SENG1050 – DATA STRUCTURES

MAJOR ASSIGNMENT 1 - SORTED DOUBLY-LINKED LISTS

OVERVIEW

- Write a program, based on Focused Assignment 2, that keeps track of fares for flights and displays information about them.

OBJECTIVES

- Create and use sorted doubly-linked lists.
- Use common algorithms to enhance software development.
- Use best practices to effectively produce quality software.

ACADEMIC INTEGRITY AND LATE PENALTIES

- Link to Academic Integrity Information
- Link to Late Policy

EVALUATION

- The evaluation of this assignment will be done as detailed in the Marking lecture in Week 2 of the C course.

PREPARATION

- Understand how singly-linked lists work.
- Review the linked list examples.

REQUIREMENTS

Changed Requirements

- Along with the destination and date for each flight, also get a floating-point number from the user that represents the fare for the flight. The order of input for each flight must be destination, date, and fare, all on separate lines.
    - You can assume that the destination and date are least one and less than 30 characters in length.
    - All fares entered by the user will be valid.

    - The fare must be stored as a floating-point number (e.g. double or float).
    - No other input is allowed.
- Put the input into a sorted doubly-linked list (instead of the unsorted singly-linked list from Focused Assignment 2), where the sorting is done by ascending by fare (e.g. 100.59 comes before 400.59). If two flights have the same fare, put the new entry after the previously-entered entry.
    - The sorting must be done such that each element is inserted into the proper location in the list in turn. Do **not** create an unsorted list and then sort the list afterwards (doing this would result in a mark of 0 and no further feedback for the assignment).
    - At the same time, put the flight into another separate sorted linked list, where the sorting is done ascending by destination (e.g. Mexico City comes before Miami). If two flights have the same destination, put the new entry after the previously-entered entry.
    - The user will never enter a flight with a destination and date that is already in the list (e.g. if "Oct. 12", with a destination "Miami" is in the list already, the user will not ask you to enter it again).
    - This means that you will have two separate head pointer variables and two separate tail pointer variables.
- Stop getting user input immediately once an invalid destination of "." or an invalid date of "." is entered by the user.
    - You should ignore the flight that had the invalid destination or date (i.e. don't put it into the linked list).
- After the input is complete, traverse the fares-sorted linked list, displaying one flight per line with destination (left-justified with a width of 35 characters), date (left-justified with a width of 35 characters), and fare (left-justified).
    - Do not double-space the output.
- Next, traverse the destination-sorted linked list, displaying with the same output format as the previous traversal.
    - Do not double-space the output.
- Next, obtain one flight destination and date pair (destination before date on two separate lines) as additional user input. Search for that pairing in the destination-sorted linked list.
    - If there is a match of destination and date, display the fare and prompt for a new fare.
        - If the fare is changed, update the destination-sorted linked list with the new fare. Then delete the flight from the fare-sorted linked list and re-insert it.

        - If the fare is unchanged, print a message indicating that and do nothing.
        - Again, the fare will always be valid.
    - If there is not an exact match (on the destination and date fields), display a message saying so and continue with the next step.
- Redisplay both linked lists as before.
- Once completely done, you must free **all** allocated memory. Yes, I do mean **all.**

ADDITIONAL FUNCTIONS

- Create the findFlight() function. It returns NULL if a flight is not found or it returns a pointer to the node containing a flight, if both the destination and date are matched (with a case-sensitive match, so you can use strcmp for this). It takes three parameters:
        - FlightNode *head: head of list
        - char *destination: pointer to null-terminated string containing destination
        - char *date: pointer to null-terminated string containing date
    - If only the destination or the date are found but both are not found in the same node, the flight is not found.
    - You must call findFlight() whenever you need to find a flight in a list.
- Create the deleteNode() function. It deletes a node, using three parameters:
        - FlightNode *node: node to delete
        - FlightNode **head: pointer to head of list
        - FlightNode **tail: pointer to tail of list
    - The key to this function is relinking pointers around the node before deleting.
    - If node is NULL, it returns immediately.
    - It returns nothing. The assumption is that the node is valid.
    - You must call deleteNode() whenever you need to delete a node in a list.

OTHER REQUIREMENTS

- The destination and date fields must be dynamically allocated to an appropriate size as in Focused Assignment 1.
- Do not get user input except as indicated in these requirements.
    - As previously stated, you can assume that all input is valid (i.e. I won't try to enter a fare of "fred" or a destination of "This string is much much much much much too long") or is intended to quit getting input.
    - I absolutely will be testing with destinations that have multiple words in them (e.g. "New Orleans").
- Do not display output except as indicated in these requirements.
- Do not clear the screen at any time in this program.
- It must not use global variables.
- It must not use goto.

- Be aware that you still cannot use C++ strings in this assignment because malloc() does not support them.
- The SET Coding Standards must be adhered to.

GIT REQUIREMENTS

- Use GitHub Classroom for revision control, similar to Focused Assignment 2. It is expected that you have a reasonable number of commits with meaningfully descriptive commit comments.

CHECKLIST REQUIREMENTS

- Create a requirements checklist. This should contain the specific requirements from this assignment as well as any relevant requirements that have been covered in lecture or that are found in the SET Coding Standards or SET Submission Standards. Do it in whatever form you wish. Hand in your completed checklist in PDF form as checklist.pdf. Not having this checklist will result in a cap of 80 on your mark.

FILE NAMING REQUIREMENTS

- You must call your source file m1.cpp.
- You must call your checklist checklist.pdf.

SUBMISSION REQUIREMENTS

- Do not hand in any other files.
- Submit your files to the *DS: Major Assignment 1* Assignment Submission Folder.
- Once you have submitted your file, make sure that you've received the eConestoga e-mail confirming your submission. Do not submit that e-mail (simply keep it for your own records until you get your mark).