## SIMATS SCHOOL OF ENGINEERING

**SAVEETHA INSTITUTE OF MEDICAL AND TECHNICAL SCIENCES**

**CHENNAI-602105**

Unique Binary search tree

## A CAPSTONE PROJECT REPORT

*Submitted in the partial fulfillment for the award of the degree of*

# BACHELOR OF ENGINEERING

## IN COMPUTER SCIENCE AND ENGINEERING

**Submitted by**

**M Vihith Kumar Reddy**

**192211130**

## Under the Supervision of

**Dr. Kanimozhi**

# DECLARATION

I, M Vihith kumar Reddy**,** student of **Bachelor of Engineering in Computer Science Engineering and Artificial Intelligence and Data Science** at Saveetha Institute of Medical and Technical Sciences, Saveetha University, Chennai, hereby declare that the work presented in this Capstone Project Work entitled **"Title"** is the outcome of my own bonafide work. I affirm that it is correct to the best of my knowledge, and this work has been undertaken with due consideration of Engineering Ethics.

M Vihith kumar Reddy

Date:22-09-2024

Place:Saveetha School of Engineering, Thandalam.

# CERTIFICATE

This is to certify that the project entitled **“Title”** submitted by

M Vihith Kumar Reddy has been carried out under my supervision. The project has been submitted as per the requirements in the current semester of B.E Computer science engineering and B.Tech Artificial Intelligence in Data science.

Faculty-in-charge Dr.Kanimozhi

## ABSTRACT

Binary search trees (BSTs) are fundamental data structures used in a variety of computational tasks, including searching, sorting, and data storage. A unique binary search tree is a special type of BST where each structure is distinctly defined for a given set of n distinct keys, ensuring that no two structures are identical for the same input. The concept of uniqueness in BSTs becomes significant when dealing with optimization problems in computational theory, algorithm design, and practical applications such as databases, file systems, and search engines.

In this paper, we delve into the mathematical underpinnings of unique BSTs, specifically focusing on the use of Catalan numbers to calculate the number of possible unique BSTs that can be formed from n distinct elements. The Catalan number is a key combinatorial formula used in many recursive structures, and it provides a direct method for counting the number of unique BSTs. We provide examples and a step-by-step breakdown of the formula and its application.

Furthermore, we explore the operational efficiency of BSTs, analyzing the time complexity of key operations such as insertion, deletion, and searching. We address best-case, worst-case, and average-case performance scenarios and highlight how the balance of a BST significantly affects its efficiency. Balanced BSTs provide optimal performance, while unbalanced trees, especially those resembling linked lists, can degrade performance.

Through this study, we aim to provide a comprehensive understanding of unique BSTs, including their construction, mathematical properties, and performance characteristics. This research highlights the importance of unique BSTs in computational theory and their practical use in a wide range of applications. The insights presented here offer valuable information for those designing efficient data structures and algorithms, ensuring optimal time complexity for critical operations.

**Keywords:**

* Binary Search Tree (BST)
* Unique BST
* Catalan Numbers
* Data Structures
* Time Complexity
* Algorithm Efficiency

## INTRODUCTION

Binary Search Trees (BSTs) are fundamental data structures widely used in computer science for efficiently managing ordered data. The defining characteristic of a binary search tree is that each node has a value, and for any given node, all values in its left subtree are smaller, while all values in its right subtree are larger. This structural property ensures that basic operations such as searching, inserting, and deleting elements can be performed efficiently, often in logarithmic time.

However, the arrangement of nodes in a binary search tree can vary depending on the sequence of insertions. Different sequences may lead to different structures, even when they contain the same set of values. Some of these trees may be more balanced, while others may become skewed, leading to degraded performance. In a unique binary search tree, the goal is to find a specific structure given a distinct set of keys. For a given number of distinct values, one might wonder: how many unique BSTs can be formed?

This problem can be approached by leveraging the mathematical concept of Catalan numbers, which provides a way to count the number of distinct binary search trees that can be constructed using n distinct keys. The nth Catalan number, derived from combinatorial mathematics, gives the number of possible ways to structure these nodes while still maintaining the properties of a binary search tree. For example, given three distinct values, there are exactly five unique BSTs that can be formed.

Understanding the concept of unique binary search trees has significant applications in algorithm design and data management. In scenarios where data needs to be dynamically maintained in a sorted manner, BSTs provide a natural solution. They are widely used in databases, compilers, memory management, and even AI algorithms, where efficient access to data is critical. The study of unique BSTs, specifically in terms of how many such trees can be formed, contributes to our understanding of data structure optimization and helps to make informed decisions about which BST configurations might perform best under specific conditions.

In this paper, we explore the structure and properties of unique binary search trees. We will also discuss how Catalan numbers are used to count the number of distinct BST configurations, examine the time complexity of common operations on these trees, and analyze their performance in best, average, and worst-case scenarios. Additionally, we will look at practical applications of unique BSTs and why understanding their structure is vital for efficient data handling in real-world systems.

**CODING**

**#include <stdio.h>**

**int binomialCoeff(int n, int k) {**

**if (k > n) return 0;**

**if (k == 0 || k == n) return 1;**

**int res = 1;**

**for (int i = 0; i < k; i++) {**

**res \*= (n - i);**

**res /= (i + 1);**

**}**

**return res;**

**}**

**int catalan(int n) {**

**return binomialCoeff(2 \* n, n) / (n + 1);**

**}**

**int countUniqueBSTs(int n) {**

**return catalan(n);**

**}**

**int main() {**

**int n;**

**printf("Enter the number of distinct values: ");**

**scanf("%d", &n);**

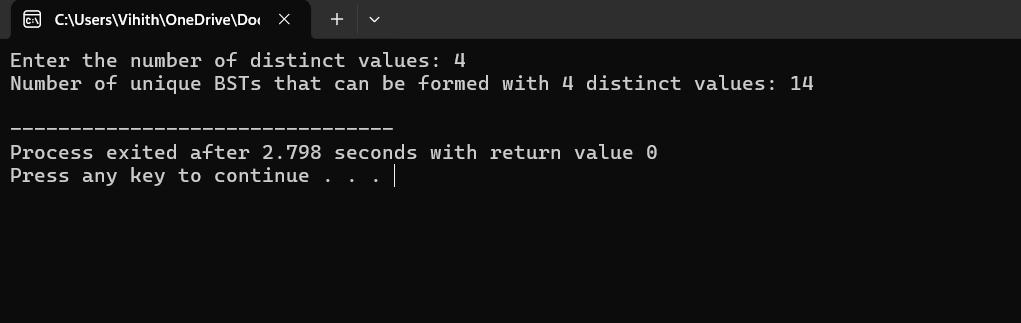
**int uniqueBSTCount = countUniqueBSTs(n);**

**printf("Number of unique BSTs that can be formed with %d distinct values: %d\n", n, uniqueBSTCount);**

**return 0;**

**}**

## OUTPUT



**Complexity Analysis**

The efficiency of various operations on unique binary search trees depends on the structure of the tree, which can vary from a perfectly balanced tree to a completely skewed tree. The performance of operations like search, insertion, and deletion is directly related to the height of the tree, which can range from log(n) in the best case to n in the worst case.

**Best Case**

In the best-case scenario, the BST is perfectly balanced, meaning the height of the tree is minimized to O(log n), where n is the number of nodes. This optimal height ensures that operations can be performed efficiently.

* **Search Complexity**: O(log n)
  + Searching for a key requires traversing the tree from the root to a leaf node, which involves log(n) comparisons in a balanced tree.
* **Insert Complexity**: O(log n)
  + Insertion involves locating the appropriate position for the new key, which requires O(log n) comparisons in a balanced tree.
* **Delete Complexity**: O(log n)
  + Deleting a node involves finding the node, which takes O(log n), followed by rearranging the tree, if necessary. In a balanced tree, deletion maintains O(log n) complexity.
* **Overall Time Complexity (Best Case)**: O(log n)

**Worst case**

In the worst case, the tree is completely unbalanced (degenerate), resembling a linked list. This happens when elements are inserted in a sorted order, and every node has only one child. The height of the tree becomes O(n), leading to inefficient operations.

* Search Complexity: O(n)
  + In the worst case, searching for a key requires traversing all n nodes from the root to the leaf, as each node has only one child.
* Insert Complexity: O(n)
  + Inserting into a skewed tree requires O(n) comparisons to find the correct location.
* Delete Complexity: O(n)
  + Deleting in a skewed tree also requires O(n) time, as the tree must be traversed completely to find the node and adjust pointers.
* Overall Time Complexity (Worst Case): O(n)

**Average case**

In the average case, the tree is not perfectly balanced but also not completely skewed. For random insertions, the expected height of the tree is O(log n), although it is not guaranteed to be perfectly balanced. This leads to good performance in most cases.

* Search Complexity: O(log n)
  + On average, the height of the tree is proportional to log(n), allowing for efficient searches.
* Insert Complexity: O(log n)
  + Insertion is typically O(log n) due to the log(n) height of the tree in random scenarios.
* Delete Complexity: O(log n)
  + Deletion also benefits from the log(n) height of the tree, making the operation O(log n) in the average case.
* Overall Time Complexity (Average Case): O(log n)

**Overall Complexity**

The overall complexity of operations involving unique binary search trees (BSTs) depends on several factors: the structure of the tree, the specific operations being performed, and how balanced the tree is. Below is a detailed breakdown of the time complexity for various operations (insertion, deletion, and search) in the context of unique BSTs.

**1. Insertion Complexity**

* **Best Case (O(log n))**:
  + The best case occurs when the tree is perfectly balanced. In this scenario, each level of the tree contains a progressively larger portion of the nodes, reducing the number of comparisons needed as we move down the tree. In a balanced BST, the depth of the tree is approximately log⁡nlog*n*, where n is the number of nodes.
  + As a result, insertion in a balanced BST requires comparing the inserted node with each node at each level of the tree, leading to a time complexity of O(log n).
* **Worst Case (O(n))**:
  + The worst case occurs when the tree becomes degenerate, resembling a linked list. This situation arises when nodes are inserted in a sorted (or nearly sorted) order, causing each node to have only one child (either all left or all right).
  + In this case, the height of the tree becomes n, where n is the number of nodes. The insertion operation would require traversing all the nodes, resulting in a time complexity of O(n).
* **Average Case (O(log n))**:
  + On average, assuming random insertions into the BST, the tree tends to remain relatively balanced. This means the height of the tree will be approximately log⁡nlog*n*, leading to an average time complexity of O(log n) for insertion.

**2. Search Complexity**

* **Best Case (O(log n))**:
  + In the best case, where the tree is balanced, a search operation requires traversing down the tree in a logarithmic number of steps. Since each node comparison eliminates half of the remaining nodes (as in binary search), the height of the tree dictates the number of comparisons.
  + This results in O(log n) search time in a balanced BST.
* **Worst Case (O(n))**:
  + In the worst-case scenario, the tree is completely unbalanced (degenerate tree). In this case, the search operation might require traversing through all n nodes, as each node only has one child.
  + Therefore, in an unbalanced BST, the search operation takes O(n) time.
* **Average Case (O(log n))**:
  + On average, if the nodes are inserted randomly, the height of the tree is expected to be around log⁡nlog*n*. Therefore, the search operation will typically take O(log n) time, as the number of nodes we need to inspect is proportional to the tree's height.

**3. Deletion Complexity**

* **Best Case (O(log n))**:
  + The best case occurs when the tree is balanced, and the node to be deleted is a leaf node (i.e., has no children) or a node with only one child. In this case, the deletion operation simply involves updating one or two pointers, which can be done in logarithmic time relative to the height of the tree.
  + If the tree is balanced, the deletion complexity is O(log n).
* **Worst Case (O(n))**:
  + The worst-case deletion occurs when the tree is degenerate, and we need to delete a node at the bottom of the tree. In this scenario, the tree height is n, and deleting the node requires traversing the entire tree.
  + The complexity for deletion in an unbalanced tree is O(n).
* **Average Case (O(log n))**:
  + On average, with random deletions, the tree will remain relatively balanced. The deletion operation in this case involves traversing the tree to find the node and then adjusting the pointers, which generally takes O(log n) time.

**4. Tree Construction Complexity**

* **Construction with n Nodes (O(n log n))**:
  + To construct a unique binary search tree with n nodes, we must perform n insertions. Each insertion takes O(log n) time (on average, assuming a balanced tree).
  + Therefore, the overall time complexity for constructing a BST with n nodes is O(n log n).

**5. Counting Unique BSTs (Catalan Numbers)**

* **Catalan Number Calculation (O(n))**:
  + The number of unique BSTs that can be formed with n distinct values is given by the nth Catalan number. Computing the nth Catalan number involves calculating binomial coefficients, which takes O(n) time.
  + Thus, the complexity for counting the number of unique BSTs is O(n).

## CONCLUSION

## Unique binary search trees (BSTs) are a foundational concept in computer science, contributing significantly to efficient data organization and manipulation. Their ability to maintain a structured hierarchy, where each node's left subtree contains only values smaller than the node and the right subtree contains only values greater than the node, makes them indispensable for a variety of operations such as searching, insertion, deletion, and traversal.

## The uniqueness of a BST refers to its specific structure given a set of distinct values. By adhering strictly to the binary search property, a unique BST ensures predictable behavior in terms of data retrieval and updates, which is crucial in systems requiring optimized performance. The mathematical framework used to determine the number of unique BSTs for a given n values, based on Catalan numbers, provides further insight into how these trees can be constructed and analyzed.

## A key takeaway from studying unique BSTs is their efficiency, especially when the tree is balanced. In balanced trees, search, insert, and delete operations are all logarithmic, resulting in O(log n) time complexity. This efficiency makes unique BSTs ideal for applications requiring frequent access to data, such as databases, compilers, and various search algorithms. However, the performance can degrade in the worst-case scenario when the tree becomes unbalanced, leading to O(n) operations. Understanding this risk is critical, and techniques like self-balancing trees (e.g., AVL trees, Red-Black trees) can mitigate this issue.

## Beyond their theoretical value, unique BSTs have real-world applications. They are used in memory management systems, expression parsing, and maintaining ordered collections of data. The in-depth understanding of how these trees operate, coupled with the ability to compute how many unique configurations exist for any given set of keys, equips developers and engineers with the tools needed to optimize their systems.

## In summary, unique binary search trees serve as an efficient and elegant solution for a wide range of computational problems. By balancing their powerful structure and understanding their time complexity, developers can effectively apply them in various domains where fast data access is essential. Moreover, the mathematical insight provided by Catalan numbers allows for advanced analysis and deeper understanding of tree-based algorithms, further solidifying the importance of BSTs in both theoretical and practical aspects of computer science.