp = temp->left ? temp->left : temp->right;
9 }
0 int leftPermutations = countPermutations(root->left);
1 int rightPermutations = countPermutations(root->right);
2 int binomialCoefficient = 1;
3 int n = leftSubtreeSize + rightSubtreeSize;
4 int k = leftSubtreeSize;
5 for (int i = 0; i < k; i++)
6 {
7     binomialCoefficient = binomialCoefficient * (n - i) / (i + 1);
8 }
9 return binomialCoefficient * leftPermutations * rightPermutations;
0 TreeNode* buildBST(int* nums, int size)
1 {
2     TreeNode* root = NULL;
3     for (int i = 0; i < size; i++)
4     {
5         root = insert(root, nums[i]);
6     }
7     return root;
8 }
9 int main()
0 {
1 }
2 int nums[] = {2, 1, 3};
3 int size = sizeof(nums) / sizeof(nums[0]);
4 TreeNode* root = buildBST(nums, size);
5 int result = countPermutations(root);
6 printf("Number of permutations that yield the same BST: %d\n", result);
7 return 0;
```

# Output and Future Scope

## 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.

## Output:

A screenshot of a Windows command prompt window. The title bar shows the file path 'C:\Users\anang\OneDrive\De...' and standard window controls. The command prompt displays the output of a program: 'Number of permutations that yield the same BST: 2', followed by a separator line of dashes, and then 'Process exited after 0.07243 seconds with return value 0' and 'Press any key to continue . . . |'.

```
C:\Users\anang\OneDrive\De: X + v
Number of permutations that yield the same BST: 2
-----
Process exited after 0.07243 seconds with return value 0
Press any key to continue . . . |
```



# 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.
- BST can degrade to  $O(n)$  time complexity in the worst case, particularly when the tree becomes skewed, which emphasizes 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