Project 6

Due July 18th at 11:59 PM

In project 5 you used a statically allocated array to hold your employee structs. Reimplement project 5 so that it uses a dynamically allocated linked list instead of an array. Each node in the list is an employee struct and should be dynamically allocated. **The same employee struct from project 5 can be reused, but with a single added member 'next' that is a pointer to the next employee in the list**.

In addition, you will implement an "Employee Bonus" feature which will allow you to choose one employee to give a bonus too. The employee name and the bonus amount to give should be entered as command line arguments. The bonus amount should be added directly to their calculated net income. If you enter an employee name which does not exist, no one gets the bonus.

Do not sort the list. If you implemented the extra credit from project 5, remove the call to selection_sort() so that the output is not sorted.

All other previous functionality of project 5 should carry over to project 6.

An example execution of the program:

```
programName employees.txt payday.txt rippetoe 100.00
```

Details

1. You must implement the following new functions
    a. `append`: add an employee struct to the linked list
        i. Allocate memory for the employee
        ii. If the list is empty, `append` should return a pointer to the newly allocated employee struct
        iii. add the employee to the end of the list and return a pointer to the first item in the list

    b. `search`: find an employee by name and return a pointer to their struct. If an employee with the given name doesn't exist return NULL.

    c. `clearList`: **Deallocates all employee structs in the linked list passed in. This functions should be called when the user exists the program, so that all the memory allocated for the linked list is freed.**

2. **These new linked list related functions should be implemented in `linkedList.c` and declared in `linkedList.h`**

3. `ClearList` should be called at the end of the program to deallocated all memory used by the linked list

4. You must update your makefile from project 6 to include the new files added

Suggestions

1. You can use a diff tool to compare the file output from your program to the expected, correct output. There is a `diff` command available on linux that will show you which lines in your file are different

```
diff file1 file2
```

   will show you the lines which are different between file1 and file2. If they are identical, the command will appear to have done nothing won't do anything.

2. START ASAP AND ASK QUESTIONS.

Before you submit:

1. **Make sure you follow the rules for the input/output files**

2. Make sure your program begins with a comment that begins with your name

3. Your makefile should compile with `c99` and the `-Wall` flag. `-Wall` shows the warnings by the compiler. Be sure it compiles on the student cluster with no errors and no warnings.

4. Test your program thoroughly with multiple different input files. Use the diff command to compare your output with the expected output

5. Place all of your source files in a zip file. The files should be in the root of the zip; not in a folder inside the zip. You can zip them on the student cluster with the following command

```
zip project6 *.c *.h makefile
```

   This will place the makefile and all of the .c and .h files in your current directory into project6.zip. Submit your zip file.

Grading Details

1. A program that does not compile may result in a zero.
2. Programs which generate compiler warnings will lose 5%
3. Commenting and style worth 10%
4. Functionality worth 90%

Programming Style Guidelines

The major purpose of programming style guidelines is to make programs easy to read and understand. Good programming style helps make it possible for a person knowledgeable in the application area to quickly read a program and understand how it works.

1. Your program should begin with a comment that briefly summarizes what it does. This comment should also include your name.
2. Variable names and function names should be sufficiently descriptive that a knowledgeable reader can easily understand what the variable means and what the function does. If this is not possible, comments should be added to make the meaning clear.
3. Use consistent indentation to emphasize block structure.
4. Macro definitions (`#define`) should be used for defining symbolic names for numeric constants. For example: `#define PI 3.141592`
5. Use underscores or camel case notation to make compound names easier to read: `total_volume` or `totalVolume` is clearer than `totalvolume`.