Sarah Dunbar
Day 8 Theory Assignment
12/6/17

1a) Queues follow the first in first out rule, so the operations performed on it are restricted. Values are enqueued to the rear of the list, and dequeued from the front of the list. Values can’t be accessed from the queue with any other method, and therefore not from the middle of the list, so that restricts the manipulation of the queue.

1b) A stack is a restricted data structure because a list implemented by a stack can only be manipulated from one position in the stack, the top. A stack follows the first in last out rule which means that we can only push and pop the last value in the stack.

2) To start, in terms of memory, it would be advantageous to use a list backed by an array in the case that we know exactly how many elements that we are going to want in our list. It is easier to allocate a specific amount of memory for our list with an array then it is with a linked list. On the other hand, if we want it to be easier to add values to our list then a linked list would be much more helpful. This is especially true because the append function for a linked list takes constant time as the size increases, while an array takes linear time. If we allocate a pointer to the tail, then we can easily jump to that tail, create a new node, and change the pointers no matter what the size of the list is. However, if we were using an array, as the size of the list increases, the amount of times we must iterate through the list also increases which creates linear time complexity.

3a)Because arrays do not have dynamic size, when adding a value to the end of the list, we will have to create an array that is one longer than the original, and then copy over all of our old values, plus the new one to our new temporary array. Because we will have to iterate through, and copy over, all of the elements in our array in order to add a new one to the end of the list, as the list size grows, the longer it will take to perform this method. Therefore, adding an element to the end of the list will be on the order of linear time complexity. O(n).

3b)When removing something from the middle of an array, we must rearrange the items in the list after that index so that they are correct after the removal. Although this will only be half of the elements in the list, since we are removing from the middle, if the array is larger, there will be more items to rearrange. Because of this, the time complexity from removing an item from the middle of the list will be on the order of linear. O(n).

3c) Assuming that it takes the same amount of time to get to every memory location, and that we know exactly where in the list the element that we are searching for is, fetching an element at a list index will be on the order of constant time. O(1).

4a) The time complexity for appending a new value to the end of a double link list will depend on personal implementation of it.   In our implementations we are keeping track of the tail cell, so, we know exactly where we have to jump to, and only have to create a new cell and change the pointers once we get there. This will give us time complexity that is on the order of constant time. O(1).

4b)In a linked list, the reason that removing an item at an index would take so long is that we would have to search the entire list for it, because we don't know where it is. However, as long as we keep track of the current element in the list, once we fetch an item, we will know exactly where in the list it will be because our current pointer will now point to it. So, as long as we have a current pointer, it should take constant time to remove the last fetched item as we know exactly where it is. O(1).

4c) In doubly linked list, we aren't necessarily keeping track of "indexes" in the list. So, in order to fetch a value by the list index, we must iterate through our list, starting at the head, until we get to the specified position. For that reason, it will have linear time complexity. O(n).

5a)Given an unsorted array, in order to find out if an element is contained in a list, we would have to create an algorithm that goes through each index of the array one by one and compares the value to the one that were searching for. Because we must iterate through an unsorted array in order to search for something, the time complexity will be on the order of n, so linear time complexity. O(n).

5b)The time complexity for a linked list would not be different than an array. The best case scenario would be that what we were searching for would be the first element in the list, taking one unit of time. However, we don't know where what were searching for will be in the list, or if it will be in the list at all, since it is unsorted. So, since we will have to iterate through the list and compare all values to the one were searching for. We will again be on the order of linear time complexity. O(n).

5c) Searching for something within a binary tree is dependent on whether the tree is balanced or not. If a tree is balanced, then the height of the tree is minimized based on the values that are inserted. An unbalanced tree means that the height is maximized based on the values inserted, making it similar to a linked list. A balanced tree will have height of logn and an unbalanced tree will have height of n. The best possible scenario is that the key were searching for is the root, and therefore will take one unit of time. However, in the case that we have maximum height n, and the key that were searching for is a leaf, we must traverse through every node in the tree in order to find the one that were looking for.
Therefore upper bound: O(n), lower bound: Ω(1).

5d) Having a complete tree would not change the upper bound because in the worst case scenario we would still have to iterate through every node in the tree which is proportional to the height of the tree, h.

6) Due to the invariants of a binary search tree, the elements stored in the tree are easily comparable and sorted in a way that makes searching extremely easy. Because of the comparisons that are made when we are following the pointers of the tree, we can essentially skip looking through an entire side of the tree every time a comparison is made. So, we essentially only have to look through half of the trees elements in order to find the one that we want. This gives us more logarithmic time complexity as opposed to slower linear time complexity when using an array.