Lab - Linked Lists

Provided Files

The files for this lab are located as usual in Mike's home directory:

```
/home/zastre/seng265/lab-08
```

Exercise A

The files in section A of the lab implement a string sorting algorithm using a linked list and dynamic memory. Read these files carefully; until you understand how this linked list operates, it will be difficult to complete subsequent sections of the lab.

The tester that is included with this section accepts a string as a command line argument, and outputs the string words in lexicographic order; that is, alphabetically but where uppercase letters come prior to lowercase letters. For example, running the tester with:

```
./tester "buy and call And CALL"
```

produces the output:

```
And
CALL
and
buy
call
```

You'll probably need to trace the code to follow it; a diagram will almost certainly be helpful. In particular, note the following:

- Each of the functions returns the head of the resulting list. Therefore, the expected use of each is something like:

```
node_t * list;
list = add_inorder(list, new_node(some_string));
```

- `node_t` is very lightweight - it only contains two pointers. Specifically, `node_t` does not contain the contents of a string - it only contains a *pointer* to a string on the heap, along with a pointer to the next `node_t` in the list.
- String data is copied with `strdup()`. Read the documentation for this function, and note that `strdup()` itself calls `malloc`! This means that there are *two* calls to `malloc` involved in creating each `node_t`.
- Take a look at the tester, and see how this program deals with the above - how many calls to `free()` are required per node?
- `add_inorder()` uses *two* `node_t` pointers internally as it searches for the proper place to insert the node_t called `new`.

Exercise B

The program `tester` in this section, when complete, will output a series of actor names and their birth years, ordered first by birth year, and then (when years match) lexicographically by last name, then first name. For example, the tester program adds the following data to the list:

```
    list = add_inorder(list, new_node("Bishop, John", 1966));
    list = add_inorder(list, new_node("Cusack, John", 1966));
    list = add_inorder(list, new_node("Rhys-Davies, John", 1944));
    list = add_inorder(list, new_node("Depp, Johnny", 1963));
    list = add_inorder(list, new_node("Wayne, John", 1907));
    list = add_inorder(list, new_node("Malkovich, John", 1953));
    list = add_inorder(list, new_node("Oliver, John", 1977));
    list = add_inorder(list, new_node("Higgins, John Michael", 1966));
    list = add_inorder(list, new_node("Glover, John", 1944));
```

When the program is complete, it will output:

```
Number of actors: 9
Wayne, John (born 1907)
Glover, John (born 1944)
Rhys-Davies, John (born 1944)
Malkovich, John (born 1953)
Depp, Johnny (born 1963)
Bishop, John (born 1966)
Cusack, John (born 1966)
Higgins, John Michael (born 1966)
Oliver, John (born 1977)
```

The code in this section is very similar to that from section A, with a few differences:

- In list.h, the definition of node_t has been modified to accept the new pieces of data to be stored in each node (`word` has been renamed to `name`, and there's a new `birth_year` field).
- In list.c, the function new_node() has been modified to accept the new data.
- The old function add_inorder() needs to be changed to accomodate our new, more complicated ordering logic.

Your task for part B is to modify the function add_inorder() to make the `tester` program produce the above output.

Debugging

As you complete part B, you'll need to confirm that your solution doesn't leak memory or commit other memory safety errors. Use `valgrind` for this purpose:

```
valgrind ./tester
```

Peruse the resulting report and ensure that your solution is free from warnings - no uninitialized memory access, no memory leaks and no violations resulting from multiple calls to `free()`.

You also may wish to use GDB to help you debug your solution. Since the makefile specifies the `-g` flag for GCC, debugging information is already available to GDB.

```
gdb ./tester
```

The following commands may help you debug your program (capital letters indicate values you need to substitute in, and the abbreviation for the command is in brackets at the end):

Command	Abbreviation	Meaning
run	r	run the program, stopping at breakpoints. Specify any command-line arguments after `run`
kill	k	stop the program from running
quit	q	stop gdb
list N	l N	print the program listing starting at line N
break N	b N	set a breakpoint at line N
continue	c	resume execution after a breakpoint is reached
step	s	run the next line of the program
next	n	run the next line, but does not enter functions
until FOO		run until line number or function name FOO
print VAR	p VAR	prints the current value of variable VAR
backtrace	bt	show the function call stack
up		move up one element in the call stack
down		move down one element in the call stack

For example, a stack overflow can be diagnosed with the `bt` command - if a function is recursively calling itself over and over, its name will be printed repeatedly. A pointers value can be printed - if that value is `0x0`, the pointer is `NULL` and should never be dereferenced.

GDB is a very sophisticated and complex program, and more powerful debugging supports are available; consult the documentation for more detail.

Exercise C & D (optional)

The remaining exercises are optional - they deal with technicalies involving function pointers. We've seen function pointers throughout the course in the context of `qsort()`, and they were also used in the `apply()` function in Exercises A and B.

Exercise C uses a comparator function to sort randomly generated numbers, which exercise D uses a similar function to sort strings. In both cases, the function takes a pair of `void *` pointers, and returns an integer. The trick with writing a comparator function is to figure out how to provide what qsort expects:

- a negative number, if the first 'thing' is less than the second 'thing'.
- a positive number, if the first 'thing' is greater than the second 'thing'.
- zero, if the two 'things' are equal.

What it means for one 'thing' to be before or less than another 'thing' depends on what kinds of things these pointers are pointing to - for real numbers this is fairly intuitive, but for other types it might be less clear. For example, should 'A' come before 'a'? The answer to this (which is a little arbitrary) is the basis of the lexicographic ordering we saw in the first two exercises.

Exercise E (optional)

Complete the comparator function `compare_two_doubles()` in `tester.c`. Note that as in the previous two sections, the function doesn't receive the values to compare directly. Instead, it receives two `void *` pointers. You'll need to figure out how to access the underlying values through these pointers.

Attendance

Make sure your TA records your attendance for the lab.