CS246—Assignment 3 (Fall 2017)

Due Date 1: Friday, October 13, 5pm
Due Date 2: Friday, October 20, 5pm
Due Date 3: Monday, October 23, 5pm

Questions 1a, 2a, 3a and 4a are due on Due Date 1; Question 1b, 3b and 4b are due on Due Date 2. Question 2b is due on Due Date 3

Note: You must use the C++ I/O streaming and memory management facilities on this assignment. Moreover, the only standard headers you may `#include` are `<iostream>`, `<fstream>`, `<sstream>`, `<iomanip>`, `<string>`, and `<utility>`. Marmoset will be programmed to **reject** submissions that violate these restrictions.

Note: Each question on this assignment asks you to write a C++ program, and the programs you write on this assignment each span multiple files. Moreover, each question asks you to submit a `Makefile` for building your program. For these reasons, we **strongly** recommend that you develop your solution for each question in a separate directory. Just remember that, for each question, you should be *in* that directory when you create your zip file, so that your zip file does not contain any extra directory structure.

Note: 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. Please code to a standard that you would expect from someone else if you had to maintain their code. Further comments on coding guidelines can be found here: `https://www.student.cs.uwaterloo.ca/~cs246/current/AssignmentGuidelines.shtml`

Note: You are not permitted to ask any public questions on Piazza about what the programs that make up the assignment are supposed to do. A major part of this assignment involves designing test cases, and questions that ask what the programs should do in one case or another will give away potential test cases to the rest of the class. Instead, we will provide compiled executables, suitable for running on `linux.student.cs`, that you can use to check intended behaviour. Questions found in violation of this rule will be marked private or deleted; repeat offences could be subject to discipline.

1. In this exercise, you will write a C++ class (implemented as a `struct`) that adds support for *rational numbers* to C++. In mathematics, a rational number is any number that can be expressed as a fraction `n/d` of two integers, a numerator `n` and a non-zero denominator `d`. Since `d` could be equal to 1, every integer is also a rational number. In our `Rational` class, rational numbers are always stored in their most simplified form e.g. 4/8 is stored as 1/2, 18/8 as 9/4. Additionally, negative rational numbers are stored with a negative numerator and a positive denominator e.g. negative a quarter is stored as -1/4 and not 1/-4.

   A header file (`rational.h`) containing all the methods and functions to implement has been provided in the `a3/a3q1` directory. You should implement the required methods and functions in a file named `rational.cc`

Implement a two parameter constructor with the following signature:
`Rational(int num = 0, int den = 1);`

To support arithmetic for rational numbers, overload the binary operators $+$, $-$, $*$ and $/$ to operate on two rational numbers. In case you have forgotten how these operations work, here is a refresher:

$\frac{a}{b} + \frac{c}{d} = \frac{ad+bc}{bd}$ $\quad \frac{a}{b} - \frac{c}{d} = \frac{ad-bc}{bd}$ $\quad \frac{a}{b} * \frac{c}{d} = \frac{ac}{bd}$ $\quad \frac{a}{b} / \frac{c}{d} = \frac{ad}{bc}$

Also, implement the following:

- A convenience unary (`-`) operator which negates the rational number.
- Convenience `+=`, `-=` operators where `a += b` has the effect of setting `a` to `a + b` (similarly for `-=`)
- The helper method `simplify()` which can be used to update a rational number to its simplest form.
- Accessor methods `getNumerator()` and `getDenominator()` that return the numerator and denominator of the rational number, respectively.
- Implement the overloaded input operator for rational numbers as the function:
  `std::istream &operator>>(std::istream &, Rational &);`

  The format for reading rational numbers is: an int-value-for-numerator followed by the / character followed by an int-value-for-denominator. Arbitrary amounts of whitespace are permitted before or in between any of these terms. A denominator must be provided for all values even if the denominator is 1 (this includes the rational number 0) e.g. the rational number 5 must be input as `5/1`.
- Implement the overloaded output operator for rational numbers as the function:
  `std::ostream &operator<<(std::ostream &, const Rational &);`

  The output format is the numerator followed by the / character and then the denominator without any whitespace. Rational numbers that have a denominator of 1 are printed as integers e.g. 17/1 is printed as 17.

A test harness is available in the file `a3q1.cc`, which you will find in your `a3/a3q1` directory. **Make sure you read and understand this test harness, as you will need to know what it does in order to structure your test suite.** Note that we may use a different test harness to evaluate the code you submit on Due Date 2 (if your functions work properly, it should not matter what test harness we use). You should not modify a3q1.cc

(a) **Due on Due Date 1**: Design a test suite for this program. Call your suite file `suiteq1.txt`. Zip your suite file, together with the associated `.in`, `.out`, and `.args` files, into the file `a3q1.zip`.

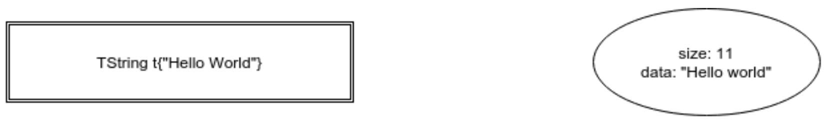
(b) **Due on Due Date 2**: Submit the file `rational.cc` containing the implementation of methods and functions declared in `rational.h`.

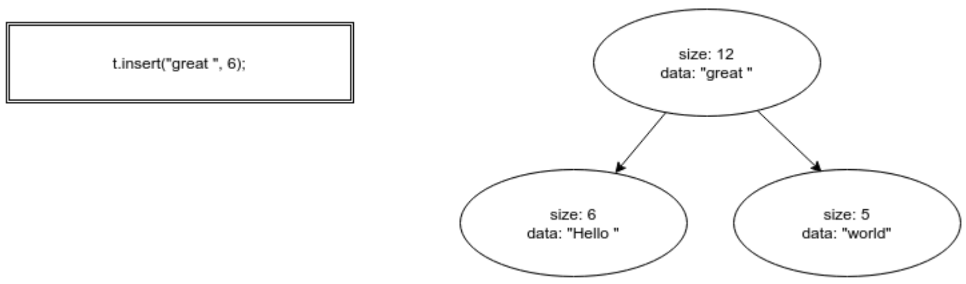
2. Coming from a C point of view it is common to think of strings as an array of characters. However, it can also be beneficial to consider a tree-like structure for strings, especially when we are working with very large strings which we want to modify often and in arbitrary locations (not just the end). Many text editors use this representation of a string since they quite often have very large strings which need to be modified often in various places. Using a tree representation means we can concatenate strings lazily without having to increase the size of a character array and copy over all the contents (by creating a new node and updating pointers). **As such you are not allowed to use string concatenation in your solution as it nullifies the usefulness of using a tree.**

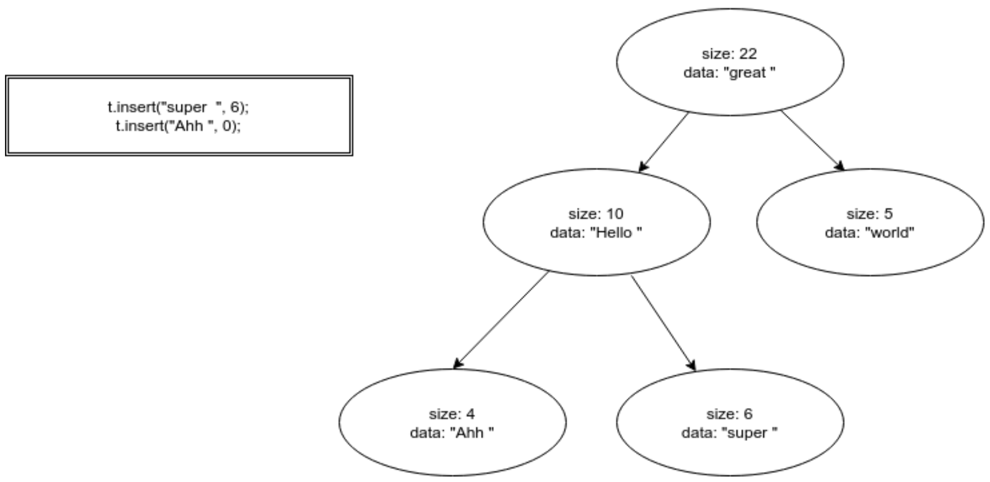
   Consider the following class definition for a tree-based string `TString` class (provided in files `tstring.h` and `tnode.h`):

```
struct TString {
    TNode *root;  // root of the tree

    TString(const std::string &);
    TString(const TString & );        // copy constructor
    ~TString();

    // Concatenate two TStrings, the returned TString should not have any
    // connection to the TStrings used to create it, that is no changes in
    // the operands in the future should reflect the returned TString and
    // vice versa.
    TString operator+( const TString & ) const;

    // Index operator. NOTE: returns a char & so the user can modify the char
    // in that index of the string.
    // Requires index < length of the string represented by the TString object.
    char &operator[] (const size_t index);

    // Insert the string denoted by the std::string into this TString at the
    // location immediately before the character located at index.
    // If index >= size of our string, append to the end of the string.
    void insert(const std::string &, const size_t index);
};
// Print the string represented by our TString.
std::ostream& operator<<(std::ostream& out, const TString &t);

struct TNode {
  // Size of the string represented by this node's left subtree, plus its own
  // string data size.
  size_t size;
  std::string data;
  TNode *left, *right;
};
```

When creating your string tree you should form it such that an in-order traversal of your tree produces the actual string you are representing. To structure the tree this way requires that whenever you insert a string into your tree you do it by creating at least one new node at the appropriate place for that index. So you must find the deepest node N in your tree for which the index i to insert at is either immediately before, immediately after, or in the middle of the data of N. If you need to insert before you need to make a new node that becomes N's new left child, updating the rest of subtree accordingly. If you need to insert after you need to make a new node that becomes N's new right child, updating the rest of the subtree accordingly. Lastly if the index you need to insert at is within the data of N then you must split N's data into two new nodes that become N's new left and right children (along with any necessary updates) and then set N's data to the string being inserted. When creating a new TString through concatenation of TStrings $t1$ and $t2$ the resulting tree must have a height of at most $max(h(t1), h(t2)) + 1$. Finally, all copies of a TString object should be deep copies. Consider the following example to help you visualize how to insert into a TString object.







A test harness is available in the file `a3q2.cc`, which you will find in your `a3/a3q2` directory. For your convenience, we have implemented a way to pretty print the tree structure representation of a TString at any time. This function has been provided to you as a compiled binary in the file `utility.o`. The 'd' option in the test harness can be used to invoke the pretty printer on any TString. **Make sure you read and understand this test harness, as you will need to know what it does in order to structure your test suite.** Note that we may use a different test harness to evaluate the code you submit on Due Date 2 (if your functions work properly, it should not matter what test harness we use). You should not modify a3q2.cc.

(a) **Due On Due Date 1**: Design a test suite for this program. The suite should be named `suiteq2.txt` and zip the suite into a zip file named `a3q2a.zip`.

(b) **Due On Due Date 3**: Full implementation of the TString class in C++. Your zip archive should contain at minimum `tstring.h`, `tnode.h`, `tstring.cc`, `tnode.cc`, the unchanged file `a3q2.cc`, and your Makefile. Your Makefile must create an executable named `a3q2`. Note that the executable name is case-sensitive. Any additional classes (if created) must each reside in their own `.h` and `.cc` files. Name the zip file `a3q2b.zip`

3. In this problem, you will implement a `Polynomial` class to represent and perform operations on single variable polynomials. We will use the `Rational` class from Q1 to represent the coefficients of the terms in a `Polynomial`. A polynomial can be represented using an array with the value at index `idx` used to store the coefficient of the term with exponent `idx` in the polynomial. For example, $(9/4)x^3 + (-7/3)x + 3/2$ is represented by the array of Rationals `{3/2,-7/3,0/1,9/4}`.

The `degree` of a polynomial is the exponent of the highest non-zero term (3 in the example just discussed). Therefore, an array of size `n+1` is required to store a polynomial of degree `n`. In order to not restrict the `Polynomial` class to a maximum degree, the array used to encode the polynomial is heap allocated.

You may assume that the coefficients of polynomials (given as input or produced through operations) can always be represented using a `Rational`. Additionally, there is no need to worry about over and under flow.

In the file `polynomial.h`, we have provided the type definition of `Polynomial` and signatures for the overloaded input and output operators for the `Polynomial` class. Implement all methods and functions. Place your implementation in `polynomial.cc`. **You are free to use your own implementation from Question 1 of the methods and functions in `rational.h` or use the implementation we have provided in the compiled binary file `rational.o`. Note that your implementation of polynomial.cc will be linked to our implementation of `Rational` during testing.**

The zero parameter constructor for `Polynomial` should create the `zero` polynomial i.e. a polynomial with no non-zero coefficient.

The overloaded arithmetic operators work identically to single variable polynomial arithmetic. For the division operation, two methods are to be implemented; `operator/` should return the quotient after long division and `operator%` should return the remainder.

The input operator reads the input stream till the end of the line and uses the read input to modify an existing `Polynomial` object. The input format is a pair for each non-zero term in the polynomial in decreasing exponent values with no exponent repeated. The first value in each pair is a rational number and follows the input format for `Rational` numbers as specified in Q1. This is the coefficient. The second value is a non-negative integer and represents the exponent of this term. Arbitrary amount of whitespace (excluding newline) is allowed within each pair and between pairs. For example, the input `3/5 5 -2/5 2 1/2 1 3/7 0` represents the polynomial `(3/5)x^5 + (-2/5)x^2 + (1/2)x + (3/7)`.

The output operator prints polynomials as the addition of terms in decreasing exponent order. Each term is output as `(a/b)x^n` and subject to the following additional requirements and exceptions:

- Terms with zero coefficients are not printed.
- Coefficients are printed using the output format for `Rational` numbers as described in Q1.
- A term whose exponent is 1 is printed as `(a/b)x` and a term whose exponent is 0 is printed as `(a/b)`.

An example of the output produced by the output operator is shown above during the discussion of the input operator.

A test harness is available in the file `a3q3.cc`, which you will find in your `a3/a3q3` directory. **Make sure you read and understand this test harness, as you will need to know what it does in order to structure your test suite.** Note that we may use a different test harness to evaluate the code you submit on Due Date 2 (if your functions work properly, it should not matter what test harness we use). You should not modify a3q3.cc

(a) **Due On Due Date 1**: Design a test suite for this program. The suite should be named `suiteq3.txt` and zip the suite into a zip file named `a3q3a.zip`.

(b) **Due On Due Date 2**: Full implementation of the Polynomial class in C++. Your zip archive should contain at minimum the unchanged file `a3q3.cc`, `polynomial.h`, `polynomial.cc` and your Makefile. Your Makefile must create an executable named `a3q3`. Note that the executable name is case-sensitive. Any additional classes (if created) must each reside in their own `.h` and `.cc` files. Name the zip file `a3q3b.zip`

4. Consider a shared document (e.g., wiki article, Piazza post, Google document, email thread, software under version control) being edited by several users. Each time someone edits the document (e.g., responds to the thread, revises the software, etc.) the result is a new version of the document. Some edits are made to the latest version; other edits are made to older versions of the document. When an older version of the document is edited, we obtain a new "latest" version (in addition to the one we already had)—the document has "branched".

   Edits of this kind can be modelled as a collection of linked lists with a common tail. For simplicity, the linked lists in this problem will contain simply ints, but in reality they could contain document edits, email replies, etc. An example of such a collection of lists is pictured below:

```
1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 0
   /                    /
10-     13 -> 12 -> 11-
             /
         14-

heads:  1 10 13 14
```

   In the diagram above, 0 is the original document. 7 is an edit to 0; 6 is an edit to 7; 5 and 11 are both edits to 6; 4 is an edit to 5; 12 is an edit to 11; etc. This example has four "latest" versions: 1, 10, 13, and 14.

   In this problem, you will simulate a sequence of edits to a shared document. Your data structure will be a collection of linked list nodes, where multiple nodes could be pointing to the same "next" node. Along with these nodes, you will have an array (of fixed size 10) of "head" pointers, so that we can track the full collection of "latest" documents.

   One of the biggest challenges with this kind of data structure is proper deallocation. If two nodes point to the same "next" node, their destructors can't both delete that node; that would result in a "double free", which leads to undefined behaviour.

   To solve this problem, we will encode a notion of *ownership* into our nodes. A node that is attached to a head node becomes the head node for that branch, and owns it successor. A node attached to any other node becomes a head node for a new branch, but does not own its successor (because another node already does). When the data structure is deallocated, a node should only delete its successor if it owns its successor. Your `Node` class will, therefore, contain a boolean flag called `ownsSuccessor` that is set accordingly.

   As stated above, your solution will support up to 10 distinct head nodes. Any attempt to create more than 10 distinct heads constitutes undefined behaviour and must not be tested.

   We will provide a test harness that you can use to test your solution. The test harness recognizes the following commands:

   - `print n` (where `n` ranges from `0` to `9`) prints the list starting from the `n`th head. For example (assuming that `13` is head number 2):

     ```
     print 2
     13 12 11 6 7 0
     ```

   - `attach n m x` (where `n` ranges from `0` to `9`). Attach a new node with label number `x` such that its `next` field points at the `m`th item (starting at 0) of the list headed at list `n`. For example, under the same assumptions as above:

     ```
     attach 2 3 17   (this would create a new list head, head number 4)
     print 4
     17 6 7 0
     ```

It is undefined behaviour if `m` is too large to denote a valid list item.

- `mutate n m x` (where `n` ranges from `0` to `9`). Mutate the `m`th item of the `n`th list, such that it now stores `x`. We wouldn't normally see this in a versioning system (i.e., we don't directly mutate old versions), but this command helps us to see that the tails of the lists are indeed shared. For example:

```
mutate 0 2 55
print 1
10 2 55 4 5 6 7 0   (assuming 10 is head number 1)
```

The data structure will start out with a single node containing 0, and this will be the only head, until edits are made. Note that nodes do not have to contain distinct labels, and that the first node can be mutated to hold something other than 0.

(a) **Due on Due Date 1**: Design the test suite `suiteq4.txt` for this program and zip the suite into `a3q4a.zip`.

(b) **Due on Due Date 2**: Implement this in C++ and place your `Makefile`, `a3q4.cc` and all `.h` and `.cc` files that make up your program in the zip file, `a3q4b.zip`.