Payton Landes
Day 8 Theory

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 has access restrictions. A queue can be added to only at the back position and can only be removed from the front position. Making the queue “first in, first out”.

1b. What are the restrictions for a stack?

A stack also has access restrictions. A stack can only be added to(push) and removed from(pop) at one position called the top “first in, last out”.

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 of these are capable of the same methods, the information that help decide which is preferable is the frequency of specific method usage.

For geting an element by index:

> Linked O(n) vs. Array O(1)

For adding an element at the end:

> Linked O(1) vs. Array O(1)

For adding an element at the beginning:

> Linked O(1) vs Array(n)

For adding an element at index:

> Linked O(n) vs. Array O(n)

Remove:

> Linked O(n) vs. Array(n)

So, by generalizing these performance attributes and considering that arrays are statically sized one can decide what is prefered. If a resizable array is required, array backed is not allowed. If one is accessing elements much more than adding and removing elements the array backing would perform better and vice versa. If the size of the data could potentially vary greatly, an array would waste storage space on empty array elements but the linked back would not. I'm not going to rehash the analysis of the book completely but it provides an interesting formula for estimating the space usage of both backing systems.

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(1)

You can access the empty cell at end of the list directly by index and change it's value.

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

O(n)

You must index in to removal index, remove that value, then shift all values higher than remove value down 1.

3c. Fetching a value by list index.

O(1)

You just index in to the value

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)

Assuming you're keeping track of the end, you just need to make tail end's next field equal the new value.

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

O(n)

From the interface we've been looking at you still have to index in and find the value that was returned. No shifting has to be done after removal.

4c. Fetching a value by list index.

O(n)

We're still going to take index * 1 steps to find the 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.

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)

The array list must be searched through by index until the value is found.

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

No, you also have to search through linearly until you find the one.

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.

O(n)

With n = height of BST being the worst case. The value you want is found to be a leaf

lower(log n)

We need to look at half as many things every time we take a step towards our value

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

No, this would not change the upper bound, there is still the worst case that the search value is in the lowest level in the tree

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, lower(log n) for BST is faster than lower(n) for list.
The logarithmic lower limit for a BST can give better performance for this task.