For the questions below, you are given a file “fragments.cpp”. Look in Resources – CSC2040 Exam 2018 or in the Assignment Tool under CSC2040 Exam 2018. The file includes the various fragments of supplied code mentioned in the questions below, to save you typing them out. The folder also contains some other files to be used for certain questions. The use of these files will be made clear later.

Open Visual Studio or your own IDE and create a new empty project for the exam. All questions below can be completed within this project, by differentiating them using different file names. But note that one project can only have one main function. So, for testing your code for one question, you may have to rename the other unused main functions to main1, main2…, to avoid a compilation error.

It is your responsibility to test each of your programs. Try to make sure that 1) each program can be compiled, and 2) each program produces the correct result.

1. Programming Language C++

In Visual Studio or your own IDE, create a source file `Q1_main.cpp`. This file will be used to hold part of your solutions for Question 1. It will also be used as the place to create your main function to test all your solutions for Question 1.

(a) In `Q1_main.cpp` write a function `myFunc`. This function will take an integer parameter `n`. When `n` is greater than zero (e.g., 3), it will calculate the sum of the whole numbers from 1 to `n` (e.g., 1 + 2 + 3), and return this sum. When `n` is smaller than zero (e.g., –3), it will calculate the sum of the fractions from 1/1, 1/2 to 1/(–`n`) (e.g., 1/1 + 1/2 + 1/3), and return this sum. When `n` equals zero, it will print out a message stating the parameter is invalid, and return zero.

[3 marks]

(b) Create a C++ class named `myClass`. Assume that the class has the following structure of member data and functions (see “fragments.cpp”):

```
class myClass
{
private:
      int* data;         // Private data, a data array
      int dataLength;    // Private data, length of the data array

public:
      // Constructor, which allocates memory for the data array to
      // the given length len, and sets all elements of the array to
      // the given value val
      myClass(int len, int val);
      // Destructor
      ~myClass();

      // Find and return the maximum number in the data array
      int max_number();

      // Overload operator >= for comparing two objects.
      // Object y >= Object x if the average of the numbers in y's
      // data array >= the average of the numbers in x's data array.
      bool operator>=(const myClass& x);
};
```

[continued overleaf]

Create the class in two separate files:

(i) First, create a header file `myClass.h` for the class, which includes the declaration of the class, as above.

(ii) Second, create a source file `myClass.cpp` for the class, which includes a suitable implementation of all the member functions of the class, as defined above. These functions include: (1) the constructor, (2) the destructor, (3) the max-number function, and (4) the operator >= function.

[5 marks]

(c) In the following declaration,

```
myClass* o = new myClass[100];
```

one tries to create a dynamic array of 100 `myClass` objects. However, the compiler will indicate an error to this declaration. Fix this problem by appropriately modifying the `myClass` definition, above`,` so that it will support the use of arrays of objects.

[2 marks]

2. *Data Structures, Trees and their Analysis*

In the same project, create a source file `Q2_main.cpp`. This file will be used to hold all your solutions for Question 2. This will also be the place to create your main function to test your solutions for Question 2. Rename the main function in `Q1_main.cpp` to main1 to avoid a compilation error.

The supplied files “ListNode.h”, “LinkedList.h” and “TreeNode.h” provide facilities to construct linked lists and trees. Save them in your project folder, add them to your project, and #include them in `Q2_main.cpp`.

(a) Copy the following partially completed function (see “fragments.cpp”) into `Q2_main.cpp`. The function creates a linked list of random integers:

```
void LinkedListOfInt(int N)
{
        // Declare a linked list of integers
        LinkedList<(1)> iList;

        // Add N random integers into the list
        for (int n = 0; n < N; n++) {
                int rand_num = rand() % 1000;
                /* (2) */
        }

        // Print the numbers in the list, from the last to the first,
        // without making changes to the list.
        // Option 1:
        for (int n = N - 1; n >= 0; n--)
                cout << *iList.getAt(n) << endl;

        // Option 2:
        /* (3) */

        /* (4) */
}
```

Complete the development. Specifically,

- At the position marked by (1), add a data type.
- At the position marked by (2), add a line of code. This line of code will add the random number `rand_num` into the `iList`, to construct the list.
- At the position marked by (3), provide a **more efficient** implementation (Option 2) than that in Option 1, for printing the list from the last item to the first item without making changes to the list.
- At the position marked by (4), add two print statements (using `cout`), to output your estimation of the complexities of code Option 1 and Option 2:

  “*The complexity of Option 1 is O(xxx).*”
  “*The complexity of Option 2 is O(yyy).*”

  where xxx and yyy are what you think their complexities are in terms of `N`.

[5 marks]

(b) Given three random numbers, 31, 5, 17, we can build a **sorted** binary tree (i.e., a binary search tree - BST) by using the following function (see "fragments.cpp"), which constructs a BST as shown in Fig.1, and returns the pointer to the tree. Calling the tree traverse function will show that the tree is sorted.

```
TreeNode<int>* small_BST()
{
	TreeNode<int>* _5 = new TreeNode<int>(5);
	TreeNode<int>* _31 = new TreeNode<int>(31);
	TreeNode<int>* bst = new TreeNode<int>(17, _5, _31);

	bst->traverse();

	return bst;
}
```

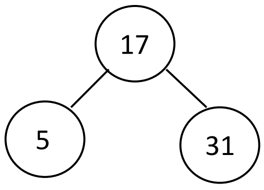

Fig.1. A BST generated by calling `small_BST()`.

Following the above example, in `Q2_main.cpp` write a function `larger_BST`, which will construct, and return the pointer to, a BST for the random number sequence: 78, 24, 69, 0, 34, 67, 41.

[2 marks]

(c) **(i)** Copy the following recursive function (see "fragments.cpp") into `Q2_main.cpp`. The function prints out a given tree in pre-order (i.e., where the item in the tree node is printed before the subtrees). However, the function has a run-time error. Find and correct the error.

```
void printTreePreOrder(TreeNode<int>* tree)
{
	cout << tree->item << endl;
	printTreePreOrder(tree->left);
	printTreePreOrder(tree->right);
}
```

(ii) In `Q2_main.cpp`, write a recursive function which takes a tree of integers (not necessarily sorted) and find the node with the smallest item. It returns the item value. Your function should have the following prototype:

```
	int min_item(TreeNode<int>* tree)
```

[3 marks]

3. *Algorithms and their Analysis*

In the same project, create a source file `Q3_main.cpp`. This file will be used to hold all your solutions for Question 3. This will also be the place to create your main function to test your solutions for Question 3. Rename the main function in `Q2_main.cpp` to main2 to avoid a compilation error.

(a) Copy the following program sketch (see "fragments.cpp") into `Q3_main.cpp`.

```
int linearSearch(int* data, int N, int val)
{
      /* Code for the function */
}

void testLinearSearch(int N)
{
      int* data = new int[N];
      for (int i = 0; i < N; i++) data[i] = rand() % 10;

      // Linear search for the first occurrence of val
      int val = 3;
      cout << "The first occurrence of " << val << " is at "
            << linearSearch(data, N, val) << endl;

      // Complexity of the search
      cout << "The complexity of the search is " << "XXX"
            << endl;

      delete[] data;
}
```

The function `testLinearSearch(int N)` creates a `data` array of `N` random numbers, and then calls a function `linearSearch`, which has a prototype defined above. The function takes the `data` array and searches for the **first** occurrence of the given value `val` in the array. It returns the index at which `val` is found in the array, or -1 if it is not found.

(i) Complete the `linearSearch` function.

(ii) Complete the statement after its call:

"The complexity of the search is XXX"

where *XXX* is what you think the complexity is in terms of the order of `N`.

[3 marks]

(b) The following outlines the Quick Sort algorithm (see “fragments.cpp”), for sorting an array `data` from the `first` element to the `last` element into **ascending** order:

```
// Partition
int i = first, j = last;
int pivot = data[(first + last) / 2]; // Not necessarily the median
while (i <= j) {
      while (data[i] < pivot) i++;
      while (data[j] > pivot) j--;
      if (i <= j) {      // Swap
            int tmp = data[i];
            data[i] = data[j];
            data[j] = tmp;
            i++; j--;
      }
}

// Recursion: i is M
if (first < i - 1) quickSort(data, first, i - 1);
if (i < last) quickSort(data, i, last);
```

(i) Modifying the above code, in `Q3_main.cpp`, write a function

`_quickSort(int* data, int first, int last)`

which sorts the supplied array from the `first` element to the `last` element into **descending** order.

(ii) Add a comment at the end of your `_quickSort` function:

// The complexity of this algorithm is XXX.

where *XXX* is what you think the complexity of your sorting algorithm is in terms of the order(s) of the number of the items in the `data` array.

[4 marks]

(c) Copy the following program sketch (see “fragments.cpp”) into `Q3_main.cpp`.

```
int binarySearch(int* data, int first, int last, int val)
{
      // Code for the function
}

void testQuickSort_BinarySearch(int* data, int N, int val)
{
      // Sort data into descending order
      _quickSort(data, 0, N - 1);

      // Search for the value val in the array
      cout << "Value " << val << " is found at "
            << binarySearch(data, 0, N - 1, val) << endl;

      // Overall complexity
      cout << "The overall complexity of _quickSort + binarySearch"
            << " is XXX" << endl;
}
```

[continued overleaf]

The above function `testQuickSort_BinarySearch` takes a `data` array of `N` unsorted integer numbers and sorts it into descending order using your function `_quickSort` above, so as to implement the binary search for the position of the given value `val` in the array, from the first position to the last position. It returns the position found, or -1 if the array does not contain the value.

(i) Compete the `binarySearch` function.

(ii) Complete the statement after its call:

“The overall complexity of _quickSort + binarySearch is XXX”

where *XXX* is what you think the overall complexity is in terms the order(s) of `N`.

[3 marks]

4. *Graphs and Hash Tables*

In the same project, create a source file `Q4_main.cpp`. This file will be used to hold part of your solutions for Question 4. This will also be the place to create your main function to test your solutions for Question 4. Rename the main function in `Q3_main.cpp` to main3 to avoid a compilation error.

The supplied class files “Graph.h” and “Graph.cpp” provide facility to construct Graphs based on the adjacency matrix representation. Save them in your project folder, add them to your project, and #include “Graph.h” in `Q4_main.cpp`.

(a) Create the definition file: `graph.txt`, for the weighted graph shown in Fig.2. The file should use a format as defined in Fig. 3. You need to define the number of vertices in the graph (*num_v*), define whether or not the graph is directed (*d*) or undirected (*u*), and define all the edges from each vertex, with the corresponding weight for each edge.

Save `graph.txt` in a convenient location so it can be easily loaded for the following test.

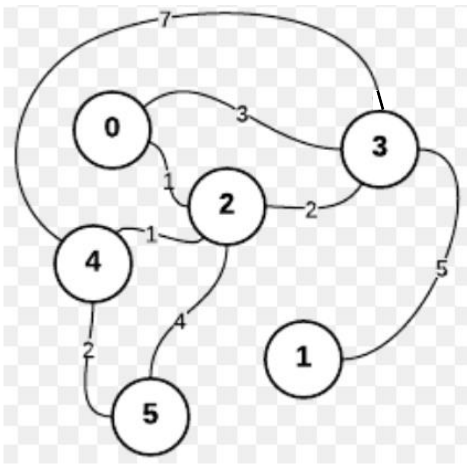

Fig. 2. A weighted graph.

num_v
d or ***u***
Edge 1 from vertex 0   ***weight***
Edge 2 from vertex 0   ***weight***
…
Edge 1 from vertex 1   ***weight***
Edge 2 from vertex 1   ***weight***
…

Fig. 3. The graph definition file *graph.txt* format.

[3 marks]

(b) In `Q4_main.cpp`, write a function

```
void testGraph(char* fname)
{
      // Task 1: Create a Graph object, to represent the graph defined in
      //         the given definition file of name fname
      // Task 2: Write code to search the graph to locate the edge with the
      //         highest weight (assuming all weights >= 0)
      // Task 3: Write code to output the found highest-weight edge in the
      //         format: "Source-vertex Destination-vertex Weight"
}
```

Test the correctness of your function by using your graph definition file `graph.txt` for Fig. 2.

[4 marks]

(c) In `Q4_main.cpp`, write a function

```
size_t hash_index(string firstname, string surname,
   string student_no, size_t tableSize)
```

which will be used to calculate the hash indices for student records, where tableSize is the hash table size. In the calculation, all the three fields are combined to form the key. The hash index is expected to distribute well so you should use a good hash function.

[3 marks]

Submission Notes

1) Save your project.

2) Make sure that you have the following **SEVEN** source files ready for submission:

   Q1: Q1_main.cpp, myClass.h, myClass.cpp
   Q2: Q2_main.cpp
   Q3: Q3_main.cpp
   Q4: Q4_main.cpp, graph.txt

3) Locate your project folder containing all your source files. ZIP this folder (by right clicking the folder and choosing 'Send To' and selecting Compressed(Zipped) folder).

4) Check that your .zip file contains the above 7 source files (double click on the .zip file. You do not need to extract the files again).

5) In case something goes wrong with your submission, first make a backup of your zip file on to a USB drive, and/or email it to yourself.

6) Upload the .zip file to Queens Online using the Assignment tool into CSC2040 Exam 2018.