CS 15: LinkedLists Lab

Introduction

This week, we are working with linked lists and `valgrind`! Like arrays, linked lists are linear data structures which store elements.

An important aspect of this data structure is that the elements of a linked list are not stored together at a contiguous location in memory; instead, a linked list is a collection of `Node` objects, which are linked together with pointers. That is, each `Node`, in addition to containing some data, contains a pointer to another `Node`. This structure makes inserting into and removing from a linked list easier than with an array, because, rather than having to shift all of the list's elements around, we just have to create a new node and link it to the necessary existing nodes or, to remove, we link necesssary nodes together and delete the desired node. Details are discussed in lecture.

We will also begin to work with `valgrind`, a tool for debugging. It will become your best friend this semester. It can detect memory leaks, segfaults, and other errors in your program. It works by instrumenting the code of your program at runtime and then analyzing the data collected during execution to find errors. `valgrind` is particularly useful for finding memory-related errors, such as memory leaks and memory errors. A **memory leak** occurs when a program dynamically allocates memory for an object but then loses the ability to free that memory because it no longer has a reference to it. On the other hand, a **memory error** can occur when a program incorrectly accesses memory. This can include things like trying to read from or write to memory that the program should not have access to or accessing memory that has already been freed.

To use `valgrind`, you need to run your program with the `valgrind` command, followed by the name of the program you want to debug or profile. For example, to check for memory leaks in a program called `planet-driver`, you would run `valgrind ./planet-driver` (*hint hint*).

For the lab, we will be working with a linked list that stores `Planet` objects! A `Planet` object contains:

- `std:string name` - the name of planet
- `int distance_from_sun` - distance from the sun (in millions of miles).

The `Planet` class is written for you - you will be responsible for a couple of functions within the `LinkedList` class, including inserting at the front of the list, inserting at the back of the list, and the destructor.

Getting Started

1. You should have a `cs15` directory within your home directory.
   (a) If you don't, create it now with `mkdir cs15`, and move inside it with `cd cs15`.
2. Create a `lab2` directory inside `cs15` and move into it.
3. Copy the starter code: `cp /comp/15/files/lab_linkedlists/* .`
   (a) Make sure you can answer these questions:
      i. What does the `.` mean?
      ii. What does the `/*` mean?
4. Remember that a good policy when programming is to write a little, compile, test, repeat.

Part 0: Understanding the Starter Code

Whenever you are given starter code, it's important that you take a few minutes to review it! Some introduction is given below.

Planet Class

In this lab, the elements we are working with will be `Planet`s! As metioned above, a `Planet` object contains a name (a `std::string`) and a distance from the sun in millions of miles (an `int`). The `Planet` class is completely written for you. Take a quick look at both `Planet.h` and `Planet.cpp` before continuing on.

LinkedList class

Regarding the `LinkedList` class, the starter code provides a complete `LinkedList` header file (`LinkedList.h`), and a partially completed driver file `LinkedList.cpp`. It would be well worth your time to review these files. Start with the header file. Notice the `Node` struct definition within the `private` section.

```
...
private:
struct Node {
      Planet info;
      Node *next;
      std::string toString() { return info.toString(); };
   };
...
```

Recall that contents of a `struct` object are public by default. However, because this `struct` is defined within the `private` section of the `LinkedList` class, its fields are only accessible by way of `LinkedList` objects. In other words, the member functions of the `LinkedList` can directly access the member fields of a `Node`.

Also, take note of the `Node *front` object in the private member section of the `LinkedList` class. This pointer will point to the first `Node` in the list.

As for the implementation file, take a quick look over it. Some functions (like `toString`) are defined for you - you would be wise to review and make use of these functions to help test and write your own code!

Driver file

Your starter code also includes a partially completed driver file - `planet-driver.cpp`. Reviewing the function headers of the already completed functions should be enough for you to understand what they do.

The Lab

In this assignment, we will create two Linked Lists of `Planet` objects - one list will be created by pushing the `Planets` to the front; the other will be created by pushing the `Planets` to the back of the list.

Each `Planet` has a name and a distance value, which represents its approximate distance from the sun (in millions of miles). When you run the program, it first asks for the number of `Planets` to be entered. Then, for that number, it asks for the user to type in the name of a `Planet`, along with its associated distance from the sun.

As each `Planet` information is entered, the program creates a `Planet` object. All of the entered `Planets` are stored in an array named `planet_array`.

Compiling and Running

For this lab, we will use `make` and an associated `Makefile` to compile and link your program. You don't need to know how to write `Makefile` just yet, just how to use them to build a program:

- **To compile:** type `make planet-driver`
- **To run:** type `./planet-driver`

Note that this is different from how you'd run `unit_test`. If you would like to use `unit_test` to test your `LinkedList` class, that's great! The `Makefile` is set up to support this. Just build your `unit_tests.h` testing file and run `unit_test`. If you do decide to test your code with `unit_test`, remember that the driver file provided won't run; thus, we suggest testing your code both with `unit_test` and with the driver.

Valgrind

To run `planet-driver` with `valgrind`, just type: `valgrind ./planet-driver`

Part 1: Inserting at the Front

As you have seen in class, for simple linked lists, it's easiest to insert a new element at the front of the list because it's fast and doesn't require keeping track of the back of the list. Your first task is to implement the `pushAtFront()` function in `LinkedList.cpp` and then to update `planet-driver.cpp` such that the `Planet` objects are pushed to the front list.

Note: you do **not** need to create the `LinkedList` class from scratch! Use the one that exists in the starter code. If you have not already, please read the **Part 0** section of this document before continuing.

When you're done implementing `pushAtFront()` and modifying the driver, be sure to compile and run the program. The output of your program should look like this:

```
Please enter the number of planets: 3
Enter: planet_name planet_distance
mars 4

Enter: planet_name planet_distance
earth 3

Enter: planet_name planet_distance
mercury 1

Part 1: PushAtFront list of planets
LinkedList of size 3
---------------------
mercury: 1
earth: 3
mars: 4
```

Part 2: Inserting at the Back

We will now insert the `Planet` objects directly at the back of the list. Open the file `LinkedList.cpp` and look for the `pushAtBack()` function. Fill in the implementation of this function, then update the driver file to correctly push the `Planets` to the back of the list.

When you're done, the output of your program should look something like this:

```
Please enter the number of planets: 3
Enter: planet_name planet_distance
mars 4

Enter: planet_name planet_distance
earth 3

Enter: planet_name planet_distance
mercury 1

Part 1: PushAtFront list of planets
LinkedList of size 3
---------------------
mercury: 1
earth: 3
mars: 4

Part 2: pushAtBack list of planets
LinkedList of size 3
```

```
--------------------
mars: 4
earth: 3
mercury: 1
```

Part 3: Valgrind / Destructor

At this point, you should run `valgrind` on your code! You might be surprised to find some memory leaks/errors! To fix these, implement the `destructor` of `LinkedList`. When finished, make sure there are no more leaks/errors with `valgrind`.

JFFE: Reversing the List

This is a great task to do to thoroughly test your knowledge of Linked Lists! Your job here is to implement the reverse function of the `LinkedList`.

Part 4: Finishing up the Programming

For this assignment, you can modify `LinkedList.h` to add additional private member functions. However, you should keep the list as singly-linked, with no tail pointer. Your changes must not change the interface or break any of the other code in the lab.

You should understand all the code in this lab. Read the code that you are given. Do you have questions? Ask! You should be able to do any of this, for example, on an exam.

Part 4: What's in a README?

For CS 15, the content of your `README` should be in **plain text**, with all lines less than 80 columns wide. It should **not** be a Word file or other fancy thing. Plain text, hard-coded line breaks. Readable on virtually any system in the world without any special software. Including a `README` file with every project (and even for sections within a large project) is standard industry practice.

You can format your `README` however you like, but it should be well-organized and readable. Include the following sections, with a heading for each one:

1. The title of the assignment and the author's name (that's you!).
2. The purpose of the program.
3. Acknowledgements of any one who helped you, including TAs, professors, friends, web sites, etc.. This is consistent with the Tufts rules on academic integrity and is also standard practice in academic and many professional settings.
4. A list of files provided with a short description of each including its purpose.
5. Instructions for how to compile, link, and run your program.

Sections 1, 3, and 5 are self-explanatory, we hope. You can always ask if you are unsure of anything.

Section 2 should be a short description of the overall goal of the program, which may not necessarily mention the data structure(s) used. For example, in a linked list assignment, the purpose should **not** say "The purpose of this program is to implement a linked list." A better description would be **The purpose of this program is to support the manipulation of characters within a list.**

The description of files in section 4 should be useful, not just a list of files. It should be clear from the comment why the file is relevant to the overarching problem and how it fits into the overall architecture. For example,

CharLinkedList.cpp/h: class that implements linked lists of characters. Includes functions to manipulate the list *e.g., addingandremovingitems.*

Each of the sections should be clearly delineated and begin with a **section heading** (not just a letter/number) describing the content of the section.

For homeworks or projects, you will insert two additional sections and perhaps an optional section:

6. An outline of the data structures and algorithms that you used. Given that this is a data structures class, you should always discuss any significant data structures you used and justify why you used them. Specifically for this assignment, you should discuss the features of linked lists, as well as major advantages and major disadvantages of utilizing a linked list in this assignment. The algorithm overview may not be relevant depending on the assignment.

7. Details and an explanation of how you tested the various parts of assignment and the program as a whole. You may reference testing files that you submitted to aid in your explanation.

8. (optional) Other information, including what works and what doesn't. Tell the truth! This section includes anything you want the grader to know. For example, you could say that you implemented the interaction between two classes in a way that violates good abstraction and that you are aware of this, but unfortunately you realized this late and did not have time to fix it. Ideally, describe how you would fix it. If you have other known bugs, please report them. Also, explain any way in which you exceeded the requirements of the assignment and how we can see this. The grader will read whatever information you give and may take it into account.

Section 6 should be well thought out and well-reasoned. You should demonstrate that you have a solid understanding of the strengths and weaknesses of the data structures you chose and how they lend themselves to or detract from the overall project. Ideally, for more complicated programs, section 6 would detail the program's **architecture**: how the modules of the program interact with each other. This is particularly useful for projects with lots of classes or complex intereactions. For anyone trying to understand the code, it won't be immediately intuitive how modules interact or how the main program uses the various modules.

Section 7 should be comprehensive. If you write "I wrote functions that tested every function", that's not enough. Hopefully, you will find some way of saying "I tested as I went and this is how I did it."

Here is a list of the things an ideal Section 7 would cover:

- Process of testing (e. g., unit tested each function after I wrote it)

- Bugs encountered while testing

- Mention of testing on inputs of different sizes (e. g., I tested CharLinkedList using a an empty file, a single character file, a 5-character file, and a file of 10,000 words)

- Good description of base cases and edge cases considered while testing. Because some cases crash or throw exceptions, students often delete them, but don't do that. Please comment them out so we can see them and describe them here. For example, you might test that accessing a list out of bounds crashes or throws an exception as expected. If it

> crashes as expected, comment it out in the code and put in a comment like "this test crashed on input X as expected." We are looking for descriptive evidence of a planned, methodical approach to testing.

That's it! Please use the draft `README` in the starter files as a jumping off point.

Part 5: Submitting Your Lab

You will need to submit the following files:

```
planet-driver.cpp
unit_tests.h
LinkedList.cpp
LinkedList.h
Planet.cpp
Planet.h
README
Makefile
```

You must submit them using Gradescope to the assignment `lab_linkedlists`. Submit these files directly, do not try to submit them in a folder.

A Note on Lab Grading

Labs are for hands-on practice in a supervised setting so you can develop your skill. They are designed to be low-pressure and fun. Come to lab, do your best, submit your work at the end of the period (your best effort given the time), and you will get a high score. If, for some reason, you miss your lab session, you can go to another one (provided there is space). If you cannot attend any session for a whole week, do it in your own tie so you will gain the skills. See the syllabus for course policies on labs.