Linked Lists and Merge Sort

Ordered Data Structures — Week 1 Project
University of Illinois

Introduction

In this assignment, we provide you with a templated **`LinkedList`** class, that is a doubly-linked list data structure. You'll implement some of the member functions for this class that perform insertions and create sorted copies of lists. In the process you'll see how to traverse a list with pointers, how to connect nodes in a doubly-linked list, and see how the classic "merge sort" and "merge" algorithms work.

C++ Environment Setup

If you are new to C++, we recommend checking out the first MOOC in this course sequence, which has more information on getting your C++ programming environment set up. In particular, we show how to set up AWS Cloud9 to do your coding entirely through your browser, but it's also possible to do this assignment on your own computer in Windows (with WSL Bash), in macOS (with some additional tools installed for the terminal), or in Linux natively. Cloud9 itself hosts a version of a Linux.

In the Week 1 Programming Assignment item on Coursera, where you found this PDF, there is a download link for the starter files: **`LinkedList_starter_files.zip`**. If you are using a web IDE like Cloud9, you should still download the zip file locally to your computer first, then upload it to your Cloud9 workspace. With the default Cloud9 workspace configuration, make sure you've updated the compiler first, by typing this command in the terminal:

```
sudo yum install -y gcc72 gcc72-c++
```

Now, in the Cloud9 interface, click **File > Upload Local Files**, and select your local copy of the starter zip file to upload it to your workspace. After that, you'll see that the zip file appears in the environment file listing on the left of the screen.

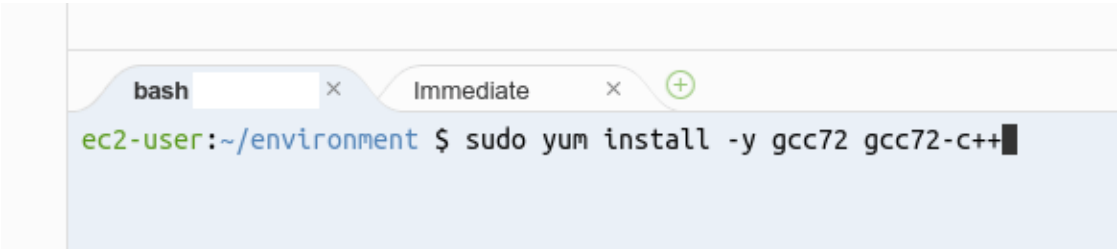

Fig. 1: Using a "yum" command in the Cloud9 terminal to update the compiler

Next, you'll want to extract the contents of the zip file to a new directory. Your terminal on Cloud9 already has some commands for this, **`zip`** and **`unzip`**. First, type **`ls`** in the terminal and make sure you see the zip file in the current directory listing. If not, then as described in the earlier readings, you should use the **`cd`** command to change directory until you see the file in the same directory. You can use the file listing on the left side of the window to help you see where you are. The **`pwd`** command ("print working directory") also reports where your terminal is currently browsing in the file system.

Once in the same directory as the zip file, enter the following command to extract the contents of the file, which contains a **`LinkedList`** subdirectory: (Be careful, because if you already have a subdirectory called **`LinkedList`**, this may overwrite files in that directory!)

```
unzip LinkedList_starter_files.zip
```

After you extract the code files, you can double-click them in the file list pane to see the editable document in the viewer.

Provided Files

You are provided with a Makefile, and so you can compile your code from the **`LinkedList`** subdirectory just by typing "**`make`**" (for the main program) and "**`make test`**" (for the testing suite). The file that you need to edit is called **`LinkedListExercises.h`**, but several other files have important clues about how to implement your solution. There are a lot of comments in the code, so you should try reading through the files.

About Templates in Header Files

You'll notice that much of the code for this assignment is contained in header files instead of cpp files. That's because *templated* C++ code generally needs to be written entirely in header files; when you use templates, the compiler generates specific versions of the templated code at compile time, depending on how you tried to apply the template in your code. In order for the compiler to use the template, it first needs to know about the entire template, which is usually done by putting all the relevant parts in the header. There are complicated tricks that programmers can use to move some parts of the template into cpp files for faster recompilation, but we'll overlook that for this assignment.

Files you should edit:

This time there's only one file you must edit, and the autograder will discard changes that you make to any other files. (You can add unit tests to the other files mentioned in the next section, but the autograder will discard those changes.)

- **`LinkedListExercises.h`**: There are two exercises to complete, which are described in this document and in the code comments of the file itself.

Files you should read:

Among the other code files provided, some of them are meant for you to read through, even though you don't need to edit them.

- **`LinkedList.h`**: You can see how the **`LinkedList`** class implementation is done. There are comments throughout the class declaration and member function definitions.
- **`LinkedListExercises.h`**: This file contains descriptions of the exercises you need to complete, but there are also a few hints in code comments.
- **`main.cpp`**: The main function that launches some examples and a few informal or experimental tests. You can see example usage of the provided code in how the informal tests are set up. These tests are separate from the unit tests you can build with **`make test`**; the Makefile is designed for you to compile and run the **`main`** and **`test`** programs separately, although they do run some similar tests on your code. You should try them both and study the output in the terminal.
- **`tests/week1_tests.cpp`**: This configures the **`test`** program you can compile, and contains unit tests in Catch library syntax. You don't need to learn how to use the Catch library syntax, but this file shows what assertions the **`test`** program will check on your code, what benchmarks are available, and the autograder on Coursera will also run the same correctness tests. (The autograder won't benchmark your solution, but a truly correct solution should achieve the desired running times on your own system.) If you are interested in Catch, the documentation is here: https://github.com/catchorg/Catch2

Other files:

There are some additional files in the provided package, although you don't need to look at them. They're part of the mechanical inner workings of the assignment. For example, there are files containing Makefile definitions for the GNU Make program to use when it launches the compiler, and there is the source code for the Catch unit testing library. You are welcome to look at this code, but we won't discuss how it works.

Compiling and Testing Your Code

To compile the code, you must use the terminal to enter the same directory where the **`Makefile`** is stored; it's in the same directory where **`main.cpp`** is located. As explained in the readings, use **`cd`** to change to the appropriate directory, use **`ls`** to view the file list in that directory (just to make sure that you're in the right place), and then type **`make`** to automatically start the compilation. If you need to clear out the files you've previously built, type **`make clean`**. If you encounter any warnings or errors during compilation, study the messages in the terminal carefully to figure out what might be wrong with your code.

If your compilation is successful, you'll get an executable file simply called **`main`** in the same directory as **`main.cpp`**. You can try running it by typing **`./main`** while you're in that directory.

As the compilation message suggests, you should also run the unit tests included in the test program. To compile it automatically, just type **`make test`** at the same prompt. If this compilation is also successful, you'll see a file named **`test`** appear in the directory. Then, you can run it by typing **`./test`** similarly to before.

To compile and run the main program:

```
make clean && make && ./main
```

It is faster to just do “`make`” and not perform “`make clean`” first, but “`make clean`” can help you resolve certain compilation problems sometimes.

To compile and run the test suite:

```
make clean && make test && ./test
```

For this assignment, you can run the same test suite that the autograder will use on Coursera.

To compile and run the benchmarks:

```
make clean && make test && ./test [.bench]
```

The benchmark test will compare the running times of some of the functions against lists of various sizes. If you tried to solve the problems in an incorrect way, yours may run much slower than the times suggested in the terminal output messages. (On the other hand, depending on what your version of the compiler does for optimizations on your processor, you might see one of the tests run twice as quickly as expected, or so on.)

Some of the implementations given to you in the starter files are not necessarily optimal. For example, for simplicity, the starter code creates or passes extra working copies of lists instead of editing lists in-place solely by rewiring the pointers linking the nodes together. (If you are bored, try redoing the assignment for maximum efficiency.)

To make a package of your files for submission:

```
make zip
```

This automatically bundles only those files that have been specified for you to edit. For this assignment, only `LinkedListExercises.h` will be collected.

The LinkedList Class

This project gives you a templated, doubly-linked list implementation called **`LinkedList`**. It can be used similarly to a double-ended queue or a stack: you can push or pop on both ends of the list, and there is constant-time access to the front and back items. The nodes are created on the heap and connected in a chain by next and prev pointers; nodes contain actual copies of data, not pointers or references to data as seen in some examples in the lectures. (However, you can make a list of pointers explicitly, or even a list of lists. Some of the member functions use these tricks to do work.)

Note that because this list type is doubly-linked, each node has both a “next” pointer to the following item, as well as a “prev” pointer to the preceding item. Therefore, you can traverse the list in both directions, but you also have to take extra care to manually adjust all the necessary pointers if you are inserting or removing a node manually.

The **`LinkedList`** class isn’t as thoroughly constructed as a more mature general-purpose container like you would find in a library like the Standard Template Library (STL) that is fundamental to modern C++, but this **`LinkedList`** does have passing similarities to the **`std::list`** type, including how some of the member functions are named. A robust C++ library is usually designed to completely hide its internal memory structure,

providing an interface of functions that allow users to operate on the object in an easy and safe way without knowing how exactly it is implemented; for example, many STL types implement “iterator” class types that can be used similarly to pointers to traverse a structure without having to actually manipulate pointers. You will learn more about that later. For now, you can perform most operations on the **`LinkedList`** type by either iterating over it directly using the node pointers that it exposes, or you can primarily treat **`LinkedList`** as a queue or stack, using its **`push`** and **`pop`** functions to cycle over it or to manipulate copies of the nodes. The provided code shows examples of both methods and you are largely free to approach the assignment how you like. (One exception to this is the exercise regarding **`insertOrdered`**, which asks you to avoid changing the address of most of the nodes in the list. In part, that requires you to adjust the existing pointers manually.)

Try to approach this week’s assignment without using any STL containers such as **`std::list`**, **`std::vector`**, **`std::queue`**, or **`std::stack`**. You can do it entirely using **`LinkedList`**! You are also encouraged to try to achieve the recommended time complexity for the exercise functions. We provide some notes on how to do that, and a benchmark suite along with the unit tests, which can help you gauge whether you’re on target.

Please take care because some of the **`LinkedList`** member functions modify the current list in-place, while other functions leave the original list unmodified and return a new, modified list. When functions do not modify an argument, the argument is marked **`const`** accordingly; entire functions are also tagged **`const`** when they do not modify the current class object itself (the instance which had called the function). Please see the notes in the provided code about “const correctness” as well for more discussion about using **`const`**.

Insertion Sort and Merge Sort

The provided code implements skeletons for several sorting algorithms. Insertion sort is a somewhat naive algorithm that sorts a list by inserting the original items, one at a time, into the correct position in a new list. In order to do this, it relies on an **`insertOrdered`** function that you will implement in the exercises. In the worst case, insertion sort has a running time of $O(n^2)$. This is because even if **`insertOrdered`** runs in linear time, any *i*th insertion may have to walk over a list of length *i*, resulting in the overall work:

$$\sum_{i=1}^{n} i \;=\; \frac{1}{2}\,n^2 + \frac{1}{2}\,n \;=\; O(n^2)$$

Merge sort is a classic sorting algorithm that actually has a desirable running time. By repeatedly splitting the list into halves and then “merging” the pieces back together in sorted order, it achieves a running time of O(*n* log *n*); to explain briefly, the process of recursively splitting data in half will naturally make a call stack O(log *n*) layers deep, and there is O(*n*) overhead work to be done at each of those levels. Merge sort can be implemented either iteratively or recursively, and the provided code shows how this might be done using our **`LinkedList`** type. Both versions use the **`merge`** function repeatedly, which you will implement in the exercises. (You really want it to merge in linear time!) There are some more notes about how merge sort works in the provided code comments, including a bit more justification of the correctness and the running time.

If you’re interested in more resources about the analysis of discrete algorithms, there are some free texts by several University of Illinois professors that you may want to bookmark (in particular, Prof. Erickson’s book is popular for its entertaining and approachable writing style, and many nice diagrams):

Prof. Margaret Fleck's *Building Blocks for Theoretical Computer Science*
http://mfleck.cs.illinois.edu/building-blocks/

Prof. Jeff Erickson's *Algorithms*
http://jeffe.cs.illinois.edu/teaching/algorithms/

Programming Objectives

Study the Code

Since we have provided you with several files with examples and complete classes and functions to use, it's always best to begin by reading through the code that you are given. There are comments throughout and many hints about how to do the exercises. Some of the member functions, examples, and tests show coding techniques that are directly applicable to the exercises.

Complete the Exercises

Your task for this project is to complete the two exercises located in the **`LinkedListExercises.h`** file. Be sure to use the **`main`** program as well as the **`test`** program (and the optional benchmarks) to verify that your solution works correctly.

Exercise 1: insertOrdered

The **`LinkedList`** member function **`insertOrdered`** assumes that the current contents of the list are already sorted in increasing order. The function takes as input a single new data item to insert to the current list. The new data item should be inserted to the list in the correct sorted position, so that you preserve the overall sorted state of the list.

For example, if your **`LinkedList<int>`** contains: [1, 2, 8, 9] and the input is 7, then the list should be updated to contain: [1, 2, 7, 8, 9]

To be more precise, a new node should be created on the heap, and it should be inserted in front of the earliest node in the list that contains a greater data element. If no such other node exists, then the new item should be placed at the end (the back of the list). Also, be sure to update the **`size_`** member of the list appropriately.

Your implementation of this function should not change the memory locations of the existing nodes. That is, you should not push or pop the existing elements of the list if it would change their address. (The member functions for push and pop will add a new node or delete one, so these operations would not leave the original nodes in place even if you recreated them with equivalent data.) You should use pointers to connect your new node at the correct insertion location, being sure to adjust the list's **`head_`** and **`tail_`** pointers if necessary, as well as any **`prev`** or **`next`** pointers of adjacent nodes in the list. Remember: **`LinkedList`** is a doubly-linked list. That means each node refers to both the previous item in the list, as well as the next.

A correct implementation of this function has O(n) time complexity for a list of length n. That is, in the worst case, you would traverse each element of the list some *constant* number of times during a single insertion operation—perhaps only once per element, for example.

Exercise 2: Linear-time Merge

The **`LinkedList`** member function **`merge`** is intended to perform the classic "merge" operation that is important to the merge sort algorithm. It combines two sorted lists into a single sorted list. This algorithm is intended to run in linear time (that is, O(*n*) where *n* is the total number of elements in the input lists), so it is not appropriate to simply concatenate the two lists and then apply some other sorting algorithm. Instead, the merge algorithm relies on the fact that the two input lists are *already* sorted separately, in order to create the final merged list in linear time.

One of the implied input lists is the **`LinkedList`** instance that is calling the function (**`*this`**), and the other input list is explicitly specified as the function argument "**`other`**". The function does not change either of the original lists directly, as the inputs are marked **`const`**. Instead, this function makes a new list containing the merged result, and it returns a copy of the new list. For example, one usage might be:

```
LinkedList<int> leftList;
// [... Add some sorted contents to leftList here. ...]
LinkedList<int> rightList;
// [... Add some sorted contents to rightList here. ...]
LinkedList<int> mergedList = leftList.merge(rightList);
```

You may assume that the two input lists have already been sorted. However, the lists may be empty, and they may contain repeated or overlapping elements. The lists may also have different lengths. For example, it's possible that these are the two input lists:

Left list: [1, 2, 2, 3, 5, 5, 5, 6]
Right list: [1, 3, 5, 7]

And the result of merging those two sorted lists will contain all of the same elements, including the correct number of any duplicates, in sorted order: [1, 1, 2, 2, 3, 3, 5, 5, 5, 5, 6, 7]

Because your implementation of this function operates on **`const`** inputs and returns a newly created list, you do not need to maintain the previous memory locations of any nodes as required in Exercise 1. You may need to make non-**`const`** "working copies" of the **`const`** input lists if you wish. You may approach this problem either iteratively or recursively, and you may use the member functions of the **`LinkedList`** class to make it simpler (such as **`push`** and **`pop`**), or you may edit the contents of lists explicitly by changing the pointers of a list or of its nodes (such as **`head_`**, **`tail_`**, **`next`**, and **`prev`**). Again, be sure that the **`size_`** member of the resulting list is updated correctly.

A correct implementation of this function has O(*n*) time complexity for a list of length *n*.

Submitting Your Work

When you're ready to hand in your work, remember that you can type this command in the main subdirectory:

`make zip`

This will automatically package the necessary files into a new zip file for submission. You can download the created zip file to your own computer from Cloud9 if necessary, and then submit the zip file on Coursera for autograding. (If you need any reminders about how to submit work on Coursera or use Cloud9, the first MOOC in this course sequence on Coursera has more detailed information about these topics in the lessons and homework assignments.)