Graph.java

This graph forms the basis for the operations in parts 1-3. It sets up a graph of stops connected by transfers and trip segments. For nodes we used a String[] array as the number of nodes was fixed at the start by the number of lines in stops.txt. For the edges/transfers we used an ArrayList as this allowed for easier appending of new items. The TST for part 3 was also defined as a member of this class.

Part 1 (shortest path between stops)

For this we used Dijkstra's algorithm to calculate the shortest paths between stops. The reason for this was because we already had previous experience and code for it from our previous assignments. Based on the input file data, a series of stops are formed, all with connections defined. Using this, a graph can be built, on which Dijkstra's algorithm is used to determine the shortest paths between all pairs of stops.

Each path between 2 stops is defined as a transfer, with the relevant info taken from the input files. Then each transfer is compared with the list of nodes, gradually moving up the list of nodes until each transfer has been added to the relevant node.

In usage, a scanner takes the stop IDs from the user, checks to ensure that they are valid, and then, using the predetermined path, outputs it.
We calculated it works in O(V^2E) time, where V is the number of vertices and E is the number of edges.

Part 2 (Search for stops by name)

For this we used a Ternary Search Tree, as per the project specifications. This was represented using the TST class. The root of the TST was represented with a TSTNode object, and its children were represented by using the next[] field of the TSTNode. Other properties of each stop were also included in each TSTNode, although they were only relevant for terminal nodes, which corresponded to the last character in a stop name.

There are N stops added, and each is of length L. Thus all the insertions take O((L+ln(N)) * N) time. A similar number of operations is needed for retrieving a subtree, as all the nodes are traversed within the given subtree. One of the parameters of getSubtree() is also called using getNodeFromkey() which once again traverses the tree of nodes for a single path,

giving O(L+ln(N)), however this is insignificant compared to the previous parts. Thus the algorithm's worst case run time is in the order of O((L+ln(N)).

Part 3 (searching for trips with a given arrival time)

Our implementation of this part was relatively simple, consisting of three primary steps. First, we use a scanner to parse the input arrivalTime string to ensure that it is a valid time value (not above 23:59:59). Then we sort our list of transfers and iterate through it and output the information for each of the transfers that match the input arrivalTime, whilst keeping a tally of the total amount of results. Lastly we have some simple conditional statements to output the amount of found results, or lack thereof.

As we are just iterating through the list of all transfers, the performance of our function is O(N), where N is the number of total transfers in our dataset.