Lab 10: Back to the Future – Linked Lists in C, Java Style!

In this week's lab, we will learn to implement a Linked List data structure in C. We will define a new type called, **`List`** that can be used as shown below:

```
List L;

L = newList(); // create a new list

for (int i=0; i < 10; i++)// Insert 10 numbers in it
   add(&L, rand()%100);
print(&L);
```

The first line defines a `List`, L. Notice how L is being sent as an argument (with a reference, &L). You can change the implementation so L is defined as a pointer to a `List`, in which case you can send L as is, as a pointer. We will discuss this alternate design in class. Next, we populate the list with 10 random integers in the range [0..99]. Lastly, we use the print function to print out the contents of the list, L. Thus, List is a list of integers. Later we can modify it to be a list of any type (e.g., `Airport`, `Flight`, etc.). First, here are all the function that we would like to define for the List type:

Table 1 The List datatype.

Type/Function Specification	Description
List	The List data type
List newList();	Creates a new empty list
int size(const List *L);	Returns the size of list-L
int empty(const List *L);	is the list-L empty?
void clear(List *L);	removes all items from list-L
void add(List *L, int item);	Add item at end of list-L
int get(const List *L, int index);	Returns item at index in list-L
int contains(const List *L, int item);	Does list-L have item?
void print(const List *L);	prints contents of list (test only)

The List Data Structure

Each list will contain a header containing three items (Figure 1):

- **size** - # of items currently in the list.
- **head** – A pointer to the first element in the list.
- **tail** – a pointer to the last element in the list.

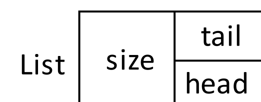

Figure 1 The List Structure

Each item in the list will be stored in a node containing the following fields:

- **data** – the data stored in this node.
- **next** – the next node in the list.

Figure 2 The node.

For example, the list of integers, [23, -103, 87, 13] will be stored as shown below:

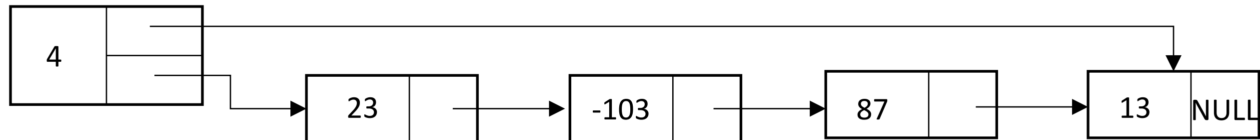

Figure 3 Example List containing [23, -103, 87, 13].

In order to implement the above, we will define two structs – **`List`**, and **node**:

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

struct node *newNode(int item);
```

We have also specified above that a function `newNode()` be defined. It returns a pointer to the the new node it created (containing the integer, `item` in its `data` field). Its `next` field will be set to `NULL`. Go ahead and create a file called, **node.h** and enter the above definitions. Complete the design of `struct` node by adding the following in a file, **node.c**:

```
#include <stdlib.h>
#include "node.h"

struct node *newNode(int item) {  // Creates a new node with item
   struct node *n = malloc(sizeof(struct node));
   n->data = item;
   n->next = NULL;
} // newNode()
```

Question: Why do we need to include the `<stdlib.h>` library above?

Next, we need to define List.

```
typedef struct {
   int size;
   struct node *head;
   struct node * tail;
} List;
```

Additionally, for each of the functions defined in Table 1, you will need to add the function headers/prototypes. These are listed below:

```
List newList();                  // Creates a new empty list
int size(const List *l);         // Returns the size of list-l
int empty(const List *l);        // is the list-l empty?
void clear(List *l);             // removes all items from list-l
void add(List *l, int item);     // Add item at end of list-l
```

```
int get(const List *l, int index);  // Returns item at index in list-l
int contains(const List *l, int item);      // Does list-l have item?
void print(const List *l);       // prints contents of list (test only)
```

To begin, you can create a file, **List.h** with the above contents. But doing so, you will need to define ALL the functions in the file, List.c. It is normally a good idea to implement the functionality slowly, one function at a time. This way you can test each function as you go, and then add the next function when ready. Let us begin by defining just the `newList()`, `size()`, `empty()` functions (in file, List.c):

```
#include <stdlib.h>
#include "List.h"

List newList() {            // Creates a new empty list
  List *L = malloc(sizeof(List));
  L->head = NULL;
  L->tail = NULL;
  L->size = 0;
  return *L;
} // newList()
```

Review the diagram of a `List` in Figure 1 and also in the definition above. Note that we need to allocate a new struct, and then initialize it as shown above. We now have an empty list of size 0. Therefore, we can define the two functions `size()`, and `empty()` next:

```
int size(const List *l) {        // Returns the size of list-l
   return l->size;
} // size()

int empty(const List *l) {       // is the list-l empty?
   return l->size == 0;
} // empty()
```

Just with the program we have so far, we can write a small test program to try out these features. Here it is:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#include "List.h"

int main(void) {

   List L;
   L = newList();
   if (empty(&L))
     printf("The list is empty.\n");
   else
```

```
     printf("The list is not empty.\n");

   printf("The list contains %d elements.\n", size(&L));

   return(0);
} // main()
```

Place the program above in a file, **testList.c**. Compile each of the three source files: **node.c**, **List.c**, and **testList.c**. Produce an executable and run the program. Check the output of the program to see it matches what you have done so far. When ready, do the task described below:

Task#1: Create a **Makefile** for this program. Use it and make sure it is a working Makefile. From now on, you will be using this file to build each version of the program.

Extending the List Implementation

Next, let us define the `add()` function. This function inserts the given item a t the end of the list. With `add()` available, we can then write the following:

```
for (int i=0; i < 10; i++)
      add(&L, rand()%100);
```

to insert 10 numbers (in the range [0..99] in the list, L in our test program from above. `add()` can be defined as shown below:

```
void add(List *l, int item) {    // Add item at end of list-l
   struct node *n= newNode(item);  // Create a new node (item, NULL)
   if (l->size == 0)       // Inserting in empty list
     l->head = l->tail = n;
   else {  // Inserting in a non-empty list
      l->tail->next = n;
      l->tail = n;
   }
   l->size++;   // We just inserted an item
} // add()
```

As we discussed in class, `add()` has two cases: inserting in an empty list; and inserting in a non-empty list. Study the function definition carefully and make sure you understand it.

Before we can test this new function, it would be a good idea to define the `print()` function from Table 1. This will allow us to view the contents of the list:

```
void print(const List *l) {
   struct node *n = l->head;
   printf("L(%d) = ", l->size);
   while (n != NULL) {
      printf("%d ", n->data);
```

```
      n = n->next;
   }
   printf("\n");
} // print()
```

The `print()` function is very useful. For example, we can add the following commands to the testList.c program:

```
for (int i=0; i < 10; i++)          // Insert 0, 1, 2, …, 9 in the list
    add(&L, i);
print(&L);
```

We are now ready to test this new code.

Task#2: Go ahead and compile your program (use the `Makefile`!). Make sure that it prints out the contents of the list as:

```
L(10) = 0 1 2 3 4 5 6 7 8 9
```

Once this is working correctly, replace the commands in testList.c to insert 10 random numbers (see above). Again, test, observe the output, and confirm that the program is doing what it is supposed to do. Also, add command(s) to print out the empty list, using `print()`.

You see, once you have a small, working program, it is easy to incrementally add functionality, and test. After Task#1, we only added two small functions, and three lines of code to testList.c. But that allowed us to test the new functions thoroughly. At the same time, our program (to implement and test all the functions in Table 1) is 2/3rd complete!

Retrieving Elements from the list: get()

Now that we can define a list, insert integers into it, we can focus on retrieving items from a list, using get:

```
int get(const List *l, int index);  // Returns item at index in list-l
```

`get()` requires us to "walk" down to the `index`th element to retrieve it. We will need to create a node pointer, start from the head of the list, and then hop over to the correct element. However, we should also make sure that the index provided is a valid index! If the index provided is incorrect, we should choose to halt the program after issuing an error message.

```
int get(const List *l, int index) {// Returns item at index in list-l

   if (index < 0 || index >= l->size) {
      printf("Error: List index out of bounds %d. Exiting!\n", index);
      exit(EXIT_FAILURE);
   }

   // index is valid, lets walk…
   struct node *n=l->head; // start at head
   for (int i=0; i < index; i++)
```

```
      n = n->next;    // hop!
   return n->data;    // we're there!
} // get()
```

Once again, it is time to test!

Task#3: Test the `get()` function by retrieving a random location in the list (you can test by generating a random index between [-1..10]). That way you will also test the invalid indices.

Recycling memory!

Lastly, we will write the `clear()` function. `clear()` removes all items from the list and resets the list to a fresh, empty state. One way to do this would be:

```
void clear(List *l) {
   l->head = l->tail = NULL;
   l->size = 0;
} // clear()
```

You should know by now that this creates a memory leak! You did not recycle all the memory the list, `l`, may have been using to store its elements. Here is the proper way to write `clear()`:

```
void clear(List *l) {      // removes all items from list-l
   struct node *n = l->head;
   struct node *nxt;
   while (n != NULL) {     // Visit each node and recycle it
      nxt = n->next;
      free(n);
      n = nxt;
   }
   l->head = l->tail = NULL;     // All recycled! Now reset.
   l->size = 0;
} // clear()
```

Task#4: Test the above before proceeding.

Task#5: Based on what you have learned so far, implement the `contains()` function and test it to make sure it is correctly implemented. Look for items that are there, as well as not there. Make it look for the first/last item in the list.

After completing Tas#5, you have a fairly complete and functional implementation of a linked list of integers. Think about how you would modify and use it to store another type of data, like Airport, from previous labs/assignments.