C950

Task 1

Western Governors University

Student ID: **#010734215**

AbdAllah M AbdAllah Khodeir

# Introduction

In the Data Structures and Algorithms II (C950) course, students are tasked with employing the Python programming language to address a package delivery-related vehicle-routing problem (VRP). The VRP is subject to various constraints, including: the maximum number of trucks that can be deployed; the maximum capacity of each truck in terms of the number of packages it can carry; time-sensitive package delivery deadlines; specific truck assignment requirements for certain packages; the need to deliver certain packages as a group; restrictions on package departure times from the hub; and the maximum total mileage accumulative for all trucks.

# A. Algorithm Identification

The code implements a HashMap data structure with a self-adjusting algorithm. The algorithm is used to dynamically adjust the size of the HashMap based on the load factor, which is the ratio of the number of items stored in the HashMap to its capacity. The algorithm is triggered when the load factor exceeds 0.7, which indicates that the HashMap is getting too full and needs to be resized to ensure efficient performance.

When the load factor exceeds 0.7, the algorithm doubles the capacity of the HashMap and reinserts all the key-value pairs from the old map into the new map. This process of resizing the HashMap is referred to as the self-adjusting algorithm, as it adjusts the size of the HashMap dynamically based on the load factor, ensuring that the HashMap remains efficient even as the number of items stored in it changes over time.  
  
The code demonstrates an implementation of an object-oriented program. The code defines two classes: HashMap and Package. The HashMap class represents a hash map data structure that stores key-value pairs. The class has several methods such as insert, lookup, remove, and resize that manipulate the hash map data. The Package class represents a package and has several attributes such as ID, address, city, state, zipcode, deadline\_time, weight, and status to store the details of the package. The Package class also has a method \_\_init\_\_ to initialize the attributes of the package when a new instance is created.

# B. Program Overview

1. Algorithm Logic using Pseudocode

The program has a logic that creates a hash table to store package names as keys and their respective values as the values. It uses a self-adjusting hash table to ensure efficient space utilization and reduce collisions. The algorithm logic can be described as follows in pseudocode:

class HashMap:

1. Initialize the map data structure with a given capacity (default = 20)

2. Set the size of the hash map to 0

3. Initialize an empty list for each bucket

Method insert(key, item):

1. Calculate the bucket number using the hash of the key

2. Get the list of key-value pairs in the bucket

3. Search for an existing key

a. If the key already exists, update the value

b. If the key does not exist, add a new key-value pair

4. Increase the size of the hash map by 1

5. Check if the load factor exceeds 0.7, if yes, resize the hash map

Method lookup(key):

1. Calculate the bucket number using the hash of the key

2. Get the list of key-value pairs in the bucket

3. Search for the key in the bucket

a. If the key is found, return the value

b. If the key is not found, raise an error

Method remove(key):

1. Calculate the bucket number using the hash of the key

2. Get the list of key-value pairs in the bucket

3. Search for the key in the bucket

a. If the key is found, delete the key-value pair

b. If the key is not found, raise an error

Method resize():

1. Store the old map data

2. Double the capacity of the hash map

3. Reset the size of the hash map to 0

4. Initialize a new empty map

5. Reinsert all the key-value pairs into the new map

2. Programming Environment

The application was created in Python 3.9.7 in PyCharm IDE on Mac M1 Chip.

3. Space-Time Complexity

Each major segment of the program has a space-time complexity of O(n), where n is the number of packages. The entire program also has a space-time complexity of O(n). The self-adjusting hash table ensures efficient space utilization, which reduces the risk of collisions and improves the overall time complexity of the program.

4. Scalability and Adaptability

The solution is capable of scaling and adapting to a growing number of packages. The self-adjusting hash table ensures efficient space utilization, which reduces the risk of collisions and improves the overall time complexity of the program. This enables the solution to adapt to a growing number of packages without sacrificing performance.

5. Efficiency and Maintainability

The software is efficient and easy to maintain due to its use of a self-adjusting hash table. The hash table provides efficient space utilization, which reduces the risk of collisions and improves the overall time complexity of the program. Additionally, the use of a hash table makes the software easy to maintain, as the logic for inserting and updating values in the table is straightforward.

6. Strengths and Weaknesses of Self-Adjusting Data Structures

The self-adjusting data structure used in the program, such as the hash table, has several strengths. The use of a hash table provides efficient space utilization, which reduces the risk of collisions and improves the overall time complexity of the program. Additionally, the hash table is easy to maintain, as the logic for inserting and updating values in the table is straightforward. However, the hash table also has some weaknesses. One of the weaknesses is that the hash table is not suitable for large data sets, as the risk of collisions increases with the size of the data set. Additionally, the hash table may require more computational resources than other data structures, such as arrays or linked lists.

# D. Explain how your data structure accounts for the relationship between the data points you are storing

The code defines a hash table data structure (class "HashMap") that can be used to store the package data in the algorithm. The hash table uses a hash function to map each key (in this case, the package ID) to a specific index in an underlying array that represents the buckets of the hash table. Each bucket is implemented as a list of key-value pairs, where each key-value pair consists of a key (the package ID) and its associated value (the package data). The hash table accounts for the relationship between the data points (the packages) by using the package ID as the key for each key-value pair in the hash table. This allows for efficient lookups, insertions, and deletions of packages based on their ID.

# G. Time Checks

1. Provide screenshots to show the status of *all* packages at a time between 8:35 a.m. and 9:25 a.m.  
     
   A computer screen capture

   Description automatically generated with medium confidence

Graphical user interface, text

Description automatically generated

Text

Description automatically generated

Graphical user interface, text

Description automatically generated

Graphical user interface, text, application

Description automatically generatedGraphical user interface, text

Description automatically generatedText

Description automatically generated

1. Provide screenshots to show the status of all packages at a time between 9:35 a.m. and 10:25 a.m.

# A computer screen capture Description automatically generated with medium confidence

Graphical user interface, text

Description automatically generated

A screenshot of a computer

Description automatically generated with medium confidence

Graphical user interface, text

Description automatically generated

Graphical user interface, text

Description automatically generated

A computer screen capture

Description automatically generated with medium confidence

Graphical user interface, text

Description automatically generated

Graphical user interface, text

Description automatically generated

1. Provide screenshots to show the status of all packages at a time between 12:03 p.m. and 1:12 p.m.

# A computer screen capture Description automatically generated with medium confidence

Graphical user interface, text

Description automatically generated

Text

Description automatically generated

Graphical user interface, text

Description automatically generated

Graphical user interface, text

Description automatically generated

Graphical user interface, text

Description automatically generated

Graphical user interface, text

Description automatically generated

Graphical user interface, text

Description automatically generated

# H. Provide a screenshot or screenshots showing successful completion of the code, free from runtime errors or warnings, that includes the total mileage traveled by *all* trucks.

Text

Description automatically generated

# I. Justify the core Algorithm

1. The algorithm used in the solution is a hash map implementation using chaining to resolve hash collisions. It has the following strengths:

* Efficient look up time: The average time complexity for lookup operation is O(1) as it calculates the bucket number using the hash of the key and then searches the bucket list to find the value associated with the key.
* Dynamic resizing: The hash map dynamically resizes itself when the load factor exceeds 0.7 to ensure that the performance remains optimal as the number of elements in the map grows.

2. Verification of Requirements:

The algorithm meets all requirements in the scenario as it provides a data structure to store the key-value pairs, it allows to store and retrieve the information of a package based on its ID, and it implements resizing to ensure the efficiency remains optimal even with many packages.

3. Other Named Algorithms:

Two other algorithms that would meet the requirements in the scenario are Trie and Binary Search Tree.

a. Comparison with Other Algorithms:

Trie: It is a tree-based data structure and is best suited for searching and retrieving strings. The Trie data structure would require additional effort to store key-value pairs in a form that it can be easily searched and retrieved.

Binary Search Tree: It is a tree-based data structure that provides efficient search and retrieval operations. However, unlike hash map, it does not provide an O(1) average time complexity for search operations as the height of the tree can grow as the number of elements increase.

# I. Different Approach

If I do this project again, I will try using Dijkstra's algorithm. Dijkstra's algorithm could be used to determine the minimum distance between the hub and each recipient address. This would involve representing the recipients' addresses as nodes in the graph and the distances between them as edges. The algorithm could then be applied to find the shortest path from the hub to each recipient, considering the package delivery deadline times. By doing so, the most efficient delivery route can be determined, ensuring that all packages are delivered on time while minimizing the total distance traveled.

# K. Data Structure Justification

1. The code implements a hash map data structure to store and retrieve the packages to be delivered.

* The time needed to complete the look-up function is affected by the number of packages to be delivered because it relies on the hash of the key to locate the correct bucket where the key-value pair is stored. The time complexity of hash map look-up operation is O(1) on average, which means it is constant, and does not depend on the number of packages. However, in the worst-case scenario, when there are collisions and multiple key-value pairs are stored in the same bucket, the time complexity can be O(n), where n is the number of key-value pairs in the bucket.
* The data structure space usage is not affected by changes in the number of packages to be delivered, as the hash map dynamically resizes itself when the load factor exceeds 0.7. The size of the hash map is proportional to the capacity, which is doubled when the load factor exceeds 0.7. The space complexity of hash map is O(n), where n is the number of key-value pairs stored in the map.
* Changes to the number of trucks or the number of cities would not affect the look-up time or the space usage of the hash map data structure, as the hash map only stores the key-value pairs of the packages to be delivered.

2. Two other data structures that could meet the same requirements in the scenario are:

* A binary search tree - a binary search tree is a data structure that stores elements in a hierarchical structure where elements to the left are smaller and elements to the right are larger. The time complexity of binary search tree look-up operation is O(log n), where n is the number of elements stored in the tree. The space complexity of binary search tree is O(n), where n is the number of elements stored in the tree. The binary search tree is different from the hash map because it relies on the values of the elements to locate the correct node, whereas the hash map uses the hash of the key to locate the correct bucket.
* An AVL tree - an AVL tree is a self-balancing binary search tree where the height of the left and right subtree of every node differs by at most 1. The time complexity of AVL tree look-up operation is O(log n), where n is the number of elements stored in the tree. The space complexity of AVL tree is O(n), where n is the number of elements stored in the tree. The AVL tree is different from the hash map because it relies on the values of the elements to locate the correct node, whereas the hash map uses the hash of the key to locate the correct bucket. The AVL tree is also different from the binary search tree because it balances itself to maintain a height difference of at most 1 between the left and right subtree of every node, whereas the binary search tree does not.