Note: These practice questions are of the style that you will see on the exam. This set of practice problems is approximately the same length as you can expect to see on the actual midterm.

Q1: What is the complexity of a function that takes exactly log (log(n)) + n * log(n^2) + 5 * n^2 steps?

$O(n^2)$

Q2: Consider the following function.

```
void removeFrontListQueue (struct ListQueue *q) {
   struct Link *tlnk;
   assert(!isEmptyListQueue(q));
   tlnk = q->head->next;
   q->head->next = q->head->next->next;
   if(q->head->next == 0)
      q->tail = q->head;
   free(tlnk); /* release the memory!*/
}
```

What is the overall complexity of removeFrontListQueue?

O(1)

Q3: Consider an empty but initialized dynamic array stack with initial capacity 10. Suppose writing a new element to the array costs 1 unit, and copying a single element during reallocation also costs 1 unit.

How many total units will it cost to call 25 consecutive push operations on this stack? Assume that the capacity of the new array is twice the capacity of the current.

55

Q4: Consider the following dynamic array struct.

```
struct DynArr {
   TYPE * data;
   int size;
   int capacity;
};
```

Write the function _setCapacityDynArr() which resizes a dynamic array to the provided capacity. You may *NOT* use realloc().

<u>Note:</u> This is one of many acceptable solutions

```
void _setCapacityDynArr(struct DynArr *v, int newCap)
{
        assert(v != 0);
        /* Allocate new temporary dynamic array with given cap */
        TYPE *temp = malloc(sizeof(TYPE) * newCap);
        assert(temp != 0);

        /* Copy elements from old array to new array */
        for (int i = 0; i < v->size; i++) {
                temp[i] = v->data[i];
        }

        /* Free the old array */
        free(v->data);

        /* Set pointer v to new array */
        v->data = temp;
        temp = 0;

        /* Update the capacity*/
        v->capacity=newCap;
}
```

Q5: For the _setCapacityDynArr() function you wrote above, what is the algorithmic execution time?

O(n)

For the next two questions, consider the following commands.

1. struct dynArr *stack = createDynArray(1)
2. pushDynArray(&stack, 2);
3. popDynArray(&stack);
4. pushDynArray(&stack, 5);
5. pushDynArray(&stack, 2);
6. pushDynArray(&stack, 4);
7. pushDynArray(&stack, 13);
8. popDynArray(&stack);

Q6: After each of the above 8 commands, what is the size and capacity of the dynamic array stack?

1. [] size = 0 ,capacity = 1

2. [2] size = 1 ,capacity = 1

3. [] size = 0 ,capacity = 1

4. [5] size = 1 ,capacity = 1

5. [5] [2] size = 2 ,capacity = 2

6. [5] [2] [4] [] size = 3 ,capacity = 4

7. [5] [2] [4] [13] size = 4 ,capacity = 4

8. [5] [2] [4] [] size = 3 ,capacity = 4

Q7: After the last of the 8 commands above, what are the contents of the dynamic array stack? Assume a resize doubles the capacity.

5	2	4	

Q8: Why is it difficult to implement a deque using the same dynamic array implementation that was used in the dynamic array stack?

While a stack only has values added to and removed from one end of the array, a deque may have values added to or removed from the front or back. The problem is that removing or adding a value located at index 0 (unless it is the only value in the deque) is very difficult. For adding, you must shift all other values up one index in the array, and for removing, you must shift all other values back one index in the array. Otherwise, it would leave a "hole" in your array. Either operation would have O(n) algorithmic execution time, which is not ideal.

For the next two questions, imagine you've been given the task of implementing a data structure to store an undo buffer for an application. This undo buffer should support a finite length 'undo' operation (e.g. you can only undo the last 20 steps) as well as support direct indexing to a particular step in your undo buffer (e.g. go back 5 steps).

Q9: For the scenario described above, what ADT is appropriate?

One must use a deque, to allow removal of the 21st item from the back and the ability to add and remove of the top item.

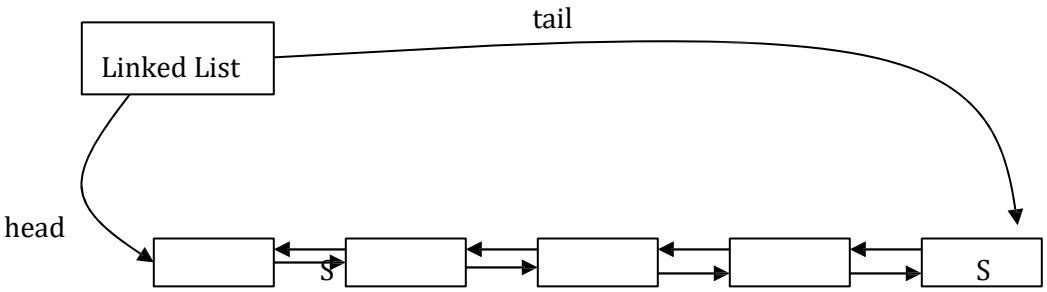
Q10: For the scenario described above, would you use the DynamicArray of LinkedList implementation of the ADT? Explain why in two sentences or less.

Dynamic Array. The linked list structure does not allow direct indexing, the entire list must be searched. A DynamicArray allows for random memory access.

Q11: The order elements are placed into the Bag ADT is as important as the order of placing elements into the Queue ADT.

False

For the next two questions, consider the following doubly-linked list with two sentinels (one at the head and one at the tail).




```
struct DLink
  { TYPE
  value;

  struct DLink *
  next; struct DLink
  * prev;

};
```

```
struct listDeque { int size;

  struct DLink
  *head; struct
  DLink *tail;

};
```

Q12: For the doubly-linked list depicted above, write the _addLinkAfter function that adds a link to the linked list after the provided link.

Note: This is one of many acceptable solutions

```
void _addLinkAfter(struct ListDeque *q, struct DLink *l, TYPE v)
{
        assert(q!=0);
        assert( l!=0);
        struct DLink *newLink = malloc(sizeof(struct DLink));
        assert(newLink!=0);
        newLink->value = v; newLink->next = l->next;
        newLink->prev = l;

        l->next->prev = newLink;
        l->next = newLink;

        list->size++
}
```

Q13: For the doubly-linked list depicted above, write the reverseLinkedList function that reverses the order of the linked list. Use assert() functions if necessary. Printing the values in reverse order is not an acceptable solution to this problem.

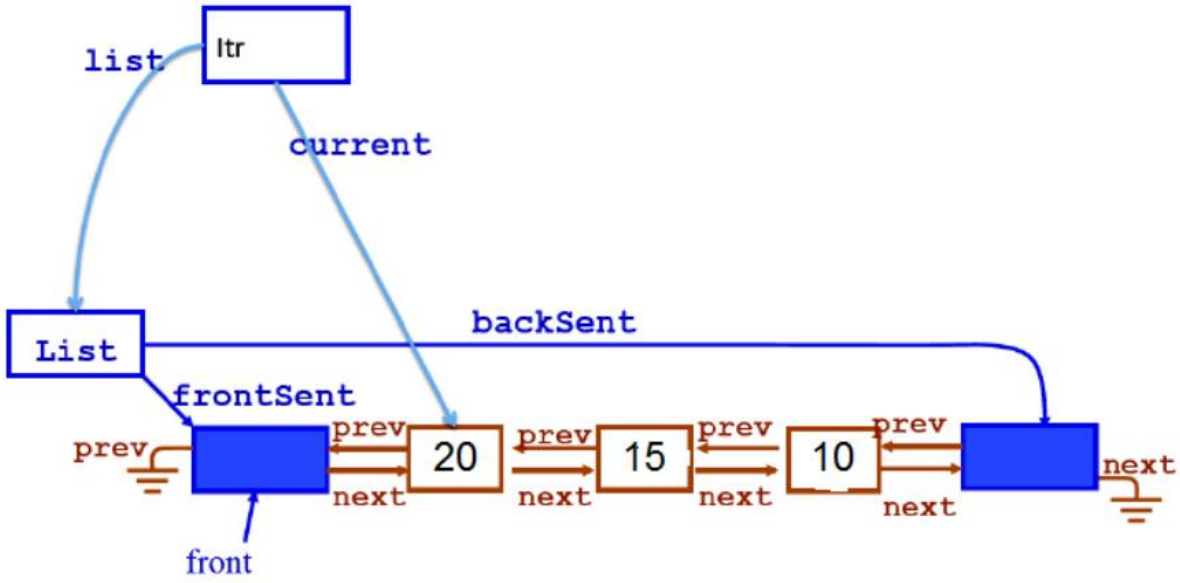
Note: This is one of many acceptable solutions

```
void reverseLinkedListBag(struct ListDeque *q)
{
        assert(q != 0);
        struct DLink *currLink = q->head;

        while (currLink != NULL) {
           struct DLink *temp = currLink->next;

           currLink->next = currLink->prev; currLink->prev = temp;
           currLink = temp;
        }

           /* switch the head and the tail */
           struct DLink * temp =q->tail;
           q->tail= q->head;
           q->head= temp;
}
```

Q14: Fill in the following average big-O execution times (in the Canvas textbox) for the simple un-ordered dynamic array bag, the linked list bag, and the ordered dynamic array bag.

	dyn array bag	linked list bag	ordered array bag
add	**O(1)+**	**O(1)**	**O(n)**
contains	**O(n)**	**O(n)**	**O(log n)**
remove	**O(n)**	**O(n)**	**O(n)**

Q15: Consider the following configuration for an iterator on a linked list:



Suppose, the implementation of iterator increments the current pointer during Next() If we execute the following sequence of operations, what is the expected output?

```
if(hasNext(itr)){
  if(hasNext(itr)){
    printf("%d\n",next(itr));
    remove(itr);
    printf("%d\n",next(itr));
  }
}
```

P.S: In this case, ignore the general assumption/rule that calls to hasNext() and Next() are interleaved.

Answer:

15

10

Q16: To perform a binary search in O(log n) time, what TWO properties must your data structure have?

1. It must be ordered

2. It must be able to be accessed randomly (any arbitrary element can be retrieved in constant time)/ It must be an array.