CSC 230 – Project #3

Due: Apr. 9th, before midnight

What to submit: After you finish the project, you need to submit the following items:

1. A PDF file containing the **pseudocode** of your program. You need to write the pseudocode of the main function and other major functions in your project.
   In the end of this description, we will introduce how to write pseudocode. Basically, the pseudocode is a description of your program in natural language (English)
2. All your source code
3. All the testing data downloaded from Canvas

Project Details:

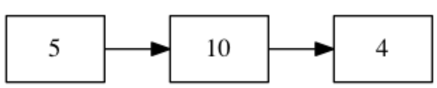
In this assignment, you will play with an important data structure called *linked list*. This project includes two parts. You will first implement a *doubly linked list* class DLL, then re-implement project 2 with the *doubly linked list* DLL (instead of array). Please read the handout and Zybook before coding. The end of this handout has important guidelines to be used in testing.

Getting help

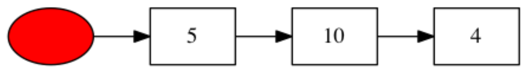
If you don't know where to start, if you don't understand testing, if you are lost, etc., please SEE SOMEONE IMMEDIATELY —an instructor, a tutor. Do not wait. A little in-person help can do the magic.

Linked lists

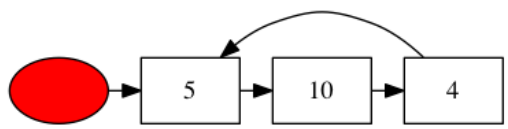
First of all, there are many different ways to implement a linked list. When you are working on a project and trying to implement a linked list, you need to ask yourself what kind of linked list fits your purpose. A simple linked list can be the following one:



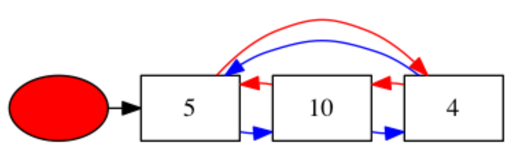
The above singly linked list has three nodes, each node has one integer value inside. In addition to that, each node has a pointer that points to the next node. In the last node of the list, the pointer of that node is null. In this simple singly linked list example, there is one problem: How can we find the beginning of the list? Well, we need a **variable (sometimes called: head pointer)** ALWAYS pointing to the first node of the list. Whenever we need to access to this singly linked list, we read the value of that particular pointer to get the first node address. In the following chart, we use the red box presenting the head pointer. Please note that the red box (head pointer) is not part of the linked list itself. We just use the pointer variable to access the linked list.



Usually, the pointer of the last node has value **NULL (or nullptr in C++11)** that is the case of above chart. However, if the pointer of the last node points to the FIRST node of the list, then we will have a *circular linked list*. The following chart shows a circular linked list example.



Starting from the node pointed by the head variable, one can progress forward and process each node in the linked list. But suppose we have a pointer to the node containing 4 and we want to obtain a pointer to its **predecessor** —the node containing 10. This requires starting from the head and working up the linked list until it is found, which can be time consuming if the list is long. If finding the predecessor is a common operation, we may want to use nodes that contain pointers to both successor and the predecessor. We will implement one class for this purpose. The following diagram below represents a circular linked list of values [5, 10, 4]. The whole list, including the head pointer, is an object of class CLL (for CircularLinkedList). Inside this CLL object, there are two instance variables. Variable ***size*** gives the number of **nodes** in the list. Variable ***headPtr*** contains a pointer to the node that contains the first value in the list, 5. Each node is an object of Node class. Inside each Node object, Variable ***val*** contains the value. Variable ***succ***, for *successor*, contains a pointer to the next value in the list. Varible ***pred***, for *predecessor*, contains a pointer to the preceding value in the list. This is a *circular doubly linked list with header*.



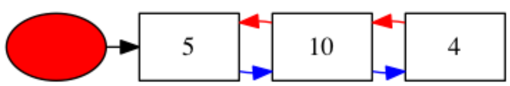
Use of linked lists

When someone choose a data structure, he/she either want to optimize the **speed** or **space** usage of the program, or both. Typically, when people tries to improve the speed of the program, he/she will try to make the most frequently used operations as fast as possible. For example, maintaining a list in an array has disadvantages: (1) The size of the array has to be determined when the array is first created, and (2) Inserting or removing values at the beginning takes time proportional to the size of the list. On the other hand, an array b has the advantage that *any* element, say number i, can be referenced in constant time, using (typically) b[i].

A linked list of any form has these advantages: (1) The list can be **any size**, and (2) Inserting or removing a value at the **beginning** can be done in ***constant*** **time** —it takes a few operations, bounded above by some constant. On the other hand, if one has only a pointer to the header, to reference element i of the list takes time proportional to i—one has to sequence through all the nodes 0..i-1 to find it.

Circular linked lists are useful for representing lists that are naturally circular. One example is the list of corners of a polygon. Another example is the set of processes on a computer; each is to be given some CPU execution time, in round-robin order, over and over again. In this application, the header could be continually set to its successor, so that the header always points to the next node to process.

A *doubly linked list* typically has a header with three fields: head and size as above and tail, which points to the last node. Each node contains fields pred, val (in this project, it should replaced by two strings for SSN and name), and succ, as above, but the **first node** has pred = **nullptr** and the **last node** has succ = **nullptr**. Thus, it is not circular. The following chart shows a *doubly linked list*. In this example, the red oval (headPtr) points to the first node of the list.



And, if p contains a pointer to a node, to test whether it is the last node, use p->succ == **nullptr**. To test whether p points to the first node, use p->pred == **nullptr**. To test a list is empty or not, use headPtr == **nullptr**.

Your Implementation

This project includes three parts.

1. Implement the unfinished methods of DLL class. Modify the methods in DLL.cpp, do NOT change anything in DLL.h or test.cpp.
2. When you implement the methods in DLL.cpp, use test.cpp to test your implementations.
3. If the implementation DLL.cpp is right, write a file called project3.cpp to re-implement project 2 with linked list.
4. If your DLL.cpp is implemented correctly, you will notice that speed performance of project 3 is better than the project 2 implementation. Write an explanation in file **answer.txt**.

To compile test.cpp and DLL.cpp, you can type the following command:

g++ test.cpp DLL.cpp –o test

The executable file will be *test*. If your DLL.cpp implementation is right, when we execute file *test*, the result looks like the following contents:

```
jli$ ./test
After insertion, we should have 10 20 30 40 50 in order
10  0x7f9b9ac032d0
20  0x7f9b9ac03310
30  0x7f9b9ac03250
40  0x7f9b9ac03290
50  0x7f9b9ac03350
Searching 30 in the list, result should be 2
2
After deletion, we should have 20 30 40 in order
20  0x7f9b9ac03310
30  0x7f9b9ac03250
40  0x7f9b9ac03290
Testing copy constructor
Contents of the original list
20  0x7f9b9ac03310
30  0x7f9b9ac03250
40  0x7f9b9ac03290
Contents of the new list, the memory address of the this list
must be different from the original list
20  0x7f9b9ac03350
30  0x7f9b9ac03390
40  0x7f9b9ac033d0
```

The hex values are memory addresses. Your output may have different hex values.

To compile project3.cpp, you can type the following command:

g++ project3.cpp DLL.cpp –o project3

A sample output of project3 is listed as follows:

```
jli$ ./project3 50000-idr
The Number of Valid Insertation :32769
The Number of Valid Deletion :3596
The Number of Valid Retrieval :3638
Item numbers in the list :29173
Time elaspsed :21.8726
```

Hints:

1. This project uses Doubly Linked List, not Circular Doubly Linked List. The **pred** value of the **first** node is **nullptr**, the **succ** value of the **last** node is also **nullptr**.
2. Do NOT rush to your keyboard. Whenever you implement an linked list, draw graphs (with nodes and pointers) on **papers**, consider the following situations on paper first:

    a. List is empty
    b. List is NOT empty
        i. Process the first node
        ii. Process the last node
        iii. Process the middle node
3. Prove your algorithm is correct on paper. Double check!!!
4. Implement your code **incrementally**. Add a few lines to the code. Compile it, then test it.
5. Implement search() function first.
6. Then implement insert() function. Test it.
7. Implement remove() function, test it.
8. Implement copy constructor last.
9. Implement project3.cpp file.

Grading:

1. The correctness of test.cpp execution result is 50 points.
2. The correctness of project3.cpp execution result is 50 points.
3. A submission that cannot be compiled successfully, the maximum grade is 30.
4. A submission using array, vector, or any other data structure other than DLL class will get grade 0.

Suggestions!

Start working on the project **EARLY**! It takes hours, or days to figure out the pointers in the DLL implementation. Do not wait until the last minute to start working. The similarity between this project and project 2 is misleading.

Methodology on testing: Write and test one method at a time! Writing them all and then testing will waste your time. If you have not fully understood what is required, you will make the same mistakes many times. Good programmers write and test **incrementally**, gaining confidence gradually as each method is completed and tested.

Wrap up:

Put all your files, including the sample.txt, all other input files, your head files and cpp files, answer.txt file into **project3.zip**. Submit this zip file to Canvas.

A Brief Introduction to Pseudocode

According to Wikipedia (https://en.wikipedia.org/wiki/Pseudocode), **Pseudocode** is an informal high-level description of the operating principle of a computer program or other **algorithm**. Here, **Algorithms** are the **ideas** behind computer **programs**. An algorithm is an **unambiguous specification** of how to solve a class of problems. But how to express your algorithm? Roughly, there are two choices:

1. Write your program in real programming language. However, the nuts and bolts of the program can obscure the idea of the algorithm.
2. Write your algorithm in pseudocode. Usually, pseudocode is highly abstract, without programming details.

When we talk about computer program, we often think about computer code, such as the following one, which is Euclid GCD solving function.

```
int euclid_gcd_recur(int m, int n)
{
    if(n == 0)
      return m;
    else
      return euclid_gcd_recur(n, m % n);
}
```

The above code is simple enough for programmers to get its idea immediately. More often, the idea behind the code is not straightforward. Given a code, even it is perfect, the programmers may have major difficulty to figure out what is the idea/logic behind the code. The natural obscurity of code is one of the major reasons that it is difficult to understand, maintain, and improve an existing code.

On the contrary, the pseudocode of above C++ program is:

```
if m < n, swap(m,n)
while n does not equal 0
    r = m mod n
    m = n
    n = r
endwhile
output m
```

Please note there is **NO** standard pseudocode language. When you see pseudocode written by other people, you find the style of the pseudocode can vary widely. However, there are a few things you need to keep in mind when you write your pseudocode:

1. Pseudocode is high-level informal description of computer program or algorithm
2. It Follows the structure convention of normal programming language (Fortran, Pascal, C, JAVA etc.)
3. It is a mixture of natural language and programming language
4. It should be easy for human being to read
5. There is no standard pseudocode syntax
6. The pseudocode MUST be **clear**
7. The pseudocode MUST be **concise**

Please keep in mind the purpose of pseudocode is to express the **idea and structure** of your algorithm, NOT details. The pseudocode should show the roadmap of the solution, not every step of the solution.

The following snapshot is a snippet of pseudocode example. In this example, math expression is used to clearly specify the solution.

```
assume onTable ⊆ C, inHand ⊆ C
let highestOnTable = max {r | (s,r) ∈ onTable}
for (s, r) in inHand:
   if r ≤ highestOnTable return false
return true
```

The following pseudocode example is a complete algorithm of some function. In this example, input values and output values are declared in the beginning of the pseudocode. The for loops and while loops are properly indented. In this example, even you do not fully understand what the code is for, you can easily figure out the structure of the algorithm. **In this course, your pseudocode MUST following this example**. You need NOT to write the pseudocode with math expressions, if there is not real necessity. The purpose of the following example is to demonstrate how elegant a pseudocode can be.

```
algorithm ford-fulkerson is
    input: Graph G with flow capacity c,
           source node s,
           sink node t
    output: Flow f such that f is maximal from s to t
    (Note that f_(u,v) is the flow from node u to v, c_(u,v) is the flow capacity from node u to v)
    for each edge (u, v) in G_E do
        f_(u, v) ← 0
        f_(v, u) ← 0
    while there exists a path p from s to t in the residual network G_f do
        let c_f be the flow capacity of the residual network G_f
        c_f(p) ← min{c_f(u, v) | (u, v) in p}
        for each edge (u, v) in p do
            f_(u, v) ← f_(u, v) + c_f(p)
            f_(v, u) ← −f_(u, v)
    return f
```