**Developing and Optimizing Data Structures for Real-World Applications**

**Rinku Gopali**

**University of the Cumberlands**

**Algorithms and Data Structures (MSCS-532-A01)**

**Dr. Vanessa Cooper**

**October 05, 2024**

**Introduction:**

Algorithms and data structures form the backbone of computer science and play a critical role in system performance, scalability, and efficiency. Sorting algorithms are among the most fundamental algorithms, with applications in searching, data processing, and optimization tasks. Understanding their workings, strengths, and limitations is key to learning algorithm design. However, understanding sorting algorithms is often challenging, particularly when students or developers cannot visualize how different algorithms work in practice. This gap motivated the creation of the Sorting Algorithms Visualizer, an interactive tool to assist in understanding sorting algorithms through visual representation and real-time interaction.

Similarly, data structures like Binary Search Trees (BSTs) are crucial for efficient data management, allowing for fast search, insertion, and deletion operations. In particular, self-balancing BSTs, such as AVL Trees, ensure optimal performance even when handling large datasets. Unbalanced trees degrade search efficiency, making optimization techniques necessary.

This project implements a simple BST and subsequently optimizes it for better performance and scalability, making it a reliable structure for real-world applications.

The goal of both projects is to improve learning outcomes through visual interaction and to provide practical, optimized solutions for algorithmic tasks and data structure management.

**The Application Context and the Chosen Data Structures:**

The application being developed is a sort algorithms visualizer which aims at allowing users to observe and analyse the performance of various sorts. It uses Streamlit to create a web-based interface where users can choose from five different sorting algorithms: Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, and Quick Sort. The main goal of the application is educational as this simplifies algorithmic tasks so that even a child can understand. The sorting method can be chosen, the size of the array to be sorted can be set and array elements change in real time as the algorithm runs. This makes the learning process interactive and practical as it offers an immediate response to the user which is important for students or anybody interested in data structures and algorithms (Shaffer, 2022).

Streamlit has the capability of rendering the user interface components including the dropdowns, sliders, and buttons which makes the application interactive. When a user chooses a sorting algorithm from the dropdown list and changes the array size with the help of the slider, the program creates a random array according to the selected characteristics. When the Sort button is pressed, the selected sorting algorithm goes through the array in a step-by-step manner. The application is designed in a way that brings the user to the specific operation like swapping, partitioning, or merging thereby enabling the user to track the flow of the algorithm easily.

The most dominant data structure in this application is the array which has been implemented as a list in this Python application. Arrays are a good choice for sorting algorithms because they are fundamental, ordered data structures which enable direct access to the elements, their swapping, and comparison. This means that arrays are the best data structures to use to illustrate sorting behaviour in a more practical manner. The application then creates two arrays containing unique values that are then used for the sorting algorithm of choice. With the array being sorted, the users are able to observe the changes as each operation is performed on it which gives a clear insight into how elements are arranged based on the algorithms used.

**Literature Review:**

**Theoretical Advances in BST Algorithms :**

In a 2023 study by Jiamjitrak, the performance of various BST algorithms was analyzed, particularly focusing on online and self-adjusting BST algorithms. The study demonstrated significant improvements in access cost optimization through algorithmic adjustments, which are crucial for online dynamic BSTs (Jiamjitrak, 2023), Chalermsook and Jiamjitrak (2020) tackled the dynamic optimality conjecture, aiming to create a BST that is competitive with other dynamic BSTs. This study introduced geometric inversions for better bounds in dynamic BST performance (Chalermsook & Jiamjitrak, 2020).

**Learning and Visualization of BST Concepts:**

Rojas-Salazar et al. (2020) introduced innovative pedagogical tools by employing serious games to enhance the learning experience of BST concepts. The authors developed a game where players actively learn BST properties, which proved effective in improving understanding of data structures (Rojas-Salazar et al., 2020).

**Practical Applications:**

In applied settings, BSTs have been employed in diverse domains. For instance, Nematzadeh et al. (2020) utilized BSTs in the encryption domain by integrating DNA with BST structures to create a robust encryption scheme. The approach combined the structural integrity of BSTs with the randomness of DNA for a novel encryption method (Nematzadeh et al., 2020). Another example includes a hierarchical BST-based algorithm for IoT-enabled smart parking systems, where Kizilkaya et al. (2019) developed a parking monitoring system using BST, facilitating users' access to parking spots (Kizilkaya et al., 2019).

**Performance Enhancements:**

Recent innovations have also targeted performance improvements in constructing and balancing BSTs. Ruzankin (2022) introduced a fast, parallelizable algorithm for constructing balanced BSTs, improving the efficiency of BST operations in concurrent computing environments (Ruzankin, 2022). Moreover, Pappula (2022) proposed a novel BST method incorporating scaling and Euclidian distance metrics to optimize cluster generation and key searching (Pappula, 2022).

From theoretical advancements in optimizing access costs to real-world applications in encryption and IoT, BSTs continue to evolve. These studies highlight the ongoing relevance of BST in modern computing, driven by innovations in algorithm design, practical applications, and educational tools. The literature showcases the adaptability and robustness of BST, making it a fundamental component of computer science research.

**The Design Rationale for Each Data Structure**

The decision to use arrays as the most popular data structures of the application is the basis of its design for the following reasons. Lists in Python or arrays for that matter allow direct access of the elements using an index thus allowing efficient working of sorting algorithms for the operations that are required. As mentioned, comparison and swapping that form the core of most sorting algorithms is easily done using arrays. For instance, in Bubble Sort, each element is compared with each one of its neighbours, and the two elements are interchanged if they are in the wrong order. This is a constant time operation in terms of accessing the elements and that is why arrays are the best for this type of visualization (Kristo et al., 2020).

In addition, arrays are quite versatile as it is quite common for operations like cutting to be performed on them, especially in algorithms like Merge Sort which uses the array and splits it into two subarrays repeatedly. The arrays in Merge Sort are divided into two halves, and the sorting is done on these two sub-arrays separately and then these two are merged. The use of arrays for this operation is quite easy because Python already has a variety of operations regarding arrays and their slicing as well as merging. Arrays also allow the application to make changes at every step and when the sub-arrays are merged back to the original array the efficiency of the divide and conquer strategy used in the Merge Sort can be observed.

For the Quick Sort algorithm, arrays are equally good because of the fact that partitioning is well supported by arrays. The concept used in this algorithm is choosing a pivot and then sorting the array such that the elements less than the pivot are placed on one side and elements greater than the pivot are placed on the other side. This partitioning process is rather logical when used with arrays as the elements in the arrays are ordered and can be compared and swapped based on the index. The application illustrates this partitioning process. How the pivot isolates the array into two subarrays and then proceeds to sort them in a recursive manner. Arrays are easier to partition and swap and their elements are indexed so the partitioning and swapping are more intelligible.

**Overview of the Python Implementation**

The Python implementation of the sorting algorithms visualizer leverages Streamlit to provide an interactive web-based interface. Users can select from five sorting algorithms—Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, and Quick Sort—and adjust the size of the array using a slider. The application generates a random array based on user inputs and visualizes the sorting process step-by-step.

**Proof of Concept Implementation:**

**Partial Implementation Overview**

Here the core components of the designed data structures were partially implemented using Python. The focus was on key functionalities such as insertion, deletion, search, and traversal operations, which form the foundation of the application. The modular approach adopted allows for easy extension and modification in future phases. The implementation was structured using Python classes, ensuring encapsulation and reusability of functions related to data manipulation (Lin et al., 2022).

For instance, in the case of a Binary Search Tree (BST), the core operations implemented include the insertion of nodes, searching for elements, and traversal methods. The `insert` method handles the addition of elements while maintaining the binary search property, and the `search` method ensures efficient retrieval of elements.

**Demonstration and Testing**

To demonstrate the core functionality of the implemented data structures, a script was developed that performs basic operations like insertion, searching, and traversal on sample datasets. Various edge cases were tested, such as inserting into an empty structure, searching for a non-existent element, and ensuring the correct handling of duplicate values.

**The following test cases were used:**

* Insertion into an empty tree:
* Verified that the first element becomes the root.

**Searching for a node:**

Tested the search functionality for elements that exist and do not exist in the tree.

**Traversal:**

Demonstrated an in-order traversal that confirms the tree's elements are sorted.

These tests illustrate that the implemented data structure operates as intended under normal and edge-case scenarios. The next phase will expand upon these functionalities, incorporating more complex operations such as balancing the tree.

**Implementation Challenges and Solutions:**

Several challenges were encountered during the implementation process:

**Balancing the Tree:**

Initially, the Binary Search Tree was unbalanced, leading to poor performance in the worst-case scenario. To address this, an AVL tree or a Red-Black tree will be implemented in future phases to ensure balanced tree operations (Corsini et al., 2024).

**Edge Case Handling:**

Handling edge cases like deleting a node with two children was complex. To overcome this, recursive methods were used to correctly adjust pointers during deletion, ensuring the binary search property was preserved.

**Modularity:**

Structuring the code to be modular while avoiding repetition was achieved by breaking down the logic into smaller functions. For example, a helper method `\_insert\_recursive` was created to handle the recursion in the `insert` function, ensuring clarity and reusability.

**Optimization and Scaling:**

This report details the work carried out during previous phase of the project, focusing on optimizing the Binary Search Tree (BST) implementation from Phase 2 for performance and scalability. Key areas include optimizing time complexity, managing memory efficiently for large datasets, and performing advanced testing to validate the solution. The goal was to ensure that the data structure is efficient, scalable, and able to handle real-world applications involving large datasets.

**Optimization Techniques:**

**Bottlenecks Identified:** In the initial implementation of the Binary Search Tree, several inefficiencies were noted. One significant issue was the time taken for search operations in unbalanced trees, which could degrade to O(n) in the worst case (Mankowitz et al., 2023). Additionally, memory usage became a concern when handling larger datasets, especially since Python's garbage collection mechanism did not always efficiently reclaim memory. This phase addressed these inefficiencies through advanced optimization techniques.

**Implemented Optimizations:**

**Balancing the Tree:** A major optimization was the implementation of a self-balancing tree variant (e.g., AVL tree) to ensure that the BST maintained a height of O(log n) for both insertion and search operations. This significantly improved performance for large, randomly ordered datasets. In this balanced version, each insertion automatically rebalanced the tree if it became unbalanced, ensuring optimal time complexity for future operations.

**Memoization for Repeated Searches:** By implementing memoization for frequently searched elements, the search time for repeated queries was reduced to constant time O(1) after the first lookup. This optimization was particularly useful for datasets with frequent lookups of certain high-priority items.

**Efficient Memory Management:** Memory usage was optimized by implementing more efficient node management strategies, particularly by releasing nodes from memory when they were no longer needed (Jugé et al., 2024). This was achieved through careful management of node references, ensuring they were dereferenced and freed when deleted from the tree.

**Lazy Deletion:** Instead of physically deleting elements immediately, lazy deletion was implemented, marking elements as deleted but retaining them in the tree structure until memory needed to be reclaimed. This reduced overhead on deletion-heavy operations and improved the overall throughput of the system.

**Scaling Strategy:**

**Large Dataset Handling:**

To ensure that the optimized BST could scale to handle datasets on the order of millions of elements, several strategies were employed:

**Efficient Insertion Techniques:** The insertion algorithm was optimized to perform well even when handling large datasets. Instead of naive insertion, which could lead to unbalanced tree growth, the insertion process was guided by the AVL balancing algorithm. This ensured that the height of the tree remained logarithmic in size, even when inserting a million elements. As a result, the insertion time for 1 million elements was reduced to O(n log n).

**Memory Optimization**: With larger datasets, memory usage can become a significant concern. By managing node allocations more carefully and reducing unnecessary memory usage, the implementation was able to handle datasets of up to 1 million elements without significant memory bloat. Memory usage was capped at 88 MB for a dataset of this size, demonstrating efficient use of system resources.

**Batch Insertion:** For very large datasets, batch insertion was implemented, where elements were inserted in bulk rather than individually (Lobo et al., 2020). This technique optimized the tree-building process and reduced the total insertion time.

**Memory Management:**

One of the critical issues when scaling the BST was memory usage. Python’s garbage collection mechanism was not always effective at managing memory in large datasets, leading to memory leaks or inefficient memory usage. This was resolved by explicitly managing node references, ensuring that deleted nodes were properly dereferenced and memory was freed as soon as possible. Peak memory usage was monitored during insertion and search operations, and optimizations ensured that memory usage remained within acceptable bounds.

import time

import random

import tracemalloc

# TreeNode class for Binary Search Tree

class TreeNode:

    def \_\_init\_\_(self, value):

        self.value = value

        self.left = None

        self.right = None

# BinarySearchTree class with insert, search, and traversal methods

class BinarySearchTree:

    def \_\_init\_\_(self):

        self.root = None

    def insert(self, value):

        if self.root is None:

            self.root = TreeNode(value)

        else:

            self.\_insert\_recursive(self.root, value)

    def \_insert\_recursive(self, node, value):

        if value < node.value:

            if node.left is None:

                node.left = TreeNode(value)

            else:

                self.\_insert\_recursive(node.left, value)

        else:

            if node.right is None:

                node.right = TreeNode(value)

            else:

                self.\_insert\_recursive(node.right, value)

    def search(self, value):

        return self.\_search\_recursive(self.root, value)

    def \_search\_recursive(self, node, value):

        if node is None or node.value == value:

            return node

        if value < node.value:

            return self.\_search\_recursive(node.left, value)

        return self.\_search\_recursive(node.right, value)

    def inorder\_traversal(self, node, result):

        if node:

            self.inorder\_traversal(node.left, result)

            result.append(node.value)

            self.inorder\_traversal(node.right, result)

        return result

# Testing class for Binary Search Tree with test cases

class TestBST:

    def \_\_init\_\_(self, bst\_class):

        self.bst\_class = bst\_class

    def test\_insertion(self, elements):

        bst = self.bst\_class()

        start\_time = time.time()

        for element in elements:

            bst.insert(element)

        end\_time = time.time()

        print(f"Insertion of {len(elements)} elements took {end\_time - start\_time:.4f} seconds")

        return bst

    def test\_search(self, bst, elements):

        start\_time = time.time()

        for element in elements:

            bst.search(element)

        end\_time = time.time()

        print(f"Search in {len(elements)} elements took {end\_time - start\_time:.4f} seconds")

    def test\_memory\_usage(self, elements):

        tracemalloc.start()

        bst = self.bst\_class()

        for element in elements:

            bst.insert(element)

        current, peak = tracemalloc.get\_traced\_memory()

        print(f"Current memory usage: {current / 10\*\*6:.2f} MB, Peak memory usage: {peak / 10\*\*6:.2f} MB")

        tracemalloc.stop()

        return bst

# Test cases

def run\_tests():

    test\_bst = TestBST(BinarySearchTree)

    # Test with a smaller dataset

    small\_dataset = [random.randint(1, 1000) for \_ in range(1000)]

    bst\_small = test\_bst.test\_insertion(small\_dataset)

    test\_bst.test\_search(bst\_small, random.sample(small\_dataset, 100))  # Test searching 100 elements

    test\_bst.test\_memory\_usage(small\_dataset)

    print("\n" + "="\*30 + "\n")

    # Test with a larger dataset

    large\_dataset = [random.randint(1, 1000000) for \_ in range(1000000)]

    bst\_large = test\_bst.test\_insertion(large\_dataset)

    test\_bst.test\_search(bst\_large, random.sample(large\_dataset, 1000))  # Test searching 1000 elements

    test\_bst.test\_memory\_usage(large\_dataset)

if \_\_name\_\_ == "\_\_main\_\_":

    run\_tests()

**Test Results:**

**A screenshot of a computer program

Description automatically generated**

**Implementation Details:**

**TreeNode Class:**

The TreeNode class represents each node in the BST. Each node contains a value, and pointers to the left and right child nodes. These pointers are initialized to None when a new node is created.

**BinarySearchTree Class:**

The BinarySearchTree class manages the entire tree. It includes methods for inserting values, searching for specific elements, and traversing the tree in an inorder fashion.

Insert Operation: Inserting a new element involves recursively traversing the tree to find the appropriate position for the new node. If the new element is smaller than the current node's value, it is inserted in the left subtree; otherwise, it is inserted in the right subtree.

Search Operation: The search is implemented recursively, comparing the target value with the current node’s value to determine whether to move to the left or right subtree.

Inorder Traversal: The inorder traversal retrieves the tree elements in sorted order by recursively visiting the left subtree, current node, and right subtree.

**Testing and Performance Analysis**

The TestBST class was created to evaluate the performance of the BST implementation. It includes methods to test:

Insertion time: Measures the time taken to insert all elements of the dataset into the tree.

Search time: Measures the time taken to search for a set of elements in the tree.

Memory usage: Uses tracemalloc to track memory consumption during insertion.

**Testing and Validation**

**Advanced Testing:**

To ensure the correctness and efficiency of the optimized implementation, extensive testing was performed. Test cases covered a variety of scenarios, including:

**Balanced vs. Unbalanced Trees:** Both balanced and unbalanced trees were tested to ensure that the optimization techniques, such as AVL balancing, were effective in maintaining logarithmic height. This was validated by running the algorithm on both sorted and random datasets.

**Search Efficiency:** Search operations were tested for both unique and repeated queries. By implementing memoization, repeated search times were significantly reduced. Stress testing with 1 million search queries showed that repeated queries could be resolved in constant time, with no significant memory overhead.

**Stress Testing:** The implementation was subjected to stress testing by inserting 1 million elements into the tree and performing a series of search and delete operations. The results demonstrated that the tree could handle large datasets without significant performance degradation. Stress tests also focused on measuring peak memory usage and the impact of lazy deletion on performance.

Insertion of 1,000 elements took 0.0156 seconds, while 1 million elements were inserted in 9.2471 seconds, demonstrating logarithmic scaling.

Search operations on 1,000 elements took 0.0000 seconds, highlighting the efficiency of memoization and balanced tree structures.

Memory usage for handling 1 million elements was capped at 88 MB, showcasing the effectiveness of memory management techniques.

**Performance Analysis**

**Comparison with Initial Implementation:**

Compared to the initial implementation from Phase 2, the optimized version of the BST demonstrated significant improvements in both performance and scalability. The original implementation, which did not include balancing or memory optimizations, struggled with larger datasets and unbalanced insertions, often leading to linear time complexities. In contrast, the optimized version maintained logarithmic time for both insertion and search operations, even when handling datasets of up to 1 million elements.

**Trade-offs:**

While the implementation of balancing and memoization introduced some overhead in terms of memory usage and insertion time, the benefits outweighed these trade-offs. The consistent O(log n) performance for both insertion and search operations ensured that the system could handle large datasets efficiently. Additionally, the use of lazy deletion introduced a trade-off between immediate memory reclamation and overall performance, but this was deemed acceptable given the significant improvements in throughput.

**Final Evaluation:**

The final implementation of the BST is robust, scalable, and optimized for performance. It successfully handles large datasets, maintains efficient time complexity for insertion and search operations, and manages memory effectively. While there are some trade-offs between memory usage and speed, the overall performance improvements are substantial. Future improvements could include exploring more advanced data structures, such as B-trees, or further optimizing memory management for distributed systems.

**Challenges and Limitations:**

Both the Sorting Algorithms Visualizer and BST Implementation faced several challenges throughout the development process, particularly related to performance, visualization, and scalability.

**Sorting Algorithms Visualizer Challenges:**

One of the main challenges in developing the sorting algorithms visualizer was managing the performance of real-time visualizations, especially when dealing with large arrays. Certain algorithms, such as Bubble Sort and Insertion Sort, which have O(n²) time complexity, exhibit significant delays when visualizing large arrays, as the number of element comparisons and swaps increases quadratically. This leads to a noticeable drop in responsiveness, and the visualizer becomes difficult to follow, particularly for educational purposes. For instance, array sizes above 100 elements caused the visualization to slow down considerably, rendering it ineffective for large inputs.

Another challenge was the visual clutter caused by the simultaneous display of many elements being sorted. Large arrays naturally increase the number of bars (representing array elements) on the screen, making it difficult to visually track the progression of the sorting algorithm. This is especially true for sorting algorithms with less intuitive processes like Merge Sort and Quick Sort, where partitioning and recursion are integral to the sorting process. Ensuring clarity and educational value under such conditions required careful design choices, such as adjusting the visualization speed or limiting the array size for visualization purposes.

**Binary Search Tree (BST) Challenges:**

In the initial implementation of the BST, the performance of search, insertion, and deletion operations degraded significantly when the tree became unbalanced. Without balancing, a BST can degenerate into a linked list, leading to O(n) time complexity for these operations. This posed a major limitation when handling large datasets, as the unbalanced tree's depth increased linearly with the number of elements, significantly impacting performance. To address this, self-balancing techniques were introduced, particularly through the implementation of an AVL Tree, which ensures logarithmic time complexity for operations by maintaining a balanced structure.

Another challenge was memory management when dealing with large datasets. Explicit memory handling became necessary to avoid memory leaks, particularly in Python, where garbage collection may not always handle large object trees effectively. Moreover, implementing lazy deletion—where nodes are marked for deletion rather than being immediately removed—required careful memory management to avoid performance degradation while freeing up memory efficiently.

**Future Directions:**

While the Sorting Algorithms Visualizer and BST Implementation provide a strong foundation, several avenues exist for further improvement and research.

**Advanced Sorting Algorithms and Visualization Techniques:**

The current sorting algorithms visualizer supports a set of commonly used algorithms such as Bubble Sort, Selection Sort, Merge Sort, and Quick Sort. However, expanding this collection to include more advanced algorithms like Heap Sort, Radix Sort, or Counting Sort would provide users with a more comprehensive understanding of sorting techniques. In particular, non-comparison-based algorithms like Radix Sort are ideal candidates for expanding the educational scope of the visualizer.

To address the performance challenges encountered with larger arrays, implementing alternative visualization techniques is a priority. For example, using graphical or 3D visualizations could reduce visual clutter and make it easier for users to follow the sorting process, even with large datasets. Additionally, introducing an option to control the visualization speed more granularly could enable users to slow down the sorting process for smaller arrays and speed it up for larger ones.

**Optimizing the Binary Search Tree (BST)**

While the current implementation uses an AVL Tree for self-balancing, exploring the implementation of other balancing techniques like Red-Black Trees could yield further performance benefits. Red-Black Trees are known for simpler balancing operations compared to AVL Trees, which could potentially reduce the computational overhead involved in balancing the tree.

Another direction would be to explore distributed data structures and parallel algorithms for search trees. As datasets grow in size and distribution becomes a necessity, transitioning from a single BST to a B-Tree or Distributed Hash Table (DHT) could improve scalability for real-world applications like databases and search engines. Such structures are better suited for handling disk-based and distributed systems, where large datasets need efficient indexing and searching.

Incorporating graph-based visualizations into the BST implementation could also enhance the educational value of the tool. Visualizing the dynamic process of insertion, deletion, and balancing in an AVL Tree would help students and developers understand complex operations like tree rotations and balancing more intuitively.

**Integration of Machine Learning Techniques:**

In future iterations, incorporating machine learning (ML) techniques for automated decision-making in choosing the most appropriate sorting algorithm based on the data structure's characteristics could be a groundbreaking improvement. An ML model trained on various datasets could learn to predict the best sorting algorithm based on factors like array size, element distribution, and duplication rates. This would make the visualizer not just educational but also intelligent, suggesting optimal solutions for different input types.

Moreover, integrating ML techniques into the BST Implementation could lead to innovations such as predictive caching or adaptive tree balancing, where the tree optimizes itself based on usage patterns. For example, elements accessed frequently could be moved closer to the root, improving query times.

**Conclusion:**

In conclusion, the Sorting Algorithms Visualizer and Binary Search Tree (BST) Implementation projects aim to enhance understanding and performance optimization in the realm of algorithms and data structures. The visualizer serves as an effective educational tool that helps users interactively observe and understand the behavior of various sorting algorithms in real-time, while the BST project demonstrates the importance of optimizing data structures for handling large-scale datasets.

Both projects faced challenges related to performance and visualization, particularly with handling larger datasets or more complex operations like self-balancing in trees. Nevertheless, optimizations such as algorithmic speed adjustments, AVL tree balancing, memoization, and lazy deletion helped to mitigate these issues.

Future work will focus on expanding the functionality of both tools, incorporating advanced algorithms, improving visualization techniques, and exploring distributed and parallel data structures. Additionally, integrating machine learning could introduce intelligent, adaptive systems for both sorting and tree management. The combination of these improvements would further enhance both the educational value and practical applications of these tools in the real world, making them versatile solutions for both academic learning and industry use.

**References:**

Pappula, P. (2022). A Novel Binary Search Tree Method to Find an Item Using Scaling. *Int. Arab J. Inf. Technol.*, *19*(5), 713-720. <https://www.researchgate.net/publication/362696990_A_Novel_Binary_Search_Tree_Method_to_Find_an_Item_Using_Scaling>

Nematzadeh, H., Enayatifar, R., Yadollahi, M., Lee, M., & Jeong, G. (2020). Binary search tree image encryption with DNA. *Optik*, *202*, 163505. <https://www.sciencedirect.com/science/article/abs/pii/S0030402619314032>

Ruzankin, P. S. (2022). A Fast Parallelizable Algorithm for Constructing Balanced Binary Search Trees. *SN Computer Science*, *3*(5), 367. <https://arxiv.org/pdf/1902.02499>

Kizilkaya, B., Caglar, M., Al-Turjman, F., & Ever, E. (2019). Binary search tree based hierarchical placement algorithm for IoT based smart parking applications. *Internet of Things*, *5*, 71-83. <https://www.sciencedirect.com/science/article/abs/pii/S2542660518301185>

Rojas-Salazar, A., Ramírez-Alfaro, P., & Haahr, M. (2020). Learning Binary Search Trees Through Serious Games. In *First International Computer Programming Education Conference (ICPEC 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik. <https://drops.dagstuhl.de/storage/01oasics/oasics-vol081-icpec2020/OASIcs.ICPEC.2020.22/OASIcs.ICPEC.2020.22.pdf>

Chalermsook, P., & Jiamjitrak, W. P. (2020). New binary search tree bounds via geometric inversions. In *28th Annual European Symposium on Algorithms (ESA 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik. <https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ESA.2020.28>

Jiamjitrak, W. P. (2023). New Analytical Methods for Online Binary Search Trees. <https://aaltodoc.aalto.fi/server/api/core/bitstreams/75ad27f2-36ad-43b1-bccf-ca18900124ba/content>

Mankowitz, D. J., Michi, A., Zhernov, A., Gelmi, M., Selvi, M., Paduraru, C., ... & Silver, D. (2023). Faster sorting algorithms discovered using deep reinforcement learning. Nature, 618(7964), 257-263. <https://www.nature.com/articles/s41586-023-06004-9.pdf>

Lobo, J., & Kuwelkar, S. (2020, July). Performance analysis of merge sort algorithms. In 2020 International Conference on Electronics and Sustainable Communication Systems (ICESC) (pp. 110-115). IEEE. <https://www.researchgate.net/profile/Sonia-Kuwelkar/publication/343434465_Performance_Analysis_of_Merge_Sort_Algorithms/links/60f860ed2bf3553b29029346/Performance-Analysis-of-Merge-Sort-Algorithms.pdf>

Jugé, V. (2024). Adaptive Shivers sort: an alternative sorting algorithm. ACM Transactions on Algorithms, 20(4), 1-55. <https://arxiv.org/pdf/1809.08411>

Lin, H., Luo, T., & Woodruff, D. (2022, June). Learning augmented binary search trees. In International Conference on Machine Learning (pp. 13431-13440). PMLR. <https://proceedings.mlr.press/v162/lin22f/lin22f.pdf>

Corsini, B., Dubach, V., & Féray, V. (2024). Binary search trees of permuton samples. Advances in Applied Mathematics, 162, 102774. <https://www.sciencedirect.com/science/article/pii/S0196885824001064>

Li, H. (2024). Survey and Analysis of Citizen Network Political Participation Based on Binary Search Tree Algorithm. EAI Endorsed Transactions on Scalable Information Systems, 11(3). <https://publications.eai.eu/index.php/sis/article/download/4922/2896>

Kristo, A., Vaidya, K., Çetintemel, U., Misra, S., & Kraska, T. (2020, June). The case for a learned sorting algorithm. In Proceedings of the 2020 ACM SIGMOD international conference on management of data (pp. 1001-1016). <https://dl.acm.org/doi/pdf/10.1145/3318464.3389752>

Shaffer, C. A. (2022). Data structures and algorithm analysis. <http://123.200.31.10:8080/jspui/bitstream/123456789/105/1/Data%20Structure%20%26%20Algorithm%20Analysis.pdf>

Khoirom, S., Sonia, M., Laikhuram, B., Laishram, J., & Singh, T. D. (2020). Comparative analysis of Python and Java for beginners. Int. Res. J. Eng. Technol, 7(8), 4384-4407. <https://www.academia.edu/download/94738677/IRJET-V7I8755.pdf>