CSCI 52 – Final Examination

Polynomial Linked list

Notes:

1. **Use of arrays is not allowed in any shape of form.**
2. **Use of recursion[1] is not allowed in any shape or form.**
3. **Use of switch is not allowed.**
4. **You can add extra function if you like, but they cannot replace required functions.**
5. **Value returning functions MUST have ONLY 1 return statement, which must be the last code line in the function.**
6. **Use of exit(0) is NOT ALLOWED.**
7. **Dummy head linked list as I described in my linked list chapter and five linked list videos IS REQUIRED. Points will be taken off if a linked list that is missing a dummy head is done. (I agree that this is being done for my grading convenience. But I need that convenience).**

As you know that polynomial of n degree is represented by an expression as below:

$A_0X^n + A_1X^{n-1} + A_2X^{n-2} + \ldots\ldots\ldots\ldots\ldots\ldots + C = 0$

Such polynomials may be stored in a linked list such that a node contains two integer data members that stores the coefficient and power of X. It would be helpful if the highest order term is stored as first node, successively followed by the lower order terms. A linked list of such polynomial would look like the picture below [Figure 1].

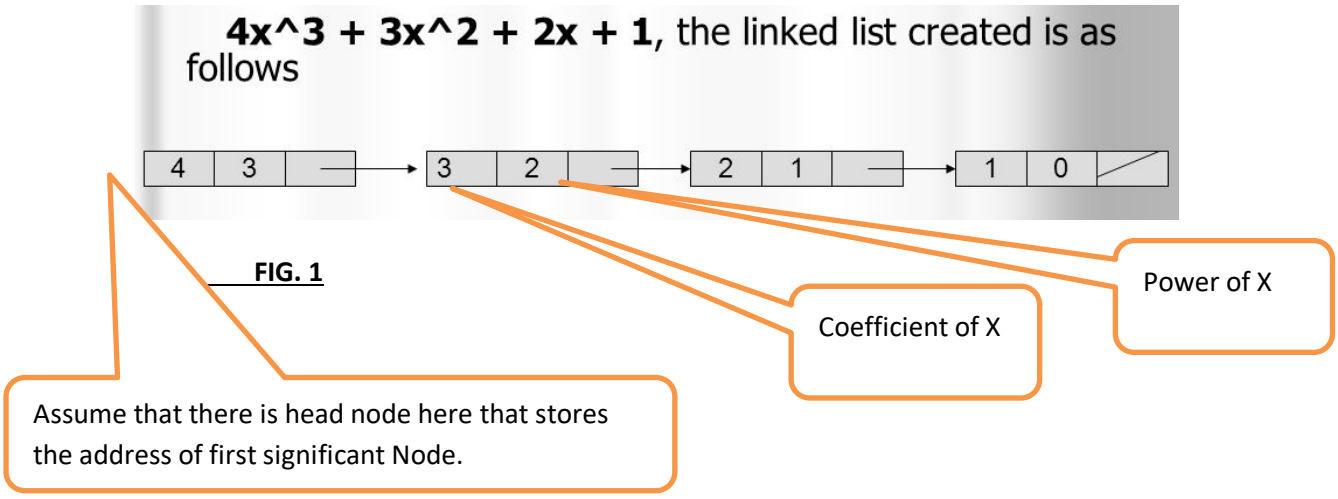

FIG. 1

[1] Recursive solutions are important. However, in this test we are testing your ability to be able to write iterative solutions using pointers to linked lists.

Understand that constant term can be zero. Coefficients can be positive or negative. And some powers may be absent!

The end node has the link field storing null pointer, which is a C++ constant called **nullptr**. In this test your goal is to write a struct called **PolyLinkedList and a main function**, which uses as its node, the struct **PolyNode**, whose partial definition is given to you below. Anything that is done in RED is done for you. Use it as is. Please do not make any changes to it.

```
// Do all necessary includes here
//Declare any global constants here
struct PolyNode{
        int coefficient;
        int power;
        PolyNode * Next;

        PolyNode(int c = 1, int pwr = 0, PolyNode * ptr = nullptr) : coefficient(c), power(pwr), Next(ptr)
        { }
        const string toString() const{
                //Write code for toString function. See my video on toString function for IntLinkedList class
        }
};
```

After writing the above struct PolyNode, write then the PolyLinkedList class (struct), and then the main function to test whole linked list class. I give you freedom to write all code in one file. If you wish you can also split code in multiple files. (But all code in one file is acceptable because it is test and not the commercial software). I am giving you the structure of software here, but you can follow this structure by hand coding first and then transfer to compiler, or directly work with compiler.
If you work with compiler directly, then be careful of correcting compile and logical errors as you go along. From now on I am giving the structure assuming that you are writing all code inside the body of struct PolyLinkedList. Yes, you can do that. C++ stuct work just like a C++ class. Only difference is that unmarked members are all public/

```
struct PolyLinkedList{
PolyNode * HeadPtr;
//Constructor below is already done. Please do not write any code in constructor body
PolyLinkedList():HeadPtr(new PolyNode('A', 'B', nullptr))
{ }
/*1. Write a copy function that would be called from inside of copy constructor and assignment operator to make
deep copy of PolyLinkedList PLL into the current object.
void copy(const PolyLinkedList & PLL)
 {

//Fill out the code for the copy function.

}
```

```
/*
2. Write a destroy function that is called from inside of destructor body and from inside of assignment operator
to de-allocate the dynamically allocated memory for linked list nodes. The destroy function and its correct use,
eliminates the possibility of memory leaks in a linked list application.
*/
void destroy()
{

// Fill out the code for destroy function. See my video on destroy function for IntLinkedList

}
//Enforcement of Rule of three
/*3. Write a copy constructor that makes deep copies when a PolyLinkedList object is passed by value to a function
or is returned by a function as a return value. The copy constructor disallows self-copy. For example, if client tries to
write a code such as below, the program warns that self-copy is not allowed and program exits.
PolyLinkedList P;
PolyLinkedList Q(P);
copy function written as item 1 above should be called from within the body of copy constructor below.
*/
PolyLinkedList(const PolyLinkedList & OtherPoly)
{

// Write the code for copy constructor

}

/*4. Write the body of assignment operator that makes a deep copy when the PolyLinkedList  OtherPoly is copied
into another object using the assignment operator (=). The code inside assignment operator also should handle the
situation when user attempts a code such as below:
PolyLinkedList P;
P = P;
The destroy function created in item 2 and copy function created in item 1 above must be used inside the
assignment operator.
*/
const PolyLinkedList & operator = (const PolyLinkedList & OtherPoly)
{


}
/*
5. Destructor simply calls the destroy function coded as item 2 to de-allocate dynamically allocated nodes of linked
list. This will just be one line of code, because function destroy does all the heavy lifting.*/
virtual ~PolyLinkedList()
{
//Fill code for the destructor

}
```

```
/*
6. The function insertInOrder inserts nodes into PolyLinkedList such that nodes are inserted from front to back in the
order of decreasing power of X, with last node being the constant. Please see the Figure 1 as to how final linked list
would look like even if the nodes were inserted in a random order. For example, user may choose to insert constant
first, and then some power of X etc. In other words, the linked list generated would look like picture in Figure 1,
even if code as below was executed.
PolyLinkedList PLL;
PLL.insertInOrder(1,0);
PLL.insertInOrder(4,3);
PLL.insertInOrder(2,1);
PLL.insertInOrder(3,2);
Additional requirement is that two nodes with the same value of the power field are not allowed. For example, if
user attempts to insert another node to PLL as below, user is informed that node already exits and such node is not
added.
PLL.insertInOrder(9,3);
*/
```

If user attempts this code after code lines above, the user is informed that node already exists and this node is not added.

```
void insertInOrder(int c, int pwr)// c is coefficient and pwr is power
{




}// end of function

/* 7. The function insertFront adds the node with coefficient C and power pwr as the first significant node after the
dummy node, only if sorted order of the linked list (descending order of power of X) is NOT disturbed, or Linked list
has no nodes in it. If adding the node in front of the list is attempted and it disturbs the sorted order of the list, then
node is NOT added and user is informed about it's not being added.
*/
void insertFront(int c, int pwr)// C is coefficient, and pwr is power
{

// Fill out the code for function insertFront

}
/* 8. The function insertTail adds the node with coefficient C and power pwr as the last significant node in the linked
list, only if sorted order of the linked list (descending order of power of X) is NOT disturbed, or Linked list has no
nodes in it. If adding the node as the last node in the list is attempted and it disturbs the sorted order of the list,
then node is NOT added and user is informed about it's not being added.
*/
void insertTail (int c, int pwr) // C is coefficient, and pwr is power
{
// Fill out the code for the function insertTail

}
```

/*9. Write the function getSize, that would iterate through the linked list and get ONLY number of significant nodes in the linked list and ignores dummy node from this count. For example, if linked list has only dummy node, getSize will return 0. Otherwise it will return number of polynomial nodes in the linked list
*/

```
size_t getSize( ) const
{

// Fill out the code for the getSize function.

}
```

/*10. Function getvalue returns the integer number that results from substituting arg for X in the linked list. Examples are shown in table below:

Polynomial	Value of argument arg	Value returned by the function getValue(int arg)
$-2X^4 + 3X^3 - 5X^2 + 2X + 10 = 0$	1	8
$-2X^4 + 3X^3 - 5X^2 + 2X + 10 = 0$	2	-14
$-2X^4 - 5X^2 + 2X + 10 = 0$	2	-38
$2X^5 - 2X^4 - 5X^2 + 2X + 10 = 0$	2	26
$2X^5 - 2X^4 - 5X^2 + 2X + 10 = 0$	3	295

*/

```
int getValue(int arg)
{

// Write the body of getValue function

}
```

/*11. Write the friend function operator == that returns true if both linked lists left and right are identical in content of all nodes, or returns false otherwise.
*/

```
friend bool operator == (const PolyLinkedList & left, const  PolyLinkedList & right)
{

// Fill out the code for operator ==

}
};// End of struct PolyLinkedList
```

/* 11. Write code in main function that you used to test your code as you went through the incrementally developing the various PolyLinkedList functions. Document the output of each test, either in cpp file or in a Microsoft word file. Test should be clear enough that if I wish to try that test, I should be able to do it. After finishing all tests, rename this function as Test*/

/*12.
Final main function: (Use and code helper stand-alone functions as needed)
1. Declare a vector of PolyLinkedList before the loop. Assume that vector is called PolyVec
2. Have a loop menu driven program in the main function with following menu items.

1. **Add Polynomials to PolyVec from keyboard data entry. This menu item should allow user to add as many polynomials to the PolyVec as user desires. This menu item should first prepare a polynomial using insertInOrder function and then use push_back to add to the PolyVec. Make sure that two of the added polynomials are identical, so that we can test == operator.**
2. **Print all polynomials to the console.**
3. **Print all polynomials to an output file. User must be able to specify the full path to the output file and validation that file is successfully opened for writing is required. Understand that full path to file may have white spaces.**
4. **Print to console values of all polynomials when X = 1, X = 2, X = 5. See table we have done for the getValue function.**
5. **Test == operator for those two polynomials in PolyVec, that are identical in content.**
6. **Print to console, the number of terms (using getSize function) for ll polynomials in PolyVec.**
7. **Repeat the test you did for testing function insertFront**
8. **Repeat the test you did for testing function insertTail**
9. **Exit**

Provide tables for all the data you used to test all menu items above.

***/**

Use this space for any scratch work for other purposes.