COMP1002 Final Assignment Optimizing Urban Parcel Logistics Using Data Structures and Algorithms

Student Name: Shayan Ali
Student ID: 22715218

1) Overview

The report outlines how I built the delivery system from CityDrop Logistics using 4 main modules. Each module uses different data structures and algorithms to handle a specific part of the delivery process. The system includes:

1. Graph-Based Route Planning

This module finds the best path between each hub through the city

2. Hash-Based Customer Lookup

This module gives quick access to all customer records using their customer

3. Heap-Based Parcel Scheduling

This module decides which parcel should be delivered first by calculating the Priority using the estimated time from Module 1 and the Priority Level from Module 2

4. Sorting Delivery Records

This module sorts the delivery records at the end of the day using Merge and Quick Sort

All the modules are connected with one central class, **LogisticsHandler.java,** which is a sub-menu for all these classes, and that is connected to one Main class, **CityDropLogistics.java**, which is the starting point of the program. The system is designed to be modular, so every module can be tested and improved separately.

Throughout the project, I have focused more on clean design, making sure the code is readable and ensuring that everything works perfectly. The final program is a read world delivery logistics manager

2) Data Structures Used

Graph – Used in Route Planning

Using an adjacency list, I stored the city map. Each node in the graph represents a hub (A,B, C, D,E, F, G, H), and each edge represents a road segment with a weight, which is the travel time. The reason for using an adjacency list is that it's more space-efficient, which is better for a delivery network. The time complexity for this module is O(n + m), where n is the number of vertices and m is the number of edges, which is compared to O(n^2) for matrices. Adjacent list is also easier to work with dynamically, and it is faster to iterate.

The module consists of a sub-menu:

```
========== Graph-Based Route Planning Optimization ==========
1. Insert Hub
2. Insert road segment
3. Insert Hubs and Road Segment Automatically
4. Display Adjacency List
5. Run Breath-First Search
6. Run Depth-First Search
7. Find Shortest Paths between Hubs
0. Return to Main Menu
```

- Insert nodes and Edges automatically

```
Enter choice: 3
Graph loaded with 8 nodes and 12 edges.
```

- Display the Adjacency List

```
--- Adjacency List of Hubs ---
A: B (4 mins) C (8 mins)
B: A (4 mins) C (2 mins) D (5 mins)
C: A (8 mins) B (2 mins) E (10 mins) H (9 mins)
D: B (5 mins) E (3 mins) F (7 mins)
E: C (10 mins) D (3 mins) F (1 mins) G (6 mins)
F: D (7 mins) E (1 mins) H (4 mins)
G: E (6 mins) H (2 mins)
H: F (4 mins) G (2 mins) C (9 mins)
```

For this module, 3 algorithms are applied:

Breadth-First Search (BFS)

BFS was used to find all hubs that were reachable from the source, and how far they were based on the number of hops they were. Based on the reason that BFS explores all the vertices at the current depth level before moving any deeper, it uses a queue to keep track.

```
--- Breath-First Search ---
Delivery traversal from hub: A B C D E H F G

Reachable delivery zones:
Started at hub A
Reached hub B in 1 hop(s)
Reached hub C in 1 hop(s)
Reached hub D in 2 hop(s)
Reached hub E in 2 hop(s)
Reached hub F in 3 hop(s)
Reached hub G in 3 hop(s)
Reached hub H in 2 hop(s)
```

Depth-First Search (DFS)

DFS is used to detect the cycles (loops) in the network. This helps identify any insufficient paths. DFS works by using a stack to explore all the vertices as deeply as possible. If it ever has to backtrack and visits the same node again, that's how it detects a cycle. Basically, if you go from one node and end up back at that same node through a different path, it means there's a loop in the graph.

```
--- Depth-First Search ---
Delivery traversal from hub: A B C E D F H G

Cycle detected! This may cause delivery delays.
```

Dijkstra's Algorithm (Shortest Path)

To calculate the shortest time from one hub to another, I used Dijkstra's algorithm, this algorithm finds the shortest path in a weighted graph. It works by starting at a source hub, for example, a Warehouse, then repeatedly picks the next unvisited hub with the smallest travel time so far and then updates its neighbours if a faster route is found(GeeksforGeeks, 2023). The original algorithm uses heaps, but I have used arrays, as my implementation loops through all unvisited hubs to find the one with the lowest time.

```
--- Shortest Paths ---
A -> A: 0 mins
A -> B: 4 mins
A -> C: 6 mins
A -> D: 9 mins
A -> E: 12 mins
A -> F: 13 mins
A -> G: 17 mins
A -> H: 15 mins
```

```java
// go through all vertices
for (int count = 0; count < numVertices - 1; count++) {
    int u = findMinDistance(dist, visited);     // Find unvisited node with the smallest distance
    if (u == -1) {
        break; // No reachable unvisited nodes are left
    }

    visited[u] = true;                          // Mark this node as visited

    DSAListNode curr = adjList[u].getHead();
    while (curr != null) {
        DSAEdge edge = (DSAEdge) curr.getValue();
        int v = getIndex(edge.getDestination());

        if (!visited[v] && dist[u] != Integer.MAX_VALUE &&
            dist[u] + edge.getTravelTime() < dist[v]) {
            dist[v] = dist[u] + edge.getTravelTime();  // Update the distance if a better path is found
        }

        curr = curr.getNext();
    }
}
```

Hash Table – Used for Customer Lookup

I used a Hash Table with linear probing to store and access customers' records using their customer ID. The record consists of CustomerID, Name, Address, PriorityLevel and DeliveryStatus. To access any record in the data, CustomerID acts as a key, and the key is used to search and delete records. The reason for using Linear Probing was that it was easy to implement using an array-based hash table, and it avoids using a custom linked list, unlike chaining. My implementation of linear probing handles collisions by storing each record in an array and using state flags to mark indexes as unused (0), used (1) and deleted (-1). When a collision occurs, the program starts probing from the original hashed index and moves forward one step at a time, wrapping around if needed, until it finds an unused slot. Occupied or deleted slots are skipped during the probe.

```java
// find method for linear probing
private int find(String inKey) {
    int hashIdx = hash(inKey); // Get the starting index using the hash function
    int origIdx = hashIdx; // Remember from where we started the orginal index
    boolean giveUp = false; // to see when we haev searched everything

    while (!giveUp) {
        DSAHashEntry entry = hashArray[hashIdx];

        // checks if the slot is empty or null then stops searching as key isnt in the table
        if (entry == null || entry.getState() == 0) {
            giveUp = true;

        // checks if the slot is is used or the key matches if it does returns the index
        } else if (entry.getState() == 1 && entry.getKey().equals(inKey)) {
            return hashIdx;

        // else will move to the next index
        } else {
            hashIdx = (hashIdx + 1) % hashArray.length;
            if (hashIdx == origIdx) {
                giveUp = true;
            }
        }
    }
    return -1;
}
```

The module consists of a sub-menu:

```
========== Hash-Based Customer Lookup ==========
1. Insert New Customer (Manual)
2. Insert Customers from File
3. Search Customer by ID
4. Delete Customer by ID
5. Display Hash Table
0. Return to Main Menu
Enter choice:
```

- Insertion is manual and also through file (eg.. customer_records.csv) also shows collision when inserting file

```
Enter choice: 2
Collision at index 0, probing next...
Collision at index 1, probing next...
Collision at index 2, probing next...
Collision at index 3, probing next...
Collision at index 4, probing next...
Collision at index 5, probing next...
Collision at index 6, probing next...
Collision at index 7, probing next...
Collision at index 8, probing next...
Collision at index 9, probing next...
Collision at index 31, probing next...
Collision at index 32, probing next...
Collision at index 33, probing next...
Collision at index 34, probing next...
Collision at index 35, probing next...
Collision at index 36, probing next...
Collision at index 37, probing next...
Collision at index 38, probing next...
Collision at index 39, probing next...
Collision at index 40, probing next...
Collision at index 9, probing next...
Collision at index 10, probing next...
Collision at index 10, probing next...
Collision at index 11, probing next...
Collision at index 11, probing next...
Collision at index 12, probing next...
Collision at index 12, probing next...
Collision at index 13, probing next...
Collision at index 13, probing next...
Collision at index 14, probing next...
Collision at index 14, probing next...
Collision at index 15, probing next...
Collision at index 15, probing next...
Collision at index 16, probing next...
Collision at index 16, probing next...
Collision at index 17, probing next...
Collision at index 17, probing next...
Collision at index 18, probing next...
Collision at index 18, probing next...
Collision at index 19, probing next...
Collision at index 40, probing next...
Collision at index 41, probing next...
Successfully loaded 50 customers from CSV.
```

- Searching for the customer through the customer ID

```
Enter Customer ID to search: C36
Found: Customer ID: C36
  Name: Shayan Ali
  Address: 9154 Curtin Waters
  Priority: 3
  Status: Delivered
```

- Deletion through Customer ID

```
Enter Customer ID to delete: C36
Customer deleted.
```

❖ Display Hash Table

```
Enter choice: 5
[0] Customer ID: C09
  Name: Frank Wright
  Address: 789 Pine Road
  Priority: 1
  Status: In Transit
[1] Customer ID: C20
  Name: Jack White
  Address: 680 Juniper St
  Priority: 1
  Status: Delivered
[2] Customer ID: C21
  Name: Mona Lewis
  Address: 246 Palm Drive
  Priority: 2
  Status: In Transit
[3] Customer ID: C22
  Name: Ray Moore
  Address: 357 Willow Ave
  Priority: 1
  Status: Delayed
[4] Customer ID: C23
  Name: Jack White
  Address: 791 Poplar Blvd
  Priority: 1
  Status: Pending
[5] Customer ID: C24
  Name: Olivia Davis
  Address: 234 Birch Blvd
  Priority: 4
  Status: Delayed
[6] Customer ID: C25
  Name: David Miller
  Address: 246 Palm Drive
  Priority: 3
  Status: In Transit
[7] Customer ID: C26
  Name: Bob Lee
  Address: 234 Birch Blvd
  Priority: 1
  Status: Delivered
[8] Customer ID: C27
  Name: Jack White
  Address: 913 Magnolia Ct
  Priority: 5
  Status: Delivered
[9] Customer ID: C28
  Name: Nate Scott
  Address: 357 Willow Ave
  Priority: 5
  Status: In Transit
[10] Customer ID: C29
  Name: Frank Wright
```

Heap – Used for Parcel Scheduling

I used Max Heap to prioritize parcel delivery based on 2 factors:

❖ The customer's Priority Level 'P' (from the hash table)

❖ The Estimated Time 'T' (from the graph's shortest path) T

Both factors were used in a computed formula to calculate the Priority = (6 – P) / 1000 / T. The formula gives each parcel a Priority, then the max heap ensures that the highest-priority parcel is always at the top and delivered first. The reason for using a max heap is that it allows you to insert elements and always get the highest priority one in the best time. T

```java
// Adds a new delivery request to the heap using priority formula
public void addDelivery(String customerID, int customerPriority, int deliveryTime) {
    // Calculate priority: (6 - P) + 1000 / T
    double priorityscore = (6 - customerPriority) + (1000.0 / deliveryTime);

    // Build a string to represent the delivery info usign value
    String value = "Customer " + customerID + " | P=" + customerPriority + ", T=" + deliveryTime;

    // Print the calculation for priority
    System.out.println(x:"--------------------------------------------------");
    System.out.println("Adding to heap: " + value);
    System.out.println("Priority = (6 - " + customerPriority + ") + 1000 / " + deliveryTime +" = " + priorityscore);

    // Insert into heap using existing method of add
    add(priorityscore, value);
}
```

The module consists of a sub-menu:

```
========== Heap Based Parcel Scheduling ==========
1. Schedule all deliveries into heap
2. View current heap state
3. Dispatch next highest-priority parcel
4. View heap state after dispatch
0. Return to Main Menu
Enter choice:
```

- Schedule all deliveries into the heap

```
Scheduling deliveries into heap...
No path to A for C09
Scheduled: C21 -> B, Priority Level: 2, Time: 4 mins, Priority: (6 - 2) + (1000 / 4) = 254.0
Scheduled: C25 -> C, Priority Level: 3, Time: 6 mins, Priority: (6 - 3) + (1000 / 6) = 169.66666666666666
Scheduled: C28 -> D, Priority Level: 5, Time: 9 mins, Priority: (6 - 5) + (1000 / 9) = 112.11111111111111
Scheduled: C41 -> E, Priority Level: 5, Time: 12 mins, Priority: (6 - 5) + (1000 / 12) = 84.33333333333333
Scheduled: C42 -> F, Priority Level: 1, Time: 13 mins, Priority: (6 - 1) + (1000 / 13) = 81.92307692307692
Scheduled: C12 -> G, Priority Level: 3, Time: 17 mins, Priority: (6 - 3) + (1000 / 17) = 61.8235294117647
Scheduled: C15 -> H, Priority Level: 4, Time: 15 mins, Priority: (6 - 4) + (1000 / 15) = 68.66666666666667
No path to A for C16
Scheduled: C18 -> B, Priority Level: 3, Time: 4 mins, Priority: (6 - 3) + (1000 / 4) = 253.0
Scheduled: C37 -> C, Priority Level: 3, Time: 6 mins, Priority: (6 - 3) + (1000 / 6) = 169.66666666666666
Scheduled: C02 -> D, Priority Level: 1, Time: 9 mins, Priority: (6 - 1) + (1000 / 9) = 116.11111111111111
Scheduled: C06 -> E, Priority Level: 1, Time: 12 mins, Priority: (6 - 1) + (1000 / 12) = 88.33333333333333
Scheduling complete.
```

- View the heap

```
---- Current Heap State ----
[0] Priority: 254.0 -> C21 (Priority: 254.0)
[1] Priority: 253.0 -> C18 (Priority: 253.0)
[2] Priority: 112.11111111111111 -> C28 (Priority: 112.11111111111111)
[3] Priority: 169.66666666666666 -> C25 (Priority: 169.66666666666666)
[4] Priority: 116.11111111111111 -> C02 (Priority: 116.11111111111111)
[5] Priority: 61.8235294117647 -> C12 (Priority: 61.8235294117647)
[6] Priority: 68.66666666666667 -> C15 (Priority: 68.66666666666667)
[7] Priority: 84.33333333333333 -> C41 (Priority: 84.33333333333333)
[8] Priority: 169.66666666666666 -> C37 (Priority: 169.66666666666666)
[9] Priority: 81.92307692307692 -> C42 (Priority: 81.92307692307692)
[10] Priority: 88.33333333333333 -> C06 (Priority: 88.33333333333333)
----------------------------
```

❖ Dispatch the highest Priority parcel

```
Dispatching highest-priority parcel
Dispatched: C21 (Priority: 254.0)
```

❖ Heap after the parcel dispatched

```
Current Heap State (after dispatch):
---- Current Heap State ----
[0] Priority: 253.0 -> C18 (Priority: 253.0)
[1] Priority: 169.66666666666666 -> C25 (Priority: 169.66666666666666)
[2] Priority: 112.11111111111111 -> C28 (Priority: 112.11111111111111)
[3] Priority: 169.66666666666666 -> C37 (Priority: 169.66666666666666)
[4] Priority: 116.11111111111111 -> C02 (Priority: 116.11111111111111)
[5] Priority: 61.8235294117647 -> C12 (Priority: 61.8235294117647)
[6] Priority: 68.66666666666667 -> C15 (Priority: 68.66666666666667)
[7] Priority: 84.33333333333333 -> C41 (Priority: 84.33333333333333)
[8] Priority: 88.33333333333333 -> C06 (Priority: 88.33333333333333)
[9] Priority: 81.92307692307692 -> C42 (Priority: 81.92307692307692)
----------------------------
```

To appoint each customer with a hub from the RouteGraph module, I used a simple method where it first checks if the delivery status is **"In Transit"**. Then, using a formula that divides the customer's index by the number of hubs, each customer is randomly assigned a hub. This assigned hub is then passed into the RouteGraph module to get the shortest time, which is later used in the heap to calculate the delivery priority.

```java
DSAHashEntry[] entries = hashTable.getAllEntries(); //get all the entries from the hash table


for (int i = 0; i < entries.length; i++) {
    DSAHashEntry entry = entries[i];

    if (entry != null && entry.getState() == 1) { //checks if the slot is used
        CustomerRec customer = entry.getValue(); //gets the customer record

        if (customer.getDeliveryStatus().equals(anObject:"In Transit")) { //checks to see if the customer records DeliveryStatus = "In Transit"
            String custId = customer.getCustomerID(); //gets CustomerID
            int priorityLevel = customer.getPriorityLevel(); //Gets priority level

            //randomly assigns a hub to each customer using index
            String assignedHub = hubs[customerIndex % hubs.length];
            customerIndex++;

            // gets the estimated time from the routeGraph module
            int estimatedTime = graph.getShortestTime(assignedHub);
            if (estimatedTime == Integer.MAX_VALUE) { //if no valid path for the hub it will skip
                System.out.println("No path to " + assignedHub + " for " + custId);
            } else {
                //Calculates the priority
                double priority = (6 - priorityLevel) + (1000.0 / estimatedTime);

                // Create a string that holds customer ID and other info for storage
                String entryData = custId + " (Priority: " + priority + ")";
                heap.add(priority, entryData);// adds the priority and entry data into the heap

                System.out.println("Scheduled: " + custId + " -> " + assignedHub + ", Priority Level: " + priorityLevel + ", Time: " + estimatedTime + " mins" + ", Priority: (6 - " + priorityLevel + ") + (1000
            }
        }
    }
}
System.out.println(x:"Scheduling complete.");
break;
```

```java
// Gets the shortest time of each hub and then gives it to heap (estimated time)
public int getShortestTime(String hub) {
    int index = getIndex(hub);

    // Skip if hub not found, dist array not ready, or distance is 0 (A → A case)
    if (index == -1 || dist == null || dist[index] == 0) {
        return Integer.MAX_VALUE;
    }

    return dist[index];
}
```

Sorting Delivery Records – Merge Sort & Quick Sort

I used Merge and Quick Sort to sort the delivery records based on the estimated time. Both algorithms are efficient and commonly used for sorting large datasets and were part of the lecture material. Both merge and quick sort divide the data set and sort it. By implementing both, I can compare their performances with different inputs. The sorting is done by the **DeliverySorter.java** module, which consists of merge and quick sort, and then is implemented into **CityDropLogistics.java**. The data input is taken through the delivery_records.csv file with different inputs of 100, 500 and 1000 and are sorted into sorted_merge.csv and sorted_quick.csv files.

The module consists of a sub-menu:

```
do {
    System.out.println(x:"========== Sorting Delivery Records ==========");
    System.out.println(x:"1. Run Merge Sort for 100 inputs");
    System.out.println(x:"2. Run Merge Sort for 500 inputs");
    System.out.println(x:"3. Run Merge Sort for 1000 inputs");
    System.out.println(x:"4. Run Quick Sort for 100 inputs");
    System.out.println(x:"5. Run Quick Sort for 500 inputs");
    System.out.println(x:"6. Run Quick Sort for 1000 inputs");
    System.out.println(x:"0. Return to Main Menu");
    System.out.print(s:"Enter choice: ");
    choice = sc.nextInt();
```

- Merge Sort for 100 inputs

```
Enter choice: 1
Sorted output written to sorted_100_merge.csv
```

```
sorted_100_merge.csv > data
CustomerID,,CustomerName,EstimatedTime
C171,Queenie,15
C103,Zack,17
C104,Mona,17
C142,Yara,17
C179,Xander,17
C128,Alice,18
C195,Ray,18
C144,Leo,19
C133,Yara,20
C150,Paul,23
C139,Alice,24
C113,Nate,25
C174,Henry,26
C118,Victor,29
C178,Karen,29
C106,Ray,30
C101,Tom,31
C132,Frank,34
C151,Olivia,35
C175,David,38
C185,Wendy,38
C149,Nate,42
C172,Xander,42
C116,Ray,43
C135,Olivia,43
C181,Jack,43
C190,Paul,43
C164,Karen,44
C180,Xander,44
C193,Uma,49
C145,Nate,50
C197,Henry,50
C166,Ray,51
C186,Queenie,52
C102,Karen,55
C115,Ray,57
C121,Alice,57
C192,Queenie,57
C105,Olivia,58
C167,Tom,58
C170,Charlie,58
C157,Frank,63
C143,Paul,64
C158,Karen,64
C122,Jack,66
C127,Alice,66
C154,Xander,67
C177,Charlie,67
C119,Mona,68
C163,Frank,69
C124,Karen,72
```

- Merge Sort for 500 inputs

```
Sorted output written to sorted_500_merge.csv
```

sorted_500_merge.csv > data

```
CustomerID,,CustomerName,EstimatedTime
C171,Queenie,15
C304,Wendy,15
C390,Yara,15
C205,Grace,16
C236,Ray,16
C305,Charlie,16
C379,Jack,16
C388,Ella,16
C475,Bob,16
C500,Jack,16
C509,Isla,16
C103,Zack,17
C104,Mona,17
C142,Yara,17
C179,Xander,17
C246,Isla,17
C327,Tom,17
C128,Alice,18
C195,Ray,18
C282,Zack,18
C314,David,18
C412,Alice,18
C490,Jack,18
C560,Paul,18
C144,Leo,19
C269,Sara,19
C369,Uma,19
C418,Ella,19
C456,Victor,19
C599,Leo,19
C133,Yara,20
C355,Bob,20
C364,Tom,20
C386,Uma,20
C502,Alice,20
C548,Charlie,20
C596,Olivia,21
C226,Frank,22
C248,Ray,22
C277,Frank,22
C333,Alice,22
C340,Jack,22
C351,Nate,22
C526,Wendy,22
C529,Ella,22
C600,Henry,22
C150,Paul,23
C524,Grace,23
```

- Merge Sort for 1000 inputs

```
Enter choice: 3
Sorted output written to sorted_1000_merge.csv
```

sorted_1000_merge.csv > data

```
CustomerID,,CustomerName,EstimatedTime
C171,Queenie,15
C304,Wendy,15
C390,Yara,15
C747,Leo,15
C777,Tom,15
C998,Ray,15
C205,Grace,16
C236,Ray,16
C305,Charlie,16
C379,Jack,16
C388,Ella,16
C475,Bob,16
C500,Jack,16
C509,Isla,16
C645,Olivia,16
C697,Tom,16
C709,Bob,16
C746,Olivia,16
C751,Ray,16
C864,Olivia,16
C937,Zack,16
C966,Isla,16
C987,Ella,16
C1096,David,16
C103,Zack,17
C104,Mona,17
C142,Yara,17
C179,Xander,17
C246,Isla,17
C327,Tom,17
C671,Jack,17
C736,Zack,17
C983,Alice,17
C1008,Charlie,17
C1018,Bob,17
C1089,Leo,17
C128,Alice,18
C195,Ray,18
C282,Zack,18
C314,David,18
C412,Alice,18
C490,Jack,18
```

- Quick Sort for 100 inputs

```
Sorted output written to sorted_100_quick.csv
```

```
sorted_100_quick.csv > data
CustomerID,,CustomerName,EstimatedTime
C171,Queenie,15
C104,Mona,17
C142,Yara,17
C103,Zack,17
C179,Xander,17
C128,Alice,18
C195,Ray,18
C144,Leo,19
C133,Yara,20
C150,Paul,23
C139,Alice,24
C113,Nate,25
C174,Henry,26
C178,Karen,29
C118,Victor,29
C106,Ray,30
C101,Tom,31
C132,Frank,34
C151,Olivia,35
C175,David,38
C185,Wendy,38
C172,Xander,42
C149,Nate,42
C116,Ray,43
C181,Jack,43
C135,Olivia,43
C190,Paul,43
C180,Xander,44
C164,Karen,44
C193,Uma,49
C145,Nate,50
C197,Henry,50
C166,Ray,51
C186,Queenie,52
C102,Karen,55
C121,Alice,57
C115,Ray,57
C192,Queenie,57
C105,Olivia,58
```

❖ Quick Sort for 500 inputs

```
Enter choice: 5
Sorted output written to sorted_500_quick.csv
```

sorted_500_quick.csv > data

```
CustomerID,,CustomerName,EstimatedTime
C171,Queenie,15
C304,Wendy,15
C390,Yara,15
C205,Grace,16
C305,Charlie,16
C379,Jack,16
C388,Ella,16
C236,Ray,16
C475,Bob,16
C500,Jack,16
C509,Isla,16
C142,Yara,17
C103,Zack,17
C327,Tom,17
C179,Xander,17
C104,Mona,17
C246,Isla,17
C314,David,18
C412,Alice,18
C128,Alice,18
C490,Jack,18
C195,Ray,18
C282,Zack,18
C560,Paul,18
C144,Leo,19
C369,Uma,19
C269,Sara,19
C418,Ella,19
C456,Victor,19
C599,Leo,19
C355,Bob,20
C364,Tom,20
C502,Alice,20
C133,Yara,20
C548,Charlie,20
C386,Uma,20
```

❖ Quick Sort for 1000 inputs

```
Enter choice: 6
Sorted output written to sorted_1000_quick.csv
```

sorted_1000_quick.csv > data

```
CustomerID,,CustomerName,EstimatedTime
C171,Queenie,15
C304,Wendy,15
C390,Yara,15
C747,Leo,15
C777,Tom,15
C998,Ray,15
C388,Ella,16
C236,Ray,16
C475,Bob,16
C500,Jack,16
C509,Isla,16
C645,Olivia,16
C697,Tom,16
C709,Bob,16
C746,Olivia,16
C205,Grace,16
C751,Ray,16
C305,Charlie,16
C864,Olivia,16
C937,Zack,16
C966,Isla,16
C987,Ella,16
C379,Jack,16
C1096,David,16
C179,Xander,17
C104,Mona,17
C246,Isla,17
C671,Jack,17
C736,Zack,17
C142,Yara,17
C327,Tom,17
C983,Alice,17
C1008,Charlie,17
```

Sorting Performance Tester

This module tests each of the 3 data sets, 100, 500 and 1000 and compares their execution time to show which of the 2 sorts is more efficient on which type of dat. To get the execution time have used **System.nanoTime();** before and after sorting the data. The module asks for the file name for example deliver_records_100_random and will give the execution time for both quick and merge sort

```java
public static void testDeliveryRecords(){
    Scanner sc = new Scanner(System.in);

    System.out.println("===== Testing Menu =====");

    System.out.print("Enter full file name (e.g., delivery_records.csv): ");
    String fileName = sc.next();

    // Load data separately for each sort
    DeliveryRecord[] recordsMerge = DeliveryRecordFileIO.loadFromCSV(fileName);
    DeliveryRecord[] recordsQuick = DeliveryRecordFileIO.loadFromCSV(fileName);

    // validation to check if the file is written right or not
    if (recordsMerge == null || recordsQuick == null) {
        System.out.println("Error loading file. Please check the file name.");
        return;
    }

    // Merge Sort
    long startMerge = System.nanoTime();
    DeliverySorter.mergeSort(recordsMerge);
    long endMerge = System.nanoTime();
    double mergeTime = (endMerge - startMerge) / 1_000_000.0;

    // Quick Sort
    long startQuick = System.nanoTime();
    DeliverySorter.quickSort(recordsQuick);
    long endQuick = System.nanoTime();
    double quickTime = (endQuick - startQuick) / 1_000_000.0;


    // Output both times
    System.out.printf("Merge Sort on %s took %.2f ms\n", fileName, mergeTime);
    System.out.printf("Quick Sort on %s took %.2f ms\n", fileName, quickTime);


}
```

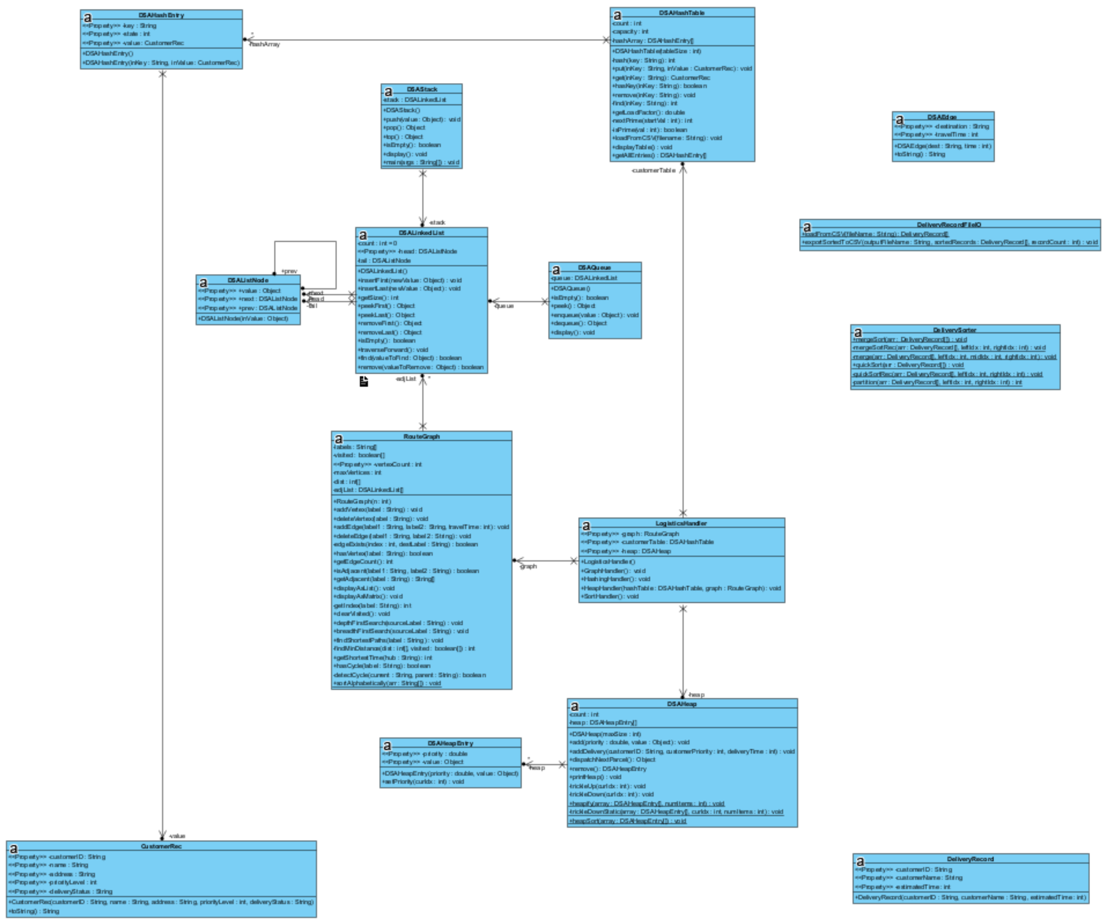
Execution Times:

DataSets	Merge Sort	Quick Sort	Best algorithm
delivery_records_100_random.csv	0.29	0.01	Quick Sort

delivery_records_500_random.csv	0.07	0.05	Quick Sort
delivery_records_1000_random.csv	0.68	0.09	Quick Sort
delivery_records_100_nearly.csv	0.02	0.03	Merge Sort
delivery_records_500_nearly.csv	0.12	1.02	Merge Sort
delivery_records_1000_nearly.csv	0.10	0.75	Merge Sort
delivery_records_100_reversed.csv	0.02	0.02	Equal
delivery_records_500_reversed.csv	0.05	0.07	Merge Sort
delivery_records_1000_reversed.csv	0.06	0.24	Merge Sort

Looking at the time execution results, Quick Sort was faster on random datasets, especially the smaller ones. But when it came to nearly sorted and reversed data, Merge Sort performed much better. This lines up with what we learned in the lectures — Quick Sort, even though it's fast, isn't reliable when the data is already sorted. Merge Sort, on the other hand, was more consistent overall, no matter which dataset was given.

Module Integration



Algorithm Complexity

Module 1: Graph-Based Route Planning

Operation	Time Complexity Best/Worst/Average
Add Vertex	O(1)
Add Edge	O(1)
BFS / DFS	O(n + m)

Dijkstra Shortest Path	$O(n^2)$

Module 2: Hash-Based Customer Lookup

Operation	Best Case	Average Case	Worst Case
Insert	O(1)	O(1)	O(n)
Search	O(1)	O(1)	O(n)
Delete	O(1)	O(1)	O(n)

Module 3: Heap-Based Parcel Scheduling

Operation	Best Case	Average Case	Worst Case
Add	O(1)	O(log n)	O(log n)
Remove	O(log n)	O(log n)	O(log n)
Heapify	O(n)	O(n)	O(n)

Module 4: Sorting Delivery Records

Merge Sort:

Case	Time Complexity
Best	O(n log n)
Average	O(n log n)
Worst	O(n log n)

Quick Sort:

Case	Time Complexity
Best	O(n log n)
Average	O(n log n)
Worst	$O(n^2)$

Reflection

This assignment helped me understand how different data structures and algorithms can work together in one system. Since I had already worked with these modules before, I thought it would be easier to implement them — but combining them in a way where they depended on each other's data was more challenging than expected. Once I brought everything together through a central menu, it made me think more seriously about modular design.

I separated each module — Graph, Heap, Hashing, and Sorting — inside LogisticsHandler, and then connected it to the main menu in CityDropLogistics.java. I also created a separate class, DeliveryRecordFileIO, to handle file input/output for both Module 2 and Module 4. I think this made the design more modular and organized.

The heap module was the most interesting challenge for me. Assigning each customer to a hub and calculating the correct priority was tricky, but once it was working, it integrated really well with the hash table for priority levels.

If I had more time, I would improve the input handling — for example, in the hashing module, the file names are hardcoded. I would have preferred to allow the user to input the file name, making it more flexible and scalable for different customer records.

Reference

GeeksforGeeks. (2023, May 22). *Dijkstra's shortest path algorithm*. https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/