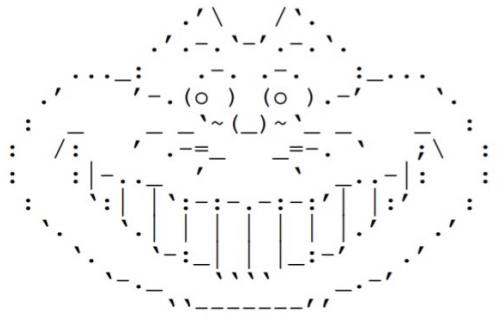
CS 15 LinkedLists Homework



You've picked up a bit of an attitude, but still curious and willing to learn, I hope.
- Cheshire Cat

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`. We'll describe the interface first, then give some implementation specifics, ask you some questions, and finally give submission instructions. Don't 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
  - In the `toString()` and `toReverseString()` functions.
  - When throwing exception messages.
- You may **not** use `std::vector`, `std::list`, or any other built-in facility that would render the assignment trivial.
- Some functions are **required** to be implemented with recursion. For these functions, you are welcome to write private helper functions that do the recursion.

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". Note that there is no newline at the end of this message.

- `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![1] **This function must use a private recursive helper function.**

- `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 format of strings is essential to get right. There is no whitespace printed, except the single spaces

[1] $[N_1..N_2)$ is *interval notation* for a "half open interval". See Wikidpedia for more.

shown between the elements inside the square brackets. (i.e. "CharLinkedList 'space' of 'space'", etc.). The capitalization must also be exactly as shown.

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

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

  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 `]`. For this function, you are **not** allowed to make a copy of the list—the fact that it's doubly linked should be a great help! Also, if you're not using a tail pointer, consider how a recursive helper function might be useful here (**hint:** this fn might also come in handy elsewhere where recursive helper functions are required!)

- `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 ASCII order, and returns nothing. When this function is called, it may assume the list is correctly sorted in ascending order, and it should insert the element at the first correct index. Example: Inserting 'C' into "ABDEF" should yield "ABCDEF" You can rely on the built-in `<, >, <=, >=,` and `==` operators to compare two chars.

- `void popFromFront()`
  A `popFromFront` function that takes no parameters and has a `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 the message "cannot pop from empty LinkedList".

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

- `void removeAt(int index)`
  A `removeAt` function that takes an integer index and has a `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 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 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. **This function must use a private recursive helper function.**

- `void concatenate(CharLinkedList *other)`
  A `concatenate` function that takes a pointer to a second `CharLinkedList` and has a `void` return type. It adds a copy of the list pointed to by the parameter value to the end of the list the function was called from. For example if we concatenate `CharLinkedListOne`, which contains "cat" with `CharLinkedListTwo`, which contains "CHESHIRE", `CharLinkedListOne` should contain "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 your 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.

JFFEs (Just For Fun Exercises)

If you complete the above functions, you may add the following functions. There is no extra credit, but they're fun and educational.

- `void sort()`
  A `sort` function that takes no input and has a `void` return type. It sorts the characters in the list into alphabetical order.

- `CharLinkedList *slice(int left, int right)`
  A `slice` function that takes a left index and a right index and returns a pointer to a new, heap-allocated `CharLinkedList`. The new list contains the characters starting at the left index and up to, but not including, the right index. If the first index is equal to or greater than the

second, it returns a new, empty list. The left index must be in the range [0..SIZE) and the right index must be in the range [0..SIZE] where SIZE is the size of the list. Since the right index is not included in the final slice, requesting a slice where the right index is 1 index past the last element is still a valid request. If the either index passed is out of range the function should throw a `C++ std::range_error` with an appropriate message.

Implementation Details

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

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

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.h` will contain your unit testing functions. We will assess the work in all three files.

Once again, we have provided you with a fully implemented `timer_main.cpp` and `Makefile`—you should not edit these. As with HW1, once your have **completed** your `CharLinkedList` implementation, run "`make timer`" to compile your timer executable program, and run `./timer` to execute it and take time measurements.

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 in your `unit_tests.h` (we have already done this for you), write a dummy test (that does nothing), and run the `unit_test` command from the command line.

This will test whether your class definition is syntactically correct.

Then implement just the default constructor. Add a single variable of type `CharLinkedList` to a `unit test`, and run your tests.

Then you have some choices. You could add the destructor next, but certainly you should add the `toString` function soon.

You will 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.

If you need help, TAs will 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.

It is important to consider all cases a function can be used in when writing tests! This can help you catch bugs that may not be obvious to the human eye. If you need help, the TAs will ask about your testing plan and ask to see what tests you have written. As a hint for writing tests: Notice that the interface for the `CharLinkedList` class is the same as the interface for the `CharArrayList` class you implemented last assignment.

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!

In addition to testing, be sure your files have header comments, and that those header comments include your name, the assignment, the date, and the file's purpose. See our style guide for more information about commenting your code.

Finally, you should review the grading process outlined on the course administration webpage. This includes useful information like how to view your autograder score, the maximum number of times you can submit, and how many late tokens you can use.

README Outline

With your code files you will also submit a `README` file, which you will create yourself. The file is named `README`. There is no .text or any other suffix. The contents is in plain text, lines less than 80 columns wide. 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 need to always discuss the **ADT** that you used and the **data structure** that you used to implement it and justify why you used it. Specifically for this assignment please discuss the features of linked lists, major advantages and major disadvantages of utilizing a linked list as you have in this assignment. The **algorithm** overview is always relevant. Please pick a couple interesting/complex algorithms to discuss in the `README`

(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.

(H) Please let us know approximately how many hours you spent working on this project. This should include both weeks one and two.

(I) Answer the following questions regarding the time measurements taken for the `CharLinkedList` operations (see the explanation of **timer_main.cpp** for instructions on taking these measurements).

1. There are three categories of operations listed (insertion, removal, and access). Within each category, list the times each operation took and rank the operations from fastest to slowest.
2. Discuss these rankings. Why were certain operations so much faster or slower than others? What are the features of linked lists that cause these disparities?
3. Now compare these measurements with the measurements you took for HW1. Which operations are faster when using array lists? Which are faster using linked lists? Why?

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

Be sure to read over the style guide before submitting to make sure you comply with all the style requirements. You will need to submit the following files:

```
CharLinkedList.h, CharLinkedList.cpp
unit_tests.h
Makefile
README
```

You must submit them using Gradescope to the assignment `hw_linkedlists`. Submit these files directly, do not try to submit them in a folder. Note, you should only be submitting these files. If you feel that you need multiple classes or structs in your solution, please consolidate them into `CharLinkedList.h` and `CharLinkedList.cpp`.

Before submitting your work, please make sure your code and documentation (including sections of the README) conform to the course style guide.