Wesleyan University, Fall 2023 , COMP 211

Lab 10: Dummy head nodes and doubly linked lists

---

1. Overview

The goal of this lab is to is to deepen our understanding of linked lists. We will change our implementation of a linked list to include what is called a dummy head node. As we shall see, this will greatly simplify the implementation of operations on a linked list. Additionally, so far we have focused on what are called singly linked lists. In this lab we will start looking at what are called doubly linked lists.

2. Create and organize your program files

For this lab, you should create a new directory lab10 in your labs directory to hold your code. Then do the following.

(1) Download the following files from the lab website. During this lab you will modify both `list.c` and `list.h`.

```
Makefile
driver.c
list.c
list.h
```

3. Dummy head nodes

When implementing the operations on a linked list in lab last week you may have observed that depending on whether the list was empty, your code would perform one set of operations, and if the list was not empty, your code would perform another set of operations.

Thus, if our linked-list always had a node in it, even when empty, our code could be simplified since we would no longer need to differentiate the case when the list was empty. We can ensure that the list always has a node in it by using what is called a dummy head node: this node does not contain any data, but simply exists to simplify operations on the linked-list.

4. Update list struct to include a dummy head node

In the file `list.h`, I have already defined for you a struct named `node` representing one node in a linked list, and a struct named `list` representing a sequence of node structures. As before, you'll see the following field in the `list` struct.

- `head` field of type `struct node*`

However, rather than point to the head of the list, the `head` field will now point to the dummy head node of the linked list, which we will create when we create the list.

5. Update linked-list functions

Last lab we implemented a number of operations on a linked-list. In today's lab, we will re-implement those same operations, but will instead take advantage of the fact that the linked-list contains a dummy head node to simplify the implementation. Rather than referring to the code from last lab to help you write those functions, you should try writing these functions from scratch, now taking into consideration the existence of a dummy head node. 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. You'll see now that a dummy head node is created and inserted into the list, so that the list always contains at least one node, even when "empty."
- `is_empty`: check whether the list is empty. Note that since a dummy head node is being used, the check for whether the list is empty must change slightly.
- `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 list. This function has already been written for you.

6. Doubly linked-lists

While a singly linked list can only be looped through in the forward direction, starting at the dummy head node and ending when `node->next` is `null`, nodes in a doubly linked list additionally contain a pointer to the previous node in the list, allowing movement in both the forward and backwards direction. Looking at `list.h` you will see the following struct declarations defining the doubly linked list. Like the dummy head node in the singly linked list, we additionally have a dummy tail node in the doubly linked list.

```
// Doubly-linked node type
struct dnode {
     char data;
     struct dnode *next;
     struct dnode *prev;
};
typedef struct dnode dnode;

// Doubly-linked List type
struct dlist {
     // Initialize head and tail to dummy nodes in create()
     dnode *head;
```

```
      dnode *tail;
 };
typedef struct dlist dlist;
```

7. Update doubly linked-list functions

By including the dummy tail node, the operations on a doubly linked list become slightly different than those on singly linked list. Thus additional function signatures for operations on doubly linked list have been included in `list.h` and `list.c`. Some of these function bodies have been partly or completely filled out. You will fill out the remaining functionality as specified in the code.

- `dlist_create`: initialize the fields in the list structure. This function has already been written for you. You'll see now that both a dummy head node and a dummy tail node are created and inserted into the list.
- `dlist_is_empty`: check whether the list is empty. Note that since a dummy head node is being used, the check for whether the list is empty must change slightly.
- `dlist_insert`: insert an element into the list at the specified location.
- `dlist_deletion`: delete an element from the list at the specified location.
- `dlist_getval`: return the value of the data field for the element at the specified location.
- `dlist_size`: return the size of the linked list.
- `dlist_print`: print all of the elements in the list. 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. You will now see the driver has options to work with a singly linked list or a doubly linked list.