CS 15: Homework 02: LinkedLists

Introduction

In this assignment you will implement a linked list data structure. The list for this assignment will be a **doubly linked** character list. You will have to write both the public and private sections of the `CharLinkedList` class. The class definition will go in a file named `CharLinkedList.h`, and the class implementation will go in a file named `CharLinkedList.cpp`. Well describe the interface first, then give some implementation specifics, ask you some questions, and finally give submission instructions. Dont forget to answer the questions in your README!

Program Specification

Important Notes

- The names of your functions / methods as well as the order and types of parameters and return types must be exactly as specified. This is important because we will be compiling the class you wrote with our own client code!
- Any exception messages should likewise print exactly as specified and use the given error type.
- You may not have any other public functions.
- All data members must be private.
- You may **not** use any `C++` strings in your `CharLinkedList` implementation, except for:
  - the `toString()` function;
  - when throwing exception messages.
- You may **not** use `std::vector` or any other built-in facility that would render the assignment trivial.

Interface

Your class must have the following interface (all the following members are public):

- Define the following constructors for the `CharLinkedList` class:
  - `CharLinkedList()`
    The default constructor takes no parameters and initializes an empty list.
  - `CharLinkedList(char c)`
    The second constructor takes in a single character as a parameter and creates a one element list consisting of that character.
  - `CharLinkedList(char arr[], int size)`
    The third constructor takes an array of characters and the integer length of that array of characters as parameters. It will create a list containing the characters in the array.
  - `CharLinkedList(const CharLinkedList &other)`
    A copy constructor for the class that makes a deep copy of a given instance.

  Recall that all constructors have no return type.

- `~CharLinkedList()`
  Define a destructor that destroys/deletes/recycles all heap-allocated data in the current list. It has no parameters and returns nothing. **This function must use a private recursive helper function.**

- `CharLinkedList &operator=(const CharLinkedList &other)`
  Define an assignment operator for the class that recycles the storage associated with the instance on the left of the assignment and makes a deep copy of the instance on the right hand side into the instance on the left hand side.

- `bool isEmpty() const`
  An `isEmpty` function that takes no parameters and returns a boolean value that is true if this specific instance of the class is empty (has no characters) and false otherwise.

- `void clear()`
  A `clear` function that takes no parameters and has a `void` return type. It makes the instance into an empty list. For example if you call the `clear` function and then the `isEmpty` function the `isEmpty` function should return true.

- `int size() const`
  A `size` function that takes no parameters and returns an integer value that is the number of characters in the list. The size of a list is 0 if and only if it isEmpty.

- `char first() const`
  A `first` function that takes no parameters and returns the first element (`char`) in the list. If the list is empty it should throw a `C++ std::runtime_error` exception with the message "cannot get first of empty LinkedList".

- `char last() const`
  A `last` function that takes no parameters and returns the last element (`char`) in the list. If the list is empty it throws a `C++ std::runtime_error` exception with the message "cannot get last of empty LinkedList".

- `char elementAt(int index) const`
  An `elementAt` function that takes an integer index and returns the element (`char`) in the list at that index. NOTE: Indices are 0-based. If the index is out of range it should throw a `C++ std::range_error` exception with the message "index (IDX) not in range [0..SIZE)" where IDX is the index that was given and SIZE is the size of the linked list. For example: "index (6) not in range [0..3)" if the function were to be called using the index 6 in a size 3 list. Note the braces and the spacing!

- `std::string toString() const`
  A `toString` function that takes no parameters and has a `std::string` return type. It returns a string which contains the characters of the CharLinkedList. The string will be formatted like this:

  ```
  [CharLinkedList of size 5 <<Alice>>]
  ```

  where, in this example, 5 is the size of the list and the elements are the characters `'A', 'l', 'i', 'c', 'e'`. The empty list would print like this:

  ```
  [CharLinkedList of size 0 <<>>]
  ```

  Note: There is **no** new line after the last `]`.

  Caution! Your output will be verified automatically, so the output format is essential to get right. There is no whitespace printed, except the single spaces shown between the elements inside the square brackets. (i.e. "CharLinkedList 'space' of 'space'", etc.). The capitalization must also be exactly as shown.

- `void pushAtBack(char c)`
  A `pushAtBack` function that takes an element (`char`) and has a `void` return type. It inserts the given new element after the end of the existing elements of the list.

- `void pushAtFront(char c)`
  A `pushAtFront` function that takes an element (`char`) and has a `void` return type. It inserts the given new element in front of the existing elements of the list.

- `void insertAt(char c, int index)`
  An `insertAt` function that takes an element (`char`) and an integer index as parameters and has a `void` return type. It inserts the new element at the specified index. The new element is then in the index-th position. If the index is out of range it should throw a C++ `std::range_error` exception with the message "index (IDX) not in range [0..SIZE]" where IDX is the index that was given and SIZE is the size of the list. NOTE: It is allowed to insert at the index after the last element. Also, the braces in this message are different from those in the `elementAt` range error.

- `void insertInOrder(char c)`
  An `insertInOrder` function that takes an element (`char`), inserts it into the list in alphabetical order, and returns nothing. When this function is called, it should insert the element at the first index after which some character greater than the inserted character appears. Example 1: Inserting `'C'` into `"ABDEF"` should yield `ABCDEF` Example 2: Inserting `C` into `"BLARE"` should yield `BCLARE"` Inserting any character into an empty list should yield a list containing that single character. You can rely on the built-in `<, >, <=, >=,` and `==` operators to compare two chars.

- `void popFromFront()`
  A `popFromFront` function that takes no parameters and has `void` return type. It removes the first element from the list. If the list is empty, it should throw a `C++ std::runtime_error` exception with message "cannot pop from empty LinkedList".

- `void popFromBack()`
  A `popFromBack` function that takes no parameters and has `void` return type. It removes the last element from the list. If the list is empty, the function should throw a `C++ std::runtime_error` exception with message "cannot pop from empty LinkedList".

- `void removeAt(int index)`
  A `removeAt` function that takes an integer index and has `void` return type. It removes the element at the specified index. If the index is out of range it should throw a C++ `std::range_error` exception with message "index (IDX) not in range [0..SIZE)" where IDX is the input index and SIZE is the size of the list.

- `void replaceAt(char c, int index)`
  A `replaceAt` function that takes an element (`char`) and an integer index as parameters and has a `void` return type. It replaces the element at the specified index with the new element. If the index is out of range it should throw a C++ `std::range_error` exception with the message "index (IDX) not in range [0..SIZE)" where IDX is the index that was given and SIZE is the size of the list.

- `void concatenate(CharLinkedList *other)`
  A `concatenate` function that takes a pointer to a `CharLinkedList` and has `void` return type. It adds a copy of the list pointed to by the parameter to the end of the list on which the function was called. For example, concatenating `CharLinkedListOne`, containing `"cat"` with `CharLinkedListTwo`, containing `"CHESHIRE"`, will result in `CharLinkedListOne` containing `"catCHESHIRE"`. Note: An empty list concatenated with a second list is the same as copying the second list. Concatenating a list with an empty list doesn't change the list. Also a list can be concatenated with itself, e.g concatenating `CharLinkedListTwo` with itself, results in `CharLinkedListTwo` containing `CHESHIRECHESHIRE"`.

You may add any private methods and data members. We particularly encourage the use of private member functions that help you produce a more modular solution.

Before you start writing any functions please sit down and read this assignment specification. Some of these functions do similar tasks. Perhaps it would be prudent to organize and plan your solution using the principles of modularity, e. g., helper functions. This initial planning will be extremely helpful down the road when it comes to testing and debugging; it also helps the course staff more easily understand your work (which can only help).

Also, the order in which we listed the public methods/functions of the `CharLinkedList` class, is not necessarily the easiest order to implement them in. If you plan your functions out and identify the easy ones it will make your work easier and final submission better.

If you are having issues planning out your assignment we encourage you to come in to office hours as early as possible.

Implementation Details

Copy the starter files from `/comp/15m1/files/hw02` to get the following files with header comments:

- `CharLinkedList.h`
- `CharLinkedList.cpp`
- `unit_tests.h`
- `Makefile`

With those files in place, you can use `make test_list` to compile and link your code; after doing so, `./test_list` will execute the testing code. The command `make clean` will remove all compiled files, leaving you with just the various files needed for submission.

Implement the list using a **doubly linked list**. This means that each node has both a next and a previous pointer.

The file `CharLinkedList.h` will contain your class definition only. The file `CharLinkedList.cpp` will contain your implementation. The file `unit_tests.cpp` will contain your unit testing functions. We will assess the work in all three files.

Implementation Advice

Do NOT implement everything at once!
Do NOT write code immediately!

Before writing code for any function, draw before and after pictures and write, in English, an algorithm for the function. Only after you've tried this on several examples should you think about coding.

First, just define the class, `#include` the .h file, define an empty main function in your test file, and compile. This tests whether your class definition is syntactically correct.

Next, implement just the default constructor. Add a single variable of type `CharArrayList` to your test main function, and compile, link, and run.

Now you have some choices. You could add the destructor next, but certainly you should add the print function soon.

As you go, you will be best off if you add one function, then write code in your test file that performs one or more tests for that function. Write a function, test a function, write a function, test function, ... This is called "unit testing." As you write your functions, consider edge cases that are tricky or that your implementation might have trouble with. You should write specific tests for these cases.

You can organize your testing file as a function (perhaps with additional helper functions) to test each item in the interface. For example, you might have a function named `test_insertAt` that runs a bunch of tests on the `insertAt` function. You can even write tests that verify an exception using the `try-catch` statement pattern. If you do not catch an exception, then it will crash your program with an unhandled exception. This is okay, but you should comment out tests that should crash with a note to the grader that the exception arose as expected (assuming that is what happened).

If you need help, TAs are likely to ask about your testing plan and ask to see what tests you have written. They will likely ask you to comment out the most recent (failing) tests and ask you to demonstrate your previous tests.

We will evaluate your testing strategy and code for breadth (did you test all the functions?) and depth (did you identify all the normal and edge cases and test for error conditions?). Don't write a test a function and then delete it!

For testing exceptions you have two choices - either is fine:

- You may test that exceptions get thrown when appropriate, let it crash your test program, and then comment out the test with a note that the exception was thrown (be honest — we will check).
- You may write a catch statement in your testing code that catches the exception, tests the message, and then prints out a success or failure indication.

Be sure your files have header comments, and that those header comments include your name, the assignment, the date, and the files purpose. See our style guide for more information about commenting your code.

README

With your code files you will also submit a `README` file, which you will create yourself. The file is named `README`. The contents should be in plain text. Format your `README` however you like, but it should be well-organized and readable. Include the following sections:

(A) The title of the homework and the author's name (you).

(B) The purpose of the program.

(C) Acknowledgements for any help you received.

(D) The files that you provided and a short description of what each file is and its purpose.

(E) How to compile and run your program.

(F) An outline of the data structures and algorithms that you used. Given that this is a data structures class, you always need to discuss the data structure that you used and justify why you used it. Specifically for this assignment please discuss the features of arrays, major advantages and major disadvantages of utilizing an array list as you have in this assignment. The algorithm overview may not be relevant depending on the assignment.

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

Each of the sections should be clearly delineated and begin with a section heading describing the content of the section. (It is not sufficient to just write "C" as a section header—write out the section title to help the reader of your file.)

QUESTIONS

In addition, answer the following questions in your `README` file:

- Q1: Which functions in the interface were easier to implement for linked lists compared to array lists? Why?
- Q2: Which functions in the interface were harder to implement for linked lists compared to array lists? Why?
- Q3: If a client had an existing program that uses CharArrayLists, what changes would they have to make to their code to switch to CharLinkedLists?

Submitting Your Work

Once your work is complete, or at least thought to be complete, you will download all of your code files, README, etc. from the CS servers where you have been working. You will upload work to Gradescope.

The Gradescope site will provide some autograding functionality that will give you feedback on your work. You should aim to submit code as early as feasible to Gradescope (to verify, for instance that all of your code compiles properly, even if some of your functions don't yet do what they are supposed to do). Autograder feedback is most useful if you get it well in advance of the submission deadline.

A recommendation: you can spend time writing up documentation, comments, etc. as you go. You can also polish that work after your coding itself is done. A good approach would be to get the code in reasonable shape, and then work on the documentation polish. If the first time you submit to Gradescope is just before the deadline, you aren't likely to get much that is useful from the autograder feedback.