CSC 362 Program #6
Due Date: Wednesday, April 26

In this lab you will experiment with an array of linked lists to test out an idea called *hashing*, which you may have implemented in CSC 364. The idea behind hashing is to make list access more efficient. In an ordered array, to find an item you use binary search. In either an unordered array or an ordered/unordered linked list you have to use sequential search. Binary search offers O(log n) complexity but sequential search is O(n). To insert and delete from a list no matter if ordered or not, array or linked list, the worst case complexity is O(n). Can we improve on these? With hashing its possible to insert, delete and locate items in O(1) time which is a significant improvement. This works because the location of an item in an array is determined solely by a *hash function*. Consider the following list of values stored in an array of 11: 14, 13, 92, 8, 22, 6 and 23. Our function is `f(value) = value % size` where *value* is the item you want to store or find, and *size* is 11 (the size of the array). After inserting the above values in the order given, the array looks like this:

Array index:	0	1	2	3	4	5	6	7	8	9	10
Value stored:	22	23	13	14	92		6		8		

Searching for an item merely requires applying the hashing function, `f(value)`. However, this simple form of hashing has a problem known as *collisions*. If we want to now store 24, we would want to place it at 24 % 11 = 2, but there's something at index 2 (13), so we have a collision. We have to find another location. Using the next available location (index 5) requires searching ahead three additional locations to find the first open location. So instead of being able to insert the value in 1 instruction, it takes a total of 4. Let's assume we have added 24 at index 5. Now we want to insert 11, which would go at index 0. This leads to 7 collisions and 11 is finally inserted at index 7. In the worst case, inserting/deleting/searching deteriorates to O(n).

Moving forward until we find a free space uses a process called linear probing. One variation of hashing is called as *chained hashing*. In this approach, the array is not an array of int values but instead an array of pointers where each pointer points to a linked list storing int values. For instance, array index 2 would point to a list of all items that hash to location 2. For our example, the list at index 2 would contain nodes storing values 13 and 24.

Two questions we might pose are: Should the linked lists storing the values be ordered or unordered? How big should the array be? In this assignment, you will implement hashing storage using linked lists in an effort to better understand the two questions posed here. Implement your program as follows.

Implement linked list functions to insert a new int value item into a linked list given its front pointer, search for an int value in a linked list to locate it given its front pointer, print the int elements stored in a linked list given its front pointer, and destroy a linked list given its front pointer. You will not have to implement a delete function. You will implement two versions of linked lists: ordered and unordered. Thus, you will have two insert functions (ordered and unordered) and two search functions (ordered and unordered) (recall that in an ordered list, once you reach a value > the one you are searching for, you can stop searching and report the item was not found).

The program will require two different structs. The first is a `node struct` storing the int `data` and a `next` pointer. The other is a `list struct` which will store a `front` pointer to the first element of a linked list and the list's current `number` of items stored there (an int). You will then create two arrays of list structs (no more than 25 array elements will be needed). These will be your hash tables: one which has pointers pointing to ordered lists of values and one which has pointers pointing to unordered lists of values.

Create an array of 1000 int values and randomly generate values to insert into this array. Elements should be between 0 and 999,999. Iterate through the array and insert each value into both hash tables. Your insert functions should both count the number of operations required to do the given insert. For the unordered insert, always insert at the beginning. This should always take 1 operation. For the ordered insert, the number of operations is equal to the location in the list where the item was inserted (for instance, 5 if inserted after the 4th element). After inserting all 1000 elements, next iterate through the same array of 1000* random values and search for each value in both hash tables. Again, the search function should count the number of operations. If the item is found at index 5, this would be a count of 5. Recall that in the ordered list, once you find a value > the value you are searching for, you can stop searching. Thus, searching for an item not in the list should yield a better result in the ordered list than in the unordered list. * - for every 3rd element sought, rather than accessing it from the original array of 1000 int values, generate a new random number from 0 to 999,999. These should be values not found and thus your search through the ordered list should have a shorter search than the search through the unordered list.

The program's output should be the size of the hash table array (e.g., 23, 24, 25) and the total number of operations needed to do the inserts and searches for the ordered list and for the unordered list as a point of comparison. Also output for each of your runs the largest sized list and the shortest sized list. The size of each list will be the same between the two implementations (ordered vs unordered) so you don't need to search both sets of lists. For instance, if your ordered list's 18th element has 52 elements and that's the largest, just output 52. You will run your program three times, once on hash tables whose arrays are size 23, once on hash tables whose arrays are size 24, and once on hash tables whose arrays are size 25. Experimental research has shown that prime numbers are often better in hashing.

Therefore, your program will experiment with what size is best and what type of list implementation is best.

Here is how your program should work in detail:

- Declare the array of 1000 int values, let's call it `values`
- Declare two arrays of 23, 24 or 25 list structs, lists1 and lists2
- Seed your random number generator
- Initialize all of the pointers in lists1 and lists2 to NULL and their sizes to 0
- Randomly generate 1000 int values between 0 and 999,999 and place them into `values`
- Iterate for each item in `values`
    - Insert `values[i]` into both linked lists using the hashing function `values[i] % size` counting the number of steps it took for both insertions. As the insert functions will need to return a pointer to the front of the list, the counters will have to be passed to the insert functions as addresses. For the unordered insert, always insert at the beginning of the list so the counter for the unordered lists will always be incremented by 1 for any insert. For the ordered insert, the counter will be incremented by the

number of elements you have to search through before finding the proper location. If the list is empty or you are inserting in the front, add 1 to this counter. Remember to increment the appropriate lists' number of elements member (e.g., `lists1[location].number++` where location).

- Iterate 5000 times and either select the item from `values[i]` or generate a random number for every 3rd element and search for that value from the two lists again counting the number of operations it took to locate (or to determine the item is not in the list).
- Determine the length of the longest and shortest of the linked lists.
- Output the results of this run.
- Destroy the lists (deallocate all nodes)

Repeat this for three experiments: array of size 23, 24 and 25. Collect all of your results. An example is shown here for my program of size 23.

```
For array size 23
Number of Operations (ordered list):     32469
Number of Operations (unordered list):   29435
Longest list:                            52
Shortest list:                           37
```

NOTE: The print function is only needed for debugging purposes. Once your program is running correctly, do not print any items out but leave the print function in your code.

Submit your program and the output results. Include in comments a brief statement on the size and type of list that you felt worked the best.