Design Document

Arrival Time Search

Functionality:
Searching for all trips with a given arrival time, returning full details of all trips matching the criteria (zero, one or more), sorted by trip id

Data structures used:
Extracting trip details from stop_times.txt:
An Arraylist of ‘tripDetails’ objects I created was used. By iterating over each line of the input file, extracting the trip details into a ‘tripDetails’ object. For each, the ‘arrival_time’ and ‘departure_time’ are checked, ensuring they are within the minimum and maximum valid times, only if both are valid then the ‘tripDetails’ object is added to the Arraylist ‘trips’.

Sorting the trip details by ‘arrival_time’ and by ‘trip_id’:
Collections.sort was used. A custom comparator was created, which allowed the sorting algorithm to sort by ‘arrival_time’ (earliest to latest) and then by ‘trip_id’ (lowest to highest). This involved comparing the ‘arrival_time’ converted to seconds, and then if a match was found the ‘trip_id’s were compared.

To return a list of the trips to the user, when given an ‘arrival_time’:
I once again used an Arraylist to achieve this. Since ‘trips’ is sorted by ‘arrival_time’ and then by ‘trip_id’, it is simply a case of incrementing through ‘trips’, and if the ‘arrival_time’ matches the user inputed ‘arrival_time’, then the ‘tripDetails’ are added to a second Arraylist ‘tripsAtSpecifiedTime’.

Why / justifications:
The extraction of the trips into an Arraylist takes time complexity of O(1) for each insertion and space complexity of O(n). This is the same as data structures such as Stacks, Queues and Linked Lists, but I choose to go with an Arraylist as it was both what I was most familiar with and it would simplify the extraction and iteration process.
To sort the trips, Collections.sort was used which implements Timsort. This has an average asymptotic running time of Theta(n log(n)). This was both very efficient, occasionally faster than comparable sorting methods and allowed me to use a custom comparator with ease. This allowed for the trips to be sorted by arrival_time and then by trip_id, making returning results to the user much easier, as only the arrival_time needed to be checked. As such, sorting is only done once, and then it is simply finding matches and extracting in order.

Bus Stop Search

Functionality:
Searching for a bus stop by full name or by the first few characters in the name, using a ternary search tree, returning the full stop information for each stop matching the search criteria.

Data structures used:
Finding bus stop by full name or by first few characters:
To find all the stop names that have a certain prefix, I used a ternary search tree. By iterating over every character of the input string, it traverses the tree. Once it gets to the end

of the input string it returns all of the words that are in the subtree at that point. When reading in the stop names, if the name started with “WB, NB, SB, EB”, these characters were moved to the end of the string as suggested.

Finding the stop based from the stop name:

For every stop name found in three, I search for its corresponding stop using a hashmap. This hashmap maps from stop name to index location of a stop stored in an arraylist. These “stops” are a class containing all of the stop information, for example, stop id, stop code, stop name, stop description and so on.

Why / justifications:
To go from stop name to the corresponding stop in the arraylist I used a hashmap. Retrieval from a hashmap has average asymptotic running time of Theta(1). This means that going from the array of stop names with the user's prefix to the stops is really efficient. I could have avoided converting from the stop names to the index of stops by placing the index in the ternary search tree. Whenever the user searches for stops with a given prefix, the index is stored as a node in the tree. This would have been more efficient, however this would have added an unnecessary layer of complexity, because it is mixing two distinct problems into the one solution.

<u>Shortest Path</u>
Functionality:
Finding the shortest route and the cost associated with going between two bus stops, taking into account bus transfers and waiting times.

Data structures used:
Storing the graph:

The graph was stored as an adjacency list. Each stop had its own object containing it’s details, along with an arraylist of edges outgoing from that stop.

Calculating the shortest path:

Dijkstra’s algorithm was chosen to calculate the shortest path, as it is a single pair shortest path algorithm that is relatively easy to implement. Another reason for choosing it was that we had implemented it before in assignments.

Indexes to speed up performance:

A hashmap index was also created to map between a stop’s id, and the stop object. This allowed faster lookups of the stop object when creating the edges between the stops. While this increased the space required, it significantly improved the lookup time from a linear operation when searching through the array of stops, to a constant time one.

Why / justifications:
Dijkstra’s algorithm was an ideal candidate for computing the shortest path, as our application had no negative weights, and takes $O(V^2)$ time to run with our unordered adjacency list. Another option we could have gone with was Floyd-Warshall, which would allow our application to answer any subsequent queries about stops much quicker, but at the penalty of having to store the graph as an adjacency matrix, taking up $O(V^2)$ space, and taking $O(V^3)$ time to run. Instead, our adjacency list could be stored in $O(VE)$ space.