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

CSE013E: COMPUTER SYSTEMS AND "C" PROGRAMMING

Lab 5 - Linked Lists
17 Points

Introduction

In this lab, you will deepen and solidify the concept of pointers through the concept of a doubly-linked list data structure. You will implement a library to create, modify, and delete linked lists. You will also learn to allocate and de-allocate space during runtime. Finally, you will use your library to compare two algorithms for counting unique words in a list.

Reading

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

Concepts

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

Required Files:

- LinkedList.c
- LinkedListTest.c
- sort.c
- README.md

Lab Files:

- **DO NOT edit** these files:
  - LinkedList.h – Contains the spec and prototypes for the linked-list functions you will implement.
  - BOARD.c/h – Standard hardware library for CSE013E.
  - GNUmakefile — used for compiling this lab on linux.
- **Edit** these files:
  - sort.c – This file contains some starter code for a demonstration of the testing for the sorting algorithms along with a function to generate a word list for the sorts to operate on.
    - It is included as sort_template.c
- **Create** these files:
  - LinkedList.c – Implement the library described in LinkedList.h.
  - LinkedListTest.c – A test harness for your LinkedList library. Should include a main().

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.
- 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:
  - Implement two algorithms for sorting linked lists inside the functions SelectionSort() and InsertionSort(). You will use these with the provided main() code to perform timing experiments that will (hopefully) demonstrate the usefulness of linked lists. You should not modify either main() or CreateUnsortedList() in this file.
- Add inline comments to explain your code.
- Create a readme file named README.md 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.[1]
  - Report the results of the timing experiment in sort.c. Which was faster, SelectionSort() or InsertionSort()? Explain why. Was this what you expected?
  - 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 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.

Doing this Lab on Linux:

For this lab we will continue using the make system. The commands as before are below. For these to work the file 'GNUmakefile' needs to be in the same directory

[1] 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.

as the rest of your files, it is only used when compiling on Linux. You will also need to use MPLABX for certain sections as the stopwatch is an MPLABX only tool.

- `$ make`
  - This command will create the final executable named Lab5
- `$ make LinkedListTest`
  - This command will compile your linked list test harness into the executable LinkedListTest

Grading:

This assignment consists of 17 points:

- 9.0: Automated Components:
  - 6.5 – LinkedList library
    - 3.5 = 0.5 points per function for each of LinkedListNew(), LinkedListCreateAfter(), LinkedListRemove(), LinkedListSize(), LinkedListGetFirst(), LinkedListGetLast(), and LinkedListData().
    - 1.0 point, all functions handle null pointer inputs
    - 1.0 point, all functions that allocate can handle malloc() failures
    - 0.5 no functions reference deallocated data
    - 0.5 no functions allocate unnecessary memory
  - 2.0 – Sort.c
    - 1.0 SelectionSort() gives correct results, and does not print.
    - 1.0 InsertionSort() gives correct results, and does not print.
- 8.5: Human components:
  - 0.5 -- LinkedListPrint()
  - 3.0 -- Test harness for LinkedList
    - 1 point for testing each function
    - 1 point for thorough and diverse tests
    - 1 point for readability of output
  - 2.0 Sort.c:
    - 1 – Correctly performing timing experiment on SelectionSort() and InsertionSort() (results reported in README.md)
    - 1 – explanation in README.md is clear, accurate, and demonstrates understanding of why linked lists are useful.
  - 1.5 points – Readability and code style
  - 1.5 points – Lab writeup
- Deductions:
  - NO CREDIT for sections where required files don't compile

- -1 changing the name of a protoyped function in sort.c (this will break our autochecker scripts!)
- -2: At least one compilation warning
- -2: Extremely bad coding practice (Used extern or goto, extreme inefficiency or unreadability)
- Other deductions at grader's discretion.
- It must compile within MPLABX. Code that does not compile within MPLABX will receive no credit.

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.[2]

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

[2] 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.

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 <-> 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.

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.[3]

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!

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

Sorting

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. In this lab, you will be focusing on two: Selection Sort is simple, intuitive, and easy to implement on a static array, but it is slow. Insertion Sort is usually significantly faster, but it relies on a data structure that allows insertion.

Selection sort partitions the list into two partitions. The first partition is sorted, while the second partition is unsorted. At the start of the algorithm, the sorted partition is empty. With each iteration, the sorted partition grows by one element, and the unsorted partition shrinks, until there are no remaining unsorted items. This is achieved by finding the smallest element in the unsorted partition and moving it to the sorted partition.

Pseudo-code for selection sort is provided below. This pseudocode is written in a way that makes it easy to apply to linkedLists, but note that in this case the pointers could easily be replaced with indexes to a static array. We use two pointers: "FU" stands for "First Unsorted", representing the first item in the unsorted portion of the list. "S" stands for "Scan", since it "scans" through the unsorted partition, looking for the smallest item.

```
FU is pointer to first item
while FU is not tail:
   S is pointer to FU's nextItem
   while S is in list:
      if FU > S:
         swap FU and S contents
      advance S
   advance FU
```

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.

Below is an example of Selection Sort in action:

D	A	C	E	B		//
^FU	^S					// D > A, swap and advance S
A	D	C	E	B		//
^FU		^S				// A < C, advance S
^FU			^S			// A < E, advance S
^FU				^S		// A < B, advance S
						// S is at end of list, advance FU
A	D	C	E	B		//
	^FU	^S				// D > C, swap and advance S
A	C	D	E	B		//
	^FU		^S			// C < E, advance S
	^FU			^S		// C > B, swap and advance S
A	B	D	E	C		//
						// S is at end of list, advance FU
A	B	D	E	C		//
		^FU	^S			// D < E, advance S
		^FU		^S		// D > C, swap and advance S
A	B	C	E	D		//
						// S is at end of list, advance FU
A	B	C	E	D		//
			^FU	^S		// E > D, swap and advance S
A	B	C	D	E		//
						// S is at end of list, advance FU
						// FU is at tail, return

Insertion sort operates in a similar way as selection sort. Like Selection sort, it partitions the list into a sorted and unsorted portion, and uses a double-loop structure to move items from the unsorted portion into the sorted portion. Unlike its slower cousin, insertion sort leverages an "insert" operation to reduce the average time spent in the inner loop. Rather than "scanning" through the unsorted partition in search of the smallest element, it scans through the sorted portion to find the best place to insert the next item. It has the advantage that this scan does not need to cover the entire sorted partition, instead stopping as soon as it finds the appropriate place to insert.

Pseudocode for an insertion sort algorithm is given below. We use three pointers: "FS" stands for "First Sorted", and represents the first item in the sorted partition of the array. "LU" stands for "Last Unsorted," and represents the last item in the unsorted partition. Again "S" stands for "scan," since its job is to scan through the sorted partition to find the appropriate insertion point.

```
FS = tail of list
```

```
while FS is not head of list:
    LU = FS’s previous item
    if LU < FS:
         FS = LU
    else:
         S = FS
         while (S is not tail of list):
            if S's next item is greater than LU:
                break
            else:
                S = S's next item
         remove LU item
         re-insert after S
```

InsertionSort is slower than Selection Sort in a static array, but can be very quick in a linked list data structure.

Note that LinkedList.h does *not* have a true insert operation. You can achieve something similar by removing an item and using CreateAfter() to insert it back in[4], but even this is not a true insert operation because it cannot insert an item at the beginning of a list. Therefore, the pseudocode above progresses from the tail of the list, avoiding the need for a head insertion.

Below, you can see the InsertionSort in action:

D	A	C	E	B				//
			^LU	^FS				// E > B, make an S pointer
				^S				// B < E, advance S
								// S is at end of list, insert
D	A	C		B	E			//
		^LU		^FS				// C > B, make an S pointer
				^S				// C < E, insert
D	A			B	C	E		//
	^LU			^FS				// A < B, advance FS and LU
D	A			B	C	E		//
^LU	^FS							// D > A, make S pointer
	^S							// D > B, advance S
				^S				// D > C, advance S
					^S			// D < E, insert
	A			B	C	D	E	// FS is head, return
	^FS							

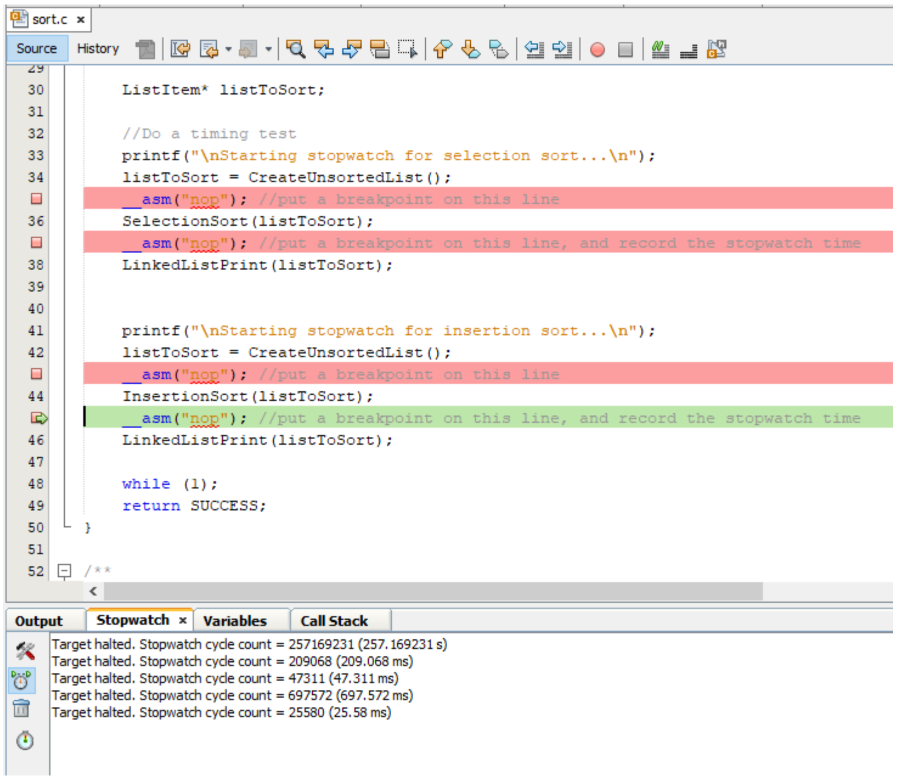
[4] Don’t forget to save the pointer to the ListItem’s data member *before* you remove!

You will write code for both of these algorithms inside of the SelectionSort() and InsertionSort() functions in sort.c.

Evaluating SelectionSort() and InsertionSort() in sort.c

Though we've stated that InsertionSort() is faster than SelectionSort(), we should test that claim experimentally. To do this, we will use MPLAB X's Stopwatch feature, which you can open under "Window -> Debugging -> Stopwatch." Stopwatch measures the time between breakpoints while the debugger is running. This should be done in the simulator, not the actual hardware.

To measure the time required to execute SelectionSort(), we need to put a breakpoint immedStely before and after SelectionSort() and run the debugger. Use the green "continue" button to advance to the next breakpoint, and log the time and cycle count that stopwatch reports:



```c
    ListItem* listToSort;

    //Do a timing test
    printf("\nStarting stopwatch for selection sort...\n");
    listToSort = CreateUnsortedList();
      __asm("nop"); //put a breakpoint on this line
    SelectionSort(listToSort);
      __asm("nop"); //put a breakpoint on this line, and record the stopwatch time
    LinkedListPrint(listToSort);


    printf("\nStarting stopwatch for insertion sort...\n");
    listToSort = CreateUnsortedList();
      __asm("nop"); //put a breakpoint on this line
    InsertionSort(listToSort);
      __asm("nop"); //put a breakpoint on this line, and record the stopwatch time
    LinkedListPrint(listToSort);

    while (1);
    return SUCCESS;
}

/**
```

```
Target halted. Stopwatch cycle count = 257169231 (257.169231 s)
Target halted. Stopwatch cycle count = 209068 (209.068 ms)
Target halted. Stopwatch cycle count = 47311 (47.311 ms)
Target halted. Stopwatch cycle count = 697572 (697.572 ms)
Target halted. Stopwatch cycle count = 25580 (25.58 ms)
```

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. At this point you are now ready for implementing SelectionSort() and InsertionSort in sort.c. The debugger will be very useful here. You may find it convenient to use a shorter list during testing.
7. Once SelectionSort() and InsertionSort() are functional, use the Stopwatch tool and the debugger to measure the time required for each sort. Record the results in README.md and explain them.