COMP108 Data Structures and Algorithms

Assignment 2

Deadline: Thursday 3rd May 2018, 4:00pm

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

Basic information

- Assignment: 2 (of 2)
- Deadline: Thursday 3rd May 2018 (Week 11), 4:00pm
- Weighting: 20%
- Electronic Submission:
  `https://sam.csc.liv.ac.uk/COMP/Submissions.pl?qryAssignment=COMP108-2`
- What to submit: three java files named A2Node.java, A2List.java and A2Graph.java
- Learning outcomes assessed:
  - Be able to apply the data structures arrays and linked lists 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
- Marking criteria:
  - Correctness: 90%
  - Programming style: 10%
- There are two parts of the assignment (Part 1: 55% & Part 2: 35%). You are expected to complete both.

1 Part 1: The List Accessing Problem (55%)

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.

1.1 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 get 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

1.2 The programs A2Node.java and A2List.java

Two programs A2Node.java and A2List.java can be downloaded from
`http://www2.csc.liv.ac.uk/~pwong/teaching/comp108/201718/assess.html`
The program A2Node.java contains a class called `A2Node` with the following attributes

```
public int data;
public A2Node next;
public int freq;
```

The program A2List.java has declared two global variables `head` and `tail` which point to the head (first node) and the tail (last node), respectively, of the list representing the file cabinet. The program contains a number of methods already written which you must **NOT** change, including

- `public static void main(String[] args)`
- `static void insertNodeHead(A2Node newNode)` which inserts the node called newNode to the head of the list
- `static A2Node deleteHead()` which deletes the node at the head
- `static void printList()` which prints the list from head to tail
- `static void emptyList()` which empties the whole list

The main method takes care of the input, reading (i) number of files in the initial cabinet, (ii) the file IDs in the initial cabinet, (iii) number of file requests, and (iv) IDs of the file requests. (Notice that the input part has suppressed printing text asking for input.) It then creates a list of the initial cabinet and calls each algorithm in turn. The order of calling the algorithm has been pre-determined and you should not change the order, otherwise, you may lose all your marks. The header of each algorithm has been provided. They can all access global variables `head` and `tail`.

1.3 Your tasks

There are three tasks you need to do. Each task is associated with each of the algorithms, and each should print and **only** print the following. See Test Cases below for examples.

(i) The number of comparisons for each request separated by a space ␣, e.g., `1␣2␣3␣`
(Note: one single space at the end is allowed.)

(ii) `x␣h` where `x` is the number of hits

(iii) The content of the final list in the form `List:␣a␣b␣c␣d␣`
where `a b c d` are the IDs of file in the final list. (Again one single space at the end is allowed.) If your list is maintained properly the method `printList` will do the job.

- **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 (15%)** 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` field of affected nodes, `head`, and `tail` are updated properly.
- **Task 1.3 (20%)** 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 requested file has the same frequency count as other files, the request file should be placed at the end among them. Again make sure you update `next`, `head`, `tail` properly.

1.4 Test cases

Below are some sample test cases and the expected output. These test cases can be downloaded as listTest01.txt, listTest02.txt, listTest03.txt on the assessment page: `http://www2.csc.liv.ac.uk/~pwong/teaching/comp108/201718/assess.html`
You can run the program easier by typing `java A2List < listTest01.txt` in which case you don't have to type the input over and over again.

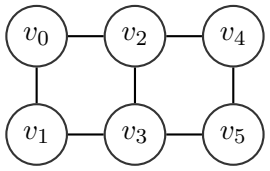
Make sure you remove or comment print statements that you use for debugging before submission. The order of output should be `appendIfMiss`, `moveToFront`, `freqCount`. Your program will be tested in this order, therefore, do NOT change the order (you should not change the main method anyway)! Your program will be marked by five other test cases that have not be revealed.

Test case	Input	Output ('␣' represents a space)
#1	3 20 30 10 6 20 30 5 30 5 20	appendIfMiss... 1␣2␣3␣2␣4␣1␣ 5␣h List:␣20␣30␣10␣5␣ moveToFront... 1␣2␣3␣2␣2␣3␣ 5␣h List:␣20␣5␣30␣10␣ freqCount... 1␣2␣3␣2␣4␣2␣ 5␣h List:␣30␣20␣5␣10␣
#2	4 20 30 10 40 9 50 10 10 20 50 50 50 40 70	appendIfMiss... 4␣3␣3␣1␣5␣5␣5␣4␣5␣ 7␣h List:␣20␣30␣10␣40␣50␣70␣ moveToFront... 4␣4␣1␣3␣3␣1␣1␣5␣5␣ 7␣h List:␣70␣40␣50␣20␣10␣30␣ freqCount... 4␣3␣1␣2␣5␣3␣2␣5␣5␣ 7␣h List:␣50␣10␣20␣40␣30␣70␣
#3	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... 2␣1␣3␣2␣3␣1␣3␣3␣3␣3␣4␣5␣5␣5␣5␣6␣6␣6␣1␣6␣ 17␣h List:␣20␣10␣30␣60␣40␣50␣ moveToFront... 2␣2␣3␣3␣3␣4␣4␣1␣1␣1␣4␣1␣1␣1␣5␣1␣1␣1␣4␣2␣ 17␣h List:␣50␣20␣40␣30␣60␣10␣ freqCount... 2␣2␣3␣1␣3␣2␣3␣3␣1␣1␣4␣5␣4␣4␣5␣6␣5␣5␣5␣3␣ 17␣h List:␣30␣50␣40␣20␣10␣60␣

2 Part 2: Distance Neighbourhood on Graphs (35%)

Suppose we have an **undirected** graph to model facebook connection (or some other similar 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.

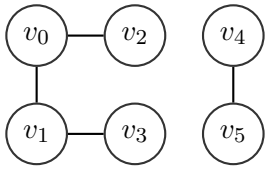


We can represent this distance-d neighbourhood relationship using a ***d-neighbourhood matrix***. An entry between vertex i and j is $d(i,j)$ if the closest distance between them is $d(i,j)$ and $d(i,j)$ is at most d. If the distance $d(i,j) > d$ or there is no path between i and j, then the entry (i,j) is 0. With the above graph, the distance-1, distance-2 and distance-3 neighbourhood matrix are shown below in the left, middle, right, respectively.

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

In addition, we are interested to know the ***degree of separation*** which is the minimum distance deg such that the deg-neighbourhood of every vertex covers every other vertices. In other words, we want to find the minimum deg such that the deg-neighbourhood matrix contains 0 entries only along the diagonal. In the above example, the degree of separation is 3.

In some cases, the given graph may not be connected, meaning that there is at least a pair of vertices that is not connected by any path. For example, the following graph is **not connected.** The distance-1, distance-2, distance-3 neighbourhood matrix are also shown below (left, middle, right, respectively).



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

2.1 The program A2Graph.java

The program A2Graph.java can be downloaded from
`http://www2.csc.liv.ac.uk/~pwong/teaching/comp108/201718/assess.html`

A global 2-dimensional array variable `adjMatrix[][]` has been declared, which stores the adjacency matrix. A number of methods already written which you must **NOT** change, including

- `public static void main(String[] args)`
- `static void input()` which reads input.
- `static void printSquareArray(int array[][], int size)` which prints the content of a 2-D size-by-size array

The main method first calls the method `input()` to get (i) the size of the graph, (ii) the adjacency matrix, and then asks for (iii) a distance. (Notice that the input part has suppressed printing text asking for input.) Then it calls the method `neighbourhood` to calculate the neighbourhood matrix, followed by the method `degreeSeparation` to find the degree of separation.

2.2 Your tasks

- **Task 2.1 (20%)** Implement the method `neighbourhood` with the following header:
  `static void neighbourhood(int distance, int result[][], int size)`
  The method should calculate the `distance`-neighbourhood matrix and store the matrix in the `result[][]` array which is of `size`-by-`size`. The main method will call the `printSquareArray` method so this method does not need to print anything.

- **Task 2.2 (15%)** Implement the method `degreeSeparation` to determine the degree of separation of the graph. The output should be: "`Degree of separation is XX`", where XX is the answer. If the graph is not connected, i.e., there is at least one pair of vertices which cannot be connected by a path, then the following message should be printed: "`The graph is not connected`".

2.3 Test cases

Below are some sample test cases and the expected output. These test cases can be downloaded as graphTest01.txt, graphTest02.txt, graphTest03.txt on the assessment page: `http://www2.csc.liv.ac.uk/~pwong/teaching/comp108/201718/assess.html`
You can run the program easier by typing `java A2Graph < graphTest01.txt` in which case you don't have to type the input over and over again.

Make sure you remove or comment all other print statements that you use for debugging before submission. The order of output should be `neighbourhood`, `degSeparation`. Your program will be tested in this order, therefore, do NOT change the order (you should not change the main method anyway)! Your program will be marked by five other test cases that have not be revealed.

Test case	Input	Output
#1	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 2	0 1 1 2 2 0 1 0 2 1 0 2 1 2 0 1 1 2 2 1 1 0 2 1 2 0 1 2 0 1 0 2 2 1 1 0 Degree of separation is 3

Test case	Input	Output
#2	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 3	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 The graph is not connected
#3	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 4	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 Degree of separation is 4

3 Additional information

Programming Style (10%)

- You should keep a good programming style. Marks will be awarded based on

  (i) 2% consistent and appropriate use of brackets

  (ii) 2% consistent and appropriate use of indentation

  (iii) 2% meaningful variable names

  (iv) 2% comments to explain the working of your programs

  (v) 2% your personal information at the beginning of the java files

Penalties

- UoL standard penalty applies: Work submitted after 4:00pm on the deadline day is considered late. 5 marks shall be deducted for each 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 code compile to classes of different names from A2List, A2Node, A2Graph, 5 marks will be deducted. If your output does not follow the same format as expected, 5 marks will be deducted. If your output does not follow the order of the algorithms as expected, 5 marks will be deducted. These penalties will not reduce the marks below the passing mark.
- You are required to implement the list and adjacency matrix yourself. It is **NOT** allowed to use built-in classes or functions. Doing so would get all marks deducted (possibly below the passing mark).

Plagiarism

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 case will be handled following the University guidelines.