Wesleyan University, Fall 2022, COMP 211

Lab 9: Linked lists and malloc

1. Overview

The goal of this lab is to is to understand linked lists, implement various operations on a linked list, and learn about how to dynamically allocate memory using malloc.

2. Create and organize your program files

For this lab, you should create a new directory lab9 in your labs directory to hold your code. Then do the following.

(1) Download the following files from the lab website, `Makefile, list.c, list.h, driver.c`. During this lab you will add code to the following files: `list.c`.

(2) Use the `Makefile` to compile the current state of the program by typing `make driver`. Then type `./driver` to run the program.

3. Linked list definition

Like an array, a linked-list is a way to represent a sequence. Compared to an array, some operations on a sequence are more efficient than an array using a linked list and other operations are less efficient. In lab today you will implement a linked-list data structure along with functions implementing operations on the linked list. Some of this code has already been written for you, and some code you will need to write.

Creating a node struct and a list struct

In the file `list.h`, I have already defined for you a struct named `node` representing one node in a linked list. You'll see the following fields in the `node` struct.

- a `data` field of type `char`. This holds the data for the node.
- a `next` field of type `struct node*`. This is a pointer and contains the memory location (address) where we will find the next node in the linked list. If there is no next node, then its value is NULL.

I have also defined for you a struct named `list` representing a sequence of node structures. You'll see the following fields in the `list` struct.

- a `head` field of type `struct node*`. This is a pointer to the first node in the linked list and contains the memory location (address) where we will find the first node in the linked list. If the list is empty then its value is NULL.

Note that there are alternative approaches to implement a linked list (such as with a dummy node), that result in cleaner code but are a bit more complex to grasp.

Creating a new node in a function

When working with linked lists, a new issue arises that we have not seen before. If we are going to insert a new node in a linked list, we need to create that node and insert it. If we did all of our operations in main, without using functions, we'd be fine. But for code clarity and flexibility, we'd like to perform operations like insertion in a function.

What is the problem with creating a struct such as node in a function, and wanting access to that variable outside the function? Can't we do that by returning the struct we create? The problem is that for insertion, we don't want to return anything, we just want to modify the list. Which means, unless we do something differently, the node struct variable we create and insert into the list, which is on the stack, will disappear when the function returns because the binding table containing that variable will have been popped.

What can we do? We instead create the node struct by allocating memory in another place, not on the stack (instead on what we saw is called the heap), using the `malloc` function. The `malloc` function, or memory allocation function, takes as an argument, the number of bytes of memory to allocate and returns a pointer to the beginning of the memory allocated. For example, in `list.c` in the `insert` function you will see the following code.

```
node *n = malloc(sizeof(node));
n->data = c;
n->next = NULL;
```

What is this code doing? This code asks malloc to allocate a block of memory of size, `sizeof(node)`, and returns a pointer returned of type `(node*)`. We then store the pointer value returned in the variable `n`. Because memory for a node has actually been allocated, we can index into that memory by dereferencing `n` and can then initialize the various fields of the node struct. Now when the function returns, this node struct will still exist.

Freeing memory created using malloc

When memory is allocated using malloc, we can no longer rely on the memory allocated to be automatically released for us. Instead, we must explicitly deallocate any memory we allocate. This is very important to do! Otherwise you can get memory leaks. This may not seem like a big deal if you only allocate memory for a few nodes, but imagine a program which inserts and removes many nodes from a linked list: while the size of the linked list would be the net of the number of nodes inserted and removed, the amount of memory allocated will be for every node ever inserted. In such a scenario, it is possible for your program to run out of memory, using all available memory and resulting in your program and computer locking up. So please free any memory allocated, at the appropriate time.

How do you free memory? By using the function `free`. Call `free` and pass it a pointer to the memory to be freed. I.e.,

```
free(n);
```

In a linked-list when should the memory you allocate be freed? Whenever a node is deleted from the linked list. You'll see a reminder in the `deletion` function to be sure to free the memory allocated for the node you are deleting.

Implement the linked-list functions

Now it is your turn to do some coding. Like the other data structures we have seen, there are a number of operations that can be performed on a linked list. The function signatures have already been written for you. Some of the function bodies have been partly or completely filled out. You will fill out the remaining functionality as specified in the code.

- `create`: initialize the fields in the list structure. This function has already been written for you.
- `is_empty`: check whether the list is empty.
- `insert`: insert an element into the list at the specified location.
- `deletion`: delete an element from the list at the specified location.
- `getval`: return the value of the data field for the element at the specified location.
- `size`: return the size of the linked list.
- `print`: print all of the elements in the stack. This function has already been written for you.

Testing the code

Compile and test your code using the `Makefile` and `driver.c` which have already been written for you.