Homework 2

Handout 5

Due Sunday, February 23 by 11:59pm

CSCI 334: Spring 2020

Turn-In Instructions

For this assignment, create one separate source code file for each question (e.g., `q1.c`).

Supply a `Makefile` (30 points) with one rule per homework question. The naming convention for targets should be the name of the source file without the `.c` extention. For example, `q1.c` should compile to `q1`. You must also provide an `all` target that builds all targets and a `clean` target that removes all of the binary files generated by the build targets.

For full credit, be sure that your code compiles without emitting warnings even when using the `-Wall` flag. Note that if you use your own computer to do this assignment, you should check your assignment using a lab machine before submitting, since different compiler versions do not always behave the same way. The final authority will be the lab environment.

Turn in your work using the Github repository assigned to you. The name of the Github repository will have the form `cs334hw2_<your user name>`. For example, my repository would be `cs334hw2_dbarowy`. You should have received an invite to commit in the repository in your email. If you did not receive this email, please contact me right away!

Honor code: You may collaborate with one or more people on this assignment, but you may not write code together. All submitted work must be your own original work. If you work with a partner, please submit a `collaborators.txt` file that includes their names.

This assignment is due on Sunday, February 23 by 11:59pm.

Reading

1. **(Required)** Read "Passing Pointers by Value" from the course packet.
2. **(Required)** Read "The Linked List: An Elegant, Recursive Data Structure" from the course packet.

Problems

Q1. (*70 points*) .... Linked Lists C

One of the most fundamental data structures in computer science is the linked list. A linked list is a *dynamic* data structure, unlike an array. A linked list can be used to store data in situations where the total amount of data that needs to be stored is not known ahead of time. The word *dynamic* refers to the fact that the size of the data structure can change while the program runs.

Linked lists have an elegant, recursive definition. A *linked list* is either:

- `NULL`, or
- a list node that stores a data element and points to a linked list (the "tail").

In this question, you will implement a linked list that stores a C string (`char *`) in each list node. You will also implement a small collection of utility functions for manipulating the list.

(a) Design a list node data structure using `struct`. Use `typedef` to name this data structure `listnode` so that you can refer to a list node pointer with `listnode *`.

(b) Write a list prepend function with the following type signature:

```
listnode *prepend(char *data, listnode *list);
```

`prepend` stores the given `data` in a new `listnode` where the given `listnode *` is the tail of the new list. `prepend` should `malloc` a `listnode`, store `data` in that node, store `list` in the tail, and return a pointer to the new node. The function should work even if `list` is `NULL`; in other words, a user can create a *new list* just by appending to `NULL`, for example:

```
listnode *list = prepend("hello", NULL);
```

The `prepend` function *should not* make a copy of the given C string; instead, it should just store a pointer to that string.

(c) Write a `head` function that takes a `listnode *` and returns the `data` item for the `listnode` at the head of the list. Calling `head` on a `NULL` list should return `NULL`.

```
char *head(listnode *list);
```

(d) Write a `tail` function that takes a `listnode *` and returns the tail of the list. Note that the tail of a list is the original list without the head `listnode`. Calling `tail` on a `NULL` list should return `NULL`.

```
listnode *tail(listnode *list);
```

(e) Write a `printlist` function with the following type signature:

```
void printlist(listnode *list, char *sepby);
```

that prints each stored C string, separated by the string specified by the `sepby` C string, in order. The function should *not* print a trailing `sepby`; instead, it should print a newline at the end of the list. For example, given the list that contains the strings `"hello"` and `"world"` and where `sepby` is `"zzz"`, the function should print:

```
hellozzzworld[new line]
```

(f) Write a `delete` function that takes a `listnode *` and `free`s all the `listnode`s in the given list, but *does not* free the memory pointed to by the `data` field of each `listnode`. If the rationale for this behavior seems unclear to you, have a look at part **Q1.**(l)i.

```
void delete(listnode *list);
```

(g) Write a `reverse` function that takes a `listnode *` and retuns a *new* list in the reverse order. As with other functions in this assignment, it *should not* make a copy of each C string.

```
listnode *reverse(listnode *list);
```

(h) Write a `length` function that takes a `listnode *` and returns an `unsigned int` representing the length of the list.

```
unsigned int length(listnode *list);
```

(i) Write a `fromfile` function that takes a `char *` representing the name of a file, reads each whitespace-separated word of the file into a `listnode` and returns the linked list representing the sequence of words in the file, *in order*.

```
listnode *fromfile(char *filename);
```

(j) Write a `zip` function that takes two `listnode *`, each one representing a different list, and returns a third, *new* list that represents the list formed by alternating between elements of the two given lists. In the event that one list is longer than the other, the longer list simply appends the rest of its elements to the new list after elements from the short list run out. Again, this function *should not* copy each C string.

```
listnode *zip(listnode *lst1, listnode *lst2);
```

For example, suppose the first list contains the strings `row`, `row`, `boat`, `down`, `stream` and the second list contains the strings `row`, `your`, `gently`, `the`, `merrily`, `merrily`, `merrily`, `merrily`. Then the zipped list should contain `row`, `row`, `row`, `your`, `boat`, `gently`, `down`, `the`, `stream`, `merrily`, `merrily`, `merrily`, `merrily`.

(k) Write a `main` function that allows a user to pass in the names of two files from the command line, and prints the following information:

- The number of words in each file.
- The first and last word from each file (properly handling the case that a file may be empty).
- The string obtained by zipping the two file lists and then calling `printlist` on the combined list.

(l) Finally, answer the following questions in a file called `PROBLEMS.md`.

i. Why shouldn't the `delete` function `free` data pointed to in the list? For completeness, you should consider the following two cases: *1*) data stored in the list comes from string literals, and *2*) a user reverses `list1`, yielding `list2`, and then `delete`s both lists. You will need to do a little research on your own to determine how string literals are stored in C.

ii. Suppose you wanted to implement an `insert` function that inserts an element into a list such that, if a list is already in sorted order, the function would create a new `listnode` and insert it in the appropriate location, returning a pointer to the modified list. What steps would you need to do in order to achieve this? Be sure to consider corner cases.

Supply your complete solution as a single C source code program called `q1.c`. For full credit, ensure that *all storage with allocated duration is deallocated* and that all files are closed before the program terminates.

Q2. (*6 points*) .... Bonus Question: Function Parameters

Sorting data structures that store sequences is a fundamental operation in computer science. However, as you probably know, there are many sorting algorithms, each with their own benefits and tradeoffs. Consequently, it is often convenient to have a generic `sort` method that can be parameterized with the desired sorting algorithm. Such a sort method can be especially convenient when a data structure is built by one programmer, but when sorting behavior needs to be determined by another.

Building such data structures is complicated in C by the fact that C does not have:

- generic types,
- lambda expressions[*], or
- interfaces, classes, abstract classes, inheritance, or polymorphism[†] of any kind.

But C *does* have pointers. And pointers can get the job done.

In this bonus assignment, write a `sort` function that takes a `listnode *` and a *function pointer* to a given sorting algorithm, and returns a `listnode *` to the sorted list. You will need to define the `sort` function's signature yourself.

You may assume that data items are C strings. As before, do not copy the C strings stored in the given list.

You can receive up to 5 additional percentage points on your assignment, one point for each algorithm implemented. Some sorting algorithms are:

- bubble sort,
- selection sort,
- insertion sort,
- quick sort,
- merge sort,
- heap sort, and
- shell sort.

There are many other interesting sorting algorithms not listed here (e.g., radix sort), so if you have a favorite or want to explore something new, find one on your own.

Given that you have already written an implementation of a linked list, you may find it convenient to refactor your entire homework assignment to use *separate compilation.* One additional bonus point will be awarded for projects that utilize separate compilation (for a grand total of 6 possible bonus points).

A separately compiled program means that: both your linked list and sort implementations will be separate C modules; they will be compiled to library ("object") files; and each question's `main` method will `#include` the necessary library files. To be clear, this means that when you are done refactoring your assignment, you will have the following files:

(a) `q1.c`, which contains your Question 1 `main` method,
(b) `list.c` and `list.h`, which contain your list implementation and function signatures, respectively,
(c) `q2.c`, which contains your Bonus Question `main` method,
(d) `sort.c`, and `sort.h`, which contain your sorting algorithm implementations and function signatures, respectively, and
(e) a `Makefile` so that your entire homework submission builds correctly.

[*]Don't worry if you don't know what these are just yet.
[†]Ditto.

To get started on separate compilation, see this excellent tutorial: `https://www.cs.bu.edu/teaching/c/separate-compilation/`.

Minimally, supply a complete solution as a single C source code program called `q2.c`. If you are doing the separate compilation step, create the files as specified. If you want to try separate compilation *without* attempting the bonus sort routines, you may also earn bonus credit for refactoring your homework submission.

Q3. (*0 points*) .... Optional Feedback

How hard was this assignment on a scale of 1 to 5? (where the number indicates the required number of beard-seconds[2] of matcha required to get through the assignment).

Do you have any additional comments or feedback that you would like me to know?

Please supply your answer as a `feedback.txt` file.

Q4. (*1 point*) .... Bonus

Do the readings "Passing Pointers by Value" or "The Linked List: An Elegant, Recursive Data Structure" have any errors? One bonus point will be awarded for every verified problem that you find and report.

Submit as a text file called `errors.txt`.