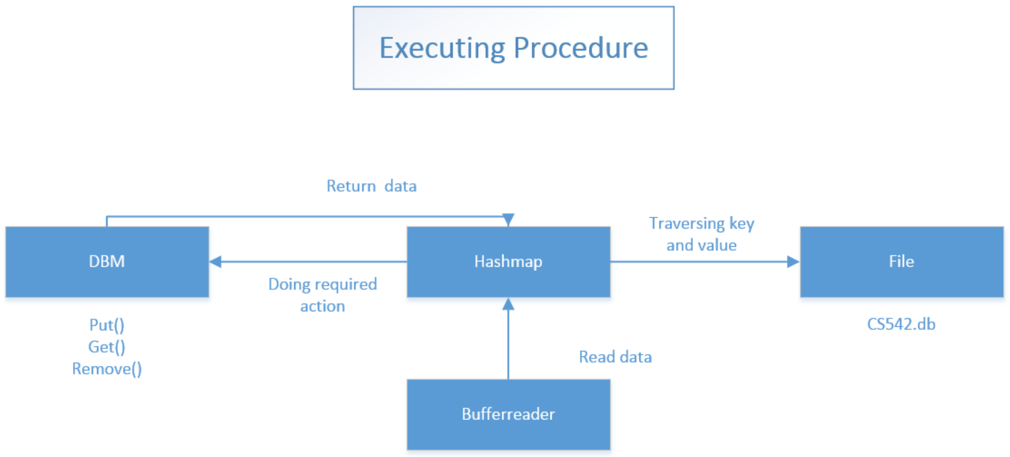
Project 1 Report

Jiang Miao and Fu Zhan

Part A

In our project design, in order to finish get, remove, put three main actions, we design that our core algorithm is hashmap, this is executing process of the whole program



1. void put(String Key, byte[] data)
In this interface firstly Calculate the metadata size(Key.length()) and data size. If the key is in the hashmap, remove the original data length of this key then plus the new data length of this key to get data length of whole file. If not, the new key data length and original data length is the data length. Also set max value length, key length and the whole file size.
writeFile() puts the key into the hashmap.
2. get(String Key)
readFile() reads the file created by createFile() in class DB_Test and in charge of getting the key from hashmap.
3. void remove(String Key)
readFile() reads the content of whole file and remove the assigned key.

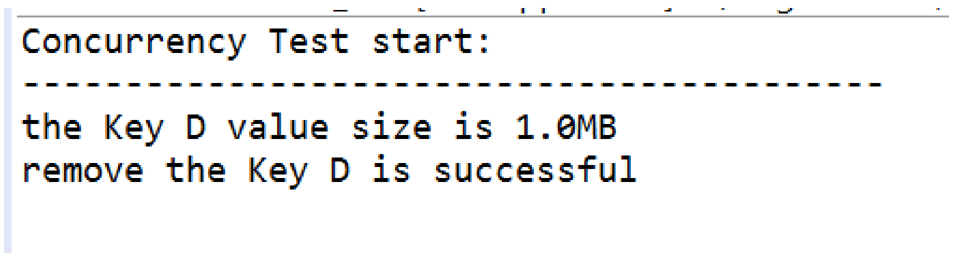
For processing multiple threads action, we use Read lock/unlock and Write lock/unlock to protect the data will be executed correctly when one user processes it. Through this method can definitely help us to process the same data at the same time, it avoids that two users will not be interrupted by each other. The performance of concurrency has been significantly improved.

Part B

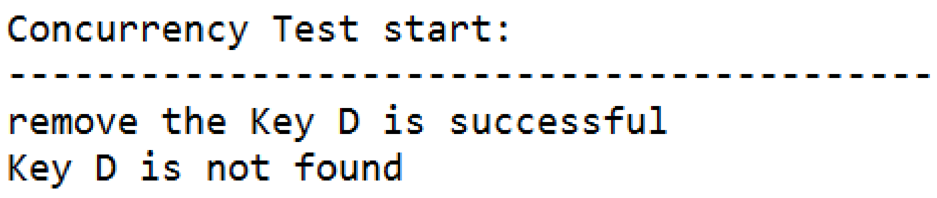
Class DB_Test includes several process of validation.
createFile() creates a file named cs542.db and sets four keys A,B,C and D with 1M value respectively.

1. Concurrency validation: Thread t1 is doing dbm.remove("D") and at the same time Thread t2 is doing dbm.get("D").
It says that these two processors have not been interrupted by each other, the remove action happens before getting value. The concurrency performance of our design is good.

There are two different conditions:
If D still exists in the file: We can get it value and remove it at the same time.


```
Concurrency Test start:
------------------------------------------
the Key D value size is 1.0MB
remove the Key D is successful
```

If D is not in the file: D is not found

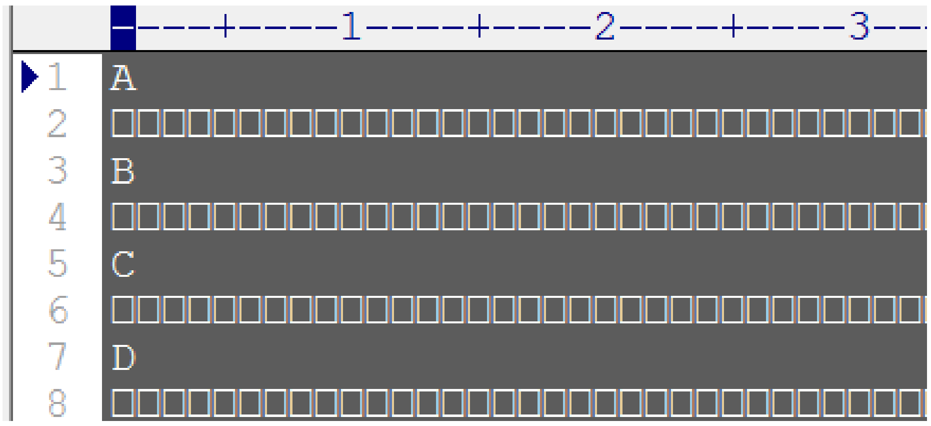

```
Concurrency Test start:
------------------------------------------
remove the Key D is successful
Key D is not found
```

2. Durability validation: the original data can be fetched by get() even though after reboot of the machine after the data has been put().
It's easy to validate for each time we want to operate data in different ways, we can find that the data structure will not be changed if we don't do these three actions (get, remove, put). The whole file is still 5MB, each data value of each key is 1MB.

3. Fragmentation: A, B, C and D have been set 1M when creating the file. From test3, key B is removed and key E (0.5M) is inserted. After that put() key F (1M) fails because there is no enough room. And then remove C and put() 1M in size for key G succeeds. Finally E is removed and H can be put successfully according to our design. The resulting is showing as following:

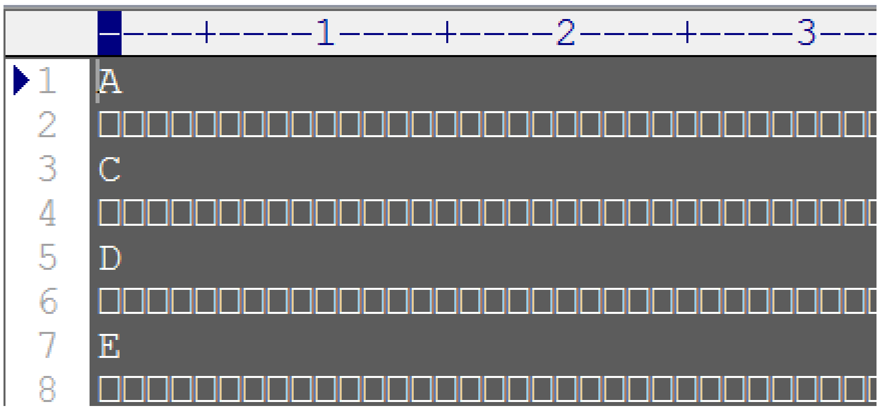
```
Fragmentation Test start:
------------------------------------------
test1
the Key B value size is 1.0MB
test2
Your value length is larger than 1MB
test3
remove the Key B is successful
putting the Key E with new size 0.5MB is successful
the data size/meta data size is too large
Fail to put the Key F
test4
remove the Key C is successful
putting the Key G with new size 1.0MB is successful
test5
remove the Key E is successful
putting the Key H with new size 1.0MB is successful
the Key H value size is 1.0MB
------------------------------------------
```

This is the beginning file structure:



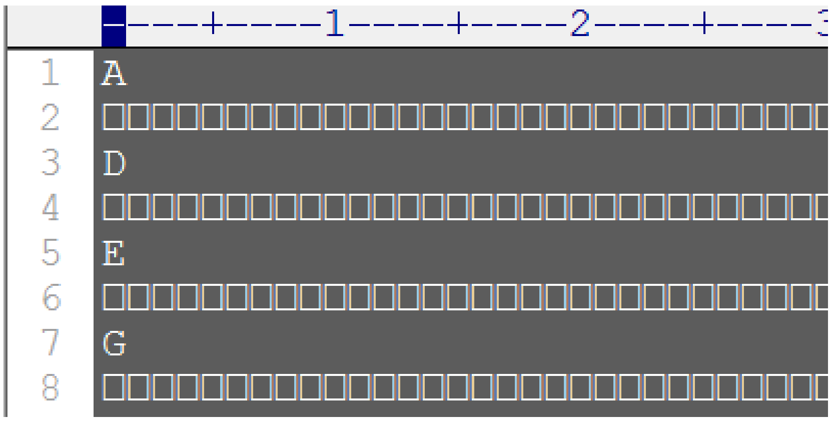
```
----+----1----+----2----+----3---
A
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
B
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
C
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
D
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
```

```
test3
remove the Key B is successful
putting the Key E with new size 0.5MB is successful
the data size/meta data size is too large
Fail to put the Key F
```



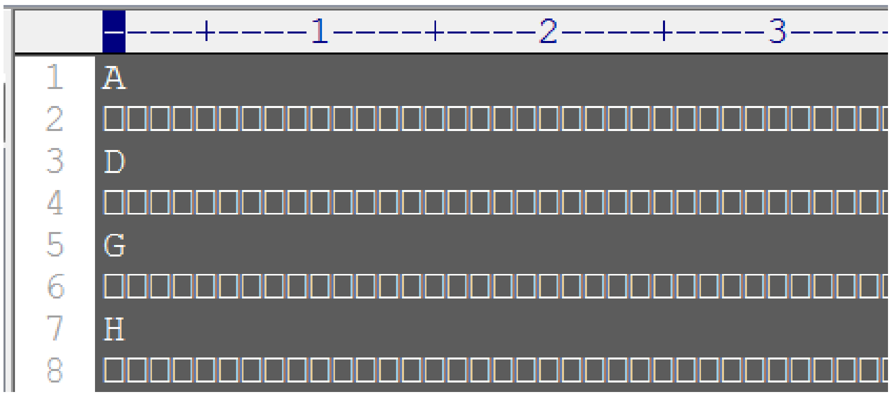
```
A
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
C
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
D
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
E
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
```

```
------------------------------------------
test4
remove the Key C is successful
putting the Key G with new size 1.0MB is successful
```



```
A
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
D
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
E
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
G
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
```

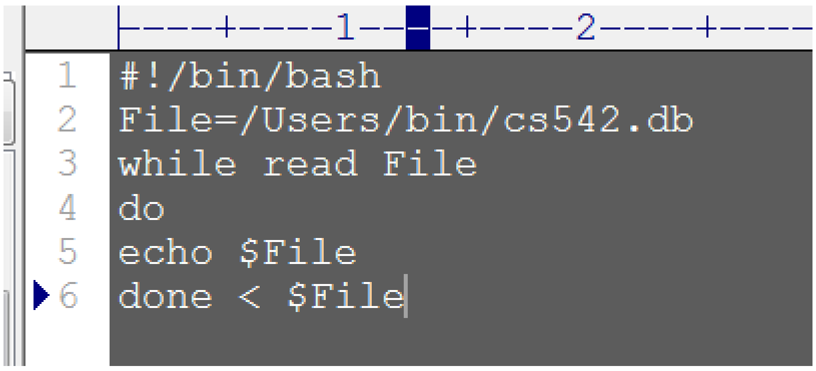
```
test5
remove the Key E is successful
putting the Key H with new size 1.0MB is successful
the Key H value size is 1.0MB
```



```
A
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
D
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
G
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
H
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
```

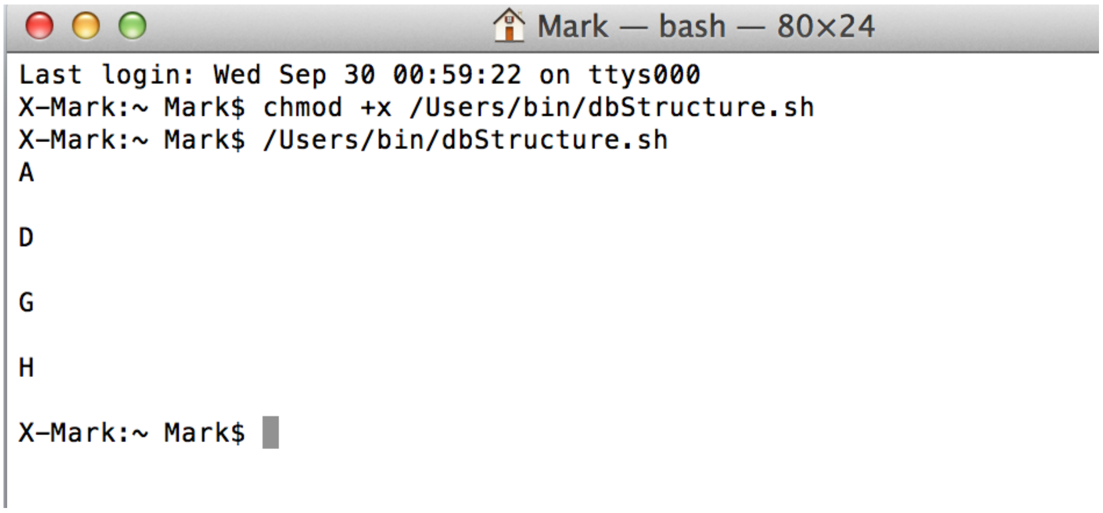
Part C

To validate the data structure of our file is right, we write the shell program in Linux:



```
#!/bin/bash
File=/Users/bin/cs542.db
while read File
do
echo $File
done < $File
```

This is the executing result:



```
Last login: Wed Sep 30 00:59:22 on ttys000
X-Mark:~ Mark$ chmod +x /Users/bin/dbStructure.sh
X-Mark:~ Mark$ /Users/bin/dbStructure.sh
A

D

G

H

X-Mark:~ Mark$ 
```

It’s the same as the last test results in Part B, which can prove out design of whole program has no problem and all of tests are successful.