COMPSCI 342: Data Structures and Algorithms
Programming Assignment #1: An Object Cache (60 points)
Due on 2/19/2008, Tuesday by 9pm.

cache: (k[aˆ]sh), n. [F., a hiding place, fr. cacher to conceal, to hide.]

1. A hole in the ground, or other hiding place, for concealing and preserving provisions which it is inconvenient to carry. –Kane. [1913 Webster]

2. That which is hidden in a cache[2]; a hoard; a stockpile.

3. (Computers) A form of memory in a computer which has a faster access time than most of main memory, and is usually used to store the most frequently accessed data in main memory during execution of a program.

1 Introduction

This programming assignment asks you to design and implement an object `Cache` class that uses the LRU (Least Recently Used) scheme. The cache has a fixed size and stores references to the most recently used objects. The cache should be able to store any comparable object that has a *key* value. We will also write a `CacheTest` class to test our cache implementation for two different types of distributions.

First create an assignment folder on `onyx` as follows:

```
mkdir program1; cd program1
```

Then copy the sample files for this assignment into that folder

```
cp -a  amit/cs342/lab/cache .
```

2 How does a Cache work?

Whenever an application requires a data item, it attempts to get it from the cache. If the object is found in the cache, we call that a *cache hit*. If the object isn't found in the cache, we call that a *cache miss*.

In case of a cache hit, the cache returns a reference to the object and object gets moved to the first position in the cache (the most recently used object). On the other hand, if it is a cache miss, then typically the application needs to read the object from the disk (or some other external source like a database) and then puts the object in to the first position of the cache. Note that if the cache is full, the last entry (the least recently used one) in the cache will be removed before a new object can be added. Depending upon the application, the removed entry may or may not needed to be stored on the disk (or some other external

location, like a database). In our case, we will assume that the removed object doesn't need to be written to the disk (or some other external location).

3 Solution Approach

- First, we need to design a data structure to hold the Cache. This could simply be a linked list or a combination of a linked list and a hash table. We should use the built-in `LinkedHashMap` data structure available in the `java.util` package as it combines a linked list and a hash table. We can either extend the `LinkedHashMap` class or create a `Cache` class that wraps around the `LinkedHashMap` class.
- Then we need to design the signatures for the methods in the Cache class. The `Cache` class should at least have the following public methods– `get, put, clear, getHitRatio`. The constructor will set the size of the cache. The `clear` method empties the Cache. The `get` method looks up an object based on the key provided by the user. The `put` method puts an object (using its key) into the Cache.
- Our cache class should be a generic template based class. It should be able to store any object that implement the interface `Element.java` that we have provided in the sample code (see above on how to get the sample code).

4 Testing

For the purposes of testing, we will use some real objects. The objects are stored in an external file. You can generate different-sized data files using the program `GenerateData.java`, which is also provided in the sample folder. The program `ReadData.java` shows you how to read the objects out of this binary file. The objects we will be storing in the cache are defined in the class `MyObject.java`. There are a few other relevant files as well in the sample folder.

We will test the cache performance using two different types of input distributions. First, we will use uniformly distributed random numbers (the default for Java's Random class). The `keyRange` is the maximum key value found in the data file. We will generate random objects with integer keys in the range [1..*keyRange*] for testing purposes. We want to measure and print the hit ratio for this experiment.

Next we will use the `nextGaussian()` method in the Random class to generate a half-normal input distribution, which mimics many real situations better. See the example code `HalfNormal.java` for an example. Again we will measure and print the hit ratio.

The test program must be called CacheTest and must accept the following command line arguments.

```
java CacheTest <maxCacheSize> <keyRange> <dataFile> <numTests> <stdDev> <seed>
```

where `maxCacheSize` is the limit on how big the cache can be,
`dataFile` is the file that stores the objects,
`keyRange` is the range of key values in the data file,
`numTests` is the number of cache lookups to attempt,
`stdDev` is the standard deviation for the Gaussian distribution, and
`seed` is the seed for the random number generator.

In case of a cache miss, for the purposes of testing, we will read in the missing object from the data file with the specified key value and put it in the cache.

Below is a sample output that you can use as a template for your program's output. The parameter values used are arbitrary. You should observe your program for a variety of values to get a better understanding of cache performance.

```
[amit@kohinoor generic-cache-v2]: java CacheTest 1000 10000 data.bin 1000 10 1221
CacheTest (Random):  maxCacheSize=1000 numTests=1000 hit ratio=5%
CacheTest (Half-Normal):  maxCacheSize=1000 std deviation = 100 numTests=1000 hit ratio=79%
[amit@kohinoor generic-cache-v2]:
```

5 Submission

Submit your program(s) on `onyx` FROM WITHIN your `program1` directory as shown below.

```
submit amit cs342 1
```