COMP108 Data Structures and Algorithms

Programming Assignment

Deadline: Friday 21st April 2023, 5:00pm

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

Basic information

- Weighting: 15% of the whole module
- Submission to codegrade via Canvas
- What to submit: two java files named **COMP108Paging.java** and **COMP108Cab.java**
- Learning outcomes assessed:
  - Be able to apply the data structure arrays & 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
  - 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: 32.5% & Part 2: 47.5%)

1 Part 1: The paging/caching problem

1.1 Background

The paging/caching problem arises from memory management, between external storage and main memory and between main memory and the cache. We consider a two-level virtual memory system. Each level can store a number of fixed-size memory units called **pages**. The slow memory contains N pages. The fast memory has a fixed size $k < N$ which can store a subset of the N pages. We call the fast memory the **cache**. Given a *request* for a page is issued. If the requested page is already in the cache, we call it a ***hit***. If the requested page is not in the cache, we call it a ***miss*** and we have to ***evict*** (remove) a page from the cache to make room for the requested page. Pages that are not evicted must stay in the same location in the cache.

1.2 The algorithms

Different eviction algorithms use different criteria to choose the page to be evicted. We consider the following eviction algorithms. To illustrate, we assume the cache contains 3 pages with initial ID content **20, 30, 10** and the sequence of requests is **20, 5, 30, 10, 5, 20**.

(i) **No eviction.** This algorithm does not evict any pages, i.e., the cache stays as the initial content. Then the hit (h) and miss (m) sequence will be **hmhhmh** since 20 is a hit, 5 miss, 30 hit, 10 hit, 5 miss, 20 hit. There are **4 h and 2 m**.

(ii) **Evict LRU.** This algorithm evicts the pages in the LRU (least recently used) principle. This means that the algorithm evicts the page whose most recent request was earliest. We assume that the initial cache content is added one by one with 20 first, then 30, then 10. The hit and miss sequence will then be **hmmmhm** with **2 h 4 m**.

request	cache beforehand	hit/miss	cache afterward	remarks
20	20 30 10	h	no change	
5	20 30 10	m	20 5 10	30 is evicted because 20 is just requested and it is assumed that the recent request of 30 is earlier than that of 10
30	20 5 10	m	20 5 30	10 is evicted because its recent request is earliest
10	20 5 30	m	10 5 30	recent request of 20 is earliest
5	10 5 30	h	no change	
20	10 5 30	m	10 5 20	5 is most recently requested, 10 second recently, 30 least recently requested

1.3 The programs

1.3.1 The program COMP108Paging.java

After understanding the eviction algorithms, your main task is to implement a class **COMP108Paging** in a java file named **COMP108Paging.java**. The class contains two static methods, one for each of the eviction algorithms. The signature of the methods is as follows:

```
static COMP108PagingOutput noEvict(int[] cArray, int cSize, int[] rArray, int rSize)
static COMP108PagingOutput evictLRU(int[] cArray, int cSize, int[] rArray, int rSize)
```

These methods are called by `COMP108Paging.noEvict()`, `COMP108Paging.evictLRU()`.

The four parameters mean the following:

- `cArray` is an array containing the cache content.
- `cSize` is the number of elements in cArray.
- `rArray` is an array containing the request sequence.
- `rSize` is the number of elements in rArray.

The results of the methods should be stored in an object of the class COMP108PagingOutput. The first line of each of the methods has been written as:

```
COMP108PagingOutput output = new COMP108PagingOutput();
```

and the last two lines as:

```
output.cache = arrayToString(cArray, cSize);
return output;
```

Details of how to use this class can be found in Section 1.3.2.

The class also contains a pre-written method to help with the tasks. **You must NOT change the content of this method.**

```
static String arrayToString(int[] array, int size)
```

This method will convert the cache content to a string. A statement is already written at the end of the two evicting methods to use it.

```
output.cache = arrayToString(cArray, cSize);
```

You can also use the method for debugging purpose.

You are encouraged to add additional methods in COMP108Paging.java that may be reused by different eviction algorithms (but this is not a requirement).

1.3.2 The program COMP108PagingOutput.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 COMP108PagingOutput has been defined in this program. It contains four attributes:

```
public int hitCount;
public int missCount;
public String hitPattern;
public String cache;
```

When we define an object of this class, e.g., the `output` in COMP108Paging.noEvict(), then we can update these attributes by, for example, `output.hitCount++`, `output.missCount++`, `output.hitPattern += "m"`.

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.

1.3.3 COMP108PagingApp.java

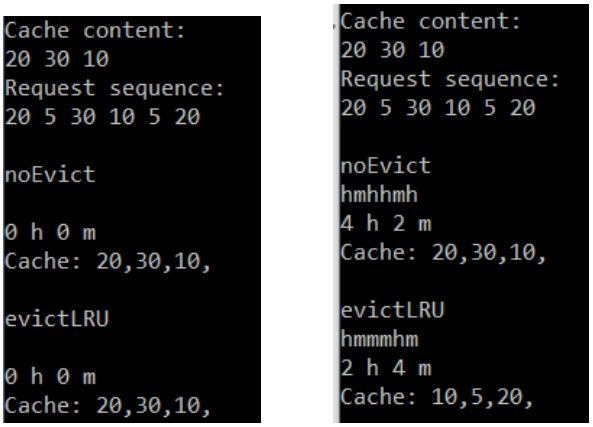
To assist you testing of your program, an additional file named COMP108PagingApp.java is provided. Once again, you should not change this program. **Any changes on this file will NOT be used to grade your submission.** This program inputs some data so that they can be passed on to the eviction algorithms to process. To use this program, you should first compile COMP108Paging.java and then COMP108PagingApp.java. Then you can run with COMP108PagingApp. See an illustration here:

```
>javac COMP108Paging.java

>javac COMP108PagingApp.java

>java COMP108PagingApp < pagingSampleInput01.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.



1.4 Your tasks

The tasks are associated with the two evicting algorithms each should return an object of class COMP108PagingOutput such that the following attributes of output is computed correctly:

(i) `hitPattern` in the form `<[h|m]*>` (see Test Cases in Section 1.5 for examples)

(ii) `hitCount` storing the number of hits

(iii) `missCount` storing the number of misses

(iv) `cache` storing the final cache content in comma-separated format (see Test Cases in Section 1.5 for examples)

Note that `hitCount + missCount` should be equal to `rSize`.

- **Task 1.1:** Implement the `noEvict()` method that does not evict any page. It should iterate the request sequence to determine for each request whether it is a hit or a miss.
- **Task 1.2:** Implement the `evictLRU()` method to evict the page whose most recent request was the earliest. You should assume that the intial cache content enters the cache in the order of `cArray[0], cArray[1], ...,` in other words, the most recent request of `cArray[0]` is earlier than that of `cArray[1]`, which in turn is earlier than that of `cArray[2]`, and so on.

  Hint: you may use another array to store when the recent request of a cache element is.

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 cache (between 1 and 10 inclusively)
- Initial content of the cache (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 5 30 10 5 20
	noEvict hmhhmh 4 h 2 m Cache: 20,30,10,
	evictLRU hmmmhm 2 h 4 m Cache: 10,5,20,

#2	Input: 5 10 20 30 40 50 4 15 50 10 20
	noEvict mhhh 3 h 1 m Cache: 10,20,30,40,50,
	evictLRU mhmm 1 h 3 m Cache: 15,10,20,40,50,
#3	Input: 4 20 30 10 40 14 40 40 30 30 20 5 5 5 15 15 15 15 10 40
	noEvict hhhhhmmmmmmmhh 7 h 7 m Cache: 20,30,10,40,
	evictLRU hhhhhmhhmhhhmm 10 h 4 m Cache: 40,10,5,15,

These test cases and their expected output can be downloaded on CANVAS. You can run the program easier by typing `java COMP108PagingApp < pagingSampleInput01.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: The List Accessing Problem

2.1 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.

2.2 The algorithms

We consider two 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

- **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

2.3 The programs

2.3.1 The program COMP108Cab.java

After understanding the algorithms, your main task is to implement a class **COMP108Cab** in a java file named **COMP108Cab.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 two methods, one for each of the reorganisation algorithms. The signature of the methods is as follows:

```
public COMP108CabOutput appendIfMiss(int rArray[], int rSize)
public COMP108CabOutput 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 COMP108CabOutput. The first line of each of the methods has been written as:
`COMP108CabOutput output = new COMP108CabOutput(rSize, 1);` and the last line as:
`return output;`
Details of how to use this class can be found in Section 2.3.3.

The class also contains a few pre-written methods to help with the tasks. **You must NOT change the content of these methods.**

```
public void insertHead(COMP108Node newNode)
public void insertTail(COMP108Node newNode)
public COMP108Node deleteHead()
public void emptyCab()
public String headToTail()
public String tailToHead()
public String headToTailFreq()
```

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 methods to illustrate how to use them.

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

2.3.2 The program COMP108Node.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 COMP108Node is defined to represent a node of the doubly linked list. It contains the following attributes:

```
public int data;
public COMP108Node next;
public COMP108Node 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.

2.3.3 The program COMP108CabOutput.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 COMP108CabOutput 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;
```

A constructor has been implemented that takes an integer parameters used to define the size of `compare[]`.

When we define an object of this class, e.g., the `output` in appendIfMiss() in COMP108Cab, 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.

2.3.4 COMP108CabApp.java

To assist you testing of your program, an additional file named COMP108CabApp.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 COMP108Cab.java and COMP108CabApp.java. Then you can run with COMP108CabApp. See an illustration below.

```
>javac COMP108Cab.java

>javac COMP108CabApp.java

>java COMP108CabApp < 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
Comparisons: 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
Comparisons: 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
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,

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,
```

2.4 Your tasks

The tasks are associated with the two algorithms each should return an object of class COMP108CabOutput 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 2.1:** Implement the **appendIfMiss()** method that appends a missed file to the end of the list and does not reorganise the list.
- **Task 2.2:** 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.

2.5 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,

	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,
	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,
	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 and their expected output can be downloaded on CANVAS. You can run the program easier by typing `java COMP108CabApp < 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.

3 Worst Case Time complexity analysis in big-O notation

- **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 justifiation of your answer.
- This should be added to the comment sections at the beginning of COMP108Paing.java and COMP108Cab.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 COMP108Paging.java, you can use any of the following in the formula: n to represent the number of requests and p the cache size.
- For COMP108Cab.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.
- You can define additional symbols if needed.
- For each algorithm, 3% is awarded to the time complexity and 2% to the justification.

4 Additional Information

4.1 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, you will lose all marks of the autotest. If your code compile to a class of a different name from COMP108Paging / COMP108Cab, again you will lose all marks of the autotest.
- **No** in-built methods can be used. For example, if you want to sort an array, you have to write your own code to do so and you are not allowed to use Array.sort(). Using in-built methods would **get all marks deducted** (possibly below the passing mark).

4.2 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. You should not show your code to other students. 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.