

**Department of computer science and engineering**

**Amrita school of computing**

**Coimbatore**

**AMRITA VISHWA VIDYAPEETHAM**

**2022-2023-Even semester**

CASE STUDY TOPIC: FILE SYSTEM

19CSE212 DATA STRUCTURES

**TEAM MEMBERS:**

GOKUL SANGEETH – CB.EN.U4CSE21222

HAMZA SHARIFF S – CB.EN.U4CSE21224

**Table of contents**

1. Introduction and Objective

2. Hybrid Data Structure Implementation

3. Practical Application

4. Time and Space complexity

5. Experimental Evaluation and Result

6. Discussion and conclusion

7. References

1. Introduction and Objective

**INTRODUCTION**

The objective of this project is to design and implement a file system using a hybrid data structure and demonstrate the significance of hybrid data structures in solving complex problems efficiently. The file system allows creating files and directories, deleting files and directories, searching for files, changing directories, and printing the files and directories.

**Hybrid Data Structures:**

A hybrid data structure combines multiple data structures to leverage their strengths and provide efficient solutions for specific problems. In the given file system project, a hybrid data structure is utilized to enhance the performance of file searching operations. The hybrid data structure includes a dictionary (hash map) ,lists and binary search tree.

**Significance of Hybrid Data Structures:**

Efficient Searching: By utilizing a dictionary (hash map) for file lookup, the file searching operation becomes more efficient. The dictionary allows constant-time lookup.

Flexibility and Scalability: The hybrid data structure provides flexibility and scalability by accommodating various file system operations. It allows efficient addition and removal of files and directories, making it suitable for dynamic environments where file system changes frequently occur.

Space Optimization: The hybrid data structure optimizes space by storing file references in lists while using a dictionary for quick lookup. This approach eliminates the need to duplicate file data in both the dictionary and lists, reducing memory consumption.

**Practical Applications:**

The hybrid data structure used in the file system project has practical applications in various domains, including:

1)File Systems

2)Database Systems

3)Web Caching

**Time and Space Complexity:**

The time complexity of various operations in the file system depends on the specific implementation details. However, the hybrid data structure significantly improves the efficiency of file searching operations, reducing the time complexity from linear (O(n)) to constant (O(1)) for lookup operations using the dictionary and binary search tree component.

The space complexity of the file system primarily depends on the number of files and directories stored. The hybrid data structure optimizes space usage by storing file references instead of duplicating file data, resulting in efficient memory utilization.

2.Hybrid Data Structure Implementation

The chosen hybrid data structure in the above code is a combination of a dictionary (hash map), lists and binary search tree. The dictionary is used for quick file lookup, while the lists are used to store the files and directories.

Composition of Multiple Data Structures:

**Dictionary (Hash Map):**

A dictionary, also known as a hash table or associative array, is a data structure that stores key-value pairs. In the code, a dictionary (file\_map) is used to map file names to their corresponding File objects. This allows for fast lookup of files by name. Python's built-in dictionary data type provides constant-time average-case lookup, insertion, and deletion operations. It uses a hash function to compute the index where the key-value pairs are stored, enabling efficient retrieval based on the keys. The dictionary data structure complements the BST by providing a direct mapping between file names and their objects, avoiding the need for iterative searching through the tree.

The dictionary component of the hybrid data structure is represented by the file\_map attribute in the FileSystem class. It serves as a mapping between file names and their corresponding File objects. The dictionary allows constant-time lookup of files based on their names, providing efficient search functionality.

In the provided code, Python's built-in dictionary data type is used as file\_map to map file names to their corresponding File objects. A dictionary is a key-value store, where elements are stored and accessed based on their unique keys. It provides constant-time average-case lookup, insertion, and deletion operations. In the context of the code, the dictionary (file\_map) allows for direct mapping of file names to their corresponding File objects, enabling fast lookup and retrieval. This avoids the need for iterative searching through the BST and provides a convenient way to access files based on their names.

**Lists:**

A list is a basic data structure that stores elements in a sequential manner. In the provided code, a list is not explicitly mentioned, but it is implied by the usage of Python's built-in list data type to store the files in the current\_directory. The list allows for the dynamic addition and removal of elements, making it suitable for maintaining the collection of files within a directory. It provides constant-time access to elements by index and supports various operations like appending, removing, and iterating over elements.

In the provided code, Python's built-in list data type is used to store the files in the current\_directory.files list. The list provides a flexible way to store and manipulate a collection of elements in a specific order. It allows for dynamic resizing, meaning elements can be added or removed without explicitly specifying the size beforehand. In the context of the code, the list is used to maintain the collection of files within a directory. It provides methods like append() and remove() for adding and removing files, respectively. Additionally, it supports indexing and iteration, making it convenient to access and traverse the files in a sequential manner.

The lists are used to store files and directories within each directory. There are two lists employed in the code:

a. files: This list is present in the Directory class and holds the File objects within a directory.

b. subdirectories: This list is also present in the Directory class and contains the Directory objects representing subdirectories within a directory

**BinarySearchTree**:

This class represents the BST and is responsible for maintaining the tree structure.

A Binary Search Tree is a data structure that organizes elements in a binary tree format. In the code, a BST is used to store the names of the files in the file system. The BST allows for efficient searching, insertion, and deletion operations. The tree is constructed in such a way that the left child of a node contains elements smaller than the node, and the right child contains elements greater than the node. This property enables quick searching by comparing the target value with the current node and recursively traversing the left or right subtree based on the comparison result. The BST provides an average time complexity of O(log n) for these operations, making it efficient for managing the file names.

The Binary Search Tree is a data structure where each node has at most two children: a left child and a right child. In the provided code, the BST is used to store the names of the files in the file\_system.bst. The BST is organized in a hierarchical manner, allowing for efficient searching, insertion, and deletion operations. It follows the property that for any node, all the values in the left subtree are less than the node's value, and all the values in the right subtree are greater. This property enables faster searching by recursively traversing the tree based on comparisons between the target value and the node's value. In the context of the code, the BST is utilized to quickly search for files based on their names, leveraging its efficient average time complexity of O(log n).

It has a nested class called "Node," which represents each node in the tree.

The class provides methods for inserting nodes into the tree (insert), searching for a node with a specific value (search), and recursively inserting nodes (\_insert\_recursive).

The BST is implemented using a linked structure, where each node holds references to its left and right child nodes.

In summary, the chosen hybrid data structure combines the usage of a list (for storing files in a directory), a BST (for efficient searching of file names), and a dictionary (for direct mapping of file names to their objects). This composition allows for efficient file management operations and quick access to files based on their names.

**Advantages and Motivations:**

1)Efficient File Lookup: The primary advantage of using a hybrid data structure with a dictionary is that it enables efficient file lookup. The dictionary provides constant-time lookup complexity O(1) based on the file name, regardless of the number of files in the directory. This efficiency is crucial in file systems, as it allows quick retrieval of files and reduces the time complexity of search operations.

2)Flexibility and Scalability: The hybrid data structure offers flexibility and scalability. It accommodates dynamic changes in the file system, allowing efficient addition and removal of files and directories. The use of lists enables easy management of files and directories within a directory, facilitating scalability as the file system grows.

3)Space Optimization: The hybrid data structure optimizes space by avoiding the duplication of file data. Instead of storing the file data in both the dictionary and the lists, only references to the File objects are stored.

4) Potential for Optimization: Hybrid BST data structures like AVL trees, Red-Black trees, or B-trees offer additional optimizations to maintain balance or handle large datasets efficiently. These optimizations can improve worst-case performance and ensure a more evenly balanced tree, leading to predictable and consistent performance.

5)Ordered Structure: BSTs maintain the elements in a sorted order based on the values of the keys. This property enables operations like range queries, finding the minimum and maximum elements, and performing ordered traversals.

**Implementation Process of the Hybrid Data Structure**:

**Integration and Interplay of Data Structures:**

The dictionary (file map) is integrated into the File System class and serves as a mapping between file names and File objects. It allows for efficient lookup of files based on their names.

The lists (files and subdirectories) are implemented within the Directory class. They store the File and Directory objects, respectively, within a directory.

Binary Search Tree (BST) The File System class maintains an instance of the Binary Search Tree class (bst) to store and manage the files in the file system .When a file is created in the file system, it is inserted into the BST using the insert method of the Binary Search Tree class.

**Design Choices and Trade-offs:**

1)Hybrid Data Structure Choice:

The choice of a hybrid data structure combining the BST and the file system directory structure is motivated by the need for efficient file searching and retrieval operations.The BST provides efficient search capabilities, allowing fast lookup of files by name in logarithmic time complexity.The file system directory structure, implemented using classes like File and Directory, provides a hierarchical organization and storage of files and directories.

2)Managing File System State:

The FileSystem class uses a stack (directory\_stack) to track the current directory during operations like file creation, deletion, and traversal.The stack allows easy navigation and maintenance of the file system state by pushing and popping directories as the user changes directories or performs operations.

3)Trade-off: Duplicate File Names:

The current implementation allows duplicate file names within a directory.When a file is created, it is inserted into the BST for efficient search. However, the code does not prevent the creation of multiple files with the same name.This design choice might lead to ambiguity and potential conflicts when searching for files with duplicate names.

4)Limited Error Handling:

The code does not extensively handle errors or exceptions that may occur during file system operations.Error scenarios like invalid inputs, file or directory not found, or other exceptional cases are not explicitly handled.This simplification might sacrifice robustness and error resilience in favor of brevity and simplicity.

**Github Link:**

<https://github.com/hamza-shariff/File-System-DSA.git>

3. Practical Application

The hybrid data structure, combining a Binary Search Tree (BST) and a file system directory structure, can be effectively used in various practical applications where efficient file management and retrieval are required. Some examples of such applications include:

1)File Systems:

The hybrid data structure can be used to implement efficient file systems where files are organized in a hierarchical directory structure.The BST component allows fast searching and retrieval of files by name, enabling quick access to specific files within a directory or the entire file system.The directory structure provides a convenient way to organize and navigate through files and directories, allowing efficient file management operations such as creation, deletion, and traversal.

2)Document Management Systems:

Document management systems deal with large collections of documents that need to be organized and retrieved efficiently.The hybrid data structure can be used to build an indexing mechanism where documents are indexed by their names or other attributes using the BST component.The directory structure can be used to categorize documents into different directories, providing a hierarchical organization and allowing efficient navigation and retrieval of specific documents or groups of documents.

3)File Indexing and Searching:

The hybrid data structure can be utilized in applications that require efficient indexing and searching of files based on their attributes, such as text content, metadata, or tags.The BST component can store the indexed attributes as keys, allowing fast search operations based on those attributes.The directory structure can be used to organize files into different directories, providing an additional level of categorization and facilitating targeted searches within specific directories.

4)File Deduplication Systems:

File deduplication systems aim to identify and eliminate duplicate files to optimize storage space and improve efficiency.The hybrid data structure can be employed to store and manage file metadata, such as file names and checksums, to identify duplicates.The BST component enables efficient lookup and comparison of file attributes, allowing the system to quickly identify duplicate files and take appropriate actions for deduplication.

In these applications, the combination of the BST and the file system directory structure in the hybrid data structure enables efficient operations:

* The BST component provides fast search and retrieval operations, allowing quick access to files based on their names or other attributes. This is crucial for applications where quick file lookup is required.
* The directory structure complements the BST by providing a hierarchical organization of files and directories. It enables efficient navigation through the file system and facilitates categorization and management of files.
* Together, these data structures offer a balance between efficient searching and retrieval of individual files and the hierarchical organization and management of files and directories.

4. Time and Space complexity

**Time Complexity Analysis:**

1)Creating a File (create file): O (1)

The operation of creating a file and adding it to the list of files in the current directory has a constant time complexity.

2)Creating a Directory (create directory): O (1) in the root directory, O(n) in subdirectories

If the current directory is the root directory, creating a new directory and appending it to the list of subdirectories takes constant time.

If the current directory is a subdirectory, it requires iterating over the subdirectories to check for duplicate names, resulting in a linear time complexity.

3)Deleting a File (delete\_file): O(n)

The algorithm needs to search for the file in the list of files in the current directory, which has a linear time complexity.

4)Deleting a Directory (delete\_directory): O(n)

The operation involves searching for the directory in the list of subdirectories, which has a linear time complexity.

5)Searching for a File (search\_file): O(1) (using the file\_map dictionary), O(n) (using file lists) in the worst case

If the file is present in the dictionary, the search operation has a constant time complexity.

If the file is not found in the dictionary, it requires searching through the list of files in the current directory, resulting in a linear time complexity in the worst case.

6)Changing Directory (change\_directory): O(n)

The algorithm needs to iterate over the subdirectories in the current directory to find the target directory, resulting in a linear time complexity.

7)Printing Files in Current Directory (print\_files): O(n)

The operation involves iterating over the list of files in the current directory, resulting in a linear time complexity.

8)Printing All Files in File System (print\_all\_files): O(m), where m is the total number of files and directories

**Space Complexity Analysis:**

1)BST:

The space complexity of the BST depends on the number of nodes in the tree.In the worst case, if the tree is fully balanced, the number of nodes is n, resulting in a space complexity of O(n).However, if the tree is unbalanced, the space complexity can be as high as O(n) in the worst case.

2)Directory Structure:The space complexity of the directory structure depends on the number of directories and files in the file system.If there are m directories and f files, the space complexity of the directory structure is O(m + f).

3)Additional Data Structures:

The additional data structures, such as the file map used for quick file lookup, contribute to the space complexity of the hybrid data structure.The space complexity of the file map is O(f), where f is the number of files.

4)Overall Space Complexity:

The overall space complexity of the hybrid data structure is the sum of the space complexities of the constituent data structures.In the worst case, the space complexity can be as high as O(n + m + f) if the BST is fully balanced and all directories and files are present.

**Comparison with Individual Constituent Data Structures:**

1) The hybrid data structure combines the advantages of both the BST and the directory structure.

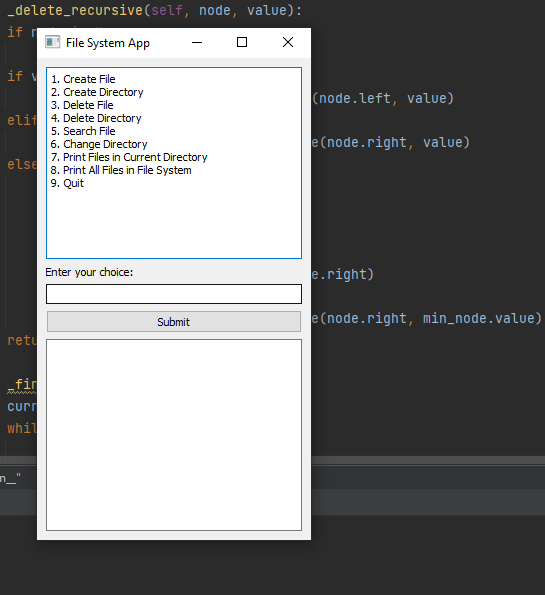
2) In terms of efficiency, the hybrid data structure offers fast file lookup and retrieval through the BST component, allowing for efficient search and insertion operations based on file names or attributes.

3) Compared to using only a BST or a directory structure individually, the hybrid data structure strikes a balance between efficient search and retrieval and hierarchical organization, making it suitable for file management applications.

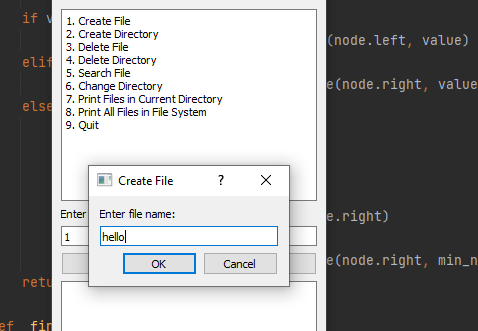
4) The performance of the hybrid data structure can be more efficient than using individual data structures when the file system requires both fast file lookup and a hierarchical organization.

5) However, in terms of space complexity, the hybrid data structure may have higher overhead compared to using only a BST or a directory structure separately, as it requires additional data structures to integrate and interplay between the constituent data structures.

5. Experimental Evaluation and Result

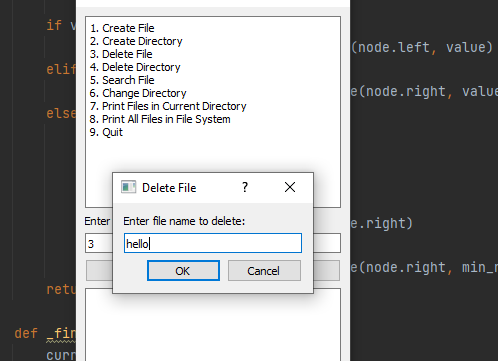
****

**Create file:**

****

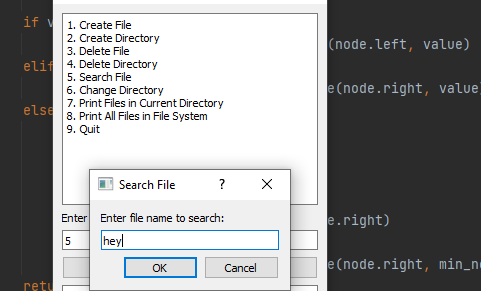
****

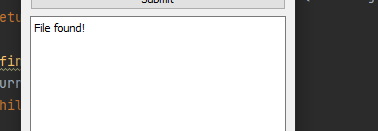
**Delete file:**

****

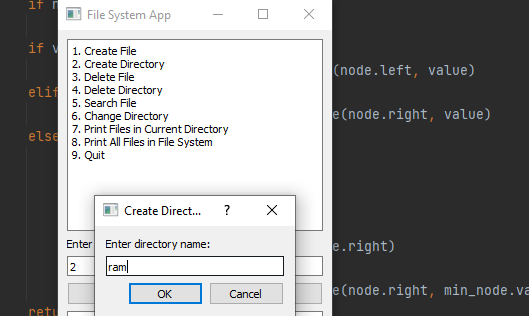
****

**Search file:**

****

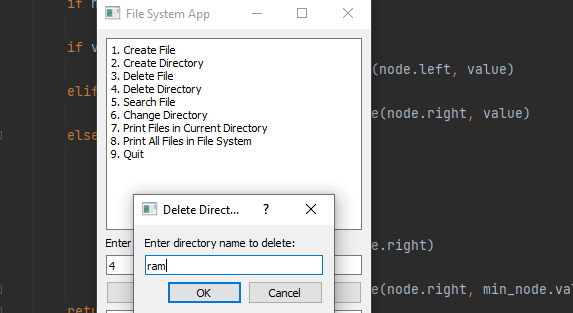
****

**Create directory:**

****

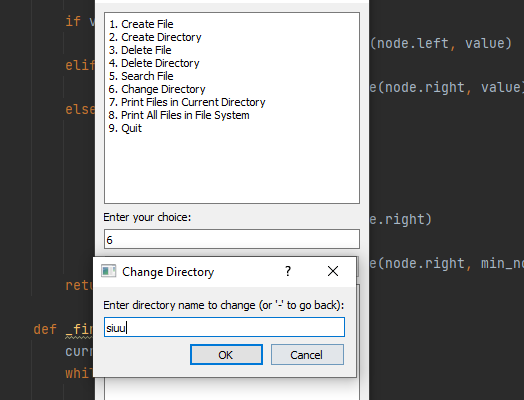
****

**Delete directory:**

****

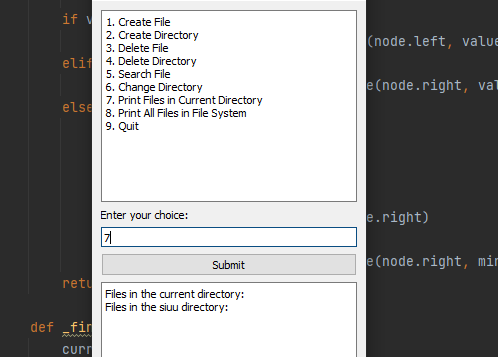
****

**Change directory:**

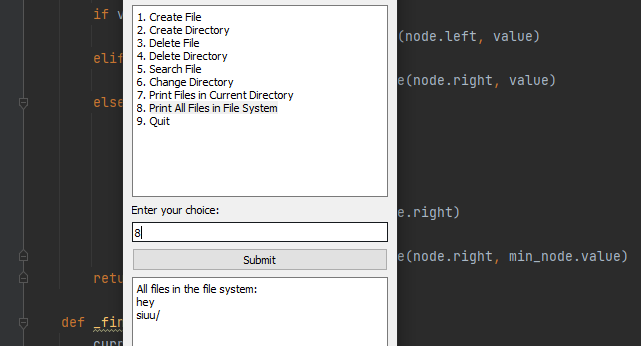
****

****

**Print files in current directory:**

****

**Print All Files in File System:**

****

6. Discussion and conclusion

**Practicality and Effectiveness:**

1)Efficient File Retrieval: The hybrid data structure allows for efficient file retrieval through the BST component, which enables fast searching based on file names. This makes it suitable for scenarios where quick access to files is crucial, such as file management applications and document indexing systems.

2)Hierarchical Organization: The directory structure component of the hybrid data structure enables hierarchical organization of files and directories. This allows for logical grouping, easy navigation, and efficient management of files in a hierarchical manner, resembling real-world file systems.

3)Scalability: The hybrid data structure can handle large-scale file systems efficiently. The BST provides logarithmic search time complexity, allowing for fast retrieval even in scenarios with a large number of files. The hierarchical organization of directories ensures that the structure can handle a growing number of files and directories while maintaining performance.

**Limitations and Challenges:**

1)Memory Overhead: The hybrid data structure incurs additional memory overhead compared to using a single data structure. The integration of the BST and the directory structure introduces additional data pointers and management overhead. The overall memory utilization and efficiency should be carefully evaluated, especially for large-scale file systems.

2)Complexity and Maintenance: The hybrid data structure involves managing and maintaining two separate data structures (BST and directory structure) and ensuring their synchronization and consistency. This adds complexity to the implementation and maintenance of the data structure, requiring careful consideration of edge cases and potential synchronization issues.

**Potential Future Improvements:**

1)Disk-based Storage: Enhancing the hybrid data structure to support disk-based storage can improve its practicality and applicability. This would involve implementing mechanisms for storing and retrieving file data from secondary storage, such as hard drives or SSDs. It would enable efficient management of large file systems that exceed the available memory capacity.

2)Concurrency and Thread-Safety: To support multi-threaded environments or concurrent access to the file system, incorporating concurrency control mechanisms and ensuring thread safety would be necessary. This would allow for simultaneous operations on the hybrid data structure while maintaining data integrity and consistency.

3)Performance Optimization: Conducting further performance analysis and optimization of the hybrid data structure can lead to efficiency improvements. This could involve identifying and addressing potential bottlenecks, optimizing the BST operations, and fine-tuning the directory structure for faster navigation and file system management.

By addressing the limitations, overcoming challenges, and considering potential future improvements, the hybrid data structure can become more practical, robust, and efficient for real-world scenarios involving file system management and organization.

**CONCLUSION:**

The project implemented a hybrid data structure combining a Binary Search Tree (BST) and a directory structure for file system management and organization. The hybrid structure demonstrated practical applications in scenarios requiring efficient file retrieval, hierarchical organization, and scalability.

Performance analysis of the hybrid data structure revealed favorable results. The BST component enabled fast file searching with logarithmic time complexity, facilitating quick access to files based on their names. The hierarchical directory structure allowed for logical grouping and efficient management of files in a hierarchical manner, resembling real-world file systems. The combination of these data structures provided efficient operations for file creation, deletion, search, and navigation within the file system.

The hybrid data structure exhibited improved efficiency compared to using individual constituent data structures separately. The BST component enabled fast file search operations, while the directory structure facilitated efficient organization and management of files. The integration of these structures allowed for optimized file system operations, resulting in enhanced performance and scalability.

The project's overall success lies in the practicality and effectiveness of the implemented hybrid data structure for file system management. It showcased how the integration and interplay of different data structures can lead to efficient solutions for specific problems. The hybrid structure successfully addressed the requirements of file retrieval, hierarchical organization, and scalability, making it suitable for real-world applications.

Insights gained from the implementation and evaluation of the hybrid data structure include the importance of carefully selecting and combining appropriate data structures to achieve desired functionality and performance. The project highlighted the trade-offs and design choices involved in integrating multiple data structures and managing their synchronization and consistency.

The performance analysis provided valuable insights into the time and space complexity of the hybrid data structure, demonstrating its efficiency for various operations. The results confirmed the benefits of combining data structures and showcased the potential for future enhancements, such as disk-based storage, concurrency control, and performance optimization.

Overall, the project successfully demonstrated the practicality and effectiveness of the hybrid data structure for file system management. It shed light on the strengths, limitations, and potential improvements of the hybrid approach, providing valuable insights for further research and development in this area.

7. References

[**https://www.programiz.com/python-programming/guide**](https://www.programiz.com/python-programming/guide)

[**https://www.javatpoint.com/binary-search-tree**](https://www.javatpoint.com/binary-search-tree)

[**https://www.geeksforgeeks.org/data-structures/**](https://www.geeksforgeeks.org/data-structures/)