

**EEM 480**  
**ALGORITHMS & COMPLEXITY**  
**HOMEWORK 4 REPORT**

**Name:** Yunus Emre

**Surname:** ESEN

**Student No:** 22280328940

**Instructor:** Assist. Prof. Dr. Emin GERMEN

## 1. PURPOSE

This Project has been created for EEM 480 homework 4. The subject of that is design a Spotify-like environment and program using hash structure. It works with command lines from given input (txt file).

## 2. THE ALGORITHM

The program has two hash structures that are contains person names and song names. Every hash cell contains hash object which has key and an object that is going to be a block which has person name and song list or song name and name list according to hash type. There are 2 figures to explain the idea below.

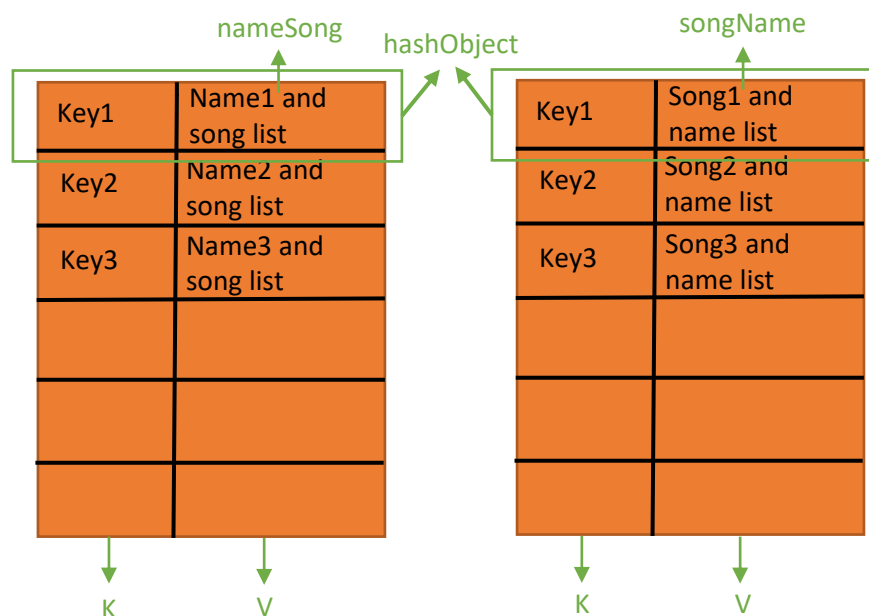


Figure 1 and Figure 2: Representation of thought hash algorithm, name hash and song hash, respectively

Hash structure is created with an array. This array can store two object type. K, represents key, and V, represents value, are part of hash object which specifies every line of hash. nameSong and songName are other blocks to keep data like name, song list and song, name list. Linked list structure is used in song and name lists. Figure 3 represents these structures.

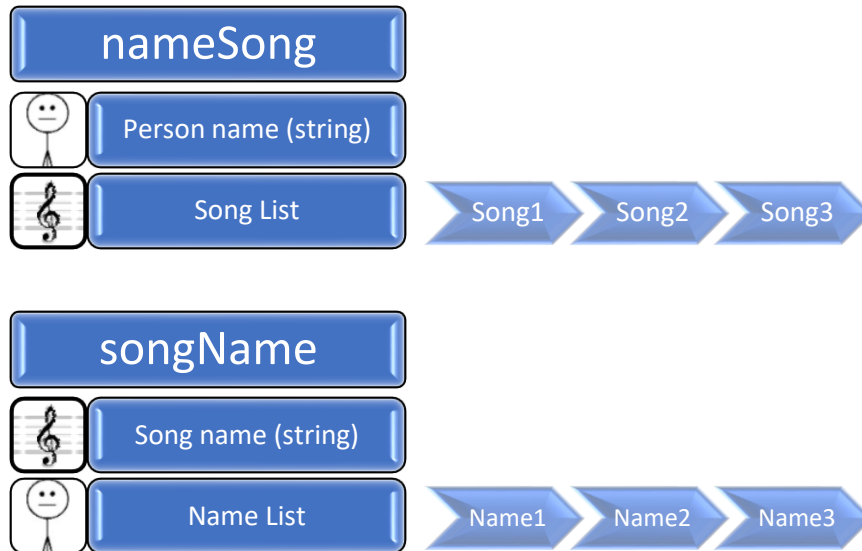


Figure 3: nameSong and songName representations

The key value, which is mentioned before K, specifies to where will be the name stores in array. A number was assigned to each alphabet letter (a=2, b=3, ..., y=27, z=28, A=29, B=30, ..., Z=53) to calculate the key value. To find the index in the hash table, the corresponding values of the first and second letters of the entered name in the alphabet conversion are multiplied by each other and taking mod size of array.

```
index = (alphabet[name[0]] * alphabet[name[1]]) % SIZE;
```

If that index is not available, then new index is found by:

```
index = ((multiply*53) + (index%53)) % SIZE;
```

Size value is defined on head of the code. It manages memory allocation and if it is entered too many logs, size must be increase. Size of given code is adjusted to 10000 to avoid size problem.

To get a value from hash, index is found again by the same algorithm as above. Find algorithm checks whether the value in that index is wanted value or not. If it is not, index is recalculated. There is an example view from the NetBeans for this algorithm.

[64]	hashObject	#295
K	Integer	64
value	int	64
V	nameSong	#310
nameOfBlock	String	"Ali"
songBlock	Song	#312
SongName	String	"Show must go on"
next	Song	#315
SongName	String	"Mambo italiano"
next		null

Figure 4: View of table [64] in NetBeans

### 3. SOLUTION

NetBeans IDE 8.0.2 was used to compile this project. The codes are written in Java. This report will not include the codes that is used because of it is too long. But it will be explaining the variables, methods and how they are work.

Seven classes have been used for realizing this project. These files are described below.

#### Person.java

This java file defines node of a person with a name and next person information.

#### Song.java

This java file defines node of a song with a name and next song information.

#### hashObject.java

It contains two objects named K and V. K is used for key, V is used for value. Key describes an index in the hash to call or put a value to V.

#### nameSong.java

This java file is used like a block which is like a person. It contains a person name by String type parameter and a Song object which defines which songs are liked from that person. Its structure is like in Figure 3.

#### songName.java

This java file is used like a block that contains song information. This block has a song name by String type parameter and a Person object which defines who is liked this song. Its structure is like in Figure 3.

#### HashTable.java

This java file contains two hash structures. These are aimed to store names and songs in different parts. Two arrays are defined to create hash structure. SIZE shows the size of the arrays to be generated. Name hash is in the `table[]` array, song hash is in the `songTable[]` array. The other array `personNames[]` store names who are in name hash. The required values for multiplication used to find the index and create hash structure are taken from `alphabet[]`. The algorithm is told in the previous page.

There are 8 main methods and 12 assistant methods in this java file. They were used nested. Assistant methods are described below briefly.

```
Object get(Integer index)_____ Returns table[index]
Object getNameSongBlock(Integer index)_____ Returns hashObject of table[index]
Object getSongNameBlock(Integer songIndex)_____ Returns hashObject of songTable[songIndex]
String getSongName(Integer songIndex)_____ Returns song name in songTable[songIndex]
String getPersonName(Integer index)_____ Returns person name in table[index]
Object getSongBlock(Integer index)_____ Returns nameSong in table[index]
Object getPersonBlock(Integer songIndex)_____ Returns songName in songTable[songIndex]
boolean isCreated(String personName)_____ Returns true if personName is in table[index],
                                           else false
boolean isSongCreated(String songName)_____ Returns true if songName is in
                                           songTable[songIndex], else false
Integer findIndex(String personName)_____ Returns an integer which is the index of the
                                           personName in the table[]
Integer findSongIndex(String songName)_____ Returns an integer which is the index of the
                                           songName in the songTable[]
Integer getCharIndex(char a)_____ Returns an integer corresponding to alphabet to
                                           number conversion for the hash process
```

Main methods are described below.

**putName(String personName) :**

This method puts `personName` to the name hash (`table[]`) if name is not created before. To do that, it creates new `hashObject`, find `index` value as told before, creates `nameSong` and combines all of them in `hashObject` with name, and puts it in `table[index]`. A name with one letter cannot be created. It is a contradiction with algorithm.

**likeSong(String personName, String songName) :**

If `personName` is created, it is added liked songs to song block of `nameSong`, which is a linked list. At the same time, if it is not in the hash table, it creates a new hash line to add song hash (`songTable[]`) and adds `personName` to `songName`'s name list (which is also linked list). If it is in the hash table, it just adds `personName` to the linked list. It does same things for adding processes as mentioned about algorithm part.

**eraseSong(String personName, String songName) :**

It erases `songName` from the list of `personName` in name hash table and `personName` from the linked list of song hash table.

**deletePerson(String personName) :**

It deletes the `personName` from the name hash table. Creates a new empty `hashObject` to replace old object. Delete process is not converting the object to the null. There will be an error if the deleting object will null.

**printSongs(String personName) :**

It prints whole songs of `personName` liked.

**match(String personName) :**

It does a matchmaking operation between `personName` and other people in the name hash table, gives a percentage of the matched songs between names. To make it real, it is created two array named `count[]` and `total[]`. It takes `personNames` songs and compare these with the names taking from `personNames[]` array. `personNames[]` just gives the name created before. A while loop compares all songs with `personName`, another while loop changes song and for loop repeats it processes unless `personNames[]` has another name. This method is not match with himself/herself.

**recommend(String personName) :**

This method contains a copy of match operation in first part. The reason of that is taking the values of percentage. Recommendation operation happens depends on percentage rate. The method recommends songs from the most matched person, if printed songs are deficient then it will be take another person who has the closer percentage of matched songs. It works until print 5 songs to the output.

Sometimes, it may be cannot prints 5 songs. Because, the most matched person with `personName` may have not 5 different songs. There is an algorithm to solve this, but if this person is first to matched, then it blocks other people to compare process. Unfortunately, there is not found any solution because of the keeping complexity  $O(n)$ .

**O(String FileName) :**

It works with an input text file. Input file needs to contain values for create name (I), like song (L), erase song (doesn't like) (E), delete name (D), matchmaking (M), recommend

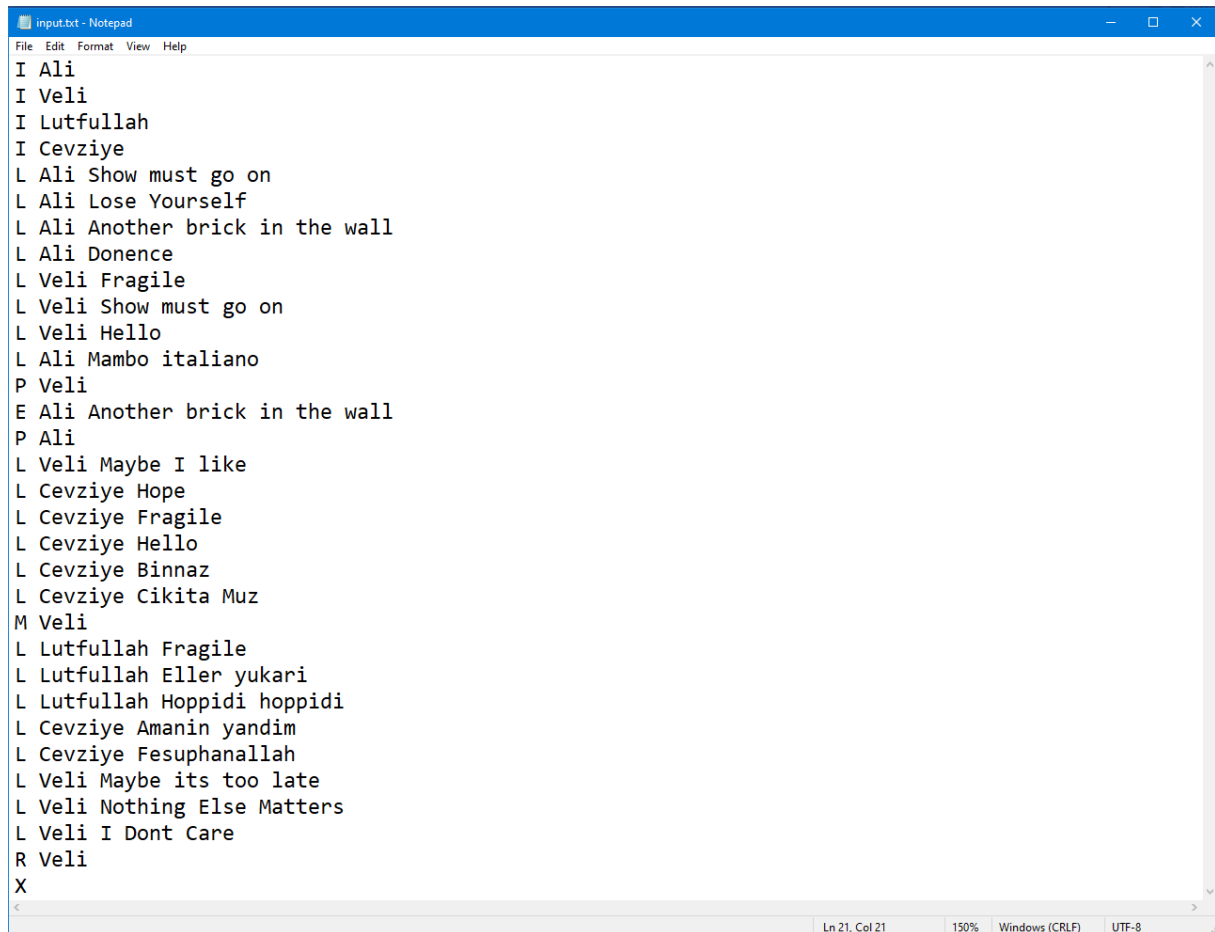
songs (R), give an input file(recursive, run another text file) (O) and exit (X). These letters are the commands to using this program. It also has an exception to if something goes wrong.

#### HW4.java

This is the main java file. It creates a new `HashTable` object named `newHash` and takes an input with `O` method.

```
HashTable newHash = new HashTable();  
newHash.O("input.txt");
```

The input contains the command lines as in Figure 5 below.



```
File Edit Format View Help  
I Ali  
I Veli  
I Lutfullah  
I Cevziye  
L Ali Show must go on  
L Ali Lose Yourself  
L Ali Another brick in the wall  
L Ali Donence  
L Veli Fragile  
L Veli Show must go on  
L Veli Hello  
L Ali Mambo italiano  
P Veli  
E Ali Another brick in the wall  
P Ali  
L Veli Maybe I like  
L Cevziye Hope  
L Cevziye Fragile  
L Cevziye Hello  
L Cevziye Binnaz  
L Cevziye Cikita Muz  
M Veli  
L Lutfullah Fragile  
L Lutfullah Eller yukari  
L Lutfullah Hoppidi hoppidi  
L Cevziye Amanin yandim  
L Cevziye Fesuphanallah  
L Veli Maybe its too late  
L Veli Nothing Else Matters  
L Veli I Dont Care  
R Veli  
X
```

Figure 5: Example input, input.txt

Results are found by the using methods described above sections. I, L, E, D, P commands are tried to find results in  $O(1)$  complexity. M and R commands are tried to making matchmaking and recommendation processes in  $O(n)$  complexity. After the input in Figure 5, the output will be shown below.

run:

Veli liked:

Fragile

Show must go on

Hello

Ali doesn't like Another brick in the wall

Ali liked:

Show must go on

Lose Yourself

Dönence

Mambo italiano

Possible friend of Veli

Ali 25% match (1 song out of 4)

Lutfullah 0% match (0 song out of 0)

Cevziye 40% match (2 song out of 5)

Recommended songs for Veli:

Eller yukari

Hoppidi hoppidi

Lose Yourself

Dönence

Mambo italiano

BUILD SUCCESSFUL (total time: 0 seconds)

*Thank you for all things that you done in the whole semester.*