

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.....	2
2. Sorting & Searching in Linear Data Structures.	4
3. AVL tree.....	7
4 & 5. Graph & Graph Algorithms.....	9
6. Problem Solving & Use of Regular Expressions	16

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 $O(\log n)$. (CITE). The best case is $O(1)$, 1 being the middle element.

List size	Suitable for small data sets.	Useful for large data sets.
Precondition	The linear search works for the sorted and unsorted list, unlike the binary search.	The precondition for using a binary search is having a sorted list.
Testing	<p>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").</p> <p>We have generated 10 000 delivery packages using the PackageGenerator class ("src/main/java/data/PackageGenerator.java").</p> <p>Our results are the following:</p> <p>Positions:</p> <pre>First element: 9254 Middle element: 37078 Last element: 64413</pre> <p>Test for first position:</p> <pre>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.</pre> <p>Test for middle position:</p> <pre>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.</pre> <p>Test for last position:</p> <pre>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.</pre> <p>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.</p>	

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 $O(n^2)$ 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	<p>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.</p> <pre> There are 10000 waiting to be sorted. Insertion sort (SORTED) 155 ms. Selection sort (SORTED) 285 ms. Merge sort (SORTED) 10 ms. </pre> <p>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</p>		

	<p>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.</p>
Implementation	<p>In our project we have used MergeSort as it is one of fastest sorting algorithm we have studied.</p> <p>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.</p> <p>We have used the sorting algorithm in the DataReader class to ensure that we work with already sorted set of clients and delivery packages.</p>

As mentioned earlier, we created a DataTransomer 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	<p>HashMap is a set of key and value pairs. The keys are unique, so no duplicates are allowed.</p> <p>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.</p> <p>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.</p> <p>HashMap performance compared to LinearSearch and BinarySearch:</p>

	<pre> 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. </pre>
--	---

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	<p>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.</p> <p>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.</p> <p>Finally, we have set up a sorting method, so we can receive the clients with the most packages.</p> <pre> 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 </pre>

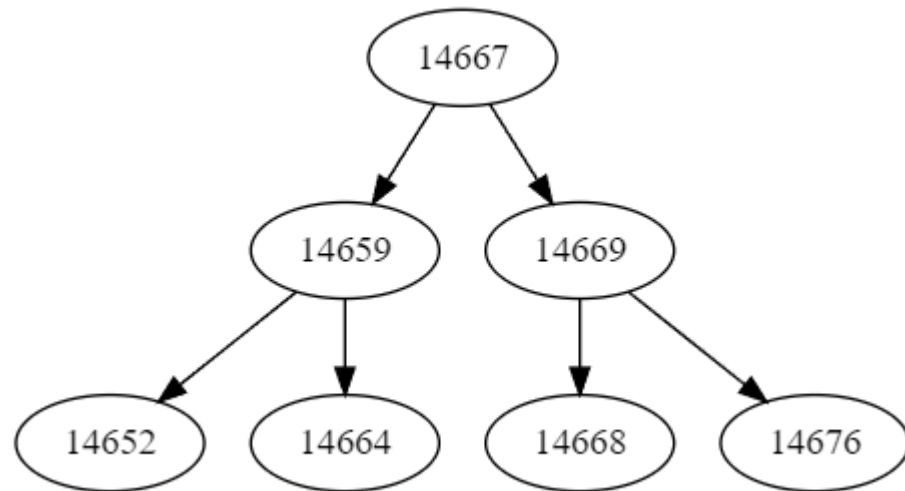
3. AVL tree

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.

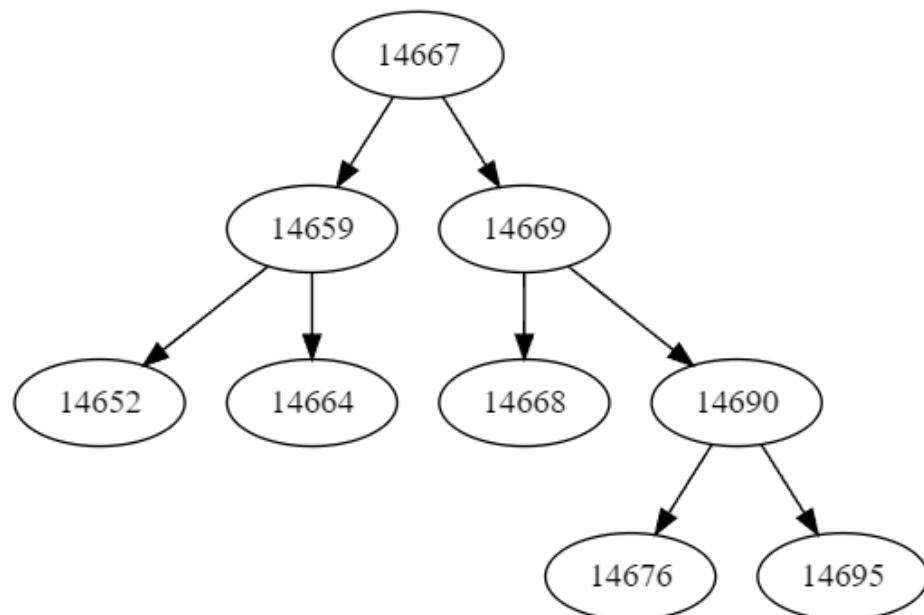
	ClientAVLtree & PackageAVLtree
Description & Implementation	<p>An AVL tree is a balanced binary search tree. This means that every time we insert or delete an element into the tree, it will rebalance, meaning that the height of the left and right subtree of any node differs by not more than 1.</p> <p>We 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.</p>
Speed & Comparison to HashMap	<p>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 $O(\log n)$ is slower than the constant $O(1)$ time complexity of the HashMap. However, their search results were quite similar.</p> <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> <p>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.</p> <p>In conclusion, both the AVL tree and the HashMap are useful in different situations.</p> <p>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.</p>

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

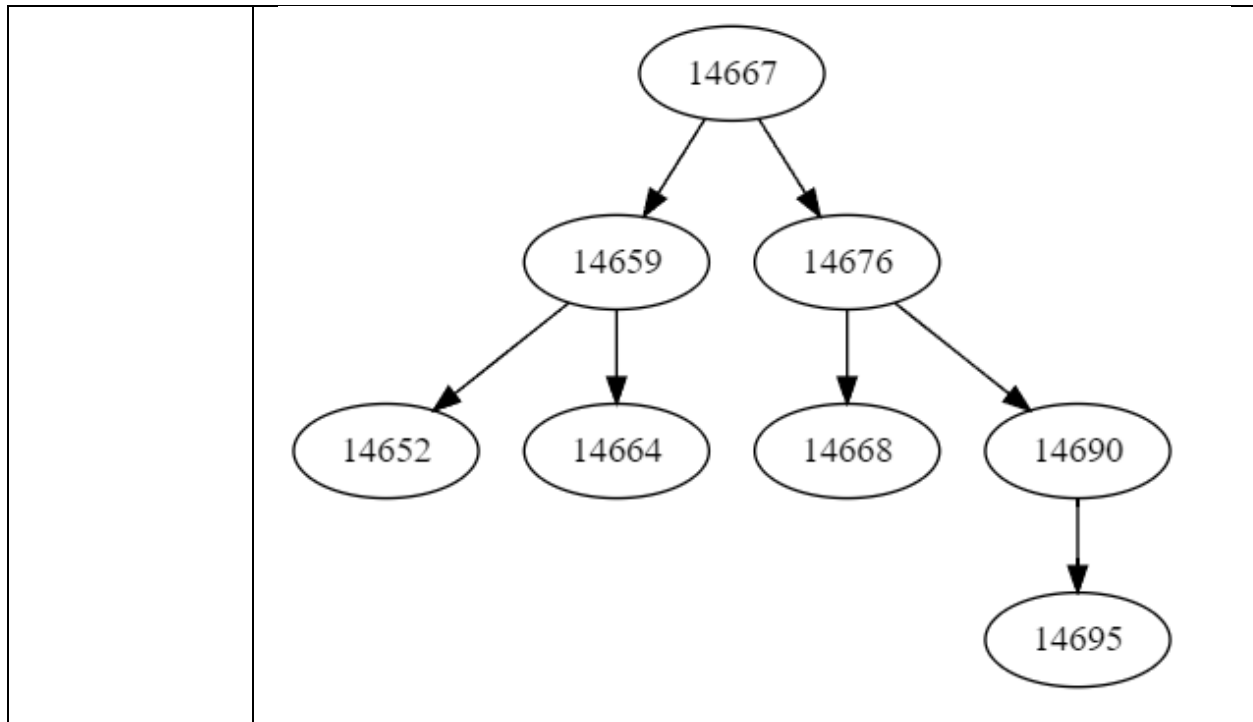
Initial tree:



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).

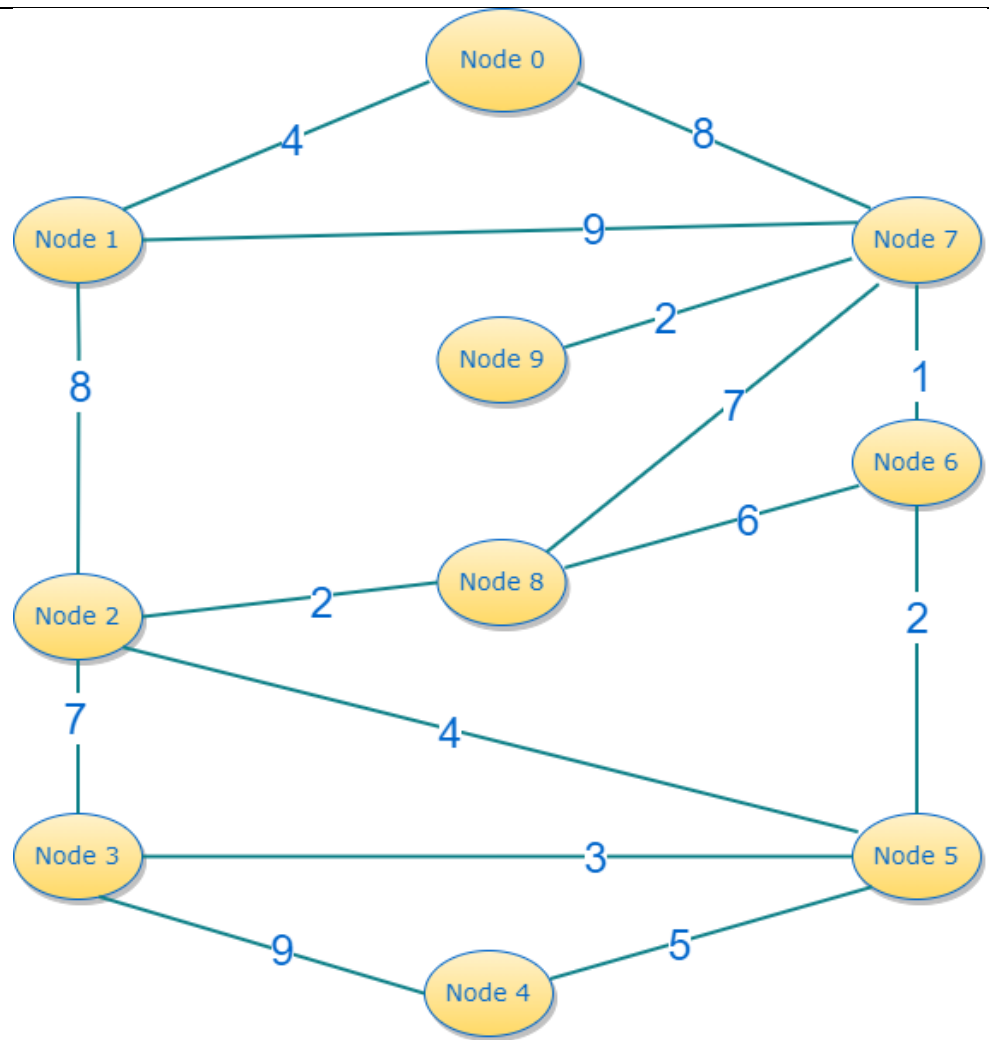
Description & Implementation	<p>We have used an adjacency matrix for our graph which is a dense graph.</p> <pre> Adjacency Matrix Graph: 217684 217689 217694 217698 217702 217711 217716 217718 217719 217721 0::217684 0 4 0 0 0 0 0 8 0 0 1::217689 4 0 8 0 0 0 0 9 0 0 2::217694 0 8 0 7 0 4 0 0 2 0 3::217698 0 0 7 0 9 3 0 0 0 0 4::217702 0 0 0 9 0 5 0 0 0 0 5::217711 0 0 4 3 5 0 2 0 0 0 6::217716 0 0 0 0 0 2 0 1 6 0 7::217718 8 9 0 0 0 0 1 0 7 2 8::217719 0 0 2 0 0 0 6 7 0 0 9::217721 0 0 0 0 0 0 0 2 0 0 </pre> <p>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.</p>
------------------------------	---

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.

```
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.

Appendix: Political Index																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																							
Year	1970	1971	1972	1973	1974	1975	1976	1977	1978	1979	1980	1981	1982	1983	1984	1985	1986	1987	1988	1989	1990	1991	1992	1993	1994	1995	1996	1997	1998	1999	2000	2001	2002	2003	2004	2005	2006	2007	2008	2009	2010	2011	2012	2013	2014	2015	2016	2017	2018	2019	2020	2021	2022	2023	2024	2025	2026	2027	2028	2029	2030	2031	2032	2033	2034	2035	2036	2037	2038	2039	2040	2041	2042	2043	2044	2045	2046	2047	2048	2049	2050	2051	2052	2053	2054	2055	2056	2057	2058	2059	2060	2061	2062	2063	2064	2065	2066	2067	2068	2069	2070	2071	2072	2073	2074	2075	2076	2077	2078	2079	2080	2081	2082	2083	2084	2085	2086	2087	2088	2089	2090	2091	2092	2093	2094	2095	2096	2097	2098	2099	2100	2101	2102	2103	2104	2105	2106	2107	2108	2109	2110	2111	2112	2113	2114	2115	2116	2117	2118	2119	2120	2121	2122	2123	2124	2125	2126	2127	2128	2129	2130	2131	2132	2133	2134	2135	2136	2137	2138	2139	2140	2141	2142	2143	2144	2145	2146	2147	2148	2149	2150	2151	2152	2153	2154	2155	2156	2157	2158	2159	2160	2161	2162	2163	2164	2165	2166	2167	2168	2169	2170	2171	2172	2173	2174	2175	2176	2177	2178	2179	2180	2181	2182	2183	2184	2185	2186	2187	2188	2189	2190	2191	2192	2193	2194	2195	2196	2197	2198	2199	2200	2201	2202	2203	2204	2205	2206	2207	2208	2209	2210	2211	2212	2213	2214	2215	2216	2217	2218	2219	2220	2221	2222	2223	2224	2225	2226	2227	2228	2229	2230	2231	2232	2233	2234	2235	2236	2237	2238	2239	2240	2241	2242	2243	2244	2245	2246	2247	2248	2249	2250	2251	2252	2253	2254	2255	2256	2257	2258	2259	2260	2261	2262	2263	2264	2265	2266	2267	2268	2269	2270	2271	2272	2273	2274	2275	2276	2277	2278	2279	2280	2281	2282	2283	2284	2285	2286	2287	2288	2289	2290	2291	2292	2293	2294	2295	2296	2297	2298	2299	2300	2301	2302	2303	2304	2305	2306	2307	2308	2309	2310	2311	2312	2313	2314	2315	2316	2317	2318	2319	2320	2321	2322	2323	2324	2325	2326	2327	2328	2329	2330	2331	2332	2333	2334	2335	2336	2337	2338	2339	2340	2341	2342	2343	2344	2345	2346	2347	2348	2349	2350	2351	2352	2353	2354	2355	2356	2357	2358	2359	2360	2361	2362	2363	2364	2365	2366	2367	2368	2369	2370	2371	2372	2373	2374	2375	2376	2377	2378	2379	2380	2381	2382	2383	2384	2385	2386	2387	2388	2389	2390	2391	2392	2393	2394	2395	2396	2397	2398	2399	2400	2401	2402	2403	2404	2405	2406	2407	2408	2409	2410	2411	2412	2413	2414	2415	2416	2417	2418	2419	2420	2421	2422	2423	2424	2425	2426	2427	2428	2429	2430	2431	2432	2433	2434	2435	2436	2437	2438	2439	2440	2441	2442	2443	2444	2445	2446	2447	2448	2449	2450	2451	2452	2453	2454	2455	2456	2457	2458	2459	2460	2461	2462	2463	2464	2465	2466	2467	2468	2469	2470	2471	2472	2473	2474	2475	2476	2477	2478	2479	2480	2481	2482	2483	2484	2485	2486	2487	2488	2489	2490	2491	2492	2493	2494	2495	2496	2497	2498	2499	2500	2501	2502	2503	2504	2505	2506	2507	2508	2509	2510	2511	2512	2513	2514	2515	2516	2517	2518	2519	2520	2521	2522	2523	2524	2525	2526	2527	2528	2529	2530	2531	2532	2533	2534	2535	2536	2537	2538	2539	2540	2541	2542	2543	2544	2545	2546	2547	2548	2549	2550	2551	2552	2553	2554	2555	2556	2557	2558	2559	2560	2561	2562	2563	2564	2565	2566	2567	2568	2569	2570	2571	2572	2573	2574	2575	2576	2577	2578	2579	2580	2581	2582	2583	2584	2585	2586	2587	2588	2589	2590	2591	2592	2593	2594	2595	2596	2597	2598	2599	2600	2601	2602	2603	2604	2605	2606	2607	2608	2609	2610	2611	2612	2613	2614	2615	2616	2617	2618	2619	2620	2621	2622	2623	2624	2625	2626	2627	2628	2629	2630	2631	2632	2633	2634	2635	2636	2637	2638	2639	2640	2641	2642	2643	2644	2645	2646	2647	2648	2649	2650	2651	2652	2653	2654	2655	2656	2657	2658	2659	2660	2661	2662	2663	2664	2665	2666	2667	2668	2669	2670	2671	2672	2673	2674	2675	2676	2677	2678	2679	2680	2681	2682	2683	2684	2685	2686	2687	2688	2689	2690	2691	2692	2693	2694	2695	2696	2697	2698	2699	2700	2701	2702	2703	2704	2705	2706	2707	2708	2709	2710	2711	2712	2713	2714	2715	2716	2717	2718	2719	2720	2721	2722	2723	2724	2725	2726	2727	2728	2729	2730	2731	2732	2733	2734	2735	2736	2737	2738	2739	2740	2741	2742	2743	2744	2745	2746	2747	2748	2749	2750	2751	2752	2753	2754	2755	2756	2757	2758	2759	2760	2761	2762	2763	2764	2765	2766	2767	2768	2769	2770	2771	2772	2773	2774	2775	2776	2777	2778	2779	2780	2781	2782	2783	2784	2785	2786	2787	2788	2789	2790	2791	2792	2793	2794	2795	2796	2797	2798	2799	2800	2801	2802	2803	2804	2805	2806	2807	2808	2809	2810	2811	2812	2813	2814	2815	2816	2817	2818	2819	2820	2821	2822	2823	2824	2825	2826	2827	2828	2829	2830	2831	2832	2833	2834	2835	2836	2837	2838	2839	2840	2841	2842	2843	2844	2845	2846	2847	2848	2849	2850	2851	2852	2853	2854	2855	2856	2857	2858	2859	2860	2861	2862	2863	2864	2865	2866	2867	2868	2869	2870	2871	2872	2873	2874	2875	2876	2877	2878	2879	2880	2881	2882	2883	2884	2885	2886	2887	2888	2889	2890	2891	2892	2893	2894	2895	2896	2897	2898	2899	2900	2901	2902	2903	2904	2905	2906	2907	2908	2909	2910	2911	2912	2913	2914	2915	2916	2917	2918	2919	2920	2921	2922	2923	2924	2925	2926	2927	2928	2929	2930	2931	2932	2933	2934	2935	2936	2937	2938	2939	2940	2941	2942	2943	2944	2945	2946	2947	2948	2949	2950	2951	2952	2953	2954	2955	2956	2957	2958	2959	2960	2961	2962	2963	2964	2965	2966	2967	2968	2969	2970	2971	2972	2973	2974	2975	2976	2977	2978	2979	2980	2981	2982	2983	2984	2985	2986	2987	2988	2989	2990	2991	2992	2993	2994	2995	2996	2997	2998	2999	3000

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

```
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.

[illegible]

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:	Depth first search:
217684 = visited	217684 = visited
217694 = visited	217721 = visited
217698 = visited	217689 = visited
217702 = visited	217719 = visited
217711 = visited	217716 = visited
217716 = visited	217711 = visited
217719 = visited	217718 = visited
217721 = visited	217702 = visited
217718 = visited	217698 = visited
217689 = 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.

```

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);
}

```

```

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);
        }
    }
}

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.

```
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");
}
```

How can you request the current status of a given package as quickly as possible?	After our research and tests, we understood that the fastest way to receive this information is by using a HashMap.
Who are the top 10 recipients in a recent period (day, month, ...)?	<p>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.</p> <p>Not also that, but our method received a date that is checked by a regular expression.</p> <pre>Pattern pattern = Pattern.compile("[0-9]{5}", Pattern.UNICODE_CASE); Matcher matcher = pattern.matcher(Integer.toString(packageId)); boolean matchFound = matcher.find();</pre>
Given a road map, what is the fastest/shortest route from the current location toward a specific client's address?	<p>We used our Graph in order to provide the fastest route for the driver.</p> <p>More information could be found in the Graph chapter.</p>
How many vans/drivers do you need per day	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.

	<p>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.</p> <pre> 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; } </pre> <p>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.</p> <pre> 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. </pre> <p>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.</p>
<p>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.</p>	<pre> public void searchTopTenRecipients(int day, int month, int year) {...} //... public void searchTopTenRecipients(int month, int year) {...} //... public void searchTopTenRecipients(int year) {...} //... public void topTenRecipients() {...} //... public void findClientById(HashMap<Integer, Client> clientHashMap, int clientId) {...} //... public void findPackageById(HashMap<Integer, DeliveryPackage> deliveryPackages, int packageId) {...} //... public void addPackagesToDeliveryCarUsingHashMap(HashMap<Integer, DeliveryPackage> deliveryPackages, int packageId) {...} //... public void addPackagesToDeliveryCarUsingArrayList(ArrayList<DeliveryPackage> deliveryPackages, int packageId) {...} </pre>