Notes for Programming in C Lab Session #3

11 October 2017

1 Introduction

The purpose of this lab session is to write some small programs that do pointer and structure manipulations.

2 Overview

In the last couple of lectures, you have learned how to define structures, pointers, and functions. In this lab, you will learn how to define functions to manipulate pointer-based data structures, by working with the simplest pointer-based data structure of all — the singly-linked list.

In C, a datatype for linked lists can be declared with the following structure declaration:

```
struct List {
  int head;
  struct List *tail;
};

typedef struct List List;
```

This defines a type `struct List` which consists of a `head` field containing an integer, and a `tail` field which contains a pointer to another `struct List`. (The typedef defines a type abbrevation `List` standing for the structure type `struct List`. This lets us write `List` in function prototypes and variable declarations rather than repeating the keyword `struct` over and over again – this is a common idiom in C programming!)

A "linked list" is then just a pointer to this structure type. Next, you will implement a small library of functions whose prototypes and specifications are given in `list.h`, and whose implementation will go in `list.c`.

3 Instructions

1. Download the `lab3.tar.gz` file from the class website.
2. Extract the file using the command `tar xvzf lab3.tar.gz`.
3. This will extract the `lab3/` directory. Change into this directory using the `cd lab3/` command.
4. In this directory, there will be files `lab3.c`, `list.h`, and `list.c`.
5. There will also be a file `Makefile`, which is a build script which can be invoked by running the command `make` (without any arguments). It will automatically invoke the compiler and build the `lab3` executable.
6. Run the `lab3` executable, and see if your program works. The expected correct output is in a comment in the `lab3.c` file.

4 The Functions to Implement

4.1 Basic Exercises

The following functions should be relatively straightforward to implement. If you find yourself writing a lot of code for these functions, you should step back and rethink your approach.

- `int sum(List *list);`

  This function takes a linked list `list`, and returns the sum of all the elments of the list, taking the empty list to have a sum of 0.

- `void iterate(int (*f)(int), List *list);`

  The `iterate(f, list)` function takes two arguments. The first argument is a function pointer `f`, which takes an integer and returns an integer, and the second argument is a list `list`. This function then updates the head of each element of `list` by applying `f` to it.

- `void print_list(List *list);`

  `print(list)` takes a list `list` as an argument, and prints out the elements. Try to print out the elements as a comma-*separated* list.

4.2 Challenge Exercises

Once you have done the basic exercises, you can try the challenge exercises, which involve more subtle pointer manipulations. These two functions are the basic routines used to implement merge sort, which can sort a linked list in $O(n \log n)$ time.

- `List *merge(List *list1, List *list2);`

  Given two increasing lists `list1` and `list2` as arguments, `merge(list1, list2)` will return a linked list containing all of the elements of the two arguments in increasing order.

  In this implementation, do not allocate any new list cells – it should be possible to merge the two lists purely through pointer manipulations on the two underlying lists.

- `void split(List *list, List **list1, List **list2);`

  Given a list `list` as an argument, update the two pointers to linked lists `list1` and `list2` with linked lists each containing roughly half of the elements of `list` each. Eg, if `list` is [0,1,2,3,4,5,6], you might set `list1` to be [0,2,4,6] and `list2` to be [1,3,5].

  (HINTS: taking alternating elements for `list1` and `list2` will make your life easier. Also, think about what to do when `list` has 0 or 1 elements.)

  In this implementation, do not allocate any new list cells – it should be possible to split the input list into two purely through pointer manipulations of the input.