Design of Key-Value store Project

Assumption

We assume that the size of the metadata will not be more than 1M.

Design

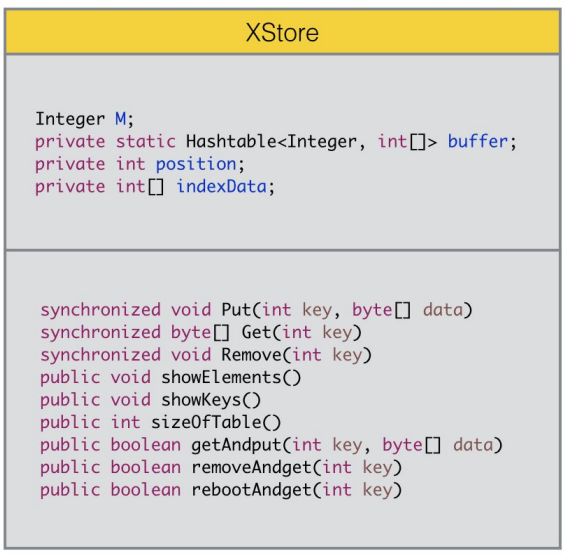
Here I will briefly introduce the design of our database. This project is combined with four java files called: XStore.java, Test#.java, StoreFileObject.java and StoreFiledata.java.

StoreFileObject.java and StoreFiledata.java
Provide the necessary functions to read and write the object and byte into files separately.

Test#.java
Includes the main functions and all the tests and necessary data we need to show our functions can work well.
Test1 for concurrency validation
Test2 for durability validation
Test3 for fragmentation

XStore.java
Contains all the other components of our database. I will give a detailed analysis below.

Here is the object diagram of XStore



The integer M is a constant refers to the size of megabyte. And the indexdata is a pair of integer to store the position and size of every block of data. Then the most important part is the hash table buffer which stores the indexdata and key and we use this buffer to index the actual data in the file. The last data element position is used to mark the position of file we should write if here comes a new data.

All the operations we have implemented is to maintain the buffer and position to make sure they record the right data according to the data in the actual data field.

Before I explain the methods in Xstore, I will explain how we manage the data and the free space of the database. At first we want to use two data structures to manage the data block and the free block separately and if we find here is no single block can store the data we will divide the data into several parts to store them into several smaller blocks if possible. But since we do not have enough time to make it perfect, we choose another easier way to do that. We store the data continuously and all the free space is just after the data field. Every time we remove the data, we will move all the data after the block we deleted backward to squeeze out the “bubble”. In this way, we can make sure all the data and the free space are continue separately.

Here is the explanation of the methods in XStore:

Put(int key, byte[] data)

This method achieves the API provided in the project description, it accepts the key and data given by the user and then put the data into the field and maintain the buffer and the position at the same time.

Get(int key)

Just like Put, we achieved this API as mentioned in the description of the project. We just pick out the data and then return it to the user.

Remove(int key)

Just like Put, we achieved this API as mentioned in the description of the project. Every time we delete the data, we will manage the free space to make sure that there is no bubble in the file.

All the other methods are used for testing including printing out the results in the console and multithread programming.

DB file:584.db is used to store metadata and real data.

1. Metadata is design to save control information of real data.
    a. Act as index when we search file and data.

b. Save data index in file to recover sa, in case the system is power off or reboot.

2. Real data is following the metadata in different block and has unique identify in Metadata

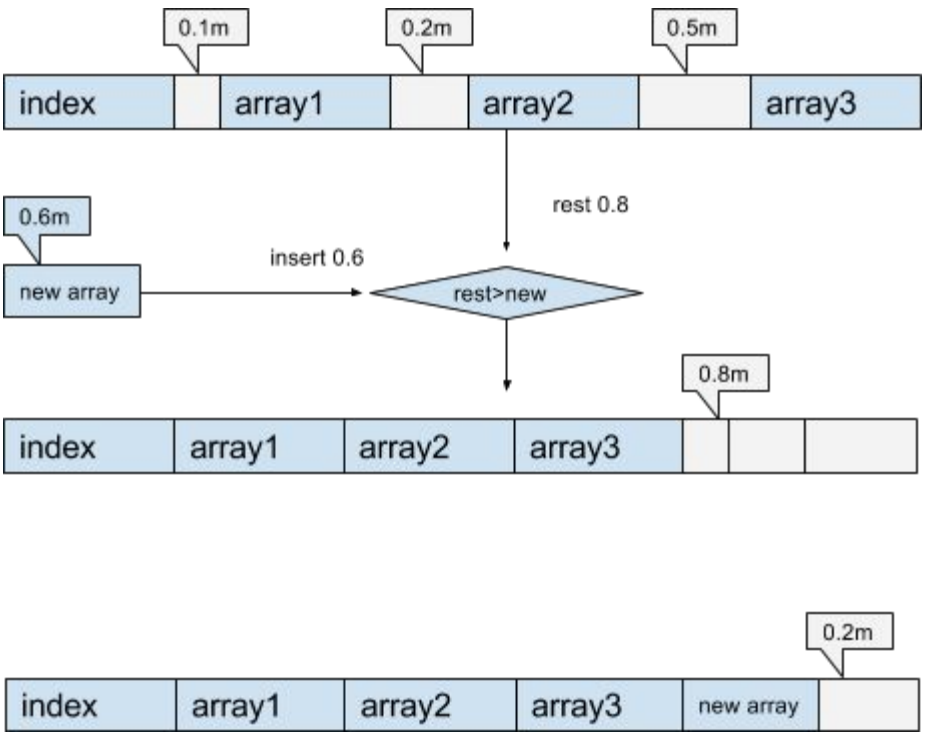
Interface

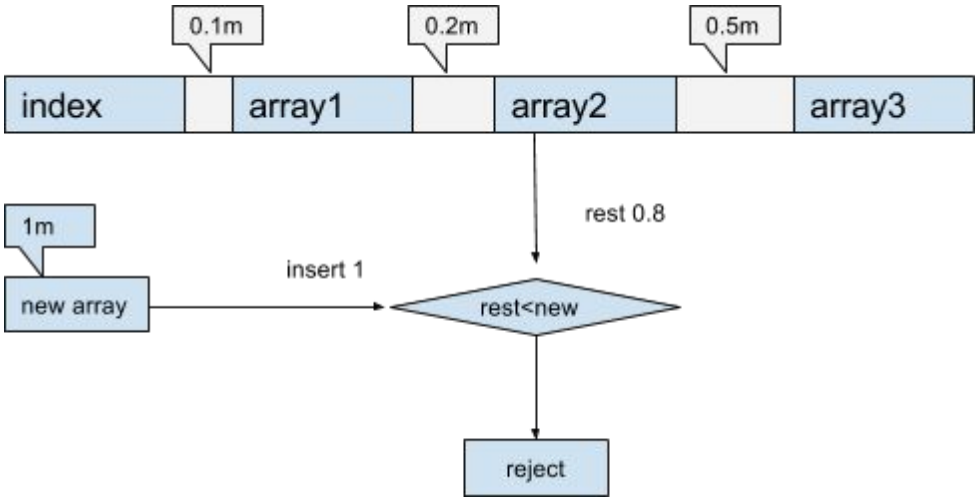
Interface	Function
void Put(int key, byte[] data)	stores data under the given key
byte[] Get(int key)	retrieves the data
void Remove(int key)	deletes the key

Method and Implementation

Fragmentation Method:

We separate the file as blocks, and each one is 1MB. When the data of one block is less than 1MB, the free space will be identified as free. One new array requests store in the space, the sum of free space is compared with the length of the new array. If new array.length is larger than the rest free space, request is rejected. If new array.length is smaller than rest free space, move forward all the existed data array and append new array at the end.





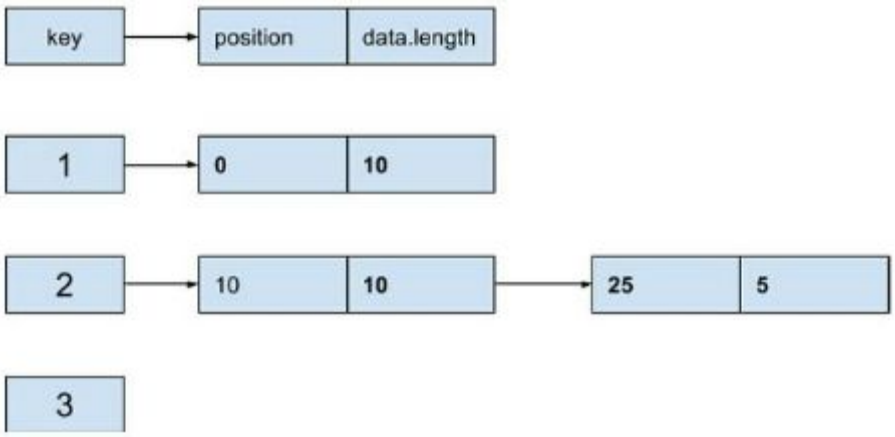
Data Structure

Hash table is used to implement an associative array, a structure that can map keys to values. It uses a hash function to compute an index into an array of buckets or slots, from which the desired value can be found

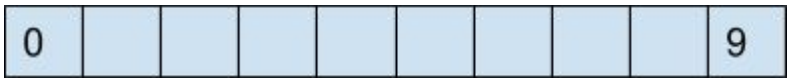
Data Structure for Metadata:

Metadata uses hashtable to store control information of data. Key stores the identify information of file, and Value stores position of data in the file and the file's length.

Here is the simple store details for Metadata.



Data Structure for Real Data:



Validation

Concurrency Validation:

Multiple users can read when the the information is put by others.

Test case of Concurrency:

The file Test1.java is used for the Durability Validation.
In the test case. There are 2 arrays. One is a array that all of the value is 1, another is all 2. Put() the array of all 1 with the key 1. Then use the putAndGet() method with the key 2 to validate the concurrency performance. There is a FileLock lock during the write process. So when the put() method write the bytes to the file. The Get() method can not get value from the file. So the return value of the Get() method should be the new value 2. As shown in the first half of the picture.

```
Here to show put and get at the same time:
GET METHOD:Here is the value under key 1
[2, 2, 2, 2, 2, 2, 2, 2, 2, 2]
Here to show Remove and get at the same time:
No value under this key.
```

Then we called the RemoveAndGet() method. The Remove() also included a write process. Therefore we can only get the value after the Remove(), so the return value of the Get() should be null.

Durability Validation

The file Test2.java is used for the Durability Validation.
The reboot is simulated by clear the metadata in the memory.

```
This is the keys of data in the cs542.db:
The key:4
The key:8
The key:7
The key:1
This is the position and size of data in the cs542.db:
[position,size]:[1000000, 1000000]
[position,size]:[3000000, 1000000]
[position,size]:[2000000, 1000000]
[position,size]:[0, 10]
Here to show reboot and get, here reboot means clear the megadata in the memory:
The store has been cleared.
Get key after the store have been cleared:
GET METHOD:Here is the value under key 1
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

Fragmentation Validation

The file Test.java is used for the Fragmentation Validation.

First Step:

Put() 4 byte arrays of 1MB, their key is 1,2,3,4. Call the showElements() and showKeys() method to show the information. Here is the result.

```
The key:4
The key:3
The key:2
The key:1
[position,size]:[3000000, 1000000]
[position,size]:[2000000, 1000000]
[position,size]:[1000000, 1000000]
[position,size]:[0, 1000000]
```

Second:

Remove() the key 2, then Put() a ½ MB array in key 5. Then Put() a 1 MB array in key 6. The Put(6) failed. Print “out of memory! “in the console.

```
out of memory!
The key:4
The key:3
The key:1
The key:5
[position,size]:[2000000, 1000000]
[position,size]:[1000000, 1000000]
[position,size]:[0, 1000000]
[position,size]:[3000000, 500000]
```

Third:

Remove() key 3 and Put() a 1MB array in key 7. We can find the key 7 have been put successfully and the key 3 have been removed.

```
The key:4
The key:7
The key:1
The key:5
[position,size]:[1000000, 1000000]
[position,size]:[2500000, 1000000]
[position,size]:[0, 1000000]
[position,size]:[2000000, 500000]
```

Fourth:

Remove() key 5 and Put() another 1 MB in key 8. Verify if the key 8 can be put into the file. We can find in the following picture that the key 8 have been put into the file successfully.

```
The key:4
The key:8
The key:7
The key:1
[position,size]:[1000000, 1000000]
[position,size]:[3000000, 1000000]
[position,size]:[2000000, 1000000]
[position,size]:[0, 1000000]
```