CISS350: Data Structures and Advanced Algorithms
Assignment 8

Objectives:

1. Applications of linked list, stacks, queues, and deques.

Q1. [Introduction to Lexing]

If you're given a string such as

`"10 + 21 * (33 - 47)"`

which correspond to a mathematical expression and you want to evaluate it, you want to convert this into a sequence of meaning items:

`"10", "+", "21", "*", "(", "33", "-", "47", ")"`

where each string in this sequence has meaning. For instance `"10"` correspond to the meaning of "ten", etc. The strings above (such as `"10"`) are called lexemes.

It is convenient to attach types to these lexemes. For instance we want to think of `"10"` as a lexeme that is different from the lexeme `"+"`. Using object-oriented programming, we will convert these lexemes into different token types. For instance `"10"` will be converted into an integer token object and `"+"` will be converted into a plus token object.

Here are the classes:

```
class Token
{};

class IntToken: public Token
{
public:
    int value;
};

class PlusToken: public Token
{};

class MinusToken: public Token
{};

class MultToken: public Token
{};

class DivToken: public Token
{};

class LParenToken: public Token // Left parenthesis
{};
```

```
class RParenToken: public Token // Right parenthesis
{};
```

(You don't have to worry about the mod % operator.)

The goal is to write a program that gets a string such as

```
"10 + 21 * (33 - 47)"
```

from the user and then produce a `std::vector` of tokens.

You must provide code to print all the above tokens. For instance

```
std::cout << IntTok(5) << '\n'; // Prints IntTok(5)
std::cout << PlusTok() << '\n'; // Prints PlusTok
```

You must also provide code to print a `std::vector` of tokens.

```
std::vector< Token > toks;
toks.push_back(IntTok(5));
toks.push_back(PlusTok());
std::cout << toks << '\n'; // Prints [IntTok(5), PlusTok]
```

Here's an execution of the program:

Test 1

```
10 + 21 * (33 - 47)
[IntTok(10), PlusTok, IntTok(21), MultTok, LParenTok, IntTok(33), MinusTok, IntTok(47), RParenTok]
```

Ignore "negative of" (the unary −). In other words, when your program sees a −, generate a `MinusTok`.

Test 2

```
10 + 21 * (33 - -47)
[IntTok(10), PlusTok, IntTok(21), MultTok, LParenTok, IntTok(33), MinusTok, MinusTok, IntTok(47), RParenTok]
```

Note. In a compiler, the process of analysing a source file is not broken up into lexing as a complete first step:

```
get string s from program source file
get toks, a vector of tokens, from s (lexing)
analyze tokens (parsing)
```

The process actually extracts enough characters from a source file to form a token and then pass the token to the part of the compiler that analyzes the structure of the program. In other words, the compiler alternates between lexing and analysis:

```
get string s from program source file
loop as long as there are characters in s:
    extract enough characters from s to get the next token t
    continue analyzing structure of program with new token t
```

So the process is more like lex-parse-lex-parse-lex-parse-... The reason is because it's too costly to lex the whole string – what if there's a very early program error in an extremely huge source file?

C++ `std::stack`, `std::queue`, and `std::list` classes

Using the web, read up on the `std::stack` and `std::queue` class templates. In the following, you should use these class.

Here's an example of some basic methods in the `std::stack` class template:

```
#include <iostream>
#include <stack>

int main()
{
    std::stack< int > s;
    std::cout << s.empty() << ' ' << s.size() << '\n';

    std::cout << "pushing 42, 23, 6 ...\n";
    s.push(42);
    std::cout << s.empty() << ' ' << s.size() << ' ' << s.top() << '\n';

    s.push(23);
    std::cout << s.empty() << ' ' << s.size() << ' ' << s.top() << '\n';

    s.push(6);
    std::cout << s.empty() << ' ' << s.size() << ' ' << s.top() << '\n';

    std::cout << "changing top to 1000 ...\n";
    s.top() = 1000;
    std::cout << s.empty() << ' ' << s.size() << ' ' << s.top() << '\n';

    std::cout << "pop until empty ...\n";
    while (!s.empty())
    {
        std::cout << s.empty() << ' ' << s.size() << ' ' << s.top() << '\n';
        s.pop();
    }

    std::cout << s.empty() << ' ' << s.size() << '\n';

    std::cout << "copy constructor, operator=, operator== ...\n";
    s.push(5); s.push(3); s.push(7);
    std::stack< int > t = s;
    std::cout << (s == t) << '\n';

    while (!t.empty()) t.pop();
    t = s;
    std::cout << (s == t) << '\n';

    return 0;
}
```

Run the program and look at the output. Check the web for more methods for this class.

Here's an example of some basic methods in the `std::queue` class template:

```
#include <iostream>
#include <queue>

int main()
{
    std::queue< int > q;
    std::cout << q.empty() << ' ' << q.size() << '\n';

    std::cout << "Putting 42, 23, 6 into the queue ... \n";
    q.push(42);
    std::cout << q.empty() << ' ' << q.size() << ' '
              << q.front() << ' ' << q.back() << '\n';
    q.push(23);
    std::cout << q.empty() << ' ' << q.size() << ' '
              << q.front() << ' ' << q.back() << '\n';
    q.push(6);
    std::cout << q.empty() << ' ' << q.size() << ' '
              << q.front() << ' ' << q.back() << '\n';

    std::cout << "changing front to 1000 ...\n";
    q.front() = 1000;
    std::cout << q.empty() << ' ' << q.size() << ' '
              << q.front() << ' ' << q.back() << '\n';

    std::cout << "changing back to -1000 ...\n";
    q.back() = -1000;
    std::cout << q.empty() << ' ' << q.size() << ' '
              << q.front() << ' ' << q.back() << '\n';

    std::cout << "dequeue until empty ...\n";
    while (!q.empty())
    {
        std::cout << q.empty() << ' ' << q.size() << ' '
                  << q.front() << ' ' << q.back() << '\n';
        q.pop();
    }

    std::cout << "copy constructor, operator=, operator== ...\n";
    q.push(5); q.push(3); q.push(7);
    std::queue< int > r = q;
    std::cout << (r == q) << '\n';

    while (!r.empty()) r.pop();
    r = q;
    std::cout << (r == q) << '\n';
    return 0;
}
```

Run the program and look at the output. Check the web for more methods for this class.

If `s` is a stack and `q` is a queue, then:

```
s.empty() -- true if s is empty; otherwise false
q.empty() -- true if q is empty; otherwise false

s.size()  -- size of s
q.size()  -- size of q

s.push(x) -- push x onto top of s
q.push(x) -- enqueue x into back of q

s.pop()   -- remove top of s
q.pop()   -- remove front of q

s.top()   -- reference to the top of stack s
q.front() -- reference to the front of queue q
q.back()  -- reference to the end of queue q
```

C++ `std::list` class is a doubly-linked list class. Here's an example of some basic methods in the `std::list` class template:

```
#include <iostream>
#include <list>

int main()
{
    std::list< int > list;
    std::cout << list.empty() << ' ' << list.size() << '\n';

    list.push_front(1);
    list.push_back(5);
    list.push_front(0);
    std::cout << list.front() << ' ' << list.back() << '\n';
    std::cout << list.empty() << ' ' << list.size() << '\n';

    list.front() = 1000;
    list.back() = 9000;
    std::cout << list.front() << ' ' << list.back() << '\n';
    std::cout << list.empty() << ' ' << list.size() << '\n';

    std::cout << "iterator ...\n";
    std::list< int >::iterator p = list.begin();
    std::cout << *p << '\n';
    ++p;
    std::cout << *p << '\n';
    ++p;
    std::cout << *p << '\n';
    --p;
    std::cout << *p << '\n';

    std::cout << "loop using an iterator ... ";
    for (typename std::list< int >::iterator p = list.begin();
         p != list.end();
         ++p)
    {
        std::cout << (*p) << ' ';
    }
    std::cout << '\n';

    std::cout << "loop using a constant iterator (if list is a constant) ... ";
    for (typename std::list< int >::const_iterator p = list.begin();
         p != list.end();
         ++p)
    {
        std::cout << (*p) << ' ';
    }
    std::cout << '\n';

    std::cout << "erase() ...\n";
    list.erase(p);
    std::cout << list.front() << ' ' << list.back() << '\n';
    std::cout << list.empty() << ' ' << list.size() << '\n';

    std::cout << "copy constructor, operator=, operator== ...\n";
    std::list< int > list2 = list;
    std::cout << (list2 == list) << '\n';
    list2 = list;
    std::cout << (list2 == list) << '\n';

    std::cout << "clear() ...\n";
    list.clear();
    std::cout << list.front() << ' ' << list.back() << '\n';
    std::cout << list.empty() << ' ' << list.size() << '\n';

    return 0;
}
```

Run the program and look at the output. Check on the web for more methods for this class. A constant iterator cannot modify the value that it points to. In other words if `p` is a constant iterator, you cannot do

```
*p = newvalue;
```

If list is a `std::list` object, then

```
list.empty()       -- true if list is empty; otherwise false

list.size()        -- size of list

list.push_front(x) -- insert x at head
list.push_back(x)  -- insert x at tail

list.pop_front()   -- remove front of list
list.pop_back()    -- remove back of list

list.front()       -- reference to the front/head of list
list.back()        -- reference to the end/tail of list

list.clear()       -- delete all nodes from list
list.erase(p)      -- delete node that iterator p points to
```

Here `p` is an iterator of the list (see example code), i.e., it's an object that acts like a pointer to the list.

Q2. [Polish notation calculator]

Write a program that evaluates an integer expression written using the Polish notation. For simplicity, assume that the user does not enter negative numbers. Make sure you test cases where the integer value has more than one digit.

Test 1

```
* 1 - + 2 - 3 5 4
-4
```

Test 2

```
* 1
ERROR
```

Try out more test cases of your own.

Q3. [Reverse polish notation calculator]

Write a program that evaluates an integer expression written using the reverse polish notation. For simplicity, assume that the user does not enter negative numbers. Make sure you test cases where the integer value has more than one digit.

Test 1

```
1 2 3 5 + - 4 - *
-10
```

Test 2

```
1 +
ERROR
```

Try out more test cases of your own.

Q4. [Infix calculator]

Write a program that evaluates an integer expression written using the infix notation. For simplicity, assume that the user does not enter negative numbers. Make sure you test cases where the integer value has more than one digit.

Test 1

```
0 - ( 1 + 2 + 3 * ( 4 - 5 ) + 6 )
-6
```

Test 2

```
0 - 6 )
ERROR
```

Try out more test cases of your own.

Q5. [Infix to reverse polish notation] [OMIT]

Write a program that converts an integer expression written using the infix notation into RPN.

Test 1

```
0 - 1 + 2 * 3 - 4 / 5 * 6
0 1 - 2 3 * + 4 5 / 6 * -
```

Try out more test cases of your own.

[Note: As a vector of tokens, the expression is `[IntTok(0), IntTok(1), MinusTok, IntTok(2), IntTok(3), MultTok, PlusTok, IntTok(4), IntTok(5), DivTok, IntTok(6), MultTok, MinusTok]`. Of course it's easy to write a function that takes a vector of tokens and print the tokens as an RPN string.]

Q6. [Sparse polynomial]

Using `std::list`, implement an integer polynomial class where zero coefficient terms are not stored. (The case of a class template is easy once the integer version is done.)

```
// A monomial is a term in a polynomial. For instance 5 * x ^ 3 is a
// monomial. For this term, we store (3, 5), i.e., the degree and the
// coefficient.
class Monomial
{
public:
    int degree;
    int coefficient;
};

class Polynomial
{
public:
private:
    std::list< Monomial > p; // The term with smallest degree is at the head.
};
```

The following describes the expected usage of the class:

```
Polynomial p("3x^2 + 7x^12 - 9x^100 + -2x^32 + x^5 + 0x^6 + 6x^2 + 32 + 1x^0");
std::cout << p << '\n'; // prints -9x^100 - 2x^32 + 7x^12 + x^5 + 9x^2 + 33

Polynomial p0(42);      // p0 = 42 as a polynomial
Polynomial p1(p);       // p1 = p as a polynomial

std::cout << (p0 == p) << '\n';
std::cout << (p1 != p) << '\n';
std::cout << (5 == p) << '\n';
std::cout << (p != 5) << '\n';

std::cout << deg(p) << '\n'; // degree of p

Polynomial q("x^2 + x + 1");

p += q;
p -= q;
p *= q;

p += 42;
p -= 42;
```

```
p *= 42;

Polynomial r; // r = 0 as a polynomial

r = p - q;
r = p * q;

r = 42 + p;
r = p + 42;

r = 42 - p;
r = p - 42;

r = 42 * p;
r = p * 42;

r = 42 * p;
```

Make sure you examine the constructors and output carefully. In particular, note that in the constructors, the input string need not list monomials in any particular order by degree. Also, in the printout of the polynomials, higher degree terms are printed first.

Note. It's a good exercise to use your `Rational` class (to model fractions) and create a polynomial of rational coefficients. Then you can also perform long division to get quotients and remainders.