Lab 8: Linked Lists

CSE/IT 113L

NMT Department of Computer Science and Engineering

"The programmers of tomorrow are the wizards of the future. You're going to look like you have magic powers compared to everybody else."

— Gabe Newell

"The amateur software engineer is always in search of magic."

— Grady Booch

"No one hates software more than software developers."

— Jeff Atwood

Contents

1 Introduction

Arrays are nice, but they have one huge limitation: static size. Arrays are created with a length and you cannot change that length if you end up with more data than expected. Therefore, arrays are a poor choice for user input because users are unpredictable.

Like arrays, linked lists are containers for data, however, linked lists are dynamically sized. A linked list is a data structure that holds a data element and a link (*pointer*) to the next element in the list. You can access the entire list by starting at the *head* of the list and moving from element to element in the list.

2 Overview

This lab is designed to tie together pointers, structures, and malloc. It will also give you first-hand experience at using the linked list data structure. In this section, we try to give you some of the basic functions you are going to need for the lab. The idea is that you have code that works (or mostly works) so you are not starting at square one for the lab.

3 Getting Started

A data element in a linked list has two parts: the data and a pointer to the next element. Because of this we need to make a **self–referencing structure**:

```
struct node_t {
        double x;
        struct node_t *next;
};
```

Notice that we have had to create a struct node_t * in order to point to the next node. But there is a problem: at the time we need to have a pointer to another node structure, we have not actually finished the definition of the structure we are defining. So, we do not have a new type to use yet. We solve this problem by simply referencing the structure itself using the structure tag node!

Now that we have the structure that holds a double, you can create a node and then add it to the list. Once you have a list with one or more elements, you can add more nodes at various locations (head, tail or somewhere in the middle), print the list, count the number of nodes, or you can traverse the list to find nodes to delete. Once you are finished with the list you can delete the entire list.

Use the following function decalarations in your code. You, of course, have to supply the definitions. Also, make sure to comment these functions with Doxygen style.

```c
/* function declarations needed for the linked list */

#include <stdio.h>
#include <stdlib.h>

struct node_t {
        double x;
        struct node_t *next;
};

struct node_t *create_node(double n);
void print_node(struct node_t *node);
void print_list(struct node_t *head);
struct node_t *insert_head(struct node_t *head, struct node_t *node);
struct node_t *insert_tail(struct node_t *head, struct node_t *node);
struct node_t *insert_middle(struct node_t *head, struct node_t *node, int pos);
int count_nodes(struct node_t *head);
struct node_t *find_node(struct node_t *head, double n);
struct node_t *delete_node(struct node_t *head, double n);
void delete_list(struct node_t *head);

int main() {

      /*add code here
        work on design/pseudo-code
        before you start coding!
        Drawing pictures really does help!
        */

      return 0;
}
```

4 Lab Specifications

Exercise 1 (lab8.c, lab8.h, l8.valgrind, l8-output.script).
Implement a linked list of real numbers. Your program should have a menu interface and the ability to store an arbitrary number of doubles—limited only by how much free memory is available for your computer. You **will** be writing a README for this lab, do not forget.

Your menu interface must have at least the following options:

1. Enter a number

   A submenu – enter item at the head, tail, or middle of the list?

2. Delete a number

3. Print all numbers

4. Tell how many items are in the list
5. Find if a number is in the list
6. Quit

In implementing your linked list, you must meet (at a minimum) the following requirements:

- When your program ends, you must correctly free all dynamic memory.
- You must use Valgrind/Leaks to make sure there are no memory leaks. Show Valgrind/Leaks running with no memory leaks in a script file titled **l8.valgrind**. To use Valgrind:

  ```
  valgrind --leak-check=full ./program
  ```

  To use Leaks (MAC only):

  ```
  leaks --atExit -- ./program
  ```

  There should be no heap memory in use at exit. Your program runs correctly if Valgrind/Leaks tells you that "All heap blocks were freed – no leaks are possible" in all possible cases of your program running (you will have to test with Valgrind/Leaks multiple times!). Of course, you cannot test *all* possible cases: try to select a good representative of all cases. For example, edge cases are especially useful in testing.
- You must implement the linked list correctly: it cannot have a built-in limit on the number of elements it can hold. System limits, such as available memory, will of course cap the length of your linked list.
- You must use `while`-loops to traverse the list in an iterative fashion. No recursion.
- You must have a *head* pointer that points to the first node in the list. You have to keep track of this pointer as insertions and deletions in your list may change the head of the list.
- You must have a *node* structure that holds a pointer to the next node in the list and a double data value.
- You must use NULL to designate the end of the list.
- You must check that malloc allocates memory successfully. *Hint:* Use the errno.h header file in your program.
- You must initialize all pointers to NULL, except for pointers that are initialized at the time of their declaration.

- You must implement **functions** (which means passing your *head* pointer around) in order to do the following. Required function names are in parenthesis. Function parameters and return types are given below.
  - Create a new node (create_node)
  - Print a node (print_node). Both the double it stores, the address of the node, and the address `next` points to. If `next` points to NULL, print the string "NULL".

    The ability to print a node is useful for debugging. After you create a node using create_node() you can print it out to make sure the node was successfully created. You can also use it to debug your linked list, i.e. is `next` pointing to the correct node in the list?
  - Add an element to a list at the head of the list (insert_head)
  - Add an element to the "middle" of the list based on position. Insert the node at the desired position. A position of 1 indicates the head of the list. If position is greater than the number of nodes in the list, then the node is inserted at the tail of the list. Error check that position is greater than or equal to one. (insert_middle)
  - Add an element to the tail of the list. Walk down the list to find the tail. (insert_tail)
  - Display the contents of the list. For each node print out the double, the address of the node, and the address of the next node on a single line. This function should call print_node to do the printing of a node. (print_list)
  - Delete a node by its value and correctly free the memory occupied by that node. If there is more than one node in the list with the same value, it will delete the first one it finds.

    This has a number of cases: the found node is at the head, the tail, somewhere in the middle, or not found at all. You must account for cases. (delete_node)
  - Delete all the nodes in the list and correctly free the memory occupied by those nodes. This functions must account for the fact that it may be called when the list is empty. (delete_list)
  - Counts elements in the list by traversing the list. (count_nodes)
  - Find if a given number is in the list by searching the list by traversal and returning a pointer to the node if the number is found or `NULL` if not. If the node is found let the user know the number was found and print the node. If it is not found, let the user know it wasn't found in the list. For the comparison of floating point numbers use the isgreaterequal() and islessequal() functions provided in `math.h` rather than ==. (find_node)
- You must return the head pointer from all functions where the head location may change.
- You must account for the fact that the list may be empty in all insert functions and you are adding the first node, so the head pointer will change.
- Create a script file of your program compiling with no errors and running. Make sure to show all menu options working. Name this file **l8-output.script**.

5 Part II

Now that you know the mechanics of a linked list, it is time to apply that knowledge. **Complete only ONE of the following exercises.** This means that you will only be required to submit the files listed for submission under **Submitting** for the exercise you choose below. For example, if you choose to complete the **Collatz Conjecture**, you will include all files listed for the linked list exercise and all files listed for the collatz exercise. You will not need to include files for the DNA exercise because you did not choose it.

5.1 Collatz Conjecture

It is conjectured that if you start with any natural number n and perform the following operations for odd and even numbers you will generate a sequence of numbers that always ends in 1:

1. if n is even divide it by 2 to get $n/2$
2. if n is odd multiply by 3 and add 1 to get $3n + 1$
3. repeat with the new value and stop once you have reached 1.

The sequence of numbers generated in this fashion is known as the hailstone sequence. See `http://en.wikipedia.org/wiki/Collatz_conjecture` for more info.

For example, if $n = 6$ the hailstone sequence is: 6, 3, 10, 5, 16, 8, 4, 2, 1. The cycle length of 6 is 9, where the cycle-length is the number of terms in the sequence.

Exercise 2 (collatz.c, hailstone).
Your job is to write a program named `collatz.c` that prompts the user for n, and prints out n, the cycle length and the hailstone sequence for n. For instance if $n = 3$, the output would be

```
n = 3
cycle length = 8
3, 10, 5, 16, 8, 4, 2, 1
```

You terminate the sequence with a new line.

As you don't know the cycle length of the sequence you are going to generate, this is a job for a linked list! You will make all insertions at the tail of the list. This way when you walk through the list to print, the sequence is the correct order. For performance reasons, it is best to keep track of a *tail* pointer to keep track of the end of the list and use the tail pointer to insert new nodes into the list.

The cycle length is equivalent to the number of nodes you creates. You can just keep track of the number of nodes you create in an integer. After you are done printing the sequence, you need to free the list. Check with valgrind. Capture the output of hailstone sequence for $n = 3$, $n = 1000$, and $n = 10000$ into a file named hailstone. Use redirection for both input and output. Name your output file `hailstone`.

5.2 DNA Base Pairs

DNA is comprised of two long strings of nucleotides. Each nucleotide has one of 4 types: cytostine (C), guanine (G), adenine (A), or thymine (T). If given a single half of a strand of DNA, its compliment can be determined by following these rules:

1. Cytostine (C) pairs with guanine (G), and vice versa
2. Adenine (A) pairs with thymine (T), and vice versa

Consider the following example:

```
G T A C C T A G G
C A T G G A T C C
```

Exercise 3 (dna.c, dna.script).
Given a valid DNA sequence of characters 'C', 'G', 'A', and 'T', and an invalid sequence of the corresponding characters with random elements missing, insert the missing characters back into the invalid sequence. Use lowercase letters to indicate which characters were inserted. For example, given ACGT and TGA, return TGcA.

A linked list lends itself toward this exercise because it is not bound to a static size. Encode both sequences into their own linked lists, then use the `insert_middle()` function to repair the invalid sequence.

Your program should allow the user to input two strings. You may assume:

1. The inputs only have valid characters (uppercase 'C', 'G', 'A', and 'T')
2. The second string is a copy of the first string with a random number of random elements missing (therefore, `strlen(input2) <= strlen(input1)`)

```
Enter two sequences:
> ACGT
> TGA
Output: TGcA
```

You have to use the linked list in the `lab8.c`. You could define enumeration for converting numbers and characters.

Name your source code `dna.c`. Capture the output in a script file named `dna.script` with 3 different paired-strings

6 README

Exercise 4 (README).
Every lab you turn in must include a README file. The name of the file is README with no file extension.The README should include the following sections Purpose, Pseudo-Code (optional), and Conclusion:

- **Purpose:** describes what the program does (what problem it solves). Keep this brief.
- **Pseudo-code:** contains the pseudo-code you wrote for the lab. This depends on the lab. Some require pseudo-code; some do not. It will be mentioned specifically in the lab document if pseudo-code is required.
- **Conclusion:**
  - What you learned. What new aspect of programming did you learn in this lab? This is the portion of the lab where you want to be analytical about what you learned.
  - Did Pair Programming help you in solving the problem and completing the prelab? Did you have problems working with your buddy?
  - Did you work with your buddy on the lab? What sections did you discuss? Did you and your buddy carry out a review session with each others code?
  - Did you encounter any problems? How did you fix those problems?
  - What improvements could you make?

The conclusion does not have to be lengthy, but it should be thorough.

Submitting

You should submit your code as a tarball file that contains all the exercise files for this lab. The submitted file should be named (**note the lowercase**)

`cse113_firstname_lastname_lab8.tar.gz`

Upload your `.tar.gz` file to Canvas.

List of Files to Submit

Exercises start on page 2.