1

How well did the data structures perform for the assigned application?
The ADT implemented in this assignment was a directed graph (a digraph). The assigned functions or tasks for this ADT to perform are building the graph or inserting vertices in non-null positions, displaying all next vertices (this is a recursive depth first search given a vertex v). Further a display all and remove all functions are also implemented. The structure performed these assigned tasks well, compared to most other data structures. The reason why this data structure performed the task well was because that there are relations between the given vertices. These relationships are the unweighted paths between the given vertices that allow the implentation of the graph algorithms like topologically sorting (not implemented) and if weighted then kruskal can be implemented.

2

Would a different data structure work better? If so, which one and why.
The assigned task would be difficult to manage with a tree which would be the only data structure capable of handling this assigned task. If the graph were weighted the tree would work with a huge memory over head (extra pointers, weights between each relationship of parent and children). This data structure still would be a pointless implentation since the graph can be decomposed into a tree using kruskal, prim or even using a topological sorting to make a linear list. Another close data structure that might be implented is a hash table which doesn't solve the connections between each of the nodes. The hash table has a similar structure being an array of linear linked lists but cannot solve the problem using the functions implented in a hash table. So for the assigned application a graph is the best structure to use.

3

What was efficient about this program design?

The efficiency in this design relied on the implmentation of the adjacency table which allows the relationship between each vertex node and the linked chains (edges). This allows for concepts like cycles between the vertices. This cannot be done with a tree structure, otherwise it wouldn't be a tree. The efficiency of the design of the graph structure is really about the problems that is being solved with the graph. For example trying to solve a sort for a given dataset would be inefficient to try to solve with a graph. The efficiency of the aglorithms implented mostly consist of simple loops and have $O(n)$ efficieny. The most interesting algorithm is the depth-first search which does not have a linear time efficiency. For the worst case the time is $O(V)$ where V is the vertex that the depth first search is being started from. Then the depth first finds all the vertices adjacent to the vertex and visit the adjacent vertices.

4

What was not efficient about this program design?
The inefficiency in the program design is that the program limits the amount of vertices that can be added to the table. Since the adjacency list is an array of linked lists the array can't be stretched beyond the parameter set at compile time or run time. This leads to issues, for example in the assigned application, if the user finds another ride, the ride vertex cannot be added to the graph without deleting and reimplenting the table. So the table size of the adjacency list plays a big part in the run time memory and implementation of the program.

5

What would you do differently?
Adding weights to the graph allows for an array of more intereting algorithms that can be implented Floyd's

form continued on next page...

algorithm given that the graph is both weighted and directed or Dijkstra's algorithm if it is not directed. Finding the minimum spanning tree can also be done using Prim's algorithm if undirected and Kruskal's algorithm if directed. Each of these algorithms have their own advantages, using the Floyd's algorithm, the user can find the shortest but compete path in the graph. Using Kruskal's algorithm the user can find which rides to finish first based on the edges. Since these algorithms were not required in the assigned application, they were not implemented but can be.