CPSC 2120/2121
Lab 02

Sorted Linked List
Due date/time are on the CPSC 2121 Canvas website

Learning Objectives

- To improve our ability to implement algorithms and data structures in C++
- To implement a data structure using a class template in C++
- To use a simple data structure (a node) as a building block to construct a more complicated data structure (a linked list)
- To become proficient in fundamental data structures used throughout computer science
- To improve our analytical skills while gaining familiarity with mathematical tools used in the analysis of algorithms
- To implement a data structure to understand its intricacies

Problem / Exercise

For this lab, you will implement a sorted linked list data structure built by linking dynamically allocated nodes together, where each node consists of a value and a pointer as defined in the provided class Node.h. The linked list for this lab will have the following property: the nodes in the list must always be sorted in descending order (based on the values of the nodes in the list).

There are many data structures that have a sorted property, and they maintain their sorted property by ensuring that when something new is inserted into the data structure, then it is inserted into the correct location to maintain its sorted property. It is good to note that these data structures that have a sorted property do not typically call a sorting function to maintain its sorted property; instead, they use an insert function that places each new object into its correct position. Your linked list will also have an insert function that will be responsible for inserting a new node into the list while maintaining the list's property of being sorted in descending order. Also, your linked list cannot call a sorting function when inserting a new node. Why? Imagine inserting many nodes into your list, and if your insert function called a sorting function each time, then the run time complexity would be much higher than it needs to be. For your linked list, the (worst-case) run time complexity of inserting a new node into your linked list should be $O(n)$, where n is the size of the list.

Before you begin coding for this lab, it is recommended that you read Node.h, List.h, lab02.cpp (a testing program), Makefile, and the examples found in this document to understand how the functions should operate. Once you have an idea of what to do, then implement the functions in List.h, and follow the additional requirements below.

Additional Requirements

- You must implement all of the functions in List.h after the interface is defined, and follow the instructions stated in the comments.
- Your class should allow nodes with the same values to be inserted into the list as shown in some of the examples.
- You may assume we will test your class and its functions using valid test cases, and a small, valid set of test cases can be found in our lab02.cpp file. Also, you may assume the class type T we test your list with will consist of comparable types (int, double, float, char, string, etc.) where comparison operations, $\{<, <=, >=, >, ==, !=\}$, are well defined.

- You CANNOT modify any function prototypes (function declarations); otherwise, your program may not compile with our tester program when we grade it, which may result in a grade of zero on the assignment.
- Your List class must use the provided Node class. You CANNOT use data structures found in the C++ STL for your List class, which includes, vector, stack, list, queue, set, array, etc. The purpose of this lab is to implement a linked list class, not to use the already implemented list class in the STL. The use of any data structures in C++ STL will result in a grade of 0 on this lab. However, you may use the string class (used in the print function in List.h). Also, you may use arrays for testing purposes in lab02.cpp, but you cannot use arrays in your list class.
- As aforementioned, your List class cannot use any built-in sorting functions. The use of any built-in sorting functions will result in a zero on this assignment.
- Your List class must compile and run with unmodified versions of Node.h, lab02.cpp, and Makefile to produce the same output as our examples. If your class does not, then you'll need to fix it. After you get your class working with our test program, then you should create several examples on your own to further test your program to make sure it works with any valid set of test cases.

Examples

Your C++ source code should be in a file called `List.h`, and it should be compiled (along with our provided `lab02.cpp` and `Node.h` files) into an executable called `lab02.out` using our provided `Makefile`. The output of our testing program in `lab02.cpp` with your List class implementation must look exactly like the examples below when run on the command line on our Unix machines.

```
./lab02.out
list1: size = 0, values =
list1: size = 1, values = -2
list1: size = 2, values = 4 -2
list1: size = 3, values = 4 -2 -8
list1: size = 4, values = 16 4 -2 -8
list1: size = 5, values = 16 4 -2 -8 -32
list1: size = 6, values = 64 16 4 -2 -8 -32
list1: size = 7, values = 64 16 4 -2 -8 -32 -128
list1: size = 8, values = 256 64 16 4 -2 -8 -32 -128
list1: size = 9, values = 256 64 16 4 -2 -8 -32 -128 -512
list1: size = 10, values = 1024 256 64 16 4 -2 -8 -32 -128 -512
list2: size = 1, values = Sisko
list2: size = 2, values = Sisko Janeway
list2: size = 3, values = Sisko Picard Janeway
list2: size = 4, values = Sisko Picard Kirk Janeway
list2: size = 5, values = Zoey Sisko Picard Kirk Janeway
list2: size = 6, values = Zoey Sisko Picard Kirk Janeway Frodo
Worf is not in list2 :(
list3: size = 1, values = z
list3: size = 2, values = z p
list3: size = 3, values = z r p
list3: size = 4, values = z r p p
list3: size = 5, values = z r p p d
list3: size = 6, values = z r p p d a
list3: size = 7, values = z r p p h d a
list3: size = 8, values = z r q p p h d a
list3: size = 9, values = z r q p p o h d a
r is in list3
```

Source Code Requirements

For this lab, you should place a copy of `List.h`, `Node.h`, `lab02.cpp`, and our provided `Makefile` on one of our Unix machines (in the same directory), and complete the `List.h` class such that it will compile and work correctly when used with other testing programs. Here are some additional requirements.

- Fill in the comment at the top of your source file(s) with your name (first and last), the date of your submission, your lab section, and the assignment's name.
- All functions should be commented. Use inline documentation, as needed, to explain ambiguous, tricky parts, or important portions of your code.
- All source code must follow good programming style standards such as properly indenting source code and all variables, functions, and classes should be well-named.
- Your class must use dynamic memory allocation and deallocation properly, which means your class cannot contain a memory leak. You may use valgrind to check for memory leaks. Refer to the Unix manual and valgrind's online documentation for more information on valgrind.

Submission

Before the date/time stated on the CPSC 2121 Canvas webpage, you need to submit your code to our CPSC 2121 Canvas webpage under the correct lab assignment. Make sure to submit all of the following.

1. All source files required for this lab (List.h)

After you submit, always double check that the file(s) you submitted were the correct version. To double check, download the submitted file(s), put them on one of our Unix machines, and make sure they compile and run correctly.

Grading: 10 points

If your class does not compile on our Unix machines (or your assignment was not submitted on time), then you'll receive a grade of 0 on this assignment. If your class uses a data structure that is part of the C++ STL (vector, list, queue, stack, etc.) or a built-in sorting function, then you'll receive a grade of 0 on this assignment. Otherwise, your class will be graded using the criteria below.

Your class works correctly with a program we will use to test your class	6 points
Proper variable names, documentation, and code organization	2 points
Your class uses dynamic memory allocation/deallocation properly (no memory leaks)	1 point
At least 3 distinct, passing test cases (not included ones provided) are submitted	1 point
Penalty for not following instructions (invalid I/O, etc.)	Penalty decided by grader

You must test, test, and retest your code to make sure it compiles and runs correctly and efficiently on our Unix machines with any given set of valid inputs. This means that you need to create many examples on your own (that are different than the aforementioned examples) to ensure you have a correctly working class. We will only test your program with valid sets of inputs.