1. Queues and stacks can be thought of as specializations of lists that restrict which elements can be accessed.
1a. What are the restrictions for a queue?
The program can only access the first thing in a queue.
1b. What are the restrictions for a stack?
The program can only access the elements at the top of the stack.

2. We have looked at lists backed by arrays and links in this class. Under what circumstances might we prefer to use a list backed by links rather than an array? (Your argument should include asymptotic complexity).
It would be better to use linked lists over arrays when the program is required to do more adding and removing. This is because the asymptotic complexity of a linked lists when adding and removing is O(1), since the process is the same every time. While the array has a O(n) rate, because it has to recreate the array with a different size.

3. Give the asymptotic complexity for the following operations on an array backed list. Also provide a brief explanation for why the asymptotic complexity is correct.
3a. Appending a new value to the end of the list.
O(n), because the array has to recreate a new array of a different size every time it appends a value.
3b. Removing a value from the middel of the list.
O(n), because the array has to recreate a new array of a different size every time it appends a value.
3c. Fetching a value by list index.
O(1), because fetch does not require changing the array but just getting the value at the given index.

4. Give the asymptotic complexity for the following operations on a doubly linked list. Also provide a brief explanation for why the asymptotic complexity is correct.

4a. Appending a new value to the end of the list.
O(1), because a linked list only has to direct the pointers and not expand or decrease in size.
4b. Removing the value last fetched from the list.
O(n). The program still has to go through a list to get to the node that holds the value.
4c. Fetching a value by list index.
O(n), because the program still has search the list to get to the node at that index.

5. One of the operations we might like a data structure to support is an operation to check if the data structure already contains a particular value.
5a. Given an unsorted populated array list and a value, what is the time complexity to determine if the value is in the list? Please explain your answer.
O(n), because the program has to go through every value in the array list and also check for the right value.
5b. Is the time complexity different for a linked list? Please explain your answer.
O(n), because the linked list also must go through every value in the list, and check for the right value.
5c. Given a populated binary search tree, what is the time complexity to determine if the value is in the tree? Please give upper and lower bound with an explanation of your answer.
The upper bound scenario would be a O(n). If the tree were to be right linear where all the nodes are on one side the space/time complexity would be O(n).This is because it would have to go through every value in the tree until it found the correct value. The lower bound scenario would be O(log n), since the tree is complete the program would know the depth of the complete tree (log(n)). So the programs lower bound will be O(log n) when searching the tree for the correct value.

5d. If the binary search tree is guaranteed to be complete, does the upper bound change? Please explain your answer.
The upper bound would be O(log n), because the depth of the tree is still log(n). Since the depth of each branch is the same the tree would have it's best case scenario. The options when searching would be cut in half every time, so it would have a tight bound of (log n).

6. A dictionary uses arbitrary keys retrieve values from the data structure. We might implement a dictionary using a list, but would have O(n) time complexity for retrieval. Since we expect retrieval to occur more frequently than insertion, a list seems like a poor choice. Could we get better performance implementing a dictionary using a binary search tree? Explain your answer.
Given a complete and organized binary search tree the time complexity would be O(log n), since the options when searching for the key will always be cut in half. While the worse case scenario for a tree with a long linear branch will have a O(n) time complexity. Using a binary search tree would be better for performance, since in better scenarios it will have a faster time complexity.