Isabel Gutierrez

1. Queues and stacks can be though of as specialization of lists that restrict which elements can be accessed.

   a). What are the restrictions for a queue?

   Queues are First in, First Out lists, which means that elements can only be taken out from the top of the list, and inserted at the bottom.

   b). What are the restrictions for a stack?

   Stacks are First in, Last Out lists. This means that elements can only inserted and taken out from one end.

2. Under what circumstances might we prefer to use a list backed by links rather than an array?

Opperation	Singly-Linked List	Doubly-Link List	ArrayList
Access an element	O(n)	O(n)	O(1)
Add/Remove at curr	O(1)	O(1)	O(n)
Add/Remove first element	O(1)	O(1)	O(n)
Append	O(1)	O(1)	O(1)
Remove last element	O(n)	O(1)	O(n)

LinkedLists are more efficient when adding and removing elements than an ArrayList. However, ArrayLists would be more efficient if the user would required to constantly look an nth element on the list. Since ArrayLists have direct references to every element in the list, accessing an element is done in a constant time. For LinkedLists, accessing a constant element takes a linear amount of time, since it has to iterate through the whole list until it finds it.

3. Give the asymptotic complexity for the following in an array backed list.

   a). Appending a new value to the end of the list.

   O(1) —It takes constant time because since the addition is at the end, elements do not need to be shifted.

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

   O(n) — linear time because array has to make a smaller copy of the original array and shift values to their new index.

   c). Fetching a value by list index.

   O(1) — ArrayLists are backed by an array that holds a direct reference to the index for each element.

4. Give the asymptotic complexity for the following operations on a double linked list.

   a). Appending a new value to the end of the list.

   O(1) — Adding a new value at the end of a double linked list would only mean changing the pointer at the tail, and adding a “previous” pointer to the new node.

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

O(1) — Removing values from a double linked list takes constant time because it is only a matter of changing the previous element pointer, and the next element pointer.

c). Fetching a value by list index.

O(n) — Since double linked lists don't hold reference to index numbers, the user would have to iterate through the whole list until it gets to the index value.

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.

a). Given an unsorted populated array list and a value, what is the time complexity to determine if the value is in the list ?

O(n) — Since there is no order, the user would need to iterate through the whole list (or until the nth value is reached) and compare the values of each element to the value desired.

b). Is the time complexity different for a linked list?

No, it would be the same time complexity. If the list has no order, there is no reference to where the value may be stored, therefore, the user would have to iterate through the list and compare values until it finds the one it is looking for.

c). 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.

The upper bound would be O(n), because in the worst case scenario, the value that is being searched for it a leaf in an unbalanced tree. This would mean traversing all the levels until it finds the nth element. The lower bound would be (log n), because a binary search tree with n nodes has at least (log n) levels.

d). If the binary search tree is guaranteed to be complete, does the upper bound change?

The upper bound would change to O(log n). The worst case scenario for a complete tree would be iterating all the levels of a tree with n nodes, which are 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?

A binary search tree would get a better performance because its time complexity is O(log n). In a binary search tree, elements are organized in an order such that half of the possible values are eliminated after each search iteration, which makes it much more efficient than a list.