Kon Aoki

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?

A queue is a first in first out data structure meaning that the user can only fetch the oldest data queued into the data structure in order. The fetching function for a queue is commonly known as deque, and it is a destructive function that pulls out the data fetched out of the list. Another restriction is that the user can only queue data at the back of the list (just like an append function).

1b. What are the restrictions for a stack?

A stack is a Last in first out data structure meaning that the user can only fetch (again a destructive function called pop) the newest data pushed onto the list. Another restriction the structure has is that it can only push new data from the top of the list in order. Unlike the queue, the output of multiple pops in order will give the input in reverse order.

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).

Both lists backed by arrays and links can have an asymptotic time complexity of O(1) for the append function. For links, all it takes is to add a pointer to the old tail to the new tail. For arrays, the asymptotic time complexity would be amortized constant time with the implementation of doubling the size of the array every time it has to resize. This is because it takes 2N time (corresponding to n new inputs) to resize but also N time until the array reached its maximum capacity and needed to resize (assuming the speed at which the data is inputted is constant), averaging out to the O(1) time complexity.

However, there are differences in removing data and fetching data. Data can be removed from the head at constant time for links but with O(n) time for arrays. Data can be fetched from an index at constant time for arrays but O(n) time for links.

For a data structure that has limitations to the fetching and adding function, such it can only fetch and add from the head or the tail, links are better off with constant time for all necessary functions. If the data requires constant fetching from a known index, arrays are better off. However, both data structures will take O(n) time for searching if a value exists meaning that other structure like a tree will be more fit for circumstances that might require more of searching.

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.

It will take constant time due to the reason explained in the first paragraph of question 2. However, if the array is resized every time a new item is appended, it will take O(n) time because all the previous items will have to be copied over to a new array with a larger capacity.

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

This will take O(n) time assuming the middle means the center of the list. After removing the value, the entire right side of the list will have to be shifted to the left by one slot which will take n/2 loops.

3c. Fetching a value by list index.

This will take constant time. The index can be used to directly point to the memory location where the item requested is stored in the array.

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.

This operation will take constant time because when a new item is appended all that needs to be done is set the next pointer of the old tail to the new tail and set the previous pointer of the new tail to the old tail. There is no recursion/loop involved.

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

Although fetching takes O(n) time, the remove by itself will take constant time for the same reason as 4a. Only the pointers at that location need to be updated unlike the array backed list where the entire list can be affected by removing a single item.

4c. Fetching a value by list index.

This will take O(n) time because to find the index, the function will need to loop through the entire list in the worst case. In the best case, it will be constant time however when the index is 0.

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.

It will take constant time in the best case where the value exists in index 0 and O(n) in the worst case where the value exists near the end. This is because the function will need to loop through the list until it finds (or doesn’t) the item requested.

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

It will be the same for a linked list because the function will still need to linearly search through the unsorted 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 lower bound, like the lists, is constant time when the value is at the root. The upper bound however is lower than both the lists at O(logn) time (assuming the tree is well balanced). This is because the height of the tree is logn. However, if the tree is not balanced, it can take O(n) time with the same structure as a list.

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

Yes. With this restriction, the upper bound will be O(logn) because complete trees are by definition well balanced, keeping the maximum height at a minimum.

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. With a binary tree, the search for a key will take O(logn) time (again assuming the tree is well balanced). Therefore, the retrieval time complexity will also be O(logn). However, if the list is sorted, retrieval will take logn time like the tree using a binary search although that essentially has the same conceptual structure as a tree.