Lab 4: ADT List

For this lab, create a new directory named `lab4` under your `cs449` directory and create your program there:

```
mkdir lab4
cd lab4
```

For this lab, you can get the starter file (`lab4.c`) to your directory using the following command:

```
cp /afs/cs.pitt.edu/usr0/tkosiyat/public/cs0449/lab4.c .
```

For those who works on your own computer, the starter file is also available on the CourseWeb.

Data Structure List

For this lab, you are going to implement a data structure list using linked list to store a list of integer. Recall that to implement a linked list, we need a data structure called `node`. In C, we use `struct` to create a structure `node` as follows:

```
struct node
{
    int data;
    struct node *next;
};
```

As you may recall, `node` will be created on the fly whenever user add a new entry into our list. To crate a new node, it is the same as telling the computer to allocate the memory for the `struct node`. The computer will return back the location of the memory that is allocated for you. To do so, we use the function `malloc()` as shown below:

```
struct node *newNode = (struct node *) malloc(sizeof(struct node));
```

Note that the function `malloc()` returns an address of type `void` pointer. Thus, we need to cast it to the right type. The argument of the function `malloc()` is the size in byte to be allocate. So, we use `sizeof(struct node)` which returns the number of bytes of the structure `node`.

At this point, you already have basic element for linked list that can be used with various kind of ADT. But for this lab, we will use a linked list to implement an ADT List. **For this implementation, the first entry on the list will be at index 0**. Recall that and ADT list implementation using linked list consists of two components:

- `numberOfEntries` of type `int` which is used to keep track of the number of entries in the list.
- `firstNode` of type `node` (pointer to `struct node`) which is used to point to the first node of the link chain. Note that `firstNode` should point to `NULL` if there is no entry in the list.

Since a list consists of two components, we can use structure to implement one in C as follows:

```
struct list
{
    int numberOfEntries;
    struct node *next;
}
```

Operations (Functions)

For simplicity, we will limit the number of operations on lists. The following are functions and their signatures that you must implement:

- `struct list *constructList()`

  A list consists of two components, `numberOfEntries` and `firstNode`. The purpose of the function `constructList()` is to allocate a chunk of memory for a list, initialize both components, and return the address of allocated memory. For example, with the function `constructList()` in hand, to create a list named `myList`, we just have to perform the following:

  ```
  struct list *myList;

  myList = constructList();
  ```

  Note that you should use `sizeof(struct list)` which will give you the number of bytes of the structure `struct list`.

- `void add(struct list *aList, int newEntry)`

  This function adds a `newEntry` into the **end** of the given list. Note that C does not have objects. When you want to add something to a list, you need to tell which list do you want to add to. In doing so, you can have one function that can all items into any list. For example, if you have two list `myList1` and `myList2`, you can use the same function to add entries into those lists as follows:

  ```
  struct list *myList1;
  struct list *myList2;
  myList1 = constructList();
  myList2 = constructList();
  add(myList1, 5);
  add(myList2, 12);
  ```

- `struct node *getNodeAt(struct list *aList, int index)`

This function should return the pointer (address) of the node associated with the given `index` of the given `aList`. This function will be a helper function for other functions.

- `int removeEntry(struct list *aList, int index)`

  This function should remove the node associated with the given `index` from the given `aList` and return the data stored in that node. For this function, you **MUST** use the function `free()` to deallocate memory that is used by the removed node.

What to Hand In

First, let us go back up to our cs449 directory:

```
cd ..
```

Now, let us first make the archive. Type your username for the USERNAME part of the filename:

```
tar cvf USERNAME_lab4.tar lab4
```

And then we can compress it:

```
gzip USERNAME_lab4.tar
```

Which will produce a `USERNAME_lab4.tar.gz` file.

If you work on `cs449.cs.pitt.edu` (`thoth`) you can skip to the next section. **If you use your own machine, you need to transfer the file to `cs449.cs.pitt.edu` first**. This can simply be done by a command line. For example, assume that your username is `abc123` and you are in the same directory as the file `abc123_lab4.tar.gz`. To transfer the file to `cs449.cs.pitt.edu` use the following command:

```
scp abc123_lab4.tar.gz abc123@cs449.cs.pitt.edu:.
```

The above command will copy the file to your home directory in `cs449.cs.pitt.edu`. If you want to copy it to your `private` directory, use the following command:

```
scp abc123_lab4.tar.gz abc123@cs449.cs.pitt.edu:./private/.
```

Copy File to Submission Directory

We will then submit that file to the submission directory:

```
cp USERNAME_lab4.tar.gz /afs/cs.pitt.edu/public/incoming/CS0449/tkosiyat/sec1
```

Once a file is copied into that directory, you cannot change it, rename it, or delete it. If you make a mistake, resubmit a new file with slightly different name, being sure to include your username. For example `USERNAME_lab4_2.tar.gz`. **Check the due date of this lab in our CourseWeb under Labs/Recitations**.