Exercise 01: List ADT (In-Lab)

List Functions for In-Lab Exercise

Starting with the answer key to the Pre-Lab Exercise we will incorporate the `appendTail` function in your code.

First, uncomment the `tail` field in our `LIST` data type definition:

```c
typedef struct list_tag{
    NODE* head;
    NODE* tail;
    //int size; //keep these two (2) lines commented out for the in-lab exercise
    //int limit;
}LIST;
```

You will only have to implement or modify the following `LIST` ADT functions for your In-Lab Exercise (these are already in `list.h`).

1. `void appendTail(LIST*, int, NODE*)` when appending the first and only node, both `head` and `tail` should point to it; requires a non-full list, an integer, and a node as parameters; this function appends the given node the last position of the list; you can use `tail` to easily append the new node and keep `tail` to always point to the last node (Sample append calls are shown in `list.h`).
2. `NODE* createNode(int)` same as in Pre-Lab; requires an integer parameter; this function creates and returns a new node with the given integer as its value.
3. `LIST* createList()` initialize `tail` to NULL; this function creates and returns a new empty list; the head field is also initialized to NULL.
4. `int isEmpty(LIST*)` same as in Pre-Lab; requires a list (pointer); this function returns `1` if the given list is empty; otherwise, this returns `0`.
5. `void insertHead(LIST*, NODE*)` when inserting the first and only node, `tail` should also point to it; requires a non-full list, and a node as parameters; this function inserts the given node before the first position of the list (Sample insertions are shown in `list.h`).
6. `int deleteHead(LIST*)` when deleting the only remaining node, `tail` should also be set to NULL; requires a non-empty list as parameter; this function deletes the node at the first position of list (Sample deletions are shown in `list.h`); this also returns the value of the deleted node.
7. `void clear(LIST*)` make sure that `tail` is also set to NULL; requires a non-empty list; this function deletes all the contents of the given list. HINT: you can call deleteHead function repeatedly to implement this.
8. `show(LIST*)`is already given to help you in debugging your implementation.

Take note that although there is no answer key provided, you can used the codes in the slides to answer this exercise. Make sure that you check your answers by drawing a stack trace of your code.

Notes on Files in this Directory

Each file in this directory are described below:

- `Exer01(InLab).pdf` 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.txt` 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. Usage of `make` for this `Makefile` is described in a latter section.
- `answers` the sub-folder containing the answer to the lab exercises

Shell Commands

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

- `a v` will append the value `v` as the new `tail` node; `v` must be a valid integer.
- `i v` will insert the value `v` as the new `head` node; `v` must be a valid integer.
- `d` will delete the value of the `head` of the list.
- `E` will report if the list is empty 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` commands

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 `<initials><surname>inlab01.zip` (e.g.`ajjacildoinlab01.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 questions, contact your lab instructor.