Assignment Linked Lists: Implementation of Linked List API Features

COSC 2336: Data Structures and Algorithms

Summer 2021

Objectives

- Gaining more experience with the concept of an Abstract Data Type (ADT) using inheritance and the List abstract definition.
- Look in more detail at classes, inheritance and Object-Oriented programming in C++.
- Also more practice using template classes.
- Much more practice with dynamic memory memory allocation and pointers.
- Learning to define and use linked list structures for lists of items.
- More examples of algorithmic analysis and the benefits and trade offs of implementing list functions using fixed arrays and dynamic linked lists.

Description

With this assignment we begin moving our focus from algorithms and algorithmic analysis to data structures. Lists are a fundamental type of data structure, a way of organizing and accessing data in a computing system. If we have a `List` of items, we usually want to keep track of the items of the list, be able to iterate through the list items efficiently, be able to add items onto the list, sometimes only on the end, sometimes at the beginning, sometimes in the middle. We might want to search the list, etc. These are are examples of the `List` abstraction, or Abstract Data Type (ADT). The `List` ADT defines an interface for the things we would like to be able to do with a list of values.

However, the ADT is only the interface of how we might use a list data type. The `List` abstraction can be implemented in different ways. In our materials for this unit we have mainly focused on comparing the implementation of a `List` using a static array (contiguous block of memory), and as a Linked List of dynamically allocated nodes.

Our class textbooks both give examples of a `List` ADT, e.g. Malik 6th edition pg. 1075, and Shaffer 3.2 C++ edition, pg. 98. The `List` ADT we use here is a combination of these two defined abstractions.

We have been defining and refining an array based implementation of a `List` data type already for the past several assignments. Some of that past work has been reused in this assignment. We are also reviewing object-oriented inheritance in this assignment. The `List` data type has been broken up into multiple files. We use C++ inheritance to define an abstract base class. An abstract base class usually does not implement any functionality, it is solely used as a definition of an interface, or an abstract data type, if you will. The `List.hpp` file contains this abstract base class definition of our `List` abstraction. Most all of the member functions in this header file are virtual member functions, that is what the `= 0` after the member function declaration does. A virtual function is a common concept in object-oriented languages. Basically what it means here is that, this function is not implemented in this abstract base class. What is expected is that concrete classes will inherit from this abstract base class. Any function that is defined as a virtual function in the interface must get a concrete implementation in any derived classes. So again, the base class here does not define any particular implementation, it only defines an interface that concrete classes must implement if they are derived from this base class. The `List.hpp` header file defines the abstract data type of our `List` class, and since it is abstract, there is actually not corresponding `List.cpp` file with any implementation.

Instead, we will implement the `List` abstraction concretely as an array based implementation in `AList.[cpp|hpp]` and as a linked list based implementation in `LList.[cpp|hpp]`. Actually the array based implementation has been given to you, and you won't be modifying it. But you should look through it and compare it with our previous assignments. You should find that we are using a fixed array of `values`, that can be dynamically grown by doubling its size, as we introduced and you used in the previous assignment. We discussed the performance implications of

this a bit, especially with respect to appending items at the end or prepending to the beginning of the list. Adding items on the end is relatively fast, as long as there is room in the current array allocation to just add another item to the end of the array. Adding on the front is slower, because we must first shift all items up 1 index in array, before we can prepend the item, $\mathcal{O}(n)$. We followed the Malik ADT more for these functions, so they are now called `insertFront()` and `insertBack()`, and there are corresponding methods to access the front or back item named `getFront()` and `getBack()` respectively. We also left in the overloaded indexing operator as part of the `List` ADT, so that you could get a reference to an item at a particular index of the array (and even change the value of the item). This operation is $\mathcal{O}(1)$ constant time for the array based implementation. But when you implement it for the linked list based implementation, it takes a bit more work.

The `List` abstraction and concrete implementations are all template classes. So the `List` data type you will be implementing is a container that can handle values of different types depending on which template data type it is instantiated with in a program. This makes the syntax of the member functions a bit crufty, but hopefully you familiarized yourself with template classes and template member functions and are comfortable enough that you can understand this `List` code and implement some new template member functions for the linked list concrete implementation.

Our primary goal in this assignment is to learn about linked list implementations, and using and managing linked lists using C/C++ dynamic memory allocation and pointers. Several of the defined `List` ADT virtual functions have been left unimplemented in the `LList` linked list concrete implementation class, and it will be your job to implement them.

Overview and Setup

For this assignment you will be given the following files that you will be using and adding code to for this assignment.

File Name	Description
`src/test-AList.cpp`	Unit tests of the array based `AList` implementation of the `List` API
`src/test-LList.cpp`	Unit tests of the link list based `LList` implementation of the `List` API
`include/List.hpp`	Header file of the ADT base `List` class that defines the `List` interface / abstraction
`include/AList.hpp`	Header file of the concrete array based implementation of the `List` abstract data type
`include/LList.hpp`	Header file of the concrete linked list based implementation of the `List` abstract data type
`include/Node.hpp`	Header file of the Node structure used by the linked list implementation
`src/List.cpp`	Implementation file of common methods of the `List` base class
`src/AList.cpp`	Implementation file for the `AList` member functions that implement the concrete array based `List`
`src/LList.cpp`	Implementation file for the `LList` member functions that implement the concrete linked list based `List`

This week you will be adding in several new member functions to the `LList` class. So all of your work will be to add in code into the `LList.[hpp|cpp]` header and implementation files.

As usual, before starting on the assignment tasks proper, you should make sure you have completed the following setup steps.

1. Copy the assignment repository on GitHub using the provided assignment invitation link for 'Assignment Recursion' for our current class semester and section.
2. Clone the repository using the SSH URL to your local class DevBox development environment.
3. Configure the project by running the `configure` script from a terminal.
4. Confirm that the project builds and runs, though no tests will be defined or run initially. If the project does not build on the first checkout, please inform the instructor.
5. You should create the issue for Task 1 and/or for all tasks for the assignment now before beginning the first task. On your GitHub account, go to issues, and create it/them from the issue templates for the assignment. Also make sure you are linking each issue you create with the `Feedback` pull request for the assignment.

Assignment Tasks

Task 1: Implement `LList insertBack()` and overloaded `operator>>()`

As usual make sure that you create Task 1 issue on GitHub and and have linked the issue to the `Feedback` pull request for this assignment before beginning.

Also as usual, you should be practicing incremental development. A good first step is to always add in the function prototype for `insertBack()` into the `LList` header file, and insert a stub function that does nothing into the implementation file. Also uncomment the declaration of the `virtual insertBack()` function from the `List.hpp` header file. I had to comment this out because your `LList` class will not compile if these functions are not defined. So as you add in the functions one by one in the tasks, uncomment them from the `List.hpp` header file as well. Notice that your function should have the same signature as defined in the base class, minus the `virtual` at the start and the `= 0` at the end of the function.

For this assignment, you can examine these function signatures implementations in the `AList` class. The implementation you do will be different, since you are using a linked list instead of an array, but the function signatures will be identical, except that the functions you write will be members of the `LList<T>::` class instead of `AList`. The first test case in the `assg-tests-LList.cpp` file contains the first set of tests you need to get to pass, so uncomment this first test case, and make sure you can compile and run the tests before proceeding.

The `insertBack()` has the same purpose for our `List` as we did in a previous assignment, it simply appends a new item to the back of the current list. So `insertBack()` takes a `value` of type `T` as its input, and it should be passed in as a constant reference parameter. As we did in the previous assignment, this method returns back a reference to itself, to support the overloaded `>>` operator. So you should be returning `*this` at the end of your function, and the return type of the function should be a `List<T>&` reference. Again see the signature of this function in the `AList` implementation if you are unsure of your function signature here.

Appending on a linked list is hopefully relatively easy. You do have to check for the special case of the list being empty and handle it. The general steps insert onto the back of the linked list are

1. Dynamically allocate a new `Node`
   - Initialize the nodes value to the input value given for the `insertBack()` function.
   - Also initialize the nodes `next` pointer to point to `nullptr`, since this node will become the new `back` node of the linked list.
2. Test if the list is empty or not, you should reuse the `isEmpty()` member function to test this.
   - If the list is not empty, then simply append the new node by linking `next` of the current `back` node to the new node, and update `back` pointer to be pointing to this new back node.
   - If the list is empty, then both `front` and `back` should be pointed to this new node, since the node is both the front and the back now of the list of 1 item.
3. Don't forget to update the `size` of the list.

NOTE: The `size` parameter is actually defined in the abstract `List.hpp` base class header file. Take a moment to look for that. There were also a couple of methods implemented in the base class (the ones without `virtual` in front of them), for example `getSize()`. It is common to have common functionality implemented in the base class of an inheritance hierarchy. Notice that `size` is declared to be `protected` instead of `private` as usual. `protected` means that this value is private for people using the `List` class, but it can be seen and used directly by child classes of `List` that inherit from `List`. Your `LList` inherits publicly from the `List` base class, thus it has a member variable named `size`. However, C++ can be a bit crufty. Even though `size` is a member variable of your `LList` because of inheritance, if you simply refer to it as `size` the compiler has some trouble determining what variable you are referring to. This has to do with namespaces of templates, and this normally isn't a problem if you are working with non-templated classes. But for your class, you will have to refer to anything defined in the base class by prepending `this`, so for example, to update the size, you can do

```
this->size += 1;
```

which will increment the size by 1 as you need here.

Once you have `insertBack()` working and the first test case passing, make sure that you uncomment the calls to `insertBack()` in the copy constructor and standard array based constructor. Both of these constructors want to make use of your `insertBack()` method to construct new lists. Once those calls are uncommented, you can then uncomment the second set of test cases that test the `insertBack()` more thoroughly.

Then you should also add in the overloaded `operator>>()` as we did before. It should be a simple method that simply calls your `insertBack()` to do the actual work. The second set of test cases in the `assg-tests-LList.cpp` file tests the overloaded append operator and the constructors for your `LList` class.

Once you are satisfied with your code, commit your changes and push them to the `Feedback` pull request of your GitHub repository.

Task 2: Implement `LList getFront()` and `getBack()` API accessors

The `getFront()` and `getBack()` accessors should be simpler than the first task you did. We would normally have done these first, but we needed a way to be able to get items onto the list so we could do some tests, thus we had to implement one of the insertion mutators first.

Both of these methods return a value of type `T`, either the value at the front or back of the list respectively. These should be relatively simple because in the linked list class implementation, we keep member variables named `front` and `back` respectively that should always be pointing to the front and back nodes of the current linked list. For a linked list of size 1, with only one item in the list, both `front` and `back` will point to the same and only node on the list. When the list is empty, both `front` and `back` should be set to the `nullptr`. As usual start by creating stub functions and don't forget to uncomment the function prototype of the virtual function from the `List.hpp` file as well.

These methods should be relatively simple, but there is a small wrinkle. You should first check if the list is empty before trying to access and return the front/back item of the list. If the list is empty, `throw()` a `ListEmptyException`. This exception has already been defined for you in the `ListException.hpp` header file. Users of lists should check that the list is not empty before attempting to get values from the list. Throwing the exception allows for a user to catch and handle this case for situations where they might want to try this without knowing if the list is empty or not before hand.

The next two set of unit tests in the `assg-tests-LList.cpp` test file test `getFront()` and `getBack()` respectively. You should uncomment them and implement the functions one at a time.

Once you are satisfied with your work, add and commit your changes to the `Feedback` pull request of your GitHub repository.

Task 3: Implement `LList insertFront()` and overloaded `operator<<()`

For task 3 you are to implement the prepending to the linked list operation. The implementation of the `insertFront()` should be pretty similar to your task 1 function to append an item. Linking a new `front` node instead of a `back` node has some slight differences but is mostly the same. This function has the same signature as `insertBack()`. And once you have the function implemented, you should also add in the overloaded `operator<<` for prepending items to lists. As usual, start with a stub function and by uncommenting the virtual function from `List.hpp` and the next set of tests cases and making sure things compile and run.

Prepending to the front of a linked list is pretty similar to the task 1 of appending to the end. The basic approach is:

1. Dynamically allocate a new `Node` again, just like before.
   - Initialize the nodes value to the input value given for the `insertBack()` function.
   - Also initialize the nodes `next` pointer to point to `nullptr`, this node could end up being the only node on the list, so you need to insure it points to `nullptr` if that happens.
2. Test if the list is empty or not, you should reuse the `isEmpty()` member function to test this.
   - If the list is not empty, then prepend the new node by linking the new nodes `next` pointer to the current `front` node, and then update `front` to point to this new node.
   - If the list is empty, then both `front` and `back` should be pointed to this new node, since the node is both the front and the back now of the list of 1 item.

Once you are satisfied with your implementation and can pass the uncommented tests of `insertBack()` and the overloaded `operator<<()`, commit your changes and push them to the `Feedback` pull request of your GitHub repository.

Task 4: Implement `LList deleteIndex()` API mutator

The fourth task is to implement the `deleteIndex()` defined mutator for the `LList` linked list API. Any method that adds or removes values to the list can be considered a mutator method, they are changing the contents of the container. `deleteIndex()` is a bit more tricky to implement than the previous tasks, and you will have to really understand iterating through and manipulating linked list nodes for this and the next task.

As usual, start by uncommenting the virtual signature in `List.hpp`, adding in the function definition to `LList.hpp` and a stub implementation to `LList.cpp`, and uncommenting the next unit tests and making sure project still compiles and runs the tests.

The following is the suggested approach / algorithm to use to implement the `deleteIndex()` method:

1. First check that the requested index to delete is a valid index. You do this by testing if the index is less than 0, or if it is greater than or equal to the current `size` of the list. If the requested index is out of bounds, `throw` a `ListMemoryBoundsException`.
2. If the index is 0, you are deleting the front node. This is a relatively simple case. Reuse the `deleteFront()` mutator method to remove this value from the list.
3. If the index is the `back` / last node of the linked list, reuse the `deleteBack()` mutator method to remove this value from the list.
4. Otherwise it is a bit more complicated, though you know that a) neither the front nor back node will be removed and b) the list must have 3 or more nodes, and it will not become empty as a result of deleting the asked for node.
    - You need to iterate through the linked list nodes until you are positioned at the node to be removed.
    - The simplest implementation of this is to keep track of the node previous to the one you need to remove, because you need to point the previous nodes `next` pointer to the node after the one you will remove to get it out of the linked list.
    - Make a temporary pointer to the node you will be deleting.
    - Re-point the previous nodes `next` pointer to the node after the one being removed.
    - It is now safe to `delete` / deallocate the node now that is being removed.

Once you are satisfied with your work on the `deleteIndex()` member function, commit your changes and push them to the `Feedback` pull request of your repository.

Task 5: Implement `LList deleteValue()` API mutator

The fifth task is to implement the `deleteValue()` container mutator of the `LList` linked list concrete class. The purpose of this method is to delete all nodes with the indicated value. Thus more than one node may be removed as a result of calling this function since duplicate values are allowed in the list of values.

As usual, uncomment the virtual function signature in `List.hpp`, add the signature to `LList.hpp` and a stub to `LList.cpp`. Uncomment the next test case and make sure project compiles and runs tests before beginning implementation.

We cannot reuse the `deletIndex()` member method in a simple way for this method unfortunately, because if we removed values while iterating through the list, it is too easy to end up referencing a removed or lost portion of the list. If we could iterate through the list in reverse (e.g a doubly linked list), then this approach might be worth trying.

However, we can use a similar general approach to the `deleteIndex()` method, by first handling removal of any `front` nodes then `back` nodes with the value we are searching for, before proceeding to remove internal nodes.

1. First remove all nodes from the front of the list that contain the value to be removed. Do this as a simple loop that will call `deleteFront()` repeatedly as long as the value at the `front` is the value to be removed. Be careful, it is possible for the list to become empty while doing this loop.
2. Next do the same for any back nodes that are the value to be removed. Use a loop that repeats as long as the `back` value is the value to be removed and call `deleteBack()` to remove it.
3. At this point we know that the `front` and `back` nodes do not contain the value to be removed. The list might be empty, or have 1 or 2 values. In all cases, if there are any values that still need to be removed, they will not be the first or last node, so in that case the node will be of size 3 or larger.
    - Iterate through the list keeping track of the previous node of the node being tested.
    - If the node being tested contains the value, it needs to be removed.
    - Make a temporary pointer to the node to be deleted.

- Re-point the previous nodes `next` pointer to the node after the one being removed.
- It is now safe to `delete` / deallocate the node now that is being removed.
- The previous pointer now has a different node after it, continue iterating checking the new next node to see if it should be removed or not.

Also we didn't mention above but it is a considered an error if an attempt is made to `deleteValue()` and no values are found in the list match that can be deleted. So you need to keep a flag that lets you know if at least 1 or more values were found and removed, and if no values were removed, you need to throw a `ListValueNotFoundException` at the end of the unsuccessful search to find values to delete.

Step 2 above makes the described algorithm a bit inefficient, because the `deleteBack()` method is $\mathcal{O}(n)$ performance, each time it is called we have to iterate through the whole list to get to the end item. Step 2 is unnecessary with a small modification to step 3. Your step 3 will probably work to remove any values that are on the end of the list as described. However, it may not properly update the `back` pointer if it does end up removing one or more values from the back. But instead of performing step 2 explicitly, you can check if you are deleting the current `back` value, and update the back pointer in step 3, which will keep the performance of this method to being $mathcalO(n)$ in total. Once you get the above described algorithm to work, you can try and improve the performance as just described for a bit of extra credit.

Once you are satisfied with your work, commit you changes and push them to the repository. This completes the assignment, so you should have a pull request created with all of the commits for these tasks, and all of the issues associated with this assignment associated with the pull request. All actions/tests should be passing now for the final commit you made for task 5.

Assignment Submission

For this class, the submission process is to correctly create pull request(s) with changes committed and pushed to your copied repository for grading and evaluation. For the assignments, you may not be able to complete all tasks and have all of the tests successfully finishing. This is ok. However, you should endeavor to have as many of the tasks completed before the deadline for the assignment as possible. Also, try and make sure that you only push commits that are building and able to run the tests. You may loose points for pushing a broken build, especially if the last build you submit is not properly compiling and running the tests.

In this problem, up to 25 points will be given for having at least 1 commit that compiles and runs the tests (and at least some attempt was made to work on the first task). Thereafter 15 points are awarded for completing each of the 5 tasks. However you should note that the autograder awards either all point for passing all tests, or no points if any test is failing for one of the tasks. Also note that even if you pass all tests, when the instructor evaluates your assignment, they may remove points if you don't follow the requirements for implementing the code (e.g. must reuse functions here as described, need to correctly declare parameters or member functions as `const` where needed, must have function documentation correct). You may also loose points for style issues. The instructor may give back comments in pull requests and/or create new issues for you if you have issues such as these, so it is good to have work committed early before the due date, so that the instructor may give feedback requesting you to fix issues with your current submission.

Program Style

At some point you will be required to follow class style and formatting guidelines. The VSCode environment has been set up to try and format your code for some of these guidelines automatically to conform to class style requirements. But not all style issues can be enforced by the IDE/Editor. The instructor may give you feedback in your pull comments and/or create issues for you for the assignment that you need to address and fix. You should address those if asked, and push a new commit that fixes the issue (or ask for clarification if you don't understand the request). In general the following style/formatting issues will be required for programs for this class:

1. All programs must be properly indented. All indentation must be consistent and lined up correctly. Class style requires 2 spaces with no embedded tabs for all code indentation levels. The editor style checker should properly indent your code when you save it, but if not you may need to check or correct this if code is misaligned or not properly indented.
2. Variable and function names must use `camelCaseNameingNotation`. All variable and function names must begin with a lowercase letter. Do not use underscores between words in the variable or function name. Often

function names will be given to you, but you will need to create variables, and maybe some functions, that conform to the naming conventions.
    - Global constants should be used instead of magic numbers. Global constants are identified using `ALL_CAPS_UNDERLINE_NAMING`.
    - User defined types, such as classes, structures and enumerated types should use camel case notation, but should begin with an initial upper case letter, thus `MyUserDefinedClass`.
3. You are required to use meaningful variable and function names. Choosing good names for code items is an important skill. The code examples and starting code tries to give examples of good and meaningful names. In general, do not use abbreviations. Single variable names should be avoided, except maybe for generic loop index variables `i`, `j`, etc. Make your code readable, think of it as writing a document to communicate with other developers (and with your instructor who will be evaluating your code).
4. There are certain white space requirements. In general there should usually never be more than 1 blank line in a row in your code. Likewise there should usually not be more than 1 blank space on a line. There should be 1 blank space before and after all binary operators like `+`, `*`, `=`, `or`.
5. Function documentation is required for all regular functions and all class member functions. You need to follow the correctly formatted Doxygen function documentation format. We will use function documentation generation, and you should be sure your documentation can be built without emitting warnings or errors. Likewise all files should have a file header documentation at the top. You should edit the file header of files where you add in new code (not simply uncommenting existing code). Make sure the information has your correct name, dates, and other information.
6. Practice using proper Git commit messages. You should refer to issues and tasks correctly in commit messages.

Additional Information

The following are suggested online materials you may use to help you understand the tools and topics we have introduced in this assignment.

- Lecture U08-1 Linked List Fundamentals
- Lecture U07-1 Object Inheritance and Composition
- Data Structures - Full Course Using C and C++ The videos from the start of this course, through 'Doubly Linked Lists' would all be excellent resources to better understand the List ADT and their implementation as a arrays and linked lists.
- C++ Classes and Objects
- C++ Inheritance
- C++ Templates