Homework 3

Due Friday, September 28 by 10pm

Handout 5

CSCI 334: Spring 2018

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 `cs334hw3_<your user name>`. For example, my repository would be `cs334hw3_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 Friday, September 28 by 10pm.

Reading

1. **(Required)** Refer to our readings on C.

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 as `listnode *`.

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

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

`prepend` stores the given data item 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`.

(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 with the `listnode` that contains the head element removed. Calling `tail` on a `NULL` list should return `NULL`.

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

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

that prints the 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 elements of the list, but *does not* free the memory pointed to by the `data` field of the `listnode`.

(g) Write a `reverse` function that takes a `listnode *` and retuns a *new* list in the reverse order.

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

(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.

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 that results from zipping the two files.

(l) Finally, answer the following questions in a comment section at the top of your file.

- Why shouldn't the `delete` function `free` data pointed to in the list? For completeness, you should consider the following two scenarios: (1) data stored in the list come from string literals, and (2) the case where 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 constants like string literals are stored in C.
- 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`.

Q2. (*20 points*) .... Bonus Question: Binary search trees in C

Binary search trees are another fundamental data structure in computer science. Like lists, they also have an elegant recursive formulation. A *binary tree* is either

- `NULL`, or
- a tree node that contains a data element, a left subtree, and a right subtree.

A *binary search tree* is a binary tree that observes an additional property called the *binary search property*: every data element stored in a binary tree node is greater than all of the data elements stored in the left subtree and less than all of the data elements stored in the right subtree. Note that this particular definition does not allow duplicate strings to be stored in the tree.

In this question, you will implement a binary search tree in C where the stored data element is a pointer to a C string. In this case, "less than" and "greather than" refer to the lexicographical ordering of strings (i.e, use `strncmp`). You will also implement a small collection of utility functions for manipulating the list.

(a) Design a binary search tree data structure using `struct`. Use `typedef` to name this data structure `treenode` so that you can refer to a binary search tree node pointer as `treenode *`.

(b) Write an `insert` function that maintains the binary search tree property. It should have the following type declaration:

```
treenode *insert(char *data, treenode *tree);
```

`insert` should `malloc` a `treenode`, store data in that node, insert the node into the appropriate location in the tree, and return a pointer to the modified tree. The function should work evenf if list is NULL; in other words, a user can create a new tree just by inserting into a NULL tree, for example:

```
treenode *bst = insert("3.1415", NULL);
```

(c) Write a `delete` function that deallocates all of the `treenode`s in the tree, but does not deallocate the string pointed to by a `treenode`.

(d) Write a `to_list` function that takes a `treenode *` and returns a `listnode *` which points to a list that represents the in-order traversal of the tree.

(e) Write a `main` function that allows a user to pass in the name of a file from the command line, which then reads in the file, word by word, and using the binary search tree you just created, prints a list of unique words from the file, in sorted order.

(f) Finally, for full bonus credit, refactor your project to use *separate compilation.* This means that you will make both your linked list and binary search tree implementations C modules; they will be compiled to library ("object") files; and each question's `main` method will `#include` the library files. To be clear, this means that when you are done refactoring your assignment, you will have the following files: `q1.c`, `q2.c`, `list.c`, `list.h`, `bst.c`, and `bst.h`. Be sure to modify your `Makefile` so that the modified project builds correctly.

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

Note that the tutorial gives `gcc` commands; you will need to adjust them to use `clang` instead.

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.

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

How hard was this assignment on a scale of 1 to 5? (where 1 = "very easy" and 5 = "very hard")

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

Does the reading "Introduction to the Lambda Calculus, Part 1" 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`.