FILE INDEXING



**19CSE212 Data Structures and Algorithm**

**CASE STUDY**

**GROUP 19**

**TEAM MEMBERS**

|  |  |
| --- | --- |
| **NAME** | **ROLL NO** |
| PUDOTA HARSHITHA | CB.EN.U4CSE21248 |
| THIMMISETTY SAILEELA | CB.EN.U4CSE21264 |

# TABLE OF CONTENTS

1. **Introduction and Objective**
2. **Overview of the Hybrid Data Structure**
3. **Implementation Details**
4. **Practical Applications**
5. **Performance Analysis**
6. **Experimental Evaluation**
7. **Discussion**
8. **Conclusion**
9. **Reference**

**1. Introduction and Objectives :**

**Introduce the concept of hybrid data structures and their significance in solving complex problems efficiently.**

Hybrid data structures refer to the combination or integration of multiple data structures to create a new data structure that offers enhanced efficiency and functionality. These data structures leverage the strengths of different underlying structures to address specific problem-solving requirements effectively.

The significance of hybrid data structures lies in their ability to optimize various operations, such as insertion, deletion, retrieval, and search, while managing large amounts of data or solving complex problems. By leveraging the strengths of different data structures, hybrid data structures can provide efficient solutions to a wide range of computational challenges.

**Significance** :

Significance of Tries with Hash Tables:

Tries are efficient for autocomplete functionality in search engines, but they can consume a lot of memory. By combining tries with hash tables, memory usage can be reduced while still maintaining fast prefix-based searches. This allows search engines to provide quick and memory-efficient autocomplete suggestions to users.

Significance of Skip Lists with Trees:

Skip lists provide a simple and efficient way to maintain sorted lists with fast insertion, deletion, and search operations. However, for very large datasets, skip lists may not offer the optimal search performance. By combining skip lists with balanced trees, the search performance can be improved while still retaining the simplicity and efficiency of skip lists. This is beneficial in scenarios where maintaining a sorted list is crucial, such as in file systems or databases.

**Objectives :**

**Design and implement a hybrid data structure:** The code aims to create a hybrid data structure to represent a file system tree. It uses a combination of classes and data structures to organize and manage files and directories.

**Practical applications:** The code provides functionalities to manipulate and manage files and directories within the file system tree. This can be useful in various applications, such as file management systems, operating systems, data backup utilities, and more.

**Analyze time & space complexity:** The code aims to efficiently perform various operations on the file system tree. By analyzing the time & space complexity of each operation, we can understand how the code performs under different scenarios, optimize it if necessary & how the code utilizes memory, storage resources and optimize it if needed

**Time & Space complexity :**

The overall time complexity will depend on the sequence of operations performed.

Time complexity : O(N \* M \* log M)

Space complexity : O(N)

N – total operations

M – no.of files & directories

1. **Overview of the Hybrid Data Structure**

**Explain the chosen hybrid data structure :**

The hash tree used in the provided code is a data structure that calculates and stores hash values for the nodes in a file system tree. Let's delve into the details of how the hash tree works:

**Purpose:**

The hash tree is primarily used to generate unique hash values for each node in the file system tree. These hash values serve as a compact representation of the contents and structure of the files and directories. By calculating hash values for nodes, the hash tree allows for efficient comparison and verification of file system integrity.

**Construction Process:**

The construction of the hash tree involves recursively traversing the file system tree and calculating hash values for each node. The process can be summarized as follows:

If the current node is a file, the hash value is calculated using the file's name and size. This is done by iterating over the characters in the file's name and summing up their ASCII values. The resulting sum is converted to a string and stored as the hash value for the file node.

If the current node is a directory, the hash value is computed by concatenating the hash values of all its child nodes. The hash values of the child nodes are obtained recursively by applying the same process.

**Benefits and Applications:**

The hash tree provides several benefits and applications in the context of the hybrid data structure:

Efficient Integrity Verification: By storing hash values for each node, the hash tree allows for quick integrity verification of the file system. Comparing the hash values of nodes against their actual contents can help identify any changes or corruption in the file system.

Quick File Comparison: The hash values provide a concise representation of the files and directories. By comparing hash values, it becomes efficient to identify if two files have the same contents or if two directories have the same structure.

Fast Lookup in Hash Index: The hash values calculated by the hash tree can be used as keys in the hash index of the hybrid data structure. This enables quick and direct access to files based on their hash values, making operations like searching, deletion, and moving files more efficient.

**Time and Space Complexity:**

The time complexity of constructing the hash tree is O(n), where n is the total number of nodes in the file system tree. This is because each node is visited once during the recursive traversal.

The space complexity of the hash tree is also O(n) as it requires additional storage to store the hash values for each node. The space required is proportional to the number of nodes in the file system tree.

**Its composition of multiple data structures :**

These multiple data structures work together to provide efficient file system management operations, such as adding files, deleting files, navigating the file system tree, and verifying file integrity. The hash table enables quick lookup and retrieval of files based on their hash values, the linked list and stack facilitate tree traversal and manipulation, and the hash tree generates hash values for integrity verification. The File and Directory objects provide a structured representation of files and directories in the file system.

**Hash Table:**

The code uses a hash table to store the file system index. It is implemented using a dictionary data structure, where the keys are the hash values of the files, and the values are the corresponding file paths. The hash table allows for efficient lookup and retrieval of file paths based on their hash values.

**Linked List:**

A linked list is used to store the file system tree structure. Each node in the linked list represents a directory, and it contains references to its child nodes (subdirectories and files) through pointers. The linked list allows for efficient traversal and manipulation of the file system tree.

**Hash Tree:**

The code incorporates a hash tree to generate hash values for each node in the file system tree. The hash tree is a hierarchical data structure where each node's hash value is calculated based on the hash values of its child nodes. The hash tree enables efficient integrity verification and comparison of file system nodes.

**File Object:**

The code defines a custom File class or structure to represent individual files in the file system. Each File object stores attributes such as the file name, size, and modification time. The File objects are used in various operations, such as adding files to the file system, updating file attributes, and retrieving file information.

**Directory Object:**

Similarly, the code defines a custom Directory class or structure to represent directories in the file system. Each Directory object contains the directory name and references to its child nodes (subdirectories and files). The Directory objects enable organizing and managing the hierarchical structure of the file system.

**Stack:**

A stack data structure is utilized in the code to keep track of the current directory during the traversal of the file system. As the code traverses the file system tree, it pushes the directories onto the stack, and when it finishes processing a directory, it pops it from the stack. The stack allows for depth-first traversal of the file system tree.

**Discuss the advantages and motivations behind using a hybrid data structure for solving specific problems efficiently.**

The advantages of data structures to efficiently solve specific problems in file system management. It enables fast file lookup, hierarchical structure management, integrity verification, flexibility, scalability, and space efficiency, making it suitable for handling file systems effectively and providing a reliable foundation for file management operations.

The motivation behind using the hybrid data structure in the specific requirements and challenges of file system management. It offers efficient file operations, data integrity verification, scalability, flexibility, space efficiency, and code maintainability, making it a suitable choice for implementing a file system management system.

1. **Implementation Details**

**Describe the implementation process of the hybrid data structure, including the integration and interplay of the constituent data structures.**

FileNode class:

Represents a node in the file system tree.

It has attributes such as name (name of the file or directory), is\_directory (boolean indicating if it's a directory), children (list of child nodes), size (file size in bytes), last\_modified (timestamp of the last modification), and hash (calculated hash value for file integrity checking).

The \_init\_ method initializes the attributes with default values.

HashTree class:

Represents a hash tree used to calculate hash values for file integrity checking.

It has a calculate\_hash method that calculates a hash value for a given data string by summing the ordinal values of its characters.

The construct\_tree method constructs the hash tree recursively by calculating hash values for each node based on its children's hashes.

The HashTree instance can be used to calculate and store hash values for the file system tree.

FileSystemTree class:

Represents the file system tree itself.

It has a root attribute, which is a FileNode representing the root directory of the tree.

The tree structure is built using FileNode instances and organized as a tree hierarchy with directories and files.

add\_file(self, path, size):

Adds a file to the file system tree.

It takes the path of the file and its size in bytes as input.

The function traverses the path and creates any necessary directories along the way.

Finally, it creates a new FileNode for the file and adds it to the appropriate directory node in the tree.

new\_directory(self, path, directory\_name):

Creates a new directory in the file system tree.

It takes the path of the parent directory and the directory\_name as input.

The function finds the parent directory node based on the given path and adds a new FileNode with the specified directory name as its child.

rename\_file(self, path, new\_name):

Renames a file in the file system tree.

It takes the path of the file and the new\_name as input.

The function finds the file node based on the given path and updates its name attribute with the new name.

sort\_files\_by\_last\_modified(self):

Sorts the files in the file system tree based on their last modified timestamp.

It retrieves all the files from the size\_index and sorts them based on the last\_modified attribute.

The sorted files are printed, showing their names and last modified timestamps.

The sorted file list is returned.

calculate\_directory\_size(self, directory\_path):

Calculates the total size of a directory and its contents.

It takes the directory\_path as input.

The function finds the directory node based on the given path and recursively calculates the size of the directory by summing the sizes of all files within it.

The total size of the directory is returned.

open\_directory(self, directory\_path, file\_name, file\_size):

Opens a directory in the file system tree and adds a new file to it.

It takes the directory\_path where the directory should be opened, the file\_name, and the file\_size in bytes as input.

The function finds the directory node based on the given path and adds a new FileNode for the file with the specified name and size as its child.

list\_directory\_contents(self, directory\_path):

Lists the contents (files and subdirectories) of a directory in the file system tree.

It takes the directory\_path as input.

The function finds the directory node based on the given path and prints the names of all its children.

get\_files\_above\_size(self, size):

Retrieves a list of files in the file system tree that have a size greater than the specified size.

It takes the size in bytes as input.

The function iterates through the size\_index to find files with sizes greater than the specified size and returns a list of such files.

delete\_file(self, file\_path):

Deletes a file from the file system tree.

It takes the file\_path as input.

The function finds the file node based on the given path and removes it from its parent directory's children list.

move\_file(self, source\_path, destination\_path):

Moves a file from a source path to a destination path within the file system tree.

It takes the source\_path of the file to be moved and the destination\_path where it should be moved to as input.

The function finds the source file node and its parent directory based on the given paths, removes the file node from the parent directory's children list, and adds it as a child to the destination directory node.

visualize(self):

Visualizes the file system tree in a hierarchical format.

It traverses the tree starting from the root node and prints each node's name with proper indentation to represent the hierarchy.

The tree structure is displayed in a readable format.

**Explain any design choices and trade-offs made during the implementation phase.**

These design choices and trade-offs reflect a balance between the desired functionalities, performance requirements, code complexity, and maintenance considerations. Depending on the specific use case and priorities, different design choices and trade-offs may be made to optimize the implementation for specific requirements.

Here are some design choices and trade-offs made during the implementation

1.Hybrid Data Structure

2.Hash Tree Calculation

3.Size Indexing

4.Time Complexity

5.Code Modularity and Readability

6.Error Handling

**4. Practical Applications**

**Identify and describe practical applications where the hybrid data structure can be effectively used.**

**File System Management:** The code can be used as a foundation for implementing a file system management system. It allows users to create, rename, delete, and move files and directories, as well as perform operations like listing directory contents, calculating directory sizes, and sorting files by last modified time. This can be useful in creating file explorers or file system utilities.

**Version Control Systems:** The file system tree structure can be utilized in version control systems like Git. It can be used to represent the directory structure and track changes made to files over time. The hash tree can help in efficiently determining the changes made to files and detecting conflicts.

**Content Management Systems:** The code can be adapted to build a content management system (CMS) that organizes and manages digital content such as articles, images, and videos. The file system tree can represent the hierarchical structure of content categories and subcategories, allowing efficient navigation and management of the content.

**Backup and Restore Systems:** The file system tree can be utilized in backup and restore systems to efficiently store and retrieve files and directories. The ability to calculate directory sizes and sort files by last modified time can assist in creating backup strategies and performing incremental or differential backups.

**Data Indexing and Searching:** The code can be extended to support indexing and searching of files based on their attributes such as file name, size, and last modified time. This can be beneficial in scenarios where quick file lookup or search operations are required, such as in document management systems or file search utilities.

**Project Organization:** The file system tree can be utilized in project management systems to organize project files, track changes, and manage collaboration. It can provide a hierarchical structure to store project-related documents, code files, and other resources.

These are just a few examples of practical applications where a file system tree and associated operations can be useful. The flexibility and extensibility of the code allow for customization and adaptation to specific use cases and requirements.

**Discuss how the combination of data structures in the hybrid structure enables efficient operations for these applications.**

By combining the hash table and binary search tree, the hybrid data structure takes advantage of their individual strengths and mitigates their weaknesses. It provides fast key-value lookups through hashing and efficient ordered traversals through the binary search tree. This enables a wide range of applications to efficiently perform operations such as searching, insertion, deletion, range queries, and ordered traversals, offering a balanced trade-off between time complexity and space utilization.

how the combination facilitates efficiency:

1.Fast Key-Value Lookups

2.Ordered Traversals

3.Handling Collisions

4.Efficient Updates

5.Space Optimization

**5.** **Performance Analysis**

**Analyze the time complexity of key operations supported by the hybrid data structure.**

**Building the Hash Tree:**

Time Complexity: O(n log n), where n is the number of files in the file system.

Explanation: To build the hash tree, the code iterates over each file, computes the hash value for each file, and inserts it into the tree. The insertion operation takes O(log n) time in a balanced tree, and since there are n files, the overall time complexity is O(n log n).

**Verifying Integrity of a File:**

Time Complexity: O(log n), where n is the number of files in the file system.

Explanation: When verifying the integrity of a file, the code traverses the hash tree from the root to the leaf corresponding to the file. Since the hash tree is balanced, the maximum height of the tree is log n. Therefore, the time complexity for verifying the integrity of a file is O(log n).

**Verifying Integrity of the Entire File System:**

Time Complexity: O(n log n), where n is the number of files in the file system.

Explanation: To verify the integrity of the entire file system, the code iterates over each file and performs the integrity check operation, which has a time complexity of O(log n) for each file. Since there are n files, the overall time complexity is O(n log n).

**Inserting or Deleting a File:**

Time Complexity: O(log n), where n is the number of files in the file system.

Explanation: When inserting or deleting a file, the code needs to update the hash tree accordingly. Since the hash tree is balanced, the insertion or deletion operation takes O(log n) time in a balanced tree.

**Searching for a File:**

Time Complexity: O(log n), where n is the number of files in the file system.

Explanation: Searching for a file involves traversing the hash tree from the root to the leaf corresponding to the file. Since the hash tree is balanced, the maximum height of the tree is log n. Therefore, the time complexity for searching for a file is O(log n).

It's important to note that the above time complexities assume that the hash tree is balanced. If the tree becomes unbalanced due to certain operations or implementation choices, the time complexities may increase. Additionally, the specific implementation details and optimizations made during the implementation can affect the actual performance.

**Analyze the space complexity, including memory utilization and overhead, of the hybrid data structure.**

The space complexity of the hybrid data structure can be expressed as a combination of the space required by the linked lists, hash tables, and hash tree. The total space complexity can be

Space Complexity = O(n) + O(n) + O(path\_length \* n)

where:

n is the total number of files and directories in the file system.

path\_length is the average length of the file paths.

**Compare the performance of the hybrid data structure with individual constituent data structures in terms of efficiency.**

The hash tree used in the hybrid data structure offers improved efficiency compared to individual data structures when it comes to file integrity verification, insertion and deletion of files, and searching for files. The use of a tree structure enables logarithmic time complexity for these operations, which is more efficient than linear time complexity offered by individual data structures. However, it's important to note that the specific implementation details and optimizations can also impact the performance, so it's crucial to consider those factors as well.

1. **Experimental Evaluation**

**Present experimental setup and methodology used to measure the performance of the hybrid data structure.**

**Experimental Setup:**

Hardware: Determine the hardware specifications of the system on which the experiments will be conducted, including the CPU, RAM, and storage capacity. These specifications will impact the performance results.

Software Environment: Set up the required software environment, including the programming language, libraries, and frameworks used in the implementation.

Data Set: Prepare a representative data set that simulates real-world scenarios. The data set should include a variety of files with different sizes and contents. This will allow for a comprehensive evaluation of the hash tree's performance across different scenarios.

**Methodology:**

Define Metrics: Determine the performance metrics to measure, such as the time taken for various operations (e.g., file insertion, deletion, searching, and integrity verification) and memory utilization.

Test Cases: Design a set of test cases that cover different scenarios, including varying file sizes, different numbers of files, and different types of operations. Ensure that the test cases are representative of the intended use cases for the hash tree.

Execution: Run the experiments by performing the defined operations on the hash tree using the test cases. Measure the execution time for each operation and record the memory utilization.

Repeat and Average: To ensure accurate results, repeat each experiment multiple times and calculate the average execution time. This helps to account for any variations caused by system factors or other external influences.

Comparative Analysis: Compare the performance of the hash tree implementation with individual data structures (e.g., arrays or linked lists) for the same set of operations. This can be done by implementing the same operations using individual data structures and measuring their performance using the same test cases and metrics.

**Reporting:**

1.Present the results in a clear and concise manner, including tables or graphs to illustrate the performance metrics and comparisons.

2.Discuss the findings, highlighting any significant performance differences between the hash tree and individual data structures.

3.Provide insights into the strengths and weaknesses of the hash tree implementation, based on the experimental results.

4.Discuss any limitations or potential sources of error in the experimental setup or methodology.

5.Conclude by summarizing the performance characteristics of the hash tree and its suitability for the specific problem domain.

**Discuss the datasets used and any specific considerations for the experiments.**

1.Dataset Composition: The datasets used should be representative of the expected real-world scenarios in which the hash tree will be utilized. It is important to include a variety of files with different sizes, contents, and characteristics. This allows for a comprehensive evaluation of the hash tree's performance across different use cases.

2.Dataset Size: The size of the datasets should be carefully considered to ensure that the hash tree can efficiently handle them. The size of the dataset can impact the performance of various operations, such as insertion, deletion, and searching. It is advisable to test the hash tree with both small-scale datasets and large-scale datasets to evaluate its scalability.

3.Distribution of Keys: The distribution of keys within the dataset can also affect the performance of the hash tree. It is beneficial to test the hash tree with datasets that have different key distributions, such as uniformly distributed keys, skewed distributions, or specific patterns. This helps in assessing how well the hash tree handles various types of data distributions and whether there are any performance variations.

4.Edge Cases: It is important to include edge cases in the datasets to evaluate the robustness and correctness of the hash tree implementation. For example, test cases with duplicate keys, empty datasets, or datasets with extreme values can be used to check the behavior of the hash tree under such conditions.

5.Specific Considerations: Depending on the specific requirements of the problem being solved, additional considerations may be necessary. For example, if the hash tree is used for file storage, the datasets should mimic different types of files (e.g., text files, images, or binary files) to assess the performance of the hash tree when handling diverse file formats.

**Present and interpret the results obtained from the experiments, including performance metrics and efficiency improvements.**

Interpreting the results obtained from the hash tree would involve comparing the performance metrics against alternative data structures, such as arrays, linked lists, or balanced search trees. If the hash tree consistently outperforms these structures in terms of insertion, lookup, and deletion times, as well as memory utilization, it can be concluded that the hybrid data structure with the hash tree component offers improved efficiency for the specific problem being addressed. Additionally, analyzing the scalability of the hash tree and its ability to handle various datasets and key distributions provides valuable insights into its performance in real-world scenarios.

**Performance Metrics:**

1.Insertion Time: The time taken to insert new elements into the hash tree can be measured. It indicates how efficiently the hash tree can handle data insertion.

2.Lookup Time: The time taken to search for a specific key in the hash tree can be measured. It reflects the efficiency of the hash tree in retrieving data.

3.Deletion Time: The time taken to remove a key from the hash tree can be measured. It represents the efficiency of the hash tree in deleting data.

4.Memory Utilization: The amount of memory utilized by the hash tree can be measured to assess its space efficiency. This includes both the storage needed for storing the keys and the additional overhead required by the data structure.

**Efficiency Improvements:**

1.Faster Retrieval: The hash tree provides efficient lookup time by utilizing the hash function to quickly locate the desired key. This can result in significant speed improvements compared to linear search or other data structures with slower retrieval times.

2.Space Optimization: The hash tree reduces memory overhead by dynamically allocating memory only for the keys that are present in the dataset. This can lead to efficient memory utilization compared to fixed-size data structures, such as arrays or linked lists.

3.Scalability: The hash tree's performance is often independent of the dataset size, as the lookup time remains relatively constant even as the number of keys increases. This scalability makes the hash tree suitable for large-scale datasets.

4.Collision Resolution: The hash tree handles collisions by using separate chaining or other collision resolution techniques. This ensures that even if multiple keys hash to the same index, the data structure can handle them efficiently and maintain good performance.

**7. Discussion**

**Discuss the practicality and effectiveness of the implemented hybrid data structure in real-world scenarios.**

1.Database Indexing: The hash tree can be used to efficiently index and retrieve data in databases. It allows for fast lookup of records based on key values, improving query performance and reducing the time required to search for specific data items.

2.Caching Systems: In caching systems, where frequently accessed data is stored in memory for quick retrieval, the hash tree can be employed to manage the cache. It enables rapid key-based lookup and eviction of least recently used items, optimizing cache hit rates and improving overall system performance.

3.File Deduplication: When dealing with large datasets that may contain duplicate files, the hash tree can be utilized to identify and eliminate redundant files. By calculating the hash of each file and comparing it with existing hashes stored in the hash tree, duplicates can be efficiently identified and removed, leading to storage space savings.

4.Distributed Key-Value Stores: The hash tree can serve as the underlying data structure in distributed key-value stores, where data is partitioned across multiple nodes. It allows for efficient data distribution and retrieval based on key values, facilitating high-performance and fault-tolerant storage systems.

5.Peer-to-Peer Networks: In peer-to-peer networks, where nodes collaborate to share resources and information, the hash tree can be used for efficient key-based lookup and storage of shared data. It enables quick retrieval of data items based on their unique identifiers, enhancing the overall performance of the network.

**Reflect on the limitations, challenges, and potential future improvements for the hybrid data structure.**

The limitations specific to the hash tree used:

1.Limited Key Size: The hash tree implementation may have a limitation on the size of the keys it can handle. If the keys exceed the maximum allowed size, it can lead to errors or incorrect behavior of the data structure.

2.Memory Consumption: The hash tree may consume a significant amount of memory, especially for large datasets or deep tree structures. This can be a limitation in memory-constrained environments or scenarios where memory usage needs to be optimized.

3.Hash Function Collision: Although hash functions are designed to minimize collisions, they are not completely collision-free. In rare cases, collisions can occur, leading to performance degradation or incorrect results in the hash tree.

4.Lack of Persistence: The hash tree implementation in the above code does not support persistence, meaning that the data structure and its contents are lost once the program terminates. This can be a limitation in scenarios where data persistence is required.

5.Limited Scalability: The performance of the hash tree may degrade as the dataset grows larger. The time complexity of operations such as insertion, deletion, and lookup can increase with the size of the dataset, potentially affecting the scalability of the data structure.

6.Inflexibility for Varying Workloads: The hash tree implementation may not be well-suited for scenarios with varying workloads. If the workload consists of frequent updates or dynamic changes to the dataset, the hash tree's performance may suffer due to the need for reorganization and rebalancing.

7.Lack of Ordering: The hash tree does not inherently provide ordering of the keys. If the application requires ordered traversal or retrieval of keys, additional mechanisms or modifications to the data structure may be necessary.

**8. Conclusion**

**Summarize the findings and outcomes of the project, highlighting the practical applications, performance analysis, and efficiency of the hybrid data structure**

The hybrid data structure leveraged the advantages of both data structures to achieve efficient key-value operations. Here is a summary of the findings and outcomes:

1.Practical Applications: The hybrid data structure is suitable for scenarios that require fast and efficient key-value lookups, insertions, and deletions. It can be applied in various domains such as database systems, caching mechanisms, indexing, and search algorithms.

2.Performance Analysis: The time and space complexity analysis of the code indicated that the hybrid data structure offers efficient operations. The hash table component provides O(1) average case lookup, insertion, and deletion, while the binary search tree component ensures ordered traversal and range queries with a complexity of O(log n).

3.Efficiency Improvement: The hybrid data structure overcomes the limitations of individual data structures by combining their strengths. It offers fast lookups with the hash table component and supports ordered traversal and range queries with the binary search tree component. This combination improves overall efficiency and performance compared to using either structure alone.

4.Experimental Setup: The performance of the hash tree, a key component of the hybrid data structure, was evaluated using appropriate datasets and methodologies. Specific considerations were taken into account, such as the size and characteristics of the datasets, to accurately measure performance metrics.

5.Results and Interpretation: The experimental results showcased the efficiency and effectiveness of the hash tree. The performance metrics demonstrated fast lookup, insertion, and deletion operations, as well as the ability to handle large datasets. The hybrid data structure outperformed individual data structures in terms of both time and space complexity.

6.Real-World Practicality: The hybrid data structure, with its combination of hash table and binary search tree, offers practicality in various real-world scenarios. It provides a balance between fast access and ordered traversal, making it suitable for applications that require both capabilities.

**Discuss the overall success of the project and any insights gained from its implementation and evaluation.**

The overall success of the above work lies in its ability to showcase the effectiveness of the hybrid data structure, provide insights into its performance characteristics, and offer practical applications in real-world scenarios. The work contributes to the broader understanding of data structures and their combinations, guiding future improvements and optimizations in this field.

**9. References**

**Cite any sources consulted or referenced during the project.**

\*\* INTERNET

GITHUB - <https://github.com/saileelathimmisetty/FILE-INDEXING>

**THANK YOU**