Assignment 2
Cache Implementation (cont'd)

August 2022

1 Student Outcomes

After completing this assignment, students will be able to

- get familiar with hash techniques.
- select and manipulate data structures suitable to desired needs.

2 Introduction

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

This assignment still uses BST for searching but requires four different policies, such as:

1. MFU - Most Frequently Used.
2. LFU - Least Frequently Used.
3. MFRU - Most Frequently Recently Used.
4. LFRU - Least Frequently Recently Used.

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 and printLP. The details of these three 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 the search engine is described in the next section.
- void printLB(): first, prints the string "Prime memory/n", then prints all the elements in the main memory of cache in ascending index order. After that, prints the string "Hash table memory/n", and print the values in the hash table (its size is equal to the main memory) in ascending index order. Known that the hash table is generated as follows:
    1. Select each element in the main memory of cache in ascending index order.
    2. For each selected element, the element's position in the hash table (index of the hash table) will be decided through a hash function (mod function), specifically: **index = key % MAXSIZE** . Where MAXSIZE is the size of the hash table.
    3. In case of collision, the hash table is searched sequentially that starts from the original location of the hash (Linear probing).

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

2.2 Search engine

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

There is only one concrete subclass of this abstract class: BST. The BST class which is from the previous assignment is used to search in BST.

BST class must have a print method which prints the elements in the buffer as follows: prints the string "Print BST in inorder:/n" and then every elements in the BST in in-order (LNR) and then prints the string "Print BST in preorder:/n" and then every elements in the BST in pre-order (NLR).

2.3 Replacement Policy

The replacement policy is implemented by an abstract class ReplacementPolicy declared in file Cache.h. You are allowed to change everything in this class except its name. There are 4 concrete subclasses to implement different replacement policies:

- **MFU**: uses **Most-Frequently-Used** policy which selects the most 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 max-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 greater 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 maximum count between the two children when the count of the parent is smaller than or equal to the count of the child. When comparing the counts of two children, if they are the same, the maximum child is conventionally the right child. When printing, this class will print the elements in the heap by level (from high to low) and from right to left.

- **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 children, 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 low to high) and from left to right.

- **MFRU**: uses **Most-Frequently-Recently-Used** which selects the most frequently recently used elements in the cache. When printing, this class will print the data in order of most used to least element. If the elements have the same number of uses, then the most recently used element will be printed first.

- **LFRU**: uses **Least-Frequently-Recently-Used** which selects the least frequently recently used elements in the cache. When printing, this class will print the data in order of least used to most element. If the elements have the same number of uses, then the least recently used element will be printed first.

There is no parameter in the construction of these classes. The size of cache must be the value of variable MAXSIZE which is declared in main.h. Moreover, it is also the size of the main memory of cache.

2.4 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, BST, MFU, LFU, MFRU, LFRU) 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 2 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 Harmony

There is a question in the exam that are related to the content of the assignment. It will be clearly stated that the question is used to harmonize the assignment. The scores of harmony questions will be scaled to 10 and will be used to recalculate scores for the assignments. Specifically:

- Let x be the score of assignment.
- Let y be the score of harmony question after scaling to 10.

The final assignment score will be:

$$\mathbf{Assignment\_Score = 2 * x * y / (x + y)}$$

5 Cheating

Similarity less than 30% in C++ code is allowed. In other words, you will get 0 if your answers are similar to another student's more than 30%. We will use the Stanford MOSS system to check the similarity