Lab 9: Linked Lists

CSE/IT 113

> "In theory, there is no difference between theory and practice, but in practice, there is."
>
> — Anonymous

> "...and then it occurred to me that a computer is a stupid machine with the ability to do incredibly smart things, while computer programmers are smart people with the ability to do incredibly stupid things. They are, in short, a perfect match."
>
> — B. Bryson

> "If you understand what you're doing, you're not learning anything."
>
> — Anonymous

> "Every time I see an adult on a bicycle, I no longer despair for the future of the human race."
>
> — H. G. Wells

---

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 Requirements

Implement a linked list of real numbers. Your program should have a menu interface and the ability to store an arbitrary number of `double`s—limited only by how much free memory is available for your computer.

Your menu interface must have <u>at least</u> 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. Quit

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

- When your program ends, you must remember to correctly free all dynamic memory.
- You must use Valgrind to make sure there are no memory leaks. To use Valgrind:

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

  There should be no heap memory in use at exit. Your program runs correctly if Valgrind 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 multiple times!). Of course, you cannot test *all* possible cases: try to select a good representative of all cases.
- You must implement the linked list correctly: it cannot have a builtin 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.
- 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`.
- You must use `NULL` to designate the end of the list.
- You must check that `malloc` allocates memory successfully.
- 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
  - Create a new node (`create_node`)
  - Print a 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. (`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`)
- 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.

3 Getting Started

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.

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.

```
/* 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 *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;
}
```

3.1 Submission guidelines

Compile and run the program and capture the output of both compilation and running in a script file. Make sure you capture all menu options. Tar the source code, script, and README into a tarball named `cse113_firstname_lastname_lab9.tar.gz` and upload to Canvas before the due date.