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?

First In, First Out (FIFO): Items must always be removed from the front of the list and inserted at the back of the list.

1b. What are the restrictions for a stack?

First In, Last Out (FILO): Items must always be removed (popped) and inserted (pushed) at the top of the stack/back of the list. Only the value at the top of the stack can be accessed.

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).

Arrays have fixed length, whereas linked lists don't. Therefore, if the number of items to be added to the list is unknown, a linked list is superior to an array-backed list. In addition, removing and appending items at the beginning or end of the list, or anywhere besides at a specific index, becomes faster with a linked list, because the current, head, and tail pointers eliminate the need to shift the entire list, as one would need to do with an array. This makes the asymptotic time complexity of a method such as remove(), for instance, constant (O(1)) for a linked list, because the method does not need to loop through the entire list to remove a value at the end. Alternatively, an array-backed list has linear (O(n)) asymptotic time complexity for a method such as remove() in the worst-case scenario in which the value to remove is at the end of the list.

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.

Linear (n) because all the values in the array must be shifted into a new array to accommodate the appended value.

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

Linear (n) because all the values in the array must be shifted into a new array to accommodate the removal of the value.

3c. Fetching a value by list index.

Constant (1) because the value at any index can be accessed simply by calling array[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.

Constant (1) so long as there is a pointer to the tail, which makes it easy to access the end of the list and simply rearrange the pointers to append a new value.

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

Constant (1) because there is a pointer to the current value, which will be the fetched value after the fetch method is called. Therefore, the current pointer does not need to be advanced to another cell in the list.

4c. Fetching a value by list index.

Linear (n) because the current pointer must be advanced from the beginning of the list until it points to the link at the desired 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.

Linear (n) because the method must loop through the entire list every time and check the value at every index to determine whether the value is in the array.

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

No, because each cell in the list must still be checked to determine if it contains the value. To do this, a loop must be implemented to advance the current pointer through the entire list.

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 is linear (O(n)) because in the case that each node of the binary search tree only has one child on the same side of the tree as the previous node, the tree is essentially a list, and the time it takes to check if each node of the tree contains a given value is still linear.

The lower bound is logarithmic ($\Omega(\log(n))$ because in the case that the binary search tree is complete, only the nodes on one side (following one linear path) of the tree will be checked. This means that the number of checks the method will need to do will be equal the number of levels in the tree, as one node is checked per level. This number is logarithmically proportional to the number of nodes/values in the tree.

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

Yes, if the binary search tree is guaranteed to be complete, then the logarithmic lower bound ($\Omega(\log(n))$ becomes the upper bound because a complete BST is the best-case instance of a BST in terms of time complexity. Even if the method has to go through every level of the tree to find the value (worst case), the time complexity will still be logarithmic.

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.

Yes, although it depends on the type of binary search tree we implement. If the binary search tree is complete, then the dictionary will retrieve keys with logarithmic time complexity ((Ω(log(n)) because it has to check only one node at every level of the tree (due to all the nodes to the left of the root being less than the root and all the nodes to the right being greater). Logarithmic time complexity would make for better performance than linear time complexity. However, if the tree is simply a series of nodes which all have only one child, then the tree is a list and has the same linear time complexity as the list-backed dictionary. This case would not improve the performance of the search method.