Data Structures and Algorithms  
Week 3 problem sheet

## Maps

Use the MapDemo.java class from the code bundle to answer the following questions about maps.

1. How do you enter a new pair into a map object?
2. Maps have at most one value per key. What happens if you enter a pair with a key that already exists in the map?
3. How do you find all the keys in a map (the map’s domain)?
4. How do you find all the values in a map (the map’s range)?
5. Describe two ways to remove a pair from a map.

* Questions 1–5 are intended to give you practice writing your own code in Eclipse, so full sample answers are not provided.
* Some brief hints, however, on what methods of the Map class you should be looking at:
  + for question 1, check out the put() method
  + for question 2: you should try this and find out. (Or check the documentation.)
  + for question 3, check out the keySet() method
  + for question 4, check out the values() method
  + for question 5, check out the remove() and clear() methods (but note that the latter removes all other keys, as well
* If you are not sure how to check whether your answers are correct, feel free to contact the lecturer.

## Collections API

1. Consider the four core interfaces of the Collections API: Set, List, Queue, Map. For each of the four assignments below, specify which of the four core interfaces is best-suited to the problem, and explain how to use an implementation of it to implement the assignment. You can complete the code for this in CollectionsDemo.java.
   1. Whimsical Toys Inc (WTI) needs to record the names of all its employees. Every month, an employee will be chosen at random from these records to receive a free toy.
   2. WTI has decided that each new product will be named after an employee but only first names will be used, and each name will be used only once. Prepare a list of unique first names.
   3. WTI decides that it only wants to use the most popular names for its toys. Count up the number of employees who have each first name.
   4. WTI acquires season tickets for the local lacrosse team, to be shared by employees. Create a waiting list for this popular sport.

* A brief guide to approaching this sort of question:
* a. A List would be fine here. Every item in a list has a specific position in the list. So to choose a random employee from a list of (say) 25 people, you just need to generate a random number from 0 to 24, and look at the name in that position.
* It is *possible* to use a Set instead, but it would require use of the Iterable class, and we won’t examine this in detail.
* b. Whenever we see a requirement to record “unique” items of some sort, it is worth considering whether a Set might be suitable, because Sets only store unique values.
* And in this case, a Set is indeed the best-suited structure.
* c. Whenever we have a collection of things, and want to store some property “about” them, it is worth considering whether a Map would be a good fit.
* In this case, a Map is indeed the best-suited interface. We could use a map of type Map<String,Integer> to store each first name (a String), and associated with that first name, the number of employees who have that name,
* d. Whenever we are asked to maintain some sort of “waiting list” or “queue” or “prioritized list” of values, it is worth considering whether a Queue might be the best data structure - and it is indeed the best-suited one, in this case. We could use a Queue to implement a waiting list maintained on a “first come, first served” basis, OR, if there were some employees who had specifl priority, we could use a Priority Queue.
* In more detail:
* For a.:
* In order to select an employee at random, we need to be able to: (i) find out how many employees there are; (ii) generate a random number from 0 to (number of employees - 1); and (iii) get a reference to a particular employee’s name.
* Any of the interfaces will let us perform task (i), and task (ii) is independent of what interface we choose. But task (iii) will be much more straightforward if we use a List interface.
* Our code would be something like (assuming our list of names is called employeeNames, and is of type List):
* Random r = new Random();  
   int i = r.nextInt(employeeNames.size());  
   System.out.println( employeeNames.get(i));
* (Assuming we just print the employee name to standard output.)
* For b.:
* If we need to get a unique list of anything – e.g. a list of unique first names – then Set is the most appropriate interface, since sets do not permit duplicate values.
* We assume here that each name in the variable employeeNames contains two names, a first name and a last name, with the first name appearing first, and with a space between the two names. (However, any reasonable assumptions about what the variable contains are okay.)
* Code to get a list of unique first names would then be:
* Set<String> uniqueFirstNames = new TreeSet<>();  
   for (String employeeName : employeeNames) {  
   uniqueFirstNames.add( employeeName.split(" ")[0] );  
   }
* (See the documentation of the [String](https://docs.oracle.com/javase/8/docs/api/java/lang/String.html) class for details of the .split method.)
* For c.:
* In this example, we need to record a value – namely, the total count, or frequency of the name – for each first name. This means that we need to maintain a map from names to name frequencies, and a Map is the most appropriate interface.
* Code to do this would be:
* Map<String, Integer> nameFrequency = new TreeMap<>();  
   for (String employeeName : employeeNames) {  
   String firstName = employeeName.split(" ")[0];  
   if (nameFrequency.containsKey(firstName)) {  
   int currFreq = nameFrequency.get(firstName);  
   nameFrequency.put(firstName, currFreq + 1);  
   } else {  
   nameFrequency.put(firstName, 1);  
   }  
   }
* Here we use a TreeMap, but any Map implementation should work.
* For d.:
* Any time we need to maintain a waiting list or queue of some sort, that is a strong hint that we need to use a Queue interface.
* From the wording of the question, it seems we only need to declare and initialize the queue – not populate it.
* We make the assumption that the waiting list should be “first in, first out”, and thus a normal queue is sufficient. If some employees had special priority, we would instead need to use a PriorityQueue.
* So to declare and initialize the queue, we can write:
* Queue<String> waitingList = new LinkedList<String>();
* (because LinkedList implements the Queue interface).

## Hash tables

1. What is a perfect hash function? Explain why it is almost always impossible to have a perfect hash function.

* A perfect hash function is defined as one which maps each input to a unique array position. However, in many cases, the number of possible inputs is theoretically infinite (e.g. “all possible strings”), or far larger than the array available to store them in. So a perfect hash function is therefore impossible.
* (If we have a very *small* number of items to store, then a perfect hash function does become more plausible; but for a small number of items, there is probably not much advantage in choosing a hash table over, say, just an array.)

## Graph algorithms

Use the AdjacencyMatrixGraph.java from the code for week 3 to answer the following problems on graphs.

1. Use the graph generation methods to generate some random graphs of different densities and see how graphs are represented using an adjacency matrix.

* You should try this out using Eclipse.

1. Describe (in words) an algorithm to count the number of edges in a **directed** graph using the adjacency matrix.

* Let our matrix be . Keep track of the total number of edges seen. Iterate over all the row positions (from 0 to   m.length - 1) and, for each row, all the possible column positions (from 0 to   m.length - 1). If     m[rowIdx][colIdx] == 1  , then add one to the total count of edges. The result will be the number of edges, since each edge will correspond to a 1 in the matrix.

1. Describe how to count the number of edges in an **undirected** graph using the adjacency matrix.

* This is exactly the same as for a directed graph, except that, at the end, we divide our total by *two*.
* In an undirected graph, if we have an edge from node **A** to node **B**, then, necessarily, that means we have an edge from node **B** to node **A**. So each edge will apear twice in the matrix.

1. Describe how to count the number of edges in an **undirected** graph using an **adjacency list** representation of a graph.

* Let our array of linked lists be **adjArr**.  
  We keep track of the total number of edges seen.  
  For each element of the array   adjArr, we calculate the length of the linked list it points to (by iterating over each node in the list). We add this length to the total so far.  
  At the end, we divide the total by two: because this is an undirected graph, the number of nodes will be twice the number of edges.

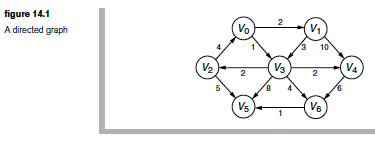
1. Describe how to generate the list of neighbours for a given node using an **adjacency matrix**.

* Let our matrix be **m**.  
  Suppose that we want to get the list of numbers for some node **i**. (We assume here that each node is just identified by an index position into the matrix.)  
  Assuming this is an undirected graph, then we just need to find all the times 1 appears in column **i**, and record the row positions. (Or, alternatively – we could identify all the times 1 appears in row **i** – in an undirected graph, they will be the same.)  
  So we iterate over all the row positions (from 0 to   m.length - 1) for column **i**, and each time we see a 1, we add our current row position to our list of neighbours.
* NB: the question does not ask for any code to be written – but if you wanted to, it would look like this:
* // assuming our matrix is m, and the node we are  
   // interested in is i  
   List<Integer> neighbourList = new ArrayList<>();  
   for (int rowIdx = 0; rowIdx < m.length - 1; rowIdx++) {  
   if ( m[rowIdx][i] == 1 )  
   neighbourList.add( rowIdx );  
   }
* This iterates over a row; very similar code could be used to iterate over a column.

1. Describe how to generate the list of neighbours for a given node using an **adjacency list**.

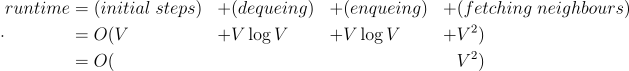
* Let our array of linked lists be **adjArr**.
* To get the list of neighbours for some node **i** (we assume each node is identified by its index position in the array), we just return   adjArr[i].

1. On paper, show the steps followed by Prim’s method for calculating a **minimum spanning tree** for graph in Figure 14.1 (Weiss) below. Start your tree from vertex V0. You could write some code using PrimMST.java to check your answer.

* 
* For this question, it is suggested you check your answer by writing code using   PrimMST.java.

1. PrimMST can be implemented using either an **adjacency matrix** or an **adjacency list** to represent a weighted graph. What difference does the choice of implementation make to the big O complexity of the algorithm? Why?

* Recall the pseudocode for Prim’s algorithm:

* 1: inital step:  
   2: - for all vertices v ∈ V:  
   3: colour[v] = white  
   4: key[v] = ∞  
   5: π[v] = undefined  
   6:  
   7: pick some starting vertex s ∈ V, enqueue it into Q  
   8: key[s] = 0  
   9: colour[s] = grey  
   10:  
   11: while Q is not empty:  
   12: w ← extract-min from Q  
   13: for each v ∈ neighbours(w):  
   14: if colour[v] is white:  
   15: colour[v] = grey  
   16: key[v] = edgeWeight(w,v)  
   17: π[v] = w  
   18: enqueue v into Q using key[v] as priority  
   19: else if colour[v] is grey:  
   20: if key[v] > edgeWeight(w,v):  
   21: key[v] = edgeWeight(w,v)  
   22: π[v] = w  
   23: colour[w] = black
* For an adjacency matrix implementation, we can analyse the runtime of the algorithm as follows.
* - The *initial steps* (lines 1–9) are run times.
* - At some point in the algorithm, every node in the graph will be *dequeued* (line 12). Dequeueing from a Java priority queue takes time (see the [API documentation](https://docs.oracle.com/javase/7/docs/api/java/util/PriorityQueue.html)), so the total cost of dequeueing operations will be .
* - Note the code for *enqeueing* a node (line 18). How many times will this happen? Not every time around the while loop, because line 18 is in an if statement.  
  What we do know is that at some point in the algorithm, every node in the graph will be enqeued. So the number of times it happens is ; and since the costs of enqueueing is , that means the total cost across the whole algorithm of enqueueing operations will be .
* - We also have to fetch the list of neighbours (line 13) for a node. For an adjacency matrix representation, the cost of doing so is ; it is done once for every iteration of the while loop, so the total cost is .
* So, the running time will be
* 
* For an adjacency *list* representation, we analyse the runtime as follows.
* - The *initial steps* (lines 1–9) are still run times.
* - At some point in the algorithm, every node in the graph will still be *dequeued* (line 12). As before, the total cost of dequeueing operations will be .
* - The cost for enqeueing nodes likewise remains the same, .
* - However, fetching the list of neighbours for a node (line 13) will be different. The worst-case cost of doing so for for an adjacency list is , where is the number of edges in the graph.  
  Does this mean the cost of fetching neighbours is , since the while loop runs times? Not exactly. Different nodes will have different numbers of edges, so the cost will vary – they can’t *all* have -many edges. What we do know is, every edge will be retrieved *once*. So the *total* cost of retreiving edges, across the whole algorithm, will just be .  
  Similarly, any operations in the “for each neighbour” loop will run times.  
  So the total cost of these operations will be .
* That means the grand total cost of the algorithm for an adjacency list representation is:
* 
* Which of these terms will predominate? We don’t know: it depends on how *dense* the graph is – what proportion of nodes have edges between them.  
  In a very *dense* graph, the number of edges will be close to – every node is connected to every node – so , and the running time won’t differ from an adjacency matrix representation.
* However, in a very *sparse* (but still connected) graph, the number of edges can be as low as – just one edge between every node.
* In such a case, the term would predominate, and the total running time for the algorithm would be .
* So the answer is, we need to know how dense the graph is to know which terms will predominate; depending on the density, the runtime can be anywhere from to .

1. Test your answer to the previous question by implementing an adjacency list version of the algorithm. You can use PrimMST.java as the basis of your second solution. Run some experiments (as in the week 1 homework) to test the run times of these two algorithms. What do you notice?

* For this question, it is suggested you check your answer by writing code using   PrimMST.java.