Algorithms and Data Structures 2 Final Project
Design Document

For the implementation of the 1st function, Dijkstra.java, DirectedGraph.java, and Stop.java are mainly used and they are called from Main.java. A Bus stop map is regarded as a directed graph since there are directed ways between bus stops, and therefore to find the shortest path from one to another, I made a directed graph from stop_times.txt and transfers.txt. I assume that users input bus stop IDs instead of bus stop names because those names could be very long, and it makes more sense for them to type in a maximum of 5-digit numbers.

In DirectedGraph.java, one HashMap is used for storing information of each ID and it has an object called Edge that stores 2 bus stop IDs and a weight between them where one is adjacent to another. Each line of stop_times.txt has only one bus stop ID and it needs to be connected to another one in the next line where they should be on the same trip, and I justified it by comparing their trip IDs, although each line in transfers.txt has two IDs and weight in the same line and there is no need to compare their trip IDs.

To find the shortest path between two bus stops, the Dijkstra algorithm is used because a source would be single, there is no negative weight (as it is regarded as time and would not be negative), there could be cycled paths, and it is less expensive than the Floyd-Warshall algorithm. In the Dijkstra algorithm in my program, a priority queue is used for getting a minimum weight from the costTo[vertex], and it gives me the shortest paths which have minimum costs from a single source of bus stops to all others. It enables users to get the shortest route when they input two bus stop IDs and, in this respect, the algorithm is the best to implement the first function. Each bus stop ID is connected to all the information such as stop_code and stop_name from stops.txt by making a HashMap and setting a stop ID as a key to get that information in Stop.java.

I made another HashMap for printing out a route between bus stops and this is updated inside of the Dijkstra algorithm function called dijkstraAlgo() in Dijkstra.java when the costTo[] is updated on the way of finding the shortest path. In this HashMap after the program, for example, (0 -> 1), (1 -> 3), (2 -> 4), and (3 -> 2) are stored where the starting point is 0, the ending point is 4, and the arrows mean adjacency. To get a route, the program would go from the end point, which is 4, and take a connecting number, which is 2, and it would continue running unless it returns to the start point, which is 0. It will be 4 to 2, 2 to 3, 3 to 1, and 1 to 0, and then it is printed out in reverse order. This method fully works in my program and can print a route between 2 bus stops.

For handling errors, the dijkstraAlgo() function returns false if there are no bus stop IDs in stop_times.txt, and transfers.txt and Main.java prints out an error message. There is also a possibility that there is no path between two bus stop IDs and to solve this error it prints out an error message if the cost is 0.

For the implementation of the 2nd function, TSTTree.java, DirectedGraph.java, and Stop.java are mainly used and they are called from Main.java. In the constructor of the TSTTree, 2 functions are called for making a TST tree. The 1st function called makeStopNameArray(stop) takes an instance of the Stop class, where all the bus stop names are stored in a HashMap and, makes an array list that stores only those names formatted to non-space strings.

Once the function ends, another method called putStopNameToTST() runs. Each bus stop name string is converted to characters, and they are stored in an array list of char. In the put() function from putStopNameToTST(), it takes an instance of the TSTNode, where it has a left, middle, and right children of itself, an array of characters, and a boolean set as false, and uses a recursive method for setting its left, middle, and right children from a character of bus stop names to make a proper TST Tree. As I have learned from the classes, the put() function sets nodes and their children, and if a character of a child node is smaller than that of its parent node, it is set as a left child, if it is greater, it is set as a right one, and if it is equal, it is set as a middle one and the function goes to the next key character. The last character of stop names has a boolean set as true and it is used for the detection of stop names' ending. After putting all the characters of all the bus stop names into the TST tree, then the get() function is called for searching the string from users. Basically, this method does the same as the put() function; if a character is less,

it takes the left link, if it is greater, it takes the right link, and if it is equal, then it takes the middle link and moves to the next key character.
Once the function finishes searching all the characters of the string, it returns the node of the last character and starts finding bus stop names starting from the string of user input. This function is written under the function called findAllBusStops() and it uses a recursive method to do that. It goes to the leftmost and then goes to the middle nodes. After finishing the middle link, then it goes back to a previous character repetitively to find the right links of each node. Once it finds a node's boolean set as true, which means that is the last character of stop names, it stores the name string to the array list called foundStopNameArray.
To print out bus stop names that are found in the above function, spaces have to be inserted into the names since all of them are removed in the format function to make a valid TST Tree. I made a HashMap where it sets non-space bus stop names as keys and stores space positions of each name using array lists. In the function called print(), as you can see, it gets the array lists and inserts space at a correct position by using StringBuilder and insert() method.
For handling errors, the print() function prints out an error message when the size of the foundStopNameArray is 0 since the array should not store anything if there is no bus stop name starting from a user input character/s.

For the implementation of the 3rd function, Trip.java and DirectedGraph.java are mainly used and they are called from Main.java. One HashMap where arrival times are set as the keys is made at the same time DirectedGraph.java reads a line from stop_times.txt. This excludes all the invalid times of both arrival_time and departure_time to follow the assignment specification.
In Trip.java, it takes an arrival time from user input and gets all the details related to the time by using the HashMap as I explained above. In the function called makeTripListOfArrivalTime() in Trip.java, another HashMap where trip IDs are set as the keys, and they have the same arrival time from users is made since it needs to be sorted by trip ID. I made an array storing those trip IDs and it is sorted by the merge sort iterative method because I found it was the best way of sorting numbers, especially for handling many of them from the assignment in terms of time complexity. After sorting the array, all the details of each trip ID, which also has the same arrival time, are printed out by using the array and the HashMap setting trip IDs as the keys as I explained above.
For handling errors, the makeTripListOfArrivalTime() function returns false if there is no arrival time taken from user inputs in the HashMap tripLists setting the time as a key, and Main.java prints out an error message.

For the implementation of the 4th function, Main.java has the main method for calling all the other classes and functions. It makes instances of the DiretedGraph and Stop classes first as they are used in multiple functions. To enable selection between the functions, the switch case method is used, and the case number is taken from user input: case 1 for the first one, 2 for the second one, 3 for the third one, and 0 for exiting the program using a boolean called exit. For all the cases, I used a scanner.nextLine() instead of next() or nextInt() to handle errors because users may input more words or numbers by mistake. For case1, it reads a line that is separated by a space and makes an array for storing each bus stop ID by using split(" "). It gives them an error if the array does not have 2 bus stop IDs. For case2, the character/s which they input could include spaces since bus stop names contain them, and therefore nextLine() is the best for the implementation. For case3, as in case 1, it takes an arrival time from users which has ":" like 19:00:00 and makes an array from that by using split(":"). It gives users errors if the array does not contain three numbers, or the time is not valid. For all the cases, I set user inputs as strings since they may input words instead of numbers by mistake. In addition, matches() methods are used for checking if users input numbers for stop IDs in the case1, UPPERCASE characters, numbers, and "-" for bus stop names in the case 2, and numbers and ":" for an arrival time in the case3 by using a regular expression. For handling the error given when users input a character instead of a number, the matches() method is also used and if the input is a character, the case number is set as 100 to go to the default case.