(6CS204) ADVANCED DATABASE SYSTEMS

NAME: DARSH S. KANTARIA

ENROLLMENT NO: 24MCD006

BRANCH: MTECH (DATA SCIENCE)

SUBMITTED TO:

Prof. Monika Shah

Darsh Kantaria **24MCD006** **MTECH (DS)**

PRACTICAL 3

Implement Extendible Hashing/Linear Hashing/Bitmap indexing, demonstrating insert, delete operations in the index table.

INTRODUCTION:

Hashing is a powerful technique used to manage large datasets efficiently by mapping data to a fixed-size table called a hash table or index. In modern databases, dynamic hashing techniques such as Extendible Hashing and Linear Hashing play a crucial role in maintaining the efficiency of data operations as datasets grow. Additionally, Bitmap Indexing is an efficient way to index data based on bitmaps, typically used when dealing with boolean data.

- **Extendible Hashing** addresses the problem of dynamically resizing hash tables by splitting and redistributing buckets when they overflow, without the need to rehash all entries.
- **Linear Hashing** is another dynamic hashing technique where the table expands one bucket at a time in a systematic manner, offering a more gradual approach to handling overflows.
- **Bitmap Indexing** is a space-efficient technique used for indexing in situations where data can be represented as bits, enabling quick search and retrieval operations.

This practical demonstrates the implementation of these three hashing techniques by handling basic operations such as insertion, deletion, and viewing the state of the index table.

IMPLEMENTATION:

The practical has been implemented using Streamlit, a Python framework for creating interactive applications. Each of the hashing algorithms is encapsulated in its own class, following a modular structure.

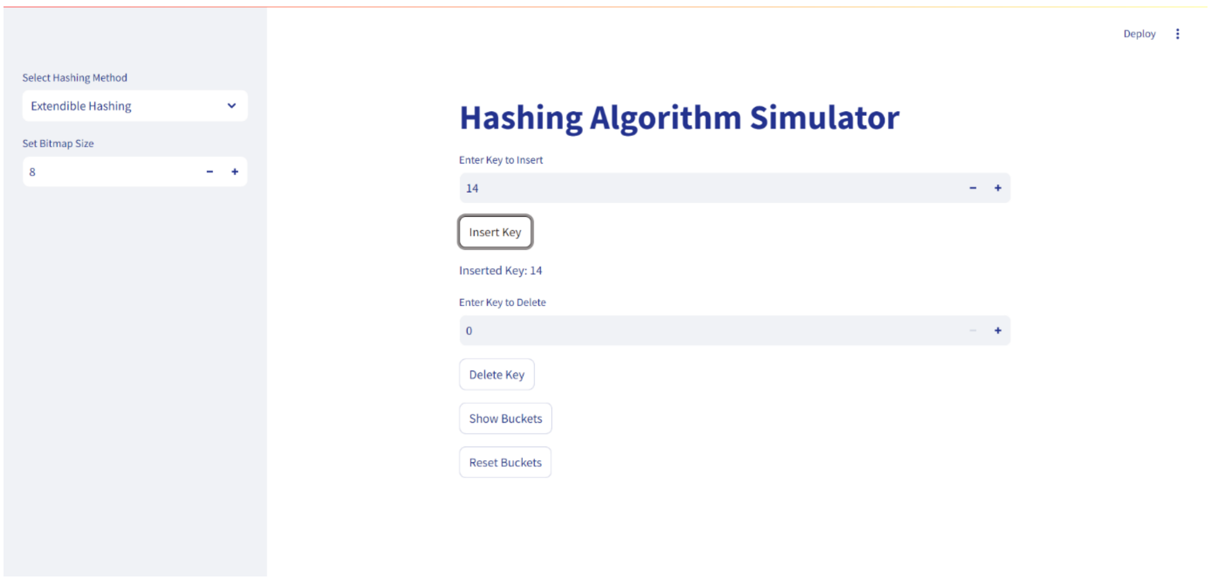
1. Extendible Hashing:

Extendible Hashing dynamically adjusts the size of its hash table by splitting individual buckets when they exceed the load factor. The number of bits (local and global depth) used for indexing increases as the buckets grow.

- **Insertion:** When a bucket overflows, it splits, and the global depth may increase. The keys in the original bucket are rehashed and redistributed.
- **Deletion:** A key is removed from the appropriate bucket without any major structural change.
- **Splitting:** Local depth of the bucket is increased when it overflows, and the global depth of the entire table increases when necessary.

The working of the Extendible Hashing is as follows:

- Initialize Extendible Hashing with a global depth of 1.
- Insert keys into the hash table. When a bucket exceeds the load factor threshold, the bucket splits, and keys are redistributed.
- Delete keys from their respective buckets and observe how the structure remains consistent.
- Display current buckets after each operation.

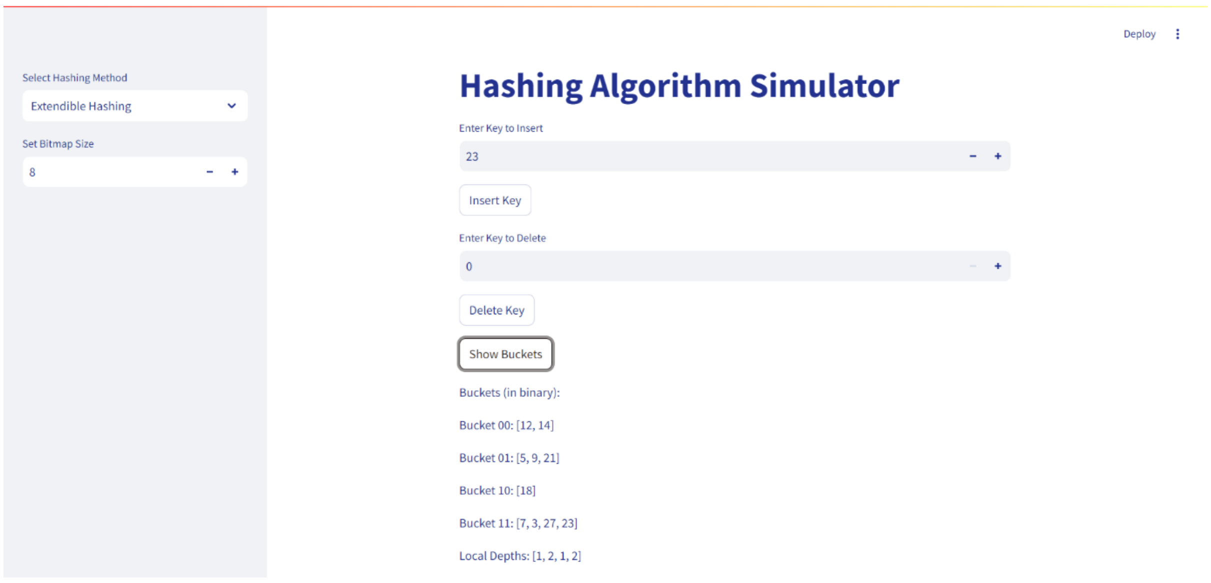


Hashing Algorithm Simulator

Enter Key to Insert
14

Inserted Key: 14

Enter Key to Delete
0


Deploy

Select Hashing Method

Extendible Hashing

Set Bitmap Size

8

Hashing Algorithm Simulator

Enter Key to Insert

23

Insert Key

Enter Key to Delete

0

Delete Key

Show Buckets

Buckets (in binary):

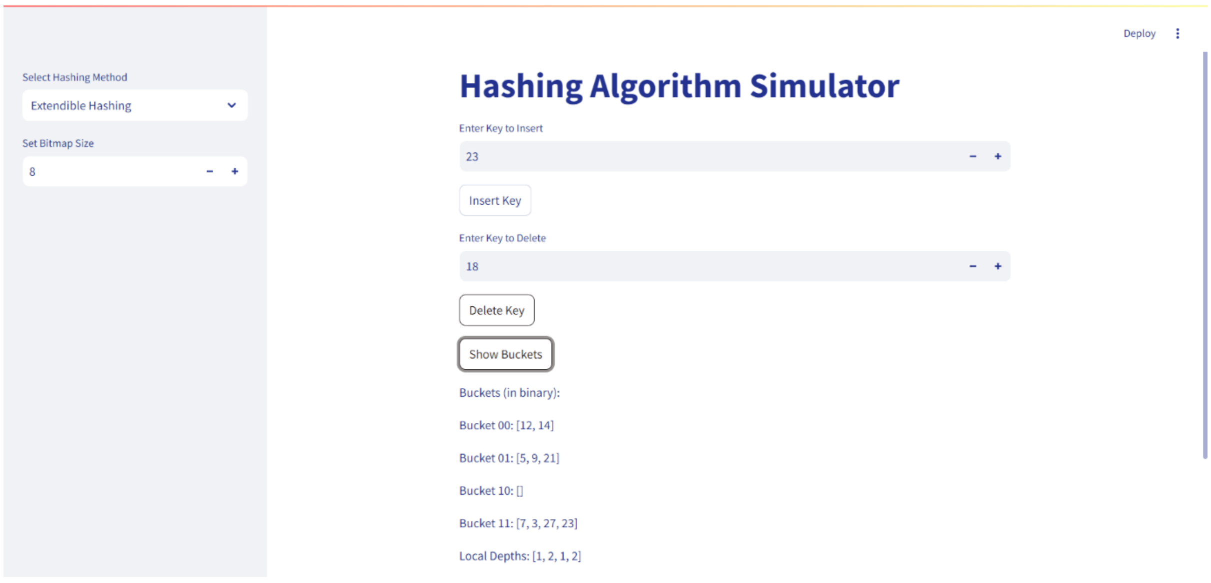
Bucket 00: [12, 14]

Bucket 01: [5, 9, 21]

Bucket 10: [18]

Bucket 11: [7, 3, 27, 23]

Local Depths: [1, 2, 1, 2]


Deploy

Select Hashing Method

Extendible Hashing

Set Bitmap Size

8

Hashing Algorithm Simulator

Enter Key to Insert

23

Insert Key

Enter Key to Delete

18

Delete Key

Show Buckets

Buckets (in binary):

Bucket 00: [12, 14]

Bucket 01: [5, 9, 21]

Bucket 10: []

Bucket 11: [7, 3, 27, 23]

Local Depths: [1, 2, 1, 2]

2. Linear Hashing:

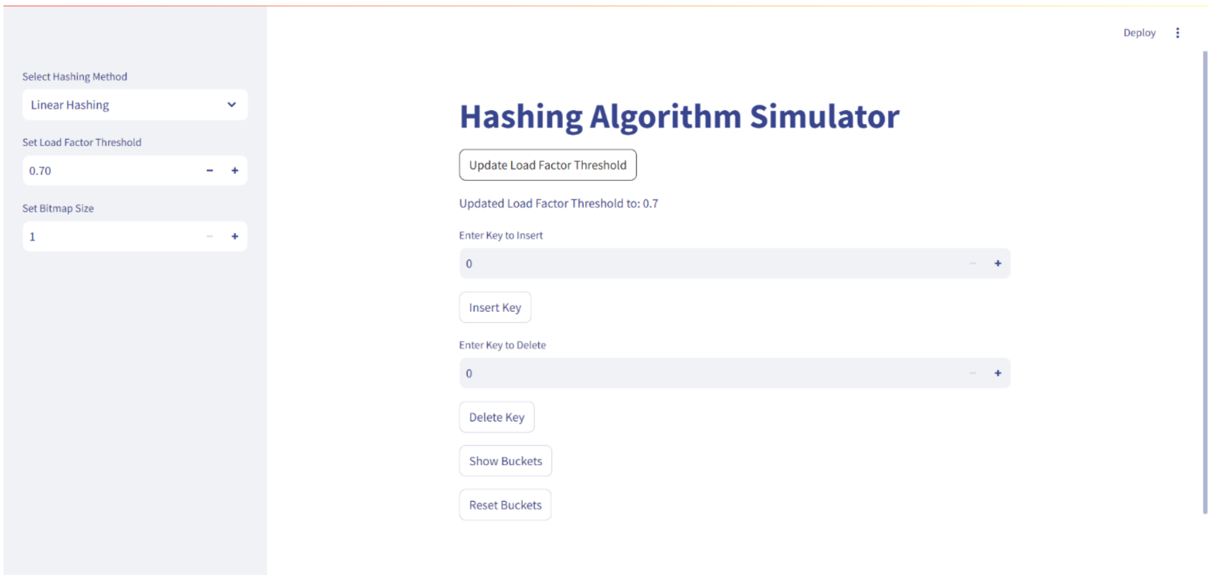
Linear Hashing expands the hash table one bucket at a time. The hash function depends on the current level of the table, and it adjusts dynamically as buckets are added.

- **Insertion:** If a bucket overflows, the next bucket is split, and the table expands one bucket at a time.
- **Deletion:** Similar to Extendible Hashing, a key is simply removed from its designated bucket.

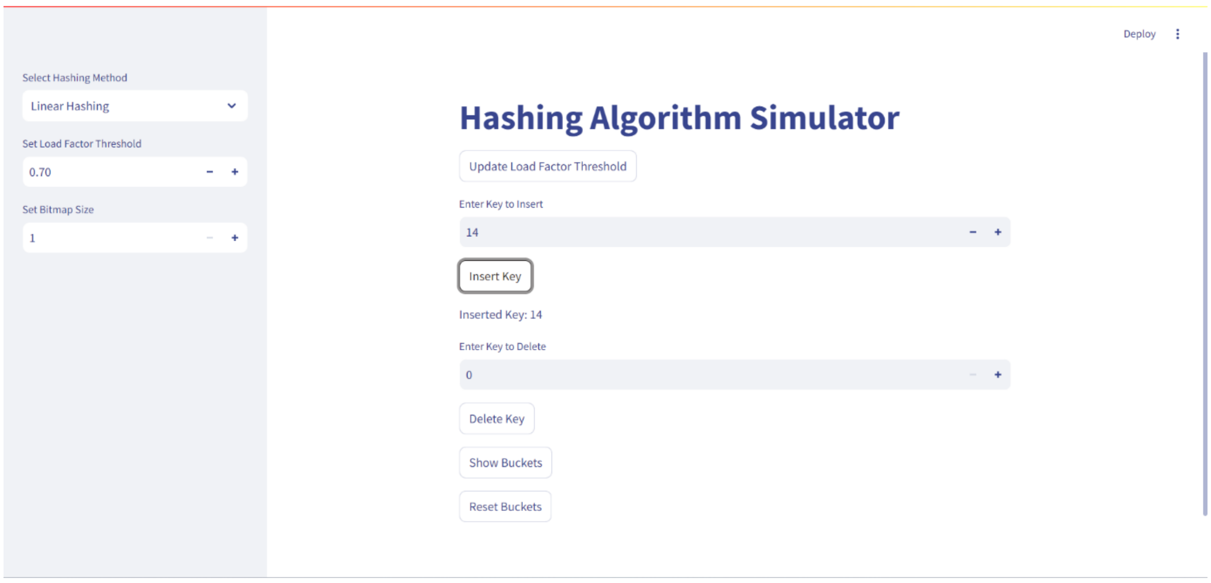
- **Splitting:** Unlike Extendible Hashing, Linear Hashing does not increase the number of bits but splits buckets one at a time based on a calculated split pointer.

  The working of the Linear Hashing is as follows:

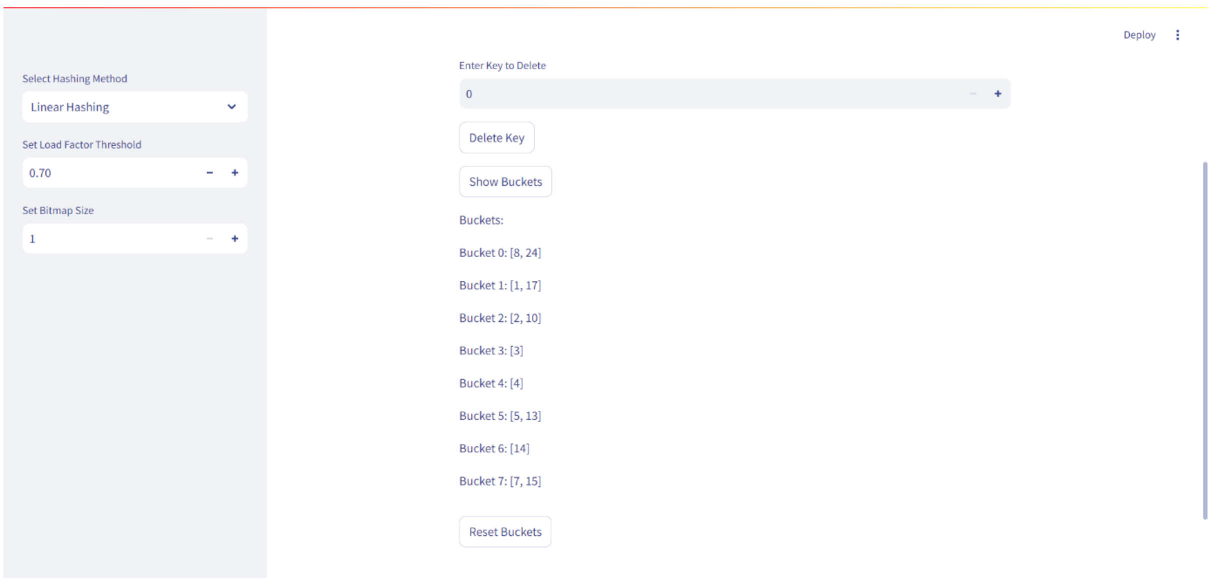
  - Start with an initial level of 1 and insert keys.
  - As buckets overflow, the table expands one bucket at a time, and keys are redistributed.
  - Deletion removes keys from the corresponding bucket.



Updated Load Factor Threshold to: 0.7



Inserted Key: 14



Buckets:

Bucket 0: [8, 24]

Bucket 1: [1, 17]

Bucket 2: [2, 10]

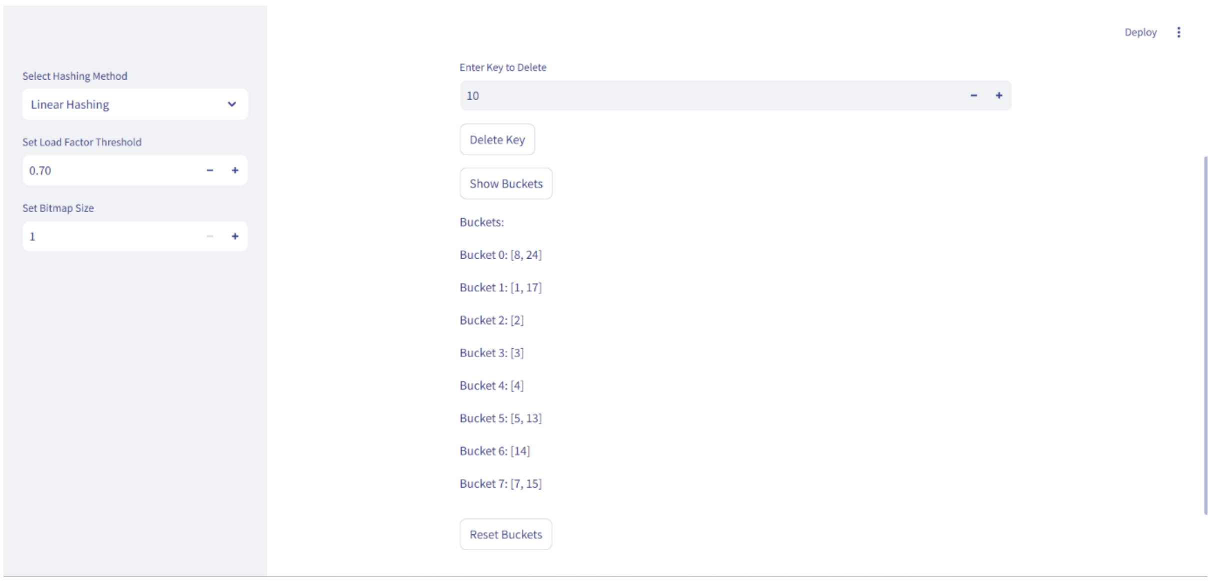
Bucket 3: [3]

Bucket 4: [4]

Bucket 5: [5, 13]

Bucket 6: [14]

Bucket 7: [7, 15]



Buckets:

Bucket 0: [8, 24]

Bucket 1: [1, 17]

Bucket 2: [2]

Bucket 3: [3]

Bucket 4: [4]

Bucket 5: [5, 13]

Bucket 6: [14]

Bucket 7: [7, 15]

3. Bitmap Indexing:

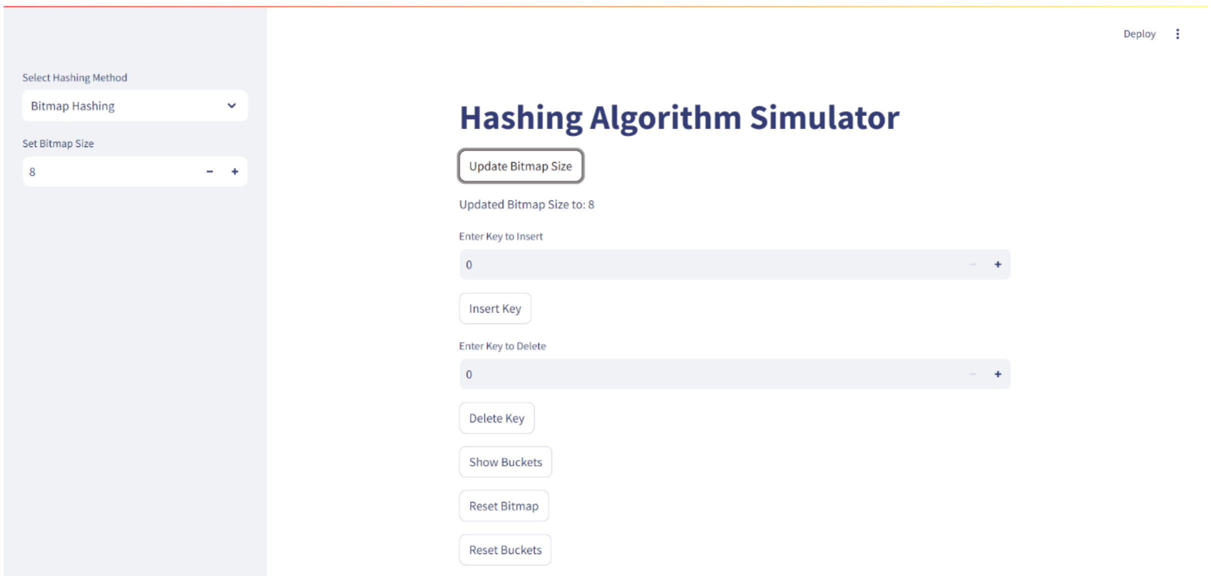
Bitmap Indexing uses an array of bits to represent the presence or absence of data items. The bitmap size is dynamically adjustable, and each inserted key maps to a specific bit in the bitmap.

- **Insertion:** The corresponding bit in the bitmap is set to 1, indicating the presence of the key.

- **Deletion:** The bit corresponding to the key is reset to 0, removing it from the bitmap.
- **Splitting:** The entire bitmap is reset to zeros, effectively clearing all entries.

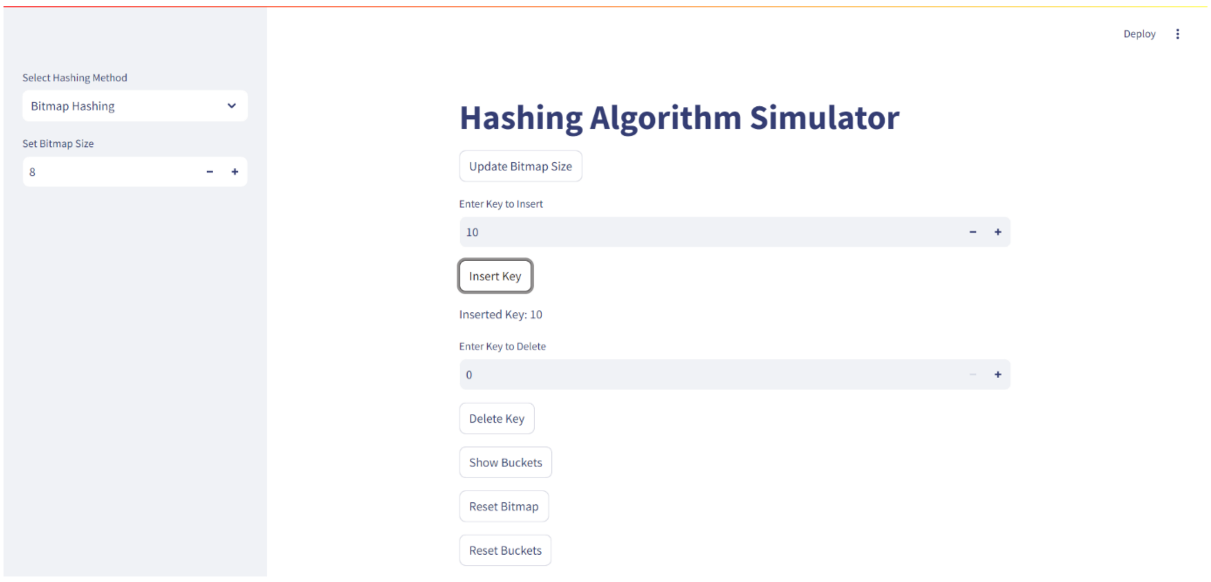
The working of the Bitmap Indexing is as follows:

- Set a bitmap size and insert keys. Each key sets the corresponding bit to 1.
- Deleting a key reset the corresponding bit in the bitmap.
- Display the current bitmap after every operation.


Select Hashing Method
Bitmap Hashing
Set Bitmap Size
8

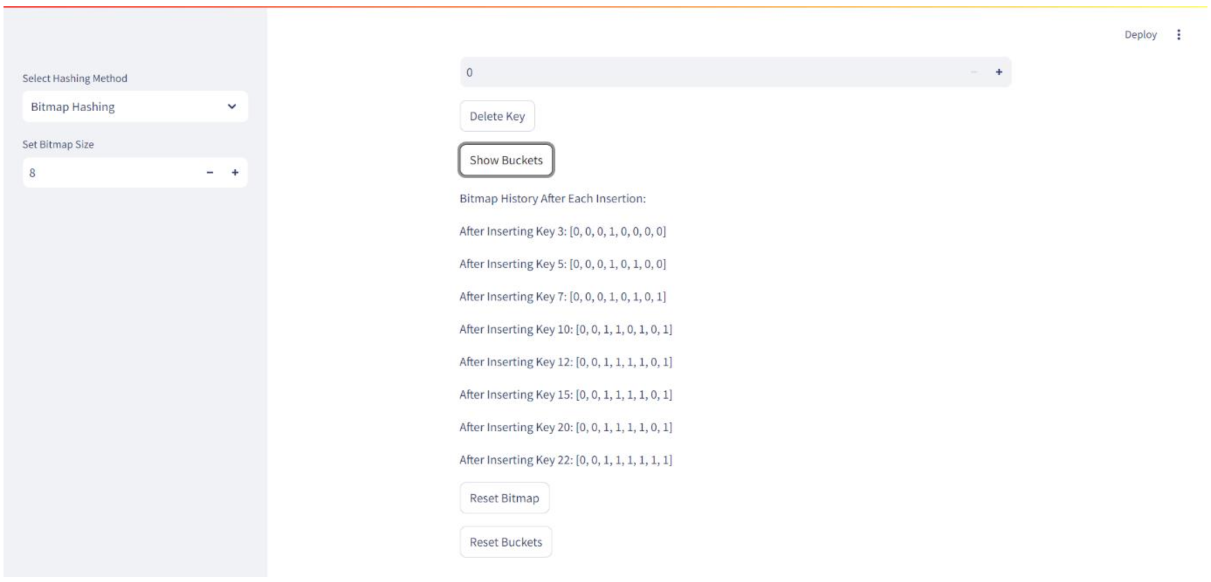
Hashing Algorithm Simulator

Updated Bitmap Size to: 8
Enter Key to Insert
0
Enter Key to Delete
0


Select Hashing Method
Bitmap Hashing
Set Bitmap Size
8

Hashing Algorithm Simulator

Enter Key to Insert
10
Inserted Key: 10
Enter Key to Delete
0

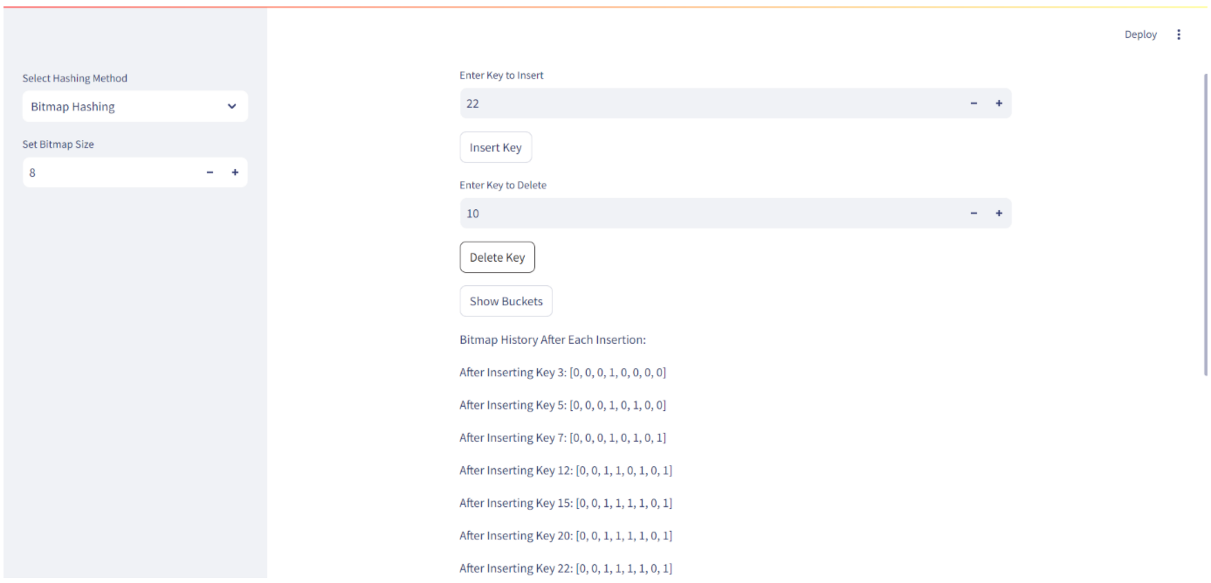


Select Hashing Method: Bitmap Hashing

Set Bitmap Size: 8

0

```
Bitmap History After Each Insertion:
After Inserting Key 3: [0, 0, 0, 1, 0, 0, 0, 0]
After Inserting Key 5: [0, 0, 0, 1, 0, 1, 0, 0]
After Inserting Key 7: [0, 0, 0, 1, 0, 1, 0, 1]
After Inserting Key 10: [0, 0, 1, 1, 0, 1, 0, 1]
After Inserting Key 12: [0, 0, 1, 1, 1, 1, 0, 1]
After Inserting Key 15: [0, 0, 1, 1, 1, 1, 0, 1]
After Inserting Key 20: [0, 0, 1, 1, 1, 1, 0, 1]
After Inserting Key 22: [0, 0, 1, 1, 1, 1, 1, 1]
```



Select Hashing Method: Bitmap Hashing

Set Bitmap Size: 8

Enter Key to Insert: 22

Enter Key to Delete: 10

```
Bitmap History After Each Insertion:
After Inserting Key 3: [0, 0, 0, 1, 0, 0, 0, 0]
After Inserting Key 5: [0, 0, 0, 1, 0, 1, 0, 0]
After Inserting Key 7: [0, 0, 0, 1, 0, 1, 0, 1]
After Inserting Key 12: [0, 0, 1, 1, 0, 1, 0, 1]
After Inserting Key 15: [0, 0, 1, 1, 1, 1, 0, 1]
After Inserting Key 20: [0, 0, 1, 1, 1, 1, 0, 1]
After Inserting Key 22: [0, 0, 1, 1, 1, 1, 0, 1]
```

CONCLUSION:

In this practical, we implemented three dynamic hashing techniques: **Extendible Hashing**, **Linear Hashing**, and **Bitmap Indexing**. Each of these techniques provides a robust method for managing data insertion and deletion, making them suitable for different types of databases and applications. While Extendible Hashing offers dynamic scaling for large datasets, Linear Hashing presents a more gradual expansion strategy. Bitmap Indexing, on the other hand, is highly space-efficient, especially for datasets that can be represented in binary form.

These techniques are crucial in ensuring that data operations remain efficient even as datasets grow, contributing to the overall performance of database systems.