Project 8 (C++): You are to implement the Graph coloring problem using the two greedy methods taught in class:

Greedy (strategy) used in method 1: for every new color, the method tries to color as many un-colored nodes as possible, as to minimize the number of colors it uses to color the graph.

Greedy (strategy) used in method 2: for each un-color node, the method tries to color it with a used color among all colors that have been used, as to minimize the number of colors it uses to color the graph.

*** In this project, we use hash table (an array of linked-list with dummy node) to represent the input un-directed graph. Nodes in the linked list of hashTable[i] are adjacent to node i. Nodes in linked lists are not sorted, new node always put in the front of the list, as in linked-list stacks.

*** This project is short and simple, serves as a coding exercise before the second coding exam. See if you can implement it in C++ without any help from AI tool. (After you are done with C++, submit it, then try to code it in Java, as an exercise for yourself.)

==

*** You will be given two test data: graph1 and graph2.
What you have to do as follows:

1) Use a blank paper, on the upper half of the paper, draw an undirected graph using graph1, then, color the graph using method1. Next, on the bottom half of the paper, draw the same graph, then, color the graph using method2.

2) Implement and run your program according to the specs below, until it passes compilation.

3) Run and debug your program with graph1 using method 1 until your program produces the same coloring assignment as on the top of your illustration.

4) Run and debug your program with graph1 using method 2 until your program produces the same coloring assignment as on the bottom of your illustration.

5) Run your program using method 1 with graph2.

6) Run your program using method 2 with graph2.

=======================================

*** Include in your hard copy:

- cover page
- your illustration. (-2 pts if not include the drawing.)
- source code

- graph1	//with caption "** below is input graph1 ***
- outFile1 using method1 on graph1	//with caption "** below is outFile1 using method-1 on graph1***"
- logFile using method1 on graph1	//with caption "** below is logFile using method-1 on graph1***"
	// limited to 3 pages if more.
- outFile1 using method2 on graph1	//with caption "** below is outFile1 using method-2 on graph1***
- logFile using method2 on graph1	//with caption "** below is logFile using method-2 on graph1***
	// limited to 3 pages if more.
- graph2	//with caption "** below is input graph2 ***
- outFile1 using method1 on graph2	//with caption "** below is outFile1 using method-1 on graph2***
- logFile using method1 on graph2	//with caption "** below is logFile using method-1 on graph2***
	// limited to 3 pages if more.
- outFile1 using method2 on graph2	//with caption "** below is outFile1 using method-2 on graph2***
- logFile using method2 on graph2	//with caption "** below is logFile using method-2 on graph2***
	// limited to 3 pages if more.

****
Language: C++
****
Project points: 10 pts
Due Date: Soft copy (*.zip) and hard copies (*.pdf):
10/10 (on time): 11/28/2025 Friday before midnight (11:59pm).
-10/10 (non-submission): 11/28/2025 Friday after midnight with 10 minutes grace period.

*** Name your softcopy and hardcopy using the naming convention given in Project Submission Requirements; -2 otherwise.
*** All on-line submission MUST include softcopy (*.zip) and hardcopy (*.pdf) in the same email attachments with **correct email subject**; otherwise, your submission will be rejected, -10/10

*** Inside the email body includes:
- Your answer to the five questions given at the end of the project submission requirements. (-1 if omitted.)
- Screen recoding link. (-2 if omit screen recording).

*** Place your screen recording in your project submission email body below the 5 questions.

****
I. Inputs:
****
1) inFile1 (use argv [1]): a text file contains a list of bi-directional edges of an undirected graph, G=<N, E>. A bi-directional edge in the data file should be treated as having two edges. For example, an undirected edge <3, 5> in the data file, means one directed edge <3, 5> and another directed edge <5, 3>. The input file format is as follows:

The first number in inFile1 is the number of nodes in the graph;
then follows by a list of undirected edges {<n_i, n_j>}. 0 is not used.
For example
```
8        // 8 nodes in the graph
1  2      // edge <1, 2> and edge <2, 1>
4  1     // edge <4, 1> and edge <1, 4>
2  4
5  2
:
```

2) whichMethod (use argv [2]): // 1 or 2. 1 means use method1; 2 means use method2

****
II. Outputs:
****
1) outFile1 (use argv [3]): As program dictates.
2) logFile (use argv [4]): As program dictates.

****
III. Data structure:
****
- A node class:
    - (int) ID // must be > 0 // We don't use 0 for node ID.
    - (node*) next .
    method:
    - constructor (…)
- A coloring class:
    - (int) numNodes // number of nodes in the graph.
    - (int) numUncolor // to indicate the number of un-colored nodes remain in the graph; initialized to numNodes.
    - (node *) hashTable // an array of unsorted linked-lists (with a dummy node), size of numNodes + 1, representing //the input graph; the index, i, of the hashTable represent a node i, and nodes in the linked list are //adjacent to node i; The hashTable [i] are pointing to a dummy node (-999).

- (int *) colorARY // a 1-D integer array to store all nodes' colors; to be dynamically allocate, size of
  //numNodes +1; initialize to 0 -- means none of nodes is colored.

methods:
- constructor (i…) // Establishes all members of the class according to the descriptions of the members.
  // Note: It needs to create hashTable [] and initialize each hashTable [i] points to dummy node.
- loadGraph (…) // read from inFile each un-directed edge <i, j>, then calls hashInsert (…) twice, once
  // hashInsert (i, j) and once hashInsert (j, i). On your own.
- hashInsert (id1, id2) // get a new node for id2; and insert it at the front of hashTable [id1]. On your own.
- printHashTable (outFile2) // output the entire hashTable [i], i = 1 … numNodes. On your own.
  For example, the output for data1 would be:

```
hashTable [1] → 3 → 2 → 4
hashTable [2] → 5 → 3 → 4 → 1
hashTable [3] → 4 → 2 → 1
hashTable [4] → 5 → 3 → 2 → 1
hashTable [5] → 4 → 2
:
```

- method1 (…) // See algorithm below.
- method2 (…) // See algorithm below.
- (bool) checkNeighbors (nodeID, color) // See algorithm below.
  // No two adjacent nodes can have the same color. The method checks to see if any of nodes in
  // hashTable[nodeID] list has been colored with color;
  // if none found, returns true, meaning color is good to use, otherwise returns false.
- printAry (ofile) // use the following format:

```
>> Method 1 (or 2) is used to color the input graph of number of nodes = << // write value
>> Below is the result of the color assignment <<

1   A     //means node 1 is colored with color A; use (char) to turn 65 to A.
2   C     //means node 2 is colored with color C
:
:
//ofile maybe outFile1 or logFile.
```

****
IV. main (…)
****

```
Step 0: check argc count is correct and each file can be opened.
        inFile, outFile1, logFile ← open via argv []
        numNodes ← inFile1
        establish and initialize all members of class according to the above descriptions.

Step 1: loadGraph (inFile)

Step 2: outFile1 ← "*** Below is the Hash table of input graph ***"
        printHashTable (outFile1)

Step 3: whichMethod ← from argv [2]

Step 4: case of whichMethod
        case 1: Method1 (outFile1, logFile)
        case 2: Method2 (outFile1, logFile)
        default: logFile ← print error message: " argv [2] only accept 1 or 2"
                 exit program

Step 5: printAry (outFile1) // print the final result.

Step 6: close all files
```

****
V. Method1 (outFile1, logFile)
****

```
Step 0: logFile ← "*** Entering Method 1 ***"
        newColor ← 64 // colors used in this method are 65, 66, 67, …, (numNodes + 64), as A, B, C, …
Step 1: newColor++ // it uses nextColor sequentially.
Step 2: nodeID ← 1 // want to use newColor to color as many nodes as possible, begins checking at node 1.
Step 3: if colorARY[nodeID] == 0 // meaning, nodeID has not been colored yet
                if checkNeighbors (nodeID, newColor) == true // meaning, no neighbor colored with newColor
                        colorARY[nodeID] ← newColor // color it with newColor
                        numUncolor--
Step 4: nodeID ++ // go on to check next node
Step 5: repeat step 3 to step 4 until nodeID > numNodes
Step 6: logFile ← "*** Printing colorARY"
        printAry (logFile)
Step 7: repeat Step 1 to step 6 until all nodes are colored, i.e., numUncolor <= 0
Step 8: logFile ← "*** Leaving Method 1"
```

****
VI. Method2 (outFile1, logFile)
****

```
Step 0: logFile ← "*** Entering Method 2 ***"
        // colors used in this method are 65, 66, 67, …, (numNodes + 64), as A, B, C, …
        lastUsedColor ← 64 // means no used color.
        nextNodeID ← 0 // use nextNodeID to indicate which node is to be colored.
Step 1: nextNodeID++ // In this method, nodes are colored sequentially.
Step 2: nextUsedColor ← 1 + 64 // check used colors sequentially, begins at 65.
        coloredFlag ← false
Step 3: if lastUsedColor > 64 && checkNeighbors (nextNodeID, nextUsedColor) == true
                colorARY[nextNodeID] ← nextUsedColor
                coloredFlag ← true
        else
                nextUsedColor++ // check the next used color.
Step 4: repeat step 3 while coloredFlag == false & nextUsedColor <= lastUsedColor
Step 5: if coloredFlag == false // meaning none of the used colors can be used.
                lastUsedColor++ // use a new color in sequence to color.
                colorARY [nextNodeID] ← lastUsedColor
                logFile ← "lastUsedColor is" // write the value.
Step 6: logFile ← "*** Printing colorARY"
        printAry (logFile)
Step 7: repeat Step 1 to step 6 until all nodes are colored (i.e., nextNodeID > numNodes.)
Step 8: logFile ← "*** Leaving Method 2"
```

****
VII. (bool) checkNeighbors (colorARY, nodeID, color)
****

```
Step 0: nextNode ←hashTable [nodeID]
Step 1: if nextNode == null
                return true
Step 2: if colorARY [nextNode's ID] == color
                return false
Step 3: nextNode ← nextNode's next
Step 4: repeat Step 1 to Step 3 while nextNode != null
Step 5: return true // Just to be sure
```