ECS 60 **Previous Final ADT Essay Questions Keys**

1. ADT design (80 points) You've been assigned to design an assembly line for an automobile manufacturing plant. An assembly line is a linear set of stations. At each station, people begin and complete specific task(s). For each task your will program will receive: 1) The time it takes to complete the task, and 2) the previous task(s) that the current task is directly dependent upon being completed before it can start. There are T tasks, and a total of D dependencies for all of the tasks.
   a) (20 points) If you assume that only one task can be done at each station, how would your program determine an ordering of the tasks that will take the minimum time? In terms of D and T, what is the big-O of your solution?

   I would construct an event-node graph in O(D + T) time. I would then use a topological sort to determine the order of tasks. This will take O(D + T) also. Any topologically correct ordering will produce an optimal time since there is no wait between tasks.

   b) (60 points) If you assume that up to two tasks can be done at each station simultaneously, how would your program determine the ordering and station of the tasks that will take the minimum time? In terms of D and T, what is the big-O of your solution?

   This is more complicated than a). I would again construct an event-node graph and create a topological sort in O(D + T) time. I would then determine the slack time of all the tasks. Using the topological sort this is again done in O(D + T) time. There is at least one path consisting entirely of zero-slack edges; the critical path(s).

   Each vertex would have associated with it a list of vertices that directly depend upon its completion. These associations could be constructed in O(D) time. Each vertex would have associated with it an integer, Unfinished, set to the number of tasks upon which it is directly dependent.

   I will have two queues, one for critical path vertices, and one for non-critical vertices, that have had all of their dependent tasks completed. These queues will have some vertices in them from the beginning. (Note that any O(1) storage container could be used. The FIFO characteristic of a queue is not necessary.)

   Now we are ready to construct the assembly line. The program would then work its way through the two queues giving preference to the critical list queue. For each station, it dequeues up to two from the critical queue. If the station does not have two critical tasks assigned to it, then one or two tasks are dequeued from the non-critical queue. After the station is filled, the Unfinished counter of the tasks dependent on the station's tasks would be decremented. If a task's Unfinished became zero, then it would be put in the appropriate queue.

   This whole construction phase would take O(T) for enqueueing and dequeueing. Updating the Unfinished counters would take O(D) time. So the whole process will take O(D + T) time!

2. ADT design (70 points) For this problem you will address two sides to the railroad business---financial and logistical.
   a) (50 points) Many railroad cars are owned by companies other than the railroad hauling them. The railroads have a contract with car owners to assure that the cars must earn at least a certain minimum amount each month. If a privately owned car is underused, a railroad must pay the owner a penalty that is geometric to underuse, e.g., the penalty for a car that misses its target by $10 might only be $5, but the penalty for a car that misses its target by $100 might be $500. Thus, the railroads must make sure that the most underused cars at a desired location are assigned work first. Therefore, they have two operations: 1) Updating the earnings of a car, based on its ID, when a cargo is delivered to a location; and 2) selecting unused cars for use based on their earnings and location. Describe and justify your choices of data structure(s) that would minimize the time used by the system. Be sure to describe how the two operations would work, and the big-O(s) involved.

   Here are some further specifications: There is plenty of RAM. There are 100,000 cars, and 1000 locations. Each car is used on an average of 10 times a month. Insertions of new cars and deletions of old cars must be possible, but will not be explored for this question. The contracts for all the cars are the same. Locations are unique shorts, car IDs are unique ints, and car earnings are floats. When a car makes a delivery, it becomes unused.

   I would have a class, CarClass, that contains the car ID and earnings of a car. I would use two data structures, one for unused cars, and one for cars that are being used.

   For unused cars I would have a quadratic probing hash table hashed on locations. Each entry in the hash table points to a MinHeap of CarClasses that sorts based on the earnings of each car. The hash table would be some prime greater than 2000 to guarantee a load factor less than 0.5, and a search of O(1) for cars available at a location. The MinHeap would be a vector that would double in size when needed. This doubling, O(N), would be rare, and easily amortized into the O(log N). Selecting, and removing the car that has the minimum earnings at a specific location would be O(1) + O(log of the number of cars at the location). After the car is removed from the unused car data structure it is inserted into the used car data structure.

   For cars in use, I would use a quadratic probing hash table of CarClass objects based on the CarIDs. By using a table with a size of the next prime after 200,000, a load factor less than 0.5, I assure that all operations will be O(1). When a car arrives at its destination, it is removed from the hash, then its earnings are updated, and it is inserted into the unused data structure based on the location of the destination.

b) (20 points) A railroad dispatcher must determine the best way to route rail traffic so that the maximum amount of cargo can be transported. Each rail connection between cities can only handle a certain number of cars a day. The dispatcher has a list of all the connections available to his/her railroad and the number of cars each can carry. The New York dispatcher often has to determine the best way to send hundreds of cars from New York to Los Angeles on any given day. No single route can handle all of the traffic. Provide a method for the dispatcher to determine the best routes for the New York cars so that as many cars as possible reach Los Angeles without becoming stuck somewhere. Be sure to explain how your method would be used with this particular problem.. We are not looking for a data structure, nor big-O.

This problem easily translates into a network flow problem. The nodes would be the cities. The edges would be the rail connections on the dispatcher's list. The capacity of each edge would be the number of cars that connection can carry. The source would be New York, and the sink would be Los Angeles.

3. ADT design (78 points) There are now programs that will give you street directions to get from any city in the United States to any other city in the United States. For this program, assume that cities are completely separated from each other, and are only connected by freeways. You are to describe <u>all</u> of the algorithms and data structures necessary to implement such a program efficiently given the following specifications. Remember to mention big-O wherever appropriate.

1. The continental United States has 20,000 connected cities (**C**), and 25,000 freeways (**F**).
   - 1.1. Each city has a name <char [44] >, state <char [2]>, and unique city ID <int>, for a total of 50 bytes.
   - 1.2. Freeways.
     - 1.2.1. Specified by its name <char [4] >, city1 ID <int>, city2 ID <int>, length in feet <int>, and a unique road ID <int>, for a total of 20 bytes.
   - 1.3. Intersections
     - 1.3.1. There is one intersection in each city. Thus, there are as many intersections as cities, **C.**
     - 1.3.2. Specified as a list of the road IDs of the Freeway(s) that meet at one physical location, and the ID of the city in which the intersection occurs <int>. No more than nine Freeways can intersect at one physical location, so an intersection can be no more than 40 bytes.
2. As input your program is given the city, and state of both the origin and destination.
3. Your program should provide an ordered list of the IDs of the Freeway(s) that is the shortest (in feet) route from the origin to destination.
4. Do not worry about insertions, deletions, or updates.
5. Your program can use 10 Megs of RAM.

First we will store all of the city information in a quadratic hash table with a load factor of 0.5. This will take 2* 20,000 * 50 = 2 Megs of RAM. We will use the city names as keys, and hash to find the two City IDs in O(1) time.

We can use Dijkstra's shortest path algorithm to solve the problem. The intersections would be the vertices, and the freeways would be the edges with their length being their weights. We can set the freeway intersection of the origin city as the distinguished vertex, and then use the shortest path algorithm until the intersection of the destination city is known. Unhappily, there can be cycles in the graph, so our algorithm cannot use a topological sort.

The Freeways themselves take up 25,000 * 20 = 500,000 bytes. If we put them in a quadratic hash table using their ID as the key, with a load factor of 0.5, it will take up one million bytes and have an O(1) access time.

To use Dijkstra's algorithm we will create a class that will contain an intersection, a bool indicating whether it is known, a distance <int>, and the ID of the previous Freeway ID. Each object of this class would take 40 + 1 + 4 + 4 ≈ 50 bytes. If we put these in a quadratic hash table with a load factor of 0.5, we have C * 50 * 2 = two million bytes. The key will be the City ID and the access time will be O(1).

There are three costs involved in Dijkstra's algorithm. First initialize the table. Since we have twice as many entries in the hash table as actual intersections, this will mean that 2 * C * 3 values are initialized which is still O(C) total time. Second, there is the cost of updating the vertices. This will be done O(F) times and, because we are using a hash table for the intersections, take only O(F) total time.

Finally, there is the cost finding the next minimum vertex to add to the path. If we simply search our hash table, we have (2 * C) entries searched up to C times which is $2C^2$ which is $O(C^2)$ total time. If the destination city is the last one set to known, there would be a total of 2 * 20,000 * 20,000 = 80 million comparisons! This is not the way to go for this problem, and will not receive full credit. Using a heap of pointers to the intersections to store the updated intersections is much better. There are at most F updates and F DeleteMins for 2 * F logF which is 50,000 * 16 = 800,000 comparisons. Remember that this is worst case, in reality many vertices would never be removed and the heap itself will never hold F values. In any case the heap method is 100 times faster than 80 million! The heap would take up F * 4 bytes/pointer which is only 100,000 bytes.

Once the destination city intersection is known, we can stop the algorithm. Starting at the destination city, we push the previous Freeway IDs onto a stack in O(C) total time and then pop them off to the user in O(C) time.