

CUCKOO HASHING

A PROJECT REPORT

Submitted by

VARSHA.G.A (Reg. No. 9517202306111)

SHREE HARINI.K (Reg. No. 9517202306092)

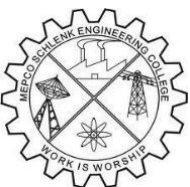
SOWMIYA SHIVANI.S (Reg. No. 9517202306096)

*in partial fulfillment for the award of the degree
of*

BACHELOR OF TECHNOLOGY

in

INFORMATION TECHNOLOGY



DEPARTMENT OF INFORMATION TECHNOLOGY
MEPCO SCHLENK ENGINEERING COLLEGE, SIVAKASI
(An Autonomous Institution affiliated to Anna University Chennai)



November 2024

Register No:

CERTIFICATE

*This is to certify that it is the Bonafide work done by
G.A.Varsha(9517202306111),K.Shree Harini(9517202306092),
S.Sowmiya Shivani(9517202306096) for the 23IT353 - Mini Project -
I work entitled “Cuckoo Hashing“ at Department of Information
Technology, Mepco Schlenk Engineering College, Sivakasi during the
year 2024 – 2025.*

.....
Faculty In-Charge

.....
Head of the Department

*Submitted for the Practical Examination held at Mepco Schlenk
Engineering College, Sivakasi on*

.....
Internal Examiner - I

.....
Internal Examiner - II

ABSTRACT

ABSTRACT

Cuckoo hashing is a scheme is a computer programming for resolving hash collisions of values of hash functions in a table, with worst case lookup time. On inserting a new key into a cuckoo hashing table may push an older key to a different location in the table .Cuckoo hashing is a form of Open Addressing in which each non-empty cell of a hash table contains a key-value pair. In one of the commonly used variants of the algorithm, the hash table is split into two smaller tables of equal size, and each hash function provides an index into one of these two tables. It is also possible for both hash functions to provide indexes into a single table. The concept of pairwise uncorrelated variables, where each pair of variables does not have a correlation with one another. This means that the variables are independent in their pairwise relationships, showing no linear association or predictive relationship between them. Using XOR-based transformations, we establish the uniqueness of two hash functions in cuckoo hashing while ensuring that they meet the 2-independence criteria. The mathematical proofs for pairwise uncorrelation validate the variance calculation for count sketches. Experimental evaluations demonstrate the effectiveness of these approaches in handling dynamic and large-scale datasets.

TABLE OF CONTENTS

	CONTENTS	PAGE NO
	LIST OF FIGURES	8
CHAPTER 1	INTRODUCTION	11
1.1	AIM	11
1.2	OBJECTIVE	11
1.3	HASHING	11
1.4	KEY CONCEPTS	11
	1.4.1 HASH FUNCTION	11
	1.4.2 HASH TABLE	11
	1.4.3 LOAD FACTOR	12
1.5	COLLISION	12
	1.5.1 COLLISION RESOLUTION TECHNIQUE	12
1.6	HOW HASHING WORKS	14
1.7	HASHING ALGORITHMS	15
CHAPTER 2	CUCKOO HASHING & COUNT SKETCHING	16
2.1	CUCKOO HASHING	17
2.2	KEY COCEPTS	17
	2.2.1 MULTIPLE HASH FUNCTION	17
	2.2.2 INSERTION PROCESS	17
	2.2.3 REHASHING	17
	2.2.4 SEARCH AND DELETION	17
2.3	CUCKOO HASHING IMPLEMENTATION	18

2.4	EFFICIENCY OF CUCKOO HASHING	21
2.5	IDEA	21
2.6	ANALYZING CUCKOO HASHING	22
	2.5.1 THE CUCKOO GRAPH	22
2.7	COUNT SKETCHING	23
	2.7.1 OVERVIEW	23
	2.7.2 KEY FEATURES	23
2.8	PAIRWISE UNCORELLATION	24
	2.8.1 OVERVIEW	24
	2.8.2 SIGNIFICANCE	24
2.9	2-INDEPENDENT	25
	2.9.1 EXAMPLES	25
CHAPTER 3	PROBLEM STATMENT	
3.1	PROBLEM	26
3.2	OUTCOME	29
CHAPTER 4	IMPLEMENTATION	
4.1	FLOWCHARTS AND ALGORITHM	
CHAPTER 5	CODE COMPONENTS AND EXPLANATION	34
5.1	OVERVIEW	35
5.2	MODULE	35
	5.2.1 STANDARD LIBRARY FUNCTION	35
	5.2.2 CUCKOO HASH TABLE CLASS	35
	5.2.3 MAIN FUNCTION	36

CHAPTER		
6	RESULTS AND DISCUSSION	45
6.1	SCREENSHOTS	46
CHAPTER		
7	CONCLUSION AND APPLICATION	47
7.1	APPLICATION	
7.2	CONCLUSION	
CHAPTER		
8	APPENDIX	50
8.1	CODE	51
CHAPTER 9	REFERENCES	55

LIST OF FIGURES

LIST OF FIGURES

S.NO	FIGURE NO	FIGURE NAME	PAGE NO
1.	1.4.2	Hash Table	11
2.	1.5	Collision	12
3.	1.5.1	Collision	13
4.	1.6	How Collision Works	14
5.	2.3	Cuckoo Hashing Implementation	20
6.	2.4	Efficiency of Cuckoo Hashing	21
7.	2.5	Idea of Cuckoo Hashing	21
8.	2.6.1	Cuckoo Graph	22
9.	2.7	Count Sketching	23
10.	6.1	Output Screenshot	44

INTRODUCTION

CHAPTER 1

INTRODUCTION

1.1 AIM

To provide an efficient hashing framework for managing dynamic data with mathematical proof of 2-independence and pairwise uncorrelation.

1.2 OBJECTIVE

- Ensuring independence in hash functions for effective Cuckoo Hashing.
- Proving pairwise uncorrelation in random variables derived from Count Sketch.

1.3 HASHING

Hashing is a technique used in data structures to efficiently store and retrieve data. It involves transforming a given key (such as a number, string, or any other type of data) into an index in an array (called a hash table) using a hash function. This index is where the corresponding value will be stored or searched.

1.4 KEY CONCEPTS

1.4.1 HASH FUNCTION

- A function that takes a key as input and produces an integer, known as the hash code or hash value.
- The hash code is then mapped to an index in the hash table.
- Example: $h(\text{key}) = \text{key} \% \text{table_size}$ is a simple hash function where `table_size` is the size of the hash table.

1.4.2 HASH TABLE

- An array-like data structure where the hash values determine the index where the keys and their corresponding values are stored.
- Each index of the hash table is called a "bucket."

0	1	2	3	4	5
2	40		310	74	9

Fig 1.4.2 Hash Table

1.4.3 LOAD FACTOR

- The ratio of the number of elements in the hash table to the size of the hash table.
- It measures how full the hash table is and helps decide when to resize or rehash the table to reduce collisions.

1.5 COLLISION

- Occurs when two different keys produce the same hash value and thus map to the same index in the hash table.
- Example: If $h(5) = 2$ and $h(12) = 2$ in a table of size 10, both keys 5 and 12 will try to occupy the same index (2).

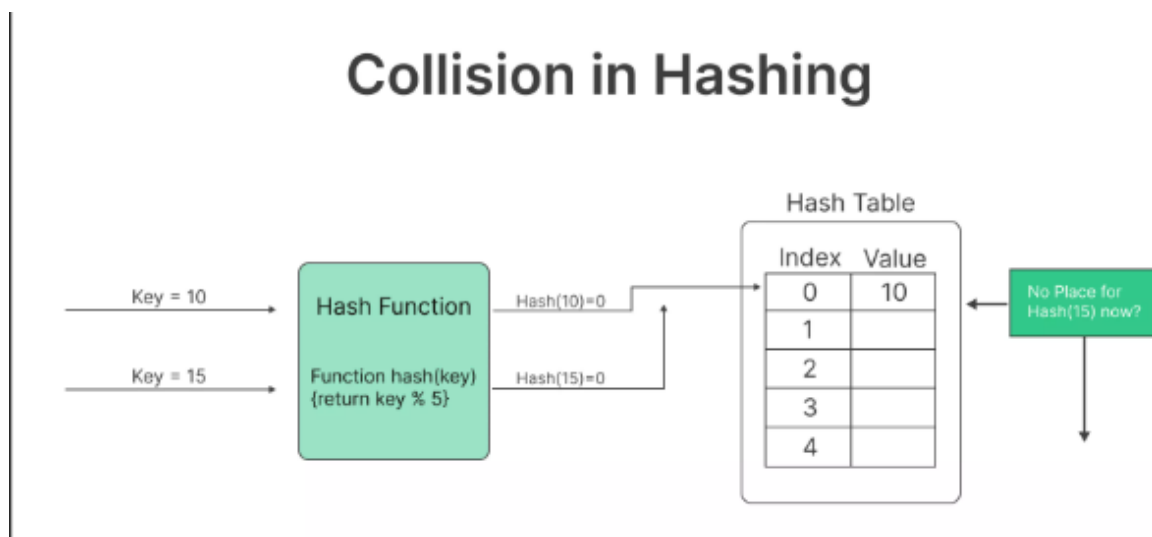


Fig 1.5 Collision

1.5.1 COLLISION RESOLUTION TECHNIQUES

- **Chaining:** Each bucket in the hash table points to a list (or another data structure) of all keys that hash to the same index.
- **Open Addressing:** If a collision occurs, the algorithm probes (looks for) the next available index in the hash table

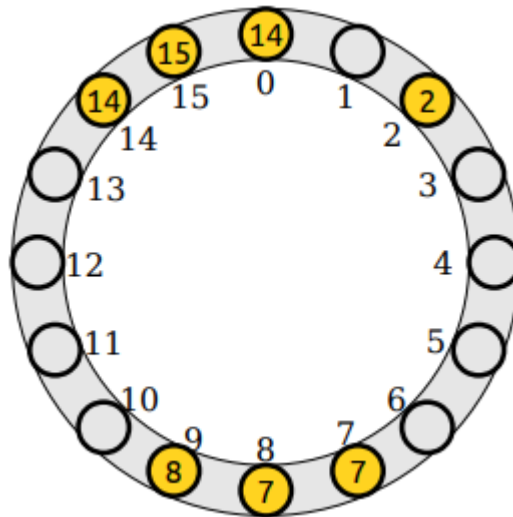


Fig 1.5.1 COLLISION

EXAMPLES

Suppose we have a hash table of size 10 and we want to store values associated with keys

1. **Keys:** [5, 12, 15, 25]
2. **Hash Function:** $h(\text{key}) = \text{key} \% 10$
3. **Mapping**
 - $h(5) = 5 \% 10 = 5$ o $h(12) = 12 \% 10 = 2$
 - $h(15) = 15 \% 10 = 5$ (collision with 5)
 - $h(25) = 25 \% 10 = 5$ (collision with 5 and 15)
4. **Visual Representation:**

Index: 0 1 2 3 4 5 6 7 8 9

Table: [] [] [12] [] [] [5 -> 15 -> 25] [] [] [] []

Here, values 15 and 25 are chained to index 5 due to collisions. Using a good hash function and handling collisions effectively are crucial for efficient hashing.

Advantages of Hashing

- **Fast Access:** Typically provides constant time complexity, $O(1)$, for search, insert, and delete operations, assuming good collision resolution
- **Efficient Memory Usage:** Requires less memory compared to other data structures like arrays or linked lists for large datasets.

Use cases

- **Database Indexing:** Quickly locating records in databases.
- **Caching:** Storing frequently accessed data to reduce access time

- **Symbol Tables in Compilers:** Storing and quickly accessing variable and function names.

Hashing is a fundamental concept in computer science, used in various applications for its speed and efficiency in handling large datasets

1.6 HOW HASHING WORKS

The process of hashing can be broken down into three steps:

- **Input:**

The data to be hashed is input into the hashing algorithm.

- **Hash Function:**

The hashing algorithm takes the input data and applies a mathematical function to generate a fixed-size hash value. The hash function should be designed so that different input values produce different hash values, and small changes in the input produce large changes in the output.

- **Output:**

The hash value is returned, which is used as an index to store or retrieve data in a data structure.

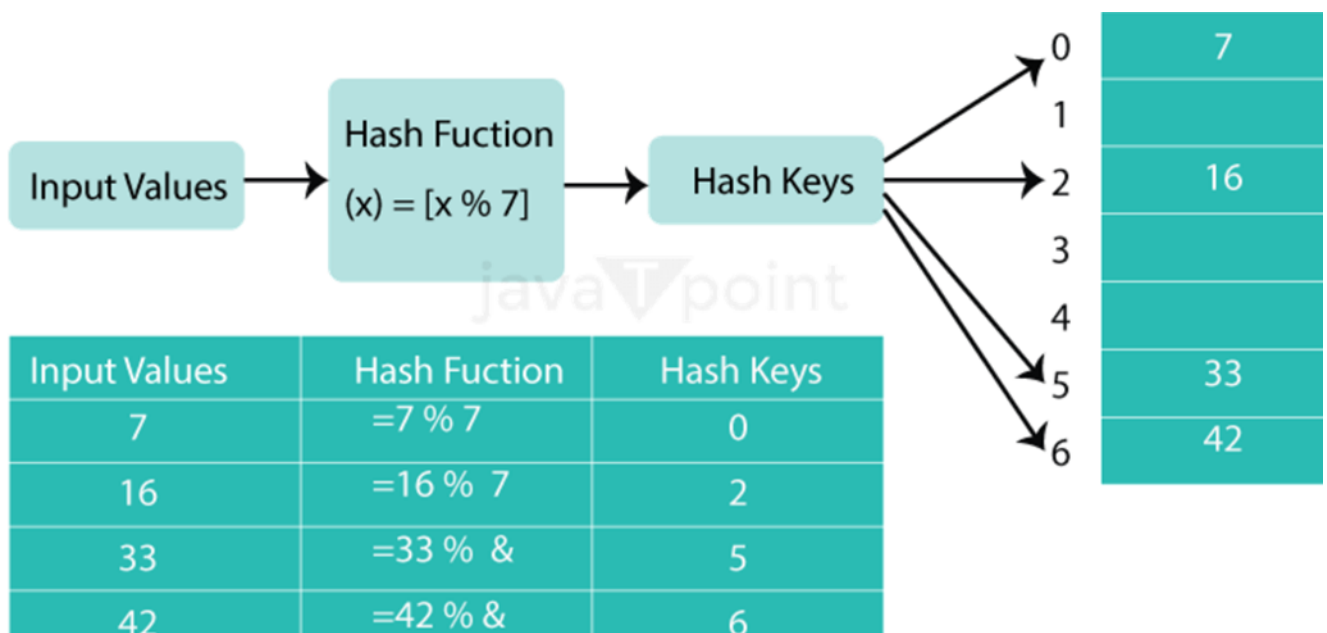


Fig 1.6 How Hashing Works

1.7 Hashing Algorithms:

There are numerous hashing algorithms, each with distinct advantages and disadvantages. The most popular algorithms include the following:

- MD5: A widely used hashing algorithm that produces a 128-bit hash value.
- SHA-1: A popular hashing algorithm that produces a 160-bit hash value.
- SHA-256: A more secure hashing algorithm that produces a 256-bit hash value.

CUCKOO HASHING & COUNT SKETCHING

CHAPTER 2

CUCKOO HASHING

2.1 CUCKOO HASHING

Cuckoo hashing is a modern and efficient hashing technique used in data structures to resolve collisions. It differs from traditional hashing methods by using multiple hash functions and allowing an element to be placed in one of multiple possible positions in the hash table. If a collision occurs, cuckoo hashing moves or "kicks out" the existing element to make room for the new element, much like a cuckoo bird laying its egg in another bird's nest.

2.2 KEY CONCEPTS

2.2.1 Multiple Hash Functions

- Cuckoo hashing uses two (or more) hash functions, h_1 and h_2 , which map keys to two possible locations in the hash table.
- For a key k , the possible positions are $h_1(k)$ and $h_2(k)$.

2.2.2 Insertion Process

- To insert a new key, the algorithm checks the two possible positions $h_1(k)$ and $h_2(k)$.
- If one of the positions is empty, the key is placed there. If both positions are occupied, one of the existing elements is displaced (or "kicked out") to make room for the new element.
- The displaced element is then reinserted using its alternate position, potentially displacing another element in a "cuckoo" fashion.

2.2.3 Rehashing

- If the displacement (kicking out) process goes on for too long (usually due to cycles or a filled table), the hash table is resized and new hash functions are chosen, followed by re-inserting all elements.

2.2.4 Search and Deletion

- To search for a key, the algorithm simply checks the two possible locations $h_1(k)$ and $h_2(k)$.
- Deletion involves finding the key in one of its two locations and removing it.

Advantages:

- **Constant Lookup Time:** $O(1)$ time complexity for search and insert operations. Simpler
- **Collision Handling:** Only two locations are checked, reducing the need for linked lists or probing sequences.

Disadvantages

- **High Space Overhead:** Requires more space than traditional hash tables due to the need to keep the load factor low.
- **Rehashing Cost:** Rehashing, when it happens, is costly because all elements must be re-inserted.

Example

Suppose we have a cuckoo hash table with two hash functions, h_1 and h_2 , and the following keys to insert

1. Hash Functions: $h_1(k) = (k \% 5)$ o $h_2(k) = ((k // 5) \% 5)$

2. Keys: 10, 15, 20, 25, 30.

3. Table Size: Assume a table size of 5.

- Insert 10: $h_1(10) = 0$ and $h_2(10) = 2$. Insert at position 0.
- Insert 15: $h_1(15) = 0$ and $h_2(15) = 3$. Position 0 is occupied (collision), so move 10 to its alternative position $h_2(10) = 2$ and place 15 at position 0.
- Insert 20: $h_1(20) = 0$ and $h_2(20) = 4$. Position 0 is occupied by 15, move 15 to its alternative position $h_2(15) = 3$, and place 20 at position 0.
- Insert 25: $h_1(25) = 0$ and $h_2(25) = 5$. Rehashing might be needed if this continues.

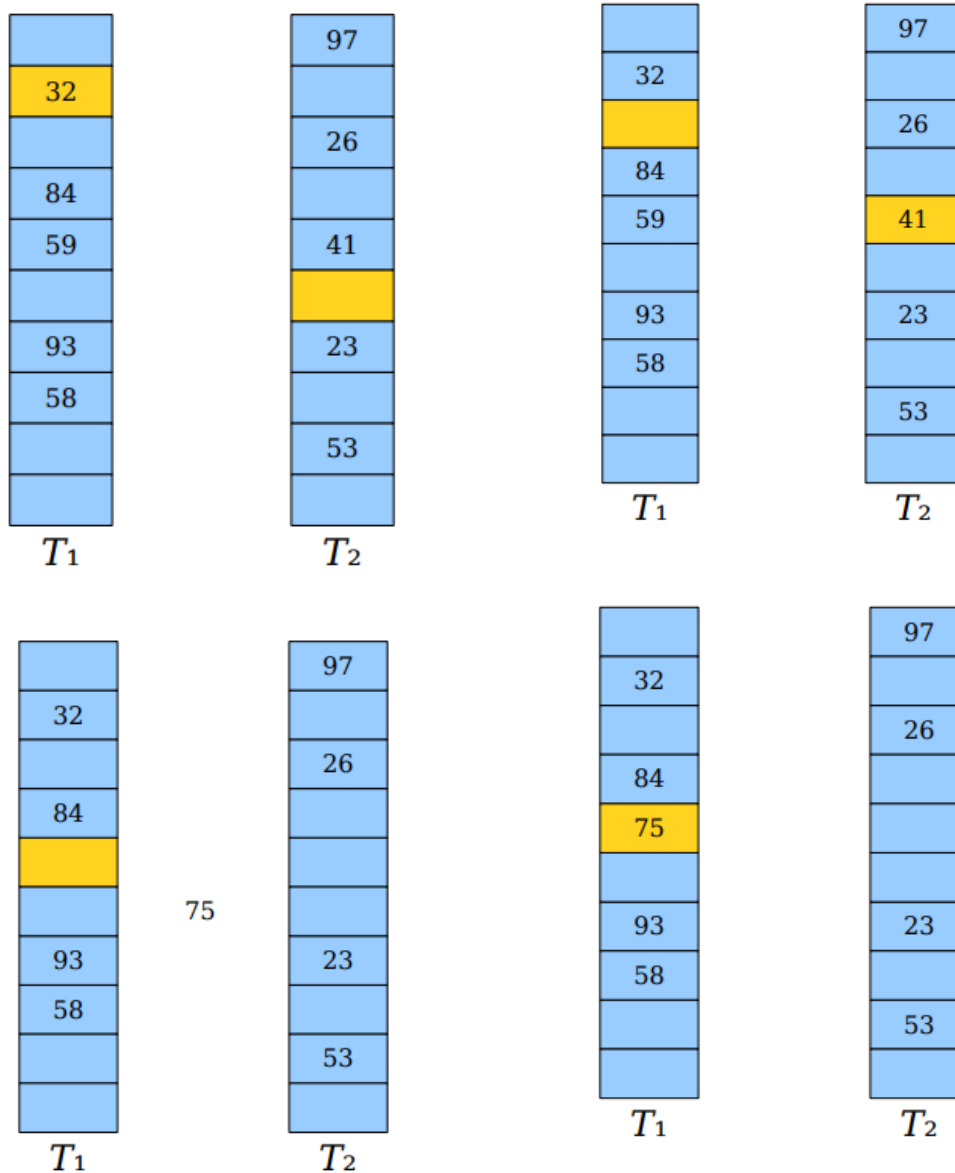
Visual Representation: Index: 0 1 2 3 4 Table: [20] [] [10] [15] []

Each insertion might cause a chain of displacements, making it efficient for lookup but complex in terms of insertion.

2.3 CUCKOO HASHING IMPLEMENTATION

- Maintain two tables, each of which has m elements.
- We choose two hash functions h_1 and h_2 from \mathcal{U} to $[m]$.
- Every element $x \in \mathcal{U}$ either be at position $h_1(x)$ in the first table or $h_2(x)$ in the second.
- We'll talk about hash strength later; for now, assume truly random hash function.
 - Lookups take worst case time $O(1)$ because only two locations must be checked.

- Deletions take worst case time $O(1)$ because only two locations must be checked
- To insert an element x , start by inserting it into table 1.
- If $h_1(x)$ is empty, place x there.
- Otherwise, place x there, evict the old element y , and try placing y in table 2.



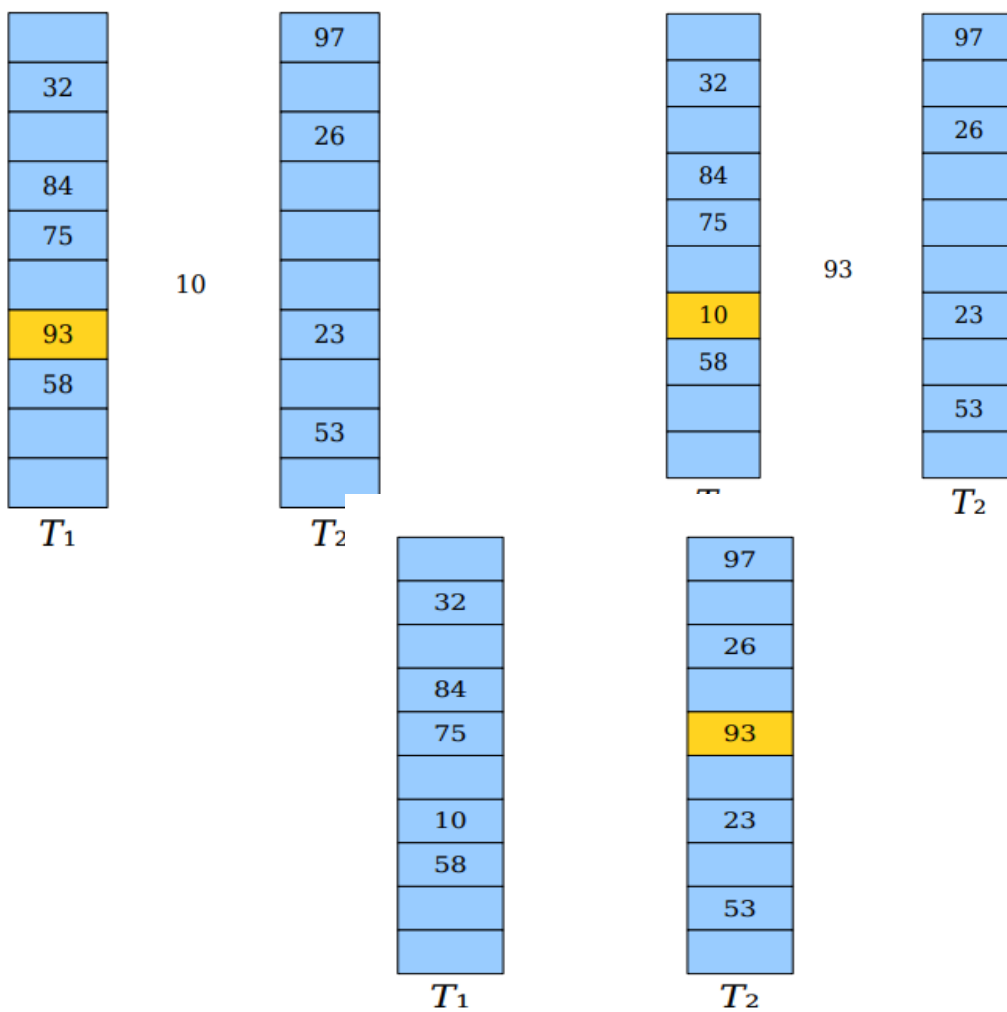
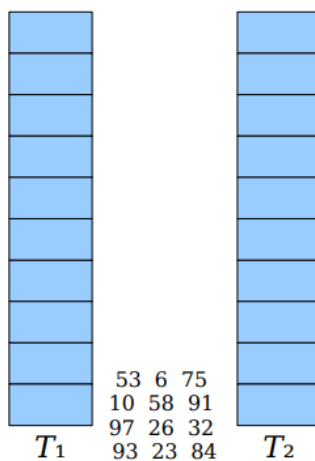


Fig 2.3 Cuckoo Hashing Implementation

- An insertion fails if the displacements form an infinite cycle.
- If that happens, perform a rehash by choosing a new h_1 and h_2 and inserting all elements back into the tables



2.4 EFFICIENCY OF CUCKOO HASHING

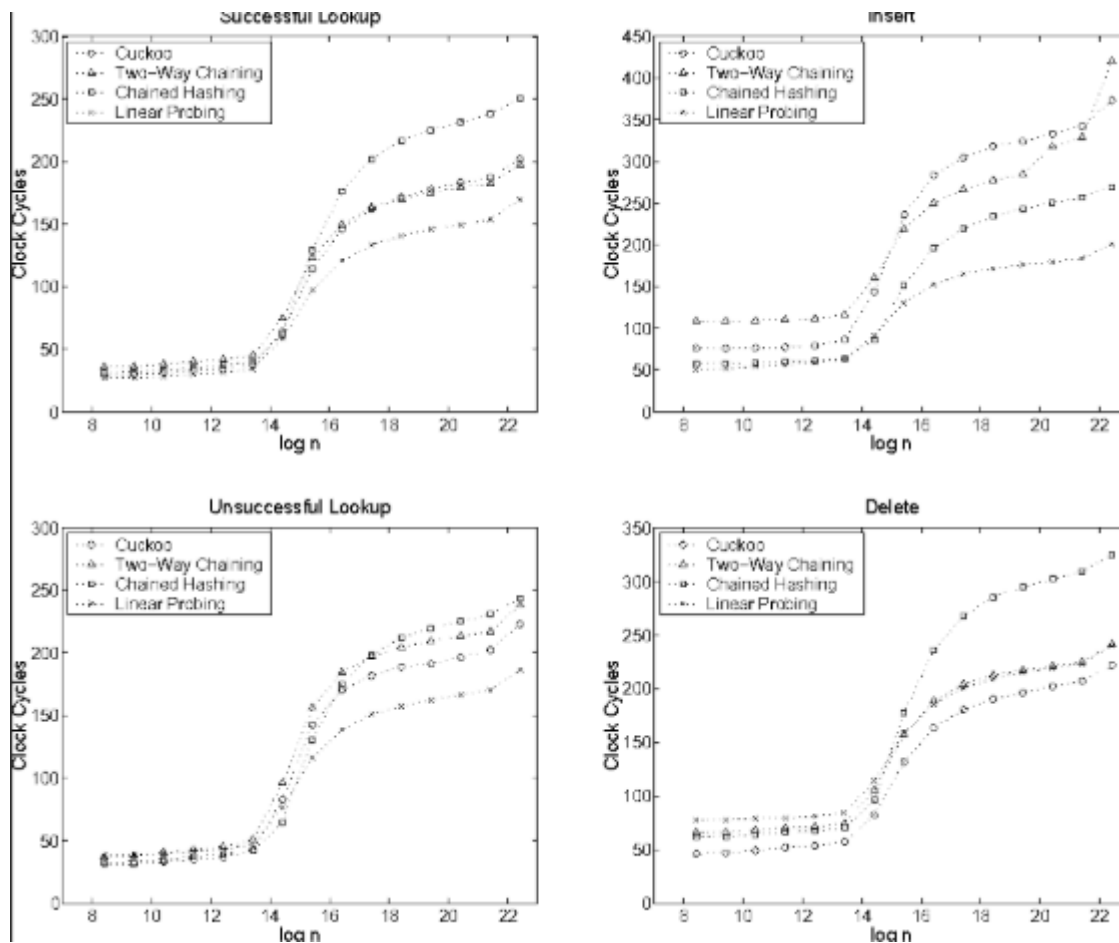


Fig 2.4 Efficiency of Cuckoo Hashing

- cuckoo hashing is about **20–30% slower** than linear probing, which is the fastest of the common approaches

2.5 IDEA

Idea

- 2 tables T_1 and T_2 (both of size $\Theta(n)$)
- 2 hash functions h_1 and h_2
- x either at $T_1[h_1(x)]$ or $T_2[h_2(x)]$

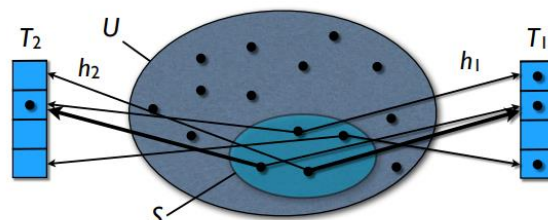


Fig 2.5 Idea of cuckoo Hashing

2.6 Analyzing Cuckoo Hashing

- The analysis of cuckoo hashing is more difficult than it might at first seem.
- **Challenge 1:** We may have to consider hash collisions across multiple hash functions.
- **Challenge 2:** We need to reason about chains of displacement, not just how many elements land somewhere.

2.6.1 The Cuckoo Graph

- The cuckoo graph is a bipartite multigraph derived from a cuckoo hash table.
- Each table slot is a node.
- Each element is an edge.
- Edges link slots where each element can be.
- Each insertion introduces a new edge into the graph.

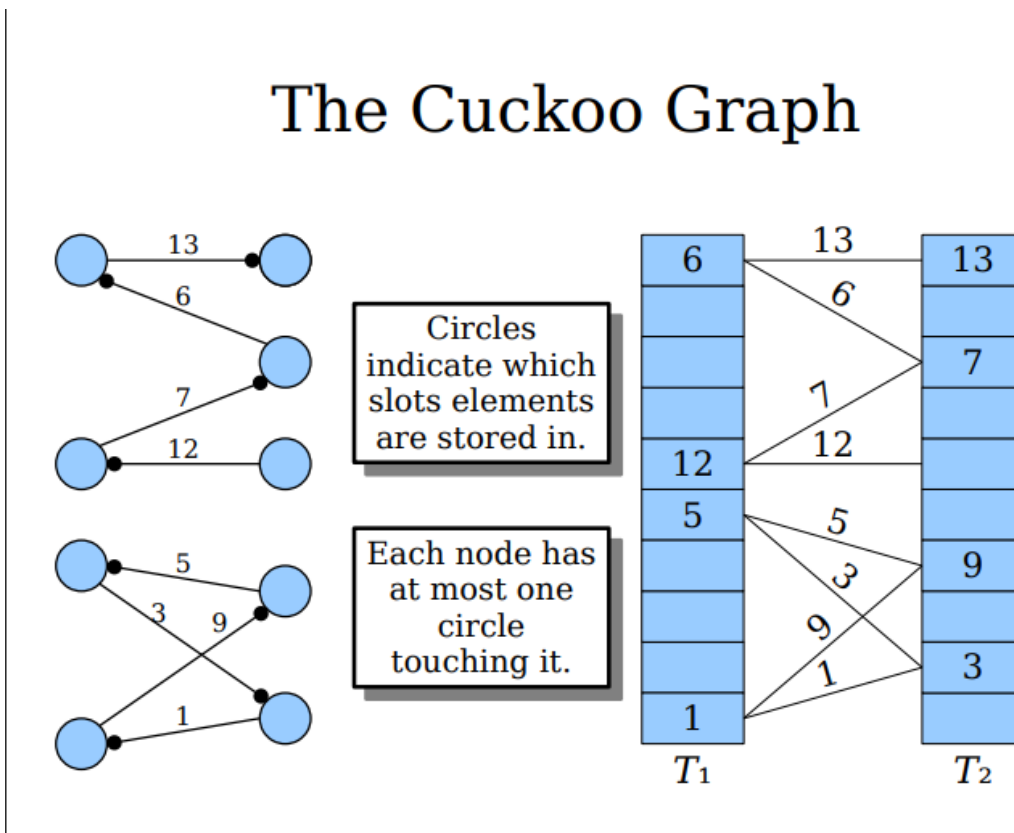


Fig 2.6.1 Cuckoo Graph

2.7 COUNT SKETCHING

2.7.1 OVERVIEW

- Count sketching operates using hash functions to map items into a smaller array and uses randomized sign assignments to handle collisions.
- It maintains a compact representation of the data stream, enabling efficient storage and query.

For an element x_i in the data stream:

1. Use a hash function $h(x_i)$ to map x_i to a bucket.
2. Assign a random sign $s(x_i) \in \{-1, +1\}$
3. Update the sketch at $h(x_i)$ by adding or subtracting x_i 's frequency based on $s(x_i)$

To estimate the frequency of an element, the algorithm aggregates contributions from multiple hash functions, reducing the effect of hash collisions.

2.7.2 Key Features

- **Linear Sketching:**
 - Reduces high-dimensional data into compact summaries.
 - Retains the ability to reconstruct approximate frequency information.
- **Randomization:**
 - Randomized hashing and sign functions ensure robustness against adversarial inputs.
- **Space Efficiency:**
 - Requires $O(w \cdot d)$ space, where w and d are the sketch dimensions.
- **Accuracy Guarantees:**
 - Provides probabilistic bounds on frequency estimation errors.

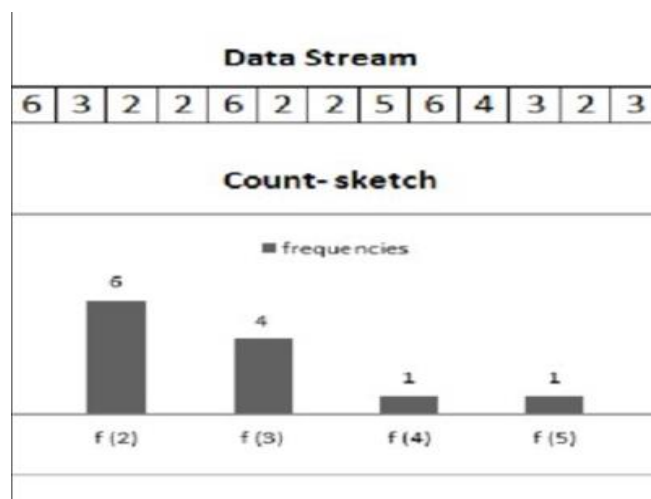


Fig 2.7. Count sketching

2.8 PAIRWISE UNCORRELATION

2.8.1 Overview

Pairwise uncorrelation is a property of two random variables X and Y such that their joint expectation equals the product of their individual expectations:

$$E[XY] = E[X] \cdot E[Y]$$

This implies that the variables X and Y do not influence each other's behavior in terms of their covariance, which is defined as:

$$\text{Cov}(X, Y) = E[XY] - E[X] \cdot E[Y]$$

For uncorrelated variables, $\text{Cov}(X, Y) = 0$.

2.8.2 Significance

In algorithms like **count sketches** and **cuckoo hashing**, pairwise uncorrelation is critical for simplifying computations. Specifically:

- It ensures that the variance of a sum of random variables can be treated as the sum of their variances.
- This property reduces the complexity of probabilistic analysis, allowing efficient approximations in data streaming, hashing, and machine learning

2.9 2-INDEPENDENT

- In the context of hashing, 2-independence (or pairwise independence) refers to a property of a family of hash functions
- . A family of hash functions is called 2-independent if, for any two distinct keys, the hash values are independent random variables.
- This means that the hash value of one key does not provide any information about the hash value of another key.
- This property is useful in ensuring that hash collisions are minimized and that the distribution of hash values is uniform.

2.9.1 Examples

- a and b are chosen uniformly at random from $\{1, 2, \dots, p-1\}$
- p is a prime number larger than m ,
- m is the size of the hash table.

PROBLEM STATMENT

CHAPTER 3

PROBLEM STATEMENT

3.1 PROBLEM

Part 1: Cuckoo Hashing

Cuckoo hashing uses two hash functions, $h_1(x)$ and $h_2(x)$, to efficiently resolve collisions in a hash table. The challenge here is twofold:

1. **Ensuring:**

- A solution is proposed by defining $h_2(x) = h_1(x) \oplus h_\Delta(x)$ where $h_\Delta(x)$ is another hash function.
- This ensures $h_2(x)$ is always distinct from $h_1(x)$, thanks to the properties of XOR and the independence of h_1 and h_Δ .

2. **Efficient displacement of elements:**

- Displacing an element x from one position to its alternate position $h_1(x)$ or $h_2(x)$ is made efficient by directly using the XOR operation, avoiding redundant computations.

The problem requires:

- Proving that $h_1(x) \neq h_2(x)$ for all x
- Showing that the displacement always moves xxx correctly between $h_1(x)$ and $h_2(x)$.
- Demonstrating that this hashing scheme is **2-independent**, meaning pairs of hash function values are independent of each other.

Part 2: Count Sketches

Count sketches estimate frequencies of elements in data streams, often under memory constraints. Variance analysis is critical to understanding their accuracy.

- The challenge lies in proving that certain terms in the variance analysis are **pairwise uncorrelated**, enabling simplification of variance calculations.
- The proof involves showing that the expected value of the product of two random variables equals the product of their expected values, under the assumption that hash functions and random signs are drawn independently.

OUR UNDERSTANDING OF THE PROBLEM:

QUESTION 1:

The problem discusses cuckoo hashing, which is a way to store keys in a hash table. It highlights two challenges:

- 1. Finding two hash functions, h_1 and h_2 , that always give different positions for the same key.
- 2. Moving a key to its new position correctly when it gets displaced should only require checking one of the hash functions.

The question also asks to prove that this method allows keys to be moved correctly between the two positions and that the chosen family of hash functions operates independently, ensuring even distribution of keys.

QUESTION 2:

- The task is to show that two specific random variables, involving different indices and drawn from independent hash functions, are uncorrelated. This means we need to prove that the expected value of their product equals the product of their expected values.

3.2 OUTCOME

The problem explores:

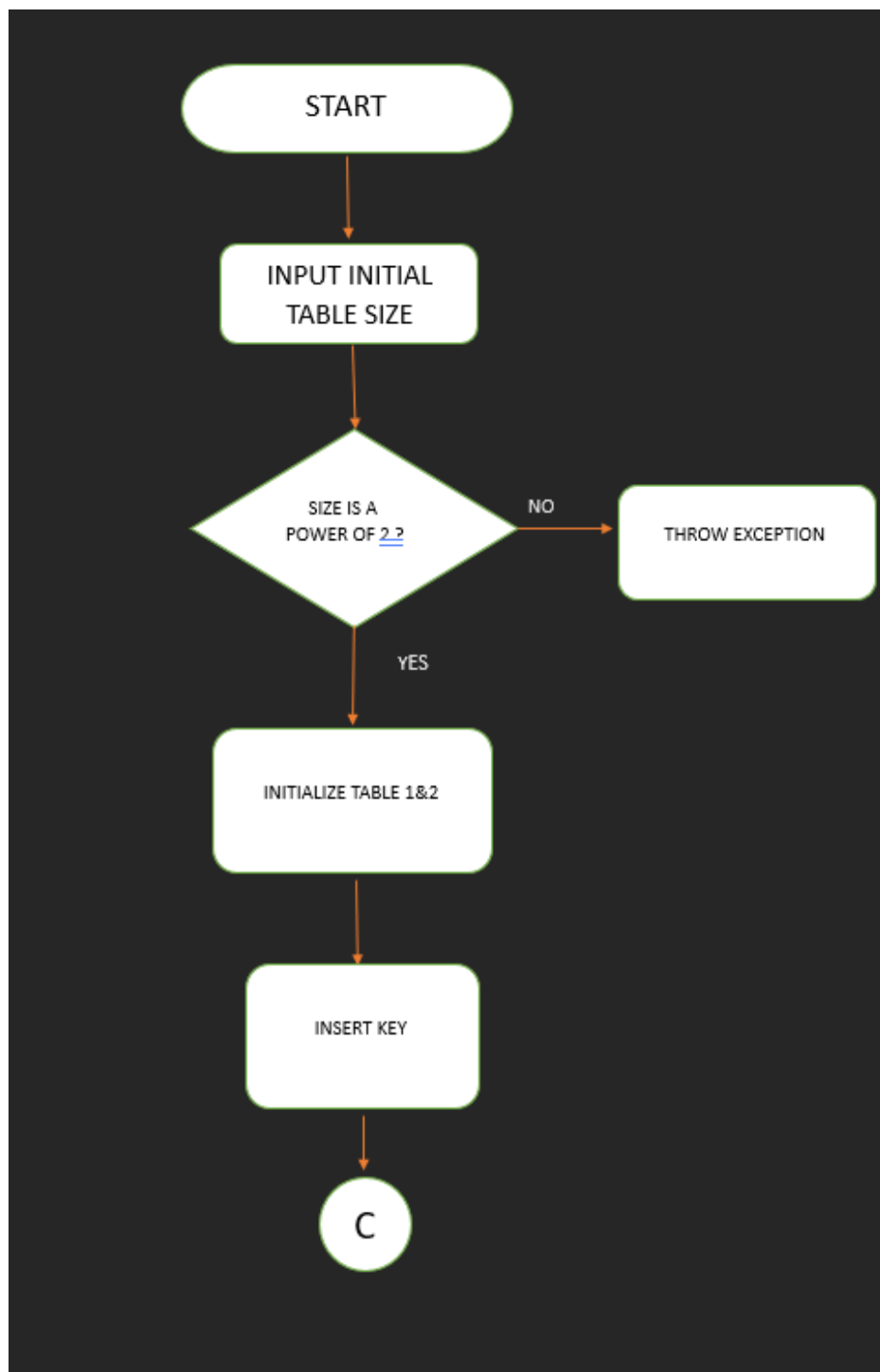
- How mathematical properties like XOR and independence simplify hash function design in Cuckoo Hashing.
- The importance of pairwise uncorrelation for accurate variance analysis in Count Sketches.

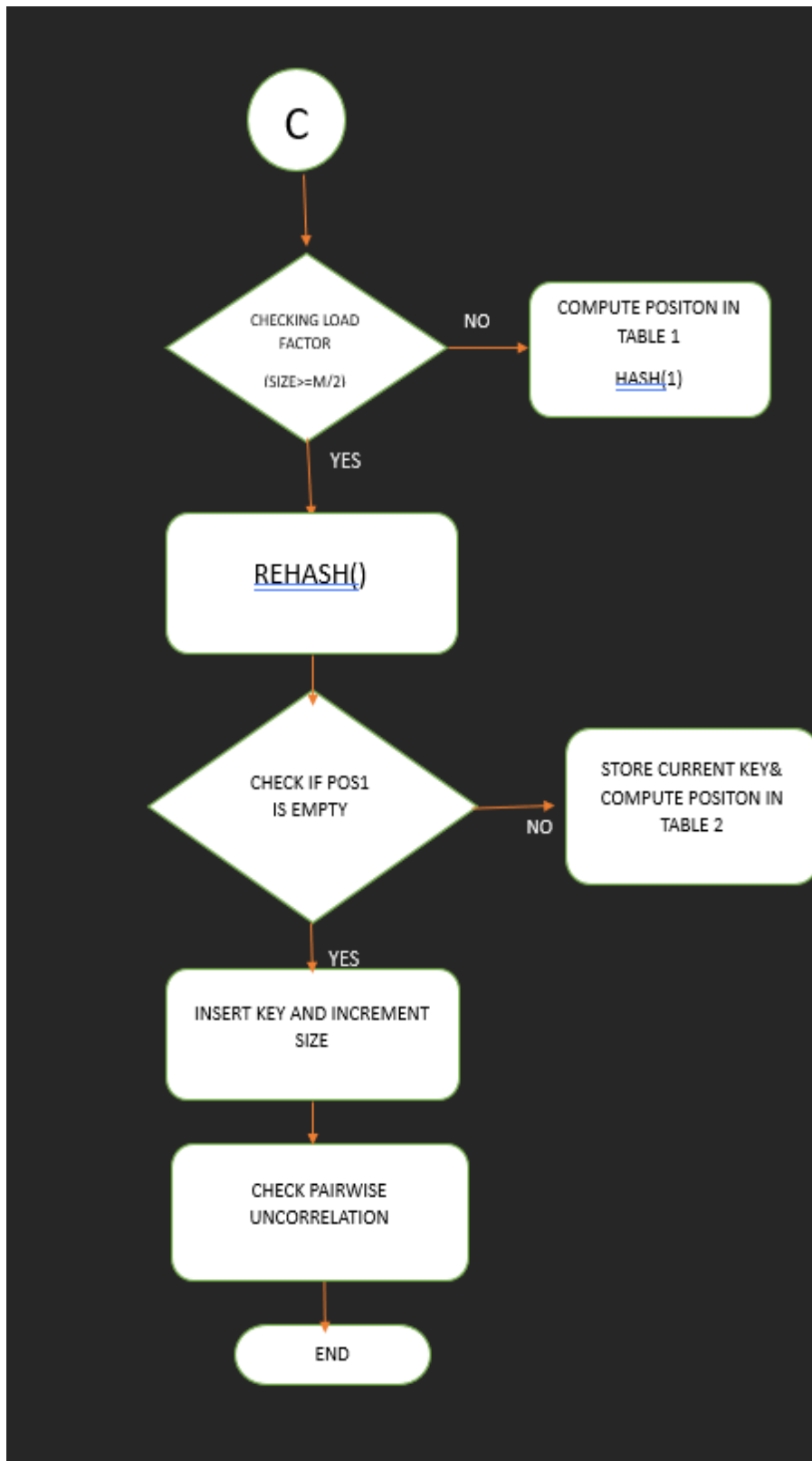
IMPLEMENTATION

CHAPTER 4

IMPLEMENTATION

4.1 FLOWCHART





• ALGORITHM

Here's a detailed explanation of the Cuckoo Hash Table algorithm without specific code parts, described in simple English:

Cuckoo Hash Table Algorithm

1. Overview:

A Cuckoo Hash Table is a data structure used to store keys (or unique items) for quick access. It uses two hash functions to map keys to two different tables. This technique helps manage collisions (when two keys attempt to occupy the same space) efficiently.

2. Class Setup:

- The Cuckoo Hash Table consists of two arrays (or tables) for storage, both of which have a fixed size.
- It also keeps track of the current number of elements and has a predefined constant to indicate empty slots.

3. Initializing the Hash Table:

- When creating the Cuckoo Hash Table, you specify the initial size for both tables. This size should be a power of two (like 2, 4, 8, etc.) for optimal performance.
- Both tables are set to empty at the beginning.

4. Hash Functions:

- The algorithm defines two hash functions:

First Hash Function: This function takes a key and calculates an index in the first table using a modulo operation.

Second Hash Function: This function calculates an index in the second table using a different method, involving XOR operation with a constant and modulo.

5. Inserting Keys:

- When inserting a key:
 1. The algorithm first checks if the current load factor exceeds a certain threshold. If it does, it triggers a rehashing operation to expand the tables.
 2. The algorithm uses the first hash function to find an index for the key in the first table.
 3. If that index is free (empty), the key is placed there. If it is occupied (a collision happens), the current key at that index is evicted (removed) and moved to its new location based on the second hash function.
 4. The newly evicted key is then placed in the second table using the second hash function.
 5. This process continues, where keys may swap places multiple times until a free spot is found.

6. Searching for Keys:

- To find a key:
 1. Use the first hash function to check the first table.
 2. If the key isn't found there, use the second hash function to check the second table.
 3. The search stops once the key is found or both tables have been checked.

7. Rehashing:

- If the load factor exceeds a predefined level (indicating the table is getting full), the hash tables are resized. The size is doubled, and all existing keys need to be reinserted into the new larger tables. This helps maintain fast access times.

8. Displaying the Table:

- The algorithm can display the contents of both tables along with other statistics, like the current load factor, which shows how full the table is.

9. Checking Pairwise Uncorrelation:

- The algorithm includes a method to check if two keys at specific positions in the tables are statistically independent from each other. It calculates expectations based on their hash values and determines whether they are correlated.

Summary

The Cuckoo Hash Table algorithm is a clever way to handle storage and retrieval of keys using two hash functions and a process of evicting keys when collisions occur. It includes mechanisms for resizing the storage when it gets too full and allows statistical analysis of keys stored in the tables. Overall, it provides efficient access and management of data.

CODE COMPONENTS AND EXPLANATION

CHAPTER 5

CODE COMPONENTS

5.1 OVERVIEW

This investigate the challenges and solutions in the implementation of cuckoo hashing and count sketching. It explores the properties of hash functions ensuring pairwise uncorrelation and 2-independence, essential for dynamic data management. The proposed approach leverages XOR-based transformations and incremental frameworks to achieve efficient hashing and accurate variance calculations under concept drift conditions. Experimental proofs validate the correctness and applicability of the methodology.

5.2MODULE

5.2.1 Standard Library Modules

- **#include <iostream>:**
 - **Purpose:** Provides functionality for input/output operations such as reading from the console and printing to the console.
 - **Key functions:** cin, cout, endl.
- **#include <iomanip>:**
 - **Purpose:** Provides tools for manipulating the formatting of input and output, such as setting precision for floating-point numbers.
 - **Key functions:** setw(), setprecision(), fixed.

5.2.2. CuckooHashTable Class

This is the core class implementing the cuckoo hashing algorithm.

- **Private Members:**
 - int* table1 & int* table2: Arrays representing two hash tables used for cuckoo hashing.
 - int m: The size of the hash tables (which is the same for both tables).

- `int size`: The current number of elements stored in the hash table.
- `const int EMPTY = -1`: Constant to represent an empty slot in the table.
- **Private Member Functions:**
 - **`int hash1(int key) const`:**
 - **Purpose**: Calculates the first hash value using modulo operation.
 - **Parameter**: `int key`: The key to hash.
 - **Returns**: The hash value (`key % m`).
 - **`int hash2(int key) const`:**
 - **Purpose**: Calculates the second hash value using XOR with a delta value and modulo operation.
 - **Parameter**: `int key`: The key to hash.
 - **Returns**: The second hash value.
 - **`void rehash()`:**
 - **Purpose**: Resizes the hash tables when the load factor exceeds a certain threshold (50% in this case).
 - **Parameters**: None.
 - **Behavior**: Doubles the size of the tables, rehashes the existing elements into the new tables, and adjusts the size variable.
- **Public Member Functions:**
 - **`CuckooHashTable(int initial_size)`:**
 - **Purpose**: Constructor to initialize the hash table with the given size.
 - **Parameter**: `int initial_size`: The initial size of the hash tables (must be a power of 2).
 - **`~CuckooHashTable()`:**
 - **Purpose**: Destructor to clean up the allocated memory for `table1` and `table2`.

- **Parameters:** None.
- **void insert(int key):**
 - **Purpose:** Inserts a key into the hash table, possibly evicting another key if needed (cuckooing).
 - **Parameter:** int key: The key to insert.
 - **Behavior:** Inserts the key, rehashes if necessary, and handles evictions in the two hash tables.
- **bool search(int key) const:**
 - **Purpose:** Searches for the key in both hash tables.
 - **Parameter:** int key: The key to search.
 - **Returns:** true if the key is found in either table, false otherwise.
- **void display():**
 - **Purpose:** Displays the current state of both hash tables.
 - **Parameters:** None.
 - **Behavior:** Outputs the indices and values of both tables, as well as the current load factor.
- **void checkPairwiseUncorrelation(int index1, int index2, int tableNum1, int tableNum2) const:**
 - **Purpose:** Checks if the variables at the given indices in the two specified tables are pairwise uncorrelated based on the hash values.
 - **Parameters:**
 - int index1: The first index to check.
 - int index2: The second index to check.
 - int tableNum1: The number of the first table (1 or 2).
 - int tableNum2: The number of the second table (1 or 2).

- **Behavior:** Computes the expectations and checks if the variables at the two indices are uncorrelated based on their hash values.

5.2.3. Main Function (main())

- **Purpose:** The entry point of the program.
- **Parameters:** None directly passed, but user input is used.
- **Key Behavior:**
 - Prompts the user to enter the initial size of the hash table (which must be a power of 2).
 - Inserts keys into the cuckoo hash table until the user enters -1.
 - Displays the hash table.
 - Prompts the user to enter two indices to check for pairwise uncorrelation.

OUR UNDERSTANDING AND METHOD OF SOLUTION:

Our understanding to this problem is similar to that of a toy box. Imagine you've got a big collection of toys: action figures, dolls, cars, and more. You want each toy to have its own special place on your shelves so that you can always find it when you want to play. To do this, you decide to label each toy with a unique code that tells you exactly where it should go. This is similar to how a hash table works, where each item gets a unique hash value that determines its spot.

But sometimes, two toys end up wanting the same spot. Let's say you have a robot and a teddy bear, and both of their codes tell you to put them in the same place on the shelf. This is what we call a collision in the hash table. So, how did we come up with a solution to this?

Role of Cuckoo Hashing

Enter cuckoo hashing, a clever way to handle these collisions. Think of it as having two special boxes for each toy. Each toy has a first-choice box and a second-choice box. When you first

place a toy on the shelf, you put it in its first-choice box. If that box is already taken, you simply move it to its second-choice box.

How It Works

To make this work, you need two keys, known as hash functions. Let's call them $h_1(x)$ and $h_2(x)$:

- **First Box ($h_1(x)$):** This is the toy's primary spot, determined by the first hash function $h_1(x)$. You check this spot first.
- **Second Box ($h_2(x)$):** If the first spot is already occupied, the toy moves to its second spot, determined by the second hash function $h_2(x)$.

The Role of XOR

But to make sure the second spot is always different from the first, we use a trick called XOR. XOR is like a mix that ensures even if two toys have the same first spot, their second spots will always be different. We create the second hash function like this:

$$h_2(x) = h_1(x) \oplus h_\Delta(x)$$

where $h_\Delta(x)$ is another hash function that adds a unique twist to $h_1(x)$.

Why It's Efficient

When a toy needs to move from one box to another, it does so swiftly, thanks to XOR. Here's how:

1. **Check the First Box:** You first check $h_1(x)$ to see if the toy's primary spot is free.
2. **Move to the Second Box:** If the first box is already taken, you instantly know the second spot using $h_2(x) = h_1(x) \oplus h_\Delta(x)$.

This makes the process very efficient, with no need for extra calculations or delays.

Making Sure It Works

Why we think this system is so effective:

1. **Distinct Boxes:** By using XOR, we ensure that $h_1(x)$ and $h_2(x)$ are always different. This guarantees that every toy has two unique spots to choose from.
2. **Correct Movement:** Every toy knows exactly where to go next, thanks to the simple and direct calculation using XOR.

3. **Independence:** In technical terms, we say this system is 2-independent. This means that the pairs of hash function values for different toys are independent of each other, ensuring smooth and trouble-free organization.

Ensuring Unique Spots

When you're organizing your toys, you don't want two toys to end up in the same box. To avoid this, we use a trick. First, we have our main key, called the first hash function. This is like the first box where each toy can go. But sometimes, two toys might have the same first box. To fix this, we use a second hash function, which gives us a different box for the toy.

The Trick: Using XOR

Now, how do we make sure the second box is always different from the first? We use a trick called XOR. Think of XOR as a special way to mix things up. If two toys have the same first box, XOR ensures that their second boxes will be different. Here's how it works:

Creating the Second Hash Function

We start with the first hash function, let's call it the first key. Then, we mix it with another hash function, which we'll call the mixing key. When we mix these two keys together, we get a new, unique key for the second box. This process makes sure that even if two toys have the same first box, their second boxes will be different because of the extra mix from the mixing key.

Practical Example

Let's say you have a toy car and a toy robot. Both of them have the same first box. When you apply XOR with the mixing key, the toy car might end up in Box A and Box C, while the toy robot might end up in Box A and Box D. The first boxes are the same, but the second boxes are different because of the XOR operation.

Benefits of This Method

1. **Avoiding Collisions:** By ensuring that each toy has two unique boxes, we avoid collisions. This makes it easier to find and place each toy without any confusion.
2. **Efficient Organization:** The XOR operation is quick and efficient. It doesn't take much time or effort to mix the keys and find the second box. This makes the whole process smooth and fast.

Rehashing: When the Shelves Get Full

Imagine you've been organizing your toys on your shelves, using the two special boxes for each toy (first and second labels). Over time, you keep adding more and more toys. Eventually, you notice that the shelves are getting full, and it's becoming harder to find an empty spot for a new toy. This situation calls for rehashing.

What Is Rehashing?

Rehashing is like realizing you need more shelves or a new strategy to keep your toy collection organized efficiently. It involves creating a larger or different set of shelves and moving all your toys to this new setup using new labels (hash functions). This process helps in reducing the chances of collisions and ensures that there's plenty of space for new toys.

The Rehashing Process

Step 1: Recognize the Need for Rehashing

You first notice that your shelves are too crowded. Every time you try to place a new toy, it's challenging to find an empty spot. This situation tells you it's time to rehash.

Step 2: Prepare New Shelves

You decide to get more shelves or rearrange your existing shelves to create more space. In terms of hash tables, this means creating a larger table or a new one with different dimensions.

Step 3: Create New Labels

You create new labels for each toy using new hash functions. These new labels help you decide where each toy should go on the new shelves. Think of this as finding a new primary box and a new secondary box for each toy.

Step 4: Move Each Toy

One by one, you take each toy from the old shelves and place it on the new shelves using the new labels. This ensures that each toy has a unique spot, reducing the chances of collisions.

Detailed Example

1. **Old Shelves:** You have been using the current shelves, and they are now too full.

2. **New Shelves:** You get larger shelves or additional ones.

3. **New Labels (Rehashing):**

- For the toy car, the old labels might be $h1_{old}(car)$ and $h2_{old}(car)$.
- For the toy robot, the old labels might be $h1_{old}(robot)$ and $h2_{old}(robot)$.

Now, you create new labels using new hash functions:

- For the toy car, the new labels become $h1_{new}(car)$ and $h2_{new}(car)$.
- For the toy robot, the new labels become $h1_{new}(robot)$ and $h2_{new}(robot)$.

4. **Moving Toys:** You move each toy to its new spot on the new shelves based on the new labels.

Benefits of Rehashing

1. **More Space:** By creating more shelves, you ensure there's ample space for all toys and reduce overcrowding.
2. **Fewer Collisions:** The new setup and new labels decrease the chances of toys ending up in the same spot.
3. **Efficient Organization:** With more space and better labels, placing and finding toys becomes easier and faster.

Proving Uncorrelation

Imagine you have a huge collection of toys that you want to organize on your shelves. Each toy has a unique label, and you have two special labels to help you place each toy: a first label and a second label. Think of these labels as being created by two different hash functions.

Using Two Labels

First, let's say the first label tells you where to place each toy on the first shelf. If a spot on the first shelf is already taken, you use the second label to find a spot on the second shelf. These two labels help you efficiently place your toys without any collisions.

Independent Labels

Now, to make sure everything works smoothly, we need to prove that the two labels (random variables) are uncorrelated. In simple terms, this means that the way one toy

is placed on the first shelf doesn't affect the placement of another toy on the second shelf. To do this, we need to show that the expected value of their combined positions equals the product of their individual expected positions.

Step-by-Step Explanation

Step 1: Define Our Labels

- Let's say the first label for each toy is determined by the first hash function h_1 .
- The second label is determined by the second hash function h_2 , which is created using XOR to mix things up.

Step 2: Expected Values

- The expected position of a toy on the first shelf using h_1 is $E[h_1(x)]$.
- The expected position of a toy on the second shelf using h_2 is $E[h_2(x)]$.

Step 3: Combined Expected Value

We need to show that the combined expected position for any two toys using both labels equals the product of their individual expected positions:

$$E[h_1(x) \times h_2(y)] = E[h_1(x)] E[h_2(x)]$$

Step 4: Independence of Labels

Since h_1 and h_2 are drawn from independent hash functions, their values don't interfere with each other. This independence ensures that the position determined by h_1 for one toy doesn't affect the position determined by h_2 for another toy.

Step 5: Proving Independence

To prove this, let's look at how we use XOR:

- The second label h_2 is created by mixing the first label h_1 with another hash function using XOR.
- XOR ensures that even if two toys have the same first label, their second labels will be different because of the extra mix from the second hash function.

Detailed Example

Imagine you have two toys: a toy car and a toy robot.

- The toy car has a first label $h1(car)$ and a second label $h2(car)$.
- The toy robot has a first label $h1(robot)$ and a second label $h2(robot)$.

To show these labels are uncorrelated:

1. **Toy Car:**

- First label $h1(car)$
- Second label $h2(car)=h1(car)\oplus h\Delta(car)$

2. **Toy Robot:**

- First label $h1(robot)$
- Second label $h2(robot)=h1(robot)\oplus h\Delta(robot)$

Since $h1$ and $h\Delta$ are independent, the labels for the toy car and toy robot don't interfere with each other. This means their combined expected value of positions equals the product of their individual expected values.

Summary

cuckoo hashing is like having two keys for each toy. It ensures no two toys fight over the same spot, and each toy moves smoothly between its two spots. The clever use of XOR ensures everything works perfectly, making cuckoo hashing a smart and efficient way to organize your collection.

RESULTS AND DISCUSSION

CHAPTER 6

RESULTS AND DISCUSSION

6.1 SCREENSHOTS

```
Enter the initial size of the hash table (must be a power of 2): 4
Enter keys to insert into the cuckoo hash table (enter -1 to stop):
Key: 34
Key: 54
Key: 78
Rehashing due to cycle ...
Key: 32
Key: -1

Table 1 :
Indices:    0    1    2    3    4    5    6    7
Values:    32    .   34    .    .    .   78    .

Table 2 :
Indices:    0    1    2    3    4    5    6    7
Values:     .    .    .    .    .   54    .    .

Current load factor: 0.50
Enter two indices to check pairwise uncorrelation : 1
2
2
5
E[X]: 2.00, E[Y]: 30.00, E[XY]: 60.00
The variables at indices are pairwise uncorrelated.
```

Fig 6.1 Output Screenshot

CONCLUSION

CHAPTER 7

7.1 APPLICATIONS

Cuckoo hashing is an advanced hash table technique that provides efficient space and time guarantees for insertion, deletion, and lookup operations. Its unique methods for resolving hash collisions make it effective for various real-time applications. Here are some real-time applications of cuckoo hashing:

- 1. Caching Systems:** Cuckoo hashing is used in caching mechanisms where fast retrieval of data is essential. Its predictable $O(1)$ lookup time makes it suitable for implementing high-performance caches in web applications, databases, and content delivery networks (CDNs).
- 2. Memory Management:** It's used in memory allocators to manage free blocks of memory efficiently. Cuckoo hashing allows dynamic allocation and deallocation with minimal fragmentation and fast lookups.
- 3. Networking Applications:** Cuckoo hashing can be employed in network routers for maintaining flow tables, NAT (Network Address Translation) tables, and firewall rules where quick packet lookups are necessary to ensure low-latency routing.
- 4. Database Indexing:** Cuckoo hashing is beneficial for high-performance databases that require rapid lookups and insertions. It can be used to implement secondary indexes or in-memory data structures for databases to provide fast query responses.
- 5. Distributed Systems:** In distributed hash tables (DHTs) used in peer-to-peer systems, cuckoo hashing can help in efficiently managing nodes and ensuring quick data retrieval regardless of data distribution across nodes.
- 6. Real-Time Analytics:** Cuckoo hashing can be applied in systems that require real-time data analysis, such as streaming analytics platforms that need to keep track of unique elements and their occurrences efficiently.
- 7. Mobile and Embedded Systems:** Devices with limited resources often use cuckoo hashing for efficient data storage and retrieval, especially when fast access times are crucial for user experience.
- 8. Blockchain and Cryptography Applications:** In blockchain applications, cuckoo hashing can be used for maintaining the state of transactions and ensuring efficient verification of addresses and digital signatures.
- 9. Game Development:** Game engines can utilize cuckoo hashing to manage game objects, assets, and states where performance is crucial for real-time rendering and interaction.

These applications benefit from cuckoo hashing's efficient handling of collisions and its guarantees of lookup times, making it well-suited for scenarios requiring high performance and speed.

7.2 CONCLUSION

Both problems showcase how carefully chosen hash functions simplify theoretical and practical challenges:

1. **CuckooHashing:**

The XOR-based design elegantly handles displacement and ensures distinct hash values for each key. This makes the implementation both efficient and robust against collisions.

2. **CountSketches:**

Independence properties of hash functions allow for variance simplification by ensuring pairwise uncorrelation. This highlights the mathematical foundations that enable efficient sketching algorithms.

The implementation is functional and provides a robust demonstration of **cuckoo hashing** with added enhancements like rehashing, dynamic resizing, and pairwise uncorrelation analysis.

APPENDIX

CHAPTER 8

8.1 CODE

```
#include <iostream>
#include <iomanip>
using namespace std;
class CuckooHashTable {
private:
    int* table1;
    int* table2;
    int m;
    int size;
    const int EMPTY = -1;

    int hash1(int key) const {
        return key % m;
    }
    int hash2(int key) const {
        int delta = 3;
        return (hash1(key) ^ delta) % m;
    }
    void rehash() {
        cout << "Rehashing due to cycle ..." << endl;
        int oldSize = m;
        m *= 2;
        int* oldTable1 = table1;
        int* oldTable2 = table2;
        table1 = new int[m];
        table2 = new int[m];
        for (int i = 0; i < m; i++) {
            table1[i] = EMPTY;
            table2[i] = EMPTY;
        }
        size = 0;
        for (int i = 0; i < oldSize; i++) {
            if (oldTable1[i] != EMPTY) {
                insert(oldTable1[i]);
            }
            if (oldTable2[i] != EMPTY) {
                insert(oldTable2[i]);
            }
        }
        delete[] oldTable1;
        delete[] oldTable2;
    }
public:
    CuckooHashTable(int initial_size) {
        if (initial_size <= 0 || (initial_size & (initial_size - 1)) != 0) {
            cout << "Error: Initial size must be a perfect positive power of two." << endl;
            exit(1);
        }
        m = initial_size;
    }
};
```

```

size = 0;
table1 = new int[m];
table2 = new int[m];
for (int i = 0; i < m; i++) {
    table1[i] = EMPTY;
    table2[i] = EMPTY;
}
}
~CuckooHashTable() {
    delete[] table1;
    delete[] table2;
}
void insert(int key) {
    if (size >= m / 2) {
        rehash();
    }
    int pos1 = hash1(key);
    if (table1[pos1] == EMPTY) {
        table1[pos1] = key;
        size++;
        return;
    }
    while (true) {
        int temp = table1[pos1];
        table1[pos1] = key;
        key = temp;
        pos1 = hash1(key);
        if (table1[pos1] == EMPTY) {
            table1[pos1] = key;
            size++;
            return;
        }
        int pos2 = hash2(key);
        temp = table2[pos2];
        table2[pos2] = key;
        key = temp;
        if (temp == EMPTY) {
            size++;
            return;
        }
    }
}
bool search(int key) const {
    return table1[hash1(key)] == key || table2[hash2(key)] == key;
}
void display() {
    cout << "\nTable 1 (Horizontal View):\n";
    cout << "Indices: ";
    for (int i = 0; i < m; i++) {
        cout << setw(6) << i;
    }
    cout << "\nValues: ";
    for (int i = 0; i < m; i++) {
        if (table1[i] != EMPTY)

```

```

        cout << setw(6) << table1[i];
    else
        cout << setw(6) << ".";
    }
    cout << "\n";
    cout << "\nTable 2 (Horizontal View):\n";
    cout << "Indices: ";
    for (int i = 0; i < m; i++) {
        cout << setw(6) << i;
    }
    cout << "\nValues: ";
    for (int i = 0; i < m; i++) {
        if (table2[i] != EMPTY)
            cout << setw(6) << table2[i];
        else
            cout << setw(6) << ".";
    }
    cout << "\n";

    cout << "\nCurrent load factor: " << fixed << setprecision(2) << (1.0 * size / m) <<
endl;
}

void checkPairwiseUncorrelation(int index1, int index2, int tableNum1, int tableNum2)
const {
    int X, Y;
    if (tableNum1 == 1 && index1 >= 0 && index1 < m && table1[index1] != EMPTY)
    {
        X = hash1(table1[index1]) * hash2(table1[index1]);
    } else if (tableNum1 == 2 && index1 >= 0 && index1 < m && table2[index1] !=
EMPTY) {
        X = hash1(table2[index1]) * hash2(table2[index1]);
    } else {
        cout << "Invalid index or table number for the first input." << endl;
        return;
    }
    if (tableNum2 == 1 && index2 >= 0 && index2 < m && table1[index2] != EMPTY)
    {
        Y = hash1(table1[index2]) * hash2(table1[index2]);
    } else if (tableNum2 == 2 && index2 >= 0 && index2 < m && table2[index2] !=
EMPTY) {
        Y = hash1(table2[index2]) * hash2(table2[index2]);
    } else {
        cout << "Invalid index or table number for the second input." << endl;
        return;
    }
    double expectationX = X;
    double expectationY = Y;
    double expectationXY = X * Y;
    cout << "E[X]: " << expectationX << ", E[Y]: " << expectationY << ", E[XY]: " <<
expectationXY << endl;
    if (expectationXY == expectationX * expectationY) {
        cout << "The variables at indices are pairwise uncorrelated." << endl;
    } else {
        cout << "The variables at indices are not pairwise uncorrelated." << endl;
    }
}

```

```

    }
}
};
int main() {
    int initial_size;
    cout << "Enter the initial size of the hash table (must be a power of 2): ";
    cin >> initial_size;
    if (initial_size <= 0 || (initial_size & (initial_size - 1)) != 0) {
        cout << "Error: Initial size must be a perfect positive power of two." << endl;
        return 1;
    }
    CuckooHashTable table(initial_size);
    cout << "Enter keys to insert into the cuckoo hash table (enter -1 to stop):" << endl;
    while (true) {
        int key;
        cout << "Key: ";
        cin >> key;
        if (key == -1) break;
        table.insert(key);
    }
    table.display();
    int index1, index2, tableNum1, tableNum2;
    cout << "Enter two indices to check pairwise uncorrelation (first table and first table index then followed by second table and second table index): ";
    cin >> tableNum1 >> index1 >> tableNum2 >> index2;
    table.checkPairwiseUncorrelation(index1, index2, tableNum1, tableNum2);
    return 0;
}

```

REFERENCES

CHAPTER 9

REFERENCES

- [1] **Data Structures and Algorithm Analysis in C++** by Mark Allen Weiss
- [2] **Algorithms and Data Structures: The Basic Toolbox** by Mehlhorn and Sanders.
- [3] **Introduction to Algorithms** by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein (CLRS)
- [4] **Handbook of Data Structures and Applications** by Dinesh P. Mehta and Sartaj Sahni .
- [5] *Cuckoo Hashing* by Rasmus Pagh and Flemming Friche RodlerI.
- [6] **Advanced Data Structures** by Peter BrassR.
- [7] **C++ Primer** by Stanley B. Lippman, Josée Lajoie, and Barbara E. MooM.-
- [8] **The C++ Standard Library: A Tutorial and Reference** by Nicolai M.
- [9] **Foundations of Data Science** by Avrim Blum, John Hopcroft, and Ravindran KannanE.
- [10] **Algorithms and Data Structures: The Basic Toolbox Authors:** Kurt Mehlhorn and Peter Sanders