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?

Queues (FIFO) are limited to enqueue() and dequeue() functionality. Enqueue() inserts a value at the top of the queue. Dequeue() removes a value at the bottom of the queue. Queues cannot operate with indexes.

1b. What are the restrictions for a stack?

Stacks (FILO) are limited to push() and pop() functionality. Push inserts a value at the top of the stack. Pop() removes a value at the top of the stack. Stacks also cannot operate with indexes.

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).

If we expect to be working without indexes, a list backed by links would probably make more sense. When adding/removing/fetching at the end of a list, lists backed by links are able to achieve constant time complexity (O(1)) because the last link is stored. Lists backed by arrays will always have linear time complexity (O(n)) when adding/removing because of the need to recreate them.

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) When adding to a list, the list needs to be recreated with an increased size, which involves iterating through the whole list to copy all the values.

3b. Removing a value from the middle of the list.

O(n) Same reasoning as appending, list needs to be recreated with a decreased size, which involves iterating through whole list to copy all the values.

3c. Fetching a value by list index.

O(1) the index for every value is stored, so no iteration, just return value for that index regardless of the size of list.

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) The Last link in list is stored, so just have to assign new last link to previous last one, regardless of the number of links.

4b. Removing the value last fetched from the list.

O(1) Last value is stored, just have to fetch it.

4c. Fetching a value by list index.

O(n) Since linked lists don't use indexing, will have to search through all links to find the right one.

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) Simply have to check every value in the list and see if it matches the particular value.

5b. Is the time complexity different for a linked list? Please explain your answer.

No, will still have to check every link.

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.

Upper bound: O(n) Lower bound: O(log n) assuming the tree is fairly balanced, there is logarithmic time complexity, because the amount of data we have search per value added decreases with each row of the tree. If the tree has a long branch, then it would be like searching through a list, which would result in linear time complexity.

5d. If the binary search tree is guaranteed to be complete, does the upper bound change? Please explain your answer.

Yes, then there is a tight bound of logarithmic time complexity, same logic as 5.c.

6. A dictionary uses arbitrary keys to 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.

Yes, if each node of the tree contained a key that conformed to the binary search tree requirements, and a value, we could achieve logarithmic time complexity by searching for our desired key.