CSE221 Data Structures (Fall 2017)
Instructor: Prof. Tsz-Chiu Au
Due date: October 9, 2017, 11:59 pm.

Assignment 1: Circularly Linked Lists

In our lectures, we talked about data structures for storing and manipulating sequential data. Although the concepts of linked lists, stacks, queues, etc. are quite simple, they actually require you to have a good programming skill and a great deal of knowledge about OOP and templates in C++ in order to come up with a good implementation that is safe, easy to use, and high performance. In this assignment, you are given the opportunities to implement *a circularly linked list with available space lists*. In addition, you will use the data structure to implement *a stack-based algorithm for evaluating a numerical expression by converting the expression into postfix notation*. Hopefully you can gain further insights into these data structures while implementing them in C++.

Task 1: Circularly Linked List with Available Space Lists

First of all, please implement a circularly linked list as described in Section 3.4 in our textbook Goodrich et al. (2011). Please read the section carefully, as your implementation will follow the description in the section as much as possible. Circularly linked lists are also discussed in Section 4.4 from Page 194 to Page 197 in Horowitz et al. (2007), which offers a similar implementation. In Horwitz et al. (2007), a circularly linked list is also called a circular list or a singly-linked circular list. However, these terms refer to the same data structure.

Section 3.4 in Goodrich et al. (2011) provides the C++ code of a class named `CircleList`, including both the class declaration and the definitions of all member functions. However, the source codes are *incomplete*—some implementations are deliberately simplified and some cases are not handled. Moreover, the codes are based on a rather old version of C++. You need to update the code in order to conform to the C++03 standard. *Notice that in all of our programming assignments you are not allowed to use any new C++ features in C++11 and beyond.*

Please change the name of the class from `CircleList` to `CircularList`, as `CircularList` is a more popular name for this data structure. The following is a list of issues that you should keep in mind when you read the codes.

- Don't use "`typedef string Elem;`" on Page 130; instead, you should define the class as a template and define Elem as a type variable in the template. Thus, change it to "`template<typename Elem>`" and add this to the appropriate locations in your code. In our grading script, we assume the class is a template class. Your template class should work no matter what type Elem is.

  By the way, the type variables in a template statement are declared using the `class` keyboard in the textbook, such as "`template<class Elem>`". However,

this usage is quite old and should be replaced with "`template<typename Elem>`" because Elem is not necessarily a class. For more information, please refer to https://stackoverflow.com/questions/213121/use-class-or-typename-for-template-parameters.

- The `private` keyword in `CNode` is redundant since all fields in a class are private by default. Notice that the `friend` statement is not affected by the private keyword since it does not declare a member of the class `CNode`.
- `CNode` should be declared inside `CircularList` since these nodes are only used by `CircularList` and therefore they should not be exposed. Define `CNode` as an inner class of `CircularList`. Our grading script will check whether `CNode` has been put outside `CircularList`.
- The textbook wrote, "To keep the code simple, we have omitted error checking." In `front()`, `back()`, `remove()`, and `advance()`, the member functions should first test whether the list is empty in order to avoid accessing the null cursor pointer. In fact, `front()`, `back()`, and `remove()` should throw exceptions if the list is empty. However, `advance()` does not need to throw an exception if the list is empty, since we can assume advancing the cursor of an empty list has no effect.
- Some of the member function definition contain only one line of codes. All these one-liners should be inline (i.e., turn them into an inline member functions).

In this exercise, you should define a template class `CircularList` that have the following *nine* member functions are:

- `bool CircularList<Elem>::empty() const;`

  Check whether the circular list is empty.

- `Elem& CircularList<Elem>::front() const;`

  Return a reference to the first element in the circular list.

- `Elem& CircularList<Elem>::back() const;`

  Return a reference to the last element in the circular list.

- `void CircularList<Elem>::advance();`

  Advance the curser to the next element in the circular list.

- `void CircularList<Elem>::add(const Elem& e);`

  Add an element after the cursor.

- `void CircularList<Elem>::remove();`

  Remove the node after the cursor.

- `int CircularList<Elem>::size() const;`

  Remove the size of the circular list (i.e., the number of elements in the list).

- `void CircularList<Elem>::reverse();`

  Reserve the circular list.

- `void CircularList<Elem>::cleanup();`

  Free the memory all unused nodes in the available space list.

The first six member functions are discussed extensively in Section 3.4 of the textbook. You should refer to the text when you implement these member functions. However, in the textbook, both `front()` and `back()` have a keyboard `const` before the return type `Elem&`, meaning that it can return a constant reference only. It is inappropriate since in many applications, we want to modify these elements. Therefore, you should not add `const` to the return type `Elem&` in `front()` and `back()`.

You should add a new member function called `size()`, which return the size of the circular list. In our lecture, we discussed two ways to implement `size()`—either scanning through the list on demand or maintaining a variable to keep track of the number of elements. While both implementations are fine in this exercise, the former makes more sense if `size()` is not invoked frequently, which is the case in this exercise. However, it is up to you to choose how to implement `size()`.

You should also implement a new member function called `reverse()`, which reverses the order of the elements in the circular list. Section 3.4.2 describes a function for reversing a doubly linked list, but it does not work for `CircularList`. Therefore, you should define your own algorithm to reserve a circular list. Notice that after reversing a circular list, the front element becomes the back element, and the back element becomes the front element.

Finally, the last member function is `cleanup()`. It is actually part of the implementation of available space lists that you are asked to implement. In our lecture, we talked about available space lists for recycling nodes in order to avoid excessive system calls for memory allocation. A full discussion of available space lists can also be found in Section 4.5 on Page 197—199 in Horowitz et al. (2007). The implementation of available space lists is actually quite simple—you only need to add one more pointer to `CNode` in `CircularList`, which will point at the first element in the available space list. Then you update the pointer accordingly in the member functions of `CircularList`. Whenever a member function needs to use the `new` operator, it should check with the available space list first. The use of available space lists can avoid deleting any node as much as possible. However, we should give the user an opportunity to free up the memory used by available space lists that could possibly use too much memory. Thus, you are asked to implement a function called `cleanup()` that deletes all nodes in the available space lists. Also, call `cleanup()` in the destructor of `CircularList` in order to free up all memory when a circular list is deleted, since this available space list is associated with an object of `CircularList`—no sharing

of available space lists between different objects of `CircularList`. Our grading script will check the memory usage of your program to see whether you have implemented available space lists.

Some other implementation issues:

- In all of our assignments, you don't need to define your own exception class when you need to throw an exception to signal some errors. You can use the existing exceptions in C++ such as `runtime_error`. Please check online how to throw these exceptions.

- In Task 2, you will use a stack to evaluate a numerical expression. We have provided you a file named `LinkedStack.h`, which contains the code of a template class called `LinkedStack`, which turns a circular list into a stack. You are only allowed to use the `LinkedStack` in Task 2. Please make sure that your implementation of `CircularList` works with `LinkedStack`.

- Please put all of codes of the `CircularList` and the definition of its member functions in a file named "`CircularList.h`". Please include all necessary header files, such as `stdexcep.h` for `runtime_error`. The code should be enclosed between the `#ifndef`, `#define`, and `#endif` statement in order to allow the code to be included from multiple file. Please look at `LinkedStack.h` to see how to use `#ifndef`, `#define`, and `#endif for this purpose`. You should write your name, your student ID, and your email address at the top of the file. Also, please describe the implementation of your functions at the top of the file.

Task 2: Evaluating a Postfix Expression.

In the second part of this exercises, you are asked to implement (1) an algorithm to convert a numerical expression in infix notation into postfix notation; and (2) an algorithm to evaluate a numerical expression in postfix notation. Both algorithms are fully described in our lecture note. For more details, please refer to Section 3.6 on Page 157—165 in Horowitz et al. (2007). In particular, the evaluation algorithm of postfix expression is discussed in Section 3.6.2 on page 162, and the infix to postfix algorithm is discussed in Section 3.6.3 on Page 165.

In this task, you do not need to worry about how tokenize an input string (i.e., how to crop up an input string into a sequence of tokens), as we have already provided you a tokenizer. Please read `tokenizer.h`, `tokenizer.cpp`, `Token.cpp`, and `Token.h` in order to see how the tokenizer works. In particular, you need to understand `Token.cpp`, and `Token.h` in order to see how to work with a token. In your code, you do not need to call the tokenizer since the main function is the only place the tokenizer will be used. Please do not modify or submit these files.

In addition, we provide you two files: `assignment1.h` and `assignment1.cpp`. In `assignment1.h`, we have defined the declaration of four functions:

- `vector<Token> postfix(const vector<Token>& expression);`

  Convert an expression in infix notation into an expression in postfix notation. The expressions are stored as a vector of tokens.

- `int eval(const vector<Token>& expression);`

  Evaluate an expression in postfix notation.

- `int getPriority(Token token);`

  Return the priority of an operator.

- `int getInStackPriority(Token token);`

  Return the in-stack priority of an operator.

Please read the comments of these functions for more detail. In particular, please read the comments of `getPriority()` and `getInStackPriority()` to see what have to be supported by these functions. Our grading script will evaluate these functions accordingly.

You are allowed to use `vector` in STL in Task 2 (not Task 1, though). However, you are not allowed to a vector as a stack; you must use the stack implementation in `LinkedStack.h`, which in turn uses `CircularList` in `CircularList.h`.

Your code should be written in `assignment1.cpp`. We have provided a code skeleton in `assignment1.cpp` and you have to insert your code there. Please include the header files that you need in your code. *You should not define any other functions or classes in `assignment1.cpp`.*

Some other implementation issues:

- In the textbook and the lecture notes, the pseudo-code of the two algorithms require you to insert a special character "#" in the expression and in the stacks. Actually, it is quite easy to avoid the insertion of any special characters to make these algorithms work. Please find ways to implement the algorithm without using "#".

- Note that the division operator "/" in the expressions is an integer division. For example, the answer of "3/2" is 1.

- Your programs should throw exceptions whenever there is a mistake in the user input. In fact, all of the four functions above will have to throw exceptions at some points in the code. Please think where you should throw exceptions. You can also play with the sample program "`postfix-eval-sample`" to see when it will

throw exceptions.

- We have provided you the `main.cpp` for you to test your functions. Our grading script will also use it to evaluate your programs. Please do not modify `main.cpp`. There is no need to submit `main.cpp`.

Testing and Submission

All files that you need in this assignment are put in a zip file called `prog1.zip`, which contains the following files:

- `handout-1.pdf` — the handout of this assignment (i.e., this file).
- `ChangeLog.txt` — the change log.
- `postfix-eval-sample` — the sample program.
- `main.cpp` — the main function of your program.
- `CircularList.h` — your implementation of circular lists. You will submit this file.
- `LinkedStack.h` — the implementation of stacks based on circular lists.
- `Token.h` — the declaration of the token class.
- `Token.cpp` — the implementation of the token class.
- `tokenizer.h` — the declaration of the tokenizer function.
- `tokenizer.cpp` — the implementation of the tokenizer.
- `assignment1.h` — the declaration of the functions for handling numerical expressions.
- `assignment1.cpp` — the implementation of the functions for handling numerical expressions. You will submit this file.
- `README.TXT` — your readme file. You will submit this file.

We have provided you a sample program called "`postfix-eval-sample`" that implements all functionalities of this program. Please run the sample program on our submission server and compare the output to your program's output. Please modify your program so that the output is the same as the output of the sample program (except the white spaces and the error messages in the exceptions). If you fail to run the sample program, please try to change the permission of the sample program to executable by this Linux commands: "`chmod 755 postfix-eval-sample`".

In this exercise, you are allowed to use the container class called `vector` in the Standard Template Library. However, you should not use other data structures in STL except the iterators of these container classes and some common exceptions such as `runtime_error`. If in doubt, please contact the instructor to ask whether a data structure or functions in STL can be used.

We will test your implementation of `CircularList` using a main function that is totally different from the one in `main.cpp`. We will test the completeness and correctness of your implementation, including whether your code will throw all necessary exceptions. We

will also check whether your program will cause memory leak. We will test whether `assignment1.cpp` uses the member functions in `CircularList.h` via `LinkedStack.h` correctly by replacing `CircularList.h` with the instructor's implementation of `CircularList.h`.

The error messages in `runtime_error` exceptions do not have to be exactly the same as the messages in the sample program we provided. You can write the error messages in your own sentences. However, we will check whether your algorithm have thrown the exceptions when necessary.

To compile your program on our submission servers, use this command:

```
g++ -o postfix-eval Token.cpp tokenizer.cpp assignment1.cpp
                                main.cpp
```

Before you submit your program, please check whether your program can be compiled correctly using this command on our submission server. We do not grade your program using other compilers or on other machines.

Please also submit a plain text file called "`README.TXT`" to tell us the extent of your implementation, more specifically which parts have been implemented and which parts have not. Also, if there are some known bugs in your program, you should state them in the file. An empty `README.TXT` is included in the source file.

You will submit only three files: (1) **`CircularList.h`**, (2) **`assignment1.cpp`**, and (3) **`README.TXT`**. These files should be self-contained and you cannot submit additional files. You should not submit `main.cpp` and `assignment1.h`, `LinkedStack.h`, `Token.h`, `Token.cpp`, `tokenizer.h` and `tokenizer.h` as you should not modify them. Please put these files in a zip file called **`assign1.zip`**, and submit it using the `dssubmit` script on our submission servers (uni06~10.unist.ac.kr). The submission command is

```
dssubmit assign1 assign1.zip
```

You should test this command long before the deadline in order to see whether you will encounter any issue. Please email Sangwoo Ha <swha@unist.ac.kr> if you fail to use this command.

Automatic Grading and Late Submission Penalty

We developed shell scripts to grade your programs automatically without human intervention. In the past, many students didn't read this handout carefully and implemented their programs with wrong function names or filenames, etc., causing compilation errors. Our TAs spent too much time to help students to fix these issues in the past. In order to avoid wasting our TAs' time, we decide to give students opportunities to see the results of the grading script running with their programs *before* the deadline, such that students can fix any issues *themselves* before the final submission. We will start running the grading scripts at midnight at least three days before the deadline. If you submit your program

earlier, you will see the grading report generated by the grading script in your account. Please fix any problems in your program according to the grading report, especially those that are caused by incorrect naming.

The scores in the pre-deadline grading reports are not final and they will not be recorded. However, the grading reports after the deadline are final. In fact, we will grade your programs three times after the deadline in order implement our late submission policy as described on our course webpage. The first real grading report will be generated right after the deadline. The second real grading report will be exactly one day after the deadline with a penalty of 15%. The third real grading report will be exactly third day after the deadline with a penalty of 30%. We will select the highest scores on these three real grading reports as your final score of this assignment. Clearly, if you do not update and resubmit your program after the deadline, the highest score will be the one right after the deadline. However, if you are not happy with the score on the first real grading report, you still have the chance to improve it by submitting your programs again. Notice that late submission will never decrease your final score.

Our grading scripts are evolving and hence we reserve rights to regrade your programs with a different script in a later time. Hence, the scores you see in the grading reports may not be final. Please report any bugs in our grading script if you find them.

Bug Reports

Please report any bugs in the codes we provide as well as any typos and errors in this handout and the grading reports to **Prof. Tsz-Chiu Au** at **chiu@unist.ac.kr**. We will look into the bug reports and fix the problems. If we have to release a new version of the codes to fix the bugs, we will announce it on our course webpage. Before you submit your program, you should check the announcement to see whether the codes have been updated. Please make sure that your program works with the latest version of the codes we provide.