# [CSU22012-202122](https://tcd.blackboard.com/webapps/blackboard/execute/courseMain?course_id=_71864_1" \o "CSU22012-202122 Algorithms and Data Structures II) Algorithms and Datastructures 2

20334565 Declan Quinn

**Data Storage**

I initially started working on this project by storing all the data regarding my algorithms in separate arrays of objects read from the input files. I had made three object classes Stop, StopTime and Transfer which would contain the necessary data required as read in from their corresponding input files. My initial struggle was that when I began to access data from these arrays the run time was poor as I at one point had to wait several minutes to return the required data needed from the arrays. I had researched on fast retrieval datastructures and found hash maps. What I found ideal about using a hashmap to store data was that in many instances I could store the unique key for an object with a unique property it held, for example stopID was used as the key for each stop in stopMap. However, the issue I ran into using hashmaps was that certain keys would be overwritten when I wanted to store multiple values with the same key. The solution to this was to place linked lists as the value stored by each key, therefore in the case of duplicate data being written to the same point that data could be appended to the linked list. The result of using hashmaps sped up fetch times significantly.

Overall I my main datastructures used to store data from the input files was approximately of size N (stop object size) in a stop array + +N(stop object size) in a hashmap of stops + N(linked lists of size L) (size of edge objects) in an edge hasmap for each stop + T(linked lists of size X)(size of time objects) in a time hasmap. As times was by far the largest input file the size of the hashmap of linked lists by far took the most memory to store.

All data was read and stored in a single Map object, all data accesses in main would be through this object. The run time of storing all this data is very slow and due to this the program takes about 25 seconds to write all this data once the program starts. Doing this however is worth the speed up in performance we get when running our searches.

**Shortest path between two stops**

I used Dijkstra’s algorithm to get the shortest path between two stops as its performance is determined by its priority queue implementation, here the minimum priority queue follows a binary heap, therefore insert, deleteMin and decrease key are all logV operations for size V vertices. In the worst case we need to account for V pops from the priority queue. It should be noted that for each vertex V it contains some E edges that need to be relaxed during each run hence the worst case runtime is (V+E)log(V). We can discount the N runs of setting the stops to infinity as they should be far less than the number of edges and vertices we need to account for. Dijkstra can be used here as there is no negative weights as distances hence a shortest path will be guaranteed if one exists. I used an array for the stop objects as I could easily duplicate this array for my distTo and edgeTo arrays required for Dijkstra’s algorithm. I could use the indexed locations that the array was written to store my values in distTo and edgeTo, this in turn allowed for fast accesses as I could easily acquire the location I was writing to and the stop object as they both shared the same index.

I used a hashmap implementing a linked list to store the edges connecting each vertex. I chose to use this as fetching from a hashmap is in constant time given the correct key. As the key I used was the stop ID I could search for a stop ID and return a linked list of edges adjacent to it. The linked list had a runtime of N depending on how many edges were written to it. This proved to be significant as some linked lists contained large numbers of stops that had to be iterated through. Using a linked list proved efficient as it could add nodes in constant time instead of an array which would have to resize itself after every insertion. The only downside to reading from the linked list was that I could not access elements in constant time, however as in many cases I had to iterate through all the elements anyway this became irrelevant.

**Searching for bus stop by name using a TST returning stop information**

I was required as of the assignment outline to use a Ternary Search Trie (TST) to search for each stop that matched a given partial or complete input string. The runtime for a TST on average is about L + log(N) for a search hit and log(N) for a search miss, this is due to the array having to search through less characters to reach the key in a miss. An insert on average takes L + log(N) as the key has to be stored L new character nodes from the nearest stored node. I had used the Princeton example of implementing a TST in my assignment. The function keysWithPrefix(String prefix) returned a Queue of stops that had been searched. This proved useful as I was able to dequeue each stop in constant time and then get the Id of said stop by passing the string name for each stop back into my TST to get the stopID value it was stored with. This would allow me to check my hashmap of stops and search for the stop by its ID in constant time to return its values.

**Searching for trips given an arrival time**

I had created a hashmap in memory containing a linked list of all times and the corresponding trips at those times. This proved to be difficult memory wise as stop\_times.txt was the largest input file and it took up significant space in memory. I chose this as I believed it was necessary for constant fetch times as when passed a time key, the hashmap would return a linked list containing all the data from stop\_times.txt at a given time. This proved invaluable for part three of the project as the worst case runtime could only be N (The length of the linked list stored), however due to the amount of times given in the file searching in a given linked list was negligible. The main issue was sorting the linked list of data in order of tripID. As the data was stored in objects it was difficult to code an algorithm that would sort them by a primitive value. I used Collections.sort over an arrayList to sort objects by tripId MergeSort in this function, MergeSort guarantees Nlog(N) run time in worst and average cases and is significantly faster than quicksort when dealing with objects.