Data Structures and Algorithms  
Week 4 problem sheet

## A. ADTs vs data structures

1. Explain which of the following are *abstract data types*, and which are (concrete) *data structures*:
   * linked list
   * array
   * queue
   * binary search tree
   * map
   * binary tree

* **Solution**
* Linked list, array, binary search tree and binary tree are concrete; queue and map are ADTs.

1. Suppose we wanted an abstract data type that represents a *Door*, for a computer system that remotely controls all the doors in the building (whether they are locked, unlocked, open, or closed).

* What *operations* would you define for this abstract data type?
* How would you implement it in Java (as an interface)?
* **Solution**
* As operations, we want two “accessor” operations, e.g. “isLocked” and “isOpen”, to tell what the state of the door is, and several “mutator” operations to change the state: “lock”, “unlock”, “open” and “close”.
* So our Java interface might look like this:
* interface Door {  
   bool isLocked();  
   bool isOpen();  
   void lock();  
   void unlock();  
   void close();  
   void open();  
   }

## B. 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.

* **Solution**
* 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).

## C. Priority Queues

1. Suppose we perform the following operations on a Priority Queue that uses ints for priority, and stores Strings as values:

* enqeue("alice", 1),  
   enqeue("bob", 5),  
   enqeue("carol", 3),  
   deqeue(),  
   deqeue(),  
   deqeue()
  1. What Java code would we use to implement this (using the linked list implementation from the sample code)?
  2. What will be the contents of the queue after each operation?
* **Solution**
* For question (a), the Java code for these operations is:
* LinkedPriorityQueue<String> queue = new LinkedPriorityQueue<String>();  
  queue.enqeue("alice", 1),  
  queue.enqeue("bob", 5),  
  queue.enqeue("carol", 3),  
  queue.deqeue(),  
  queue.deqeue(),  
  queue.deqeue()
* The contents will be:
  + (alice, 1)
  + (alice, 1), (bob,5)
  + (alice, 1), (carol,3), (bob,5)
  + (carol,3), (bob,5)
  + (bob,5)
  + (empty)

## D. Graphs

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].