Lab 3

Pointer Manipulation Lab

Due: Week of February 7, before the start of your lab section

This is an individual-effort project. You may discuss concepts and syntax with other students, but you may discuss solutions only with the professor and the TAs. Sharing code with or copying code from another student or the internet is prohibited.

The purpose of this assignment is to give you more confidence in C programming and to begin your exposure to the underlying bit-level representation of data. You will also gain practice with pointers and with file input/output.

The instructions are written assuming you will edit and run the code on your account on the *csce.unl.edu* Linux server. If you wish, you may edit and run the code in a different environment; be sure that your compiler suppresses no warnings, and that if you are using an IDE that it is configured for C and not C++.

Learning Objectives

After successful completion of this assignment, students will be able to:

- Recognize the hazards of indeterminate values.
- Use C's string functions from `string.h`[1].
- Use C's file I/O functions from `stdio.h`[2].
- Alias and reassign pointers.
- Create and traverse a linked list.

Continuing Forward

Being able to understand the mistakes in Sections 1.1 and 1.2 will help you avoid them in future labs. Being able to work with pointers – that is, with variables that hold addresses – will help you specifically in future labs that use pointers but more generally in future labs that require you to think about accessing memory.

[1] See §7.8.1 and §B.3 of Kernighan & Ritchie's *The C Programming Language*, 2nd ed.
[2] See §7.5, §7.7, and §B1.1 *The C Programming Language*, 2nd ed.

During Lab Time

During your lab period, the TAs will provide a refresher on linked lists and will describe Insertion Sort. The TAs will also describe some string functions and some I/O functions from C's standard library. During the remaining time, the TAs will be available to answer questions.

No Spaghetti Code Allowed

In the interest of keeping your code readable, you may *not* use any **`goto`** statements, nor may you use any **`continue`** statements, nor may you use any **`break`** statements to exit from a loop, nor may you have any functions **`return`** from within a loop.

Scenario

Working at the Pleistocene Petting Zoo certainly is proving to be interesting. You're glad that you don't have to worry about the problem of the giant sloths very slowly chasing their handlers, but now it seems that Archie has decided to try to write a program or two. At a glance, his code is smellier than the wooly rhinoceros' enclosure. But you take a closer look anyway to try to understand why his code acts strangely.

1 Stray Values in Memory

1.1 Pleistocene Petting Zoo Marquee

Archie shows you his first program, which he hoped would be used to greet guests, but it doesn't always work right:

```
1  /*****************************************************************
2   * This program will output
3   **          Welcome to the
4   **     Pleistocene Petting Zoo!
5   **
6   ** Get ready for hands-on excitement on the count of three! 1.. 2.. 3..
7   ** Have fun!
8   * With brief pauses during the "Get ready" line.
9   *****************************************************************/
10
11 #include <stdio.h>
12 #include <unistd.h>
13
14 void splash_screen() {
```

```
15      const char *first_line = "\t     Welcome to the\n";
16      const char *second_line = "\tPleistocene Petting Zoo!\n";
17      printf("%s%s\n", first_line, second_line);
18  }
19
20  void count() {
21      int i;
22      sleep(1);
23      printf("Get ready for hands-on excitement on the count of three! ");
24      while (i < 3) {
25          fflush(stdout);
26          sleep(1);
27          i++;
28          printf("%d.. ", i);
29      }
30      printf("\nHave fun!\n");
31  }
32
33  int main() {
34      splash_screen();
35      count();
36      return 0;
37  }
```

Sometimes the output was what he expected:

```
     Welcome to the
 Pleistocene Petting Zoo!

Get ready for hands-on excitement on the count of three! 1.. 2.. 3..
Have fun!
```

But sometimes the output was missing the "`1..  2..  3..`":

```
     Welcome to the
 Pleistocene Petting Zoo!

Get ready for hands-on excitement on the count of three!
Have fun!
```

What mistake did Archie make? What change to *one* line will fix Archie's bug? Place your answers in *answers.txt*.

1.2 Math Doesn't Work Right . . . Or Does It?

"Wow," Archie says. "I can't believe that I missed that. Maybe you can help me with some other code that I wrote for one of my other start-ups. Here it is." Archie shows you the code:

```
1  /*****************************************************************
2   * This program will add two numbers and then it will multiply two other
3   * numbers. Finally, it will subtract the second result from the first
4   * result.
5   *****************************************************************/
6
7  #include <stdio.h>
8
9  int *add(int a, int b) {
10     int result = a + b;
11     return &result;
12 }
13
14 int *multiply(int p, int q) {
15     int result[1];
16     result[0] = p * q;
17     return result;
18 }
19
20 int main() {
21     int *sum = add(4, 5);
22     printf("sum = %d\n", *sum);
23     int *product = multiply(2, 3);
24     printf("product = %d\n", *product);
25     printf("sum - product = %d\n", *sum - *product);
26     return 0;
27 }
```

Archie explains that when he compiles the program with the **clang** compiler and then runs it, he gets this output:

```
sum = 9
product = 6
sum - product = 0
```

And when he compiles the program with the **gcc** compiler and then runs it, the program terminates with a segmentation fault. "I'm pretty sure that the segmentation fault is just gcc trying to protect me from trusting an incorrect answer – but why does the code produce an incorrect answer?"

What mistake did Archie make? Why does `*sum - *product` produce the value 0? Place your answer in *answers.txt*.

2 Challenge and Response

You plug in your shiny, new keyboard, tune your satellite radio to the Greatest Hits of the 1920s, and settle in to solving a more interesting problem.

To protect against corporate espionage, you are responsible for writing code for a challenge-and-response system. Anybody can challenge anyone else in the Pleistocene Petting Zoo's non-public areas by providing the name of a book and a word contained within the book, and the person being challenged must respond with another word from that book, based on certain rules:

- All of the book's words are sorted alphabetically without regard to capitalization (for example, "hello" occurs after "Hear" and before "HELP")
- The challenge word occurs *occurrences* times in the book
- If *occurrences* is an even number then the response word is the word *occurrences* places **before** the challenge word in the alphabetized list; if the challenge word is less than *occurrences* places from the start of the list then the response word is the first word in the list
- If *occurrences* is an odd number then the response word is the word *occurrences* places **after** the challenge word in the alphabetized list; if the challenge word is less than *occurrences* places from the end of the list then the response word is the last word in the list

Here is a simple example. Suppose the words in the specified book are:

word	*occurrences*
apple	7
banana	4
carrot	15
date	3
eggplant	2
fig	6
granola	9
horseradish	9
ice	6
jelly	3
kale	1
lemon	2
mango	8
naan	7
orange	5
pineapple	1
quinoa	11
raisin	4
spaghetti	10
tomato	12

If the challenge word is "horseradish" then because horseradish occurs 9 times in the book, the response word is "quinoa," which is 9 places in the list after "horseradish." If the challenge word is "eggplant" then the response is "carrot," which is 2 places earlier in the list than "eggplant." If the challenge word is "banana" then the response word is "apple," which is the first word in the list. If the challenge word is "quinoa" then the response word is "tomato," the last word in the list.

You break the problem down into four sub-problems:

1. Designing the Data Structure and Its Algorithms
2. Alphabetizing Words
3. Inserting Words
4. Responding to a Challenge

The Books

Four small "books" are included with the starter code:

- "Animals" (sorted, 7 words)
- "Plants" (unsorted, 7 words)

- "Cars" (sorted; 74 words)
- "Food" (unsorted; 125 words)

Two real books have also been reduced to one word per line:[3]

- Mary Shelly's *Frankenstein; Or, The Modern Promethius* (filename "Frankenstein") https://www.gutenberg.org/ebooks/84 (sorted; 74,363 words)
- Arthur Conan Doyle's *The Lost World* (filename "TheLostWorld") https://www.gutenberg.org/ebooks/139 (unsorted; 77,268 words)

The very small files of 7 words can be useful for debugging, and the moderate-sized files of 74-125 words should give you confidence in the correctness of your solution. The real books of more than 74,000 words will be used for grading (as will smaller problem instances). The files marked as *sorted* have all of their words already in alphabetically sorted order, ignoring capitalization; the files marked as *unsorted* do not have their words sorted (the words in "Plants" and "Food" are in a randomly-selected order; the words in "TheLostWorld" appear in the order that they appear in the original *The Lost World*).

Each book file, "*file*", has a corresponding "*file*-table.md" that contains a Markdown-formatted table of the challenge words, the number of occurrences for each challenge word, and the corresponding response word. You may use these files to confirm the correctness of your solution.

Throughout the assignment, we note that if building the list takes more than a few seconds, there is a bug in your code; for context, we can build the list for *Frankenstein* in under 1.9 seconds and the list for *The Lost World* in under 2.4 seconds. We can locate a word (or determine the absence of a word) in the *Frankenstein* list in under 0.5ms and in the *The Lost World* list in under 0.9ms.

2.1 Preparation

Copy your files *problem2.c* and *problem3.c* from KeyboardLab into the directory that has *pointerlab.c* and *challenge-response.c*. If you did not successfully complete KeyboardLab, we will provide alternate implementations for you.

2.2 Designing the Data Structure and Its Algorithms, part 1

You decide that a linked list is the best data structure option for the challenge-and-response system. You probably learned about linked lists in CSCE 156, RAIK 184H, or SOFT 161; however, we will provide a refresher.

[3]The text for these books was obtained from Project Gutenberg. In accordance with Paragraph 1.C of the Project Gutenberg License, all references to Project Gutenberg have been removed from the "derived works" that we are distributing. (Removing the references to Project Gutenberg was also necessary to ensure that *only* the words from the books are used for the challenge-and-response system.)

2.2.1 Singly-Linked List

For now, you will design a linked list that will work for challenge words with an odd numbers of occurrences; that is, challenge words whose response word is later in the list than the challenge word.

A *linked list* is linear collection of data. Like an array, each element (or *node*) has a particular position in the list, and when you iterate over the list, you always access the elements in the same order every time (unless you change or re-order the elements).

In an array, the elements are contiguous in memory, and you can access a specific element by indexing the array (or, equivalently, performing pointer arithmetic). In a linked list, however, the nodes can be in arbitrary locations in memory, and the nodes are connected by references (in C, pointers). You can access a specific element only by following pointers from one node to the next until you reach the desired node.

The simplest linked list is a *singly-linked list*. A node consists of a *payload* (the data that we care about) and a reference to the *next* node.

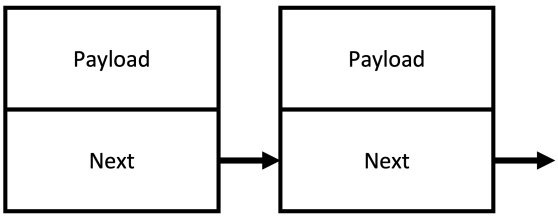

Figure 1: Nodes in a singly-linked list consist of the payload data and a reference that points to the next node.

A linked list's greatest advantage over an array is that inserting and removing a node at an arbitrary location takes constant time, whereas inserting an element into an array (assuming there is sufficient memory allocated for the array) or removing an element from an array requires moving all of the elements that follow the element's index. Inserting a new node, C between adjacent nodes A and B (where $B = A.next$) requires connecting $C.next$ to B and re-assigning $A.next$ to C.

As with an array, you do need to maintain a variable that points to the list. Conventionally, this is a reference to the *head* of the list. (Note that if a new node is inserted before the current head node, then the new node becomes the head of the list, and your `head` variable would need to be updated.) It is not uncommon to also maintain a reference to the *tail* of the list.

2.2.2 Equivalent Java Code

In Java, you probably wouldn't implement your own linked list; instead, you would use `java.util.LinkedList`, which has been available since J2SE 1.2. A list of hypothetical `Payload` objects would be created with:

```
List<Payload> payloads = new LinkedList<>;
```

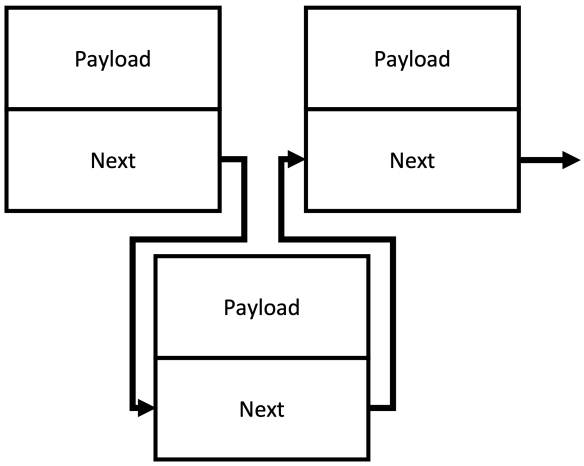

Figure 2: Inserting a new node into a singly-linked list only requires assignments to the affected *next* pointers.

C doesn't have a built-in linked list data type, so you will need to design one. Let us consider what a custom linked list would look like in Java.

```
 1 public class Node {
 2     private final String word;
 3     private int occurrences;
 4     private Node next;
 5     private Node previous;
 6
 7     public Node(String word) {
...         ...
12     }
13
14     public void insertAfter(Node existingNode) {
...         ...
23     }
24
25     public void insertBefore(Node existingNode) {
...         ...
34     }
...    ...
99 }
```

Creating and inserting a new node would look something like this:

```
200     Node node = new Node("eggplant");
201     Node otherNode = ... // code to determine where the new node goes
202     node.insertAfter(otherNode);
```

Recall that in Java, all variables except primitive types (such as `occurrences` on line 3) are references. This means that the `next` field on line 4 is a reference to another Node, just as we described in Section 2.2.1. The payload is the `word` and how many `occurrences` the word has, exactly what we need for the challenge-and-response system.

Recall also that in java, the **`new`** keyword allocates space for the new object, and the constructor call – `Node("eggplant")` – initializes the object.

2.2.3 C Implementation

In *challenge-response.h*, you'll see a **`struct`** with the same fields as our Java example:

```
36 struct node {
37     char word[MAXIMUM_WORD_LENGTH];
38     int occurrences;
39     struct node *next;
40     struct node *previous;
41 };
```

In *challenge-response.c*, you'll also see the **`create_node()`** function:

```
23 /* Allocates memory for a new node to hold the word, and initializes the
24  * node's fields. Returns a pointer to the new node. */
25 struct node *create_node(const char *word) {
26     struct node *new_node = malloc(sizeof(struct node));
27     /* WRITE THE REST OF THIS FUNCTION */
28     return new_node;
29 }
```

As you can see, it allocates space for a new node using **`malloc()`**. The code that you will need to add to it will copy the `word` argument into the `word` field and set an appropriate initial value for the `occurrences` field. Since we don't yet know where this node will go, set the `next` and `previous` pointers to `NULL`.

The other function you need to write now is **`insert_after()`**:

```
31 /* Inserts new_node into the list after existing_node; that is, new_node
32  * becomes existing_node's "next". If existing_node's original "next" is
33  * non-NULL, then that will become new_node's "next". */
34 void insert_after(struct node *existing_node, struct node *new_node) {
35     /* WRITE THIS FUNCTION */
36 }
```

As the name and documentation indicate, you need to add code that will update the nodes' `next` pointers so that `new_node` is placed in the list immediately after `existing_node`. For now, you can ignore the `previous` pointers.

After you have implemented **`create_node()`** and **`insert_after()`**, go to the **`main()`** function in textitpointerlab.c and un-comment the call to **`test_linked_list_functions()`**.

```
54 int main() {
55 //     test_linked_list_functions();
```

Build the executable with the command: `make pointerlab`. Be sure to fix both errors and warnings. When the program compiles without generating any warnings or errors, run it. The output should indicate a list with the nodes in the order of "first node," "fourth node," "second node," and "third node."

2.3 Alphabetize Words

In KeyboardLab Problem 2, you wrote code to convert uppercase letters to lowercase letters. Add code to **`word_to_lowercase()`** that calls that function to convert all letters in a word to lowercase letters (do not copy the **`to_lowercase()`** function into *challenge-response.c*; we will link to the function in *problem2.c*).

```
48 /* Returns a copy of the word that has all uppercase letters replaced
49  * with lowercase letters. The original string is unchanged. */
50 char *word_to_lowercase(const char *word) {
51     /* WRITE THIS FUNCTION */
52     return NULL;
53 }
54
55 /* Compares two words based on alphabetical order.
56  * Returns a negative value if word1 occurs alphabetically before word2.
57  * Returns a positive value if word1 occurs alphabetically after word2.
58  * Returns 0 if the two words are identical.
59  * This function is really just a wrapper around strncmp. */
60 int compare_words(const char *word1, const char *word2) {
61     return strncmp(word1, word2, MAXIMUM_WORD_LENGTH);
62 }
```

The starter code includes a function to compare two words (you do not need to write this function) but it assumes that both words are completely lowercase.

If you did not complete KeyboardLab, then place this code in your *problem2.c* file:

```
1 #include <ctype.h>
2
3 char decapitalize(char character) {
4     return tolower(character);
5 }
```

2.4 Inserting Words, part 1

Comment-out (or delete) the call to **`test_linked_list_functions()`**.

For this sub-problem, the user will be prompted to enter the name of a book, which will be the filename of a file that contains all of the book's words. All punctuation has already been removed from the files, and each line in the file contains exactly one word. For now, you will design the code to work with files whose contents are already sorted.

```
67 /* Determines if the word is already in the list. If it is, then the
68  * number of occurrences for that word is incremented. If it isn't, then
69  * a new node is created for the word and inserted into the list at the
70  * correct alphabetic location. Returns a pointer to the head of the
71  * list, which is either the original head or a node containing the word
72  * (if the word occurs before the original head's word or if the
73  * original head is NULL). */
74 struct node *insert_word(struct node *head, const char *word) {
75     /* WRITE THIS FUNCTION */
76     return NULL;
77 }
78
79 /* Given the name of the book file from the user, reads the file
80  * line-by-line. Under the assumption that there is exactly one word per
81  * line and that all punctuation has been removed, this function builds
82  * a doubly-linked list of the words in alphabetical order, keeping
83  * track (as part of a node's payload) how many times each word occurs
84  * in the file. */
85 struct node *build_list(const char *filename) {
86     /* WRITE THE REST OF THIS FUNCTION */
87     return NULL;
88 }
```

Add code to **`insert_word()`** and **`build_list()`** to read the specified file one line at a time.[4] For each word, convert it to lowercase, and then traverse the list to find the appropriate place for the word. (Note that there will not be a list to traverse when your code reads the first word!) If the word is not in the list then create a node for that word and insert it into the list at the correct location. If there is already a node containing that word, then increment that node's variable that tracks the number of occurrences. Be sure that *only* the word is placed in a node; specifically, do not include a newline character nor any other characters that are not part of the word.

Build the program and correct all warnings and errors. When the program compiles without generating any warnings or errors, run it. If your program requires more than a few seconds to build the list, there is a bug in your code. If your program does not produce the expected output, the **`print_list()`** utility function will help you see the list that your created.

[4] See §7.5 and §B1.1.1 of Kernighan & Ritchie's *The C Programming Language*, 2nd ed. for **`fopen()`** and **`fclose()`**, and §7.7 and §B1.1.4 for **`fgets()`**.

2.5 Respond to a Challenge, part 1

You now have implemented enough of the other sub-problems that you can write the code to respond to a challenge if the number of occurrences is odd.

```
93  /* Given an alphabetically sorted list of words with the number of
94   * occurrences of each word, and given the challenge_word, will return
95   * the response word based on the following rules:
96   * - If the number of occurrences is an even number then the response
97   *   word is that many places *before* challenge_word in the list
98   *   - If the challenge_word is fewer than that number of places from
99   *     the head of the list, then the response word is the word at the
100  *     head of the list
101  * - If the number of occurrences is an odd number then the response
102  *   word is that many places *after* challenge_word in the list
103  *   - If the challenge_word is fewer than that number of places from
104  *     the tail of the list, then the response word is the word at the
105  *     tail of the list
106  * - If challenge_word is not present in the list, then the response is
107  *   "<challenge_word> is not present!" */
108 char *respond(const struct node *list, const char *challenge_word) {
109     /* WRITE THIS FUNCTION */
110     return NULL;
111 }
```

After the word list is complete (after you have inserted all words in the file), the user will be prompted to enter the challenge word. Add code to **`respond()`** that traverses the word list to find that word. If the word is not present in the list, return "(word) is not present!", where "(word)" is the challenge word.

In KeyboardLab Problem 3, you wrote code to determine whether an integer value is even. Add code to **`respond()`** such that if the challenge word is present in the list and the number of occurrences is even, the function returns "(word) has (number) of occurrences", where "(word)" is the challenge word and "(number)" is the number of occurrences. If the number of occurrences is odd then use the number of occurrences recorded in that word's node to find the response word as described in the challenge-and-response rules, and return that word. (Do not copy the **`is_even()`** function into *challenge-response.c*; we will link to the function in *problem3.c*.)

If you did not complete KeyboardLab, then place this code in your *problem3.c* file:

```
1 int is_even(int value) {
2     return !(value % 2);
3 }
```

If your program does not provide the response word nearly instantaneously, there is a bug in your code.

Next Steps

You have now completed a significant portion of the assignment: you can build the word list if the words in the file are already sorted, and you can provide the response to a challenge word if the number of occurrences is odd. The next steps will require modifying some of your code, so this would be a good time to make a backup copy of *challenge-response.c* or to commit it to a private Git repository.

In Section 2.6 you will implement a *doubly-linked list* which will allow you to write code to respond to a challenge word if the number of occurrences is even. In Section 2.7 you will implement the *Insertion Sort* algorithm which will allow you to build a list even when the words in the file are not already sorted.

You may tackle either of the remaining tasks before tackling the other.

2.6 Designing the Data Structure and Its Algorithms, part 2, and Respond to a Challenge, part 2: Doubly-Linked List

A *doubly-linked list* is a linked list with the property that each node maintains a link not only to the `next` node but also a link to the `previous` node. In C, these links are pointers.

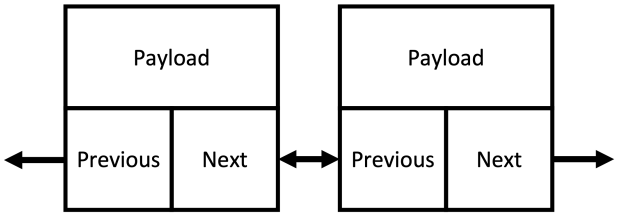

Figure 3: Nodes in a singly-linked list consist of the payload data and references that point to the previous and next nodes.

Inserting new node, C between adjacent nodes A and B (where $B = A.next$ and $A = B.previous$) requires connecting $C.previous$ to A and $C.next$ to B, and re-assigning $A.next$ to C and $B.prevous$ to C.

Modify your **`insert_after()`** function to update not only the `next` pointers but also the `previous` pointers so that `new_node` is placed between `existing_node` and the node that originally was located immediately after `existing_node`.

After the program compiles without warnings or errors, you may want to use **`print_list`** to confirm that the `previous` pointers are updated correctly.

Now implement **`insert_before()`**:

```
48 /* Returns a copy of the word that has all uppercase letters replaced
49  * with lowercase letters. The original string is unchanged. */
50 char *word_to_lowercase(const char *word) {
51     /* WRITE THIS FUNCTION */
```

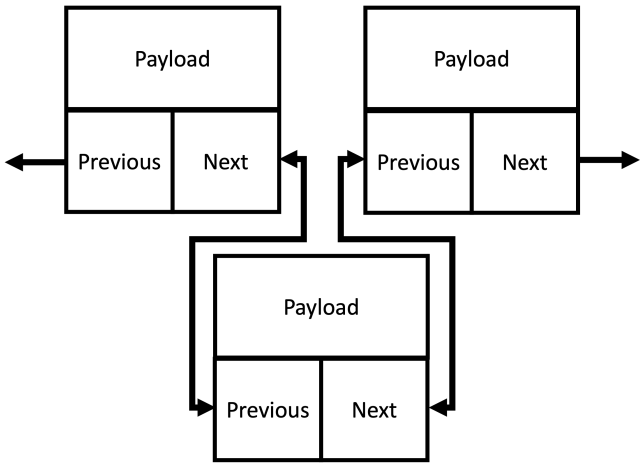

Figure 4: Inserting a new node into a doubly-linked list requires assignments to the affected *previous* and *next* pointers.

```
52      return NULL;
53  }
```

If your program requires more than a few seconds to build the list, there is a bug in your code.

Update your **respond()** function so that when the number of occurrences is even, the returned string is not "(word) has (number) of occurrences" but is instead the appropriate response word as described in the challenge-and-response rules.

You can now provide the response to a challenge word regardless of whether the number of occurrences is even or odd. This would be a good time to make a backup copy of *challenge-response.c* or to commit it to a private Git repository.

2.7 Inserting Words, part 2: Insertion Sort

While you probably learned about sorting in CSCE 156, RAIK 184H, or SOFT 161, you may not have learned about *Insertion Sort*. If you did learn about Insertion Sort, you probably learned that it's a $\mathcal{O}(n^2)$ algorithm that is less efficient than $\mathcal{O}(n\log n)$ sorting algorithms such as Merge Sort and Quick Sort. Insertion Sort has a particular advantage in that it can be applied *as the list is built*, making for a much simpler and less error-prone implementation than a different sort that requires the list to already be built.

The Insertion Sort algorithm reads an input and then traverses a sorted list to find the proper location in the sorted list for the input. The input is then inserted into the list at that location.

Your current implementation of **insert_word()** is based off of the assumption that the word is either at the tail of the list or is not present in the list. Update **insert_word()** so that it looks for the word in the sorted list. If the word is found, then update the number of occurrences as before. If the word is not present in its proper location in the sorted list,

then insert a new node for that word at its proper location in the sorted list.

If your program requires more than a few seconds to build the list, there is a bug in your code.

You can now provide build the word list regardless of whether the words in the file are already sorted. This would be a good time to make a backup copy of *challenge-response.c* or to commit it to a private Git repository.

Turn-in and Grading

When you have completed this assignment, upload *answers.txt* and *challenge-response.c* to Canvas.

This assignment is worth 30 points.

_____ **+2** Student's answers in *answers.txt* demonstrate an understanding of the bug in Section 1.1's code and how to correct it.

_____ **+2** Student's answer in *answers.txt* demonstrate an understanding of the bug in Section 1.2's code.

_____ **+1** **`create_node`** creates and initializes a **`struct node`** as specified.

`insert_after` correctly places a new node in a list by updating:

_____ **+2** `next` pointers (singly- and doubly-linked lists).

_____ **+2** `previous` pointers (doubly-linked lists).

_____ **+2** **`insert_before`** correctly places a new node in a doubly-linked list.

_____ **+2** **`word_to_lowercase`** returns a copy of the input string with uppercase letters replaced with lowercase letters

`insert_word`:

_____ **+2** creates a new node at the end of the list when the word properly belongs at the end of the list and is not already present in the list.

_____ **+2** does not create a new node but instead updates the number of occurrences, when the word is present at the end of the list.

_____ **+2** creates a new node at the appropriate location in the list, regardless of where its appropriate location is, when it is not already present in the list.

_____ **+2** does not create a new node but instead updates the number of occurrences when the word is present, regardless of its location in the list.

`build_list` opens a file for reading, builds a list by reading one line at a time and the word that is read to **`insert_word()`**, and closes the file after the last line has been read, when:

_____ **+2** the words in the file are pre-sorted, and the `next` pointers are updated (*i.e.*, part 1 of the sub-problems are complete).

_____ **+1** the words in the file are not pre-sorted, and the `next` pointers are updated (*i.e.*, Section 2.7 is complete).

_____ **+1** the words in the file are pre-sorted, and the `previous` pointers are updated (*i.e.*, Section 2.6 is complete).

_____ **+1** the words in the file are not pre-sorted, and the `previous` pointers are updated (*i.e.*, part 2 of the sub-problems are complete).

`respond` produces the correct response word in accordance with the specified rules when:

_____ **+1** the words in the file are pre-sorted, and the `next` pointers are updated (*i.e.*, part 1 of the sub-problems are complete).

_____ **+1** the words in the file are not pre-sorted, and the `next` pointers are updated (*i.e.*, Section 2.7 is complete).

_____ **+1** the words in the file are pre-sorted, and the `previous` pointers are updated (*i.e.*, Section 2.6 is complete).

_____ **+1** the words in the file are pre-sorted, and the `previous` pointers are updated (*i.e.*, part 2 of the sub-problems are complete).

Penalties

_____ **-1** Newline characters are included in the word strings when building a list.

_____ **-1** for each **`goto`** statement, **`continue`** statement, **`break`** statement used to exit from a loop, or **`return`** statement that occurs within a loop.

Epilogue

You hear somebody enter the room. "*Frankenstein*, 'boat'," is the challenge, and she answers, "borne." Archie introduces you to the new arrival, "Lil, this is our new developer, the one who wrote the app we just used." He turns to you: "This is Lilith Redd from business operations." He turns back to her and continues, "Lil, what's the good word?"

"The word isn't good, I'm afraid. I just heard back from the insurance company."

To be continued...