Project 2: Cache Simulator

DUE: February 25th, 2024 at 11:59pm

Extra Credit Available for Early Submissions!

Basic Procedures

You must:

- Have a style (indentation, good variable names, etc.) and pass the provided style checker (See P0).
- Comment your code well in JavaDoc style and pass the provided JavaDoc checker (See P0).
- Implement all required methods to match the expected behavior as described in the given template files.
- For methods that come with a big-O requirement (check the provided template Java files for details), make sure your implementation meet the requirement.
- Have code that compiles with the command: `javac *.java` in your user directory without errors or warnings.
- Have code that runs with the command in your user directory: `java Simulator InputFile [-d]`

You may:

- Add additional helper methods, however they **must be private**.
- Add additional class/instance variables in some classes (check the template files for details), however they **must be private or protected**.

You may NOT

- Make your program part of a package.
- Add additional public methods or public class/instance variables. Remember local variables are not the same as class/instance variables.
- Use any built in Java Collections Framework classes anywhere in your program (e.g. no ArrayList, LinkedList, HashSet, etc.).
- Declare/use any arrays anywhere in your program (except the provided **`buckets`** field in **`BasicMap`**).
    - You may not call toArray() methods to bypass this requirement.
- Alter any method signatures defined in the template code. Note: "throws" is part of the method signature in Java, don't add/remove these.
- Add @SuppressWarnings to any methods unless they are private helper methods for use with a method that we provided which already has an @SuppressWarnings on it.
- Alter any fully provided classes (e.g. **`Node`**) or methods that are complete and marked in a "DO NOT EDIT" section.
- Add any additional import statements (or use the "fully qualified name" to get around adding import statements).
- Add any additional libraries/packages which require downloading from the internet.

Setup

- Download the **`p2.zip`** and unzip it. It contains a template for all the files you must implement.

Submission Instructions

- Make a new temporary folder and copy there your **.java** files. Do not copy the test files, jar files, class files, etc.
- Upload the temporary folder to OneDrive as a backup (this is not your submission, just a backup!)
- Follow the Gradescope link provided in Blackboard->Projects and upload the files from your temporary folder onto Gradescope's submission site. **Do not zip the files or the folder.**
- VERIFY WHETHER YOU HAVE UPLOADED THE RIGHT THING. **Submitting the wrong files will result in a 0 on the assignment!**

Grading Rubric

Due to the complexity of this assignment, an accompanying grading rubric pdf has been included with this assignment. Please refer to this document for a complete explanation of the grading, including extra credit for early submissions.

Overview

Cache can be viewed as a repository that keeps copies that can be accessed more quickly than the original. In computing, the idea of caching is used in many areas. Some examples:

- CPU cache with data copied from memory;
- Web cache with web pages downloaded from remote servers;
- DNS cache with mapping of names to IP addresses obtained from DNS resolvers.

To be cost-effective, caches must be relatively small and hence are commonly filled up soon. When a cache is full and some new data are retrieved, some previously existing entry of the cache must be removed to make room for the new member. Different cache replacement policies have been proposed to decide how to pick the victim entry to be removed. For this project, we will construct a cache simulator to demonstrate and compare different cache replacement policies.

Terms:

- **Cache Block**: One entry in cache that is the unit of cache storage and access.
- **Cache Access**: The operation of searching for a particular address in cache and returning the block associated with that address.
  - **Cache Hit**: When the address we attempt to access is already in cache, the access is a cache hit.
  - **Cache Miss**: When the address we attempt to access is not present in cache, the access is a cache miss. The block of the requested address will need to be loaded into cache. If all cache entries are already in use, then an existing block needs to be replaced.
- **Cache Replacement**:
  - The operation of evicting one existing block to make room for a new block. This is only performed at a cache miss when the cache is full.

Cache Replacement Policies we will implement in this project:

First In First Out (FIFO) Replacement

The cache behaves like a FIFO queue. The block to replace is always the one that loaded into cache the earliest in time.

Table 1: FIFO Cache Sample

Access	C	D	C	E	D	C	F	C	A
Cache Content (First In to Last In)	C	C	C	C	C	C	D	E	F
		D	D	D	D	D	E	F	C
				E	E	E	F	C	A
Replaced	-	-	-	-	-	-	C	D	E
Hit/Miss	Miss	Miss	Hit	Miss	Hit	Hit	Miss	Miss	Miss

NOTES:

- Assume a FIFO cache with capacity 3; start with an empty cache.
- Blue color highlights the latest block loaded into cache. Red color marks the block evicted by a miss.
- Cache content in the table is presented in the ascending order of the time that each block is loaded into the cache. Top block is the first one added into the cache while the bottom one is the last one loaded in. This is essentially a FIFO queue.

Least Recently Used (LRU) Replacement

The block to replace is the block that has not been accessed for the longest time, or, the least recently accessed block.

Table 2: LRU Cache Sample

Accesses	C	D	C	E	D	C	F	C	A
Cache Content (Least Recently Accessed to Most Recently Accessed)	C	C	D	D	C	E	D	D	F
		D	C	C	E	D	C	F	C
				E	D	C	F	C	A
Replaced	-	-	-	-	-	-	E	-	D
Hit/Miss	Miss	Miss	Hit	Miss	Hit	Hit	Miss	Hit	Miss

NOTES:

- This is based on the same sequence of accesses as in Table 1.
- Assume an LRU cache with capacity 3; start with an empty cache.
- Cache content in the table is presented in the ascending order of the time that each block is last accessed. Top block is the least recently accessed while the bottom one is the most recently accessed. Therefore, even for a cache hit with no replacement, the cache content, or more specifically, the ordering of the blocks may still change.

Least Frequently Used (LFU) Replacement

The number of accesses of each block is recorded and the block to replace is the one with the lowest count. If there is a tie in access frequency, pick the one that is least recently accessed to evict.

Table 3: LFU Cache Sample

Address	**C**	**D**	**C**	**E**	**D**	**C**	**F**	**C**	**A**
Cache Content (Least Frequently Accessed to Most Frequently Accessed)	C,1	C,1	D,1	D,1	E,1	E,1	F,1	F,1	A,1
		D,1	C,2	E,1	C,2	D,2	D,2	D,2	D,2
				C,2	D,2	C,3	C,3	C,4	C,4
Replaced	-	-	-	-	-	-	E	-	F
Hit/Miss	Miss	Miss	Hit	Miss	Hit	Hit	Miss	Hit	Miss

NOTES:

- This is based on the same sequence of accesses as above.
- Assume an LFU cache with capacity 3; start with an empty cache.
- Cache content in the table is presented in the ascending order of the number of times each block has been accessed since they were loaded in. For each block, we also show the access counter in the table. When there is a tie, blocks are ordered from least recently accessed to most recently accessed. For each access, the cache content would need to be updated (with a new block or with a new counter value), and the ordering of blocks may change.

To simulate a cache, we will use a linked list to store the blocks loaded into cache. Based on the current list, we need to be able to determine whether the next request is a cache hit or cache miss, and to perform necessary updates to the cache based on different replacement policies. One other analysis we will implement for this project is to use a Hash Map to keep track of the number of accesses and the number of hits we have observed for each address in a sequence. The example table below shows the record we have at the end of the sequence from Table 3.

Table 4: History Record of Addresses from Table 3

Address	**C**	**D**	**E**	**F**	**A**
Number of Accesses	4	2	1	1	1
Number of Hits	3	1	0	0	0

Implementation/Classes

This project will be built using a number of classes representing linked lists, caches of different replacement policies, and the cache simulator. Here we provide a description of these classes. Template files are provided for each class in the project zip file and these contain further comments and additional details. You must follow the instructions included in those files.

- **Node (Node.java)**: The node used in linked list class **BasicList** and **SortedList**. This class is provided to you and you should NOT change the file.
- **BasicList (BasicList.java)**: The implementation of a generic singly linked lists. We will use this to construct other classes of this project.
- **SortedList (SortedList.java)**: The implementation of a linked list with nodes sorted. It is a subclass extended from **BasicList**.
- **Cache (Cache.java)**: The interface that define the common operations supported by all caches. This class is provided to you and you should NOT change the file.

- **FifoCache(FifoCache.java)**: The implementation of a cache following FIFO replacement policy. This class implements **Cache** interface. The storage of the cache is built upon **BasicList.**
- **LruCache(LruCache.java)**: The implementation of a cache following LRU replacement policy. This class implements **Cache** interface. The storage of the cache is built upon **BasicList.**
- **LfuCache(LfuCache.java)**: The implementation of a cache following LFU replacement policy. This class implements **Cache** interface. The storage of the cache is built upon **SortedList.**
- **BasicMap (BasicMap.java)**: The implementation of a hash map using separate chaining for conflict resolution. It is also built upon **BasicList**.
- **Simulator (Simulator.java)**: The implementation of a cache simulator. It constructs a cache, reads in a sequence of addresses from an input file, sends the access request of each address to the cache one at a time, and keeps track of the access/hit counters for each address in the sequence. This class is provided to you and you should NOT change the file.

- There are a number of fields and methods already implemented and provided to you in some of the java classes above. Do not change the provided methods but you will need to add JavaDocs for them.

Requirements

An overview of the requirements is listed below, please see the grading rubric and template files for more details.

- **Implementing the classes** - You will need to implement required classes and fill the provided template files.
- **JavaDocs** - You are required to write JavaDoc comments for all the required classes and methods.
- **Big-O** - Template files provided to you contains instructions on the REQUIRED Big-O runtime for many methods. Your implementation of those methods should NOT have a higher Big-O.

How To Handle a Multi-Week Project

While this project is given to you to work on for about three weeks, you are unlikely to be able to complete everything in one weekend. We recommend the following schedule:

- Step 1 (**`BasicList, SortedList`**): First week (by 02/11)
  - Implement and test methods of the linked lists.
- Step 2 (**`FifoCache, LruCache, LfuCache`**): Before second weekend (02/11-02/16)
  - Implement and test methods of different caches**.**
  - You will be able to use the provided **`Simulator`** to check cache simulation for any access sequence (excluding the option **-d** for detailed history record).
- Step 3 (**`BasicMap`**): Second weekend (02/17-02/18)
  - Implement and test methods of **`BasicMap`**.
  - You will be able to use the provided **`Simulator`** with the **-d** option to include a detailed record for each address in the access sequence.
- Step 4 (**`Wrapping-up`**): Last week (02/19-02/25)
  - Additional testing, debugging, get additional help.
  - ☺ Also, notice that if you get it done early in the week, you can get extra credit! Check our grading rubric PDF for details.

Testing

The main methods provided in the template files contain useful example code to test your project as you work. You can use command like "**`java BasicList`**" or "**`java BasicMap`**"to run the testing defined in **`main()`**.

- ***Note:*** passing all yays does NOT guarantee 100% on the project! Those are only examples for you to start testing and they only cover limited cases. Make sure you add many more tests in your development. You could edit **`main()`** to perform additional testing.
- JUnit test cases will not be provided for this project, but feel free to create JUnit tests for yourself. A part of your grade *will* be based on automatic grading using test cases made from the specifications provided.

We have provided the UI in **`Simulator`** so that you can run the cache simulation with an input file to specify the sequence of address accesses to the cache. The simulation will have two modes: ***normal*** and ***detailed***. In ***normal*** mode, the simulation will process one access at a time, displaying the cache content to show how cache is updated for each access, and report a cache miss rate for the complete sequence in the end. In ***detailed*** mode, the simulation will further record the number of accesses and number of hits for each address in the sequence, and generate a report at the end of the simulation. The **`Simulator`** is run with in the following ways:

- **`java Simulator InputFile`**
  This will run the simulation in the normal mode.

- **`java Simulator InputFile -d`**
  This will run the simulation in the detailed mode.

To help with testing, we provide a number of input files that you can use with **`Simulator`** under the folder **input/**. We also include multiple sample-runs in **`project2-sample-runs.pdf`** to show you the expected behavior/printing.

- ***Note:*** as always, matching all provided sample runs does NOT guarantee 100% on the project!
- You should test with more scenarios with more files of your own.