CS/240/Lab/4

In this lab, we write a word frequency counter to discover the most popular topics on Twitter. You will create a library `liblist.a` and program `trends`.

Words in this lab are, as before, sequences of alphabetical characters. For instance, `"1this is_a@test?"` contains the words `"this"`, `"is"`, `"a"`, `"test"`.

You may not use `libwords_ref.a` or `libutils_ref.a`, but you are free to use your own implementations of the `words` and `utils` functions (just copy them to `list.c`). You are also free to use any functions in the standard C library.

On Piazza, you'll find: `list.h` (prototypes), `list.c` (template for you to fill in), `liblist_ref.a` (reference implementation of `liblist.a`), `trends_ref.o` (reference implementation of `trends.c`), `get_timeline` (a program to get Twitter timelines). In addition, the following files are still available from previous labs: `libutils_ref.a`, `libwords_ref.a`, `parser_ref.o`. You may use them to build the parser to format input for the `trends` program, but may *not* link them with `trends`.

Q1: Word list

Before we can start counting word trends, you'll need to be able to store all those words. In this lab, you will store the words in a list. You will implement a data type `struct lnode`, which is a node in a doubly-linked list. Each node contains a word, along with an integer which stores the number of times the word has been seen (the count), and another integer which stores the last line the word was seen on. How you implement your `struct lnode` is up to you. The head and tail pointers (`struct lnode *head`, `struct lnode *tail`) are variables of the *caller*; as such, the functions you implement will take a pointer to the head and tail pointers (`struct lnode **head`, `struct lnode **tail`). `*head` and `*tail` will initially be `NULL`. All strings passed to or returned from these functions *must* be `'\0'`-terminated. You will implement the following functions:

```
struct lnode *newNode(struct lnode **head, struct lnode **tail, char *word, int line)
```

Description: In the linked list with `*head` and `*tail` as head and tail pointers, add a newly-created node to the tail of the list. The new node has the given word and line, and a count of `1`. If the list is empty (`*head` and `*tail` are both `NULL`), then both `*head` and `*tail` should be set to point to the new node. In any case, `*tail` should be set to the new node before the function returns. You must duplicate `word`, as the original `word` may be modified by the caller. Return the newly-created node.

```
void deleteNode(struct lnode **head, struct lnode **tail, struct lnode *node)
```

Description: Removes the specified node from the list, and `free`s all memory the node is using. Remember that if `*head` or `*tail` are the node, they will need to be updated!

```
struct lnode *nodeGetNext(struct lnode *node)
```

Description: Simply returns the next node in the list, or `NULL` if there are no further nodes.

`char *nodeGetWord(struct lnode *node)`

Description: Simply returns the word in the given node.

`int nodeGetLine(struct lnode *node)`

Description: Simply returns the line in the given node.

`int nodeGetCount(struct lnode *node)`

Description: Simply returns the count in the given node.

`int nodeIncCount(struct lnode *node)`

Description: Increments the count in the given node, and returns the new count.

`struct lnode *getNode(struct lnode **head, struct lnode **tail, char *word, int line)`

Description: If the linked list with `*head` and `*tail` as head and tail pointers contains a node with the specified word, that node is returned, with its line updated to the specified line. Otherwise, returns `NULL`.

`void deleteList(struct lnode **head, struct lnode **tail)`

Description: Deletes *every* node in the list with `*head` and `*tail` as head and tail pointers. All memory used by the list should be `free`d, and `*head` and `*tail` should both be `NULL`.

Compiling & Testing

The reference implementation `trends_ref.o` is used to build `trends`. But before putting everything together, unit test first and make sure your `list.c` satisfies the autograder. To create liblist.a do:

```
gcc -std=c99 -c list.c; ar rcu liblist.a list.o
```

The command to link your `liblist.a` with our reference `trends_ref.o` is:

```
gcc -std=c99 -o trends trends_ref.o -L. -llist
```

Note the `.` after `-L` and that there are two `l`s in `-llist`. To test with messages in `msg.txt`, use

```
cat msg.txt | ./trends
```

The command to link your `liblist.a` with your own unit testing suite in `test.c` is:

```
gcc -std=c99 -o test test.c -L. -llist
```

The command to link our reference `liblist_ref.a` with our reference `trends_ref.o` is:

```
gcc -std=c99 -o trends trends_ref.o -L. -llist_ref
```

To determine the trends in a file full of tweets, using the parser we built in the previous lab, do:

```
cat msg.txt | ./parser the of and a to in is you that it | ./trends
```

If the parser from the previous lab is not sufficient, you may create a parser from only our reference parts:

```
gcc -std=c99 -o parser parser_ref.o -L. -lwords_ref -lutils_ref
```

Q2: Trends

Your `trends` program will read in a series of tweets and count the frequency of each unique word in the series. Once the end of the series is reached, it will print out a sequence of words and how often it saw them, in no particular order. It will keep these words in the word list you implemented in Q1. Words with no repetitions should only be kept for 100 lines. To be precise, if you start on line 1, then when you read line 101 (but before processing its words), every word in your word list with line 1 and count 1 should be removed. Words with count greater than 1 should always be kept. Once the end of the series is reached, every word with count greater than 1 will be printed, along with its count. The output should contain each word, followed by a single space, then the count. Each word/count pair should be on a separate line, e.g.:

```
the 1593
python 147
ruby 234
haskell 67
```

We are not providing a template for `trends.c`. It should work with both our reference `liblist_ref.a` and your own `liblist.a`. You may use any function in the C standard library.

Compiling & Testing

The command to link `trends` using *your* `liblist.a` is:

```
gcc -std=c99 -o trends trends.c -L. -llist
```

The command to link `trends` using *our* `liblist_ref.a` is:

```
gcc -std=c99 -o trends trends.c -L. -llist_ref
```

Listening to Twitter

To see what topic is popular on Twitter, we have created accounts that follow people talking about programming languages. To get the list of all their recent tweets, we ask Twitter for the "timeline" for each account. We provide a tool that does this and dumps the text of all the tweets to standard output. This makes testing and debugging easier. Your trend calculation can also be tested using a file:

```
cat msg.txt | ./parser the of and a to in is you that it | ./trends
```

To make realistic test data, you can save the results of `get_timeline` to the file `msg.txt`, and use that:

```
./get_timeline | ./parser the of and a to in is you that it > msg.txt
cat msg.txt | ./trends
```

You can use `get_timeline` to track the lastest tweets directly:

```
./get_timeline | ./parser the of and a to in is you that it | ./trends
```

Turning in

Go to the submission web page, under "currently open projects" will be listed "lab 4 list" and "lab 4 trends". Click "lab 4 list", use the file selector to choose your `list.c` file, then click "submit". Click "return to list of projects", then "lab 4 trends". Use the file selector to choose your `trends.c` file, then click "submit".

Lab is due Monday, February 13th before midnight. No late labs accepted.

Grading criteria

List

- source file is named `list.c`
- code compiles
- code runs without error
- newNode must insert a new node with the specified value of word and line in the tail of the list. And the word string to store must be a copy of the passed one.
- getNode must return the pointer to the requested node if there is one in the list and the line field must be updated. getNode must return NULL otherwise.
- nodeGetNext, nodeGetWord, nodeGetCount, nodeGetLine, nodeIncCount must work as described.
- deleteNode must work on all cases when the node to delete is head, or tail, or in the middle of the list. You can assume the node to delete is always one on the list. All memory belongs to the node must be freed, which includes the word string and the node itself.
- deleteList: after called, the list must be empty, and all memory belongs to the list must be freed.

Trends

- source file is named `trends.c`
- code compiles
- code runs without error
- trends must print all words mentioned in the input and repeated within 100 lines and the correct frequency.
- trends must not print anything else.