Assignment 3
Cache Implementation (cont'd)

Dr. Nguyen Hua Phung

May 2021

1 Student Outcomes

After completing this assignment, students will be able to

- implement a hash search
- select and manipulate data structures suitable to desired needs.

2 Introduction

In the previous assignment, a cache is implemented using AVL tree for searching and a queue for FIFO replacement policy. Actually, there are many different approaches for searching and replacement policies. Every approach has advantages and disvantages. In this assignment, you are required to implement different searching and replacement policies. In addition, these approaches must be combined in a system so that the cache can be applied with different combinations.

For searching, this assignment requires two different approaches: AVL tree and hashing while for replacement, there are four different policies: Least recently used (LRU), Most recently used (MRU), Least frequently used (LFU) and First In First Out (FIFO). These policies must be implemented as different classes. When creating a cache, it must be passed an object for searching and another object for replacement policy. The details will be described in next section.

2.1 Cache Implementation

A cache, which is implemented in class Cache, has two parameters when it is initialized: a search engine and a replacement policy. The main interface (read, put, write) of the cache is unchanged, while other methods (print, preOrder, inOrder) are replaced by printRP and printSE. The details of these two methods are described as follows:

- **void printRP()**: prints the value in the buffer of the replacement policy. This method must print the string "Print replacement buffer\n" and then call the print method of the corresponding replacement policy. Printing detail for each replacement policy is described in the corresponding policy.

- **void printSE()**: prints the value in the buffer of the search engine. This method must print the string "Print search buffer\n" and then call the print method of the corresponding search engine. Printing detail for each search engine is described in the next section.

The declaration of the Cache class is in file main.h.

2.1.1 Search Engine

The search engine is implemented as an abstract class SearchEngine declared in file Cache.h but you are allowed to change the this class except its name.

There are two concrete subclasses of this abstract class: AVL and DBHashing. The AVL class which is from the previous assignment is used to search in AVL tree while the DBHashing class is used to search using double hashing. The constructor of DBHashing must have 3 parameters: the first and second parameters are two hashing functions (hash1, hash2) while the third parameter (size) is the size of the hash table. They are used to determine the position of the **key** in the hash table by the function

hp(key,i) = (hash1(key) + i * hash2(key)) % size.

Each concrete class must have a print method which prints the elements in the buffer as follows:

- **AVL**: prints the string "Print AVL in inorder:\n" and then every elements in the AVL in in-order (LNR) and then prints the string "Print AVL in preorder:\n" and then every elements in the AVL in pre-order (NLR).

- **DBHashing**: prints the string "Prime memory:\n" and then all existence elements in the prime memory of the cache in the ascending order of the index.

2.2 Replacement Policies

The replacement policy is implemented by an abstract class **ReplacementPolicy** declared in file Cache.h. You are allowed to change everything of this class except its name.

There are 4 concrete subclasses to implement different replacement policies:

- **FIFO**: uses First-In-First-Out policy which is implemented in the previous assignment. When printing, this class will print the elements in the cache in the descending order of the element living time in the cache (like the previous assignment)

- **MRU**: uses Most-Recently-Used policy which selects the most recently used element by a read or write to remove when the cache is full. To implement this policy,self-organizing list with move-to-front approach should be used. When printing, this class will print the elements in the self-organizing list from the first to the last.

- **LRU**: uses Least-Recently-Used policy which selects the least recently used element. This policy also uses the move-to-front approach but removing the last element. When printing, this class will print the same as the MRU class

- **LFU**: uses Least-Frequently-Used policy which selects the least frequently used elements in the cache. A count field added to each element in the cache is used to count how many times the corresponding element is read or written. A min-heap must be applied to rearrange elements in the cache based on the count field. When applying re-heap up (moving an element from a leaf up to root), the child will swap with its parent when the count of the child is less than the count of its parent. When applying re-heap down (move an element down to the leaf), the parent will swap with the child which has the minimum count between the two children when the count of the parent is higher than or equal to the count of the child. When comparing the counts of two childern,if they are the same, the minimum child is conventionally the left child. When printing, this class will print the elements in the heap by level from left to right.

There is no parameter in the construction of these classes. The size of cache must be the value of variable MAXSIZE declared in main.h.

2.3 Instructions

To complete this assignment, students must

- Download the initial code, unzip it

- There are 4 files: main.cpp, main.h, Cache.cpp and Cache.h. You MUST NOT modify files main.cpp and main.h.

- Modify files Cache.h and Cahe.cpp to implement the cache memory. Just keep the class names (ReplacementPolicy, SearchEngine, AVL, DBHashing, FIFO, MRU, LRU, LFU) unchanged.

- Make sure that there is only one **include** directive in file Cache.h that is **♯include "main.h"** and also one **include** in file Cache.cpp that is **♯include "Cache.h"**. No more include directive is allowed in these files. If the submission violates this requirement, it gets 0 mark.

3 Submission

Files Cache.h and Cache.cpp are required to submit as attachments before the deadline is given in the link "Assignment 3 Submission". There are some simple testcases used to check your solution to make sure your solution can be compiled and run. You can submit as many as you like but just the last submission is marked. As the system cannot response too many submissions in the same time, you should submit as soon as possible. You will take your own risk if you submit on the time of deadline. After the deadline, you cannot submit anymore.

4 Plagiarism

You must complete this assignment by yourself and you must prevent your solution from being stolen by someone else. Otherwise, you will be punished by the university rule for plagiarism.