C950 Task-2 WGUPS Write-Up

John Mahon

ID 001193994

WGU Email: jmaho24@wgu.edu

6/10/2025

C950 Data Structures and Algorithms II

# Hash Table

**SCREENSHOT of ChainingHashTable in hashmap.py:**

# A screenshot of a computer AI-generated content may be incorrect.

**SCREENSHOT 2:**

A screenshot of a computer program

AI-generated content may be incorrect.

# Look-Up Functions

**Search function in hashmap.py**

A computer screen shot of a program

AI-generated content may be incorrect.

# C. Original Code

**Package\_1: Package class and constructorA screenshot of a computer

AI-generated content may be incorrect.**

**Package\_2: \_\_str\_\_() formatting logic**

**A screenshot of a computer

AI-generated content may be incorrect.**

**Package\_3: Status/address logic used in time-based simulation**

**A screenshot of a computer program

AI-generated content may be incorrect.**

**TRUCK\_1: Class definition and init method**

**A screenshot of a computer

AI-generated content may be incorrect.**

**TRUCK\_2:** **Core methods (load\_package and deliver\_package)**

**A screenshot of a computer

AI-generated content may be incorrect.**

**Routing\_1: A screenshot of a computer

AI-generated content may be incorrect.**

**Routing\_2:** **A screenshot of a computer

AI-generated content may be incorrect.**

**DISTANCE HELPER:** **A screenshot of a computer

AI-generated content may be incorrect.**

**MAIN BLOCK\_1:** **A screenshot of a computer

AI-generated content may be incorrect.**

**MAIN BLOCK\_2(truck simulation):**

**A screen shot of a computer program

AI-generated content may be incorrect.**

**MAIN BLOCK\_3 (special package handling):** **A computer screen shot of a program

AI-generated content may be incorrect.CLI LOOP\_1:** **A screenshot of a computer

AI-generated content may be incorrect.**

**CLI LOOP\_2:** **A screenshot of a computer program

AI-generated content may be incorrect.**

# C1. Identification Information

# A screenshot of a computer AI-generated content may be incorrect.

# C2. Process and Flow Comments

**Comments for package\_helper.py, my package csv loader utility:** **A screenshot of a computer

AI-generated content may be incorrect.**

**Comments for my distance\_helper.py, the distance matrix utility:** A screenshot of a computer program

AI-generated content may be incorrect.

# D. Interface

**Screenshot of the CLI interface, showing 2 queries for package 9 displaying the address change: A screenshot of a computer

AI-generated content may be incorrect.**

# D1. First Status Check

**Status of all packages at 8:35 AM:**

A screenshot of a computer

AI-generated content may be incorrect.A screenshot of a computer

AI-generated content may be incorrect.A screenshot of a computer

AI-generated content may be incorrect.A screenshot of a computer

AI-generated content may be incorrect.A screenshot of a computer

AI-generated content may be incorrect.

# D2. Second Status Check

**Status of all packages at 10:25 AM:A screenshot of a computer

AI-generated content may be incorrect.** **A screenshot of a computer

AI-generated content may be incorrect.**A screenshot of a computer

AI-generated content may be incorrect.A screenshot of a computer

AI-generated content may be incorrect.A screenshot of a computer

AI-generated content may be incorrect.

# D3. Third Status Check

**Status of all packages at 01:12 PM:** A screenshot of a computer

AI-generated content may be incorrect.A screenshot of a computer

AI-generated content may be incorrect.A screenshot of a computer

AI-generated content may be incorrect.

# A screenshot of a computer AI-generated content may be incorrect.A screenshot of a computer AI-generated content may be incorrect.

# E. Screenshot of Code Execution

**Program Execution and Final Mileage Output:**

A screenshot of a computer

AI-generated content may be incorrect.

# F1. Strengths of the Chosen Algorithm

I selected the Nearest Neighbor algorithm because it matched the way I naturally approach problem-solving, by breaking complex tasks into smaller, more manageable pieces and addressing the most immediate priority first. Nearest Neighbor operates on a similar principle, selecting the closest unvisited location at each step, which made it an intuitive and practical fit for the package delivery scenario. It is also computationally efficient, requiring far less processing power than more exhaustive approaches like Brute Force, which would have been excessive for a 40-package problem. One of the key advantages I found during testing was that Nearest Neighbor consistently produced low-mileage delivery routes with minimal complexity. While I knew it wouldn’t guarantee the absolute optimal route, its greedy nature allowed me to stay well within the 140-mile constraint, even when I manually adjusted truck assignments. Overall, it provided a reliable, fast, and flexible foundation for building out the simulation.

# F2. Verification of Algorithm

The Nearest Neighbor algorithm I implemented in my project meets all this task’s requirements. It successfully delivers all 40 packages using three trucks while keeping the total milage under 140 miles (118.8 is my best). I was able to deliver all packages within their deadlines and accounting for address changes, without any major modifications or additional layers of prioritization / weighting logic. The algorithm was tested across multiple start times and scenarios, ensuring consistent and reliable performance. All packages were delivered by 12:02 PM, which I was pleased with!

# F3. Other Possible Algorithms

Two other algorithms I could have used for this project would be Kruskal’s Algorithm and Dijkstra’s Algorithm.

# F3a. Algorithm Differences

**Dijkstra’s Algorithm:** this would be a graph-based solution that finds the most optimal path between a starting point and all other points in a weighted graph. In the context of the WGUPS delivery problem, each address could be represented as a node, with the distances between them modeled as weighted edges. Unlike the Nearest Neighbor algorithm, which selects the next closest stop one at a time using a greedy approach, Dijkstra’s calculates the shortest path from the starting location to every other node based on total distance. However, it is not designed to determine the most efficient sequence of multiple deliveries. On its own, Dijkstra’s wouldn’t solve the full route optimization problem required by WGUPS, since it focuses on point-to-point paths rather than planning an entire delivery loop. Additionally, it wouldn’t account for constraints such as multiple trucks, delayed packages, or delivery deadlines without a lot of additional code.

**Kruskal’s Algorithm:** this would also be a graph-based approach to solving the WGUPS delivery scenario. It connects all delivery points in a tree-like structure using the shortest possible total distance, without forming any loops. Unlike the Nearest Neighbor algorithm, which builds a route by visiting the next closest stop in order, Kruskal’s focuses on creating the most efficient overall network of connections. However, it builds a tree, not a route. This means it does not define the order of deliveries, and it doesn’t include a return to the hub. To use Kruskal’s for package delivery, significant extra logic would be needed to convert the spanning tree into a usable route that delivers all packages, returns to the hub, and accounts for deadlines and truck constraints.

# G. Different Approach

If I were to do C950 again, I would aim to programmatically load the trucks instead of assigning packages manually. In my current solution, I grouped packages by hand based on factors like deadlines, special notes, and context from the WGU-provided area map. While this approach worked, it required a lot of manual tweaking to stay under the mileage constraint — which, in hindsight, felt a bit like a workaround. A better solution would involve logic that automatically sorts and assigns packages based on factors such as delivery deadlines, delays, and geographic zones. This would make the solution more scalable, consistent, and better equipped to handle different or larger datasets without manual intervention.

Additionally, I would improve how the simulation handles time-based logic. Right now truck departure times are pre-set, and time-sensitive conditions (like delayed packages or address corrections) are handled through static conditional statements for specific packages. In a future version, I would implement dynamic, real-time adjustments like having the simulation recalculate a truck’s route when a package’s address changes, rather than simply planning to set up Truck 3 to wait at the hub until the address change comes through. This would make the system more responsive and better reflect real-world logistics.

# H. Verification of Data Structure

The primary data structure used to store and manage packages in my solution was a custom hash table. This structure stores all 40 package objects using their package ID as the key and supports constant-time O(1) insertion, retrieval, and lookup. It met all requirements in the scenario by efficiently updating package statuses, addresses, and delivery times as the simulation progressed. The hash table was adapted from WGU’s provided code repository, which originally demonstrated the concept using a set of movie titles. The example helped me better understand how hash tables function and how to apply them to this project.

# H1. Other Data Structures

Other data structures that could have been used for this project are Python’s native Dictionary and a Binary Search Tree.

# H1a. Data Structure Differences

**Python Dictionary:** Python’s native dict type provides a built-in hash table that functions very similarly to the self-built hash table I implemented. Instead of manually creating a hash function (e.g., bucket\_index = hash(key) % table\_size), the dict type automatically uses an optimized internal hashing mechanism (Python Software Foundation, n.d.-b). It also handles collision resolution internally, eliminating the need to implement methods for insertion or removal. While Python’s dict would likely perform faster and more efficiently, the C950 project required students to build their own data structure to better understand how hash tables work under the hood.

**Binary Search Tree:** A binary search tree (BST) is a hierarchical data structure where each node has up to two children. The left child contains a smaller value than the parent, and the right child contains a larger value. A BST can perform searching, insertion, and deletion in O(log n) time when the tree is balanced. Unlike hash tables, which use hash functions for constant-time access, BSTs rely on key comparisons to navigate the structure. While hash tables are generally faster and do not require structural maintenance, BSTs must be kept balanced to remain efficient. If left unbalanced, a BST can degrade into a structure resembling a linked list, which results in poor performance. This would be problematic in the WGUPS package delivery scenario. Although a BST could meet the requirements, it would require additional logic to maintain balance and ensure acceptable performance.

# I. Sources

Python Software Foundation. (n.d.-b). *Glossary: Dictionary.* Python 3.10 documentation. https://docs.python.org/3/glossary.html#term-dictionary

Lysecky, R., & Vahid, F. (2018, June). *C950: Data structures and algorithms II.* zyBooks. Retrieved June 5, 2025, from https://learn.zybooks.com/zybook/WGUC950AY20182019/