DBMS Lab 4, 5

Maniya Yash Rajeshbhai (B20CS033)
Tatvam (B20CS077)
Ruthvik K. (B20AI037)

Author_ID	Book_ID	Author_Name	Book
An_Ch_0103	Aest_AC_0103	Anjan Chatterjee	The Aesthetic Brain
An_Da_0104	Self_AD_0104	Antonio Damasio	Self Comes to Mind
Ca_Sa_0319	Anim_CS_0319	Carl Safina	What Animals Think
Jo_Ro_1018	Deat_JR_1018	Joanne K. Rowling	Deathly Hallows_Harry Potter
Jo_Ro_1018	Fant_JR_1018	Joanne K. Rowling	Fantastic Beasts and Where to Find Them
Jo_Ro_1018	Gobl_JR_1018	Joanne K. Rowling	Goblet of Fire_Harry Potter
Jo_Ro_1018	Phil_JR_1018	Joanne K. Rowling	Philosopher's Stone_Harry Potter
Jo_Ro_1018	Pris_JR_1018	Joanne K. Rowling	Prisoner of Azkaban_Harry Potter
La_Ch_1203	Mind_LC_1203	Lars Chittka	The Mind of a Bee
Ma_Mi_1313	Emot_MM_1313	Marvin Minsky	Emotion Machine
Ma_Mi_1313	Soci_MM_1313	Marvin Minsky	Society of Mind
Pe_Wo_1623	Aunt_PW_1623	Pelham G. Wodehouse	Aunts Aren't Gentlemen
Pe_Wo_1623	Wode_PW_1623	Pelham G. Wodehouse	Wodehouse at the Wicket
Vi_Ra_2218	Emer_VR_2218	Vilayanur Ramachandran	The Emerging Mind
Vi_Ra_2218	Phan_VR_2218	Vilayanur Ramachandran	Phantoms in the Brain

Q 1.

a. Choose a primary key for the given table. Reasons for doing so? [5]

   Primary key - {Book_id + Author_id}

   As it can uniquely determine all the other attributes in the table.

   Book_id alone isn't enough since there may be a case where there are multiple authors who wrote a book collaboratively. In such a case Book_id would be an inappropriate primary key.

b. Design a hash function for what you choose as the primary key for the given table. Use the evaluated hash values to work on the following questions [5]

   Following code represents our hash function in c++ :

```cpp
int ascii_val(string s){
    int ans=0;
    for(auto i:s){
        ans+=(int)i;
    }
    return ans;
}

int hash_function(Row R){
    string a_id = R.Author_ID;
    string b_id = R.Book_ID;
    int ans = 0;
    ans+=ascii_val(a_id);
    ans+=ascii_val(b_id.substr(0,4));
    return ans;
}
```

   Here we pass the complete Row in out hash function and the hash function uses the primary key i.e., the book_id and author_id to calculate the hash value which is further being used in all the 3 types of hashing done in question number 2,3 & 4.

c. Comment on the provided codes for book_id and author_id. Do you think these are sufficiently effective? [5]

Yes, Book_ID, Author_ID combined are sufficiently effective to determine all the other attributes. Since using Book_ID we can get the book name and using the Author_ID we can get the author name.

Q 2.

b. Changing the bucket size in the extendible hashing will surely have an effect on the effectiveness of the hashing. But, one can not surely say if increasing or decreasing the bucket size will decrease the execution time or not. It largely depends on the data one need to store. Keeping the bucket size will lead to more number of changes in the global bucket size of the hashing and so it will lead to increase in execution time while, increasing the bucket size can convert the search operation into nearly linear search and hence may increase the execution time of the operations.

Case I: Bucket Size = 4

```
// };

int main(){
    // Hashing database;
    // R.Author_ID = "An_Ch_0103";
    // R.Author_Name = "Anjan Chatterjee";
    // R.Book = "The Aesthetic Brain";

    // R.Book_ID = "Aest_AC_0103";
    // cout<<"yee"<<endl;
    int bucketSize;
    cout<<"Insert the Bucket Size : ";
    cin>>bucketSize;
    bucket_size = bucketSize;
    bmap[0] = new Bucket;
    bmap[1] = new Bucket;
    // insert(R);

    Row R1("An_Ch_0103","Aest_AC_0103","Anjan Chatterjee","The Aesthetic Brain");
    Row R2("An_Da_0104 ","Self_AD_0104","Antonio Damasio","Self Comes to Mind");
```

OUTPUT PROBLEMS DEBUG CONSOLE TERMINAL JUPYTER

```
PS C:\Users\shah_\OneDrive\Desktop\LABS\DBMS\LAB-4&5> cd "c:\Users\shah_\OneDrive\Desktop\LABS\DBMS\LAB-4&5\" ; if ($?) { g++ q2.cpp -o q2 } ; if ($?) { .\q2 }
Insert the Bucket Size : 4
time taken by the insert queries to get executed is 0.000993 sec
1
PS C:\Users\shah_\OneDrive\Desktop\LABS\DBMS\LAB-4&5>
```

Case 2 : Increasing the Bucket Size to 6

```
// };

int main(){
    // Hashing database;
    // R.Author_ID = "An_Ch_0103";
    // R.Author_Name = "Anjan Chatterjee";
    // R.Book = "The Aesthetic Brain";

    // R.Book_ID = "Aest_AC_0103";
    // cout<<"yee"<<endl;
    int bucketSize;
    cout<<"Insert the Bucket Size : ";
    cin>>bucketSize;
    bucket_size = bucketSize;
    bmap[0] = new Bucket;
    bmap[1] = new Bucket;
    // insert(R);

    Row R1("An_Ch_0103","Aest_AC_0103","Anjan Chatterjee","The Aesthetic Brain");
    Row R2("An_Da_0104 ","Self_AD_0104","Antonio Damasio","Self Comes to Mind");
```

OUTPUT PROBLEMS DEBUG CONSOLE TERMINAL JUPYTER

```
PS C:\Users\shah_\OneDrive\Desktop\LABS\DBMS\LAB-4&5> cd

> cd "c:\Users\shah_\OneDrive\Desktop\LABS\DBMS\LAB-4&5\" ; if ($?) { g++ q2.cpp -o q2 } ; if ($?) { .\q2 }
Insert the Bucket Size : 6
time taken by the insert queries to get executed is 0.000997 sec
1
PS C:\Users\shah_\OneDrive\Desktop\LABS\DBMS\LAB-4&5>
```

Case 3 : Decreasing the Bucket Size to 2

c. We could use an unordered map instead of linear list. The map data structure uses the Red-Black Trees to store the data.

Case 1 : Using map and Bucket size as 4

Case 2 : Using map and Bucket Size as 6

Case 3 : Using map and Bucket Size as 2

Q3 :

A. We chose Global Bucket Order (n) of 7. Since there are 15 entries, on average if two entries get in the same bucket, roughly 7 buckets will be needed to store almost all the entries.
   If we would have chosen a lesser (n) value, some buckets may overflow and we need to extend our hash table with greater (n) value.
   Else if we chose a greater order, greater space will be required

B. Implemented Linear Hashing with global bucket order (n) of 7 and bucket size of 4, which can be extended further if any buckets are full.

```
Inserted all the entries

0       =>      Phil_JR_1018,

1       =>      Self_AD_0104 , Wode_PW_1623,

2       =>      Phan_VR_2218,

3       =>      Anim_CS_0319, Fant_JR_1018, Pris_JR_1018, Aunt_PW_1623,

4       =>      Soci_MM_1313, Emot_MM_1313, Emer_VR_2218,

5       =>      Aest_AC_0103, Gobl_JR_1018,

6       =>      Mind_LC_1203, Deat_JR_1018,


Metrics for Insertion queries :

Initial Global Depth (n) : 7
Bucket size : 4
Time taken : 4.277 ms
```

C. Results on varying (n) and keeping bucket size constant - 4 :

```
Inserted all the entries

0       =>      Phil_JR_1018,

1       =>      Self_AD_0104 , Wode_PW_1623,

2       =>      Phan_VR_2218,

3       =>      Anim_CS_0319, Fant_JR_1018, Pris_JR_1018, Aunt_PW_1623,

4       =>      Soci_MM_1313, Emot_MM_1313, Emer_VR_2218,

5       =>      Aest_AC_0103, Gobl_JR_1018,

6       =>      Mind_LC_1203, Deat_JR_1018,


Metrics for Insertion queries :

Initial Global Depth (n) : 7
Bucket size : 4
Time taken : 4.277 ms
```

Case 1 : Decreasing (n) to 3 :

```
Inserted all the entries

0       =>      Gobl_JR_1018, Wode_PW_1623,

1       =>      Emot_MM_1313,

2       =>      Pris_JR_1018,

3       =>      Aest_AC_0103,

4       =>

5       =>      Fant_JR_1018,

6       =>      Deat_JR_1018, Soci_MM_1313, Phan_VR_2218,

7       =>

8       =>      Emer_VR_2218,

9       =>      Phil_JR_1018, Aunt_PW_1623,

10      =>      Anim_CS_0319, Self_AD_0104 ,

11      =>      Mind_LC_1203,


Metrics for Insertion queries :

Initial Global Depth (n) : 3
Bucket size : 4
Time taken : 7.266 ms
```

Case 2 : Increasing (n) to 11 :

```
Inserted all the entries

0       =>      Self_AD_0104 ,

1       =>      Gobl_JR_1018, Aunt_PW_1623,

2       =>      Aest_AC_0103,

3       =>      Emot_MM_1313, Wode_PW_1623,

4       =>

5       =>      Pris_JR_1018,

6       =>      Deat_JR_1018, Fant_JR_1018,

7       =>      Soci_MM_1313,

8       =>      Phan_VR_2218,

9       =>

10      =>      Mind_LC_1203, Anim_CS_0319, Phil_JR_1018, Emer_VR_2218,


Metrics for Insertion queries :

Initial Global Depth (n) : 11
Bucket size : 4
Time taken : 9.266 ms
```

We see that taking small (n) values can result in inefficient queries. Because while inserting values, the buckets might get full due to less (n) value. And hence we need to extend its value by doubling it.
Also quite large (n) would increase execution time for insert query. As more buckets would be created.

D. Results on varying bucket size and keeping (n) constant - 7 :

```
Inserted all the entries

0       =>      Phil_JR_1018,

1       =>      Self_AD_0104 , Wode_PW_1623,

2       =>      Phan_VR_2218,

3       =>      Anim_CS_0319, Fant_JR_1018, Pris_JR_1018, Aunt_PW_1623,

4       =>      Soci_MM_1313, Emot_MM_1313, Emer_VR_2218,

5       =>      Aest_AC_0103, Gobl_JR_1018,

6       =>      Mind_LC_1203, Deat_JR_1018,


Metrics for Insertion queries :

Initial Global Depth (n) : 7
Bucket size : 4
Time taken : 4.277 ms
```

Case 1 : Decreasing the Bucket Size to 2 :

```
Inserted all the entries

0       =>

1       =>

2       =>      Phan_VR_2218,

3       =>      Fant_JR_1018, Aunt_PW_1623,

4       =>      Soci_MM_1313, Emer_VR_2218,

5       =>      Aest_AC_0103,

6       =>      Deat_JR_1018,

7       =>      Phil_JR_1018,

8       =>      Self_AD_0104 , Wode_PW_1623,

9       =>

10      =>      Anim_CS_0319, Pris_JR_1018,

11      =>      Emot_MM_1313,

12      =>      Gobl_JR_1018,

13      =>      Mind_LC_1203,


Metrics for Insertion queries :

Initial Global Depth (n) : 7
Bucket size : 2
Time taken : 8.621 ms
```

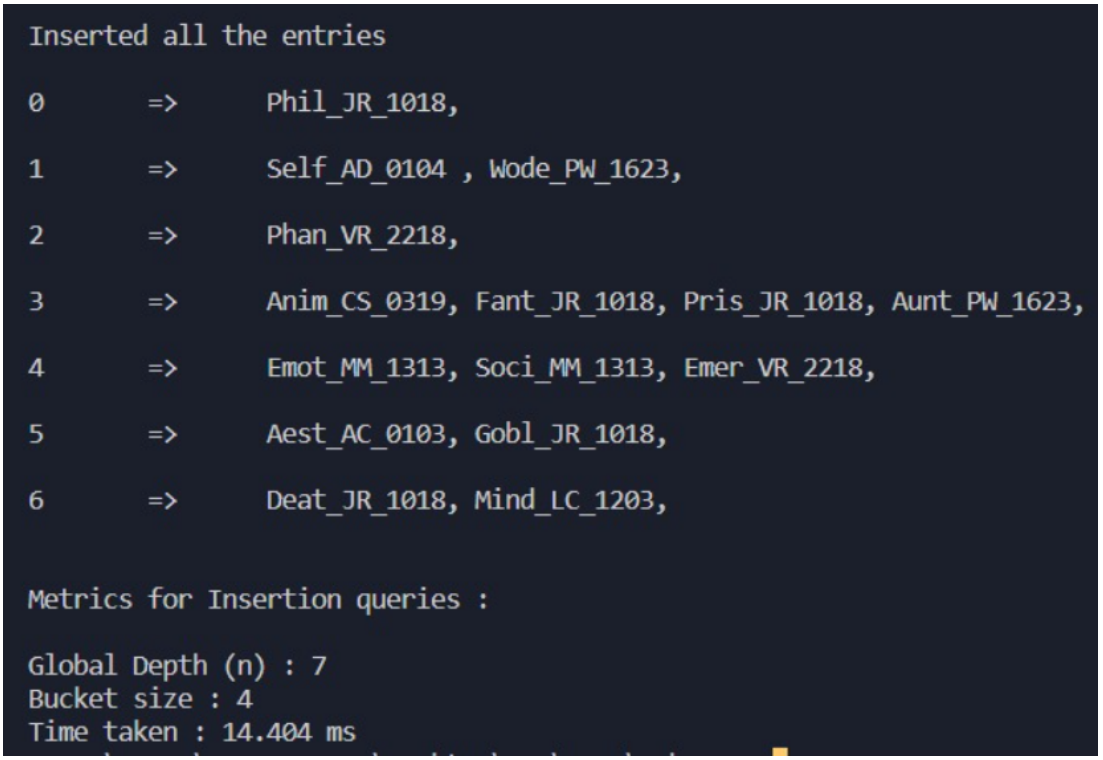
Case 2 : Increasing the Bucket Size to 6

```
Inserted all the entries

0       =>      Phil_JR_1018,

1       =>      Self_AD_0104 , Wode_PW_1623,

2       =>      Phan_VR_2218,

3       =>      Anim_CS_0319, Fant_JR_1018, Pris_JR_1018, Aunt_PW_1623,

4       =>      Soci_MM_1313, Emot_MM_1313, Emer_VR_2218,

5       =>      Aest_AC_0103, Gobl_JR_1018,

6       =>      Mind_LC_1203, Deat_JR_1018,


Metrics for Insertion queries :

Initial Global Depth (n) : 7
Bucket size : 6
Time taken : 11.968 ms
```

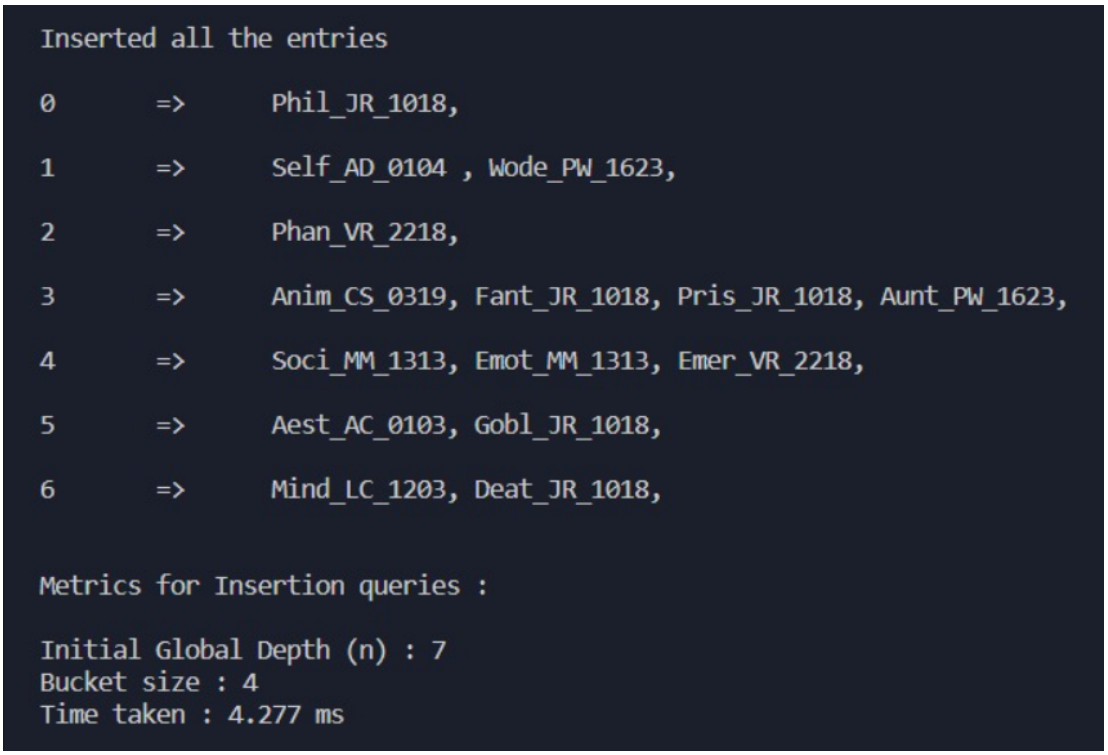
We see that keeping a small bucket size will increase execution time of queries. As we might have to extend the structure when buckets are full due to small size.
Also for quite large bucket sizes, queries become inefficient.

E. Comparisons on varying bucket structure :

Linear Structure :



Non - Linear Structure :



We can see that non - linear bucket structure takes less time than the other, as for linear structure, we search for the key in O(bucket_length) time whereas the other takes logarithmic time.

Q4.

a. The value of n that has been chosen is 4. As there were a considerable amount of values in the table taking the value of n, a bit high would give better results i.e; then by varying the value of n and checking the compilation time in making the distributed hash tree we could find the effective value of n .

b. Using Distributed Hashing with a bucket size of 4 and as mentioned considering the value of n as 4 we were able to build the distributed hash table.

→ For n=4 and bucket size=4

```
PS E:\compi> cd "e:\compi\" ; if ($?) { g++ distributed_hashing.cpp -o distributed_hashing }
4
4
time taken by the query to get executed is 0.000537 sec
```

c. Initially we considered n=4 and the compilation time for the insertion was 0.000537 sec.

For checking effective value of n as mentioned in the question considering one smaller value n=2 and bucket size 4 output will be

→ For n=2 and bucket size=4

```
PS E:\compi> cd "e:\compi\" ; if ($?) { g++ distributed_hashing.cpp -o distributed_hashing }
4
2
Error : The bucket size is exceeded. Hence, the given n is invalid
```

Considering the value of n=2 there will be overflow so our code throws an error of bucket size exceeded as shown above.

So we take another smaller value of n i.e; 3 and try to get the execution time

→ For n=3 and bucket size=4

```
PS E:\compi> cd "e:\compi\" ; if ($?) { g++ distributed_hashing.cpp -o distributed_hashing }
4
3
time taken by the query to get executed is 0.000583 sec
1
```

Now we repeat this process for a larger value of n (6) and bucket size 4

→ For n=6 and bucket size=4

```
PS E:\compi> cd "e:\compi\" ; if ($?) { g++ distributed_hashing.cpp
4
6
time taken by the query to get executed is 0.000406 sec
```

We can clearly see that execution time is reduced when we increase the value of n keeping the bucket size fixed.

Checking it for another greater value of n (8)

→ For n=8 and bucket size=4

```
PS E:\compi> cd "e:\compi\" ; if ($?) { g++ distributed_hashing.cpp -o distributed_hashing } ;
4
8
time taken by the query to get executed is 0.000381 sec
```

As we can see the execution time is further reduced.

d. For understanding whether changing bucket size we again experiment with taking different bucket sizes one lesser and larger values keeping the value of n constant (4) and calculating the compilation time for both these situations.

→ For n=4 and bucket size=3

```
PS E:\compi> cd "e:\compi\" ; if ($?) { g++ distributed_hashing.cpp -o distributed_hashing }
3
4
Error : The bucket size is exceeded. Hence, the given n is invalid
```

For bucket value 3 there will be overflow so considering smaller values than 3 will also give the same error so next we check for bucket size greater than 4.

→ For n=4 and bucket size=5

```
PS E:\compi> cd "e:\compi\" ; if ($?) { g++ distributed_hashing.cpp -o distributed_hashing } ;
5
4
time taken by the query to get executed is 0.000540 sec
```

Considering bucket size 8 and n=4 we get execution time as

→ For n=4 and bucket size=8

```
PS E:\compi> cd "e:\compi\" ; if ($?) { g++ distributed_hashing.cpp -o distributed_hashing } ;
8
4
time taken by the query to get executed is 0.000634 sec
```

e. Considering unordered_map and repeating the experiment for n=4 and bucket size 4 the execution time will be

→ when vector is the data structure considered for a bucket

```
PS E:\compi> cd "e:\compi\" ; if ($?) { g++ distributed_hashing.cpp -o distributed_hashing } ;
4
4
time taken by the query to get executed is 0.000254 sec
1
```

When the data structure was vector we had the following result

→ when Unordered_map is the data structure considered for a bucket

```
PS E:\compi> cd "e:\compi\" ; if ($?) { g++ distributed_hashing.cpp -o distributed_hashing }
4
4
time taken by the query to get executed is 0.000537 sec
```

We can clearly see the considerable difference in the execution times , there is decrease in the execution time when the data structure is unordered_map as the time complexity for searching and insertion will be less in case of unordered_map which is exactly what we see here.

Q 5.

a. Comparing all the hashing mechanisms, we found out the following results (These results also depend on the device it is run on since each of the three of us did one hashing code each and ran on our personal devices) :

b. Inserting a new record ***Finding Muchness by Kobi Yamada :***

Extendible Hashing :

```
    // end = clock();
    // double time_taken = double(end - start) / double(CLOCKS_PER_SEC);
    double time_taken = chrono::duration_cast<chrono::microseconds>(end - start).count();
    time_taken/=1e3;
    cout<<"\nTime taken by the insert queries using Extendible Hashing to get executed is "<<fixed<<time_taken<<setprecision(9);
    cout<<" ms"<<endl;
    // cout<<xd<<endl;
}
```

```
PROBLEMS 1  OUTPUT  DEBUG CONSOLE  TERMINAL  JUPYTER

Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\Users\YASH MANIYA\Desktop\C++> cd "c:\Users\YASH MANIYA\Desktop\C++\DBMS\Lab 4,5\" ; if ($?) { g++ saved.cpp -o saved } ; if ($?) { .\saved
Insert the Bucket Size : 4

Time taken by the insert queries using Extendible Hashing to get executed is 5.702000 ms
PS C:\Users\YASH MANIYA\Desktop\C++\DBMS\Lab 4,5>
```

Linear Hashing :

```
Inserted given book "Find_KY_2418" in bucket 6

Time taken to insert "Finding Muchness by Kobi Yamada" : 1.508 ms
```

Distributed Hashing :

```
    //Row R16("Ko_Ya_2418","Find_KY_2418","Kobi Yamada","Finding Muchness");
    insert(R16);
    // end = clock();
    auto end = chrono::high_resolution_clock::now();
    // double time_taken = double(end - start) / double(CLOCKS_PER_SEC);
    double time_taken = chrono::duration_cast<chrono::nanoseconds>(end - start).

    time_taken *= 1e-9;
    // cout<<time_taken<<endl;
    cout<<"time taken by the query to get executed is : "<<fixed<<time_taken<<se
    cout<<" sec"<<endl;
   // cout<<xd<<endl;
```

```
PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE

PS E:\compi> cd "e:\compi\" ; if ($?) { g++ q5_b_distributed_hashing.cpp -o q5_b_distributed
4
4
time taken by the query to get executed is : 0.002079 sec
```

c. Retrieving a record ***What Animals Think by Carl Safina :***

Extendible Hashing :

```
    bool flag = search(R3);
    sleep(1);
    auto end = chrono::high_resolution_clock::now();


    // end = clock();
    // double time_taken = double(end - start) / double(CLOCKS_PER_SEC);
    double time_taken = chrono::duration_cast<chrono::microseconds>(end - start).count();
    time_taken/=1e3;
    cout<<"\nTime taken to search query using Extendible Hashing is "<<fixed<<time_taken<<setprecision(9);
    cout<<" ms"<<endl;
    // cout<<xd<<endl;
}
```

```
PROBLEMS 1  OUTPUT  DEBUG CONSOLE  TERMINAL  JUPYTER

Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\Users\YASH MANIYA\Desktop\C++> cd "c:\Users\YASH MANIYA\Desktop\C++\DBMS\Lab 4,5\" ; if ($?) { g++ saved.cpp -o saved } ; if ($?) { .\saved }
Insert the Bucket Size : 4

Time taken to search query using Extendible Hashing is 4.120000 ms
PS C:\Users\YASH MANIYA\Desktop\C++\DBMS\Lab 4,5>
```

Linear Hashing :

```
Found item :
Anim_CS_0319    What Animals Think      Ca_Sa_0319      Carl Safina

Time taken to search "What Animals Think by Carl Safina" : 1.995 ms
```

Distributed Hashing :

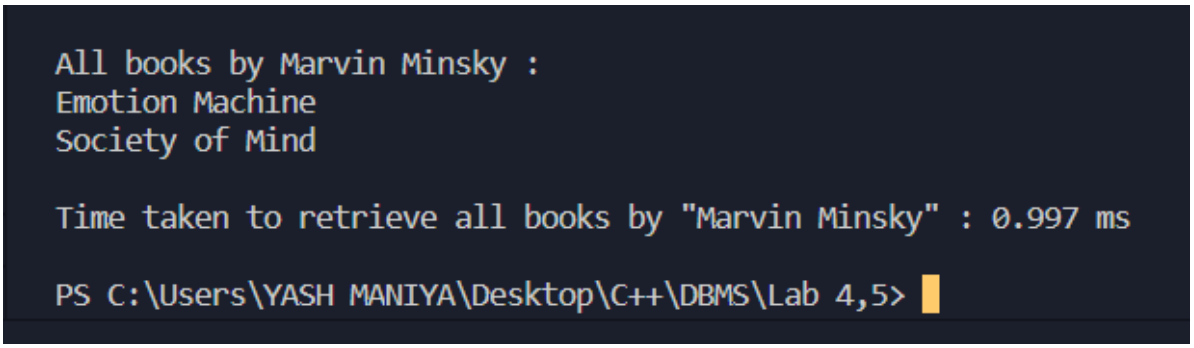
```
148     bool xd = search(R3);
149     // end = clock();
150     auto end = chrono::high_resolution_clock::now();
151     // double time_taken = double(end - start) / double(CLOCKS
152     double time_taken = chrono::duration_cast<chrono::nanoseco
153
```

```
PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE

PS E:\compi> cd "e:\compi\" ; if ($?) { g++ q5_c_distributed_hashing.cpp
4
4
time taken by the query to get executed is 0.001582 sec
```

d. Retrieving names of all books by Marvin Minsky :

Extendible Hashing :

Linear Hashing :



Distributed Hashing :