Complex Data Structure

Case: Parcel Service
Student Names: Bako Asaad (510944) & Emin Fikret (511275)
Group: 4

Quartile One: First attempt
Date: 11/06-2022

Content

1. Linear Data Structures, Linear & Binary Search.

We have started our work on this project by first creating entities for the packages ("src/main/java/entities/DeliveryPackage.java") and clients ("src/main/java/entities/Client.java"), and a DataReader class ("src/main/java/data/DataReader.java"). The purpose of this class is to read and sort by id the information from the Package.csv and Client.csv into ArrayLists.

Afterward, we created a DataTransformer class ("src/main/java/data/DataTransformer.java") which helps us by transforming the array lists into different data structures, e.g PriorityQueue (line 57-72), HashMap (12-43), Array (46-56) or Stack.

Moreover, we created a package called search ("src/main/java/search") which contains three classes, LinearSearch and BinarySearch class. We have chosen to search clients and packages by id.

```
public class BinarySearch {

    /** Binary search for package id in the ArrayList package data. **/
    FikretEmin
    public void binarySearchInPackagesForPackageId(int searchedPackageId, ArrayList<DeliveryPackage> receivedDeliveryPackages) {...}

    /** Binary search for client id in the ArrayList package data. **/
    FikretEmin
    public void binarySearchForClientId(int clientId, ArrayList<Client> clientArrayList) {...}
```

```
public class LinearSearch {

    /**
     * Linear search for package id in the ArrayList package data.
     **/
    FikretEmin
    public void linearSearchInPackagesForPackageId(int searchedPackageId, ArrayList<DeliveryPackage> receivedDeliveryPackages) {...}

    /** Linear search for package id in the Array package data. **/
    FikretEmin
    public void linearSearchInPackagesForPackageId(int searchedPackageId, DeliveryPackage[] receivedDeliveryPackages) {...}

    /** Linear search for package id in the PriorityQueue package data. **/
    FikretEmin
    public void linearSearchInPackagesForPackageId(int searchedPackageId, PriorityQueue<DeliveryPackage> receivedDeliveryPackages) {...}

    /** Linear search for client id in the ArrayList package data. **/
    FikretEmin
    public Client linearSearchForClientId(int clientId,ArrayList<Client> clients) {...}
}
```

	Linear search	Binary search
Description	Linear search is a way to find an element by traversing a list until the element is found.	The binary search looks for the middle element of the list compares it to the searched element and selects a side of the list to process again. It does that until the element is found.
Big O	The time complexity of the linear search in the worst-case scenario is O(n), n being the size of the list. The best case is O(1), 1 being the first element in the list.	The time complexity in the worst-case scenario for the binary search is O(logn). (CITE). The best case is O(1), 1 being the middle element.

<table>
<tr><td>List size</td><td>Suitable for small data sets.</td><td>Useful for large data sets.</td></tr>
<tr><td>Precondition</td><td>The linear search works for the sorted and unsorted list, unlike the binary search.</td><td>The precondition for using a binary search is having a sorted list.</td></tr>
<tr><td>Testing</td><td colspan="2">We used both search methods to find the first, middle and last element in a sorted package list. (If you would like to test the linear and binary search, you could do that at our ParcelServiceTest class (“src/main/java/testing/ParcelServiceTests.java”).

We have generated 10 000 delivery packages using the PackageGenerator class (“src/main/java/data/PackageGenerator.java).

Our results are the following:

Positions:
First element: 9254
Middle element: 37078
Last element: 64413

Test for first position:
The searched value (9254) has been found at position 1
Finding a delivery package with LINEAR SEARCH takes 6997000 nanos.

The searched value (9254) has been found at position 1
Finding a delivery package with BINARY SEARCH takes 999700 nanos.

Test for middle position:
The searched value (37078) has been found at position 5000
Finding a delivery package with LINEAR SEARCH takes 3998100 nanos.

The searched value (37078) has been found at position 5000
Finding a delivery package with BINARY SEARCH takes 0 nanos.

Test for last position:
The searched value (64413) has been found at position 10000
Finding a delivery package with LINEAR SEARCH takes 4997900 nanos.

The searched value (64413) has been found at position 10000
Finding a delivery package with BINARY SEARCH takes 1001900 nanos.

We expected the speed of the binary search for the middle and last positions to exceed the search speed of the linear method. What we did not expect were the results of the first position.</td></tr>
</table>

2. Sorting & Searching in Linear Data Structures.

In order to tackle this task, we created a “sort” package (“src/main/java/sort”) containing couple of sorting methods and a “sort checker”. We have implemented BubbleSort, InsertionSort, MergeSort, SelectionSort and a SortChecker in the “sort” package.

	InsertionSort	SelectionSort	MergeSort
Description	Insertion sort works like splitting a deck of cards, sorted and unsorted. By moving an element from the unsorted deck to the correct position in the sorted deck, we eventually sort or deck of cards.	This sorting algorithm searcher for the smaller elements and puts it in the right position. Repeats that until the list is sorted.	MergeSort is a recursive method which halves the array until there is one element left. After that sort and combines to spitted array until the initial list is full its size.
Big O	This sorting algorithm has $0(n^2)$ time complexity because it loops the list two types in order to check the elements.	The best, average and worst-case complexity is $O(n^2)$. It does not matter if the list is sorted or not, the method will loop it two times, meaning n * n.	The best, average and worst-case complexity is $O(n \log n)$. It is the fastest sorting algorithm we have studied next to Quicksort and the one we have used in this project. The reason the time complexity is n log n is because the list recursively halves itself.

Testing	We have tested the speed of each algorithm in our ParcelServiceTest class in order to implement and see the concepts we have learned in action.
	
	`There are 10000 waiting to be sorted.`
	
	`Insertion sort (SORTED) 155 ms.`
	`Selection sort (SORTED) 285 ms.`
	`Merge sort (SORTED) 10 ms.`
	
	The insertion and selection sorting methods have $O(n^2)$ or so-called quadratic time complexity. As we learned in Parallel Computing, this means that the runtime is proportional to the square root of the size. If the input is X times as big the runtime will		

	be X^2 as long. In the case below, the runtime will be around 4 times bigger than the runtime displayed in our initial test. There are 20000 waiting to be sorted. Insertion sort (SORTED) 928 ms. Selection sort (SORTED) 1410 ms. Merge sort (SORTED) 21 ms.
Implementation	In our project we have used MergeSort as it is one of fastest sorting algorithm we have studied. We did not incorporate the MergeSort class in our application, we used it for testing purposes, instead we implemented the Comparable interface to our entities, overwrote the “toCompare” method and started using the Collections class to sort and shuffle our lists. We have used the sorting algorithm in the DataReader class to ensure that we work with already sorted set of clients and delivery packages.

As mentioned earlier, we created a DataTransromer class (“src/main/java/data/DataTransformer.java “) which is used to transform our initial array lists to different data structures, one of which was the HashMap.

	HashMap
Description & Implementation	HashMap is a set of key and value pairs. The keys are unique, so no duplicates are allowed. The time complexity of receiving an element from a HashMap is O(1). This time complexity is called constant, because the runtime is the same regardless of the number of elements in the list. We have created HashMaps using the ids of our entities as keys and the entities themselves as the values. This allows us to immediately receive an object and work with it. HashMap performance compared to LinearSearch and BinarySearch:

	First element: 11248 Middle element: 66058 Last element: 120561 The searched value (120561) has been found at position 20000 Finding a delivery package with LINEAR SEARCH takes 6999000 nanos. The searched value (120561) has been found at position 20000 Finding a delivery package with BINARY SEARCH takes 1000200 nanos. The searched value (120561) has been found at position 20000 Finding a delivery package with HASHMAP takes 0 nanos.

Other other task was to set up a customer linear data structure. We set up the ClientParcelLDS (“src/main/java/data/customlineardatastructures/ClientParcelLDS.java”).

	ClientParcelLDS
Description & Implementation	We have created a HashMap-based linear data structure. Elements can be added and removed. We have also included a search for an element and a sorting method with the help of the Comparable interface. The data structure is basically a HashMap which accepts integers as keys and ArrayList as values. We have set it up in a way that it will have the function to look at the packages data, take every single client and save their packages into arrays as values in the hashmap. The best feature of this data structure is that by having the client id a person will be able to receive all packages of the client in O(1) time. Finally, we have set up a sorting method, so we can receive the clients with the most packages. 218617 has 34 218099 has 31 219472 has 33 217722 has 30 219212 has 32 220072 has 30 218367 has 32 219162 has 29 219793 has 32 220291 has 29 219252 has 31 218252 has 29

3. AVL tree

In week 3 we started by creating a package, called “avltrees” (“src/main/java/avltrees”). This package contains two AVLTrees. One for the clients and another for the package data.

<table>
<tr><th></th><th>ClientAVLtree & PackageAVLtree</th></tr>
<tr><td>Description & Implementation</td><td> An AVL tree is a balanced binary search tree. This means that every time we insert or delete an element into the tree, it will rebalance, meaning that the height of the left and right subtree of any node differs by not more than 1. We also learned that the AVL trees are used in database applications in which insertions and deletions are less but there are frequent lookups for data. That is why we decided that it will be a good idea if we balance our AVL trees by id. This will allow us to sort our data. Every time there is a deletion or insertion in the AVL tree, it will rebalance, and the information will be kept sorted. </td></tr>
<tr><td>Speed & Comparison to HashMap</td><td> We have tested our PackageAVLtree by searching for an element. We expected to see a lower speed than the HashMap because the average time complexity of the AVL Tree which is 0(log n) is slower than the constant 0(1) time complexity of the HashMap. However, their search results were quite similar.
<pre>
There are 20000000 sorted packages.

First element: 14652
Middle element: 55018733
Last element: 110038658

The searched value (55018733) has been found at position 10000000
Finding a delivery package with LINEAR SEARCH takes 52080300 nanos.

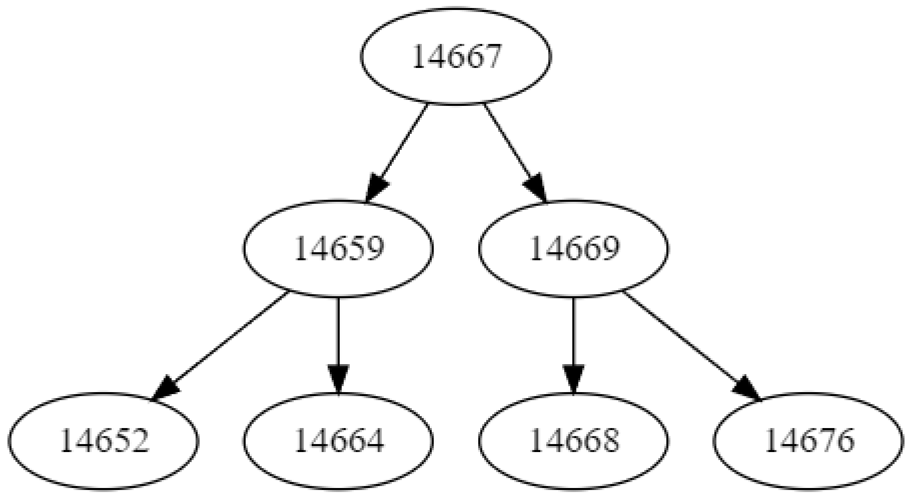
The searched value (55018733) has been found at position 10000000
Finding a delivery package with BINARY SEARCH takes 199501 nanos.

Finding a delivery package with HASHMAP takes 30000 nanos.

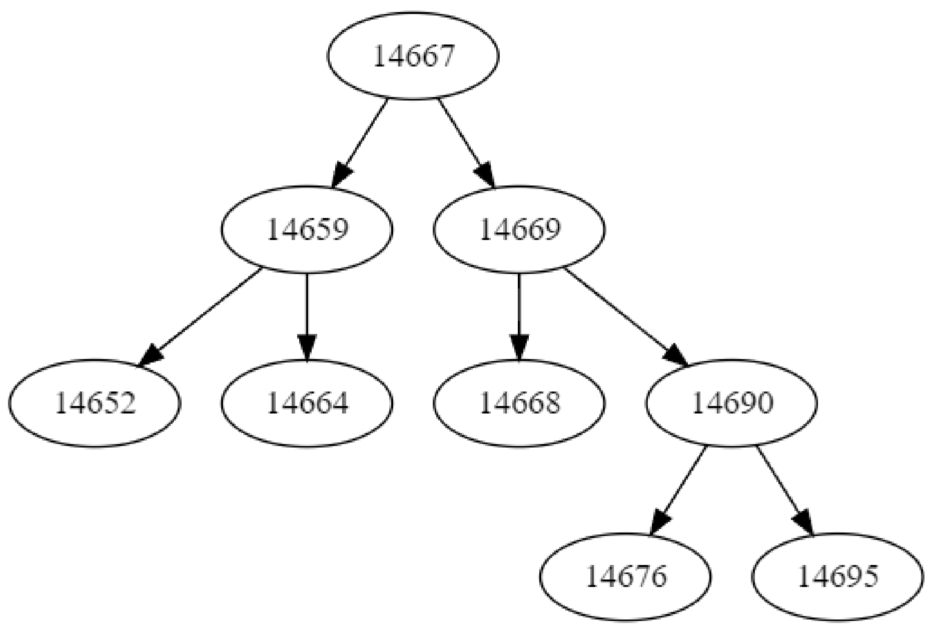
The searched value has been found.
Package Id: 55018733, length: 95, breadth: 30, height: 30, weight: 3.507, entry date: +15719-11-09, clientId: 124696.
Finding a delivery package with PackageAVLTree search takes 30400 nanos.
</pre>
Moreover, we learned that the main advantage of the hash table over AVL trees is the constant search speed. The AVL tree wins in different scenarios, for example, it is better if we are working with a large data set that requires constant sorting. In conclusion, both the AVL tree and the HashMap are useful in different situations. Moreover, we wanted to see if the AVL Trees are working correctly. In our trees we implemented “traverseInOrder”, “traversePreOrder”, “traversePostOrder” and a printing method which prints the elements in order. </td></tr>
</table>

We worked with a small amount of data for our test in order to display our results.

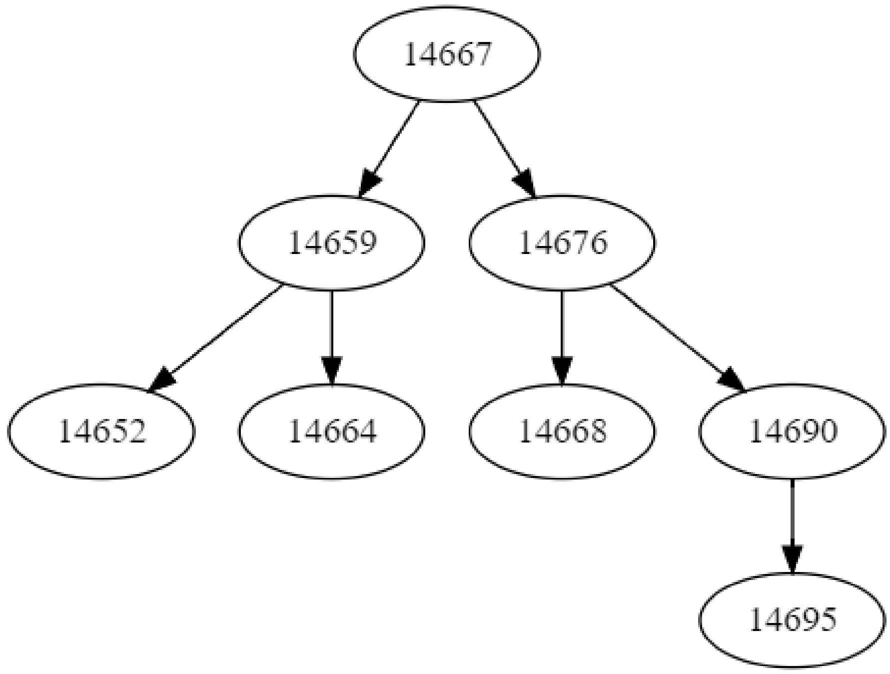
Initial tree:



After insertion of (14690, 14695):



After deletion of (14669):

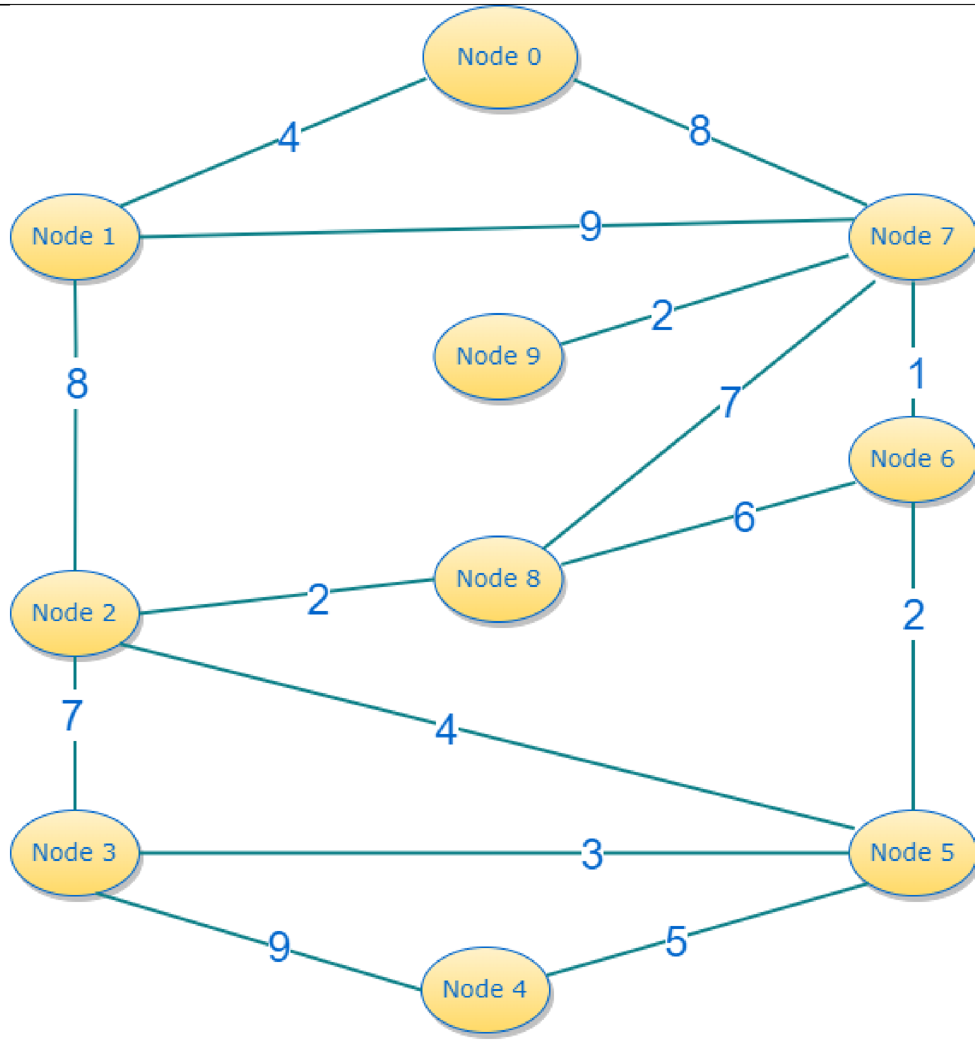
4 & 5. Graph & Graph Algorithms

Our work on the graph started by creating a package called graphs (“src/main/java/graphs”). There we created our Node and Graph class. And to show and print our results we have implemented them in the class ParcelService -> method: (findShortestPath).

<table>
<tr>
<td>Description & Implementation</td>
<td>
We have used an adjacency matrix for our graph which is a dense graph.

Adjacency Matrix Graph:
<table>
<tr><th></th><th>217684</th><th>217689</th><th>217694</th><th>217698</th><th>217702</th><th>217711</th><th>217716</th><th>217718</th><th>217719</th><th>217721</th></tr>
<tr><td>0::217684</td><td>0</td><td>4</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>8</td><td>0</td><td>0</td></tr>
<tr><td>1::217689</td><td>4</td><td>0</td><td>8</td><td>0</td><td>0</td><td>0</td><td>0</td><td>9</td><td>0</td><td>0</td></tr>
<tr><td>2::217694</td><td>0</td><td>8</td><td>0</td><td>7</td><td>0</td><td>4</td><td>0</td><td>0</td><td>2</td><td>0</td></tr>
<tr><td>3::217698</td><td>0</td><td>0</td><td>7</td><td>0</td><td>9</td><td>3</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>4::217702</td><td>0</td><td>0</td><td>0</td><td>9</td><td>0</td><td>5</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>5::217711</td><td>0</td><td>0</td><td>4</td><td>3</td><td>5</td><td>0</td><td>2</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>6::217716</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>2</td><td>0</td><td>1</td><td>6</td><td>0</td></tr>
<tr><td>7::217718</td><td>8</td><td>9</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>7</td><td>2</td></tr>
<tr><td>8::217719</td><td>0</td><td>0</td><td>2</td><td>0</td><td>0</td><td>0</td><td>6</td><td>7</td><td>0</td><td>0</td></tr>
<tr><td>9::217721</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>2</td><td>0</td><td>0</td></tr>
</table>
A dense graph is a graph in which the number of edges is close to the maximal number of edges. In other words, almost all nodes in the graph are connected.
</td>
</tr>
</table>

	Moreover, our graph is considered weighted because we add a weight to the created edges. Also, it is undirected because later on we implemented the Prim algorithm and that algorithm would work only on an undirected graph.
	
	Furthermore, we have manually added only 10 clients and edges with weight to them so we can easily test and print our results.
	
	`graph.addNode(new Node(clients.get(5).getClientId()));`
	`graph.addNode(new Node(clients.get(6).getClientId()));`
	`graph.addNode(new Node(clients.get(7).getClientId()));`
	`graph.addNode(new Node(clients.get(8).getClientId()));`
	`graph.addNode(new Node(clients.get(9).getClientId()));`
	
	`graph.addEdge(0,1,4);`
	`graph.addEdge(0,7,8);`
	`graph.addEdge(1,2,8);`
	`graph.addEdge(7,8,7);`
	`graph.addEdge(1,7,9);`
	`graph.addEdge(7,6,1);`
	Based on these manually 10 client id we added as a node and created edges between them we have created a graph:



However, one of our ideas is to add each client in the graph and possibly edge to each other client with a weight calculated by the initial client's address and all that follow. In this way, we can check which clients are closer to the initial one.

```java
for (int i = 0; i < clients.size(); i++) {
    graph.addNode(new Node(clients.get(i).getClientId()));

    for (int j = 0; j < clients.size(); j++) {
        if (j != i) {
            int x = clients.get(i).getAddressX() - clients.get(j).getAddressX();
            int y = clients.get(i).getAddressY() - clients.get(j).getAddressY();

            graph.addEdge(i, j, -(x + y));
        }
    }
}
```

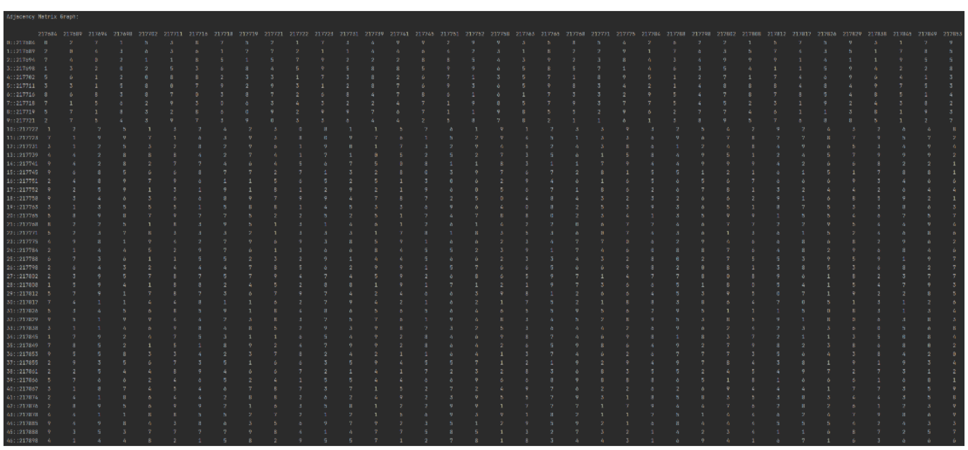
Here are the results of our graph implementation.

Moreover, we implemented the graph by using random numbers between 1 – 10 as a weight.

```java
for (int i = 0; i < clients.size(); i++) {
    graph.addNode(new Node(clients.get(i).getClientId()));

    for (int j = 0; j < clients.size(); j++) {
        if (j != i) {
            Random random = new Random();
            graph.addEdge(i, j, random.nextInt(1,10));
        }
    }
}
```

Our results using random numbers as a weight for the whole csv file.



Traversing Algorithms

As traversing algorithms, we used Depth-first search and Breadth-first search.

Breadth-first search:

BFS algorithm visits every nodes on one level before going to the next level. And for this we have used a data structure called queue.

Depth-first search:
DFS visits every nodes in a branch till it gets the final node of that line, and then moves back to up. And for this we have used a data structure called stack.

In the implementation we have printed the client id as a node.

```
Breadth first search:
217684 = visited
217694 = visited
217698 = visited
217702 = visited
217711 = visited
217716 = visited
217719 = visited
217721 = visited
217718 = visited
217689 = visited
```

```
Depth fist search:
217684 = visited
217721 = visited
217689 = visited
217719 = visited
217716 = visited
217711 = visited
217718 = visited
217702 = visited
217698 = visited
217694 = visited
```

Dijkstra & Prim Algorithm
To find the shortest path between nodes we have used two searching algorithms, Dijkstra and Prim Algorithm. The main difference between these two algorithms is that Dijkstra can be used for both directed and undirected graphs, unlike Prim which can be used only for undirected graphs, and because of the prim algorithm we made our graph in an undirected way. Another important difference is that Dijkstra provides the shortest path from one point to another, and Prim provides a sub-graph that connects all the nodes with the lowest possible sum of their edge weights.
These are our results for the 10 custom nodes. And for simplicity and to look easier on the eye, we have used small numbers between 0-9 instead of the ids of the clients.

```
Prim Algorithm
Edge  -->  Weight
0 - 0 --> 0
0 - 1 --> 4
1 - 2 --> 8
5 - 3 --> 3
5 - 4 --> 5
2 - 5 --> 4
5 - 6 --> 2
6 - 7 --> 1
2 - 8 --> 2
7 - 9 --> 2
Total cost is : 31

Dijkstra Algorithm
Node     Distance   Path
0 -> 1       4      0 1
0 -> 2       12     0 1 2
0 -> 3       14     0 7 6 5 3
0 -> 4       16     0 7 6 5 4
0 -> 5       11     0 7 6 5
0 -> 6       9      0 7 6
0 -> 7       8      0 7
0 -> 8       14     0 1 2 8
0 -> 9       10     0 7 9
```

For the Prim algorithm we have implemented two methods, in one of them we have used an integer array to store the weighted between each two nodes and another integer array to store

In the implementation of the Dijkstra graph we have implemented them in three methods, to print the shortest paths, we have stored them in an array, and to print the shortest path we used recursion, and then to print everything (Node, the weight of each path were calculated between source node to the destination node as(distance), and paths), and for the prim algorithm itself, we have used an integer array to store the shortest distances and a Boolean array so we would know when a node is visited.

```java
public void dijkstraAlgo(int[][] adjacencyMatrix, int start) {
    int[] shortestDistances = new int[matrix.length];
    boolean[] visitedNode = new boolean[matrix.length];
    for (int i = 0; i < matrix.length; i++) {
        shortestDistances[i] = Integer.MAX_VALUE;
        visitedNode[i] = false;
    }
    shortestDistances[start] = 0;
    int[] parents = new int[matrix.length];
    parents[start] = PARENT;
    for (int i = 1; i < matrix.length; i++) {
        int nearestNode = -1;
        int shortestDistance = Integer.MAX_VALUE;
        for (int j = 0; j < matrix.length; j++) {
            if (!visitedNode[j] && shortestDistances[j] < shortestDistance) {
                nearestNode = j;
                shortestDistance = shortestDistances[j];
            }
        }
        visitedNode[nearestNode] = true;
        for (int k = 0; k < matrix.length; k++) {
            int edgeDistance = adjacencyMatrix[nearestNode][k];
            if (edgeDistance > 0 && ((shortestDistance + edgeDistance) < shortestDistances[k])) {
                parents[k] = nearestNode;
                shortestDistances[k] = shortestDistance + edgeDistance;
            }
        }
    }
    printDijkstra(start, shortestDistances, parents);
}
```

```java
private void printDijkstra(int startVertex, int[] distances, int[] parents) {
    int nNodes = distances.length;
    System.out.println("Dijkstra Algorithm");
    System.out.print("Node\t Distance\tPath");
    for (int i = 0; i < nNodes; i++) {
        if (i != startVertex) {
            System.out.print("\n" + startVertex + " -> ");
            System.out.print(i + " \t\t ");
            System.out.print(distances[i] + "\t\t");
            printDijkstraPath(i, parents);
        }
    }
}
```

```java
private void printDijkstraPath(int currentVertex, int[] parents) {
    if (currentVertex == PARENT) {
        return;
    }
    printDijkstraPath(parents[currentVertex], parents);
    System.out.print(currentVertex + " ");
}
```

6. Problem Solving & Use of Regular Expressions

For an efficient service, there are several problems that need to be solved. We created a ParcelApp which contains the solution of a couple of these problems.

```java
System.out.println("-----------------------------------------------");
System.out.println("Welcome to our Parcel Service Application (PSA).");
System.out.println("-----------------------------------------------\n");

Scanner inputReader = new Scanner(System.in);

boolean keepRunning = true;
while (keepRunning) {
    System.out.println("1. Find delivery package by id");
    System.out.println("2. Top 10 recipients in a recent period");
    System.out.println("3. Efficient route");
    System.out.println("4. Calculate vans needed for day\n");
```

<table>
<tr><td>How can you request the current status of a given package as quickly as possible?</td><td>After our research and tests, we understood that the fastest way to receive this information is by using a HashMap.</td></tr>
<tr><td>Who are the top 10 recipients in a recent period (day, month, ...)?</td><td>In order to tackle this issue, we used our custom-made HashMap-based linear data structure. We basically looped the package data, saved the client id as a key and put all packages of the client into the Array List value of the corresponding key.

Not also that, but our method received a date that is checked by a regular expression.

<pre>Pattern pattern = Pattern.compile("[0-9]{5}", Pattern.UNICODE_CASE);
Matcher matcher = pattern.matcher(Integer.toString(packageId));
boolean matchFound = matcher.find();</pre></td></tr>
<tr><td>Given a road map, what is the fastest/shortest route from the current location toward a specific client's address?</td><td>We used our Graph in order to provide the fastest route for the driver.

More information could be found in the Graph chapter.</td></tr>
<tr><td>How many vans/drivers do you need per day</td><td>Firstly, we created a class Van, one of the important methods in that class related to this problem is the van area method. It multiplies the length and width of the van as we cannot stack delivery packages.</td></tr>
</table>

<table>
<tr>
<td></td>
<td>

Moreover, we created a method in the delivery package class that calculates what is the most efficient way to place a package in the van.

```java
public int deliveryPackageArea() {
    if (height >= length  && height >= breadth) {
        return length * breadth;
    } else if (length >= height && length >= breadth) {
        return height * breadth;
    } else if (breadth >= length && breadth >= height) {
        return length * height;
    }
    return breadth * length;
}
```

Our method takes a specific period and saves the packages in a linear data structure. After that it starts filling a van, if a van is ready, it will create a new one until no packages are left.

```java
if (Pattern.matches("([1-9]|[1-3][0-9])-([0-9]|[0-9][0-9])-[0-9]{4}", date)) {
    String[] startDateParts = date.split("-");
    LocalDate startDate = LocalDate.of(Integer.parseInt(startDateParts[2]), Integer.
```

Furthermore, and most importantly we used **Stack** as a linear data structure for the packages in a van. Because when the driver delivers packages, he or she will always take out the last one that was putted in the van.

</td>
</tr>
<tr>
<td>We have also implemented some methods for some simple scenarios to know what is the most suited data structure. This implementation was done in the Data Reader class.</td>
<td>

```java
public void searchTopTenRecipients(int day, int month, int year) {...}
new *
public void searchTopTenRecipients(int month, int year) {...}
new *
public void searchTopTenRecipients(int year) {...}
new *
public void topTenRecipients() {...}
new *
public void findClientById(HashMap<Integer,Client> clientHashMap,int clientId) {...}
new *
public void findPackageById(HashMap<Integer,DeliveryPackage> deliveryPackages,int packageId) {...}
new *
public void addPackagesToDeliveryCarUsingHashMap(HashMap<Integer,DeliveryPackage> deliveryPackages,int packageId) {...}
new *
public void addPackagesToDeliveryCarUsingArrayList(ArrayList<DeliveryPackage> deliveryPackages, int packageId) {...}
```

</td>
</tr>
</table>