Question 1. (6 points) A *priority queue* is a data structure that supports storing a set of values, each of which has an associated key. Each key-value pair is an entry in the priority queue. The basic operations on a priority queue are:

- `insert(k, v)` – insert value v with key k into the priority queue
- `removeMin()` – return and remove from the priority queue the entry with the smallest key

Other operations on the priority queue include `size()`, which returns the number of entries in the queue and `isEmpty()` which returns true if the queue is empty and false otherwise.

Two simple implementations of a priority queue are an unsorted list, where new entries are added at the end of the list, and a sorted list, where entries in the list are sorted by their key values.

Fill in the following table to give the running times of the priority queue operations for these two implementations using *O*() notation. You should assume that the implementation is reasonably well done, for example, not performing expensive computations when a value can be stored in an instance variable and be used as needed.

Operation	Unsorted List	Sorted List
size, isEmpty	***O*(1)**	***O*(1)**
insert	***O*(1)**	***O*(*n*)**
removeMin	***O*(*n*)**	***O*(1)**

Question 2. (12 points) In this course we've seen many different data structures, including the following:

List (linked list or array)
2-dimensional (or higher) array
Stack
Queue
Hash table
Tree
Binary search tree
Undirected graph
Directed graph
Directed Acyclic Graph (DAG – a directed graph with no cycles)

For each of the following applications, indicate which of these data structures would be *most* suitable and give a brief justification for your choice. For data structures like trees and graphs, describe what information is stored in the vertices and edges, and, if the edges are weighted, describe what information is stored in the weights.

(Question continued on the next page)

Question 2. (cont.) Describe the most appropriate data structure from the list on the previous page, with details about vertices and edges where appropriate.

(a) Map of the Puget Sound highway system used to display traffic travel times on a web page. The map displays principle cities, intersections, and major landmarks, the roads that connect them, and the travel times between them along those roads. Travel times along the same road may be different in different directions. **A directed graph. Vertices = cities, intersections, landmarks, etc. A pair of directed edges between each pair of connected nodes, each edge giving the travel time in one of the two directions.**

(b) Chess board – an 8 x 8 board used for a game of chess. Each square on the board is either empty or contains a chess piece. **A 2-D array or similar structure. (Note: an array of boolean values is not sufficient, since it's not enough just to indicate that a square is empty or occupied. It's also necessary to know what color piece is on each square and what sort of piece it is.)**

(c) A computer model showing the dependencies between the steps needed to assemble a B787 airplane at Boeing's Everett plant. **A DAG. Vertices = job steps that need to be done. Edge from each step to successor steps that depend directly on it. (Note: the graph had better not have any cycles, otherwise it would not be possible to finish building the plane!)**

(d) A list of the legal words in a Scrabble©®™ game. We want to be able to quickly check whether words used by players do, in fact, exist in the list. **A hash table. Provides *O*(1) lookup for words. (Note: A binary search tree would be better than, say, an unsorted list, but that would still require more time to search for words.)**

(e) Description of the inheritance relationships between classes and interfaces in a Java program. **A DAG. Vertices = classes and interfaces. Directed edge from each class/interface to every other class/interface that it directly implements or extends. (Note: A tree would be sufficient to model class inheritance only, but a more general graph is needed to handle interface relationships.)**

(f) The history list recording sites visited by the user of a web browser. As new sites are visited they are added to the list. The list also supports the operation of going back to the web page that was previously visited before the current page and going forward to the next page visited. **Either a list or a pair of stacks (one stack for "past" history, the other for "future" history if we've already backed up and want to be able to go forward again).**

Question 3. (12 points) The nodes in an integer binary tree can be represented by the following data structure.

```
public class BTNode {    // Binary tree node
   public int item;      // data in this node
   public BTNode left;   // left subtree or null if none
   public BTNode right;  // right subtree or null if none
}
```

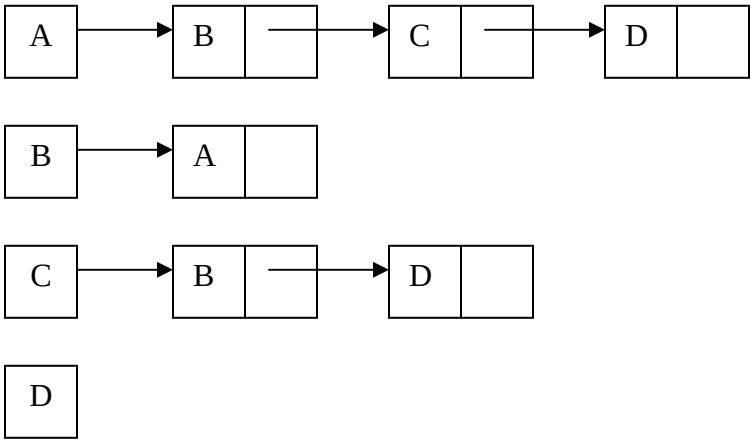
Complete the definition of method BFS below to perform a breadth-first traversal of a binary tree and print the node data values in the order they are reached during the search.

In your solution, you may create and use instances of other standard data structures (lists, queues, stacks, trees, hash tables, graphs, or whatever) and their operations if they are useful, without having to give details of their implementations.

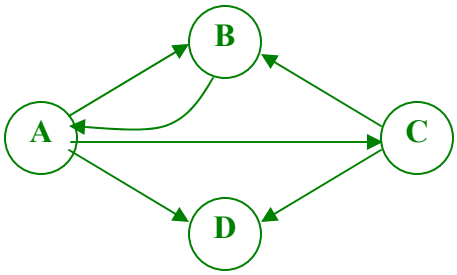
```
// Perform a breadth-first traversal of the binary tree with
// root r and print the node values as they are encountered
public void BFS(BTNode r) {

   Queue q = new Queue();

   // Assumption in this solution: null is never placed in
   // the queue

   if (r != null) {
      q.enqueue(r);
   }

   while (!q.isEmpty()) {
      BTNode n = (BTNode) q.dequeue();
      print(n.item);
      if (n.left != null) {
         q.enqueue(n.left);
      }
      if (n.right != null) {
         q.enqueue(n.right);
      }
   }

}
```

Note: In grading this problem, we were looking for fairly precise code or pseudo-code, but didn't count off for minor things, like a missing cast when a node was extracted from the queue.

Question 4. (10 points) Here is an adjacency list representation of a *directed* graph where there are no weights assigned to the edges).



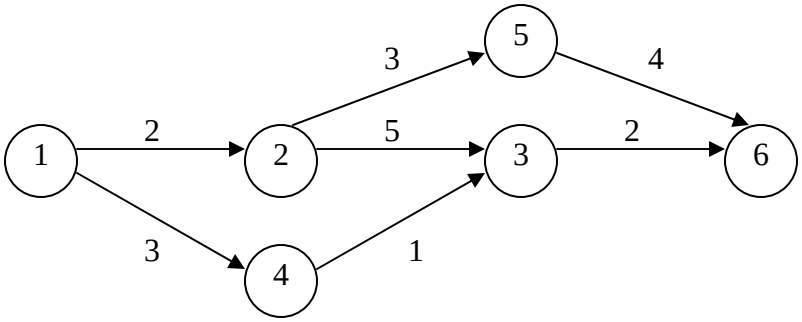
(a) Draw a picture of the directed graph that has the above adjacency list representation.



(b) Another way to represent a graph is an *adjacency matrix*. Draw the adjacency matrix for this graph.

	A	B	C	D
A	**0**	**1**	**1**	**1**
B	**1**	**0**	**0**	**0**
C	**0**	**1**	**0**	**1**
D	**0**	**0**	**0**	**0**

Question 5. (12 points) Consider the following directed graph.



We want to use Dijkstra's algorithm to determine the shortest path from vertex 1 to each of the other vertices. Update the entries in the following table to indicate the current shortest known distance and predecessor vertex on the path from vertex 1 to each vertex as the algorithm progresses. (Cross out old entries when you add new ones.) The initial values for the distances are given for you. Below the table, list the vertices in the order that they are added to the "cloud" of known vertices as the algorithm is executed.

Vertex	D (distance from vertex 1)	Predecessor vertex
1	0	---
2	~~∞~~ **2**	~~?~~ **1**
3	~~∞~~ ~~7~~ **4**	~~?~~ ~~2~~ **4**
4	~~∞~~ **3**	~~?~~ **1**
5	~~∞~~ **5**	~~?~~ **2**
6	~~∞~~ **6**	~~?~~ **3**

List of vertices in the order they are processed by the algorithm:

1 , **2** , **4** , **3** , **5** , **6**

Question 6. (4 points) If $|V|$ is the number of vertices in the directed graph, and $|E|$ is the number of edges, what is the running time of Dijkstra's algorithm in $O()$ notation? Give a brief justification for your answer. You should also briefly describe any assumptions you are making about the implementation that would affect the answer.

Dijkstra's algorithm is a bit more complicated to analyze than some because we need to think about some of the implementation details. If we use an adjacency list to represent the graph we can step through the vertices connected to any particular vertex in time proportional to the number of connected vertices. An efficient implementation of the priority queue would allow us to extract the min value in $O(\log n)$ time, and if the entries are location-aware, they can also be updated in $O(\log n)$. If we look at the total number of times these operations are performed, we get a total running time of $O((|V| + |E|) \log n)$.

Question 7. (4 points) A *topological sort* of a directed acyclic graph (a graph without cycles) yields a list of vertices such that if there is a path from vertex i to vertex j, then i precedes j in the topological sort. In $O()$ notation, what is the running time of a topological sort on a graph with $|V|$ vertices and $|E|$ edges? Give a brief but precise explanation justifying your answer.

During a topological sort, each vertex and its adjacent edges are removed from the list of unprocessed vertices and edges once, and not considered again. That gives a total running time of $O(|V| + |E|)$.

Question 8. (8 points) (Sorting)

(a) Two of the most common divide-and-conquer sorting algorithms are quicksort and mergesort. In practice quicksort is often used for sorting data in main storage rather than mergesort. Give a reason why quicksort is likely to be the preferred sorting algorithm for this application.

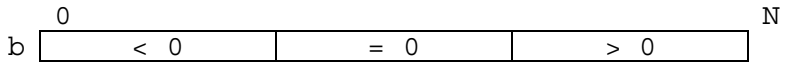
As long as the pivots for quicksort are chosen carefully (next question), both algorithms run in $O(n \log n)$ time, so there is no reason to prefer one over the other based on time. However, mergesort requires that half of the available space be used to merge the two lists created when the problem is divided, so it can only sort half as much data in a given amount of memory.

(b) Quicksort's worst-case running time is $O(n^2)$, but it has an expected running time of $O(n \log n)$ if the partition function works well. What needs to be true about the partition function in order for the running time to be $O(n \log n)$? In practice, how can we ensure that this happens?

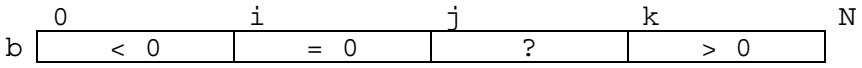
The key is that the partition step must divide the section of the list to be sorted into two roughly equal-sized parts by picking a pivot value that is roughly the median of the values in that section of the list. If this happens then the overall depth of recursion is bounded by $O(\log n)$ and the total running time is $O(n \log n)$.

There are several ways to pick a good pivot value. Two common ones are to pick a median of 5 values scattered throughout the list section (front, middle, end, and halfway between the middle and each end), or to randomly pick some value in the interval.

Question 9. (12 points) Suppose we have an array `b` containing *n* integers. Initially the integers are in some random order. We would like to rearrange the array so all the negative integers precede all the zeros, and the positive integers appear at the end. In pictures, we would like to rearrange the contents of `b` so that at the end we get the following picture:



Simply trying to hack this up is a mess (try it for 2 minutes if you are not convinced). However if we draw a picture showing what the computation looks like when it is partially finished and use that to guide the coding, it is much more tractable. Specifically, once we have partially, but not completely moved the data, the array should look like this:



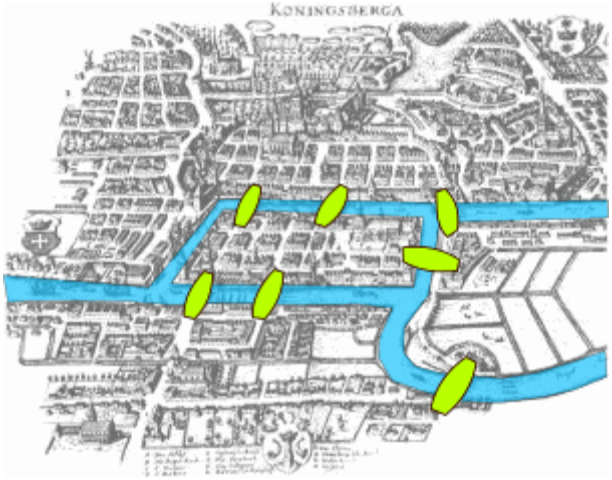
In other words, the entries in `b[0..i-1]` are negative, `b[i..j-1]` zero, `b[j..k-1]` have not been processed, and `b[k..n-1]` are positive. Some of these intervals may be empty if no numbers with the correct values have been found.

Complete the following code to partition the array. Hint: think about the loop body first, then worry about the initialization conditions. Requirement: The algorithm **must** partition the array in **linear** ($O(n)$) time. You may write `swap(x,y)` to interchange the values of variables `x` and `y`.

```
//  initialize

i = 0 ; j = 0 ; k = n ;

// repeat until no unprocessed elements remain in b[j..k-1]

while ( j != k ) {
   // decrease size of b[j..k-1] by 1 by moving one or more
   // items and adjusting i, j, and/or k

   if (b[j] < 0) {
      swap(b[i], b[j]);
      i++;
      j++;
   } else if (b[j] == 0) {
      j++;
   } else {  // b[j] > 0
      swap(b[j], b[k-1]);
      k--;
   }

}
```

Question 10. (12 points) A famous problem is figuring out whether an undirected graph contains an Eulerian path. Such a path contains all of the edges in the graph exactly once, but may pass through any of the vertices in the graph as many times as desired.

The problem originated from the Bridges of Konigsberg puzzle: is it possible to walk through the town of Konigsberg, crossing all of the bridges in the town once and only once? (Story and picture included for cultural enrichment, but feel free to ignore it and skip to the next paragraph if it is confusing.)



This problem can be formulated as the Eulerian path problem by treating each of the bridges as an undirected edge in a graph, and the parts of the city as the vertices.

Here is the crucial fact: A graph contains an Eulerian path if

1. It consists of just one connected component (meaning that all vertices can be reached from any other vertex), and
2. It contains no more than two vertices of odd degree.

On the next page is a set of data structure definitions for a `Graph` data type. You should complete the `hasEulerianPath` method so it returns `true` if the graph parameter `g` has an Eulerian path and `false` if it does not.

Hint: You can use breadth-first search (BFS) or depth-first search (DFS) to determine if the graph has a single connected component – i.e., starting at some arbitrary vertex in the graph all other vertices can be reached.

Question 10. (cont.) Here are the definitions for the graph data structures

```
public interface Graph {
  Collection vertices();    // returns a collection of all the
                            //   Vertex objects in the graph
  Collection edges();       // returns a collection of all the
                            //   Edge objects in the graph
  Collection incidentEdges(Vertex v); // returns a collection of
                                      //   Edges incident to v
  boolean isAdjacent(Vertex v, Vertex w); // return true if v and
}                                         //   w are adjacent

public class Vertex {
  public boolean visited;  // initially false for all vertices
}

public class Edge {
  Vertex v1, v2;                  // undirected edge
  Vertex opposite(Vertex v);      // given a vertex return the one
}                                 // at the other end of this edge
```

Complete the method `hasEulerianPath` below. You may assume that the `visited` field in each `Vertex` is initially `false` in order to check whether the graph contains a single connected component. You may define additional (auxiliary) methods if you wish, or you can write all the code in this method. But use comments to make it easy to figure out what the major parts of the code are doing.

```
public boolean hasEularianPath(Graph g) {

      int oddDeg = 0;
      for(Iterator i = vertices.iterator(); i.hasNext(); ) // opt.
        ((Vertex)i.next()).visited = false;                // opt.
      Queue q = new Queue();
      q.add(vertices.iterator().next());
      while(!q.isEmpty()) {
        Vertex v = (Vertex)q.dequeue();
        if (v.visited)
          continue; // cross-edge
        v.visited = true;
        if (v.incidentEdges().size() % 2 == 1)
          oddDeg++;
        for(Iterator i=v.incidentEdges().iterator(); i.hasNext(); ){
          Vertex w = ((Edge)i.next()).opposite(v);
          if (!w.visited)   // not a back-edge
            q.enqueue(w);
        }
      }
      for(Iterator i = vertices.iterator(); i.hasNext(); )
        if (!((Vertex)i.next()).visited)
          return false;
      return oddDeg <= 2;

}
```