CSCI 2270, Fall 2014
HW4, Doubly linked list polynomial (this assignment was written by Professor Michael Main, and lightly edited by me)

The Assignment:

Implement a **`polynomial`** class that uses a doubly-linked list of **`polynodes`** to store the **`polynomial`**'s terms. Each **`polynode`** of the list holds the **`coefficient`** and **`exponent`** for one term. The terms are kept in order from smallest to largest **`exponent`**. Each **`polynomial`** also maintains a pointer called **`recent_ptr`** to the most recently accessed **`polynode`**.

Purposes:

Ensure that you can write a class that uses a linked list, with a roving cursor (recent_ptr) to make the list more efficient.

Due Date:

Part 1 (steps 1-6): Sunday, April 6, 11:50pm, via Moodle.
Part 2 (step 7 and extra credit): Sunday, April 13, 11:50pm, via Moodle.

Files that you must use:

1. **`poly.h`**: The header file for the new **`polynomial`** class.
2. **`poly.cxx`**: The implementation file for the new **`polynomial`** class. You'll complete the code in this file.

Other files that you may find helpful:

1. **`polytest.cxx`**: A simple interactive test program.
2. **`polyexam.cxx`**: A non-interactive test program that will be used to grade the correctness of your **`polynomial`** class.
3. **`Makefile`**: compile the test files with the **`polynomial`** class.

The Polynomial Class with a Linked List
Discussion of the Assignment

As indicated above, you will write the **polynomial** class, which uses a doubly-linked list to store **coefficients**.

Step 1. **Polynomials** keep linked lists of **polynodes**. Look at the **polynode** class in **poly.h**. This class defines a single term (**coefficient** plus **exponent**) in a **polynomial**. All of the functions for a **polynode** are defined for you already in this header file.

Please also look at the **private** member variables for the **polynomial** class, also in **poly.h**:

```
private:


    // Head pointer for list of nodes
    polynode* head_ptr;
    // Tail pointer for list of nodes
    polynode* tail_ptr;
    // Pointer to most recently used node
    mutable polynode* recent_ptr;
    // Current degree of the polynomial
    unsigned int current_degree;
```

The meaning of the **mutable** keyword will be covered in class. But a brief explanation now: Our plan is to keep the **recent_ptr** always pointing to the most recently used **polynode**. For example, when we call the **coefficient** member function, we will move the **recent_ptr** to point to the **polynode** that contains the requested **exponent**. With a normal member variable, we could not do this (since the **coefficient** is a **const** member function and it is forbidden from changing normal member variables). So the meaning of the **mutable** keyword is to indicate that changing the member variable does not change the value of the **polynomial** in a meaningful way (and therefore, the compiler will let **const** member functions change a **mutable** variable).

In your **poly.cxx**, write a clear description of how the member variables of a **polynomial** are used. The **head_ptr** and **tail_ptr** are the head and tail pointers for a doubly-linked list of **polynodes** that contain the **polynomial**'s

terms in order from smallest to largest **exponent**. To make certain operations simpler, we will always keep a **polynode** for the zero-order (x^0) term. But other **polynodes** are kept only if the **coefficient** is non-zero. We always maintain **recent_ptr** as a pointer to some **polynode** in the list--preferably the most recently used **polynode**. The degree of the **polynomial** is stored in **current_degree** (using zero for the case of all zero **coefficients**).

Step 2: The **poly.h** header file contains a prototype for a **private** member function that will make it easier to implement everything else:

```
// A private member function to aid the other functions:
void set_recent(unsigned int exponent) const;
```

The **set_recent** function will set the **recent_ptr** to the **polynode** that contains the requested **exponent**. If no such **exponent** exists, then **recent_ptr** should be set to the last **polynode** that is still less than the specified **exponent**. Note that **set_recent** is a **const** member function, but that it can still change the **mutable recent_ptr** member variable. My implementation of **set_recent** used four cases:

If the requested **exponent** is zero, then set **recent_ptr** to the head of the list.

Else if the **exponent** is greater than or equal to the **current_degree**, then set **recent_ptr** to the tail of the list.

Else if the **exponent** is smaller than the **exponent** in the recent **polynode**, then move the **recent_ptr** backward as far as needed.

Else move the **recent_ptr** forward as far as needed.

Step 3. Implement the constructors, destructor, and assignment operator.

For the default constructor, you could start by creating a valid empty **polynomial** (with a head **polynode** that has **exponent** and **coefficient** of zero).

Then call **assign_coef** to set the one term. (Note that you need to make a 2-**polynode** list if the **exponent** here is greater than 0.)

The copy constructor should also start by creating a valid empty **polynomial**. Then have a loop that steps through the terms of the **source** (using **source.next_term**). Each term of the **source** is placed in the **polynomial** (using **assign_coef**).

The assignment operator first checks for a self-assignment, then clears out the terms (using the **clear** function). Finally, it has a loop (similar to the copy constructor) to copy the terms of the **source**.

Step 4. Implement the **assign_coef** function. After **calling set_recent(exponent)**, my implementation had these cases:

If there is a zero **coefficient** and **exponent** greater than **current_degree**, return with no further work.

Else if there is currently no **polynode** for the given **exponent** (so that **recent_ptr->exponent()** is less than the new **exponent**), make a new **polynode** and connect it into the list.

Else if the **coefficient** is non-zero or the **exponent** is zero, just change the **coefficient** of an existing **polynode** in the list.

Else if the **exponent** equals the **current_degree**, the **coefficient** is zero (otherwise we would have hit the previous case), so we remove the tail **polynode**, set the **recent_ptr** to the new tail, and reduce the **current_degree**.

Else, in this last case, we must have a new zero **coefficient** for a **polynode** that is neither the head nor the tail. We just remove this **polynode** and set the **recent_ptr** to the *previous* **polynode** in the list.

Step 5. Implement the **add_to_coef** function, using logic like what you used in step 4.

Step 6. Write **next_term** and **previous_term** so that they each start with **set_recent**.

Hint for **`next_term`**: Start by calling **`set_recent(exponent)`**. Normally, the right answer is then in the **`polynode`** after **`recent_ptr`** (but there is one exception that you should handle).

Hint for **`previous_term`**: Start by checking the special case where **`exponent`** is zero. Then call **`set_recent(exponent – 1)`**. Normally, the right answer is then in the **`recent_ptr polynode`** (but again, there is one special case that you need to handle--the case where the **`polynode`** has **`exponent`** and **`coefficient`** of zero).

Step 7. When these above functions are working, use them to write the **`operator +`**, **`operator -`**, and **`operator *`** functions to compute the sum, difference, and product of two **`polynomials`**. Write the **`derivative()`** function to compute the derivative of a **`polynomial`**, using the power rule. And write **`eval`** (and its sibling function, **`operator()`**) to evaluate a **`polynomial`** at a particular value of x.

For 5% extra credit, you may write the root-finding function **`find_root`**, which uses Newton's method to locate a value of x that makes the **`polynomial`** (evaluated at x) equal to zero.