UNIVERSITY OF CALIFORNIA, SANTA CRUZ
BOARD OF STUDIES IN COMPUTER ENGINEERING

CMPE-13/L: COMPUTER SYSTEMS AND "C" PROGRAMMING

Lab 5 - Linked Lists

Commit ID Form: https://goo.gl/forms/l8pOiHGzyZEQawCs1

17 Points

Introduction

This lab introduces the concept of pointers, both to data and functions, through the concept of a doubly-linked list data structure. This is a very common structure that is used in many programs for storing expandable lists of ordered data. You will also be given a function to initialize a list of unsorted data. You will need to sort and create an algorithm to count reoccurrences of data stored in the items.

You will be implementing a LinkedList library that can store, sort, and print strings. Once that library is complete you will use an existing linked list that we have created for you to write code that sorts a linked list containing words and then implement a word-count function that operates on a sorted list. I have provided code that creates a list and does a word count on it, but it assumes that the list is unordered. You will need to write another word counting algorithm that is faster than the one provided to you (which is possible as you know the list is already sorted).

Reading

- **K&R** – Chapters 5, 6.7, 7.8.5, appendix B5

Concepts

- Doubly-linked list
- Memory allocation
- Sorting
- Pointers (including NULL)
- Algorithmic analysis

Provided files

- sort.c – contains:
    - `main()` – which contains some "Starter Code" that initializes an unsorted list of words, prints it out, and counts reoccurrences of words. You are to complete main() to sort that list, print out the sorted version

and then design and implement an algorithm to count the occurrences of words that is more efficient than the unsorted algorithm (remember that this algorithm has the knowledge that the list is presorted before it is given).

    - The functions used to implement the starter code are given below `main()` with comments to document their functionality. There is also a specified place to define a `SortedWordCount()` function.
- LinkedList.h – provides #define constants for return values along with the function prototypes that you will be implementing. Comments above the function prototypes describe their functionality. Do not copy this whole file when creating LinkedList.c, just copy the individual function prototypes. **Do not modify this file!**[1]
- LinkedListTest.c – provides a `main()` that is used for unit testing the LinkedList functions that you have coded. Just like in previous labs, you will need to code your own test harness, and exclude it from the project when done testing.
- BOARD.c/h - Contains initialization code for the UNO32 along with standard #defines and system libraries used. Also includes the standard fixed-width datatypes and error return values. **You will not be modifying this file at all!**

Assignment requirements

- Your program will implement the functions whose prototypes are provided in LinkedList.h. The functions all have appropriate documentation and describe the required functionality.
    - While not required, it's suggested that you create a helper function, CompareStrings(), that takes in two ListItem pointers and returns -1, 1, or the result of strcmp() on their data pointers. This will make implementing LinkedListSort() easier. If you do this, remember that its prototype, definition, and all uses of it sure exist solely within LinkedList.c.
- Within LinkedListTest.c you will:
    - Create a test harness that tests your linked list functions and ensures that they return the correct values. As in previous labs, at least two tests are required per function.
    - Once you are done testing the functionality of LinkedList.c, you will need to exclude LinkedListTest.c from the project.
- Within sort.c you will:

[1] **We do NOT use your LinkedList.h file when grading this lab. If you have changed it and your code does not compile, we do not attempt to fix it, and you will get a 0.**

    - Create a new algorithm for counting occurrences of words within a function named `SortedWordCount()` (whose definition follows from `UnsortedWordCount()`) that uses the knowledge that it's processing a presorted word list to be more efficient than `UnsortedWordCount()`. You should first read and understand how `UnsortedWordCount()` works before attempting to write `SortedWordCount()` so that you can be sure that yours is faster. To think of how fast an algorithm is, just consider how many times it accesses each element in the list.
        - Be sure to check that your word count handles the empty string and no-string (NULL) cases properly!
    - Print out the data array along with the array of occurrences found using your new `SortedWordCount()` function. The complete output when the program is run should be two sets of the array printed along with its word count. Check the Expected Output section of this lab.
    - Once the necessary output is displayed, you will free up all of the memory used by the linked list. You SHOULD NOT free the data contained within each linked list item. BE CAREFUL HERE: it is not safe to try to use data after it has been `free()`'d and you will not get the points if you do.
- Add inline comments to explain your code.[2]
- Create a readme file named README.txt containing the following items. Spelling and grammar count as part of your grade so you'll want to proof-read this before submitting. This will follow the same rough outline as a lab report for a regular science class. It should be on the order of three paragraphs with several sentences in each paragraph.
    - First you should list your name & the names of colleagues who you have collaborated with.[3]
    - In the next section you should provide a summary of the lab in your own words. Highlight what you thought were the important aspects of the lab. If these differ from how the lab manual presents things, make a note of that.
    - The following section should describe your approach to the lab. What was your general approach to the lab? Did you read the manual first or what were your first steps? What went wrong as you worked through it? What worked well? How would you approach this lab differently if

[2] about 1 comment every 6 lines of code is probably right

[3] NOTE: collaboration != copying. If you worked with someone else, be DETAILED in your description of what you did together. If you get flagged by the code checker, this is what will save you.

you were to do it again? How did you work with other students in the class and what did you find helpful/unhelpful?
    - The final section should describe the results of you implementing the lab. How did it end up finally? How many hours did you end up spending on it? What'd you like about it? What did you dislike? Was this a worthwhile lab? Do you have any suggestions for altering it to make it better? What were the hardest parts of it? Did the points distribution for the grading seem appropriate? Did the lab manual cover the material in enough detail to start you off? Did examples or discussions during class help your understand this lab or would more teaching on the concepts in this lab help?
- Make sure that your code triggers no errors or warnings when compiling as they will result in a significant loss of points.
- Follow the style guidelines.
- **Required Files**
    - LinkedList.c
    - LinkedListTest.c
    - sort.c
    - README.txt

Grading

This assignment consists of 17 points:

- 6 points – Properly implemented doubly-linked list
- 3 points – Property implemented test functions for linkedlist
- 6 points – Proper sorting and printing functionality
- 1 point - Provided a readme file
- 1 point – Followed proper GIT hygiene
- 1 point – Follows style guidelines (<= 5 errors in all code) and has appropriate commenting

You will lose points for the following:

- NO CREDIT for sections where required files don't compile
- -2: Any compilation warnings.
- -2: Used extern or goto

Pointers

Pointers are covered very thoroughly in the required reading for this lab, so if you are having trouble, refer back to chapter 5 of K&R. However, there is one concept about pointers that is not directly addressed in the reading: null pointers. A null pointer is a pointer to nothing. These pointers must also be handled as a special case if they are passed to a function that expects non-null data pointers. This is one of the major sources of crashes in programs.

The main problem with null pointers arises from when you try to dereference it (assuming x is an int pointer and equal to `NULL`): `*x = 6;`

The reason for why becomes obvious when you think about what memory location `x` points to. 0, or NULL, is an invalid memory location, and a "null pointer dereference" error occurs because there is no memory location to write to, so an error occurs. This is a "fatal" error, which means that the program has no way to handle it, so the only thing it can do is crash! This is a common cause of Windows blue-screen-of-death errors. The solution to this is to check for null pointers before dereferencing. An especially important case for checking to see if a pointer is null is after any call to `malloc()` or `calloc()`, which we will cover a little later.

Doubly-linked lists

In computer programs, much as in real life, keeping a list of things can be useful. Usually the number of items that will be in this list is known ahead of time and so in a computer program this list could be kept in a standard C array. There will be occasions, like when processing user input, when the number of items to be stored in a list is not known ahead of time. This is a problem with C's statically-allocated arrays. The common solution is to use another data type called a linked list.

Linked lists are exactly what they sound like: a collection of objects that are all linked together to form a single list. As each item is linked to at least one other item in the list there is a set ordering to the list: from a "head" item at the start to a "tail" item at the end. Since these items are all connected it is easy to access any item from any other item by just traversing or "walking" through the list.

For this lab you will be implementing a doubly-linked list, the more useful sibling of the linked lists. A doubly-linked list is also straightforward: each item is linked to both the item before it and after it. This allows for traversal of the list from any element to any other element by walking along it, which makes using the list very easy.

The items in the list you are implementing are stored as `structs` in C because they will be storing a few different pieces of data. Specifically it holds a pointer to the previous `ListItem`, which will be `NULL` if it's the head of the list; a pointer to the next `ListItem`, which will be `NULL` if it's at the end of the list; and a pointer to any kind of data (`NULL` if there's no data). The `typedef` and the name after the "`}`" let you refer to the `struct` in

a similar fashion to any other data type, by using the single name "`ListItem`" instead of the longer "`struct ListItem`".
The definition of the ListItem struct in LinkedList.h:

```
typedef struct ListItem {
          struct ListItem *previousItem;
          struct ListItem *nextItem;
          char *data;
} ListItem;
```

Now that you understand the structure of a linked list we will introduce the various operations that can be performed upon a list. The standard operations are creating a new list, adding elements to a list, finding the head of a list, and removing elements from a list.[4]

Creating a new list: A new list is created by just making a single ListItem. As this ListItem is both the head and tail of the list there is no item before it or after it in the list.

Adding to a list: Now that you have a list, how do you add more elements to it? With the arrays that you are familiar with, you need to know two things: the position to insert into and the data that will be inserted. With linked lists it's a little different because there's never a "free spot" to insert a new item into. What is done instead is that the position of the new list item is relative to an existing item, generally the item before it in the list. So to insert an item into the list, that item is inserted after an existing item. If the list went A <-> B <-> C and you want to insert D after B then the list would become A <-> B <-> D <-> C. So that means that the previous item and next item pointers of both B and C will need to change to accommodate the new item D.

Finding the head: The head of a list is a special item because it has no preceding element (represented by a `NULL` pointer). Since all the elements in a list are connected, finding the head merely requires traversing the list until a list item is found with no preceding element. A function that finds the head of the list has one odd scenario; see if you can figure out what it is.

Removing an element: Removing an element from a list is the opposite of adding to it. Following the example above you'd go from a list like A <-> B <-> D <-> C to A <-> B <->

[4] It is incredibly useful to your understanding to draw this out on a piece of paper or a white board. Make boxes for each member of the struct, and use arrows to point to the next list element and the previous ones (in other words, use arrows to show what the pointers point to). Go through all of the functions and make sure you understand what you need to do. Once you understand it conceptually, coding it up is very simple.

C. The pointers of B and C both need to be modified to account for the removal of D. Generally the data that was stored within D is also desired after the removal of the item and should be returned.

Selection sort

Sorting is an incredibly important function in computer programming. While you may not think it is used a lot, it is quite common within a program to have the need to sort a series of numbers. Sorting is an entire field of study within computer science and so there are a huge number of algorithms that do just that. Selection sort is the one you will be focusing on in this lab. It is very slow on average, but easy to understand and implement. It is called an in-place sorting algorithm, because it doesn't need to use a temporary array to store data. Pseudo-code for selection sort is provided below:

```
for i = 0 to length(A) – 2 do
   for j = i + 1 to length(A) - 1 do
        if A[j] < A[i] do
             swap A[j] and A[i]
        end if
   end for
end for
```

Selection sort is best understood by thinking of it as building a sorted list on the left by choosing the minimum value from the original unsorted list on the right as the next sorted value. The outer for loop effectively tracks the right-most element of the sorted array filling up the left portion of the array. This means that for each iteration of the outer-loop, the inner-loop can perform many element swaps. An example is shown below. The left-hand column holds the array at the start of an outer-loop iteration with the bold items comprising the already-sorted elements. Each swap within the inner-loop is shown in the right-hand column with the bold items having been swapped. The bold items on the left-hand side indicate the now-sorted portion of the array.

Outer loop array	**Inner-loop swaps**
64 25 12 22 11	**25 64** 12 22 11 **12** 64 **25** 22 11 **11** 64 25 22 **12**
11 64 25 22 12	11 **25 64** 22 12 11 **22** 64 **25** 12 11 **12** 64 25 **22**
11 12 64 25 22	11 12 **25 64** 22 11 12 **22** 64 **25**
11 12 22 64 25	11 12 22 **25 64**

11 12 22 25 64	-

Note: the algorithm you will be implementing arranges the items in descending order!

`malloc()`, `calloc()`, and `free()`

This lab also relies on the use of memory allocation using `malloc()` (and/or `calloc()`) and `free()`. These are discussed somewhat in chapter 5 of **K&R**. As they are standard library functions they are documented thoroughly online or in the Linux man-pages. Refer to those resources to understand them.

It should be emphasized here that after any call to `malloc()` or `calloc()` you should *always* check for NULL pointers! Memory allocation relies on the heap, which the PIC32 doesn't have by default. You will need to specify a heap size for your project of at least a couple hundred bytes for `malloc()` and `calloc()` to work.[5]

Note that this makes it easy to test that your code is properly checking for NULL pointers: if you set the heap to 0, ALL calls to `malloc()/calloc()` will fail; if your code doesn't crash, it's working!

SortedWordCount()

Once you have your library complete, you will be implementing a word counting algorithm within SortedWordCount(). In that function you will store the number of occurrences of each word in an integer array that is also passed into the function. Since the input this function is assumed to be a sorted list, you only need two loops: One to iterate through every item in the list and the other to search for the same word after the first occurrence of it. It will end up looking like the pseudo-code provided below.

```
int SortedWordCount(ListItem *list, int *wordCount):
     if list is not the head:
          error out
     for every item in list:
          if item is NULL:
               set word count to 0 for this item and continue
          else:
               for every newItem after this one:
                    if newItem is the same as item:
                         increment a counter
               save the word count for this item
               for every newItem after this one:
                    if newItem is the same as item:
                         save the negative of the word count for this newItem
```

[5] File -> Project Properties -> xc32-ldd -> Heap size (bytes).

Program overview

Now your code within main() in sort.c will use SortedWorkCount() will perform the following steps (following closely from the code provided):

1. Call InitializeUnsortedWordList() passing it a pointer to a ListItem* variable.
2. Sort the list using LinkedListSort().
3. Print the list using LinkedListPrint().
4. Create an integer array the same size as the linked list you're working with
5. Print the integer array using a for-loop based on the size of the linked list.
6. Free the entire **sorted** list by going through every ListItem, calling LinkedListRemove() on it. **Important:** be sure to save the nextItem of the ListItem you're deleting because it won't be accessible after you call LinkedListRemove() on it!
7. Free the entire **unsorted** list by going through every ListItem, calling LinkedListRemove() on it. **Important:** be sure to save the nextItem of the ListItem you're deleting because it won't be accessible after you call LinkedListRemove() on it!

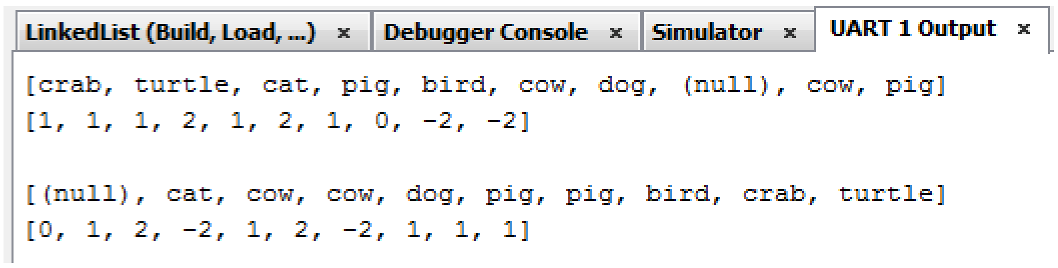
Approaching this lab

Like all labs for this class, you should first start with implementing the LinkedList library. Be sure to handle when malloc() returns NULL, NULL pointers as arguments to functions, and whether the function expects the head of the list or not.

1. Implement LinkedListNew().
   Test this by writing code to create a new list of size 1. Manually inspect the resultant struct that's created using the Variables window in MPLAB X to see that it's correct.
2. Implement LinkedListCreateAfter().
   Test this by creating a list of multiple sizes greater than 1. Manually inspect the resultant list using the Variables window in MPLAB X.
3. Now that you can create lists of a multitude of sizes, implement LinkedListGetFirst(). This function will be helpful for implementing the other functions.
   Test this function by creating a few different lists, storing the pointer to the head node. Pass a non-head node to GetFirst() and see if it matches the memory address of the head node.
4. Implement LinkedListGetSize().
   Run it on the different size lists you created earlier and confirm that results are as expected.
5. Implement LinkedListPrint() and LinkedListSwapData().
   These should be straight-forward to test.
6. Implement LinkedListSort().
   Test by providing very simple lists with NULL-strings, duplicate strings, and strings that are various lengths.

At this point you are now ready for implementing the SortedWordCount() algorithm, and then implementing main() inside sort.c.

Example Output



```
[crab, turtle, cat, pig, bird, cow, dog, (null), cow, pig]
[1, 1, 1, 2, 1, 2, 1, 0, -2, -2]

[(null), cat, cow, cow, dog, pig, pig, bird, crab, turtle]
[0, 1, 2, -2, 1, 2, -2, 1, 1, 1]
```