Assignment 8 — Structs and Allocation

Due: Friday, March 16th

The Unix utility `grep` can be used to search a file for occurrences of a word. For this assignment, you'll use command line arguments, file I/O, and dynamic memory allocation to implement a similar program in C.

Deliverables:

Demo — *none*

Write-up — *none*

Handin — `myGrep.c`, `wordList.h`, `wordList.c` (required)
`myGrep2.c`, `wordList2.h`, `wordList2.c` (extra credit)
GitHub Classroom Link: `https://classroom.github.com/a/PEC928WB`

This is an **individual assignment**. You are not allowed a lab partner, nor are you allowed to work together, even if you write your own code. All your work must be your own.

Part 1: Ground Rules

- You may not use any global or static local variables.
- You must free any memory you allocate and close any files you open.
- All of your code must compile on Cal Poly's Unix servers using:

```
gcc -Wall -Werror -ansi -pedantic
```

- Do not alter the given prototypes or header file, and do not write code in any files other than those specified above. Your functions will be tested individually.

Part 2: A List of Occurrences

Begin by accepting the GitHub Classroom assignment: `https://classroom.github.com/a/PEC928WB`.

As your program reads through a given file for occurrences of a given word, it will store those occurrences in a linked list. `wordList.h` contains a definition of a linked list's node, along with the prototypes of three functions that operate on that linked list.

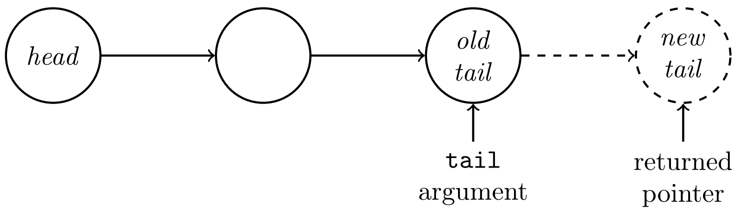
Begin by implementing the following function within `wordList.c`, which allocates, initializes, and adds a single node representing a single occurrence to the end of the linked list:

```
Node *addToTail(Node *tail, char *line, int lineNum, int wordNum);
```

- Adds a new occurrence to the end of a list.
- Takes four arguments:
  - `tail` — A pointer to the current tail of a list, `NULL` if it's empty
  - `line` — The line in which the word occurs
  - `lineNum` — The number of that line within the file
  - `wordNum` — The position of that word within that line
- Returns a pointer to the new tail of the list.

Requirements:

- Include any additional required libraries at the top of `wordList.c`.
- Do not use any helper functions.
- Your function must dynamically allocate memory for a new node.
- Your function must initialize the new node's member variables using the values passed as arguments.
- You must use the given `Node` definition in `wordList.h`.
- Your function must add a new node to the end of a linked list, for example:



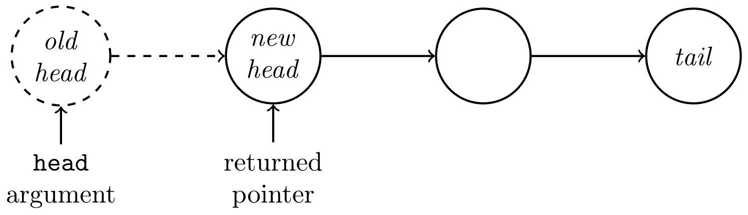
Since the nodes created by `addToTail` are dynamically allocated, they must eventually be removed from the list and freed before your program terminates. Implement the following function within `wordList.c`:

```
Node *rmFromHead(Node *head, char *line, int *lineNum, int *wordNum);
```

- Removes an occurrence from the beginning of a list.
- Takes four arguments:
  - `head` — A pointer to the current head of a list, `NULL` if it's empty
  - `line` — A pointer at which to store the removed line
  - `lineNum` — A pointer at which to store the removed line number
  - `wordNum` — A pointer at which to store the removed word number
- Returns a pointer to the new head of the list, `NULL` if it was or is now empty.

Requirements:

- Do not use any helper functions.
- Your function must free the memory that was allocated (by `addToTail`) for the node being removed.
- Your function must store the removed node's member variables using the pointers passed as arguments.
- You must use the given `Node` definition in `wordList.h`.
- Your function must remove a node from the beginning of a linked list, for example:



As you write and debug your complete program, it will be useful to be able to print out an entire linked list. Implement the following function within `wordList.c`:

```
void printList(Node *head);
```

- Prints out every node in the list, according to the following format:
  ```
  Node:
   - line:     <line>
   - lineNum: <line number>
   - wordNum: <word number>
  ```
- Takes one argument:
  - `head` — A pointer to the head of a list, `NULL` if it's empty

Requirements:

- Do not use any helper functions.
- Your function should not alter the list, only traverse the list and print out each of its nodes' contents.
- You may decide whether to implement this function iteratively or recursively.

Part 3: Command Line Arguments and File I/O

Develop a program that accepts, as command line arguments, a word and the name of a file. This program should search for every occurrence of that word within that file, storing information about each occurrence in a linked list so that they can be printed out later.

Additionally, your program should find the longest line in the file (regardless of whether or not it contains the specified word) and count the number of lines in the file.

[*Hint:* Develop incrementally. Begin by making sure you can open a file, then read and echo each of its lines. You can use `fgets` to read single line from a file and `strtok` to iterate over the words in a line.]

Requirements:

- Write your program in a file named "`myGrep.c`".
- Before it begins its search, your program must verify and echo its command line arguments.
- Your string comparisons may be case-sensitive.
- You may assume that no line (or word) will be longer than 100 characters.
- You may *not* make any assumptions about how many times the word occurs in a line or in the file.
- You must read through the file exactly once.
- You must store each occurrence of the word in a linked list, using the functions in `wordList.c` and the definitions in `wordList.h`.
  - Store the line in which the word occurs.
  - Store the number of that line within the file.
  - Store the position of the word within that line.
  - Both the lines in the file and the words in a line are numbered from 0.
- You must free any memory that you dynamically allocate.

Sample Runs:

(Assuming your executable is named "`a.out`". Italicized characters are typed by the user. Your program will be tested using `diff`, so your output must match *exactly.*)

- Using the given sample text file:

```
>$ ./a.out the sample.txt
./a.out the sample.txt
Longest line (50 characters): Polyana, in the province of Tula. His mother died
Number of lines: 17
Total occurrences of "the": 7
Line 1, word 4: August 28, 1828, at the family estate of Yasnaya
Line 2, word 2: Polyana, in the province of Tula. His mother died
Line 4, word 2: Placed in the care of his aunts, he passed many
Line 7, word 1: entered the university. He cared little for the
Line 7, word 7: entered the university. He cared little for the
Line 11, word 4: English, and Russian novels, the New Testament,
Line 12, word 6: Voltaire, and Hegel. The author exercising the
```

- If the word cannot be found:

```
>$ ./a.out foo sample.txt
./a.out foo sample.txt
Longest line (50 characters): Polyana, in the province of Tula. His mother died
Number of lines: 17
Total occurrences of "foo": 0
```

- If the file cannot be opened:

```
>$ ./a.out the badFile.txt
myGrep: Unable to open file: badFile.txt
```

- If there aren't exactly three command line arguments:

```
>$ ./a.out
myGrep: Improper number of arguments
Usage: ./a.out <word> <filename>
```

Part 4: Extra Credit

Notice that, in the above output, the word ("the") occurs twice on line 7 ("entered the university. He cared little for the"), and `myGrep.c` prints out a separate line for each occurrence.

Develop a second version of your program that outputs a single line for multiple occurrences of the word within the same line:

Sample Run:

```
>$ ./a.out the sample.txt
./a.out the sample.txt
Longest line (50 characters): Polyana, in the province of Tula. His mother died
Number of lines: 17
Total occurrences of "the": 7
Line 1, word(s) 4: August 28, 1828, at the family estate of Yasnaya
Line 2, word(s) 2: Polyana, in the province of Tula. His mother died
Line 4, word(s) 2: Placed in the care of his aunts, he passed many
Line 7, word(s) 1, 7: entered the university. He cared little for the
Line 11, word(s) 4: English, and Russian novels, the New Testament,
Line 12, word(s) 6: Voltaire, and Hegel. The author exercising the
```

(Assuming your executable is named "`a.out`". Italicized characters are typed by the user. Your program will be tested using `diff`, so your output must match *exactly*.)

Requirements:

- Write your program in files named "`myGrep2.c`", "`wordList2.h`", and "`wordList2.c`".
- You may decide which functions and data structures to implement and how to implement them.
- You must store each occurrence of the word using linked lists.
  - Store the line in which the words occur.
  - Store a list of the positions of the word within that line. You must use a linked list for this as well — that is, your data structure for storing occurrences *must be a linked list of linked lists.*
  - Store the number of that line within the file.
  - Both the lines in the file and the words in a line are numbered from 0.
- Before it begins its search, your program must verify and echo its command line arguments.
- Your string comparisons may be case-sensitive.
- You may assume that no line (or word) will be longer than 100 characters.
- You may *not* make any assumptions about how many times the word occurs in a line or in the file.
- You must read through the file exactly once.
- You must free any memory that you dynamically allocate.
  - Note that the nodes of the inner lists must be individually freed, in addition to freeing the nodes of the outer list.