Sure! Let's structure the **Amazon Locker System** code and explanation similarly to the **Amazon Transportation System** you provided. This will cover all aspects of the locker system while ensuring clarity, organization, and an understanding of the relationships between classes.

1. **Problem Statement: Designing Amazon Locker System**

The Amazon Locker system allows users to store and retrieve packages securely. It manages locker assignments, ensures efficient package retrieval, and handles different package sizes, allowing for scalability and maintainability.

---

2. **System Requirements and Solutions**

1. **Locker Management**
    - *Solution*: The `Locker` class represents a locker that can store packages. It has properties like size and availability.
2. **Package Handling**
    - *Solution*: The `Package` class defines the attributes of packages, including size and ID, enabling efficient storage and retrieval.
3. **Package Assignment**
    - *Solution*: The `PickupLocation` class manages lockers and handles the assignment of packages to available lockers based on size.
4. **Package Retrieval**
    - *Solution*: Packages can be retrieved from their assigned lockers, ensuring users can access their items easily.
5. **Traffic Management**
    - *Solution*: The system prioritizes larger packages for lockers during peak times, ensuring better space utilization.

---

3. **Key Classes, Functions, and Relationships**

1. **Class:** `Package`
    - Represents a package with size and ID attributes. Provides methods to access its size and ID.
2. **Class:** `Locker`
    - Represents a locker with a size and manages the storage of a package.
3. **Class:** `PickupLocation`
    - Manages multiple lockers and their availability. Assigns packages to lockers based on their size and retrieves them.

---

4. **Data Structures Used**

- **Map** (in `PickupLocation`):
  - Used to map package IDs to their respective lockers, enabling quick access to retrieve packages.
  - **Complexity**: O(1) for lookups and inserts.
- **Queue** (for available lockers):
  - Used to efficiently manage available lockers of different sizes.
  - **Complexity**: O(1) for insertions and O(n) for lookups based on queue size.

---

5. **High-Level Design**

- **Package**: Represents the package with attributes like size and ID.
- **Locker**: Manages the storage and retrieval of packages.
- **PickupLocation**: Oversees the assignment of packages to lockers and handles retrieval processes.

---

6. **Clarifications to Ask the Interviewer (with Solutions)**

1. **What types of packages will be stored?**
   - *Solution*: If there are multiple package sizes (e.g., small, medium, large), the system should support them effectively by prioritizing their placement.
2. **Should the system support user notifications?**
   - *Solution*: If required, notifications can be added for users when their package is ready for pickup.
3. **Are there limits on locker usage?**
   - *Solution*: If there are limits, the system should handle locker capacity and manage overflow scenarios efficiently.

---

7. **Open-Ended Questions from the Interviewer**

1. **How would you scale the system for thousands of lockers?**
   - *Solution*: Implement a distributed system for managing lockers across multiple locations, utilizing microservices for scalability.
2. **How would you manage expired packages in lockers?**
   - *Solution*: Introduce a timestamp for each package, and periodically check for expired packages for automatic retrieval or notification.

---

8. **C++ Code Implementation**

```cpp
#include <iostream>
#include <unordered_map>
#include <queue>
#include <string>

using namespace std;

enum Size { SMALL, MEDIUM, LARGE }; // Enum for package sizes

// Class representing a package
class Package {
public:
    Size size; // Size of the package
    string id; // Unique identifier for the package

    Package(Size sz) : size(sz), id(to_string(rand())) {} // Generate random ID

    Size getSize() const {
        return size; // Return package size
    }

    string getId() const {
        return id; // Return package ID
    }
};

// Class representing a locker
class Locker {
private:
    Size size; // Size of the locker
    string id; // Unique identifier for the locker
    Package* storedPackage; // Pointer to the stored package

public:
    Locker(Size sz) : size(sz), storedPackage(nullptr), id(to_string(rand())) {}
// Initialize locker

    void storePackage(Package* package) {
        storedPackage = package; // Store package in the locker
    }

    Package* releasePackage() {
        Package* package = storedPackage; // Release the stored package
        storedPackage = nullptr; // Clear locker
        return package;
    }

    bool isEmpty() const {
        return storedPackage == nullptr; // Check if locker is empty
    }

    Size getSize() const {
        return size; // Return locker size
```

```
    }

    string getId() const {
        return id; // Return locker ID
    }
};

// Class managing locker assignments
class PickupLocation {
private:
    unordered_map<Size, queue<Locker*>> availableLockers; // Map of available
lockers by size
    unordered_map<string, Locker*> packageMap; // Map of package ID to locker

public:
    PickupLocation(unordered_map<Size, int> lockerSizes) {
        // Initialize lockers for each size
        for (const auto& entry : lockerSizes) {
            Size lockerSize = entry.first;
            int count = entry.second;
            queue<Locker*> lockerQueue;
            for (int i = 0; i < count; i++) {
                lockerQueue.push(new Locker(lockerSize)); // Create lockers
            }
            availableLockers[lockerSize] = lockerQueue; // Store in the map
        }
    }

    Locker* assignPackage(Package* package) {
        // Assign package to an available locker based on size
        for (Size sz = package->getSize(); sz <= LARGE; sz = static_cast<Size>(sz
+ 1)) {
            if (!availableLockers[sz].empty()) {
                Locker* locker = availableLockers[sz].front(); // Get available
locker
                availableLockers[sz].pop(); // Remove from queue
                locker->storePackage(package); // Store package
                packageMap[package->getId()] = locker; // Map package ID to locker
                return locker;
            }
        }
        return nullptr; // No locker available
    }

    Package* retrievePackage(const string& packageId) {
        if (packageMap.find(packageId) == packageMap.end()) return nullptr; //
Package not found

        Locker* locker = packageMap[packageId]; // Find locker for package
        Package* package = locker->releasePackage(); // Retrieve package
        availableLockers[locker->getSize()].push(locker); // Return locker to
available pool
        packageMap.erase(packageId); // Remove from package map
        return package;
```

```
    }

    ~PickupLocation() {
        // Clean up dynamically allocated lockers
        for (auto& entry : availableLockers) {
            while (!entry.second.empty()) {
                delete entry.second.front(); // Delete locker
                entry.second.pop(); // Pop from queue
            }
        }
    }
};

int main() {
    unordered_map<Size, int> lockerSizes = {
        { SMALL, 2 },
        { MEDIUM, 2 },
        { LARGE, 1 }
    };

    PickupLocation location(lockerSizes); // Create pickup location

    // Create packages
    Package* package1 = new Package(SMALL);
    Package* package2 = new Package(MEDIUM);
    Package* package3 = new Package(SMALL);

    // Assign packages to lockers
    Locker* locker1 = location.assignPackage(package1);
    Locker* locker2 = location.assignPackage(package2);
    Locker* locker3 = location.assignPackage(package3);

    // Output assignments
    if (locker1) cout << "Package 1 assigned to locker: " << locker1->getId() <<
endl;
    if (locker2) cout << "Package 2 assigned to locker: " << locker2->getId() <<
endl;
    if (locker3) cout << "Package 3 assigned to locker: " << locker3->getId() <<
endl;

    // Retrieve a package
    Package* retrievedPackage = location.retrievePackage(package1->getId());
    if (retrievedPackage) cout << "Retrieved package: " << retrievedPackage-
>getId() << endl;

    // Clean up
    delete package1;
    delete package2;
    delete package3;

    return 0;
}
```

Explanation of Code

- `Package`: Represents a package with size and ID. It provides methods to retrieve these attributes.
- `Locker`: Represents a locker that can store a package. It manages the package's storage and retrieval, and maintains its size and ID.
- `PickupLocation`: Manages multiple lockers. It assigns packages to available lockers based on size and retrieves them when needed.
- **Memory Management**: Dynamically allocated lockers are cleaned up in the destructor of `PickupLocation`.

This design provides a clear, scalable, and maintainable locker system for Amazon, with the potential for future enhancements such as handling more complex package management or notifications.