Prelab 9: Linked Lists

CSE/IT 113

1 Introduction

A major limitation of arrays is their static size. Arrays are created with a length, and you can't change that length at run-time. Say you created a structure to hold information on a CD (title, artist, year released, etc) and allocated an array of structures of some SIZE to hold the users CD information. What happens when the user wants to enter the SIZE + 1 record? Your program doesn't have enough room. Not a very practical program. How many CDs the user of the program is going to enter is unknown. We need a way to add CD records on the fly. Linked lists allow us to add as many records as we want, without having to know how many records are going to be entered.

Like arrays, linked lists are containers; however, linked lists are created at run-time. A linked list consists of a chain of elements called **nodes**. A node is a structure that holds the data to be stored and a *pointer* to the next node in the list. As new data is created, a new node is created and added to the end of the list. To indicate the end of the list, the last node in the list points to the NULL pointer. You access the entire list by starting at the *head* of the list and moving to each node in the chain.

2 Reading

Pages 142 - 149 *Understanding and Using C Pointers.*

3 Background Material

In this lab, you will be focusing on two key concepts: linked lists and pointers (which are what allow linked lists to work).

3.1 Pointers - More In-depth!

This prelab starts out by revisiting pointers. If you are **completely** comfortable with pointers, feel free to skip to section 4. If not, this section tries to explain pointers in a different way.

You should remember that the variable arguments to a `scanf` statement are prefixed by an ampersand (`&`). You have been told that the `&` makes a *reference* to the variable. A *pointer* is a variable that holds a reference to another variable.

You declare a pointer in the following way:

```
type *ptr;
```

This bit of code gives you a pointer to a variable named `ptr` of type `type`. You then assign another variable to that pointer in the following way:

```
ptr = &var;
```

At this point, `*ptr` is interchangeable with `var`.

For example, consider the following program (while it isn't very useful, it at least demonstrates how a pointer works):

```c
/* Example 1 */
#include <stdio.h>

int main(void)
{
        int a;
        int *b;
        b = &a;
        a = 0;
        printf("a: %d\nb: %d\n\n", a, *b);
        *b = 5;
        printf("a: %d\nb: %d\n\n", a, *b);
        a = 2;
        printf("a: %d\nb: %d\n", a, *b);
        return 0;
}
```

As a more useful example, consider the following program that uses pointers to be able to change passed variables from a function:

```c
/* Example 2 */
#include <stdio.h>
void sort(int *a, int *b);

int main(void)
{
        int x, y;
        x = 3;
        y = 5;
        printf("x: %d\ny: %d\n", x, y);
        sort(&x, &y);
        printf("x: %d\ny: %d\n", x, y);
        return 0;
}

void sort(int *a, int *b)
{
        int tmp;
        printf("sort();\n");
        if (*a < *b)
        {
                tmp = *a;
                *a = *b;
                *b = tmp;
        }
}
```

For reference, compare the above code to the following that doesn't use pointers (notice that it doesn't work without the pointers):

```c
/* Example 3 */
#include <stdio.h>

void sort(int a, int b);

int main(void)
```

```
{
        int x, y;
        x = 3;
        y = 5;
        printf("x: %d\ny: %d\n", x, y);
        sort(x, y);
        printf("x: %d\ny: %d\n", x, y);
        return 0;
}

void sort(int a, int b)
{
        int tmp;
        printf("sort();\n");
        if (a < b)
        {
                tmp = a;
                a = b;
                b = tmp;
        }
}
```

3.1.1 Creating and Deleting Pointers

The above example programs use a pointer that points to an existing variable. What if we wanted to just make a new variable at that point? A function exists to do just that: `malloc`. `malloc` takes in one argument, the number of bytes to allocate. Memory that is allocated by `malloc` is placed on the heap.

To determine the number of bytes to allocate you use the **`sizeof`** operator. To allocate contiguous memory (i.e. arrays) multiply the **`sizeof`** operator by how many elements you want. `malloc` is found in the C standard library `stdlib.h`

Normally, you call `malloc` in the following way (for whatever type you are working with):

```
/* ptype now points to the memory allocated by malloc */
TYPE *ptype = NULL; /* pointer to data of type TYPE */
ptype = malloc(sizeof(TYPE));

/* or for more than one element */
/* ptype = malloc(number_elements * sizeof(TYPE)); */
```

where `TYPE` is a primitive type (`int, short, long, char, float, double`) or a user defined type such as a structure.

`malloc` returns a `void` pointer, which is a generic pointer. If `malloc` did not return a `void` pointer, you would have to have functions that returned each primitive type and you would still have the problem of how to return user defined types.

Using `malloc` is straightforward. For example,

```
int *p = NULL;
p = malloc(sizeof(int));
if (!p) {
    printf("malloc failed\n");
    exit(10);
}
```

allocates space for an integer and since `malloc` may fail, tests for that possibility and exits if no memory was allocated.

Now consider the following code, which is a rewrite of Example 2 using `malloc`:

```c
/* Example 4 */

#include <stdio.h>
#include <stdlib.h>
void sort(int *a, int *b);

int main(void)
{
        int *x = NULL;
        int *y = NULL;
        x = malloc(sizeof(int));
        if (!x)     /* malloc may fail */
                exit(10);
        y = malloc(sizeof(int));
        if (!y)
                exit(10);
        *x = 3;
        *y = 5;
        printf("x: %d\ny: %d\n", *x, *y);
        sort(x, y);
        printf("x: %d\ny: %d\n", *x, *y);
        return 0;
}

void sort(int *a, int *b)
{
        int temp;
        printf("sort();\n");
        if (*a < *b)
        {
                temp = *a;
                *a = *b;
                *b = temp;
        }
}
```

Whenever you use `malloc` to create a variable, you must `free` the memory space when you are done. Unlike automatic variables that are deleted when they go out of scope (automatic storage duration), variables created with `malloc` are not. They will only be deleted for you when your program ends. Failing to free variables causes a *memory leak*. While this isn't a serious problem for programs that don't run very long, it is a good habit to always free variables created with `malloc`.

`free` is fairly simple to use:

```c
free(var);
```

However, this will only work on a variable that was created using `malloc`. Using `free` on a variable not made with `malloc`, or on a variable that has already been freed, will crash your program.

3.1.2 Pointers to Structures

Pointers to structures are common. However, accessing a value in a structure pointer gets a bit ugly. Suppose you have a structure that has `foo` as a data member and you have a pointer to the structure named `mystruct`. To access the member `foo` you could do something like this:

```c
(*mystruct).foo
```

Accessing members this way gets unwieldy. Because of this, C has a special syntax for accessing a value in a structure pointer:

```
mystruct->foo
```

Why? Assume `structA` holds a `structB` that holds a `structC` that holds an integer.
Accessing the integer without the special syntax:

```
(*(*(*structA).structB).structC).val
```

Accessing the integer with the special syntax:

```
structA->structB->structC->val
```

As well as being a bit shorter, many people find the latter form to be more readable.

You can dynamically create structures using `malloc`. The following code creates a new node of type node_t, where the data being held in the node_t is of type `int`.

```
struct node_t {
        int data;
        struct node_t *next; /* pointer to a structure of the same type */
};

....

int main()
{
        struct node_t *pnode; /* create a pointer to a node_t */
                              /* no memory is allocated */

                              /* allocate space for a node */
        pnode = malloc(sizeof(struct node_t));
        if (!pnode)
                exit(100);

        /* do something with pnode */
        pnode->data = 7;
        pnode->next = NULL; /* initialize next pointer to NULL */
        printf("data = %d\n", pnode->data);

        /* free memory created with malloc once done with pnode */
        free(pnode);

        return 0;
}
```

4 Linked Lists

A linked list is a data structure that holds a list of nodes. The most common way to implement a linked list is to have a node that holds the data and points to the next node in the list. To keep track of the start of the list you use a pointer of type node, that points to the head of the list. You can append nodes to either the front, the rear, or in the middle of the list.

Appending a node to the front of the list is done by creating a new node object and placing it at the start of the list. It should be noted appending a node at the head of the list will reverse the order of where things are in the list compared to the order in which they were entered. This is a very useful idea. And if you just limit your insertions and deletions to the head of a list you have implemented a stack.

Appending at the rear can be done by traversing the list. That is, by following the links (the node pointer) to the end of the list. Another way to add elements at the end of the list is to use a tail pointer: a pointer that points to the tail node of the list. You can then access the end of the list without having to follow all the links in the list.

Inserting in the middle of the list typically requires some criteria to find a node and then you have to decide do you want to insert before the node or after the node.

5 Questions

Assume you have the following structure and pointers

```
struct node_t {
        int data;
        struct node_t *next;
};

struct node_t *head, *tail;
```

The head node points to the beginning of the list, while the tail pointer points to the end of the list. Capital letters (A, B, C, etc) are structures of type `node_t`. The notation A, B, C, D, E, F represents a linked list with A →next = B, B→next = C, etc. The head of the list is A, and the tail is F (head = A, tail = F)

To add a new node to an empty list, the pseudocode would look like this:

```
head = NULL;
tail = NULL;

//adding node to empty list
A = malloc(node_t)
A->next = NULL;
head = A;
tail = head;
//list contains A and both tail and head pointers
//point to A.
```

Note: head→next = NULL, as well as tail→next = NULL, after these assignments.

To delete this node

```
head = NULL;
tail = NULL;
free(A);
//list is now empty
```

For each question, write pseudocode to show the list before the requested operation, and after the requested operation. Be sure to show what the head and tail pointers point to, malloc (add) and free (delete) operations, and where the appropriate next members point to. Make sure you have the order of operations correct, because with pointers, order does matter. Your pseudocode should follow the format of the following example sans comments (//).

```
EXAMPLE:
Add a node J to the middle of the list, in between I and K, of the list H, I, K, L.

Pseudocode:

List before addition of J:
```

```
head -> H
H-> next = I
I-> next = K
K-> next = L
L-> next = NULL
tail -> L

To add J:
//a two step process -- create the node and then add it to the list

//create and initialize the node
malloc J
J-> next = NULL

//now add the node to the existing list
J-> next = K
I-> next = J

//**Note** If we reverse the order of those two steps, then we have lost our list.
//If we declare I-> next to be J without first setting J-> next to be K, then
//nothing points to K and we have successfully lost that memory.

New list after the addition of J:

head -> H
H-> next = I
I-> next = J
J-> next = K
K-> next = L
L-> next = NULL
tail -> L
```

Prelab Questions:

1. Add a node H to the end of the list D, C, B, G. Use the tail pointer.
2. Add a node E to the head of the list A, B, C, D. Use the head pointer.
3. Add Z before the D in the list A, D, C, H, E, F
4. Delete G from the list D, E, F, G, H, I, J, K
5. Delete A from the list A, B, C, D,
6. Delete D from the list A, B, C, D, Use the tail pointer.

6 What you need to turn in

Write your pseudocode in a text file called `cse113_fname_lname_prelab9.txt`.
Submit your text file to Canvas before the due date.