Assignment 1: Dynamic Array and Linked List
Due at 11:59 pm on Monday, 7/4/2022

This assignment is intended to get you up and running with some of the tools we'll be using in this course and also to start programming in C. The assignment has three parts, described below.

Part 0. Download the skeleton code and unzip

You may download the skeleton code for this assignment here, or use the wget command:
wget https://classes.engr.oregonstate.edu/eecs/summer2022/cs261-001/assignments/assignment1.zip

To unzip the file, use the following command:
`unzip assignment1.zip`

Part 1. Implement a dynamic array

In part 1 of the assignment, you will implement a dynamic array.
The interface for the dynamic array (i.e. the structures and the prototypes of functions a user of the dynamic array interacts with) is already defined for you in the file `dynarray.h`. Your task is to implement definitions for the functions that are listed in `dynarray.c`.
Importantly, you may not modify the interface definition with which you are provided. Specifically, do not modify any of the already-defined dynamic array function prototypes. We will use a set of unit tests to test your implementation, and if you change the dynamic array interface, it will break these unit tests, thereby (negatively) affecting your grade. Beyond the already-defined interface, though, feel free to add any additional functions or structures your dynamic array implementation needs.
The dynamic array functions you'll need to implement are outlined briefly below. All of these functions use a type called `struct dynarray`, which is defined in `dynarray.c` and represents the dynamic array itself. For more details, including information on function parameters and expected return values, see the documentation provided in `dynarray.c`.

- `dynarray_create()` – This function should allocate, initialize, and return a pointer to a new dynamic array structure.

- `dynarray_free()` – This function should free the memory held within a dynamic array structure created by `dynarray_create()`. Note that this function only needs to free the `data` array itself. It does not need to free the individual elements held within that array.
- `dynarray_size()` – This function should return the number of elements stored in a dynamic array (NOT the capacity of the array).

- `dynarray_insert()` – This function should insert a new element **at the end** of a dynamic array. In other words, if the array currently contains n elements

(in indices 0 through n-1), then the new element (the n+1'th element) should always be inserted at index n. If there is not enough space in the dynamic array to store the element being inserted, this function should **double the size** of the array.

- `dynarray_remove()` – This function should remove the element at a specified index from a dynamic array. After the element is removed, there will be a "hole" where the element used to be. All elements after the removed one (i.e. with higher indices) should be moved forward one spot to fill that hole. In other words, if the element at index i is removed, then the element at index i+1 should be moved forward to index i, the element at index i+2 should be moved forward to index i+1, the element at index i+3 should be moved forward to index i+2, and so forth.

- `dynarray_get()` – This function should return the element value stored at a specified index in a dynamic array.

- `dynarray_set()` – This function should update (i.e. overwrite) the value of an element at a specified index in a dynamic array.

Part 2. Implement a Singly Linked List

In part 2, you will implement a singly linked list.
Again, the interface for the linked list (i.e. the structures and the prototypes of functions a user of the linked list interacts with) is already defined for you in the file `list.h`.
Your next task in this assignment is to implement definitions for the functions that are listed in `list.c`.
Again, do not modify the interface definition with which you are provided. This will break the unit tests we use for testing, which will cause your grade to suffer. You may still feel free to implement any additional functions you need beyond the ones defined in the interface.

The functions here will make use of two different structures:

- `struct node` – This structure represents a single node in the linked list. It has one field in which to store the data element associated with the node and one field that will point to the next node in the list.

- `struct list` – This structure represents an entire list and contains a single field to represent the head of the list. This is the structure with which the user of your list implementation will interact.

As with the dynamic array, see comments in `list.c` for more information on function parameters, expected return values, etc. for the linked list interface. The linked list functions you'll need to implement are outlined briefly below.

- `list_create()` – This function should allocate, initialize, and return a pointer to a new linked list structure.

- `list_free()` – This function should free the memory held within a linked list structure created by `list_create()`, including any memory allocated to the individual links themselves. Note, though, that this function should not free the `values` held in the individual links.

- `list_insert()` – This function should insert a new value **at the beginning** (i.e. as the head) of a linked list. Importantly, this function will need to allocate a new `struct node` in which to store the new value and add that node at the head of the list. The current head should become the next element after the new one.

- `list_remove()` – This function should remove the **first instance** of a specified value from a linked list. If multiple instances of the specified value exist in the list, only the first (i.e. the one closest to the head) should be removed. If the specified value doesn't exist in the list, this function doesn't need to do anything. This function will be passed a *function pointer* that you can use to determine whether the value to be removed matches any of the values stored in the list. Importantly, this function will also need to free the memory held by the node being removed (it does not need to free the stored value itself, just the node).

- `list_position()` – This function should return the list position (i.e. the 0-based "index") of the first instance of a specified value within a linked list. If multiple instances of the specified value exist in the list, the "index" of the first one (i.e. the one closest to the head) should be returned. If no instances of the specified value exist in the list, this function should return the special value -1. This function will be passed a function pointer that you can use to determine whether the value to be located matches any of the values stored in the list.

- `list_reverse()` – This function should reverse a linked list **in place**, that is within the memory of the existing list, without allocating new memory.

Test your work

In addition to the skeleton code provided here, you are also provided with some application code in `test_dynarray.c` and `test_list.c` to help verify that your dynamic array and linked list implementations are behaving the way you want them to. In particular, the testing code calls the functions from `dynarray.c` and `list.c`, passing them appropriate arguments, and then prints the results. You can use the provided `Makefile` to compile all of the code in the project together, and then you can run the testing code as follows:

```
make
./test_dynarray
./test_list
```

Example output of these two testing programs using correct implementations of the dynamic array and linked list is provided in the `example_output/` directory.
In order to verify that your memory freeing functions work correctly, it will be helpful to run the testing application through `valgrind`, i.e.:

```
valgrind ./test_dynarray
valgrind ./test_list
```

Submission

In order to submit your homework assignment, you must create a **zip file** that contains `assignment1/` folder with your implementation. This zip file will be submitted to TEACH . In order to create the zip file, go to the directory where you can access the `assignment1/`, and use the following command:

```
zip assignment1.zip assignment1 -r
```