**Algorithms & Data Structures II**

**Final Project**

**Design Document**

# Design Flow:

I wanted to start designing the project incrementally, by doing the tasks one at a time and then fine tuning them.

So,

1. Begin Task 1
   1. Implement methods to read in all three files to begin filling data structures
   2. Implement a method add vertices and edges to an Edge Weighted Digraph
   3. Implement a method to run a Dijkstra shortest path algorithm and print the path.
2. Begin Task 2
   1. Build on reading in the stops to fill a Trinary search tree
   2. Edit the stop names, moving NB/EB/SB/WB from the start to the end of a string
   3. Implement a method to run a keys with prefix method and print all lines that match the criteria
3. Begin Task 3
   1. Build on reading in stop\_times.txt to fill a new data structure dedicate to arrival times
   2. Implement a method to add values to a red-black binary search tree
   3. Implement a method to return all lines from stop\_times.txt that contain the specified arrival time
4. Begin Task 4
   1. Implement a new file, that contains a main method with several loops to repeatedly ask the user for what method they wish to run and their inputs
   2. Begin Error Handling and Management
   3. Clean up the code

# Analyses

I used the package from ‘Algorithms, 4th Edition’ by Robert Sedgewick and Kevin Wayne to implement data structures like the Red-Black BST, TST and EdgeWeightedDigraph, and algorithms such as Dijkstra’s shortest path as their packages already have been optimised for both runtime and space complexity.

## Outside of tasks: Start up:

Before the program can run any of the tasks, it must complete two methods, *fillStops()* and *fillGraph(), fillStops()* uses a buffered reader to read in the stops.txt file line by line, in it lies a while loop that will run until it has read every line in stops.txt, hence we can consider so far a worst time of O(N), almost all operation occurring in that while loop are constant time operations including adding to an arraylist, hashmap and searching the hashmap for a key, but it calls another method to fill a TST, which must check if a key is present first, which has a worst case time complexity of O(LogN), hence we can assume the worst case runtime for *fillStops()* is O(NLogN) and a space complexity of O(N). I used the hashmap instead of another hashtable to allow a stop name to reference it’s stop ids as several stops have multiple ids, as it contains an arraylist, which is nullable, I had to use a hashmap to store the information as it allows nullable values without causing a nullPointerException. But I used a hashtable to quickly refer a stop id to its index in the arraylist ‘stops’, which allows for a constant time for a get and as the hashtable has both keys and values that are integers that which cannot be nulled. The brief required the use of a TST for searching for stops, so The TST is updated here.

For *fillGraph()*, the construction of the digraph has a time and space complexity of O(V + E), the number of vertices + the number of edges. The method is comprised of two sections, one to read in in transfers.txt, which it’s process only contributes to the total space time complexity, and the other reads in stop\_times.txt, that however references another method that fills a red-black binary search tree, which uses contains and put which take O(LogN) time. Hence knowing E1 + E2 = E, the worst case runtime for *fillGraphs()* is O(V + E1 + E2LogE2) with a similar space complexity. I used an Edge Weighted Digraph because, the graph I was constructing has directed edges with weights/costs hence it was the only data structure that suits, the arraylist ‘stops’ indexes every stop from stops.txt when reading them in, to use the Edge Weighted Digraph which can only operate on vertices 0-(V-1) where V is the number of vertices. The method also updates the red-black search trees. The outer tree has keys referring to the arrival time in seconds, this allows us to quickly get to only the trips that arrive at that time, the inner tree has a key of the trip id, and a value of the string in the file, so we want to return the entire inner tree of the matching arrival time, and this allows us to do so quickly.

## Task 1: Shortest Path Algorithm:

The shortest path algorithm starts with several error checks that have a constant run time, then it reads in values from a hashmap and hashtable for the algorithm, each only have a runtime of O(LogN), before reaching a construct for the Dijkstra shortest path algorithm which has a time complexity of O(ELogV) and O(V) extra space. *hasPathTo()* is a constant time operation but *pathTo()* has a worst case time complexity of O(V). Hence we can assume the method has a worst case time complexity of O(V + ELogV) and addition space complexity of O(V).

## Task 2: Searching stops Algorithm:

Task implements a search algorithm on a ternary search tree as per the request of the brief. The method uses several lines in constant time, followed by keysWithPrefix(), which searches all subsequent branches for keys that contain a value, we can therefore assume the worst case runtime to be O(N), if the tree was to return every key. I decided to re-read in the file to save on space but only print out the lines required, so I had to sort the keys by their values with a *Collection.sort()* which has a runtime of O(NLogN). The method therefore only has a runtime of O(NLogN) and space complexity of O(N).

## Task 3: Searching Times Algorithm:

The brief asks for returning all trips that arrive at a specified time, sorted by their trip ID. stop\_times.txt has well over a million lines, hence using the same method as above wouldn’t be effective and would require me to take more space and another sort. So, I decided to implement a data structure that sorts itself on every input and has a fast method to get and put values. I decided to use a nested red-black binary search tree, the outer representing the time in seconds and the inner representing. We can therefore assume there is a time complexity of O(M+ LogN), where N is the number of times in seconds and M is the number trips with that arrival time. But there is a space complexity of O(M) for an Iterable containing all the keys we need to print.

## Task 4: Interface Program:

A very simple approach, simply using some while loops and Booleans affected by inputs, the program will run so long the user allows, with constant time and space.

# References:

Algorithms 4th Edition: https://algs4.cs.princeton.edu/home/

Red-Black Binary Search Tree: https://algs4.cs.princeton.edu/33balanced

Directed Edge: https://algs4.cs.princeton.edu/44sp

Edge Weighted Digraph: https://algs4.cs.princeton.edu/44sp

Dijkstra’s shortest path: https://algs4.cs.princeton.edu/44sp

Ternary Search tree: https://algs4.cs.princeton.edu/52trie