Project 4: OPT Cache Simulation

DUE: Tuesday, May 1st at 11:59pm

Basic Procedures

You must:

- Fill out a readme.txt file with your information (goes in your user folder, an example readme.txt file is provided)
- Have a style (indentation, good variable names, etc.)
- Comment your code well in JavaDoc style (no need to overdo it, just do it well)
- Have code that compiles with the command: javac *.java in your user directory

You may:

- Add additional methods, fields, and classes, however these must be private (or package default for fields/methods inside nested classes).
- **Allowed classes to use as the starting point of your implementation (either import or include source code in your submission):**
    - Java class: util.HashMap, util.TreeSet
    - Any Weiss code from http://users.cs.fiu.edu/~weiss/dsj4/code/code.html
    - One additional Weiss class: http://users.cs.fiu.edu/~weiss/dsaajava3/code/AvlTree.java
    - Any class you implemented for a previous project of CS310 this semester

You may **not**:

- Make your program part of a package.
- Add additional public methods, variables, or classes.
- Import / use any built-in Java Library classes that are not included in the above allowed list (e.g. no ArrayList, LinkedList, Queue, etc.) or add any additional import statements.
- Alter any method signatures defined in this document or the template code.
- Add any additional libraries/packages which require downloading from the internet.

Setup

- Download the project4.zip and unzip it. This will create a folder `section-yourGMUUserName-p4`
- Replace "`section`" with `z001, z002, k003`, and `z004` for the 4 sections respectively.
- Rename the folder replacing `yourGMUUserName` with the first part of your GMU email address/netID. Example: `k003-jkrishn2-p4`
- Complete the `readme.txt` file (an example file is included)

Submission Instructions

- Make a backup copy of your user folder!
- Remove all test files, jar files, class files, etc.
- **In additional to your java files and your readme.txt, you will need to submit a design document** as either a plain TEXT file (design.txt) or a PDF file (design.pdf) which include
    - Description of the underlying data structures for the three interfaces you need to implement for this project.
    - The source (reference) of any class that you used in your code that is not written by you. These must be from the list of allowed classes from above.

    - For your implementation of `MySequence` class, a description and algorithm sketch of method `countNoSmallerThan()`
- Zip your user folder (not just the files) and name it as "`section-yourGMUUserName-p4.zip`" (no other type of archive) where "`yourGMUUserName`" is your GMU email address / netID.
- Submit to blackboard.

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.

Topics Covered

Self-Balancing Binary Search Tree, Priority Queue, Hash Table

Overview

In this project, your task is to simulate how cache works using the OPT replacement policy to replace the addresses stored. Cache is the component that stores data that is in demand and can be quickly accessed. However, unlike long-term storage spaces, cache consists of only a limited amount of space that consequently requires one of the addresses to be replaced with a newly accessed address that is not in the cache. If an address requested by a reference is already in cache, the reference is considered as a cache **hit**; otherwise, it is considered as cache **miss**. A new address will need to be added into the cache when there is a cache miss. If there is still unused space in cache, it is straightforward to add. If the cache is already full, we will have to select one existing address from cache to **replace**. Check http://en.wikipedia.org/wiki/CPU_cache for an overview of cache.

There are a lot of cache replacement policies. You can find an overview of them here: http://en.wikipedia.org/wiki/Cache_replacement_policies. The one that you need to implement is called **OPT** (Optimal Page Replacement) policy[1]. When a replacement is needed, this policy selects the address whose next use will occur farthest in the future as the victim to be swapped out. It outperforms other cache replacement policies but not implementable in practice because future information is needed to make the decision of replacement. Check and [1] and page 5 of this document for some examples.

We will assume that the complete sequence of references is available and simulate a cache with OPT replacement policy in this project. The simulation is performed in two rounds of processing on the given sequence of accesses [1].

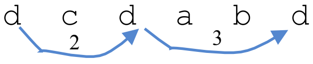
- The first round is used to collect necessary information regarding future references so that we can always order the existing addresses in a cache based on who will not be needed for the longest time. This will be performed by scanning the given sequence of references in reverse order.
- The second round will simulate the OPT cache, access the address sequence in order one by one, using the information collected in the first round to decide replacement candidate when needed, and accumulate the number of cache hits/misses.

Round 1

For the first round, we scan the given sequence of references in reverse order, and collect an LRU stack distance defined as below for each access.
Definition 1: **LRU Stack distance** of a reference to **`X`** at time **`T`** is the number of distinct addresses referenced at time **`T', T'+1, T'+2, . . ., T-1`** where **`T'`** is the last time **`X`** is referenced. If **`X`** has never been referenced before, the stack distance is infinity.

For example, using a letter to represent each address, we can represent a reference sequence and the corresponding LRU stack distance for each access as below:

Access time:	0	1	2	3	4	5	6	7	8	9
(Reverse) Access sequence:	d	c	d	a	b	d	a	c	b	a
LRU stack distance:	∞	∞	2	∞	∞	3	3	4	4	3



Explanation for some examples:

- The access to address d at time 0 has an infinite stack distance since d has never been accessed before.
- The access to address d at time 2 has a stack distance of 2 because two distinct addresses (c and d itself) have been accessed since last time d is accessed (at time 0).
- The access address c at time 7 has a stack distance of 4 (note not the same as time distance!) since starting from time 1, four distinct addresses (d, a, b and c itself) have been accessed.

In order to measure the LRU stack distance efficiently, we will need two supporting data structures:

1. A symbol table that remembers the last access time for each address. The symbol table needs to support efficient search for the last access time based on an address, adding a new address, and updating the last access time for existing address. Check **`SymbolTable.java`** for detailed Big-O requirement and decide which data structure you should use.
2. Storage of the last access times of all addresses we have processed. They need to be stored as a sorted sequence and must support efficient insertion, deletion, and counting how many values are there are no smaller than a specified value. Check **Sequence.java** for detailed Big-O requirement and decide which data structure you should use. **Hint:** any self-balancing search tree will satisfy most of the requirements except for the counting. You will need to augment the normal self-balancing search tree to support this.

Round 1 Classes Overview

SymbolTable: Maintains the table for unique symbols (addresses) from a sequence of references. In the pair `<SymbolType, RecordType>` *`SymbolType`* represents different symbols and *`RecordType`* defines the record of each symbol (address), in our case the last access time. The class should support operations to count how many symbols are present in the table (`size`), to check whether a symbol is present in the table (`hasSymbol`), to find and return the record we store in table for a symbol (`getRecord`), to set the record for a symbol (`putRecord`), and to find and remove the entry of a symbol from the table (`removeSymbol`). Class

`SymbolTable.java` defines the required interface while **`MySymbolTable.java`** is the actual class you need to write code and implement SymbolTable interface.

Sequence: Stores a collection of values as a sorted sequence with no duplicates. It should support operations such as inserting a new value in to the collection (`insert`), removing a value from the collection (`remove`), finding if a value is present in the collection (`contains`), finding how many values are present in the collection (`size`), and finding a string representation of all values in collection in ascending order (`toStringAscendingOrder`). It also should support the method (`countNoSmallerThan`) to count the values in the collection that are greater than or equals to a specific value. Class **`Sequence.java`** defines the required interface while **`MySequence.java`** is the actual class you need to write code and implement Sequence interface.

StackDistCollector: The class that calculates the (LRU) stack distance for any given sequence of accesses. Class **`StackDist.java`** has been partially implemented and you will need to use both symbol table and sequence in this class. The only method you need to implement in this class is **`access`**. This method takes in an address (represented as an integer in this project) and returns an integer value representing the LRU stack distance for that access. The inner class `SymbolRecord`, stores record we want to keep for every symbol (address) from the access sequence used in symbol table construction. Below is a scratch of steps to take to implement **`access()`**. For an access to **`X`** at time **`T`**:

1. Search the symbol table to find the last time **`X`** is accessed as **`T'`**
2. Count and report how many distinct accesses are there in range **`[T', T-1]`** in the sequence
3. Remove record for **`T'`** from the sequence storage and add a record for **`T`** in it
4. Update symbol table for the new last access time for **`X`**

Round 2

For the second round, we will assume the future information is available (collected in round 1) and scan the given sequence of references in order to simulate an OPT cache. The cache is maintained as a priority queue with the following features:

- The cache size is fixed
- If the cache is full and a new address needs to be accessed, the address with the maximal forward distance (defined as below) will be replaced

We should therefore implement cache as a fix-sized max-priority queue using the forward distance as the priority of each address. If there is a tie between forward distances, use the (unique) address to break the tie – the larger address has a higher priority.

Definition 2: **Forward distance** of an address to **`X`** at time **`T`** is the number of distinct addresses referenced at time **`T+1, T+2, . . ., T'`** where **`T'`** is the next time that **`X`** is referenced. If **`X`** will not be accessed ever again, the forward distance is infinity.

At time T, if X is accessed and the LRU stack distance of that access is L (from round 1), we should update the forward distance for all addresses in cache based on the following rules:

- forward_distance(X) = L (same as LRU stack distance)

- for address Y ≠ X
    - if forward_distance(Y) <= L, forward_distance(Y) = forward_distance(Y) -1
    - otherwise, forward_distance(Y) not changed

Example:

Access time:	0	1	2	3	4	5	6	7	8	9
Access sequence:	a	b	c	a	d	b	a	d	c	d
LRU stack distance:	3	4	4	3	3	∞	∞	2	∞	∞
(collected reversely in round1)										
Cache*	a(3)	b(4)	c(4)	c(4)	d(3)	b(∞)	b(∞)	b(∞)	c(∞)	c(∞)
of size 3:		a(2)	b(3)	a(3)	a(2)	d(2)	d(1)	d(2)	d(1)	d(∞)
			a(1)	b(2)	b(1)	a(1)	a(∞)	a(∞)	a(∞)	a(∞)
Hit/Miss	M	M	M	H	M	H	H	H	M	H
Replacement	-	-	-	-	c	-	-	-	b**	-

* cache organized as a max priority queue, with the max item kept at top. X(N) represents an address with forward distance N.
** both a and b have the same priority; break the tie using alphabetic order.

Round 2 Classes Overview

SymbolTable: Same as part 1.

Priority Queue: The class maintains a max priority queue where priority of a parent node is higher than the priority of its children. It should support operations to count the number of items (`size`), to return the max item (`peek`), to remove the max item (`remove`), to insert an item with its corresponding priority in to the queue (`insert`), to check whether there exists an item with a priority (`contains`), and to update the priority of an item (`updatePriority`). Class **`PriorityQueue.java`** defines the required interface while **`MyPriorityQueue.java`** is the actual class you need to write code which implements PriorityQueue interface.

OPTCacheSimulator: The class that simulates a cache with OPT replacement policy. Class **`OPTCacheSimulator.java`** has been partially implemented and you will need to use both symbol table and priority queue in this class. The only method that you need to implement is **`access`**. This method takes in an address and the corresponding stack distance, simulates the cache processing and returns a boolean value representing whether the access is cache hit or miss. Below is a scratch of steps to take to implement **`access()`**. For an access to **`X`** at time **`T`** with stack distance **`L`**:

1. Search the symbol table to determine hit/miss and report that
2. If it is a miss but the cache is not full, add **`X`** into cache. If it is a miss and the cache is full, remove address with the maximum forward distance, add **`X`**.
3. Update the symbol table
4. Update the forward distances for addresses in cache as described above

Big-O

Template given to you in the starter package contains instructions on the REQUIRED Big-O runtime for a subset of methods. For those methods, your implementation should not have a higher Big-O and you will be graded on this. When you need to select a data structure that satisfies all big-O requirement for an interface.

Testing

Test cases will not be provided for this project. However, feel free to create test cases by yourselves. In addition, the main methods provided along with the template classes contain useful code to test your code. You can use command like "`java OPTCacheSimulator`" to run the testing defined in main(). You could also edit main() to perform additional testing. As always, a part of your grade will be based on automatic grading using test cases that are not provided to you. A set of input files is provided to you for testing purpose. Take a look at `processSequence` methods in `OPTCacheSimulator` and `StackDistCollector`, which can be used to perform a batch of tests. The method resets everything, opens a file, read in from the file, and process accesses one by one. More sequences will be available in the following couple of weeks.

References

[1] Mattson, R. L., Gecsei, J., Slutz, D. R., et al. *Evaluation Techniques for Storage Hierarchies*. IBM Systems Journal 9 (2): 78-117, 1970. Retrieved from: http://inst.eecs.berkeley.edu/~cs266/sp10/readings/mattson70.pdf