## **Overview of the System:**

The Vancouver Bus System that I designed allows the user to make use of three different functionalities – searching for the shortest path between two bus stops (1), finding bus stop information by searching its name (2), and searching for all trips with a given arrival time (3). The system is implemented using the Interface class. This class allows the user to choose a functionality to utilize and makes calls to the classes implementing the chosen functionality. The called classes make use of both simple and complex data structures. After the chosen functionality has been implemented, the user will have the option to utilize another functionality or quit the system.

## **Interface:**

I used JOptionPane to implement the user interface and take all user input. The RunSystem() function presents the user with a welcome screen and three options – the three different functionalities outlined in the assignment brief. From there, depending on the user’s selection, the interface calls the takeUserInput() function from one of the three classes. Following the implementation of one of those classes, the user is brought back to the interface class and given the option to select another option or quit the system. A while loop was implemented to ensure that the system runs until the user decides to quit. The implementation of option boxes to record user inputs ensured that errors were avoided.

## **Shortest Path Class: Functionality 1:**

I used this class to find the shortest path between two bus stops provided by the user. It returns the cost associated with taking the shortest path and a list of the necessary stops en route. I implemented this functionality using Dijsktra’s shortest path algorithm.

Initially, my algorithm reads and stores the potential routes and the costs associated with them in a 2D array. A function is called to find the highest bus stop number in the stops.txt class. This would only need to be run once and hardcoded into the system, but I felt that it was a valuable part of my solution. As a result of its inclusion, the highest stop will be calculated each time the class is called. The highest stop is stored in an instance variable and the cost storage array is initialized. The algorithm filters through stop\_times.txt and transfers.txt, updating the matrix with the associated routes and costs.   
This approach allows the algorithm to easily access the costs associated with each different route. It also ensured that the edges were directed. I sacrificed space in order to achieve this – as not all numbers from 1-12789 corresponded to bus stops. I briefly considered the implementation of a 2D ArrayList to take advantage of the contains() function, but I felt that it would negatively impact the other aspects of my approach. In order to find the shortest path, Dijkstra’s shortest path algorithm is implemented. I used JOptionPane to display the path from source to sink and the cost associated with that particular path.   

## **SearchBusStop and TST: Functionality 2:**

This class allows the user to search for a bus stop by full name or by the first few characters. It returns the full stop information for all stops matching the search criteria. I used a Ternary Search Tree within this implementation. I imported the TST.java class from Sedgewick and Wayne’s online textbook – reference given in the .java file. By using this class, I was able to create and store bus stop names and their information in a TST. TST.java creates a symbol table with String keys and generic values, which perfectly fits the problem. In each key-value pair, the bus stop name represented the key, and a formatted String of its associated information was stored as the corresponding value. The use of LinkedList functionality within the TST class created a dynamic data structure – I edited the code slightly. This facilitated cost-efficient additions and searches of the TST.

Within the SearchBusStop class - I created methods to (1) take user input, (2) create the TST using the stops.txt file, (3) move the bus stop keywords, and (4) find bus stop information using the TST. I used a scanner to read the stop.txt file. Before pushing the keys (bus stop names) and values (bus stop information) into the TST, I formatted the Bus Stop name to fit the criteria listed in the assignment brief. I used the move () function to check if the word ‘Flagstop’ was present. If it was, I moved it to the end of the Bus Stop name. Following that, I check for other keywords using the charAt () function. If the stop name contained a two-letter keyword at the beginning of the name, I moved it to the end. If the stop name contained both the 'Flagstop' keyword and a two-letter keyword, both were moved to the end.

Within the findUsingTST () function, the keysWithPrefix function allowed me to easily implement the ‘first few characters’ functionality. I dealt with the ‘full name’ functionality by using the get() function within the TST class. A JOptionPane panel displays the stop name(s) along with all their relevant information – or a null message if a bus stop isn’t found.

## **SearchTripArrival: Functionality 3:**

I used this class to implement the third piece of functionality listed in the assignment brief. I used a HashMap to store trips according to their arrival time. I created a private class TripData so that each trip could be represented as an individual object. Within the private class, I initialized variables to store all trip information. ArrayLists of TripData objects were stored as values within the HashMap with their corresponding arrivals times as keys. By storing the ‘trips’ in an ArrayList, I was able to easily sort the trips under each arrival time using Collections.sort(). This method sorted the individual TripData objects by ‘Trip ID’.

I originally considered using a more exhaustive search progress. I filtered through the stop\_times.txt file and compared each trip time arrival to the user’s input. I broke down the arrival times into hours, minutes, and seconds and stored the values in individual arrays. I used these arrays to compare each of the 6 values in (HH:MM:SS). If an arrival time from stop\_times.txt matched the user’s input, I stored that trip information in an ArrayList. At the end of this process, I converted the ArrayList() to a regular array and passed it to a mergeSort function to sort by Trip ID. I decided against this implementation in favour of a HashMap. The HashMap approach facilitated faster average running time (constant), and also facilitated the storage of the stop\_times.txt information in a logical way. If I was to continue developing the software, and the user had the option to continue searching for bus times, until quit, the HashMap would facilitate a much more efficient process. In comparison, the other approach would have forced us to iterate through the 1.7 million lines of information in stop\_times.txt for each individual search.