Dylan Dunsheath NJIT Homework 6 Description CS 288

**Problem 1**

This program was a C Program that demonstrates the basic operations on a singly linked list, more specifically, initlalizing the data to each node, creating all the nodes, adding to the end of the linked list, printing, and reversing (which is the goal of the problem). The first step that is required for creating a linked list is constructing a structure (in my case, I called it node) that holds an integer (called data) and a pointer called next that points to the next node in a linked list. Since a linked list contains a head node (or a struct node POINTER that points to the head) we are required to initially initialize it as NULL or empty, since the pointer isn’t pointing to anything in the list (and let the program know that we need a pointer that points to the head). There is also three function prototypes: insertNewValue (which takes an int param – adds to end of list), display(); which displays the list, and displayRev; which displays the list in reverse order.

In the main function, we have two distinct variables: n; which is the value that the user is entering to add to the list and a integer called ‘repeat’ that continuously loops until the user hits the EOF (End of File) by Ctrl + D. If the user enters an integer, it adds it to the end of the linked list with the insertNewValue(n) function. Otherwise it breaks out of the while loop that we are in. We then call the display function to potentially display the list (or a message if it is empty). Finally, in the main function, we check if head is null. If it is, the only thing that is outputted to the screen is the given message that the list is empty. Otherwise, we display the message that the next list we are printing out is the Reversed List and then the actual list itself.

Within the insertNewValue function, that takes the inputted value from the user as a parameter, we use this data to insert a new node into the list: where we use two pointers: newNode that dynamically allocates memory for each node and does the following to assign data to each node: newNode ->data = item (where item is our passed parameter) and findLast to find the last node in our list. After we assign memory for the node using malloc(), we check if each node has a value. If it’s not NULL, we get the data for each node as previously mentioned with the newNode -> data – item and check where it is going to be placed in the list. If the list is empty, we simply put the node as the head node and make the pointer that is pointing to the NEXT part of the list is null. If it is not empty (can only be empty or not empty in this case), we use findLast that is initially pointing to the head of the list and iterate until the next is null and assign the next node of the CURRENT last node point to this newNode (ie: if the current node we are on is 5, which is the last node in the list and we have a new node that holds 6, 5’s next will be 6). The newNode is then pointing to null.

The display function was really simple. Within this part of the program, I initialized a pointer called iterator that will (as it says) iterate through the list (where the iterator is pointing to the head initially). If the iterator is null, that means the head is also null so we print that the list is empty. Otherwise, we run a while loop that continuously runs while iterator is NOT null (!= NULL) where it outputs “Data = [value here]” and we get the data by iterator -> data. We then need to go to the next node in list.

The displayRev is the function that displays the elements/values of the list in reverse order and is implemented with recursion due to its simplicity. In this function, it recursively calls itself with the next node until it reaches the end of the list (where the base case is the head is null).During the unwinding of the recursion, we print the data for each node. The recursive call is done by doing displayRev(head -> next);. Here is a sample of what happens:

Let’s say we have a list: 5 -> 8 -> 12 -> 3 -> NULL

Inside this displayRev function, we are making recursive calls with the next node each time until we reach the end of the list. So:

* displayRev(5 -> next) gets 8
* displayRev(8 -> next) gets 12
* displayRev(12 -> next) gets 3
* displayRev(3 -> next) gets NULL (end of list)

When head is NULL, it immediately returns without further recursion (we reach the base case). We then start printing the data in the reverse order from the last to the first node

**Problem 2**

This program involved sorting a linked list in ascending order via bubble sorting by swapping nodes, not data. The program was fairly easy but it can definitely be pretty difficult to some; as I can tell why they may think so. To start off, I declared a struct called node (similar to problem #1) that has the same fields: int data and struct node \*next. This allows a linked list to have a integer and a pointer to the next node. Similarly, I also declared a head node that is initially null (since the list is currently empty).

Now that we enter the main function of our program, I also made two other struct node pointers (pointers for our linked list): current which is for new nodes and temp which iterates over our list; which are both set to null. There is also a variable called input that takes integer input from the user and scans it until we reach the End of File (similar to #1). For every node, it mallocs (dynamically allocates) the amount required for each individual node based on the structure we previously defined outside the main function. It will then initialize the data for the current node based on input (current -> data = input) and makes the pointer also point to null for all the values inputted. Based on whether the list is empty or not, it will either add the current node as the head or continuously traverse the nodes until we reach the end of the list; and add the new node to the end of our linked list. When we are finished, we break out of the input portion. One of the important steps was to check if head was NULL still, before outputting the data that the user entered and the sorted list, as there would be no point in also outputting the “Data entered in the list are:” and “Sorted Linked List:” If it is not empty, we traverse the linked list and print the data found in each node initially (the initial list). We then perform bubble sort on the linked list to put the elements in ascending order. I used a flag called swap which is initialized to 1 to start the sorting (while (0) will not run). Current, prev and next are used to help manipulate the list: where current is initialized to the head of the list, prev is for the previous node, and next is the next of the current node.

Essentially, I used a while loop that will continue until the list is fully sorted (next is null). Current and next are used to compare adjacent nodes and perform swaps as needed while next is not null. If the current data is greater than next data (or if the previous node is greater than the adjacent node), a swap is required. We then set the swap to 1. In this case, if prev is NOT null, update prev-> next to point to next (prev goes to where next currently is). Otherwise, update head to point to next (change in head of list). Current -> next = next -> next is where I redirect the next pointer of current to go to the followed following the node that’s referenced by next. This ensures that nodes remain connected properly after a swap. I also update next -> next to point to current and update the prev and next pointers for the next iteration if required. Otherwise if the current data is <= next data, prev is updated to point to current, current is updated to move to next, and next is updated to point to next -> next.

Afterwards, I output the sorted list.