CS542 PROJECT ONE

Xiaoyan Sun & Shi Wang

1. Data & Algorithm Design

(1) Data storage

Category	Content	Data Type
Database	Value	byte[ ]
Metadata	Key; Length of the Value; Address	Hashmap<Integer, int [ ]>

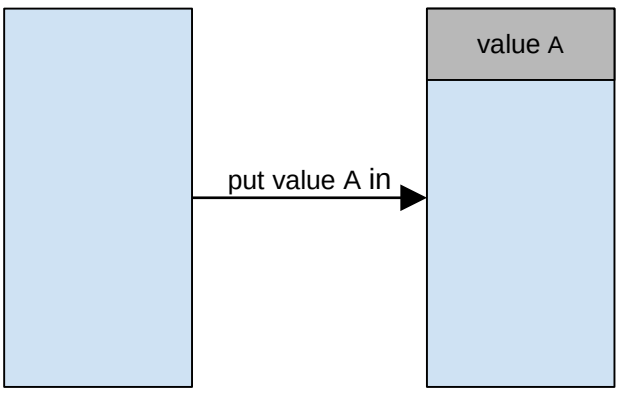
(2) Data Management

We save the data in two parts. The actual data part is for saving the value and the data address. The metadata part is for saving the key of the value, the length of the value and the address of the value. By finding the key in Metadata, we could get the address to find the value in actual data.

(3) Algorithm (logical diagram)

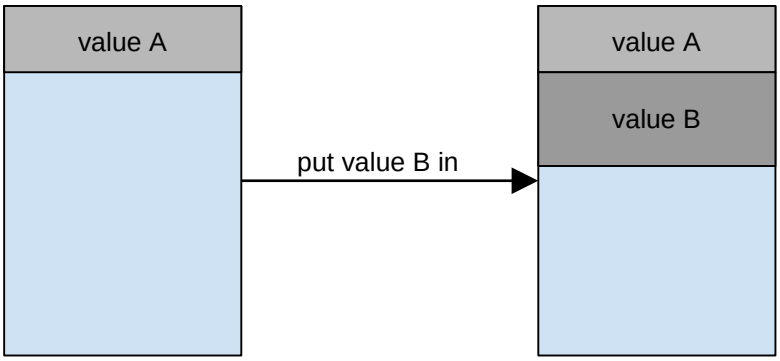
(a) Put a value in to the database:

- The Database is empty
  When the database is empty, put the value from the start of the actual data storage.

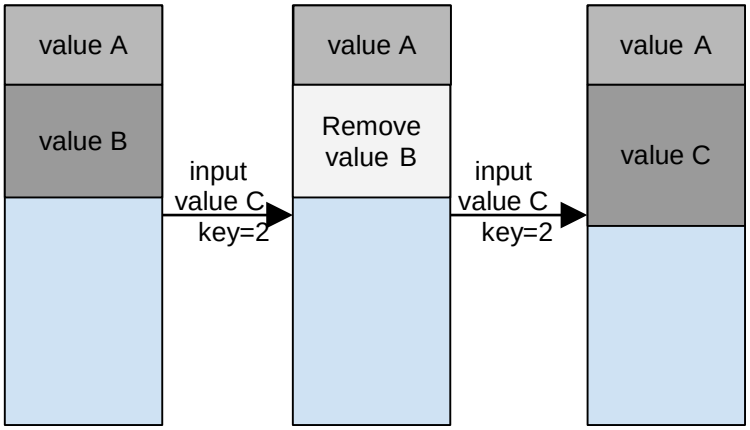


- The Database is not empty(key is not exist)
  When the database is not empty and the key of the value not exists, put the value into the storage by searching empty blocks. Then put the value

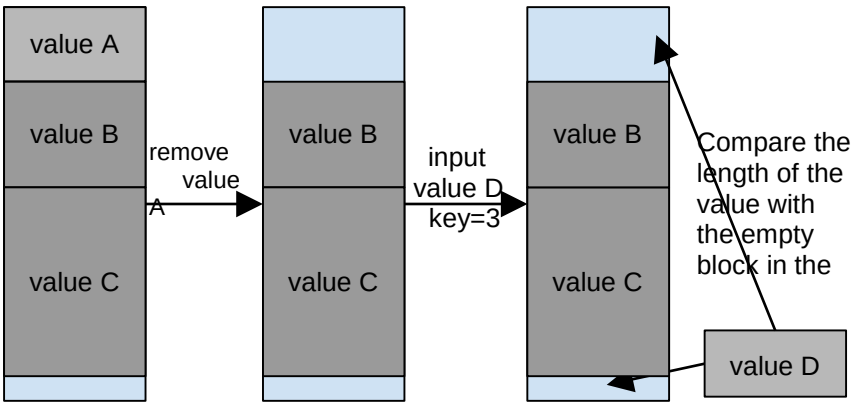
sequentially.



- The Database is not empty(the key exists)
  When the value of the key exists in the database, first remove the value with the same key, then put the new value in.

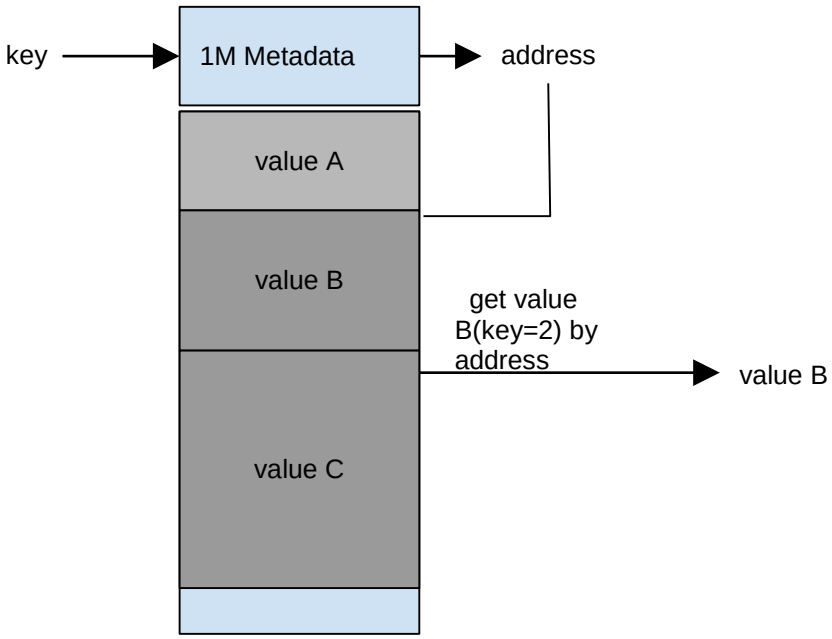


- Put value after remove



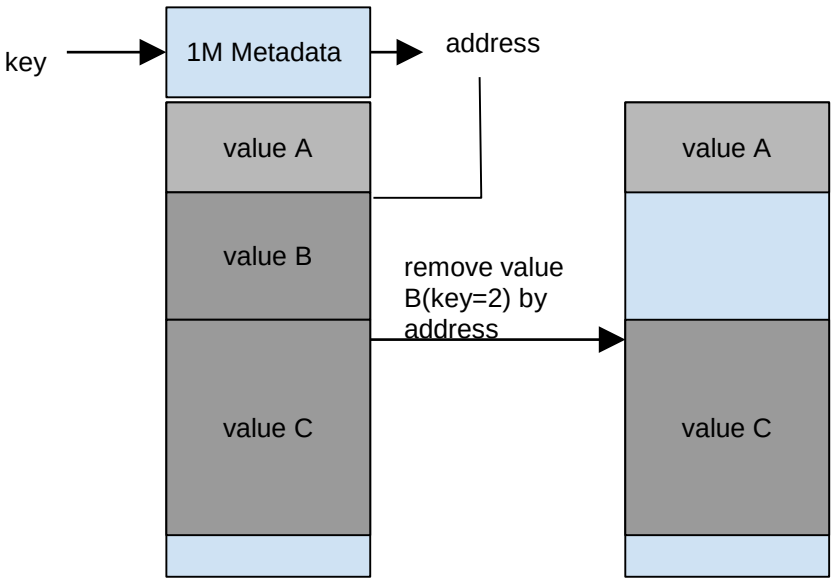
(b) Get a value from the database

When getting a value from database, search for the value's address by the key of the value in metadata, then using the address to find the value in actual data.



(c) Remove a value from database

When removing a value from database, using the key of the value to search for its address. Then using the address to find the value and remove it.



2. Implementation

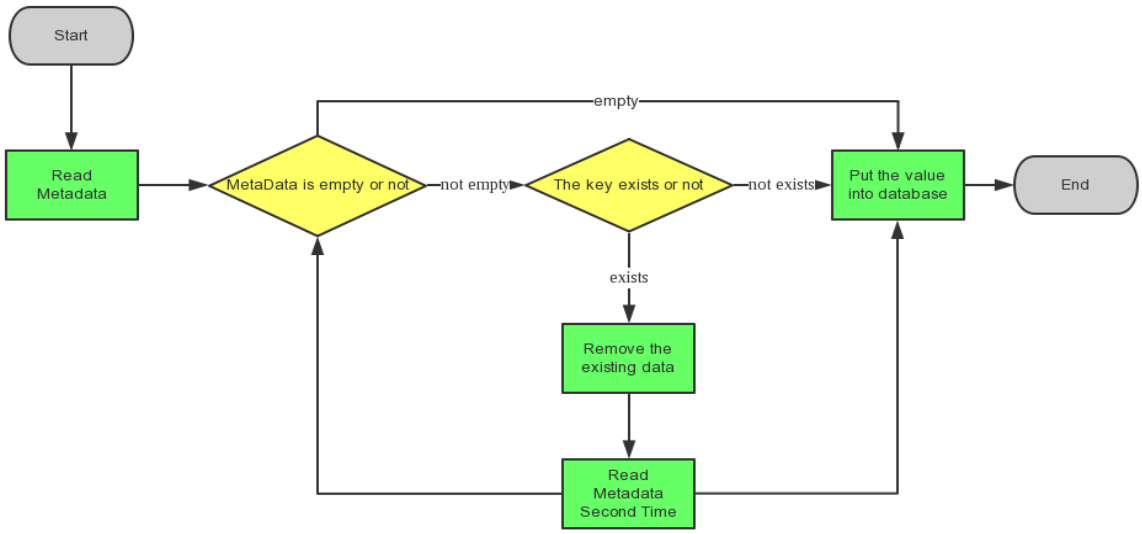
The public methods: Put(), Get(), and Remove() are provided as interface

The private methods put(), get(), and remove() are implemented as the following,

(1) put(int key, byte[] value)

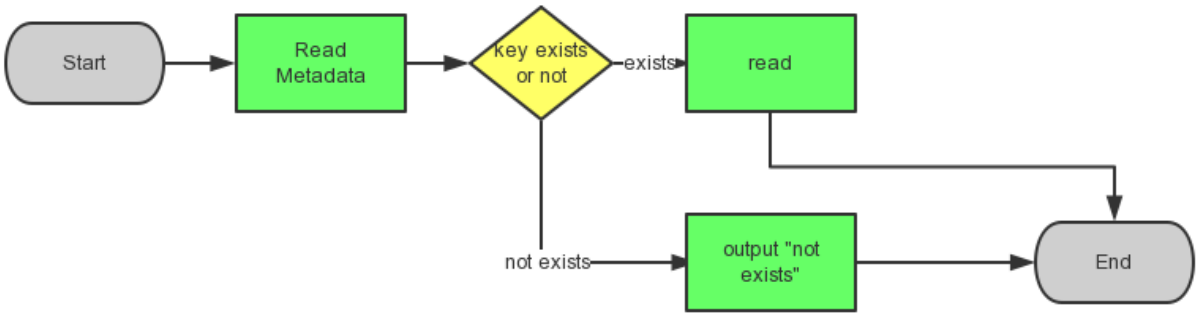
The main algorithm we use to put in data is by searching for available space in the DB. We use the information from meta data, which are key, length and address, to construct two arrays that contains the starting points and ending points of all existing data. After sorting the arrays and subtracting them, we are able to get information about the length and starting addresses of all available pieces of space. If the inputting value is not larger than the first available space found, we insert the value there.

This algorithm requires sorting and subtracting arrays of addresses every time a value need to be inserted into the database. Sorting algorithm is not considered in this implementation. Other data structures for data addresses will be considered in the future.



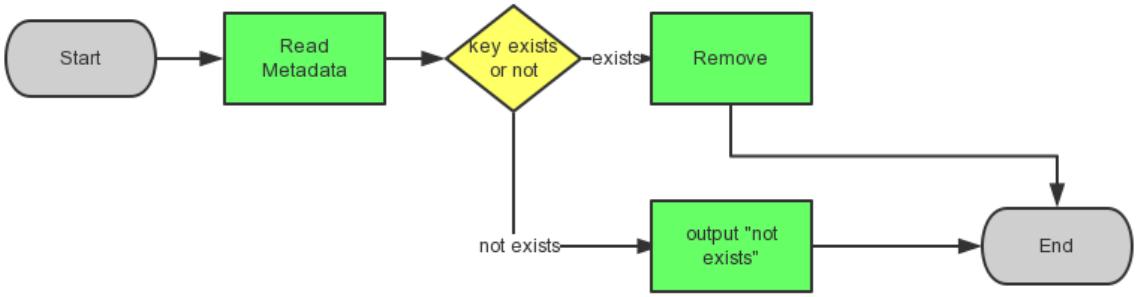
(2) get(int key)

To get a certain value by using its key, we check first whether this key exists in meta data. If not, messages would be sent out informing that no such data exists in the database. Otherwise, if the key was found in meta data, we go find the value at the address provided by the meta data.



(3) remove(int key)

If key does not exist in database, system output not existed. If key found in meta data, it will be deleted from meta data together with the data's length and address. No actual operation would be done on database, however, the space that contains the data will be seen as available for future inputs, and also, a Get() will not be able to fetch the data.



3. Tests and Results

(1) Concurrency Validation

Method:

Using two threads to test the methods worked on database. To implement threads, we used runnable interface.

Process:

- put() and get()
  Thread one: put a value into the database
  Thread two: get a value from the database with the same key.
- get() and remove
  Thread one: remove a value from the database

Thread two: sleep for milliseconds, then get the value which has the same key with the removed value.

Result:

- When the put() function works in thread one and get() function works in thread two the first time, the get() function could only get the original data in the database but not the new data put() function sent.
  Then the second time, let thread two sleep milliseconds. After the sleeping, thread two uses get() function again to read the value with the same key. Then thread two can get the recent value renewed by put() in thread one.
- Thread two cannot get the value removed by thread one.

```
*****Thread 1 and thread 2 are ready to start.
*****Thread 1 and thread 2 runs at the same time
*****Thread 2 first got the original value in database
*****Thread 1 put a new value into the database
*****Thread 2 got the new value after 1000 sleep
*****Concurrency Test 1 over.
Thread[Thread-1,5,main]---Thread 2 is ready to get data 1.
---Thread 2 is sleeping for time 1000
Thread[Thread-0,5,main]---Thread 1 is ready to put data 1.
---Thread 1 is sleeping for time 1000
---Thread 2 is getting value from Database which has a key=1
---The value thread 2 got is
1 1 1 1 1 1 1 1 1 1 ---Thread 1 is sleeping for time 1000
Key 1 existed! Now replacing it with new value.
Deleted 1 in DB..
Successfully put 1 into db...
---Thread 1 is putting (1,A) into Database which has a key=1 and value=A[i]=1
Thread[Thread-0,5,main]---The data 1 has been put.
---The value thread 2 got is
2 2 2 2 2 2 2 2 2 2

*****Thread 1 and thread 2 are ready to start.
*****Thread 1 and thread 2 runs at the same time
*****Thread 1 first remove the value with a key=1
*****Thread 2 gets null after 1000 sleep
Thread[Thread-1,5,main]---Thread 2 is ready to get data 1.
Thread[Thread-0,5,main]---Thread 1 is ready to remove data 1.
Key 1 existed! Now replacing it with new value.
Deleted 1 in DB..
---Thread 2 is sleeping for time 1000
Successfully put 1 into db...
Deleted 1 in DB..
---Thread 1 is removing Data which has a key=1
Thread[Thread-0,5,main]---The data 1 has been removed.
Cannot get data: key 1 not existed.
---Thread 2 is getting value from Database which has a key=1
---The data not exists
```

(2) Durability Validation

Method:

Since both meta data and actual data are written back to file after each call the Put(), Get(), and Remove() methods, these data are secure on disk during machine shut down. We simulate the durability by clearing out objects that are used at running time, and trying read in data again to see whether data was lost.

Process:

- put in a data
- clear out objects
- get the data that has the same key
- display the data

Results:

Data is successfully read in and value remain the same

```
---Put data 111 into DB...
Key 111 existed! Now replacing it with new value.
Deleted 111 in DB..
Successfully put 111 into db..
---Machine restarted...
---Get 111 from DB...and display the data..
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

(3) <u>Fragmentation</u>

Method:

Put in and remove multiple data to see if data would be stored at the expected position. Our algorithm for data storage limits the ability to store fragmented data, since data is always put into database in a continuous manner. With proper improvement, this could be done in the future.

Results:

This is an approximate graphic result of the status of the 0M to 1M space during the process, generated by our test program, indicating where data are at, and where is available.

```
---Put data 1 (1M) into DB:
Successfully put 1 into db..
---Put data 2 (1M) into DB:
Successfully put 2 into db..
---Put data 3 (1M) into DB:
Successfully put 3 into db..
---Put data 4 (1M) into DB:
Successfully put 4 into db..

---Remove data 2 (1M) from DB:
Deleted 2 in DB..
---Put data 5 (0.5M) into DB:
Successfully put 5 into db...

---Display the layout in DB. *: has data; -: no data
---Data exists at 1M to 3.5M and 4M to 5M
----------------------0M
------------------------
------------------------
------------------------
------------------------
------------------------
------------------------
------------------------
************************
************************
************************
************************
************************
************************
************************
************************
************************
************************
************************
************************
------------------------
------------------------
------------------------
------------------------
************************
************************
************************
************************
************************
************************
************************
************************
************************
************************
************************
************************
************************
************************
************************
************************
```

```
---Put data 6 (1M) into DB: (fail expected)
Put value 6 to database fail: not enough space.
---Remove data 3 (1M) from DB:
Deleted 3 in DB..
---Put data 7 (1M) into DB: (success expected)
Successfully put 7 into db...
---Remove data 5 (0.5M) from DB:
Deleted 5 in DB..
---Put data 8 (1M) into DB:
Put value 8 to database fail: not enough space.

---Display the layout in DB. *: has data; -: no data
---Data exists at 1M to 2M , 2.5M-3.5M, and 4M to 5M
----------------------0M
-------------------------
-------------------------
-------------------------
-------------------------
-------------------------
-------------------------
-------------------------
*************************
*************************
*************************
*************************
*************************
*************************
*************************
*************************
-------------------------
-------------------------
-------------------------
-------------------------
*************************
*************************
*************************
*************************
*************************
*************************
*************************
*************************
-------------------------
-------------------------
-------------------------
-------------------------
*************************
*************************
*************************
*************************
*************************
*************************
*************************
*************************
```

The results of the test are the same as expected. The database manager we designed is able to search for available space that is large enough for the data. The last value was not able to be put in, although there is 1M space left in the database, which is enough for a 1M value. This is due to the fact, as mentioned before, that data has to be stored in a continuous way in this design, and the remaining space are two pieces of 0.5M.