# CMPT 125: Lab Work Week 6

## Structures

1. Consider the following sequence of **RationalNumber** objects:

   **`r1 = 1/4, r2 = 1/2, r3 = 6/8, r4 = 20/16, r5 = 256/128, r6 = 6656/2048,...`**

   That is the first element is 1/4, the second element is 1/2 and every element afterwards is the sum of the two elements preceding it. Please note that the rational numbers in the sequence are not simplified (reduced) because our binary addition operator doesn't perform any simplification. This is alright because we are not interested in mathematical details in this question. Instead we are interested to demonstrate C/C++ structs, objects, and operator overloading.

   Define a function named **elementAt** that takes an integer argument **n** and returns the **$n^{th}$RationalNumber** object in the sequence.

   For example, the function call **elementAt(1)** must return 1/4, **elementAt(2)** must return 1/2, **elementAt(5)** must return 256/128 etc. Write a test main program to test your function.

   **Hint:-** First define appropriate operator functions to help you solve this problem in the same way you would solve it if the elements were integer numbers.

2. Consider the sequence given above. Define a function named **elementIndex** that takes a **RationalNumber** object and returns its index in the sequence if the **RationalNumber** object is found in the sequence; otherwise returns -1. For example, the function call **elementIndex(1/4)** must return 1, **elementIndex(1/2)** must return 2, **elementIndex(6/8)** must return 3, **elementIndex(6656/2048)** must return 6, **elementIndex(3/2)** must return -1, etc.

   Write a test program to test your function.

   **Hint:-** First define appropriate operator functions to help you solve this problem in the same way you would solve it if the elements were integer numbers.

3. Define a function named **`sortArray`** that takes an array of RationalNumber objects and sorts the array using any of the sequential sorting algorithms (insertion sort, bubble sort or selection sort).

   Write a main program to test your function. In your program ask the user for the size of an array, create a dynamic array of RationalNumber data type with the specified size, fill the array with some random rational numbers of your choice, print the elements of the array, sort the array by calling the sortArray function, and finally print the sorted array to see the correctness of your sorting function

   **Hint:-** First define appropriate operator functions to help you solve this problem in the same way you would solve it if the elements were integer numbers.

4. Consider a complex number object such as in **`3+5i`** which you have learned in mathematics. In general, a complex number is written as **`a+bi`** such that **`a`** is called the real part of the complex number and **`b`** is called the imaginary part of the complex number. We can use a struct to create a complex number data type with two double data type member variables that store the real part and the imaginary part of the complex number.

   Declare a C++ struct named **`ComplexNumber`** with two double data type member variables named and then define all the required functions in order to get the following test program work without any syntax, linking, runtime, or semantic errors.

```
int main()
{
	ComplexNumber c1, c2, *c3;

	c1.real = 0;
	c1.imaginary = 0;
	cout << "c1 real part is " << c1.real << endl;
	cout << "c1 imaginary part is " << c1.imaginary << endl;
	cout << "c1 is " << c1 << endl;

	c2.real = 1.4;
	c2.imaginary = -2.5;
	cout << "c2 is " << c2 << endl;

	c3 = new ComplexNumber;
	*c3 = c1 - c2;
	ComplexNumber c4 = c1 + c2;

	cout << "*c3 is " << *c3 << endl;
	cout << "c4 is " << c4 << endl;

	ComplexNumber c5 = -c2;
	cout << "c5 is " << c5 << endl;

	++c1;
	cout << "Now c1 is " << c1 << endl;
	cout << "c2 is " << c2 << endl;
	cout << "c2++ is " << c2++ << endl;
	cout << "Now c2 is " << c2 << endl;

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

**Sample Run Output**

```
c1 real part is 0
c1 imaginary part is 0
c1 is 0 + 0i
c2 is 1.4 + -2.5i
*c3 is -1.4 + 2.5i
c4 is 1.4 + -2.5i
c5 is -1.4 + 2.5i
Now c1 is 1 + 0i
c2 is 1.4 + -2.5i
c2++ is 1.4 + -2.5i
Now c2 is 2.4 + -2.5i
Press any key to continue . . .
```

**5.** In the imperial system, a **Weight** is represented by two integer values representing the pounds and ounces where one pound is equal to 16 ounces. Declare a C++ struct named **Weight** that represents weight in the imperial system. Then define any appropriate functions that you think are required for a good program design and finally write a test program to test your design. Follow a similar approach to test your design as shown for the ComplexNumber objects given above.

Please note that in the imperial system, any Weight object must keep the value of the pound greater than or equal to zero and the value of the ounces between 0 and 15 ONLY because if you have 16 or more ounces then you should take out the full pounds from the ounces and add them to the pouns. For example, you should never have a Weight object with 5 pounds and 67 ounces because it should be stored instead as 5+4 = 9 pounds and 3 ounces.

# Linked Lists

6. Define a function named **countNodes** that takes the head pointer of a linked list and an integer arguments and returns the number of nodes in the linked list whose data is equal to the integer argument.

   Write an appropriate test program to test your function definition.

7. Define a recursive function named **countNodesRecursive** that takes the head pointer of a linked list and an integer arguments and returns the number of nodes in the linked list whose data is equal to the integer argument. Your function definition must be recursive.

   Write an appropriate test program to test your function definition.

8. Define a function named **all_equal** that takes the head pointer of a linked list and returns true if the data values of all the nodes of the linked list are equal; otherwise returns false.

   Write an appropriate test program to test your function definition.

9. Define a recursive function named **all_equal_Recursive** that takes the head pointer of a linked list and returns true if the data values of all the nodes of the linked list are equal; otherwise returns false.

10. Define a function named **all_different** that takes the head pointer of a linked list and returns true if the data values of all the nodes of the linked list are different; otherwise returns false. Hint:- Use the **countNodes** function defined earlier.

    Write an appropriate test program to test your function definition.

11. Define a recursive function named **all_different_Recursive** that takes the head pointer of a linked list and returns true if the data values of all the nodes of the linked list are different; otherwise returns false.

12. Define a function named **bubbleSortLinkedList** that implements the bubble sort algorithm for linked lists.

13. Define a function named **selectionSortLinkedList** that implements the selection sort algorithm for linked lists.

14. Define a recursive function named **selectionSortLinkedListRecursive** that implements the selection sort algorithm for linked lists.

    Please note that the pseudo-code description of the recursive selection sort algorithm is as follows

```
➔ If the container is empty
      o return
➔ Else
      o Find the position of the minimum item in the container
      o Swap the values of the minimum item and the first item
      o Sort the items in the container excluding the first item
```

15. Define a function named **distinct_nodes** that takes the head pointer of a linked list and returns a new linked list whose nodes store the distinct data values of the nodes of the linked list argument.

    For example, if the data values in the nodes of the linked list are 3, 9, 2, 3, 1, 2, 4, 3 then this function must return a new linked list whose nodes store the data values 3, 9, 2, 1, 4.

    Write an appropriate test program to test your function definition.

16. Define a function named **remove_all** that takes the head pointer of a linked list and a search value arguments and that removes all the nodes in the linked list whose data value is equal to the search value argument. Hint: Use the **remove_node** function discussed in the lecture.

    Write an appropriate test program to test your function definition.

17. Define a function named **insert_grouped** that takes the head pointer of a linked list and a data value as arguments such that the nodes of the linked list argument grouped in to even numbers and odd numbers. Your function must insert a new node into the linked list in an appropriate position so that the data values of all the nodes of the linked list will be grouped in to even and odd groups after the insertion operation.

    If the linked list argument is empty, then your function must perform a head insert operation.

    For example, if the data values in the linked list argument are -5, 13, 9, 34, 20 and the data value to be inserted is 12, then it must be inserted anywhere after the node whose data value is 9.

    Write an appropriate test program to test your function definition.

18. Define a function named **insert_increasing** that takes the head pointer of a linked list and a data value as arguments such that the nodes of the linked list argument are already ordered in increasing order of their data values. Your function must insert a new node into the linked list in an appropriate position so that the data values of all the nodes of the linked list will be ordered in increasing order after the insertion operation.

    If the linked list argument is empty, then your function must perform a head insert operation.

    For example, if the data values in the linked list argument are -5, 2, 9, 14, 20 and the data value to be inserted is 12, then it must be inserted after the node whose data value is 9.

    Write an appropriate test program to test your function definition.

19. Define a function named **insert_grouped_increasing** that takes the head pointer of a linked list and a data value as arguments such that the nodes of the linked list argument grouped in to even numbers and odd numbers and that within each group the data values are ordered in increasing order. Your function must insert a new node into the linked list in an appropriate position so that the data values of all the nodes of the linked list will be grouped and ordered in increasing order after the insertion operation.

    If the linked list argument is empty, then your function must perform a head insert operation.

    For example, if the data values in the linked list argument are -5, 9, 13, 20, 34 and the data value to be inserted is 11, then it must be inserted after the node whose data value is 9.

    Write an appropriate test program to test your function definition.

20. Define a function named **remove_prime** that takes the head pointer of a linked list argument and that remove the first node in the linked list whose data value is a number greater than 1 that is a prime number.

    Write an appropriate test program to test your function definition.

21. Define a function named **remove_all_primes** that takes the head pointer of a linked list that removes all the nodes in the linked list whose data values are prime numbers.

    Write an appropriate test program to test your function definition.

22. Define a function named **insert_before** that takes the head pointer of a linked list, a search value, and a data value arguments and that inserts a node whose data is the data value argument before the first node found in the linked whose data is equal to the search value argument.

If no node is found in the linked list satisfying the condition, then this function doesn't insert any node into the linked list.

Write an appropriate test program to test your function definition.

23. Define a function named **`is_increasing`** that takes the head pointer of a linked list argument and that returns true if all the nodes of the linked list are ordered in increasing order of their data values; returns false otherwise.

    Write an appropriate test program to test your function definition.

24. Define a function named **`get_reversed_linked_list`** that takes the head pointer of a linked list as argument and returns a new linked list whose nodes store the data values of the linked list argument in reverse order.

    Write an appropriate test program to test your function definition.

25. Define a function named **`reverse_linked_list`** that takes the head pointer of a linked list argument and that reverses the linked list.

    Write an appropriate test program to test your function definition.