CS 112 - Homework 8

Deadline

11:59 pm on **Friday, November 4**

Purpose

To answer questions related to vectors and linked lists, to practice more with vectors, and to write two more linked list functions.

How to submit

You will complete **Problems 1 and 2** on the course Canvas site (short-answer questions on vectors and linked lists).

For **Problems 3 onward**, you will create the specified `.cpp`, `.h`, and `.txt` files on the CS50 IDE, and then submit those to the course Canvas site.

NOTE: While I list the files you need to submit for each problem below, I have set up Canvas to *also* accept `.zip` files.

That is,

- you can submit each `.cpp`, `.h`, and `.txt` file to Canvas.
- OR, if you prefer, you may compress your files to be submitted into a single `.zip` file and submit that `.zip` file to Canvas.

Problem 1 - 9 points

Problem 1 is correctly answering the "HW 8 - Problem 1 - Short-answer questions on vectors" on the course Canvas site.

Problem 2 - 12 points

Problem 2 is correctly answering the "HW 8 - Problem 2 - Short-answer questions related to linked lists" on the course Canvas site.

Setup for Problems 3 - onward

- FIRST: in the CS50 IDE, in your folder for this homework create copies of the following:
  - `Node.h` and `Node.cpp` from Week 10 - Lecture 1 or 2 (they should be identical)
  - the versions of `linked-list-functs.h`, `linked-list-functs.cpp`, and `linked-list-tests.cpp` from Week 10 - Lecture 2.
- At the beginning of **each** of `linked-list-functs.h`, `linked-list-functs.cpp`, and `linked-list-tests.cpp`, in their opening comment block:
  - add `and` *Your Name* to the end of the `by:` line
  - for `last modified:`, add a new FIRST line (moving the existing original date to the next line) listing your Homework 8's last-modified date and the functions you are implementing for Homework 8.

Problem 3 - play a bit more with vectors!

As some additional vector practice, **choose ONE of the suggestions below** for this problem. (You are

encouraged to try both if you like, but I will only grade one of them, in the interests of time.)

Problem 3 - Option 1 - `main` function in file `words_vector.cpp`

Write a `main` function in a file `words_vector.cpp` that does the following:

- It asks the user for the name of a file assumed to contain JUST words, and it tries to open that file for reading.
- It declares an empty vector able to hold strings, and reads all of the words from that file into that vector.
- Print to the screen how many words were read into the vector, and then print the vector's contents, one word per line.
- Now that you have that vector of strings, do *something* of your choice with them -- for example:
  - compute the average length of the words in that vector, and print that average to the screen
  - determine the shortest word length and the longest word length, and print those to the screen
  - ask the user to guess a word, and then tell them if their guess is in the vector
  - (or some action(s) where you have to do something with each word in the vector)

Submit your file `words_vector.cpp` and at least two `.txt` files (each with a different number of words) that you used for trying out your program. (If your choice of additional actions involves calling other function(s), include their `.cpp` and `.h` files, also.)

Problem 3 - Option 2 - function `read_words_vector`

Write the following vector-based variation of Homework 4 - Problem 4's `read_words` function:

Function `read_words_vector` expects just two arguments: a desired file name and an empty **pass-by-reference** vector, able to hold `strings`, and has the side-effects of:

- trying to open that file for reading
- reading in the words in the file and storing them into the passed vector (thus actually changing the argument vector)

...and that returns, in this case, the number of words read into the argument vector.

So, **notice** that the file, in this case, is *not* assumed to start with the number of words in the file; it is assumed to just contain words. And, `read_words_vector` does not need to have the size of the passed vector as a parameter.

Submit your files `read_words_vector.cpp`, `read_words_vector.h`, `read_words_vector_test.cpp`, and the `.txt` files used in testing `read_words_vector` in `read_words_vector_test.cpp`.

Optional extensions:

- Write this so that it works with non-empty as well as empty vector arguments -- that is, if the argument vector is not empty, it first overwrites any current words with those it reads, and then adds to the end of the vector as needed.
- As you are reading words from the file, only write them to the argument vector if they are of length 5.
- As you are reading words from the file, only write them to the argument vector if they are of length 5 and only contain letters.
- As you are reading words from the file, only write them to the argument vector if they are not already in

the argument vector.

Problem 4 - function `get_size`

If we had a `List` class implemented using a linked list, it would be good for it to have a size data field, as that would be convenient (and the methods could reasonably maintain that). But we just have a collection of linked list functions right now, so a function getting the size of a linked list would be useful.

- Add a function `get_size` to `linked-list-functs.cpp` and `linked-list-functs.h`.
- `get_size` should expect a pointer to the beginning of a linked list of `Node` instances, and return the size of that list (that is, the number of nodes in that list).
- In `linked-list-tests.cpp`, add at least the following after the `cout` printing that you are testing `get_size`:
  - Print to the screen the result of comparing the value of a call to `get_size` on an empty list to what it should return.
  - Use function `insert_at_front` to create a linked list of at least 5 items.
  - Print to the screen the result of comparing the value of a call to `get_size` on that list of at least 5 items to what it should return.
  - Call `delete_list` to free/deallocate the memory for your linked list of at least 5 items. (Note: you can shift this call to `delete_list` *after* your other function's tests below if you'd like to use your list for testing it, also.)

Problem 5 - function `sum_list`

Add a function `sum_list` to `linked-list-functs.cpp` and `linked-list-functs.h`.

- `sum_list` should expect a pointer to the beginning of a linked list of `Node` instances, and return the sum of the values in the `data` fields of that linked list. It should return a sum of `0` if the linked list is empty.
  - Use a return type of `NodeDataType` for `sum_list` -- and note that this is a function that will likely need modification if `NodeDataType` is ever changed to a type that does not have + as an operator! 8-)
- In `linked-list-tests.cpp`, add at least the following:
  - Print to the screen a message saying that you are testing `sum_list`.
  - Print to the screen the result of comparing the value of a call to `sum_list` on an empty list to what it should return.
  - Use function `insert_at_front` to create a linked list of at least 5 items (or use your at-least-5-items linked list from testing `get_size`).
  - Print to the screen the result of comparing the value of a call to `sum_list` on that list of at least 5 items to what it should return.
  - Call `delete_list` to free/deallocate the memory for your linked list of at least 5 items.

Submit your resulting files `linked-list-functs.h`, `linked-list-functs.cpp`, and `linked-list-tests.cpp`.