# FIC - CMPT 135 2023-01 - Assignment 2

## Due Date and Time: Wednesday 15 March 2023 at 11:55PM

## Instructor: Dr. Yonas T. Weldeselassie (Ph.D.)

In this assignment, we will be working with C++ classes to learn data structures in C++. A data structure is a programmer defined data type (either using a class or using a struct) in order to store data in some structured way. To this end, we will build C++ classes to represent the nodes of linked lists and the linked list data structure. We will then use the linked list data structure in order to store the unsigned binary, sign and magnitude binary, and two's complement binary representations of decimal numbers.

For a detailed description of the linked list data structure, please refer to the supplementary material posted on Moodle together with this assignment.

## Part I: Representing Linked Lists with C++ class

We may design a C++ class in order to represent a linked list. From our discussion in the supplementary material, we observe that a linked list class requires only one member variable which is the head pointer and then we could implement all the functions discussed in the supplementary material as member functions.

You are provided the following `Node` class declaration and definition as well as `LinkedList` class declaration. The definitions of the member and friend functions of the `LinkedList` class are almost identical to the non-member functions discussed in the supplementary material. The only difference is that the head pointer parameter of the non-member functions discussed in the supplementary material is removed from these member and friend functions because the head pointer is a member variable of the class and therefore each of these member and friend functions has access to it and doesn't need it as a parameter.

```
#include <iostream>
#include <cmath>
#include <cassert>
using namespace std;
class Node
{
typedef Node* NodePtr;
private:
        int data;
        NodePtr link;
public:
        Node();
        Node(const int &);
        Node(const Node &);
        int getData() const;
        NodePtr getLink() const;
        void setData(const int &);
        void setLink(const NodePtr &);
        friend ostream& operator << (ostream &, const Node &);
};
typedef Node* NodePtr;
Node::Node() : data(0), link(nullptr) {}
Node::Node(const int &d) : data(d), link(nullptr){}
Node::Node(const Node &n) : data(n.data), link(n.link){}
int Node::getData() const { return data; }
NodePtr Node::getLink() const { return link; }
void Node::setData(const int &d) { data = d; }
void Node::setLink(const NodePtr &p) { link = p; }
ostream& operator << (ostream& out, const Node& n)
```

```
{
	out << n.data;
	return out;
}
typedef Node* NodePtr;
class LinkedList
{
private:
	NodePtr head;
public:
	LinkedList(); //default constructor: assigns the head pointer member variable a nullptr value
	LinkedList(const LinkedList &); //copy constructor (deep copy)
	~LinkedList(); //destructor (must delete all the nodes from the heap)
	LinkedList& operator= (const LinkedList &); //Assignment operator (deep copy)
	bool isEmpty() const; //return true if the length of the calling object is 0 and false otherwise
	NodePtr getHeadPtr() const; //return the head member variable of the calling object
	int getLength() const; //return the number of nodes in the calling object
	void head_insert(const int &); //as described in the supplementary material
	NodePtr search_node(const int &) const; //as described in the supplementary material
	void insert_after(const NodePtr &, const int &) const; //as described in the supplementary
material
	void remove_node(const NodePtr &); //as described in the supplementary material
	void remove_node(const int &); //as described in the supplementary material
	void remove_all(const int &); //as described in the supplementary material
	void tail_insert(const int &); //as described in the supplementary material
	void insert_before(const NodePtr &, const int &); //as described in the supplementary material
	friend ostream& operator << (ostream&, const LinkedList &); //Implemented for you
};
ostream& operator << (ostream &out, const LinkedList &LL)
{
	if (LL.isEmpty())
		out << "EMPTY LINKED LIST";
	else
	{
		NodePtr temp = LL.head;
		while(temp != nullptr)
		{
			out << *temp << " ";
			temp = temp->getLink();
		}
	}
	return out;
}
```

You are required to implement the `LinkedList` class taking into account the following restrictions and requirements.

## Restrictions and Requirements

- You are not allowed to add or remove any include directive, namespace, member function, or friend function to the provided Node and `LinkedList` classes.
- You are not allowed to declare, define, or use any container variables (objects) such as static arrays, dynamic arrays, or any STL container such as vectors in your `LinkedList` class definition.
- You are not allowed to use any STL algorithm in your `LinkedList` class definition.

## Testing the Node and LinkedList classes

You are provided the following test program and its sample run output in order to help you test your program. Please note that you don't need to submit this test program when you submit your work. It is only provided to help you confirm your classes are built correctly and you can then use them in the next section of the

assignment. Please make sure this test program works without any syntax, runtime, linking, or semantic errors in order for you to be successful in the next section.

```
int main()
{
	//Test the default constructor
	LinkedList LL1;
	cout << "Default object LL1 is " << LL1 << endl;

	//Let us test the copy constructor, destructor and assignment operator
	//We will perform several tests to make CERTAIN these member functions are correct
	//DO NOT PROCEED TO THE NEXT SECTION BEFORE MAKING CERTAIN THAT THESE
	//MEMBER FUNCTIONS ARE WORKING CORRECTLY WITHOUT ANY SYNTAX, RUNTIME, OR SEMANTIC ERRORS.

	for (int testCase = 1; testCase <= 5; testCase++)
	{
		cout << endl << "Test Case #" << testCase << endl;
		//Test the head insert member function
		for (int i = 1; i <= 10; i++)
			LL1.head_insert(testCase*i+1);
		cout << "After inserting ten numbers using head insert, LL1 is " << LL1 << endl;

		//Test the copy constructor
		LinkedList LL2 = LL1;
		cout << "After constructing LL2 which is a copy of LL1, LL2 is " << LL2 << endl;
		cout << "LL1 is still " << LL1 << endl;

		//Test destructor
		LL1.~LinkedList();
		cout << "After destruction, LL1 is " << LL1 << endl;
		cout << "LL2 is still " << LL2 << endl;

		//Test assignment operator
		LL1 = LL2;
		cout << "After assigning LL1 the value of LL2, LL1 is now " << LL1 << endl;
		cout << "LL2 is still " << LL2 << endl;

		//Test destructor
		LL2.~LinkedList();
		cout << "After destruction, LL2 is " << LL2 << endl;
		cout << "LL1 is still " << LL1 << endl;

		//Test destructor
		LL1.~LinkedList();
		cout << "After destruction, LL1 is " << LL1 << endl;
		cout << "LL2 is still " << LL2 << endl;
	}

	system("Pause");
	return 0;
}
```

### Sample Run Output

```
Default object LL1 is EMPTY LINKED LIST

Test Case #1
After inserting ten numbers using head insert, LL1 is 11 10 9 8 7 6 5 4 3 2
After constructing LL2 which is a copy of LL1, LL2 is 11 10 9 8 7 6 5 4 3 2
LL1 is still 11 10 9 8 7 6 5 4 3 2
After destruction, LL1 is EMPTY LINKED LIST
LL2 is still 11 10 9 8 7 6 5 4 3 2
After assigning LL1 the value of LL2, LL1 is now 11 10 9 8 7 6 5 4 3 2
LL2 is still 11 10 9 8 7 6 5 4 3 2
After destruction, LL2 is EMPTY LINKED LIST
```

```
LL1 is still 11 10 9 8 7 6 5 4 3 2
After destruction, LL1 is EMPTY LINKED LIST
LL2 is still EMPTY LINKED LIST

Test Case #2
After inserting ten numbers using head insert, LL1 is 21 19 17 15 13 11 9 7 5 3
After constructing LL2 which is a copy of LL1, LL2 is 21 19 17 15 13 11 9 7 5 3
LL1 is still 21 19 17 15 13 11 9 7 5 3
After destruction, LL1 is EMPTY LINKED LIST
LL2 is still 21 19 17 15 13 11 9 7 5 3
After assigning LL1 the value of LL2, LL1 is now 21 19 17 15 13 11 9 7 5 3
LL2 is still 21 19 17 15 13 11 9 7 5 3
After destruction, LL2 is EMPTY LINKED LIST
LL1 is still 21 19 17 15 13 11 9 7 5 3
After destruction, LL1 is EMPTY LINKED LIST
LL2 is still EMPTY LINKED LIST

Test Case #3
After inserting ten numbers using head insert, LL1 is 31 28 25 22 19 16 13 10 7 4
After constructing LL2 which is a copy of LL1, LL2 is 31 28 25 22 19 16 13 10 7 4
LL1 is still 31 28 25 22 19 16 13 10 7 4
After destruction, LL1 is EMPTY LINKED LIST
LL2 is still 31 28 25 22 19 16 13 10 7 4
After assigning LL1 the value of LL2, LL1 is now 31 28 25 22 19 16 13 10 7 4
LL2 is still 31 28 25 22 19 16 13 10 7 4
After destruction, LL2 is EMPTY LINKED LIST
LL1 is still 31 28 25 22 19 16 13 10 7 4
After destruction, LL1 is EMPTY LINKED LIST
LL2 is still EMPTY LINKED LIST

Test Case #4
After inserting ten numbers using head insert, LL1 is 41 37 33 29 25 21 17 13 9 5
After constructing LL2 which is a copy of LL1, LL2 is 41 37 33 29 25 21 17 13 9 5
LL1 is still 41 37 33 29 25 21 17 13 9 5
After destruction, LL1 is EMPTY LINKED LIST
LL2 is still 41 37 33 29 25 21 17 13 9 5
After assigning LL1 the value of LL2, LL1 is now 41 37 33 29 25 21 17 13 9 5
LL2 is still 41 37 33 29 25 21 17 13 9 5
After destruction, LL2 is EMPTY LINKED LIST
LL1 is still 41 37 33 29 25 21 17 13 9 5
After destruction, LL1 is EMPTY LINKED LIST
LL2 is still EMPTY LINKED LIST

Test Case #5
After inserting ten numbers using head insert, LL1 is 51 46 41 36 31 26 21 16 11 6
After constructing LL2 which is a copy of LL1, LL2 is 51 46 41 36 31 26 21 16 11 6
LL1 is still 51 46 41 36 31 26 21 16 11 6
After destruction, LL1 is EMPTY LINKED LIST
LL2 is still 51 46 41 36 31 26 21 16 11 6
After assigning LL1 the value of LL2, LL1 is now 51 46 41 36 31 26 21 16 11 6
LL2 is still 51 46 41 36 31 26 21 16 11 6
After destruction, LL2 is EMPTY LINKED LIST
LL1 is still 51 46 41 36 31 26 21 16 11 6
After destruction, LL1 is EMPTY LINKED LIST
LL2 is still EMPTY LINKED LIST
Press any key to continue . . .
```

Once the test program given above works correctly then observe that you can now treat a LinkedList object just like the way you treat simple data types such as int. This means you can copy an object, you can assign an object, you can pass an object as an argument to a function (by value, pointer, or reference), and you can return an object from a function (by value, pointer, or reference). Moreover, you don't need to worry about deleting heap memory because the destructor will do it automatically whenever objects go out of scope.

## Part II: Information Representation using Linked Lists

Now consider the following test program that makes use of the `LinkedList` data structure in order to store the unsigned binary, sign and magnitude binary, and two's complement binary representations of decimal numbers.

```
int selectComputation()
{
	cout << "Select your computation" << endl;
	cout << "   1. Unsigned Binary Representation Computation" << endl;
	cout << "   2. Sign and Magnitude Representation Computation" << endl;
	cout << "   3. Two's Complement Representation Computation" << endl;
	cout << "   4. Exit Program" << endl;
	int selection;
	cout << "Enter your selection (1, 2, 3, or 4): ";
	cin >> selection;
	while (selection != 1 && selection != 2 && selection != 3 && selection != 4)
	{
		cout << "Please enter a correct choice: ";
		cin >> selection;
	}
	return selection;
}

int main()
{
	cout << "This program demonstrates the Linked List Data Structure in C++" << endl;
	cout << "Linked Lists will be used for numeric information representation using" << endl;
	cout << " *** Unsigned Binary Representation" << endl;
	cout << " *** Sign and Magnitude Binary Representation" << endl;
	cout << " *** Two's Complement Binary Representation" << endl << endl;
	cout << "In addition, the program demonstrates" << endl;
	cout << " *** Two's complement binary addition, and" << endl;
	cout << " *** Conversion from two's complement to decimal." << endl << endl;
	do
	{
		int selection = selectComputation();
		if (selection == 1)
		{
			int bit_pattern_size, num;
			cout << endl << "Enter a positive integer for the bit pattern size: ";
			cin >> bit_pattern_size;
			while (bit_pattern_size <= 0)
			{
				cout << "You must enter a positive integer. Enter again please: ";
				cin >> bit_pattern_size;
			}
			cout << "Enter a non-negative integer: ";
			cin >> num;
			while (num < 0)
			{
				cout << "You must enter a non-negative integer. Enter again please: ";
				cin >> num;
			}
			LinkedList LL = computeUnsignedBinary(num, bit_pattern_size);
			cout << "The unsigned binary representation of " << num << " in " <<
bit_pattern_size << " bit is " << LL << endl;
			cout << endl;
		}
		else if (selection == 2)
		{
			int bit_pattern_size, num;
			cout << endl << "Enter a positive integer for the bit pattern size: ";
			cin >> bit_pattern_size;
```

```
                        while (bit_pattern_size <= 0)
                        {
                                cout << "You must enter a positive integer. Enter again please: ";
                                cin >> bit_pattern_size;
                        }
                        cout << "Enter an integer: ";
                        cin >> num;
                        LinkedList LL = computeSignAndMagnitudeBinary(num, bit_pattern_size);
                        cout << "The sign and magnitude binary representation of " << num << " in " <<
bit_pattern_size << " bit is " << LL << endl;
                        cout << endl;
                }
                else if (selection == 3)
                {
                        int bit_pattern_size, num1, num2;
                        cout << endl << "Enter a positive integer for the bit pattern size: ";
                        cin >> bit_pattern_size;
                        while (bit_pattern_size <= 0)
                        {
                                cout << "You must enter a positive integer. Enter again please: ";
                                cin >> bit_pattern_size;
                        }
                        cout << "Enter an integer: ";
                        cin >> num1;
                        LinkedList LL1 = computeTwosComplementBinary(num1, bit_pattern_size);
                        cout << "The two's complement binary representation of " << num1 << " in " <<
bit_pattern_size << " bit is " << LL1 << endl;
                        cout << endl;
                        cout << "Enter a second integer: ";
                        cin >> num2;
                        LinkedList LL2 = computeTwosComplementBinary(num2, bit_pattern_size);
                        cout << "The two's complement binary representation of " << num2 << " in " <<
bit_pattern_size << " bit is " << LL2 << endl;
                        cout << endl;
                        LinkedList LL3 = LL1 + LL2;
                        cout << "The binary sum of " << LL1 << " and " << LL2 << " is " << LL3 << endl;
                        int sum = twosComplementBinaryToDecimal(LL3);
                        cout << "The integer value of the binary sum is " << sum << endl;
                        if (sum == num1 + num2)
                                cout << "This is a correct result." << endl;
                        else
                                cout << "This is not correct result because our bit pattern is too small
to store the result." << endl;
                }
                else
                        break;
                system("Pause");
                cout << endl << endl;
        }while (true);


        system("Pause");
        return 0;
}
```

You are required to implement these non-member and non-friend functions `computeUnsignedBinary`, `computeSignAndMagnitudeBinary`, `computeTwosComplementBinary`, `operator +`, and `twosComplementBinaryToDecimal` used in the main program and get the given main program to work without any syntax, runtime, linking, or semantic errors. In order to define these functions easily, you may also need to define the following supporting/helper non-member and non-friend functions `flipBits`, `reverse`, and `addOne`. Of course you can choose not to define and not to use these supporting/helper functions. However, I strongly advise you to define these supporting/helper functions to make your work much easier.

## Restrictions and Requirements

- You are not allowed to add or remove any include directive or namespace besides the ones already given in this document.
- You are not allowed to declare, define, or use any container variables (objects) such as static arrays, dynamic arrays, or any STL container such as vectors when you define these non-member and non-friend functions.
- You are not allowed to use any STL algorithm when you define these non-member and non-friend functions.

## Descriptions of the non-member and non-friend functions

Each of the non-member and non-friend functions is described below. You are also given the function declarations in order for all of us to have the same function declarations. You must use these given function declarations exactly as they are given without any modification.

**1.**
```
LinkedList computeUnsignedBinary(const int &num, const int &bit_pattern_size)
{
    /*
    This function first asserts that num is a non-negative integer and then computes and stores the
    unsigned binary representation of num in bit_pattern_size bits in a linked list object in such a way
    that the head node stores the most significant bit while the tail node stores the least significant
    bit. The function then returns the linked list object.
    */

    assert(num >= 0);

    //Put your remaining code here
}
```

**2.**
```
LinkedList computeSignAndMagnitudeBinary(const int &num, const int &bit_pattern_size)
{
    /*
    This function computes and stores the sign and magnitude binary representation of num in
    bit_pattern_size bits in a linked list object in such a way that the head node stores the most
    significant bit while the tail node stores the least significant bit. The function then returns then
    linked list object.
    */

    //Put your code here
}
```

**3.**
```
void flipBits(LinkedList &LL)
{
    /*
    This function flips the bits stored in the nodes of the linked list object parameter LL. Assume the
    data in each node of the linked list is a bit (0 or 1).
    */

    //Put your code here
}
```

**4.**
```
void reverse(LinkedList &LL)
```

```
{
    /*
    This function reverses the linked list object parameter LL. Reversing a linked list means the order
    of the nodes in the linked list is reversed.
    */

    //Put your code here
}
```

**5.**
```
LinkedList operator + (const LinkedList &LL1, const LinkedList &LL2)
{
    /*
    This function first asserts that the lengths of the linked list object parameters LL1 and LL2 are
    equal and then adds the bits in LL1 with the bits in LL2 the way we add binary numbers in
    mathematics. The function stores the sum in a linked list object and then returns the linked list
    object. Assume the data in each node of the linked lists is a bit (0 or 1).
    */

    assert(LL1.getLength() == LL2.getLength());

    //Put your remaining code here
}
```

**6.**
```
void addOne(LinkedList &LL)
{
    /*
    This function first asserts that the linked list object parameter LL is not empty and then adds 1 to
    the bits stored in the nodes of the linked list the way we add binary numbers in mathematics. Assume
    the data in each node of the linked list is a bit (0 or 1).
    */

    assert(!LL.isEmpty());

    //Put your remaining code here
}
```

**7.**
```
LinkedList computeTwosComplementBinary(const int &num, const int &bit_pattern_size)
{
    /*
    This function computes and stores the two's complement binary representation of num in
    bit_pattern_size bits in a linked list object in such a way that the head node stores the most
    significant bit while the tail node stores the least significant bit. The function then returns then
    linked list object.
    */

    //Put your code here
}
```

**8.**
```
int twosComplementBinaryToDecimal(const LinkedList &LL)
{
    /*
    This function first asserts that the linked list object parameter LL is not empty and then computes
    and returns the decimal value represented by the bits of the nodes of the linked list assuming two's
    complement binary representation. Assume the data in each node of the linked list is a bit (0 or 1).
    */
```

```
    assert(!LL.isEmpty());

    //Put your remaining code here
}
```

Please note that any overflow bit during your computations must be ignored. For example when you represent a given non-negative integer number in unsigned binary representation say for example in 5 bit but then the integer number requires more than 5 bits to represent it correctly, then only 5 bits should be stored in the linked list and the remaining over flow bits must be ignored. Similarly any overflow bit during arithmetic in two's complement representation must be ignored.

## Starter Code

You are provided a starter code text file (StarterCode.txt file) that contains the code provided in this problem statement file and a sample run output of the test program for your convenience to copy and paste the given code and to check the correctness of your work. This starter code text file is posted together with this assignment.

## Submission Format

You will find a submission link for Assignment 2 on Moodle under Week 8 and you are required to submit through Moodle your source code file (that is .cpp file) that must contain

- The Node class declaration and definition,
- The LinkedList class declaration and definition,
- The definitions of the non-member and non-friend functions described above, and
- The test program.

## Submission Due Date and Time

The due date and time to submit your work is Wednesday 15$^{th}$ March 2023 at 11:55PM. You are required to submit your work through Moodle before the due date and time. No email submission will be accepted.