ESC190H1S Winter 2022: Lab 3
Linked Lists (and Stacks)

TA evaluations in practicals: February 15th/16th, 2022
Code submission due: February 17th, 2022 at 11:59pm

Overview

In Lab 2, we reviewed pointers and memory. In this lab, we will practice:

- Working with pointers, double pointers, structs
- Using a linked list (to implement a stack)
- Inserting, indexing, deleting, and sorting nodes for a linked list
- Thinking critically about ways to organize data

Lab Grading

The following three files have been provided for this lab (only submit `lab3.c` for grading):

- `lab3.c`, which will contain function definitions
- `test3.c`, which will contain code for testing functions
- `lab3.h`, which will contain function declarations - do not modify this file!

Ensure the filenames and function names are identical to those specified in this lab handout and use Valgrind to ensure there is no memory mismanagement by your code. Define any helper functions needed in the `lab3.c` file. There are some short answer questions included in this document to discuss with your TA during the lab evaluation session.

Part 1. Virtual TA Appreciation Party

Saima and Cindy want to show their appreciation for all the hard work the ESC190 TAs have been putting in this term, fixing IDE problems left and right. Saima suggests planning a little virtual party for the TAs with no IDEs in sight. Cindy thinks that's a wonderful idea and a great opportunity to practice using linked lists in C. Cindy has asked all the TAs to send her their requests for what items to purchase for the party, which are to be stored in a stack data structure to deal with closer to the day of the party.

Task: Let's write some functions to implement a stack data structure using a linked list. We will use the following node definition, provided in `lab3.h`:

```
struct party_node{
        char* item; // name of the party item
        double price; // price of the item
        char* ta; // name of TA requesting the item
        struct party_node *next; //pointer to the next node
};
```

Note that `item` and `ta` should point to dynamically allocated strings.

Recall that for a stack data structure, items are added to and removed from the stack in "last in, first out" (LIFO) order. In our linked list implementation of a stack, we will insert and delete from the front of the list to follow LIFO order. This means that the most recently added item will always be found at the front of the list.

Question: Is LIFO a good method for handling the party requests? Why or why not? It may be easier to answer this question after completing the functions described below.

In the `lab3.c` file, complete the following function definitions to add and remove requests from the linked list that stores the TAs' party item requests:

```
int add_request(struct party_node **head, char *item,
        double price, char *ta);
```

This function will create a `party_node` struct containing a TA's party request and add it to the linked list of requests - unless the request is for an "IDE". For LIFO order, the new node should be added to the front of the list. The integer returned will indicate if the request was added to the list.

Inputs:

- `head` is a double pointer to the first node in the linked list.
- `item` is a string containing the requested party item.
- `price` is the estimated price of the party item.
- `ta` is a string containing the name of the TA. requesting the party item.

Output:

- Allocate memory for the new node, as well as for the `item` and `ta` fields of the new node - unless the requested item is an"IDE". Fill in the data fields of the new node with the provided inputs, and add it to the front of the list. Return `0` to indicate that the request was added.
- If the requested `item` is an "IDE", then do not create a new node (do not allocate any memory) and do not add this request to the list. In this case, just return `-1` to indicate the request was not added.

```
void remove_request(struct party_node *head);
```

In case a party item request was added by mistake, this function will remove the most recently added `party_node` from the linked list of requests. For LIFO order, the most recent node is the node at the front of the list.

Input: `head` is a double pointer to the first node in the linked list.

Output: Removes the node that was most recently added to the list, freeing any memory allocated for that node (and its data fields).

The `print_list(..)` function, which prints all the item requests in a linked list, has been provided for you in `lab3.c` to help visualize the changes to the stack of requests as you test your code. Some testing code has been provided for you in `test3.c`, but you should do additional testing to convince yourself that your code works.

Part 2. Go Big or Go Home

Cindy has gathered and stored all the TAs' party item requests. As she is leaving to go shopping, Saima reminds her that there is a strict budget for the party and they might not be able to afford everything the TAs asked for. To impress the TAs and stay on budget, Cindy decides to sort the requests by price and prioritize purchasing the more expensive items. Note that this is just an imaginary lab scenario - you do not need to make big purchases to impress people in real life.

Task: Open your `lab3.c` file and complete the following function definitions to create a sorted linked list and finalize the shopping list for the party:

```
void make_sorted(struct party_node **head);
```

This function will use an unsorted linked list of party requests to generate a new sorted linked list where the `party_node`s are sorted by the `price` data field of each `party_node`.
You may generate a sorted linked list by adding all the TA party item requests from the original unsorted list to a new sorted linked list. Nodes in the sorted list should be sorted in descending order, from highest `price` at the front of the list to `lowest` price at the end of the list. Assume no item requests will have the same price. Free all the memory allocated for the original unsorted list once the new sorted list is constructed.
Optional: If you really want to impress your TA, you can attempt to sort the original list in-place so that no allocation or free-ing is necessary!
Input: `head` is a double pointer to the first node in an unsorted linked list.
Output: A sorted linked list is generated (either out-of-place or in-place) and `*head` points to the first node of the sorted list.

```
double finalize_list(struct party_node **head, double budget);
```

This function will trim the sorted linked list to generate a finalized shopping list that Cindy can use to stay on budget. We will prioritize the more expensive items in the following way:

- Starting from the party item request at the front of the list (the most expensive), check if we would exceed the budget by purchasing this item.
- If we would be over budget, remove this item request from the list (free-ing all the memory associated with the request).
- If we would not be over budget, then leave the item request in the list and check the next item request in the list.

Note that you will need to keep track of the remaining budget as you designate items to stay in the list so Cindy knows how much of the budget will be leftover.

Input:

- `head` is a double pointer to the first node in a sorted linked list.
- `budget` is the budget of available funds for purchasing party items.

Output: `*head` points to the first node of the finalized shopping linked list, i.e. the sorted linked list after `party_nodes`, representing item requests, may have been removed to stay on budget. If no item requests are left in the linked list after trimming, then `*head` should point to `NULL`. Return the remaining budget available after purchasing all the finalized party items as a `double` value.

Some testing code has been provided for you in `test3.c`, but you should do additional testing to convince yourself your code works.

Question: What are other ways to sort the data and trim the list? What are the advantages and disadvantages of these alternatives compared to the approaches used above?