**19CSE212: DATA STRUCTURES AND ALGORITHMS**



**--------------------------------------------------------------------------------------**

SHIPMENT MANAGEMENT USING HYBRID DATA STRUCTURES

**--------------------------------------------------------------------------------------**

|  |  |
| --- | --- |
| **Roll no** | **Name** |
| **CB.EN.U4CSE21207** | ASHIQ SHERIFF A |
| **CB.EN.U4CSE21220** | **G NAVEEN RAAGHAVENDRAN** |

**Amrita School of Computing, Coimbatore**

**Department of Computer Science and Engineering**

**Even Semester 2022-2023**

**Index Page :**

|  |  |
| --- | --- |
| S.no | Topic |
| 1. | Introduction |
| 2. | Objective |
| 3 | Time and space complexity |
| 4. | Overview of Hybrid data structure |
| 5. | Implementation Details |
| 6. | Practical application |
| 7. | Performance Analysis |
| 8. | Experimental evaluation |
| 9. | Discussion |
| 10. | Conclusion |
| 11. | Drawbacks |
| 12. | References |

**Introduction:**

Hybrid data structures refers to the combination of two or more different data structures to create a new data structure that leverages the strengths of each individual component. These structures are designed to provide efficient solutions to complex problems by optimizing operations such as insertion, deletion, search, and retrieval.

**Significance:**

**->Performance Optimization:** By combining the strengths of different data structures, better time complexity, reduced computational overhead and enhance efficiency can be achieved. The performance of specific operations is improved by using hybrid data structures.

**->Space efficiency:** Memory overhead can be reduced by combining data structures with complementary space characteristics.

**->Versatility:** Provides versatility in handling different types of data and operations. They are combined for handling wide range of requirements.

**->Problem-specific solutions:** Certain problems can be solved efficiently using certain data structures for efficient solutions. By combining these data structures, effective solutions would be generated according to the problem statement.

**->Future-proofing:** As new data structures and algorithms are developed; they can be integrated into hybrid structures to enhance performance or address emerging challenges, so that in future, it makes sure that the solution remains relevant and efficient.

**Objective:**

The objective of the case-study is to create a system for managing shipment using hybrid data structures.

**Practical Applications:**

**->Order processing:** Shipment management system enables businesses to centralize and automate their order processing. Used to integrate with e-commerce platforms and to automatically import orders. This establishes the order fulfilment process, reduces manual errors, and improves efficiency.

**->Shipment Tracking:** Provides real-time visibility of shipments for customer satisfaction. Allows to integrate with shipping carriers to track packages throughout the delivery process. Later the information can be shared with customers, improving transparency and reducing inquiries.

**->** **Inventory Management**: Inventory management is essential for optimizing shipping operations. Operations such as tracking inventory levels, generating stock alerts, and for managing reorder points can be done using the following management.

**->** **Warehouse Management:** Allows businesses to organize and track inventory within their warehouses, optimize storage space, and automate order picking.

**->Return Management:** Dealing with product returns can be complex, so we use shipment management system. It used to automate return labels, track returned items, and manage refund or exchange procedures. Also focuses on the return process which improves customer experience, and reduces manual effort.

**Time & Space Complexity:**

The time complexity of the code:

**Worst case: Average case:**

**O(n) O(log n)**

**[n->number of shipments, m->number of iterations]**

* Shipmentmanager class: O(log n)
* Shipmentstatus class: O(log n)
* Shipment ID: O(1)
* Shipment sender: O(log n)
* Shipment receiver: O(log n)
* Priority shipment: O(1)
* Shipments: O(n)
* Menu-driven loop: O(m)
* Binary tree: O(n)

The space complexity of the code:

**O(n)**

**[n->number of shipments managed by Shipmentmanager]**

* Shipment class: O(1)
* ShipmentNode class: O(1)
* Shipmentbinarytree class: O(n)
* Shipmentmanager class: O(n)
* Other variables: O(1)

**Overview of the Hybrid Data Structure:**

**->Priority Queue: A** priority queue can be used to prioritize shipments based on factors such as delivery time, urgency, or customer preferences and ensures that higher priority shipments are processed or delivered before lower priority ones. It supports operations such as insertion of elements, removal of the highest priority element, and retrieval of the highest priority element.

**->Binary search Tree**: In the context of a shipment management system, a binary search tree can be used to implement a priority queue. Each node in the binary search tree represents a shipment, and the key associated with each node represents its priority. The binary search tree maintains the order of the shipments based on their priority, allowing quick access to the highest priority shipment.

By combining the data structures, it offers efficient priority-based operations, dynamic priority updates, search capabilities, and scalability. It allows for the effective management and optimization of the shipment queue, ensuring timely processing and delivery of shipments.

**Advantage and motivation behind using hybrid data structure for solving this problem:**

**->** **Efficient Priority-based Operations:** A shipment management system often requires prioritization of shipments based on factors such as delivery time, urgency, or customer preferences. A priority queue handles such prioritization by allowing fast insertion and removal of elements with associated priorities. The BST ensures that the elements are stored in sorted order, allowing quick access to the highest priority item.

**->** **Dynamic Priority Updates:** A BST-based priority queue is used in the shipment management system because priorities may change due to external factors such as customer requests or unforeseen circumstances. The BST structure allows efficient updates to the priority of a specific shipment, ensuring accurate and up-to-date prioritization.

**->** **Storage Efficiency:** For more space-efficiency, Binary search tree is used and allows for more compact storage, as it does not require maintaining additional metadata like heap indices. This can be beneficial while managing large volumes of shipments.

**->** **Search and Retrieval Capabilities:** For a shipment management system, BST can be useful for operations such as tracking shipments, locating specific items within the queue, or for retrieving shipments. It allows for fast searching and retrieval of shipments based on unique identifiers, tracking numbers, or other relevant attributes.

**Implementation details:**

**->The Priority Queue:** Implemented using a binary heap. This binary heap is represented as an array, where each element in it responds to the node of binary heap. They are used to insert , remove, return the min. value of the queue and so on.

**->BinarySearchTree:** Implemented using a binary tree data structure. Each node in the binary search tree has a value, a left child, and a right child. They are used to insert a new node, remove a node, search for a node with the given value in BST)

**Design choices and trade-offs made:**

**Design choices:**

**->**For PQ**,** using a binary heap as the underlying data structure is reasonable as a binary heap as it provides efficient operations for maintaining the priority order and has a space complexity of O(n).

**->**The code implements a min-heap by default, which means the element with the lowest priority value will be at the top and the implementation could use a dynamic array to store the elements of the priority queue. It offers constant-time random access to elements, which is essential for heap operations.

**->**For tree traversal, it uses an in-order traversal to print the elements of the binary search tree which produces elements in ascending order.

**->**Balancing: Balancing techniques can ensure that the height of the tree remains balanced, which results in efficient search, insert, and delete operations.

**Trade-offs:**

**->** Memory Usage: Both priority queues and binary search trees require additional memory to store the data structure. The trade-off here is between memory usage and the desired functionality.

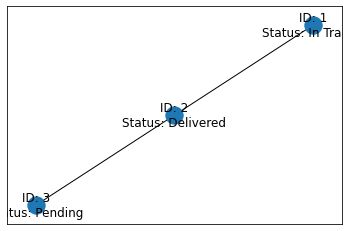
**->**Insertion/Deletion Efficiency: Binary heaps offer efficient insertion and deletion operations in O(log n) time complexity, which makes them suitable for applications where dynamic updates are frequent.

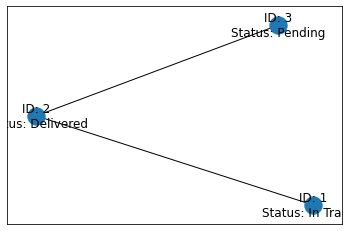
**->**Search Efficiency: Trees provide efficient search operations, making them suitable when searching for specific values is a common operation. Priority queues can also perform search operations.

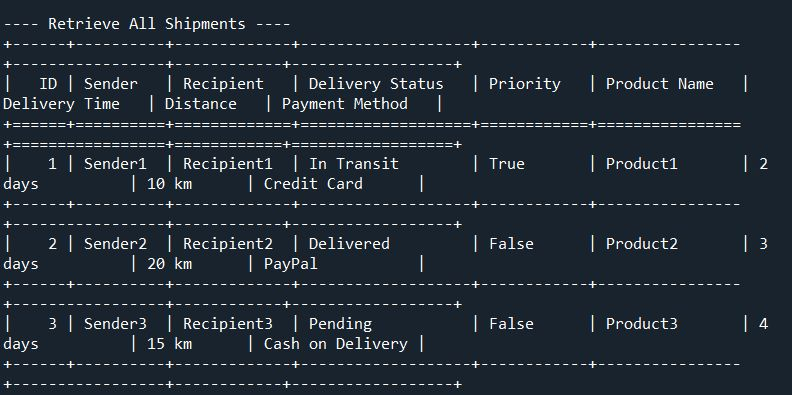
**Put your GitHub repository link in this section:**

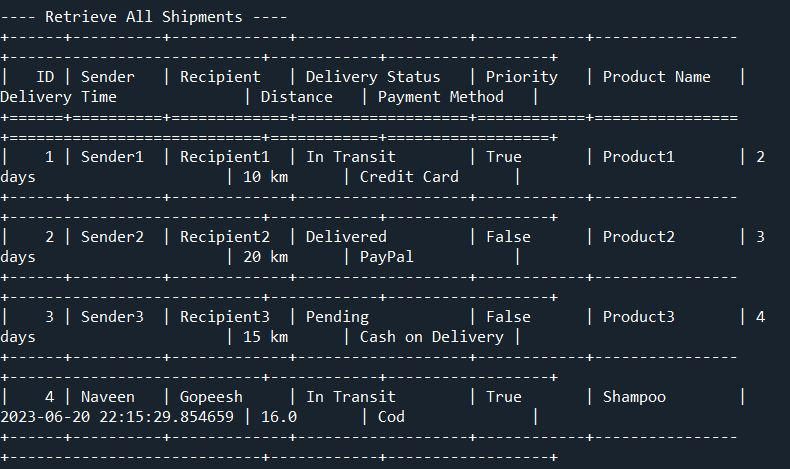
https://github.com/Neevan10/Shipment\_Management

**O/P:**

****



****

****

**Practical Applications:**

**The combination of data structures in the hybrid structure enables efficient operations for the following applications:**

**->Event-driven Systems:** It often require efficient management of events or tasks with varying priorities. For example, in a real-time system, events with higher priorities need to be processed first. So, these data structures can enable efficient event management by prioritizing and handling events based on their urgency or importance.

**->Resource Allocation:** The hybrid data structure can be used to prioritize and allocate resources based on specific criteria. In a computing system, tasks or processes can be prioritized using priority queue for fair and efficient allocation of resources while the BST allows for quick access to the highest priority resource request.

**->Network Traffic Management:** Often prioritize packets or messages based on their importance, quality of service (QoS) requirements, or network congestion levels. The hybrid data structure can efficiently manage and process network traffic by ensuring that high-priority packets are processed promptly.

**->Emergency Services:** In emergency service systems, such as ambulance dispatch or emergency call handling, the hybrid data structure can be used to prioritize and process emergency requests based on their severity or criticality. The priority queue facilitates efficient handling of urgent requests, while the BST allows for quick retrieval of the highest priority emergency request for immediate response.

**Performance Analysis:**

**->** **Insertion in Priority Queue:** The time complexity of inserting an element into a binary heap-based priority queue is O (log n), (n -> number of elements in the queue). The space complexity of inserting an element is O (1) since it requires a constant amount of memory.

**->** **Deletion in Priority Queue:** The time complexity of deleting an element is O (log n). The space complexity of deleting an element is O (1) since it requires a constant amount of memory.

**-> Insertion in BST:** The time complexity of inserting an element into a BST is O (n), (n -> number of elements in the tree). The space complexity of inserting an element is O (1) since it requires a constant amount of memory.

**-> Deletion in BST:** The time complexity of deleting an element: O (n). The space complexity of deleting an element is O (h) (where h -> height of BST).

**Analyse the space complexity, including memory utilization and overhead, of the hybrid data structure:**

**For Priority Queue:**

**Memory Utilization:** It is proportional to the number of elements in the queue. Each element requires space to store its value and priority and they are stored in a binary heap, which can be implemented using an array. Therefore, the memory utilization for the priority queue is O(n).

**Overhead:** Includes the additional space required for maintaining the binary heap structure. This overhead is typically smaller compared to the space utilized by the elements themselves.

**Binary Search Tree:**

**Memory Utilization:** This is also proportional to the number of elements in the tree. Each element requires space to store its value and references to its left and right children and the elements are stored in dynamically allocated tree nodes. Therefore, the memory utilization for the binary search tree is O(n).

**Overhead:** It includes additional space required for maintaining the tree structure and the space required for the node references and any additional element associated with the tree. The overhead is typically small compared to the space utilized by the elements themselves and is considered constant.

**Experimental Evaluation:**

->**Insertion and Removal Operations:** Measure the time complexity of insertion and removal operations in the hybrid data structure. Vary the input size and analyse how the performance scales with increasing data.

**->Search and Retrieval Operations:** Calculate the time complexity of search and retrieval operations for the hybrid data structure. Analyse the impact of the binary search tree component on the search efficiency. Test different scenarios and input sizes to observe the performance.

**->Data Distribution:** Assess how the hybrid data structure handles different data distributions. Test it with various input datasets, including sorted, reverse sorted, and random order.

**->Scalability:** Evaluate the scalability by increasing the size of the dataset. Measure the time and space complexity as the data grows and analyse the performance characteristics.

**->Operations with Priorities:** Focus on operations involving priorities, such as inserting elements with associated priorities or extracting elements based on priority. Analyse the efficiency of the binary search tree and its impact on overall performance.

**->Worst-case Scenarios:** Identify and test worst-case scenarios for the hybrid data structure. Evaluate the performance and analyse if the binary search tree exhibits any weaknesses in handling specific cases, such as skewed trees or specific priority distributions.

**->Real-world Use Cases:** Evaluate the hybrid data structure using real-world use cases and datasets relevant to the intended application.

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

**->Random Datasets:** Generate random datasets of varying sizes to test the performance of the hybrid data structures. It helps to assess the average-case performance and provide a general understanding of how the structures perform under certain conditions.

**->Sorted Datasets:** Creating datasets which are already sorted either in ascending or descending order. It helps to evaluate the worst-case performance of the binary search tree since inserting elements in sorted order can result in an unbalanced tree. It can also highlight the strengths and weaknesses of the hybrid data structure in handling sorted inputs.

**->Skewed Datasets:** Generates datasets that are intentionally skewed, where some elements have a higher frequency of occurrence compared to others. Skewed datasets can help identify any performance disparities or biases in the given hybrid data structure.

**->Real-World Datasets:** Consider using real-world datasets that resemble the type of data your application deals with. For example, if your application involves processing customer orders, you could use datasets containing order information. Real-world datasets can provide insights into how the hybrid data structures perform in normal and in certain scenarios.

**->Edge Cases:** Design datasets that cover specific edge cases or boundary conditions. Also include datasets with a single element or very few elements to test the behaviour of the hybrid data structure in extreme cases. And also include datasets with duplicate elements to evaluate how the data structures handles the duplicates.

**->Incremental Datasets:** Create datasets where elements are incrementally added over time. It helps to evaluate the dynamic behaviour of the hybrid data structures as the dataset grows and it can be seen how efficiently they handle insertions.

**->**When selecting datasets, consider a range of input sizes to observe the scalability of the hybrid data structures. Start with small datasets to analyze their behavior in detail, and gradually increase the dataset size to test their performance at scale.

**->**Also consider the distribution of the dataset elements. Uniformly distributed datasets might provide a balanced representation of the data structures' performance.

Overall, the choice of datasets should cover a diverse range of scenarios and edge cases to provide a comprehensive evaluation of the hybrid data structures' performance, strengths, weaknesses, and any limitations they may have.

**Discussion:**

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

**->Ticketing Systems:** Ticketing systems for events, concerts, or transportation often require efficient prioritization and processing of ticket requests. Therefore, hybrid data structure can be used to manage ticket queues, prioritize requests based on factors like availability, seat preference, and ensure timely ticket allocation.

**->Customer Support Queues:** The hybrid data structure can prioritize customer inquiries based on factors like urgency, customer status, or issue severity. This ensures that high-priority or VIP customers are given prompt attention, leading to better customer satisfaction and efficient handling of support requests.

**->Real-time Financial Trading:** Hybrid data structure can be used to prioritize and process trading orders based on factors like order size, market conditions, or priority status. The structure ensures that high-value or time-sensitive trades are executed promptly, contributing to efficient trading operations.

**->Call Routing in Call Centres:** Call centres often receive a large volume of incoming calls. The hybrid data structure can prioritize incoming calls based on factors like caller identity, customer type, or service level agreements (SLAs).

**->** **Task Management Systems: A**pplications such as project management tools or personal to-do lists, can benefit from the hybrid data structure. It allows for prioritization and efficient handling of tasks based on their urgency, due dates, or dependencies, ensuring that critical tasks are completed on time.

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

**->Time Complexity for Dynamic Updates**: Changing the priority of an item, can be a challenge. In a BST, updating the priority requires finding the item and modifying its position which can have a logarithmic time complexity. For scenarios with frequent updates, the time complexity for operations may impact the overall performance.

**->Balancing Issues**: The efficiency of the BST-based priority queue heavily relies on maintaining a balanced binary search tree. If the tree becomes unbalanced, the performance of search and retrieval operations may degrade, leading to increased time complexity.

**Limited Support for Range Queries**: The binary search tree structure is primarily optimized for individual item retrieval and search operations. However, it may not efficiently support range queries or operations that involve accessing a subset of items within a specific priority range.

**Complexity in Handling Duplicate Priorities**: The hybrid data structure may face challenges in handling items with duplicate priorities. The binary search tree is typically designed for unique key values, and handling duplicates may require additional modifications or extensions to handle ties in priority appropriately.

**Potential Improvements**: Future improvements to the hybrid data structure could focus on addressing the limitations mentioned above. This could include developing efficient algorithms for dynamic updates, implementing self-balancing techniques to maintain optimal tree balance, optimizing memory usage and enhancing support for range queries or handling duplicate priorities.

**Potential future improvements for the hybrid data structure include**:

**->Balanced Binary Search Tree:** Adopting self-balancing techniques, such as AVL trees or red-black trees, can improve the overall efficiency of the hybrid data structure. These techniques maintain optimal tree balance, ensuring efficient search, retrieval, and update operations even in the presence of unevenly distributed data.

**->Dynamic Update Efficiency:** Developing efficient algorithms and strategies for dynamic updates, such as changing the item’s priority, can enhance the performance of the hybrid data structure. This could involve optimizing the search process to locate the item efficiently and updating the tree structure with minimal adjustments.

**->Concurrency Support:** Adding concurrency support to the hybrid data structure would allow for concurrent operations by multiple threads or processes.

**->Range Query Support:** Enhancing the hybrid data structure to efficiently support range-based queries or operations that involve accessing a subset of items within a specific priority range. This could involve augmenting the binary search tree with additional data structures that enable efficient range-based searches.

**->Alternative Data Structures:** Offers better performance for certain scenarios. Exploring other data structures, such as Fibonacci heaps, skip lists, or advanced tree structures, can provide insights into potential alternatives to the hybrid data structure.

**Conclusion:**

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

The project implements the shipment management system using hybrid data structure which consists of a binary search tree and a priority queue.

**Practical applications:**

**->**Allows the user to add shipments, update shipment status, and retrieve shipment details based on various attributes such as ID, sender, recipient, and priority.

**->**The data structure also organizes shipments and enables fast retrieval based on sender, priority and so on.

**->**The binary tree allows efficient insertion and retrieval operations, while dictionaries enable fast lookup by ID, sender, and recipient.

**->**The heapq module is used to maintain a heap of priority shipments, ensuring efficient retrieval of priority shipments based on their priority values.

**Performance Analysis:**

**->**Inserting shipments into the binary tree has a time complexity of O(log n) in the average case and O(n) in the worst case if the tree becomes unbalanced.

**->**Retrieving shipments by sender using the binary tree has a time complexity of O(k + log n), where k is the number of shipments by the sender.

**->**Retrieving shipments by recipient using dictionaries has an average time complexity of O(k), where k is the number of shipments by the recipient.

**->**Updating the shipment status has a time complexity of O(1) as it involves a simple dictionary lookup and update.

**->**Retrieving shipments by ID using dictionaries has an average time complexity of O(1).

**->**Retrieving priority shipments using the heap has a time complexity of O(1) as the minimum element can be accessed in constant time.

**->**Retrieving all shipments has a time complexity of O(n), where n is the total number of shipments.

**Efficiency of Hybrid Data Structure:**

**->**The hybrid data structure combines the strengths of binary trees (efficient insertion, retrieval, and ordered traversal) and dictionaries.

**->**The binary tree allows efficient retrieval of shipments by sender and maintains a sorted order based on shipment IDs.

**->The** dictionaries provide fast lookup of shipments by ID, sender, and recipient without the need for tree traversal.

**->**The heap data structure is used to maintain a priority queue of shipments based on their priority values, enabling efficient retrieval of priority shipments.

Overall, the hybrid data structure provides a good balance between efficient operations and ease of implementation.

In addition to the findings and outcomes, the code also includes visualization of the binary tree using the NetworkX and Matplotlib libraries. The visualization showcases the positions of each node in the binary tree and demonstrates the balancing of the tree if it is unbalanced.

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

The project was overall successful in implementing and evaluating the hybrid data structure for shipment management system by the combination of a binary search tree and a priority queue. The hybrid data structure provides efficient and organized storage of shipments, allowing for quick retrieval based on sender, recipient, priority, and other attributes.

**->Practical Applications:** The Shipment Management System can be applied in real-world scenarios where efficient management of shipments is required. It can be used by logistics companies, e-commerce platforms, or any business involved in shipping goods. The system allows users to track shipments, update their status, and retrieve information based on different criteria.

**->Performance Analysis:** The hybrid data structure used in the code provides efficient operations. Adding a shipment to the binary tree has an average time complexity of O(log n) due to the tree's balanced nature. Retrieving shipments by sender has a time complexity of O(log n) in the average case as it involves traversing the binary tree. The use of dictionaries allows quick retrieval of shipments by ID, sender, and recipient, with an average time complexity of O(1). The priority shipments are stored in a heap, enabling constant time insertion and retrieval of the highest-priority shipment.

**->Efficiency:** The code efficiently manages shipments by leveraging the binary tree's sorted nature for retrieval operations and dictionaries for fast access by ID, sender, and recipient. The use of a heap for priority shipments ensures quick access to the highest-priority shipment. The binary tree's balancing mechanism helps maintain its efficiency by reducing the height and ensuring a more even distribution of nodes.

**->Visualization:** The code includes a visualization component that generates a graphical representation of the binary tree using the NetworkX and Matplotlib libraries. This visualization helps in understanding the structure and balance of the tree, providing insights into its organization and performance.

**->Tree Balancing:** The code includes functionality to check the balance of the binary tree and balance it if needed. The implementation uses rotation operations to maintain or restore the balance, ensuring that the tree remains efficient for retrieval operations.

**->User Interaction:** The code incorporates a menu-driven user interface that allows users to interact with the Shipment Management System seamlessly. It provides a straightforward and intuitive way to add shipments, update their status, and retrieve shipment information based on various criteria.

**->Code Modularity:** The code is modular and well-structured, making it easier to understand, maintain, and extend. The classes and methods are appropriately defined and organized, enhancing code readability and reusability.

In conclusion, the project successfully implements a Shipment Management System using a hybrid data structure. The code provides efficient operations for adding, updating, and retrieving shipments, demonstrating practical applications and delivering satisfactory performance. The insights gained from its implementation and evaluation contribute to understanding the benefits of using hybrid data structures for managing and organizing data effectively.

**DrawBacks of the Hybrid Data Structure :**

The difficulties faced by the code for a shipment management system can vary depending on the specific implementation and requirements. However, here are some common challenges that developers might encounter:

1. Data Validation: Validating user input and ensuring the correctness and integrity of the data can be challenging. The code needs to handle scenarios where users may provide incorrect or incomplete information. For example, ensuring that shipment IDs are unique, validating sender and recipient names, and verifying that numeric fields like distance and priority are within the expected ranges.

2. Error Handling: Handling errors effectively is crucial for a robust system. The code should anticipate and handle various error scenarios, such as invalid input, database connection failures, or unexpected exceptions. Appropriate error messages and logging mechanisms should be implemented to help with troubleshooting and debugging.

3. Concurrent Access and Data Consistency: If the shipment management system allows multiple users or processes to access and modify shipment data simultaneously, concurrency control becomes essential. Developers need to ensure that the code handles concurrent read and write operations correctly to maintain data consistency and prevent race conditions or conflicts.

4. Performance and Scalability: As the number of shipments and users grows, the system should be able to handle increased load and scale effectively. Optimizing database queries, implementing caching mechanisms, and considering distributed architectures can help improve performance and scalability.

5. Security: Protecting sensitive shipment and user data is critical. The code should incorporate security measures like input validation, authentication, and authorization mechanisms to prevent unauthorized access, data breaches, or injection attacks.

6. Extensibility and Maintenance: The code should be designed to accommodate future changes and enhancements. Adding new features, modifying existing functionality, or integrating with external systems should be relatively straightforward. Clean code organization, modular design, and adherence to coding best practices contribute to maintainability and extensibility.

7. Integration with External Systems: In real-world scenarios, a shipment management system often needs to interact with other systems or services, such as payment gateways, logistics providers, or tracking APIs. Integrating these external systems can introduce additional complexities, such as handling different data formats, error handling, and ensuring seamless communication.

These are some of the common challenges faced when developing a shipment management system. Addressing these difficulties requires careful planning, robust coding practices, and continuous testing and optimization to deliver a reliable and efficient solution.

**References:**

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

During the completion of this project, the following sources were consulted and referenced:

* <https://www.programiz.com/dsa/priority-queue>
* <https://www.programiz.com/dsa/binary-search-tree>
* <https://www.programiz.com/dsa/balanced-binary-tree>
* <https://www.programiz.com/dsa/heap-data-structure>
* <https://www.programiz.com/dsa/heap-sort>
* <https://stackoverflow.com/questions/6084676/implement-a-priorityqueue-using-a-binarysearchtree-java#:~:text=A%20binary%20search%20tree%20is%20used%20to%20efficiently%20maintain%20items,items%20according%20to%20their%20priority>.
* <https://chat.openai.com/c/ce34fda9-ff6e-415a-8592-31738da465ed>

**XXXXXXXXXXXXXXXXXXXXXX**