**Student Record using hash table:**

The data structure is implemented using bucket array with separate chaining mechanism to store the elements. The key to the hash table is the student Id which will identify the bucket array index and a linked list in at each index of the bucket array to store the elements falling in that bucket. The implementation is such that any operation on the hash table by passing the student Id (eg: get(studentId)) should return in close to O(1) time complexity i.e. there should be as minimum collision as possible in the bucket array.

**HashId()**

1. **HashCode function:**

The key to be used is the student id which is of YYYYAAADDDD format. We tried the following hash functions for different input sizes.

1. Polynomial hash function (with the polynomial constant of 33) to convert AAA to an integer and add this integer with YYYY and the roll number DDDD.
2. Shift-Add-XOR hash -

**for** (**int** i = 0; i < stud\_id.length(); i++)

hashCode ^= (hashCode << 5) + (hashCode >> 2) + stud\_id.charAt(i);

Each character in the Id is taken and hascode is computed using the above formula

1. Bernstein hash – Same as above except the fomula for hashcode used is different

**for** (**int** i = 0; i < stud\_id.length(); i++)

hashCode = 33 \* hashCode ^ key.charAt(i);

We see that **Brenstein hash function** gives the most efficient distribution. For inputs of variable sizes 8K records, 16k records and 36k records we see that around 65% of the elements were retrieved in exactly O(1) time i.e. among the filled buckets in the bucket array, 65% of them had exactly one element. 90% of the filled buckets had exactly 1 or 2 elements. Maximum number of elements in any filled bucket array was not more than 5 for all the three inputs.

For the same 8k, 16k and 36k records input, the **Shift-Add-XOR hash function** was closer to Bernstein hash. 60% of the filled buckets had exactly one element and 85% of the filled buckets had exactly 1 or 2 elements. Maximum number of elements in any filled bucket array was not more than 5 for all the three inputs.

In the **polynomial hash function** approach, only 20% of the filled buckets had exactly one element. Rest of the 80% of the filled buckets had more than 1 element. There were buckets having up to 11 elements in a single bucket.

Based on above results, we have used the Brenstein hash function.

1. **Compression Map:**

We use the h(k) mod N formula to convert the hash code h(k) to one of the index in the bucket array of size N. We initialize the hash table with bucket array of size 31. This prime number ensures that the modulo returns unique numbers and avoids collision. We use the load factor as 0.75. Whenever the bucket array is 75% full, we choose the prime number that is close to the double of the existing size. Choosing a prime number for modulo redistributes the elements more effectively than just doubling. Once we determine the new table size, we redistribute the entries of the hash table across the buckets of new size.

**Input Validations:**

1. The input file should have student id and the cgpa separated by delimiter “,”
2. Student Id should be of YYYYAAADDDD format where
   1. YYYY – year from 2008 to 2018
   2. AAA – Department code should be one of the CSC / ARC / ECE / MEC
   3. DDDD – 4 digit roll number in integer padded with 0 i.e if the roll number is 1 then this should be 0001.
3. The cgpa should be float number
4. Any record / line not conforming to 1, 2 and 3 above will not be considered / added to the hash table

**Functions:** Since the key is the student id and most of the requested functions deal with the values (cgpa) of the hash table we need to iterate through all the elements in the hash table. As per dictionary ADT, iterating through the elements of the hash table is of O(n) complexity. If any operation was requested via studentId, then complexity would be of O(1). The implementation sticks to the definitions of the dictionary of the ADT like iterating through the values and keys is via iterator, rehashing when the number of elements goes beyond the load factor.

**initializeHash**: Initializes the hastable with bucket array of size 31. 31 (prime number) is taken so that the compression map will use the h(k) mod 31 which is likely to avoid as much collision as possible.

**populateHashTable**: This reads the input file from the location input/Input.txt which is in the class path. This input/Input.txt is placed under resources folder which is already a part of classpath as per the J2EE specifications. Each record is read from the input file using buffered reader and they are split as per the delimiter comma (,). If a record is not delimited by comma delimiter we ignore that record i.e record is not put in the hash table. The student id and the cgpa are validated as per the input validations specified in the above section and if it is valid, the record is put in the hash table. Else the record is ignored and the ignored records are printed in the console.

While adding the record in the hash table we do couple of things

1. **Duplicate check:** We check if the input student\_id already exists in the hash table. This is done in O(1) complexity as the implementation of hash function avoids as much collision as possible. If student\_id already exists, we override the cgpa with the input record. This steps ensures that a student’s record is available only once in the hash table
2. **Rehashing**. After adding the record in the hash table, we check if the bucket array size had to be increased to avoid collision. For this we check if [Number of elements in hashtable] / [Total bucket array size] > 0.75 (load factor) and if it is, then we increase the size to a higher prime number that is closer to double of existing size and rehash the elements as per the new bucket array size.

**hallOfFame**: Here we find all the students whose cgpa is greater than the input cgpa. For iterating through the elements we use an iterator. The iterator, iterates through the bucket array and finds if an index has a linked list associated to it. If it is, then it will iterate through the linked list elements and returns the student record when next() is called. Once all the elements in the list of a particular index are iterated, it will proceed to the next index in the bucket array which has linked list associated to it. This way all the elements in the hash table are visited and those whose cgpa is greater than input cgpa are printed to the hallOfFame.txt. The location of this file is governed by the system property java.io.tmpdir (which is defaulted to user\_home/AppData/Local/Temp/ ). The complexity is O(n) as we are iterating through all the elements.

**newCourseList**: We use the same iterator used in the hallOfFame to iterate through the records and find if the year part of the studentId is within the 5 year and the corresponding cgpa is within the input cgpa. If it is then the record is printed in the corseOffer.txt. The location of this file is governed by the system property java.io.tmpdir (which is defaulted to user\_home/AppData/Local/Temp/ ). The complexity is O(n) as we are iterating through all the elements.

**depAvg**: We use the same iterator used in the hallOfFame to iterate through the records and find the department of the student from the studentId. For each department we maintain a DeptInfo which has the maxCgpa , total cgpa and total Students. For each record in the iterator we determine the department and set the following in the corresponding DeptInfo

1. Set DeptInfo.maxCgpa to current iterator element’s cgpa if DeptInfo.maxCgpa is less than the current iterator element’s cgpa
2. add the current iterator element’s Cgpa to the DeptInfo totalCgpa variable and
3. increment the total students by 1

From the DeptInfo, we print the maxCgpa and average (totalCgpa/totalStudents) to departmentAverage.txt. Complexity is O(n) as we are iterating through all the elements.

**destroyHash**: Points the bucket array to null and reset the counter variables.

**generateInput**: This generates the input file. For each record it randomly picks one dept from CSC / ECE / MEC / ARC and year from 2007 to 2019 and cgpa from 0 to 9.9.