EECS 211 Lab 5

Linked Lists + Review for the Midterm

Winter 2017

In this week's lab, we will be going over linked lists. Then, we will review for the exam. You'll notice that we have given you more practice programs than you can probably complete through in your section. We recommend finishing them outside of discussion as they are good practice to study for the exam! We've included a full set of test cases so to run - when a function is implemented correctly, it should pass all of the its tests. We organized the questions by general topic, so consider jumping around between the types of questions, or focusing on ones you feel weak at.

If you have any lingering questions during the lab, don't hesitate to ask your peer mentor!

Getting the code

Download the zip file from the course site:

http://users.eecs.northwestern.edu/~jesse/course/eecs211/lab/eecs211-lab05.zip

After you have downloaded the zip file onto your laptop, extract the zip file into its own folder. Make sure you keep track of which folder it's in! Next, open up CLion and Click on File –> Open Project, and click on the Lab 4 project that you just unzipped.

Once you open the project, try building the lab and then running the lab5 executable. You should see some output printed in your output subwindow. If you need a reminder on how to build and run code in CLion, consult lab 3/4 or ask your TA. Once this works, you're ready to start the lab!

Linked Lists

The general idea of a linked list is that there are nodes that have a data member to contain information about the node, as well as a pointer to the next node in the linked list. As you know from the homework, a common linked list implementation would look something like this:

```
struct ListNode
{
    int data;
    shared_ptr<ListNode> next;
};
```

Each node has pointer to the next node in the linked list, until one node's pointer is the null pointer, signifying the end of the linked list. When you write functions using linked lists, you generally are passed (by reference) in the node that is at the front of the linked list. From here, since the node is usually passed by reference, when you edit the front of the tree's pointers, the changes propagate back to the arguments to the original function call.

Note that in our implementation of linked lists, they have both an int for data and a Dog

Exam Review

Linked List Practice Questions

Now, let's write some functions for working on linked lists. Look at your Dog.h and ListNode.h files for struct definitions. The following function skeletons have been provided for you. Each function is described below, give them a try!

findDog

For this function, you are still in charge of a dog sanctuary. In ListNode.cpp, write a function *findDog* that takes in a pointer to a ListNode, and looks for the first ListNode that has the desired identification number. When you find the desired identification number, return the dog from that ListNode.

Hopefully you've been feeding them!

Just like in the homework, we are abbreviating writing out a pointer to a ListNode as just a List

removeHalf

In ListNode.cpp, write a function called *removeHalf* that removes every other element from a linked list. For example, 1 -> 2 -> 3 -> 4 -> nullptr would become 1 -> 3 -> nullptr. Remember to make sure the ListNode exists before accessing its data types!

This is important, as de-referencing a nullptr is going to break your program.

squareIDNumbers

In ListNode.cpp, write a function, *squareIDNumbers*, that squares the identification number of each node in the linked list.

toVector

In ListNode.cpp, write a function, *toVector*, that takes in a linked list, and puts every element of the list into a vector.

Challenge function: *swapDogs*

In ListNode.cpp, write a function, *swapDogs*, that takes in a linked list, and two indices, and swaps the Dogs located at those indices.

Challenge function: *reverseList*

If you're up for a challenge, write a function, *reverseList* that takes in the front of a linked list (not by reference), and returns a new list, which is the original list in reversed order.

One of the easier strategies could be modifying your *toVector* function first to be for ListNodes as opposed to Dogs, then using that vector to help you create a new list that is backwards. However, there are many viable ways of creating this function!

General Practice Questions

meanAge

In Dog.cpp, write the function called *meanAge* which goes through a vector of Dogs, and calculates the mean age of the dogs in your sanctuary.

swapTreats

In Dog.cpp, write the function *swapTreats*, which swaps the favorite treats of two dogs which are passed by reference into the function, after they probably had an altercation.

divisibleByAll

Create a function in lab5.cpp called *divisibleByAll* where you are passed two vectors, nums and divisors, where you have to return a new vector of only the ints in nums that are divisible by all of the numbers in divisors.

factorial

In lab5.cpp, write a function, *factorial* that takes an integer, and returns the factorial of that number using a loop.

Remember that factorial(n) means n * (n-1) * (n-2) * ... * 1, and that the factorial of 0 is 1

Challenge function: ***vectorizeInt***

In lab5.cpp, write a function called *vectorizeInt*. This function should take a positive integer, and put it's digits into a vector, such that its most significant digits are in the least significant indicies of the vector.

For example: Given the number 108, your vector you would return would have 1 in the 0th index, it would have 0 in the first index, and it would have 8 in the second index. Furthermore, this vector would be equivalent to the vector created from the following:

Coincidentally the number of years before the Cubs won the World Series again this year

```
vector<int> v;

v.push_back(1);
v.push_back(0);
v.push_back(8);

return v;
```

You need to generalize this for every number.