1. What attributes of a problem might you consider when deciding how to represent a graph?

   Is number of edges and vertices (overall size) known? Is the number of edges per vertex variable? How important is access efficiency?

2. What are some advantages of using a linked structure to represent a graph? Disadvantages?

   Graphs are structures that feature pairwise links between objects. However, any object in the graph can host multiple links to other objects. In other data structures, such as a stack implemented with a linked list, there are generally a fixed number of pointers (e.g. two pointers for "right" and "left"). Because of this difference, my initial thought is that it may be more difficult to implement a graph using a linked structure, since the number of links is variable. As for other advantages/disadvantages, I suppose they would be the same as for other data structures that are implemented as a linked structure; that is, dynamic allocation as an advantage, but possibly slow sequential access as a disadvantage.

3. Why might an array be better (or worse) than a linked structure to represent a graph?

   As mentioned above, the same advantages/disadvantages exist for use of an array to represent a graph as for use of an array to represent some other structure, like a stack. A structure implemented using an array is a fixed size that must be specified in advance. However, access to any item in an array does not require the same chain of references that is required in linked structures – an array is random access and the cost to access any element is constant.

   For a graph specifically, it seems that due to the variable number of edges at each vertex, implementation could be difficult. There would have to be some way to keep track of everything. In other structures that are more "defined", this is easy. Take a binary tree for instance – the child nodes are located at indices 2n and 2n+1.

4. Can you name some applications for which a graph representation might be useful?

   A graph could be a useful representation for a large number of things. Some examples might be computer networks, social structures, transit maps, and disease tracking/epidemiological surveillance.

5. What aspects of graphs make them a good (or poor) match for recursive solution?

   My current thought is that graphs would be a poor match for a recursive solution. The structure requirements of a graph are not very rigid relative to other data structures whose general form is repeated (e.g. binary tree). Any node on a graph could be linked to one, or two, or three, etc other nodes. A graph could be cyclic or it could be acyclic. Because of this, it would be difficult to implement a recursive solution, define base cases, etc.

6. How is a priority queue different from a traditional queue? What applications would make the best use of such a data structure?

   I believe in one of the previous homework assignments it was said that a traditional/standard queue is basically the same as a priority queue, except that time, rather than some other parameter, is the basis of priority. In a priority queue, the elements of the queue are sorted by some other parameter. A min-priority queue is used in Dijkstra's algorithm for finding the shortest path between nodes in a graph. A simple example of a priority queue is triage. Patients are prioritized based on severity of illness/injury/etc.

7. Is there an implicit prioritization in a traditional queue? If so, what is it?

   Yes – time. The "priority" of a traditional queue is the time at which the item was placed in the queue. The first item in (oldest time) is the highest priority. The last item in (most recent time) is the lowest priority.

8. What data structures lend themselves to effective implementation of a priority queue?

   Heaps are generally used to implement priority queues. A heap is a binary tree that maintains order and shape. In a min-heap, the lowest value/priority will be the root node of the heap. A max-heap will be the opposite. In this way, the highest or lowest priority item will always be at the root node.

9. How does the order of a tree (a general tree with no special properties) affect the size of the tree and layout of the nodes in the tree? How do you decide which child path to follow?

   The order is the maximum number of children a node may have. Two complete trees of the same level but different orders will have a different number of nodes (i.e. the size of the tree will be different). Child node selection can be arbitrary, or they can follow a prescribed form. For example, in one scheme, any node inserted is compared to the parent node – smaller values become the left child while larger values become the right child.

10. How would a Huffman tree look different if we used different tie-breakers, and how would that impact the potential compression?

    The tie breaker used will not impact the potential compression, because regardless of tie-breaker, if the frequency of occurrence is the same, the total number of bits created (as specified by the Huffman mapping) will be the same. For example, if there are three leaves of the same frequency, and two are represented by 110 and 111, while the third is represented by 11, it does not matter which character is represented by which binary because the total number of bits used is the same (1x3 + 1x3 + 1x2). The only importance of specifying the tie breaker is so that whatever is encoded is decoded properly (using the same tie-breaker).