COMP108 Data Structures and Algorithms

Assignment 1

Deadline: Friday 12$^{\text{th}}$ March 2021, 5:00pm

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

Basic information

- Assignment: 1 (of 2)
- Deadline: Friday 12th March 2021 (Week 5), 5:00pm
- Weighting: 15% of the whole module
- Electronic Submission: Submit to SAM on departmental server
- What to submit: a java file named **COMP108A1Paging.java**
- Learning outcomes assessed:
  - Be able to apply the data structure arrays 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: 60%
  - Time complexity analysis: 20%
  - Programming style: 20%

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.

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, 30, 5, 30, 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 **hhmhmh** since 20 is a hit, 30 hit, 5 miss, 30 hit, 5 miss, 20 hit. There are **4 h and 2 m**.

(ii) **Evict FIFO.** This algorithm evicts the pages in the FIFO (first-in-first-out) principle. 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 now become **hhmhhm** with **4 h and 2 m**.

request	cache beforehand	hit/miss	cache afterward	remarks
20	20 30 10	h	no change	
30	20 30 10	h	no change	
5	20 30 10	m	5 30 10	20 in the cache longest and evicted
30	5 30 10	h	no change	
5	5 30 10	h	no change	
20	5 30 10	m	5 20 10	30 in the cache longest and evicted

(iii) **Evict LFU.** This algorithm evicts the pages in the LFU (least frequently used) principle. This means that the algorithm keeps a count of how many times a page in the cache has been requested since it is placed in the cache and evicts the page that has the smallest count. If there are two or more pages with the same count, the one with a smaller index (i.e., leftmost such page in the cache) is evicted. *We assume that the initial cache content each has* ***a count of 1*** *to start with.* The frequency count is reset to 1 if a page is evicted, in other words, a page entering the cache will have a count of 1. The hit and miss sequence will then be **hhmhhh** with **5 h and 1 m**.

request	cache beforehand	hit/miss	cache afterward	remarks
20	20 30 10	h	no change	
30	20 30 10	h	no change	
5	20 30 10	m	20 30 5	20 and 30 have count 2 while 10 has count 1 and is evicted
30	20 30 5	h	no change	
5	20 30 5	h	no change	
20	20 30 5	h	no change	

(iv) **Evict LFD.** This algorithm evicts the pages in the LFD (longest forward distance) principle. This means that the algorithm looks at future requests and evicts the page whose next request is the latest. If there are two or more pages with the same "distance" to next request because they are not requested anymore, the one with a smaller index (i.e., leftmost such page in the cache) is evicted. The hit miss sequence is **hhmhhh** with **5 h and 1 m**.

request	cache beforehand	hit/miss	cache afterward	position of next request
	20 30 10			1 2 ∞
20	20 30 10	h	no change	6 2 ∞
30	20 30 10	h	no change	6 4 ∞
5	20 30 10	m	20 30 5	6 4 5
30	20 30 5	h	no change	6 ∞ 5
5	20 30 5	h	no change	6 ∞ ∞
20	20 30 5	h	no change	∞ ∞ ∞

1.2 The programs

1.2.1 The program COMP108A1Paging.java

After understanding the four eviction algorithms, your main task is to implement a class **COMP108A1Paging** in a java file named **COMP108A1Paging.java**. The class contains four static methods, one for each of the eviction algorithms. The signature of the methods is as follows:

```
static COMP108A1Output noEvict(int[] cArray, int cSize, int[] rArray, int rSize)
static COMP108A1Output evictFIFO(int[] cArray, int cSize, int[] rArray, int rSize)
static COMP108A1Output evictLFU(int[] cArray, int cSize, int[] rArray, int rSize)
static COMP108A1Output evictLFD(int[] cArray, int cSize, int[] rArray, int rSize)
```

These methods are called by COMP108A1Paging.noEvict(), COMP108A1Paging.evictFIFO(), etc.

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 COMP108A1Output. The first line of each of the methods has been written as:

```
COMP108A1Output output = new COMP108A1Output();
```

and the last line as:

```
return output;
```

Details of how to use this class can be found in Section 1.2.2.

You are encouraged to add additional methods in COMP108A1Paging.java that may be reused by different eviction algorithms (but this is not a requirement).

1.2.2 The program COMP108A1Output.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 COMP108A1Output has been defined in this program. It contains three attributes:

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

When we define an object of this class, e.g., the `output` in COMP108A1Paging.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.

1.2.3 COMP108A1PagingApp.java

To assist you testing of your program, an additional file named COMP108A1PagingApp.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 COMP108A1Paging.java and then COMP108A1PagingApp.java. Then you can run with COMP108A1PagingApp. See an illustration here:

```
javac COMP108A1Paging.java

javac COMP108A1PagingApp.java

java COMP108A1PagingApp < sampleInput01.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.

```
Cache content:
20 30 10
Request sequence:
20 30 5 30 5 20

noEvict

0 h 0 m

evictFIFO

0 h 0 m

evictLFU

0 h 0 m

evictLFD

0 h 0 m
```

```
Cache content:
20 30 10
Request sequence:
20 30 5 30 5 20

noEvict
hhmhmh
4 h 2 m

evictFIFO
hhmhhm
4 h 2 m

evictLFU
hhmhhh
5 h 1 m

evictLFD
hhmhhh
5 h 1 m
```

1.2.4 Additional tool for checking your program before submission

Your program will be marked by check50. Some of the tests will be released to you before your submission so that you can check if (i) your program compiles, (ii) your methods have the correct header, (iii) your methods return appropriate objects. You can refer to a guidance on how to use the check50 tool.

1.3 Your tasks

There are six tasks you need to do. The first four tasks are associated with the four evicting algorithms each should return an object of class COMP108A1Output such that the following attributes of output is computed correctly:

(i) `hitPattern` in the form `<[h|m]*>` (see Test Cases in Section 1.4 for examples)

(ii) `hitCount` storing the number of hits

(iii) `missCount` storing the number of misses

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

- **Task 1 (15%)** 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 2 (15%)** Implement the `evictFIFO()` method to evict pages in a first-in-first-out principle. You should assume that the initial cache content enters the cache in the order of `cArray[0], cArray[1], ...,` in other words, `cArrray[0]` should be evicted first if needed, then `cArray[1]`, and so on.
- **Task 3 (15%)** Implement the `evictLFU()` method to evict the page which is least frequently used (requested). You should assume that each of the initial cache content is used once. When a cache page `cArray[i]` is evicted and replaced by a new page, the counter of `cArray[i]` should be reset to 1. When there are multiple pages with the same frequency of usage, the "leftmost" such page should be evicted; e.g., if `cArray[x]` and `cArray[y]` have the same lowest frequency of usage and `x < y`, then `cArray[x]` should be evicted and not `cArray[y]`.

  Hint: you may use an additional array to store the frequency of usage of the cache elements.
- **Task 4 (15%)** Implement the `evictLFD()` method to evict the page whose next request is the latest. When a cache page is requested or evicted and replaced by a new page, the position of next request has to be updated accordingly. When there are multiple pages with the same position of next request, the "leftmost" such page should be evicted.

  Hint: you may use an additional array to store the position of next request of the cache elements.
- **Task 5 (20%)** For each of the algorithms you have implemented, give a time complexity analysis (in big-O notation). You can use n to represent the number of requests and p the cache size in your formula. Give also a short justification of your answer. This should be added to the comment sections at the beginning of COMP108A1Paging.java. No separate file should be submitted.
- **Task 6 (20%)** (This is not a separate task.) In your implementation, you should keep a good programming style. Marks will be awarded based on consistent use of brackets, indentation, meaningful variable names, etc.

1.4 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 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	noEvict	evictFIFO	evictLFU	evictLFD

#1	`3`
	`20 30 10`
	`6`
	`20 30 5 30 5`
	`20`
		`hhmhmh`
		`4 h 2 m`
			`hhmhhm`
			`4 h 2 m`
				`hhmhhh`
				`5 h 1 m`
					`hhmhhh`
					`5 h 1 m`

#2	`5`
	`10 20 30 40 50`
	`4`
	`15 50 10 20`
		`mhhh`
		`3 h 1 m`
			`mhmm`
			`1 h 3 m`
				`mhmh`
				`2 h 2 m`
					`mhhh`
					`3 h 1 m`

#3	`4`
	`20 30 10 40`
	`14`
	`40 40 30 30 20`
	`5 5 5 15 15 15`
	`15 10 40`
		`hhhhhmm`
		`mmmmmhh`
		`7 h 7 m`
			`hhhhhmh`
			`hmhhhhh`
			`12 h 2 m`
				`hhhhhmh`
				`hmhhhmh`
				`11 h 3 m`
					`hhhhhmh`
					`hmhhhhh`
					`12 h 2 m`

These test cases can be downloaded as sampleInput01.txt, sampleInput02.txt, sampleInput03.txt on CANVAS and the output as sampleOutput01.txt, sampleOutput02.txt, sampleOutput03.txt.

You can run the program easier by typing `java COMP108A1PagingApp < sampleInput01.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 Additional Information

2.1 Time complexity analysis (20%)

- The time complexity has to match your implementation. Marks will NOT be awarded if the corresponding eviction algorithm has not been implemented.
- You are expected to write the time complexity and short justification in the comment section at the beginning of COMP108A1Paging.java.
- For each eviction algorithm, 3% is awarded to the time complexity and 2% to the justification.

2.2 Programming Style (20%)

- You should keep a good programming style. Marks will be awarded based on
  (i) 4% consistent and appropriate use of brackets
  (ii) 4% consistent and appropriate use of indentation
  (iii) 4% meaningful variable names
  (iv) 4% comments to explain the working of your program
  (v) 4% your personal information at the beginning of COMP108A1Paging.java

2.3 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 file is not named COMP108A1Paging.java, 5 marks will be deducted. If your code compile to a class of a different name from COMP108A1Paging, 5 marks will be deducted. These penalties will not deduct the marks below the passing mark.
- **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 built-in methods would **get all marks deducted** (possibly below the passing mark).

2.4 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.