Linked Lists

Tuesday, September 20, 2016

Contents:

Linked List Visualization (Code)

Attachment: ***make-list.c (https://webcourses.ucf.edu/courses/1222520/files/57593784/download) (https://webcourses.ucf.edu/courses/1222520/files/57593784/download) (https://webcourses.ucf.edu/courses/1222520/files/57593784/download)***

We started class today with a discussion of the anatomy of a linked list. I ran a program that generated a linked list diagram, where the number of nodes in the diagram was based on command line input. I've attached the code here in case you're interested in checking it out, but it's not strictly necessary. It's just a bunch of printf() magic. (See comments on how to run this at the command line.)

Comparative Analysis: Arrays and Linked Lists

Attachment: ***linked-list-notes.txt (https://webcourses.ucf.edu/courses/1222520/files/57593779/download) (https://webcourses.ucf.edu/courses/1222520/files/57593779/download) (https://webcourses.ucf.edu/courses/1222520/files/57593779/download)***

We then examining key differences between arrays and linked lists. For my notes on this, see the attachment above.

Linked List Functions (Code)

Attachment: ***linked-lists.c (https://webcourses.ucf.edu/courses/1222520/files/57593782/download) (https://webcourses.ucf.edu/courses/1222520/files/57593782/download) (https://webcourses.ucf.edu/courses/1222520/files/57593782/download)***

We then discussed the basic structure of a linked list in code, messed with pointers to print list contents, and implemented a few linked list manipulation functions to create nodes, add nodes to our linked lists, and print out our lists, both iteratively and recursively. (See attachment above.)

Tail Pointers

We saw in class that we can insert a new node at the head of a list in O(1) time, but inserting at the tail of a list takes O(n) time. One way we can get around that slow runtime for tail insertion is to maintain a pointer to the tail of a list (in addition to a head pointer). If we have a pointer to the tail of a list, then tail insertion is an easy-peasy, O(1) operation:

```
// Plunk something at the end of the linked list.
tail->next = create_node(data);

// Move the tail pointer forward to point to the new node
// at the end of the list.
tail = tail->next;
```

Of course, the above code does not work if you have an empty list. In that case, tail would be NULL, and attempting to access tail->next would cause a segmentation fault. There are a few details to work out here if we want to write a robust tail_insert() function, which we'll discuss next week.

What's next?

On Thursday, we have Exam #1.

Next week, we'll be covering two new data structures: stacks and queues. We'll discuss how to implement them with arrays and linked lists. I'll also try to sneak some additional linked list exercises into my lectures over the next week or two. If time permits, we'll talk briefly about doubly linked lists and circularly linked lists.

Practice Exam Problems

1. Try implementing the functions shown in class on your own: node creation, insertion at the end of a linked list, insertion at the head of a linked list, and a list printing function.

2. (*Covered in class.*) Write a recursive *printList()* function.

3. (*Covered in class.*) Write a recursive *tailInsert()* function.

4. (*Covered in class.*) Write a function that inserts nodes at the beginning of the linked list.

5. (*Covered in class.*) Write a recursive function that prints a linked list in reverse order. The function signature is: *void printReverse(node *head);*

6. (*Covered in class.*) Write a recursive *destroyList()* function that frees all the nodes in a linked list.

7. Now implement *destroyList()* iteratively.

8. Write a *deleteHead()* function that deletes the head of a linked list and returns the new head of the list.

9. Write a function that deletes the *n*th element from a linked list. If the linked list doesn't even have *n* nodes, don't delete any of them. The function signature is: *node *deleteNth(node *head, int n).* Try implementing the function iteratively and recursively. (In terms of how to interpret *n*, you can start counting your nodes from zero or one; your choice.)

10. Write a function that deletes every other element in a linked list. (Try writing it both ways: one where it starts deleting at the head of the list, and one where it starts deleting at the element just after the head of the list.) Can you write this both iteratively and recursively?

11. Write a function that deletes all even integers from a linked list.

12. Write a function that takes a sorted linked list and an element to be inserted into that linked list, and inserts the element in sorted order. The function signature is: *node *insertSorted(node *head, int n);*

13. Write a function that takes the tail of a linked list and inserts a new element at the tail. Ensure that the function returns the new tail of the list. How would you have to call this function from main() in order to ensure you update your head and tail pointers correctly? (In what case(s) do we need to update the head pointer when performing tail insertion?) The function signature is: *node *tailInsert(node *tail, int data);*

14. One of the problems with the first *insertNode()* function from today is that it requires us to call it using *head = insertNode(head, i)*. That's a bit dangerous, because we could forget the "*head =*" part very easily. Re-write the function so that it takes a pointer to head, thereby allowing it to directly modify the contents of *head* without any need for a return value. The function signature is: *void insertNode(node **head, int data)*. The function will be called using *insertNode(&head, i)*.

15. A doubly linked list is one in which each node points not just to the next node in the linked list, but also the previous node in the linked list. The struct definition is:

```
typedef struct node
{
   int data;
   struct node *next;
   struct node *prev;
} node;
```

Implement *tail_insert()*, *head_insert()*, *tail_delete()*, *head_delete()*, and *delete_Nth()* functions for doubly linked lists. Repeat these exercises with doubly linked lists in which you maintain a tail pointer. How does the tail pointer affect the runtimes of these functions? Are any of these functions more efficient for doubly linked lists with tail pointers than they are for singly linked lists with tail pointers?