Data Engineering
Lab 4&5

Team Member	Roll number
Mukul Shingwani	B20AI023
Sawan Patel	B20CS063
Jaimin Sanjay Gajjar	B20AI014

Q1A)
The primary key for uniquely identifying the queries will be the composite key of BookId and AuthorID.
Composite Key = {BookID, AuthorId} = Primary Key

Reason →
We cannot uniquely identify the rows using just BookID or AuthorID since there can be multiple books written by the same author and vice-versa. Maybe two authors jointly write a book

Q1)B)
We can concatenate the keys together, and then take the sum of their ASCII value (ignoring the underscore) which will give us an integer hash value generated from the keys. The minimum value of h can be 1034 and the max value 1676.

```
#include<iostream>
using namespace std;

int hash_value(string book_id, string author_id){
    int n1 = book_id.size() - 1;  //12
    int n2 = author_id.size() - 1;  //10

    int h = 0;
    for (int i=0; i<=n1; i++){
        if (book_id[i] == '_'){
            continue;
        }
        h += book_id[i];
```

```
    }

    for (int i=0; i<=n2; i++){
        if (author_id[i] == '_'){
            continue;
        }
        h += author_id[i];
    }

    return h; //1034 min value , 1676 max value of h
}

int main()
{
    string book_id = "Aest_AC_0103";
    string author_id = "An_Ch_0103";
    int h = hash_value(book_id, author_id);
    cout << h;
    return 0;
}
```

Q1)C)
No, they are not adequate since many authors can have the same starting initials, which can result in probably the same ID, say,(Anjan Chatterjee and Anita Chaturvedi → so An_ch_<4digitcode> but this will fail in book_id generation)also say there is a part 2 of a book (Deathly Hallows pt.2) that will result in same BookID because they involve book initials followed by the author id initials and 4 digit code

---

Q2)A) Code in C++ submitted

Q2)B) We experimented with 2 values of bucket size other than 4

- Case 1: **size = 3**
    - This caused an increase in time of extendible hashing from 394 microseconds to 516 microseconds since the number of buckets was reduced to a deficient number, the number of collisions increased, and hence rehashing (more bits were considred) was increased.

- Case 2: **size = 7**
    - This took 153 microseconds as compared to 394 in bucket size 4, since rehashing was less and data settled in more properly and appropriately.

Q2)C) Instead of using a dynamic array **(vectors)**, which is a linear data structure, we used the **Set** data structure, which is coded using a **Red-Black Tree** internally in the STL and so is nonlinear.
Both of the data structures were equally efficient maybe since the database was of a smaller size, but In bigger databases using Set base, implementation might be more beneficial since when rehashing happens, linear structures are really expensive in terms of both time and space. Also depending on the use case if we want records of the database in sorted order then trees have that function naturally(if implemented using BST) but in linear structures that will require additional efforts.

Q3)A) The **global bucket size chosen was 5**, since its a prime number so it will not have any divisor other than 1 and 5, Moreover, there are only 15 entries in the database, and the given local bucket, size is 4, so 5 seemed like an even distribution choice along with being prime.

Q3)B) Done in C++ (Code submitted)

Q3)C) We experimented with 2 values of global bucket

- Case 1: **size = 3**
    - This caused an increase in time of linear hashing from almost negligible (0 microseconds) to 500 microseconds since the number of global buckets was reduced to a deficient number, the number of collisions increased, and hence rehashing was increased.
- Case 2: **size = 7**
    - This also took negligible time i.e 0 microseconds, same as in the case of bucket size as 5.

so maybe 5 is the most appropriate choice of global bucket size was this dataset since it requires minimal to almost no rehashing and beautifully accommodates the complete database.

Q3)D) In this case we kept a constant global bucket size (5) and varied the local bucket size from 4 to two cases to understand better

- Case 1: **Size = 3**
    - Again, this caused an increase in time of linear hashing from almost negligible (0 microseconds) to 543 microseconds, since the number of local buckets was reduced to a deficient number, the number of collisions increased, and hence rehashing was increased.
- Case 2: **Size = 5**
    - This also took negligible time i.e 0 microseconds, same as in the case of bucket size of 5, since maybe the combination of local bucket size of 4 and global bucket size of 5 was good enough to handle this database.

Q3)E) Instead of using a dynamic array **(vectors)**, which is a linear data structure, we used the **Set** data structure, which is coded using a **Red-Black Tree** internally in the STL and so is nonlinear.
Both of the data structures were equally efficient maybe since the database was of a smaller size, but In bigger databases using Set base, implementation might be more beneficial since when rehashing happens, linear structures are really expensive in terms of both time and space. Also depending on the use case if we want records of the database in sorted order then trees have that function naturally(if implemented using BST) but in linear structures that will require additional efforts.

Q4)A) We took the higher positioned bits as 5 since it seemed like an appropriate depth to grow the tree, neither too shallow nor too deep, and since the dataset also contains just 15 tuples as of now with hash values ranging from 1034 and the max value 1676, choosing 5 seemed a reasonable choice.

Q4)B) code in C++(submitted)

Q4)C) We experimented with 2 values of global bucket

- Case 1: **size = 3**
    - This caused an increase in time of distributed hashing from almost negligible (0 microseconds) to 429 microseconds since the number of global buckets was reduced to a deficient number, the number of collisions increased, and the depth of tree was too shallow to accommodate MSB's in its nodes so linear component may have been traversed thus increasing the time.
- Case 2: **size = 7**
    - This also took negligible time i.e 0 microseconds, same as in the case of bucket size as 5. But it can surely have more memory wastage if the numbers are not in the higher ranges

Q4)D) In this case we kept a constant global bucket size (5) and varied the local bucket size from 4 to two cases to understand better

- Case 1: **Size = 3**
    - This caused an increase in time of distributed hashing from almost negligible (0 microseconds) to 517 microseconds since the number of local buckets was reduced to a deficient number, the number of collisions increased, and hence rehashing was increased and tree's depth would also have been impacted.
- Case 2: **Size = 5**
    - This also took negligible time i.e 0 microseconds, same as in the case of bucket size of 5, since maybe the combination of local bucket size of 4 and global bucket size of 5 was good enough to handle this database.

Q4)E) Instead of using a dynamic array **(vectors)**, which is a linear data structure, we used the **Set** data structure, which is coded using a **Red-Black Tree** internally in the STL and so is nonlinear.
Both of the data structures were equally efficient maybe since the database was of a smaller size, but In bigger databases using a Set base, the implementation might be more beneficial since when rehashing happens, linear structures are really expensive in terms of both time and space. Also depending on the use case

if we want records of the database in sorted order then trees have that function naturally(if implemented using BST) but in linear structures that will require additional efforts.

—----------

Q5A)

- Extensible Hashing
    - Advantages
        - A Bucket will have more than one pointer pointing to it if its local depth is less than the global depth.
        - Data retrieval is less expensive (in terms of computing).
        - With dynamic changes in hashing function, associated old values are rehashed w.r.t the new hash function.
    - Disadvantages
        - Size of every bucket is fixed
        - Memory is wasted in pointers when the global depth and local depth difference becomes drastic.

- Linear Hashing
    - Advantages
        - Requires less memory, simple and easy to code
        - Insertion is efficient when not bucket overflow
    - Disadvantages
        - Clustering can happen if the hash function is not appropriate, thus wasting memory
        - In case of rehashing O(n^2) time is roughly required

- Distributed Hashing
    - Advantages
        - Uses the concept of the tree to store the initial MSB's thus is time and space efficient
    - Disadvantages

- If tree depth is not appropriate it may give bad results since it can be
    - too shallow to accommodate larger numbers (in this case buckets will store most of the bits, thus increasing time of retrieval).
    - if too deep in case of small numbers then memory wastage will happen.

For all bucket sizes, Extensible Hashing produces consistently better storage utilization than linear hashing. Linear hashing gives cyclic storage utilization since the buckets are split linearly regardless of their load.
In both Extensible Hashing and Liner hashing, as the bucket size rises, the storage utilization becomes more fluctuating. One more observation is that Linear hashing requires more buckets to hold the same number of records than External hashing does.

The successful search and the unsuccessful search are equally costly in extenible and linear hashing.
The splitting of a bucket is costlier in Linear hashing. This is due to the fact that an extra read access is needed to read the bucket to be split.
The insertion cost is slightly higher in Extendible hashing, since a conversion to binary is required then selection of bits is there to put it into right bin.

Q5)B) Comparing time of insertion (in microseconds)

Mechanism	Extendible	Linear	Distributed
Time taken	246	333	68

Q5)C) Comparing time of 1 record retrieval (in microseconds)

Mechanism	Extendible	Linear	Distributed
Time taken	49	8	6

Q5)D) Comparing time of all records retrieval (in microseconds)

Mechanism	Extendible	Linear	Distributed
Time taken	7	16	6

We can clearly see that **Distributed Hashing** took least time hence was the **best** hashing mechanism out of the lot for this database atleast. Since, it has a sort of tree based approach linked to it, which helps in suitable management of the bits starting from MSB, it'a average case time is better than both the other algorithms and rehashing is the most expensive thing in other two, hence it performs the best.

—---------- END ----------