Possible Questions:

1. **How does your chosen data structure support the CRUD operations in your system?**

Our chosen data structure, **JSON,** excellently supports CRUD operations in our hotel booking system. For **Create operations, we utilize the POST HTTP method,** allowing us to easily add new JSON objects representing user registrations, room reservations, and guest reviews. The flexible nature of JSON allows us to then incorporate new data fields as needed. **Read operations are efficiently handled using the GET HTTP method,** where we can quickly query and retrieve JSON data for room details, reservation information, and user profiles. The nlohmann JSON library's querying capabilities enable complex searches, such as finding available rooms for specific dates. **Update operations use the PUT/PATCH HTTP methods**, allowing us to modify existing JSON objects for reservation changes, profile updates, and administrative tasks like updating room availability. The tree-like structure of JSON makes it easy to navigate and modify nested data. **Finally, Delete operations use the DELETE HTTP method** to remove JSON objects representing cancelled reservations or deleted user accounts. The JSON-based storage system allows for efficient removal of data while maintaining the integrity of the remaining structure. This is done and managed fully in C++ using the said external library, which does unfortunately mean that it is not supported in the Standard Template Library.

2. **Can you explain how the conceptual framework aligns with the principles of data structures we've covered in class?**

Our conceptual framework aligns closely with several of the key principles of data structures covered in our Data Structures and Algorithms course you've taught us. This is mainly because, at its core, our system utilizes the tree-like structure inherent to JSON, which mirrors the hierarchical tree data structures we've studied. This allows for efficient traversal and manipulation of nested data, such as room details within a hotel or amenities within a room. We've applied the principle of abstract data types by encapsulating our data operations within custom C++ objects, which are then serialized to and deserialized from JSON. This abstraction allows us to work with complex data types while maintaining a clean interface. The use of hash table-based indexing for quick lookups demonstrates our application of hash table principles, achieving average-case O(1) time complexity for frequent operations. Our implementation of B-tree indexing for certain JSON fields showcases the application of balanced tree structures, enabling efficient range queries and sorted traversals. The LRU (Least Recently Used) caching algorithm we've employed for frequently accessed JSON objects applies the queue data structure concept, optimizing for both temporal and spatial locality. Furthermore, our use of bitset operations for quick filtering of room attributes demonstrates the application of bit manipulation techniques often associated with low-level data structure optimizations.

3. **In what ways does your project demonstrate the efficiency concepts we've discussed in relation to data structures?**

Well primarily, we've implemented hash table-based indexing for key fields such as room numbers and booking IDs, which allows for O(1) average-case time complexity for lookups. This showcases the efficiency gains possible with proper hash function implementation and collision resolution strategies. We've also employed B-tree indexing for certain JSON fields, enabling logarithmic-time range queries and sorted traversals. This demonstrates the efficiency of balanced tree structures for operations that require maintaining sorted order. Again, the use of the LRU caching algorithm for frequently accessed JSON objects optimizes access times for commonly requested data. The implementation of bitset operations for quick filtering of room attributes shows how low-level bit manipulation can lead to significant performance improvements too. Our JSON diff and patch algorithm for efficient updates demonstrates how we can minimize data transfer and processing overhead by only transmitting and applying changes. The use of the nlohmann JSON library's querying capabilities, supplemented by custom query methods, allows for efficient data retrieval even in complex nested structures. Lastly, our multi-version concurrency control (MVCC)

algorithm showcases how we can achieve efficiency in a multi-user environment by allowing non-blocking reads while ensuring data consistency during updates.

Minor Questions:

4. **How do you plan to handle data persistence in your system, and which data structures are involved?**

We handle data persistence using JSON files for storage, with in-memory caching for frequent access. We use hash tables for quick lookups, LRU for cache management, and implement MVCC for concurrent operations.

5. **Can you elaborate on how you'll implement searching and sorting functionalities in your project?**

We implement searching using hash table lookups for exact matches. This is known to be the fastest search lookups in most web server applications. We also use binary search with bitset operations for complex queries, and trie structures for text searches. Sorting leverages B-tree indexing and comparison-based algorithms like QuickSort.

6. **What considerations did you make when choosing between arrays, linked lists, or other structures for data storage? Why did you use a tree?**

Briefly, JSON is inherently a tree structure in both by the way it's programmed and the way it's used. We chose JSON over arrays or linked lists due to its hierarchical nature, flexibility, and ease of serialization. We supplement it with hash tables for indexing and vectors for list storage where appropriate.

7. **You said you used a tree structure because it's inherent to JSON. Then, how does your project utilize tree or graph structures, if at all?**

We utilize tree structures through JSON's hierarchical nature, B-tree indexing for efficient queries, tries for text searches, and graphs for room recommendations. There are things faster to operate on trees, and especially for example when adding add-ons to bookings, this hierarchical structure makes it sure that the add-on flag takes precedence before adding add-ons so the program knows to check and account for it.

8. **In what ways does your proposed UI/UX design reflect the underlying data structures?**

Our UI/UX reflects the data structures through component-based architecture mirroring JSON structure, tree-like navigation, trie-powered autocomplete, and visualizations based on our data organization. **Essentially, we want to also showcase the four tenets of Object-Oriented Programming too. We want the experience to be seamless for users, full-fledged for admins, and easy-to-understand for coders.**

9. **How do you plan to optimize the time complexity of your CRUD operations?**

We have made no cuts in time-budgeting for optimization especially. We realized that local servers may operate very fast, but only if the server can handle requests asynchronously. We also optimize time complexity of CRUD operations through strategic use of data structures. Create and Delete operations leverage JSON's flexibility for O(1) insertion and removal. Read operations use hash table-based indexing for O(1) average-case lookups on key fields. For complex reads, we employ B-tree indexing, ensuring O(log n) time complexity for range queries. Update operations combine hash table lookups with efficient JSON manipulation, maintaining O(1) average-case complexity for most updates.

10. **Can you explain how you'll implement error handling and data validation using appropriate data structures?**

We implement error handling and data validation using a combination of JSON schema validation and custom validation rules. We use a trie-based structure to efficiently store and check against allowed values for enumerated fields. This is where one of the benefits of using tree structures comes in handy, as it cannot move to the next node if we do not clear the current one. For complex validations, we employ a decision tree

structure, allowing for quick traversal of validation rules. Error messages are stored in a hash table for O(1) lookup and quick feedback to users. This is either recorded in the client's console, the server's, or both.

11. **What role do algorithms play in your project beyond the basic CRUD operations?**

Algorithms play absolutely crucial roles beyond just the CRUD operations. We use graph algorithms for room recommendations, shortest path algorithms for mapping hotel amenities, and machine learning algorithms for price prediction. Scheduling algorithms optimize room assignments, while string matching algorithms enhance search functionality. We also employ compression algorithms for efficient data storage and transfer. It's not only for optimizations. CRUD doesn't include type checking. We have found that, in general, the nlohmann JSON library uses generalizations in the way it accepts it's data. That means, for example, that casting '2' will be returned as int, but '2.51' will be returned as float or double. We can always static cast all of the data, but this will significantly slow down our process. Instead, we use checking algorithms (similar to how templates are used) to parameterize the data and then operate depending on what we expect the data to be.

12. **How does your project demonstrate the practical application of stack or queue data structures?**

Admittedly, we don't use either the stack or queue data structure explicitly in our project, which may be included in our limitations. However our project uses stack-like logic for managing undo/redo functionality in the booking process. Queues are utilized in our task scheduling system for managing housekeeping and maintenance tasks. We also implement a priority queue-like for handling super administrator requests. In application, one could implement a circular list for rotational promotions on coupons that our program has already implemented, but that is out of our scope.

13. **In what ways does your proposed system scale, and how do the chosen data structures support this?**

Our system scales through distributed data storage and processing. The JSON structure allows for easy sharding of data across multiple servers. We use consistent hashing for distributing data, ensuring efficient scaling of our hash table-based indexing. B-tree indexing supports scalability by maintaining logarithmic-time operations even as data grows. Our caching system uses a distributed LRU algorithm to scale across multiple nodes. JSON isn't the best at scaling serverwise, but it is known to be easy to understand, and in and up-to medium-scale companies, it may scale fairly well.

14. **How do you plan to balance memory usage and processing speed in your implementation?**

We balance memory usage and processing speed through careful data structure choices. Our JSON storage is compressed to reduce memory footprint. We use lazy loading techniques already implemented in Next.js. The LRU caching algorithm optimizes memory usage while maintaining speed for common operations. For large datasets, we implement memory-mapped files, balancing between RAM usage and fast access to disk-stored data.

15. **Can you describe a scenario where you might need to switch from one data structure to another as your project evolves?**

That is a valid question. One of the most important is the switch to MySQL. JSON is more popular to be done on independently-deployed applications where all the storage is done client-side. MySQL is simply better at handling bigger volumes of data. As our project scales, we might switch from a simple B-tree to a more distributed structure like a Log-Structured Merge-tree (LSM-tree) for our indexing needs. This would better handle high write volumes while maintaining good read performance. We might also transition from in-memory hash tables to distributed hash table implementations to handle larger datasets across multiple servers.

16. **How does your project incorporate the concept of abstract data types?**

We use abstract data types (ADTs) to encapsulate the complexity of our data structures. For example, our Room and Booking classes are implemented as ADTs, providing a clean interface for operations while hiding the underlying JSON representation. This abstraction allows us to modify the internal implementation without affecting the rest of the system.

17. **In what ways does your project demonstrate the trade-offs between different data structures we've studied?**

Our project demonstrates trade-offs in several ways. Using JSON for flexibility comes at the cost of larger storage compared to fixed-structure arrays. Hash tables provide O(1) average-case lookups but require additional memory for the hash map. B-trees offer balanced performance for both reads and writes but are more complex to implement than simple binary search trees. There may also be better ways to implement these, and may be a must for a larger-scale distribution of this server, but that is also out of the scope of the project.

18. **How do you plan to measure and optimize the performance of your data structure implementations?**

The Visual Studio IDE already provides us with memory usage and performance indicators server-side. We can monitor the website data and memory usage in the website console. Typically, during our runtime the server averages only a 2MB memory usage.

– PLEASE DO NOT MEMORIZE THIS –
THIS IS ONLY MEANT AS A GUIDE.