Lab 01: List ADT

Before we start with anything else, welcome to CMSC123: Data Structures. In this class we will implement lots of *ADTs* and try to analyze their performance using the *“komsay”*-way of measuring algorithmic performance. (*How do you do that anyway? Soon, we will know in a later topic.*)

Abstract Data Types

Most programming language, like C, offers predefined data types (*e.g.* `int` and `float`). Each of these types is also associated with a set of operations. For example, the data type `float` supports numerical operators (*e.g.* `+` and `-`), relational operations (*e.g.* `<` and `>`), and other numerical functions (*e.g.* `abs()` and `sin()`). These predefined data types provide the building blocks to create more sophisticated data types, which can be a collection of related and organized data. In order to differentiate the data types you create from the predefined data types, we refer to them as **Abstract Data Types** or simply **ADTs**.

As stated before, an ADT is a more complex (user-defined) data type which is a collection of related data (*of any type*), which also have a special data structure and its own set of operations.

On this laboratary, you will create your very own ADT (*or rather “recreate” since you’ve already done this in your CMSC21!*).

List ADT

List is simply collection of homogeneous data, wherein we can add and delete items from. Data items (which are `ints`) are organized using **singly-linked** nodes (each node contains the data). The `LIST` ADT as described in `list.h` is as follows:

```
typedef struct list_tag{
    NODE* head;
    NODE* tail;
    int length;
    int maxLength;
}LIST;
```

The data items in our list are described below:

1. `head` : the head pointer of the linked-list.
2. `tail` : the tail pointer of the linked-list.
3. `length` : the current length of the linked-list.
4. `maxLength` : the maximum number of elements that can be contained by the linked-list.

Each *singly* node contains the `int` data and a pointer to the next node, as illustrated below:

```
typedef struct node_tag{
    int value;
    struct node_tag* next;
}NODE;
```

List Operations

Since ADTs have their own set of operations, associated list operations must be implemented for the `LIST` ADT. The following are the functions that must be implemented (this are already in `list.h`):

1. `NODE* createNode(int)` requires an integer parameter; this function creates and returns a new node with the given integer as its value.

2. `LIST* createList(int)` requires an integer paramter; this function creates and returns a new list whose maximum length is the given integer; other fields are also initialized.

3. `int isFull(LIST*)` requires a list (pointer); this function returns `1` if the given list is full; otherwise, this returns `0` .

4. `int isEmpty(LIST*)` requires a list (pointer); this function returns `1` if the given list is empty; otherwise, this returns `0` .

5. `void insertAt(LIST*, int, NODE*)` requires a non-full list, an integer, and a node as parameters; this function inserts the given node at specified integer position of list (Sample insertions are shown in `list.h`).

6. `int deleteAt(LIST*, int)` requires a non-empty list and an integer as parameters; this function deletes the node at specified integer position of list (Sample deletions are shown in `list.h`); this also returns the value of the deleted node.

7. `void clear(LIST*)` requires a non-empty list; this function deletes all the contents of the given list.

`printList(LIST*)` is already given to help you in debugging your implementation.

Notes on Files in this Directory

Each file in this directory are described below:

- `README.md` is this file; the first file you must open and read, since this contains the description and guide for everything.

- `list.h` is a C-header file which contains important definitions for our `LIST` ADT, including structure definitions and forward declarations of functions (a.k.a function prototypes). You are **NOT ALLOWED** to change any part of this file.

- `list.c` is a C file which will contain all implementations of all functions declared in `list.h` (other helper functions can also be added here). We will make this a standard practice - we will give you a header (e.g. `file.h`) file and you will implement all functions in a corresponding implementation file (e.g. `file.c`).

- `main.c` is a simple interpreter for a simple shell program that interacts with the `LIST` ADT. Using a shell program is easier for testing our ADTs.

- `program.cs` is the shell program (you can call this a C-shell, *pun intended*) that manipulates a list. The commands for this shell are described in a latter section. The expected output for this shell program is in `expected.out`

- `expected.out` is the expected output result when `program.cs` is run.

- `Makefile` is a configuration file for the `make` utility to ease our *code, compile, link, execute* cycle. If you want to learn about the `make` utility, go here (///P://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/%29). Usage of `make` for this `Makefile` is described in a latter section.

Shell Commands

A simple shell program can be created to interact with the `LIST` ADT. The available commands are described below:

- `+ ix v` will insert the value `v` at index `x` ; `i` is just a prefix for indices; `x` and `v` must be valid integers.
- `- ix` will delete the value at index `x` ; `x` must be a valid position.
- `E` will report if list is empty or not.
- `F` will report if list is full or not.
- `p` will report the contents of the list.
- `Q` will terminate the program.

Invalid commands will be reported in the `stdout`

Manual Build

To compile and run your implementation, follow the steps below:
1. Compile your implementation file. This will produce `list.o` .

```
gcc -c list.c
```

2. Compile the driver program. This will produce `main.o`

```
gcc -c main.c
```

3. Link the two object files to create an executable file.

```
gcc -o run main.o list.o
```

4. Run the program and use `program.cs` as its input.

```
./run < program.cs
```

Steps above can be simplified into two steps:

1. Compile `.c` files together and create the executable file.

```
gcc -o run main.c list.c
```

2. Run the program and use `program.cs` as its input.

```
./run < program.cs
```

The steps above are automated using a `Makefile`.

`make` commmands

The following `make` commands are available:

- `make run` will compile and build the program; then the shell program is executed.
- `make build` will create the executable file `run`
- `make compile` will compile `main.c` and `list.c`.
- `make clean` will delete all object files created by the `make` command
- `make` default action is `make run`

Submission

Submit a `.zip` file named `SurnameEx01.zip` containing the following:

1. `list.h`
2. `list.c`
3. `main.c`

4. `program.cs`
5. `Makefile`

Upload your `zip` file in our Google Classroom.

Questions?

If you have any questions, approach your lab instructor.