# CSC 230 – Project #3

## Due: Friday, Mar.31st, before midnight

**Project Details:**

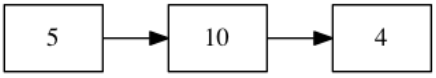
In this assignment, you will play with an important data structure called *linked list*. This project includes two parts. You will first implement a *doubly linked list* class DLL, then re-implement project 2 with the *doubly linked list* DLL (instead of array). Please read the handout and Zybook before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

***Getting help***

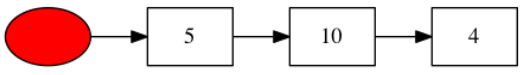
If you don't know where to start, if you don't understand testing, if you are lost, etc., please SEE SOMEONE IMMEDIATELY —an instructor, a tutor. Do not wait. A little in-person help can do the magic.

***Linked lists***

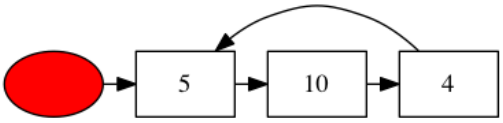
First of all, there are many different ways to implement a linked list. When you are working on a project and trying to implement a linked list, you need to ask yourself what kind of linked list fits your purpose. A simple linked list can be the following one:



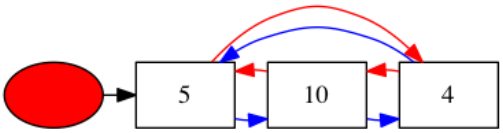

The above singly linked list has three nodes, each node has one integer value inside. In addition to that, each node has a pointer that points to the next node. In the last node of the list, the pointer of that node is null. In this simple singly linked list example, there is one problem: How can we 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**

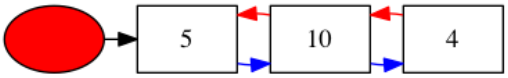
When someone choose a data structure, he/she either want to optimize the **speed** or **space** usage of the program, or both. Typically, when people tries to improve the speed of the program, he/she will try to make the most frequently used operations as fast as possible. For example, maintaining a 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.jar**. Submit this jar file to Canvas.