

COMP108 Data Structures and Algorithms

Assignment 2

Deadline: Wednesday 27th April 2022, 5:00pm

Important: Please read all instructions carefully before starting the assignment.

Basic information

- Assignment: 2 (of 2)
- Deadline: Wednesday 27th April 2022 (Week 10), 5:00pm
- Weighting: **15%** of the whole module
- Electronic Submission:
Submit to https://sam.csc.liv.ac.uk/COMP/CW_Submissions.pl?qryAssignment=COMP108-2 on departmental server
- What to submit: TWO java files named **COMP108A2Cab.java** and **COMP108A2Graph.java**
- Learning outcomes assessed:
 - Be able to apply the data structures linked lists & graphs and their associated algorithms
 - Be able to apply a given pseudo code algorithm in order to solve a given problem
 - Be able to apply the iterative algorithm design principle
 - Be able to carry out simple asymptotic analyses of algorithms
- Marking criteria:
 - Correctness: 80%
 - Time Complexity Analysis: 20%
- There are two parts of the assignment (Part 1: 65% & Part 2: 15%). You are expected to complete both.

1 The List Accessing Problem

1.1 Background

Suppose we have a filing cabinet containing some files with (unsorted) IDs. We then receive a sequence of requests for certain files, given in the IDs of the files. Upon receiving a request of ID, say *key*, we have to locate the file by flipping through the files in the cabinet one by one. If we find *key*, it is called a *hit* and we remove the file from the cabinet. Suppose *key* is the *i*-th file in the cabinet, then we pay a *cost* of *i*, which is the number of **comparisons** required. If *key* is not in the cabinet, the cost is the number of files in the cabinet and we then go to a storage to locate the file. After using the file, we have to return the file back to the cabinet at the original location or we may take this chance to reorganise the file cabinet, e.g., by inserting the requested file to the front. As the file can only be accessed one after another, it is sensible to use the data structure **linked list** to represent the file cabinet.

1.2 The algorithms

We consider three accessing/reorganising algorithms. To illustrate, we assume the file cabinet initially contains 3 files with IDs **20 30 10** and the sequence of requests is **20 30 5 30 5 20**.

- **Append if miss:** This algorithm does not reorganise the file cabinet and only appends a file at the end if it was not originally in the cabinet. Therefore, there will be **5** hits, the file cabinet will become **20 30 10 5** at the end, and the number of comparisons (cost) for the 6 requests is respectively **1 2 3 2 4 1**. The following table illustrates every step.

request	list beforehand	hit?	# comparisons	list afterward
20	20 30 10	yes	1	no change
30	20 30 10	yes	2	no change
5	20 30 10	no	3	20 30 10 5
30	20 30 10 5	yes	2	no change
5	20 30 10 5	yes	4	no change
20	20 30 10 5	yes	1	no change

- **Move to front:** This algorithm moves the file just requested (including newly inserted one) to the front of the list. In this case, there will be **5** hits. The file cabinet will become **20 5 30 10** at the end. The number of comparisons (cost) for the 6 requests is respectively **1 2 3 2 2 3**. The following table illustrates every step.

request	list beforehand	hit?	# comparisons	list afterward
20	20 30 10	yes	1	no change
30	20 30 10	yes	2	30 20 10
5	30 20 10	no	3	5 30 20 10
30	5 30 20 10	yes	2	30 5 20 10
5	30 5 20 10	yes	2	5 30 20 10
20	5 30 20 10	yes	3	20 5 30 10

- **Frequency count:** This algorithm rearranges the files in non-increasing order of frequency of access. This means that the algorithm keeps a count of how many times a file has been requested. When a file is requested, its counter gets increased by one and it needs to be moved to the correct position. If there are other files with the same frequency, the newly requested file should be put **behind** those with the same frequency. We assume that the files initially in the cabinet has **frequency of 1** to start with. A newly inserted file also has a **frequency** of 1.

In this case, there will be **5** hits. The file cabinet will become **30 20 5 10** at the end. The number of comparisons (cost) for the 6 requests is respectively **1 2 3 2 4 2**. The following table illustrates every step.

request	list beforehand	hit?	# comparisons	list afterward	frequency count afterward
20	20 30 10	yes	1	no change	2 1 1
30	20 30 10	yes	2	no change	2 2 1
5	20 30 10	no	3	20 30 10 5	2 2 1 1
30	20 30 10 5	yes	2	30 20 10 5	3 2 1 1
5	30 20 10 5	yes	4	30 20 5 10	3 2 2 1
20	30 20 5 10	yes	2	no change	3 3 2 1

1.3 The programs

1.3.1 The program COMP108A2Cab.java

After understanding the three algorithms, your main task is to implement a class **COMP108A2Cab** in a java file named **COMP108A2Cab.java**.

The class contains two attributes **head** and **tail** which point to the head and tail, respectively, of the **doubly linked list** to be maintained. A constructor has been defined to set both to null to start with. **Important: You must implement this assignment using the concept of linked list. If you convert the file cabinet list to an array or other data structures before processing, you will lose all the marks.**

The class also contains three methods, one for each of the reorganisation algorithms. The signature of the methods is as follows:

```
public COMP108A2Output appendIfMiss(int rArray[], int rSize)
public COMP108A2Output moveToFront(int rArray[], int rSize)
public COMP108A2Output freqCount(int rArray[], int rSize)
```

The two parameters mean the following:

- **rArray** is an array containing the request sequence.
- **rSize** is the number of elements in rArray.

Note: to make the assignment more accessible, the request sequence will be stored as an array so that you can focus on the linked list for the file cabinet.

The results of the methods should be stored in an object of the class COMP108A2Output. The first line of each of the methods has been written as:

```
COMP108A2Output output = new COMP108A2Output(rSize, 1); and the last line as:
return output;
```

Details of how to use this class can be found in Section 1.3.3.

The class also contains a few pre-written methods to help with the tasks.

```
public void insertHead(COMP108A2Node newNode)
public void insertTail(COMP108A2Node newNode)
public COMP108A2Node deleteHead()
public void emptyCab()
public String headToTail()
public String tailToHead()
public String headToTailFreq()
```

You must NOT change the content of these methods

The methods **insertHead()**, **insertTail()**, **deleteHead()**, **emptyCab()** are provided to create and demolish a doubly linked list. In other words, you don't have to worry about building a list. Instead you can assume that a list is already created and you can reorganise it if needed.

The methods **headToTail()**, **tailToHead()**, **headToTailFreq()** are provided to help with creating output in a format that can be read by the auto-checker. You should **only** use them for this purpose. The following statements are already written at the end of the three methods to illustrate how to use them.

```
output.cabFromHead = headToTail();
output.cabFromTail = tailToHead();
output.cabFromHeadFreq = headToTailFreq(); // this is only for freqCount()
```

1.3.2 The program COMP108A2Node.java

Note: **DO NOT** change the content of this file. **Any changes on this file will NOT be used to grade your submission.**

A class COMP108A2Node is defined to represent a node of the doubly linked list. It contains the following attributes:

```
public int data;
public COMP108A2Node next;
public COMP108A2Node prev;
public int freq;
```

The attribute `freq` is only to be used for `freqCount()` algorithm. A constructor has been implemented that takes an integer parameter and updates the `data` attribute to the argument.

1.3.3 The program COMP108A2Output.java

Note: **DO NOT** change the content of this file. **Any changes on this file will NOT be used to grade your submission.**

A class COMP108A2Output has been defined to contain the following attributes:

```
public int hitCount;
public int missCount;
public int[] compare;
public String cabFromHead;
public String cabFromTail;
public String cabFromHeadFreq;
```

Note that there is another attribute that is for Part 2 and will be discussed later.

A constructor has been implemented that takes two integer parameters and the first parameter is used to define the size of `compare[]` while the second parameter is for Part 2 and takes any positive number.

When we define an object of this class, e.g., the `output` in `appendIfMiss()` in `COMP108A2Cab`, then we can update these attributes by, for example, `output.hitCount++`, `output.missCount++`, `output.compare[i]++`.

IMPORTANT: It is expected your methods will update these attributes of `output` to contain the final answer. Nevertheless, you don't have to worry about printing these output to screen, as printing the output is not part of the requirement.

1.3.4 COMP108A2CabApp.java

To assist you testing of your program, an additional file named `COMP108A2CabApp.java` is provided. Once again, you should not change this program. **Any changes on this file will NOT be used to grade your submission.** This program inputs some data so that they can be passed on to the reorganisation algorithms to process. To use this program, you should compile both `COMP108A2Cab.java` and `COMP108A2CabApp.java`. Then you can run with `COMP108A2CabApp`. See an illustration below.

```
> javac COMP108A2Cab.java
> javac COMP108A2CabApp.java
> java COMP108A2CabApp < cabSampleInput01.txt
```

Before you implement anything, you will see the output like the left of the following figure. When you complete your work, you will see the output like the right.

```
Initial cabinet:
20 30 10
Request sequence:
20 30 5 30 5 20

appendIfMiss
0,0,0,0,0,0,
0 h 0 m
From head to tail: 20,30,10,
From tail to head: 10,30,20,

moveToFront
0,0,0,0,0,0,
0 h 0 m
From head to tail: 20,30,10,
From tail to head: 10,30,20,

freqCount
0,0,0,0,0,0,
0 h 0 m
From head to tail: 20,30,10,
From tail to head: 10,30,20,
Frequency from head to tail: 1,1,1,
```

```
Initial cabinet:
20 30 10
Request sequence:
20 30 5 30 5 20

appendIfMiss
1,2,3,2,4,1,
5 h 1 m
From head to tail: 20,30,10,5,
From tail to head: 5,10,30,20,

moveToFront
1,2,3,2,2,3,
5 h 1 m
From head to tail: 20,5,30,10,
From tail to head: 10,30,5,20,

freqCount
1,2,3,2,4,2,
5 h 1 m
From head to tail: 30,20,5,10,
From tail to head: 10,5,20,30,
Frequency from head to tail: 3,3,2,1,
```

1.4 Your tasks

There are three tasks you need to do. Each task is associated with each of the three algorithms each should return an object of class COMP108A2Output such that the following attributes of output is computed correctly:

- (i) `hitCount` storing the number of hits
 - (ii) `missCount` storing the number of misses
 - (iii) `compare[]` storing the number of comparisons for each request
 - (iv) `cabFromHead` storing the data attribute of the final list (from head to tail) in the form produced by the method `headToTail()`
 - (v) `cabFromTail` storing the data attribute of the final list (from tail to head) in the form produced by the method `tailToHead()`
 - (vi) only for `freqCount()`: `cabFromHeadFreq` storing the freq attribute of the final list (from head to tail) in the form produced by the method `headToTailFreq()`
- **Task 1.1 (20%)** Implement the `appendIfMiss()` method that appends a missed file to the end of the list and does not reorganise the list.
 - **Task 1.2 (20%)** Implement the `moveToFront()` method that moves a requested file or inserts a missed file to the front of the list. When moving a node in the list, you have to make sure that the `next` and `prev` fields of affected nodes, `head`, and `tail` are updated properly.

- **Task 1.3 (25%)** Implement the `freqCount()` method that moves a requested file or inserts a missed file in a position such that the files in the list are in non-increasing order. Importantly when the currently requested file has the same frequency count as other files, the requested file should be placed at the end among them. Again make sure you update `next`, `prev`, `head`, `tail` properly.

1.5 Test cases

Below are some sample test cases and the expected output so that you can check and debug your program. The input consists of the following:

- The size of the initial file cabinet (between 1 and 10 inclusively)
- Initial content of the file cabinet (all positive integers)
- The number of requests in the request sequence (between 1 and 100 inclusively)
- The request sequence (all positive integers)

Your program will be marked by five other test cases that have not be revealed.

Test cases	Input / Output
#1	Input: 3 20 30 10 6 20 30 5 30 5 20
	appendIfMiss Comparisons: 1,2,3,2,4,1, 5 h 1 m From head to tail: 20,30,10,5, From tail to head: 5,10,30,20,
	moveToFront Comparisons: 1,2,3,2,2,3, 5 h 1 m From head to tail: 20,5,30,10, From tail to head: 10,30,5,20,
	freqCount Comparisons: 1,2,3,2,4,2, 5 h 1 m From head to tail: 30,20,5,10, From tail to head: 10,5,20,30, Frequency from head to tail: 3,3,2,1,
#2	Input: 4 20 30 10 40 9 50 10 10 20 50 50 50 40 70
	appendIfMiss Comparisons: 4,3,3,1,5,5,5,4,5,

	7 h 2 m From head to tail: 20,30,10,40,50,70, From tail to head: 70,50,40,10,30,20, moveToFront Comparisons: 4,4,1,3,3,1,1,5,5, 7 h 2 m From head to tail: 70,40,50,20,10,30, From tail to head: 30,10,20,50,40,70, freqCount Comparisons: 4,3,1,2,5,3,2,5,5, 7 h 2 m From head to tail: 50,10,20,40,30,70, From tail to head: 70,30,40,20,10,50, Frequency from head to tail: 4,3,2,2,1,1,
#3	Input: 3 20 10 30 20 10 20 30 10 60 20 30 30 30 30 40 40 40 40 50 50 50 50 20 50 appendIfMiss Comparisons: 2,1,3,2,3,1,3,3,3,3,4,5,5,5,5,6,6,6,1,6, 17 h 3 m From head to tail: 20,10,30,60,40,50, From tail to head: 50,40,60,30,10,20, moveToFront Comparisons: 2,2,3,3,3,4,4,1,1,1,4,1,1,1,5,1,1,1,4,2, 17 h 3 m From head to tail: 50,20,40,30,60,10, From tail to head: 10,60,30,40,20,50, freqCount Comparisons: 2,2,3,1,3,2,3,3,1,1,4,5,4,4,5,6,5,5,5,3, 17 h 3 m From head to tail: 30,50,40,20,10,60, From tail to head: 60,10,20,40,50,30, Frequency from head to tail: 6,5,4,4,3,1,

These test cases can be downloaded as cabSampleInput01.txt, cabSampleInput02.txt, cabSampleInput03.txt on CANVAS and the output as cabSampleOutput01.txt, cabSampleOutput02.txt, cabSampleOutput03.txt.

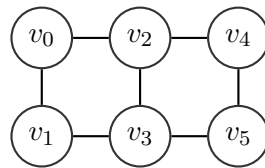
You can run the program easier by typing `java COMP108A2CabApp < cabSampleInput01.txt` in which case you don't have to type the input over and over again. The test files should be stored in the same folder as the java and class files.

2 Part 2: Neighbourhood Distance on Graphs (30%)

2.1 Background

Suppose we have an **undirected** graph to model connections on a social network. There are n users, each represented by a vertex. If two users are friends of each other, then there is an edge between the two corresponding vertices. This forms a simple graph (at most one edge between any two vertices) with no self-loop (a vertex has no edge to itself). This friend-relationship can be represented by an adjacency matrix of size $n \times n$. The entry (i, j) is 1 if vertex i and j have an edge between them, and 0 otherwise.

For any two vertices that are not immediate neighbour (i.e., no edge between them), we want to know if they are “connected” via the same neighbour. In other words, they are distance-2 apart. We can generalise this concept to distance- d for $d \geq 1$. For example, in the following graph, v_0 is a distance-2 neighbour of v_3 and v_4 but not v_5 while v_0 is a distance-3 neighbour of v_5 .



We can represent this neighbourhood relationship using a ***neighbourhood matrix***. An entry between vertex i and j is $d(i, j)$ if the closest distance between them is $d(i, j)$. If there is no path between i and j , then the entry (i, j) is 0. We want to compute the neighbourhood matrix to contain this information for all pairs of vertices. With the above graph, the neighbourhood matrix is shown below.

$$\begin{pmatrix} 0 & 1 & 1 & 2 & 2 & 3 \\ 1 & 0 & 2 & 1 & 3 & 2 \\ 1 & 2 & 0 & 1 & 1 & 2 \\ 2 & 1 & 1 & 0 & 2 & 1 \\ 2 & 3 & 1 & 2 & 0 & 1 \\ 3 & 2 & 2 & 1 & 1 & 0 \end{pmatrix}$$

2.2 The programs

2.2.1 The program COMP108A2Graph.java

The task is to implement the class **COMP108A2Graph** in a java file named **COMP108A2Graph.java**.

The class contains a method `neighbourhood` with the following signature:

`neighbourhood(int[][] adjMatrix, int gSize)`

The two parameters means the following:

- `adjMatrix` is a 2-dimensional array containing the adjacent matrix.
- `gSize` is the size / number of vertices of the graph, i.e., `adjMatrix` is `gSize` by `gSize`.

The result, i.e., the neighbourhood matrix, of the method should be stored in an object of the class COMP108A2Output. The first line of the method has been written as:

```
COMP108A2Output output = new COMP108A2Output(1, gSize); and the last line as:  
return output;
```

Details of how to use this class can be found in Section 2.2.2.

2.2.2 The program COMP108A2Output.java

Note: **DO NOT** change the content of this file. **Any changes on this file will NOT be used to grade your submission.**

A class COMP108A2Output has been defined to contain the following attributes:

```
public int[] [] neighbourMatrix;
```

Note that there are other attributes that are for Part 1 and we have discussed them earlier.

A constructor has been implemented that takes two integer parameters and the second parameter is used to define the size of the graph and hence `neighbourMatrix[] []` while the first parameter is for Part 1 and takes any positive number.

2.2.3 COMP108A2GraphApp.java

To assist you testing of your program, an additional file named COMP108A2GraphApp.java is provided. Once again, you should not change this program. **Any changes on this file will NOT be used to grade your submission.** This program inputs some data so that they can be passed on to the reorganisation algorithms to process. To use this program, you should compile both COMP108A2Graph.java and COMP108A2GraphApp.java. Then you can run with COMP108A2GraphApp. See an illustration below.

```
>javac COMP108A2Graph.java  
  
>javac COMP108A2GraphApp.java  
  
>java COMP108A2GraphApp < graphSampleInput01.txt
```

Before you implement anything, you will see the output like the left of the following figure. When you complete your work, you will see the output like the right.

0 0 0 0 0 0	0 1 1 2 2 3
0 0 0 0 0 0	1 0 2 1 3 2
0 0 0 0 0 0	1 2 0 1 1 2
0 0 0 0 0 0	2 1 1 0 2 1
0 0 0 0 0 0	2 3 1 2 0 1
0 0 0 0 0 0	3 2 2 1 1 0

2.3 Your tasks

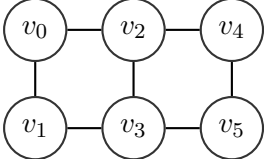
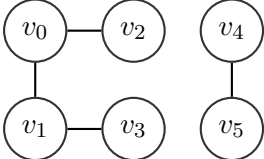
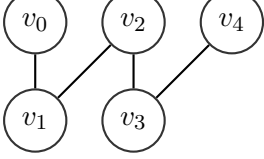
- **Task 2 (15%)** Implement the `neighbourhood()` method that computes the neighbour Matrix for all pairs of vertices. The method should compute the results and stores them at `output.neighbourMatrix`.

2.4 Test cases

Below are some sample test cases and the expected output so that you can check and debug your program. The input consists of the following:

- The size of graph (between 2 and 10 inclusively)
- The adjacency matrix

Your program will be marked by five other test cases that have not be revealed.

Test case	Input	Output
#1 graphSample01.txt 	6 0 1 1 0 0 0 1 0 0 1 0 0 1 0 0 1 1 0 0 1 1 0 0 1 0 0 1 0 0 1 0 0 0 1 1 0	0 1 1 2 2 3 1 0 2 1 3 2 1 2 0 1 1 2 2 1 1 0 2 1 2 3 1 2 0 1 3 2 2 1 1 0
#2 graphSample02.txt 	6 0 1 1 0 0 0 1 0 0 1 0 0 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 1 0	0 1 1 2 0 0 1 0 2 1 0 0 1 2 0 3 0 0 2 1 3 0 0 0 0 0 0 0 0 1 0 0 0 0 1 0
#3 graphSample03.txt 	5 0 1 0 0 0 1 0 1 0 0 0 1 0 1 0 0 0 1 0 1 0 0 0 1 0	0 1 2 3 4 1 0 1 2 3 2 1 0 1 2 3 2 1 0 1 4 3 2 1 0

3 Additional information

3.1 Worst Case Time Complexity analysis in big-O notation (20%)

- **Task 3 (20%)** For each of the algorithms you have implemented, give a worst case time complexity analysis (in big-O notation). Give also a short justification of your answer. This should be added to the comment sections at the beginning of COMP108A2Cab.java and COMP108A2Graph.java. No separate file should be submitted.
- The time complexity has to match your implementation. Marks will NOT be awarded if the corresponding algorithm has not been implemented.
- For COMP108A2Cab.java, you can use any of the following in the formula: f to represent the initial cabinet size, n the number of requests and d the number of distinct requests.
- For COMP108A2Graph.java, you can use n to represent the number of vertices in the graph.

- You can define additional symbols if needed.
- For each algorithm, 3% is awarded to the time complexity and 2% to the justification.

3.2 Penalties

- UoL standard penalty applies: Work submitted after 5:00pm on the deadline day is considered late. 5 marks shall be deducted for every 24 hour period after the deadline. Submissions submitted after 5 days past the deadline will no longer be accepted. Any submission past the deadline will be considered at least one day late. Penalty days include weekends. This penalty will not deduct the marks below the passing mark.
- If your code does not compile successfully, 5 marks will be deducted. If your submitted files are not named COMP108A2Cab.java and COMP108A2Graph.java, 5 marks will be deducted. If your code compile to a class of a different name from COMP108A2Cab and COMP108A2Graph, 5 marks will be deducted. These penalties will not deduct the marks below the passing mark.
- You are required to implement the list and adjacency matrix yourself. **No** in-built methods can be used. Among others, you are definitely not allowed to use the LinkedList class in java. Using in-built methods would **get all marks deducted** (possibly below the passing mark).
- You must implement Part 1 using the concept of linked list. If you convert the file cabinet list to an array or other data structures before processing, you will lose all the marks.

3.3 Plagiarism/Collusion

This assignment is an individual work and any work you submitted must be your own. You should not collude with another student, or copy other's work. Any plagiarism/collusion case will be handled following the University guidelines. Penalties range from mark deduction to suspension/termination of studies. Refer to University Code of Practice on Assessment Appendix L — Academic Integrity Policy for more details.

In Assignment 1, **six collusion cases involving thirteen students** have been reported to the Departmental Academic Integrity Officer. Some of these cases involve knowingly sharing one's work with other students, which is strictly prohibited according to University rules. It is still an offense even if you have done your work and share it with your friends.

The automatic checking system *moss* will be used to detect plagiarism/collusion cases.