CS247—Assignment 2 (Spring 2023)

R. Evans

Due Date 1: Friday June 14th, 5:00PM
Due Date 2: Friday June 21st, 5:00PM

Note: On this and subsequent assignments, you will be required to take responsibility for your own testing. As part of that requirement, this assignment is designed to get you into the habit of thinking about testing before you start writing your program. If you look at the deliverables and their due dates, you will notice that there is no `C++` code due on Due Date 1. Instead, you will be asked to submit test suites for `C++` programs that you will later submit by Due Date 2.
There will be a handmarking component in this assignment, whose purpose is to ensure that you are following an appropriate standard of documentation and style, and to verify any assignment requirements not directly checked by Marmoset. You are not expected to follow a specific style guide, however your code should be readable and you should code with good style. That is, abstract out functions instead of having repetitive code segments, follow good OOP style, as well as not writing overly complex code when unnecessary. Please be aware that if handmarking shows you have not followed the requirements of the question, correctness marks from test cases can partially or totally be taken away.

Some or all of these programs will ask you to develop C++ modules in order to make a provided test harness file which includes those modules work. Since you must use the main provided in the test harness to test your program, it is suggested that you first begin by providing empty implementations of all of the necessary functions. Then you will at least be able to compile, and as you implement each function you can recompile and test them immediately.

Note: All of your program must not leak resources. If your program produces the correct result but in doing so leaks resources then test case will not be considered as having been passed. We will use the tool valgrind to test your programs (which is available on the student servers) so you can too. You code must compile and run correctly on the student server.

Allowed headers for this assignment: For this assigment you may only use the headers `<iostream>`, `<utility>`, `<sstream>`, `<iomanip>`, and `<string>`. Inclusion of any other headers (not created by yourself) may result in a loss of some or all of your marks. Note, some of these headers are only included as they may be used in the provided code, you do not need to necessarily need to make use all of these headers yourself.

1. **Templated Lists**

   For this question you may not use any STL container, nor may you use any STL smart pointers. You must implement the class in question by managing memory yourself. These headers are already banned from the assignment, but this is to remind you.

   In this exercise, you will fill in the implementation of a `List` class.

   You may recall implementing a list as a series of linked Node classes in class, where the Node class was equipped with copy and move constructors and assignment operators. This time, however, while you will still be building a linked list of Nodes, this functionality will be implemented by the List class and **not** by each individual Node. YOU MUST NOT CHANGE the provided `Node` class.

The `List` class we wrote in class also only supported strings. Your `List` class should be templated with the ability to store any type of data inside the List.

In addition, you will also implement both iterators and constant iterators for the List class.

Starter code and method signatures have been provided in `list.h`, along with a sample executable. The public interface of list.h must stay the same. This means any additional methods or fields you add should be private.

The test harness `a2q1.cc` is provided with which you may interact with your list for testing purposes. The test harness is not robust and you are not to devise tests for it, just for the `List` class. Do not change this file.

Memory Management Requirements: You must manage your own memory, using `new` and `delete`. All copies must be deep copies (can test this behavior in sample executable).

Additionally your move constructor and move assignment operator must be constant time! Your code will be handmarked for these requirements, if you fail to meet them then you will lose some or all of your correctness marks.

a) **Due on Due Date 1:** Design the test suite `suiteq1.txt` for this program and zip the suite into `a2q1a.zip`.

b) **Due on Due Date 2:** Implement this in C++ and submit `list.h`

2. **Tier List Iterator** In this question, you are given an implementation of a `TierList` class: a ranked collection of *tiers*, each tier containing a set of elements ranked at that tier. Examples of tier lists can be found at https://tiermaker.com/.

   Traditionally the tiers in a tier list are ranked S to F, with S being the best tier and F being the worst tier. For simplicity, we will order our tiers by number, with 0 representing the best tier, 1 representing the second-best tier, 2 representing the third-best tier, etc.

   Your task is to implement an *iterator* for the `TierList` class that will iterate through the tier list from the items in the best tier to the items in the worst tier.

   In addition to the standard operations an iterator provides, you will also implement overloads of the `<<` and `>>` operators for the iterator. These will return a new iterator pointing to the start of the tier that is `n` tiers before/after (respectively) the tier of the current item.

   Starter code and method signatures have been provided in `tierlist.h`, along with a sample executable. **You may not change the contents of `tierlist.h`, other than by adding your own private methods, variables, and comments, i.e., the interface must stay exactly the same.**

   The test harness `a2q2.cc` is provided, with which you may interact with your tier list for testing purposes. **The test harness is not robust and you are not to devise tests for it, just for the `TierList` class. Do not change this file.**

   Implementation notes:

   - It is possible that one or more tiers could be empty.
     - `TierList::begin()` either sets the iterator to the first item of the first non-empty tier or sets the iterator to the end iterator if there are no non-empty tiers.
     - Calling `TierList::operator++` on an iterator pointing at the last item in a tier list must produce an iterator that compares equal to the tier list's end iterator.
     - If `TierList::iterator::operator++` lands you on an empty tier, you should return an iterator pointing to the first item in the next non-empty tier following that (or an end iterator if there are no remaining items).

- If an operation `>> n` lands you on an empty tier, you should return an iterator pointing to the first item in the next non-empty tier following the empty one (or an end iterator if there are no remaining items). Similarly, if an operation `<< n` lands you on an empty tier, return an iterator pointing to the first item in the next non-empty tier preceding the empty one (behaviour is undefined if this is not possible).

- One of the main challenges in this problem will be figuring out what to store in your iterator class for the tier list. Remember that an iterator functions primarily as an indication of location, so think about what information you would need to store in order to unambiguously identify a position in the tier list.
- Another challenge you are likely to face is how to represent the "end" iterator for a tier list (or perhaps even a "begin" iterator, in the case that the tier list is empty). There are a number of ways you can do it, and we leave it up to your ingenuity. One approach is to have a notion of a "dummy" iterator that can be used as a sentinel. As a potential help, if you find yourself needing a list iterator, but have no list to produce an iterator from, you can always use an expression like `List{}.begin()` or `List{}.end()` to produce a "dummy" list iterator. There exist solutions to this problem that use this technique; there also exist solutions that do not. The design is up to you!

a) **Due on Due Date 1:** Design the test suite `suiteq2.txt` for this program and zip the suite into `a2q2a.zip`.

b) **Due on Due Date 2:** Implement this in C++ and place the files `Makefile`, `a2q2.cc`, `tierlist.h`, `tierlist.cc`, `list.h` and `list.cc` in the zip file, `a2q2b.zip`. The `Makefile` should create an executable named `a2q2` when the command `make` is given.

3. **RegExp Evaluator**

In this problem you will create a program to evaluate regular expressions.

When it comes to processing general regular expressions, one of the first questions we must confront is that of *precedence*: does `c*d|a` mean `c*(d|a)` or `(c*d)|a`? We will assign the highest precedence to `*`, then concatenation, then `|`. For example, in the case of concatenation over `|`, we would interpret the regular expression `(a|b)(c|d)|e` as meaning `((a|b)(c|d))|e`.

We can give an inductive definition of regular expressions like so. A regular expression is:

- a disjunction of two regular expressions;
- or a concatenation of two regular expressions;
- or a star operator applied to a regular expression;
- or a single character.

For this problem, you are to complete classes corresponding to these four types of regular expression: `Disjunction`, `Concatenation`, `Star`, `Char`; with an abstract `RegExp` superclass from which they may inherit. For example, the regular expression `(c|d)*b` could be encoded as

```
new Concatenation{
  new Star{
    new Disjunction{
      new Char{'c'},
      new Char{'d'}
    }
  },
  new Char{'b'}
};
```

Your classes must provide a virtual method called `matches` that takes a `string` and answers `true` if the string exactly satisfies the regular expression (i.e., with no characters left over), and `false` otherwise. You may assume that the word in the `Word` class contains no special characters, or rather, if it does, that those characters should be taken literally.

We have provided a sample main program that performs the parsing of the regular expression for you. The implementation of the parsing and printing of regular expression is given to you in a `providedRegexp.o` object file. Your responsibility is to implement the `matches` method, the constructor, and the destructor for each of the classes.

a) **Due on Due Date 1:** Design the test suite `suiteq3.txt` for this program and zip the suite into `a2q3a.zip`.

b) **Due on Due Date 2:** Implement this in C++ and place the files `Makefile`, `a2q3.cc`, `regexp.h`, `regexp.cc`, and `providedRegexp.o` in the zip file, `a2q3b.zip`. The `Makefile` should create an executable named `a2q3` when the command `make` is given.