CptS 122 – Data Structures

Lab 3: Data Structures and Dynamic Doubly Linked Lists in C

Assigned: Week of January 29, 2024
Due: At the end of the lab session

I. Learner Objectives:

At the conclusion of this programming assignment, participants should be able to:

- Design, implement and test a dynamic doubly linked list in C
- Programmatically traverse through a doubly linked list
- Implement insertions, in order, into a doubly linked list
- Implement deletions from a doubly linked list
- Support modifications and printings of data in a doubly linked list
- Develop a menu system
- Apply standard library functions malloc () and free ()
- Design and implement basic manual unit tests

II. Prerequisites:

Before starting this programming assignment, participants should be able to:

- Analyze a basic set of requirements for a problem
- Compose a small C language program
- Compile a C program using Microsoft Visual Studio 2022
- Create test cases for a program
- Apply and implement arrays and strings in C
- Apply and implement recursion in C
- Apply and implement structs in C
- Apply and implement pointers in C
- Apply and implement dynamic memory in C

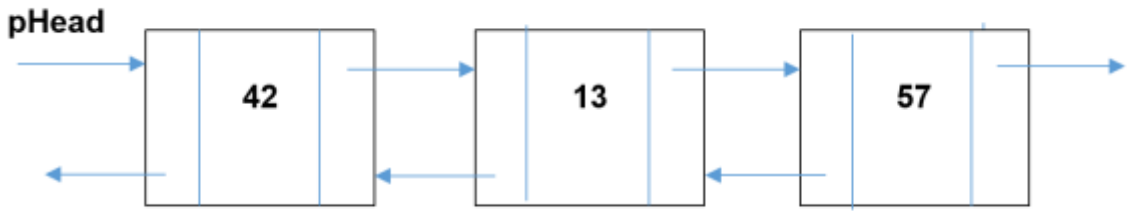
III. Overview & Requirements:

This lab, along with your TA, will help you navigate through designing, implementing, and testing a dynamic linked list.

Labs are held in a “closed” environment such that you may ask your TA questions. Please use your TAs knowledge to your advantage. You are required to move at the pace set forth by your TA. Please help other students in need when you are finished with a task. You must work in teams! However, I encourage you to compose your own solution to each problem. Have a great time! Labs are a vital part to your education in CptS 122 so work diligently.

Tasks:

1. This lab is similar to the one that you worked on last week. However, you will be working with doubly linked lists for this lab. Linked lists may be used to implement many real-world applications. Recall, linked lists are data structures, which represent collections of nodes that may be accessed sequentially via a pointer to the first node. A node contains data and a pointer to the next node in sequence and for doubly linked lists a pointer to the previous node in sequence. When the last node in the list is reached, its next pointer is NULL. A logical view of a *doubly* linked list of integers is illustrated below:



This is an extension to Lab 2. If you did not complete all of the functions last week, this is your chance. However, you will need to update your prior implementation to accommodate two links in your nodes. Did you do a good job of designing your functions so that you could make the necessary changes to the nodes without impacting the code so much?

Build an application, in C, for storing contact information (you must have one header file and two source files). For each contact you must store a name, phone number, email, and professional title. Your application must support insertions in order (based on last name), deletions, modifications, and printings of contacts. For this task you will be required to implement a dynamic doubly linked list, which grows and shrinks at runtime. Build a menu that allows the user to add, remove, edit, print, store, and load contact information. The *store* feature should write contact information found in the list to a file. The *load* feature should read contact information from the same file into the list. Note: you should store the contact information in a struct called `Contact`. Each `Node` must be a struct, which consists of a `Contact` and a pointer to the next `Contact` in the list. Please see below.

```
typedef enum boolean
{
	FALSE, TRUE
} Boolean;

typedef struct contact
{
	char name[50];
	char phone[12]; // 18005557577
	char email[50];
```

```
	char title[20];
} Contact;

typedef struct node
{
	Contact data;
	struct node *pNext;
	struct node *pPrev; // this lab is now using two links
} Node;
```

Which list operations should you support? There are more than the ones listed below!

```
// Description: Allocates space for a Node on the heap and initializes
the Node with the information found in newData.
// Returns: The address of the start of the block of memory on the heap
or NULL if no memory was allocated
Node * makeNode(Contact newData);

// Description: Uses makeNode () to allocate space for a new Node and
inserts the new Node into the front of the list.
// Returns: TRUE if memory was allocated for a Node; FALSE otherwise
Boolean insertContactAtFront(Node **pList, Contact newData);

// Description: Uses makeNode () to allocate space for a new Node and
inserts the new Node into the list in
//              alphabetic order ('a' - 'z') based on the name field
// Returns: TRUE if memory was  allocated for a Node; FALSE otherwise
Boolean insertContactInOrder(Node **pList, Contact newData);

// Description: Deletes a Contact in the list based on the name field;
deletes the first occurrence of the name
// Returns: TRUE if the Contact was found; FALSE otherwise
Boolean deleteContact(Node **pList, Contact searchContact);

// Description: Edits a Contact in the list based on the name field;
edits the first occurrence of the name
// Returns: TRUE if the Contact was found; FALSE otherwise
Boolean editContact(Node *pList, Contact searchContact);

// Description: Loads all Contact information from the given file, in
alphabetic order, based on the name, into the list
// Returns: TRUE if all Contacts were loaded; FALSE otherwise
Boolean loadContacts(FILE *infile, Node **pList);

// Description: Stores all Contact information from the list into the
given file
// Returns: TRUE if all Contacts were stored; FALSE otherwise
Boolean storeContacts(FILE *infile, Node *pList);
```

```
// Description: Prints all contact information in the list
// Returns: Nothing
void printList(Node *pList);
```

2. Test your application. In the same project, create one more header file `testList.h` and source file `testList.c` (for a total of at least five files). The `testList.h` file should contain function prototypes for test functions you will use on your list functions. The `testList.c` source file should contain the implementations for these test functions. You will be designing and implementing *unit* tests. You will have at least one test function per application function. Your test functions must display a message "test failed" or "test passed" depending on the results. For example, you will have an application function called `deleteContact()` (or a function very similar) that was used to remove contact information from the list. In this task, you will need to create a test function called `testDeleteContact()` that passes in various contact information directly into `deleteNode()` to see if it works correctly.

IV. Submitting Labs:

- You are not required to submit your lab solutions. However, you should keep them in a folder that you may continue to access throughout the semester.

V. Grading Guidelines:

- This lab is worth 10 points. Your lab grade is assigned based on completeness and effort. To receive full credit for the lab you must show up on time, continue to work on the problems until the TA has dismissed you, and complete at least 2/3 of the problems.