

**A Course End Project Report**

**On**

**CUCKOO HASHING**

**Adavnce datat structures(A8513)**

Submitted in the Partial Fulfilment of the Requirements

for the Award of the Degree of

**Bachelor of Technology**

**IN**

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

**Submitted By**

**AERRA SINDHU 23885A0501**

**Under the Esteemed Guidance of Dr. Shanthi Makka**

**Professor CSE**

**Department of Computer Science and Engineering**

**VARDHAMAN COLLEGE OF ENGINEERING, HYDERABAD**

(AUTONOMOUS)

Affiliated to JNTUH, Approved by AICTE, Accredited by NAAC with A++ Grade, ISO 9001:2015 Certified

**Kacharam, Shamshabad, Hyderabad – 501218, Telangana, India**

**January, 2024**

**VARDHAMAN COLLEGE OF ENGINEERING, HYDERABAD**

An autonomous institute affiliated to JNTUH

**Department of Computer Science and Engineering**

**CERTIFICATE**

This is to certify that the Course End Project titled **“Advance data structures” is** carried out by Mr./Ms.**A.SINDHU**, Roll Number **23885A0501**towards **A8602 – Object Oriented Programming Laboratory** course and submitted to **Department of Computer Science and Engineering**, in partial fulfilment of the requirements for the award of degree of **Bachelor of Technology** in **Department of Computer Science and Engineering** during the Academic year 2023-24.

**Name & Signature of the Instructor**

## Dr. Shanthi Makka Professor, CSE

**Name & Signature of the HOD**

## Dr. Ramesh Karnati HOD, CSE



**ACKNOWLEDGEMENT**

The satisfaction that accompanies the successful completion of the task would be put incomplete without the mention of the people who made it possible, whose constant guidance and encouragement crown all the efforts with success.

We wish to express our deep sense of gratitude to **Dr. M .Naresh Goud**, Professor, Department of Computer Science and Engineering, Vardhaman College of Engineering, for her able guidance and useful suggestions, which helped us in completing the design part of potential project in time.

We particularly thankful to **Dr. Ramesh Karnati**, Associate Professor & Head, Department of Computer Science and Engineering for his guidance, intense support and encouragement, which helped us to mould our project into a successful one.

We show gratitude to our honorable Principal **Dr. J.V.R.Ravindra**, for having provided all the facilities and support.

We avail this opportunity to express our deep sense of gratitude and heartfelt thanks to **Dr. Teegala Vijender Reddy**, Chairman and **Sri Teegala Upender Reddy**, Secretary of VCE, for providing a congenial atmosphere to complete this project successfully.

We also thank all the staff members of Computer Science and Engineering for their valuable support and generous advice. Finally, thanks to all our friends and family members for their continuous support and enthusiastic help.

## AERRA SINDHU

## – 23885A0501



***INDEX***

1. Introduction…………………………………………………………………………….5
2. Objective of the Project…………………………………………………………………6
3. Problem statement 7
4. Software and hardware requirements 9
5. Project Description 10
6. Flowchart/Algorithm/Procedure 11
7. Code 18
8. Result(s) 20
9. Conclusion and Future work 21
10. References……………………………………………………………………………..22



# INTRODUCTION:

Cuckoo Hashing is a dynamic hashing technique designed to efficiently handle collisions in hash tables. Traditional hash tables often encounter performance bottlenecks when collisions occur frequently, leading to degradation in search and insertion times. Cuckoo Hashing addresses this challenge by providing a simple yet effective mechanism to resolve collisions.

In Cuckoo Hashing, two hash functions are employed to map keys to two different tables. When a collision occurs in either table, the existing item is displaced to its alternative location in the other table. This process is repeated until all items find a stable position, or a predetermined threshold of displacement attempts is reached. The simplicity of the displacement mechanism makes Cuckoo Hashing easy to implement and guarantees constant time complexity for both insertion and search operations in the average case.

This abstract explores the key concepts of Cuckoo Hashing, its advantages, and potential challenges. We delve into the design considerations, addressing load factor management, hash function selection, and cycle detection strategies. Additionally, we discuss practical applications and scenarios where Cuckoo Hashing proves to be a viable and efficient solution for handling collisions in hash tables.

Cuckoo Hashing offers a balance between simplicity and performance, making it an attractive choice for scenarios where real-time constraints and minimal overhead are crucial. Understanding its underlying principles and trade-offs provides valuable insights for developers and researchers working on scalable and efficient data storage solutions.

# OBJECCTIVE OF THE PROJECT:

Objective of the Project:

**1.Constant-Time Operations:**

--Lookup: Achieve constant-time worst-case lookup complexity to ensure efficient retrieval of stored elements.

--Insertion: Aim for constant-time worst-case insertion complexity for adding new elements to the hash table.

**2.Collision Resolution:** Implement a mechanism to resolve collisions by using multiple hash functions and multiple hash tables.

Choose a suitable algorithm for relocating elements when a collision occurs. In cuckoo hashing, this involves swapping elements between different hash tables until a suitable position is found.

**3.Load Factor Management:** Maintain an appropriate load factor to balance the efficiency of the hash table. A higher load factor may lead to more frequent rehashing and increased complexity.

Implement strategies for dynamic resizing or rehashing to adjust the size of the hash table based on the number of elements.

**4.Hash Function Design:** Design effective hash functions that distribute keys uniformly across the hash tables, reducing the likelihood of collisions.

Use a sufficient number of hash functions to minimize the chances of collisions and improve the overall performance.

**5.Memory Efficiency:** Optimize memory usage by minimizing the overhead associated with maintaining multiple hash tables.

Explore techniques to reduce the space complexity of the data structure.

**4.DATA STRUCTURE USED:**

**1.Hash Tables:** Cuckoo hashing uses two or more hash tables. The number of tables is usually denoted as K.Each table is an array where elements are stored based on the output of different hash functions.

**2.Hash Functions:**Multiple hash functions are employed to determine the position of an element in each hash table**.**The choice of hash functions is critical to achieving a uniform distribution of keys across the hash tables, minimizing the likelihood of collisions

**3.Key Storage:**Keys are stored directly in the hash tables rather than in linked structures (as in chaining).Each key has multiple possible locations across the hash tables, determined by the hash functions.

**4.Load Factor:**The load factor is the ratio of the number of stored elements to the total number of slots in all hash tables.Managing the load factor is crucial for balancing efficiency and memory usage. High load factors may lead to more frequent rehashing.

# PROBLEM STATEMENT:

Develop an efficient Cuckoo hashing algorithm to handle collisions in a hash table. The goal is to achieve constant-time worst-case lookup and insertion operations while maintaining a low rate of displacements

In Cuckoo Hashing, two hash functions are employed to map keys to two different tables. When a collision occurs in either table, the existing item is displaced to its alternative location in the other table. This process is repeated until all items find a stable position, or a predetermined threshold of displacement attempts is reached. The simplicity of the displacement mechanism makes Cuckoo Hashing easy to implement and guarantees constant time complexity for both insertion and search operations in the average case.

This abstract explores the key concepts of Cuckoo Hashing, its advantages, and potential challenges. We delve into the design considerations, addressing load factor management, hash function selection, and cycle detection strategies. Additionally, we discuss practical applications and scenarios where Cuckoo Hashing proves to be a viable and efficient solution for handling collisions in hash tables.

# SOFTWARE AND HARDWARE REQUIREMENTS

# Software Requirements:

# Operating System:

**1.Hash Tables:**

Cuckoo hashing uses two or more hash tables. The number of tables is usually denoted as K.Each table is an array where elements are stored based on the output of different hash functions.

**2.Hash Functions:**

Multiple hash functions are employed to determine the position of an element in each hash table**.**The choice of hash functions is critical to achieving a uniform distribution of keys across the hash tables, minimizing the likelihood of collisions

**3.Key Storage:**

Keys are stored directly in the hash tables rather than in linked structures (as in chaining).Each key has multiple possible locations across the hash tables, determined by the hash functions.

**4.Load Factor:**

The load factor is the ratio of the number of stored elements to the total number of slots in all hash tables.Managing the load factor is crucial for balancing efficiency and memory usage. High load factors may lead to more frequent rehashing.

5.,**Operating System:**

* + The server-side should run on a stable and secure operating system. Linux distributions like Ubuntu Server, CentOS, or Red Hat are commonly used.
  + Client-side applications can be designed to be platform-independent, accessible through web browsers. Ensure compatibility with major browsers like Chrome, Firefox, Safari, and Edge

# **Hardware Requirements:**

**Server Hardware:**

* + Powerful server hardware to handle concurrent user connections and database transactions.
  + Sufficient storage space for the database and backups.

**Network Infrastructure:**

* + High-speed and reliable internet connectivity to handle the traffic during the voting period.
  + Load balancing and redundancy to ensure system availability.

**Firewalls and Security Appliances:**

* + Install firewalls and security appliances to protect against unauthorized access and cyber threats.

**Client Devices**

**Backup and Recovery Systems:**

* + Regularly back up the database and other critical data. Implement a disaster recovery plan in case of system failures.

**Hardware Security Modules (HSMs):**

* + CONSIDER using HSMs for securing sensitive cryptographic operations, such as key management and digital signatures.

**Scalability:**

* + Design the system to be scalable to accommodate an increasing number of voters and transactions.

**Physical Security:**

* + Ensure physical security for server rooms and data centers to prevent unauthorized access.



# PROJECT DESCRIPTION:

1. **Hash Functions:** Two hash functions, h1 and h2, are chosen.
2. **Hash Tables:** Two separate hash tables, T1 and T2, are created.
3. **Insertion:**
   * When inserting a key, first compute its hash using both h1 and h2.
   * If the slot in T1 designated by h1 is empty, place the key there.
   * If the slot in T1 designated by h1 is occupied, replace the existing key with the new key, and insert the displaced key into T2 using h2.
   * If the slot in T2 designated by h2 is occupied, replace the existing key, insert the displaced key into T1 using h1, and repeat the process until an empty slot is found.
4. **Lookup:**
   * When searching for a key, compute its hash using both h1 and h2.
   * Check both tables at the corresponding positions.
   * If the key is found in either table, the search is successful.
5. **Deletion:**
   * To delete a key, find its position using both hash functions.
   * If the key is in T1, remove it from T1. If it's in T2, remove it from T2.

The term "cuckoo" comes from the behavior of cuckoo birds, which are known for laying their eggs in the nests of other birds, forcing the host birds to raise the cuckoo chicks. Similarly, in cuckoo hashing, keys may be "relocated" from one table to another to resolve collisions.

Cuckoo hashing guarantees constant time for lookup operations, assuming there are no cycles in the table dependencies. However, it may enter into an infinite loop during insertion if there are cycles, so there needs to be a mechanism to handle such situations, such as resizing the hash tables or rehashing.

# FLOW CHART:

**5.ALGORITHM :**

Cuckoo hashing involves a series of steps for insertion, lookup, and resizing. Below is a high-level algorithm for cuckoo hashing:

**1.Initialization:**

>Choose the number of hash tables k and the size of each table.

>Initialize an array for each hash table, each with the chosen size.

>Initialize multiple hash functions, one for each table.

**2.Insertion:**

>Hash the key using each hash function to determine its possible locations in each hash table.

>If any of the positions is empty, insert the key into that position.

>If all positions are occupied, choose one of the occupied positions and evict the element currently there.

>Recursively try to insert the evicted key into its alternative position in the other hash table.

>If successful, the evicted key is inserted, and the process is complete.

>if the recursion reaches a maximum displacement limit, resize the hash tables and retry the insertion.

**3.Lookup:**

>Hash the key using each hash function to determine its possible locations in each hash table.

>Check each possible location for the presence of the key.

>If the key is found in any of the positions, return that the key is present.

>If the key is not found, return that the key is not in the hash tables.

**4.Resizing:**

>When the load factor exceeds a certain threshold, choose new hash functions and resize the hash tables.

>Create new hash tables with an increased size.

>Recompute hash values for all existing keys and insert them into the new hash tables.

>If insertion fails due to continuous displacement attempts, consider further increasing the size of the hash tables or rehashing with different functions.



**6.CODE**

#include <stdio.h>

#include <stdlib.h>

#define TABLE\_SIZE 10

#define K 2 // Number of hash tables

#define MAX\_RETRIES 10

// Hash function 1unsigned int hash\_function\_1(int key) {

return key % TABLE\_SIZE;

}

// Hash function 2

unsigned int hash\_function\_2(int key) {

return (key / TABLE\_SIZE) % TABLE\_SIZE;

}

// Initialize hash tables

int table[K][TABLE\_SIZE];

// Function to insert a key into the hash tables

int insert(int key) {

int retries = 0;

while (retries < MAX\_RETRIES) {

for (int i = 0; i < K; i++) {

// Calculate hash positions using two hash functions

 unsigned int hash\_pos\_1 = hash\_function\_1(key);

unsigned int hash\_pos\_2 = hash\_function\_2(key);

if (table[i][hash\_pos\_1] == 0) {

// If the position in the current table is empty, insert the key

table[i][hash\_pos\_1] = key;

return 1; // Success

} else {

// Evict the existing key and insert the new key

int evicted\_key = table[i][hash\_pos\_1];

table[i][hash\_pos\_1] = key;

// Try to insert the evicted key into the other table

key = evicted\_key;

}

}

retries++;

}

// Resizing needed if maximum retries reached

return 0; // Failure, need resizing

}

// Function to check if a key is present in the hash tables

int lookup(int key) {

for (int i = 0; i < K; i++) {

unsigned int hash\_pos\_1 = hash\_function\_1(key);

if (table[i][hash\_pos\_1] == key) {

printf("Key %d found in table %d\n", key, i);

return 1; // Key found

 }

}

printf("Key %d not found\n", key);

return 0; // Key not found

}

// Function to print the hash tables

void print\_tables() {

for (int i = 0; i < K; i++) {

printf("Table %d: ", i);

for (int j = 0; j < TABLE\_SIZE; j++) {

printf("%d\t", table[i][j]);

}

printf("\n");

}

}

int main() {

// Initialize hash tables

for (int i = 0; i < K; i++) {

for (int j = 0; j < TABLE\_SIZE; j++) {

table[i][j] = 0;

}

}

// Insert keys

insert(10);

insert(20);

insert(30);

insert(25);



insert(35);

// Print hash tables

printf("After Insertion:\n");

print\_tables();

// Lookup keys

lookup(20);

lookup(25);

lookup(40);

return 0;

}

**OUTPUT:**

After Insertion:

Table 0: 30 0 0 0 0 35 0 0 0 0

Table 1: 20 0 0 0 0 25 0 0 0 0

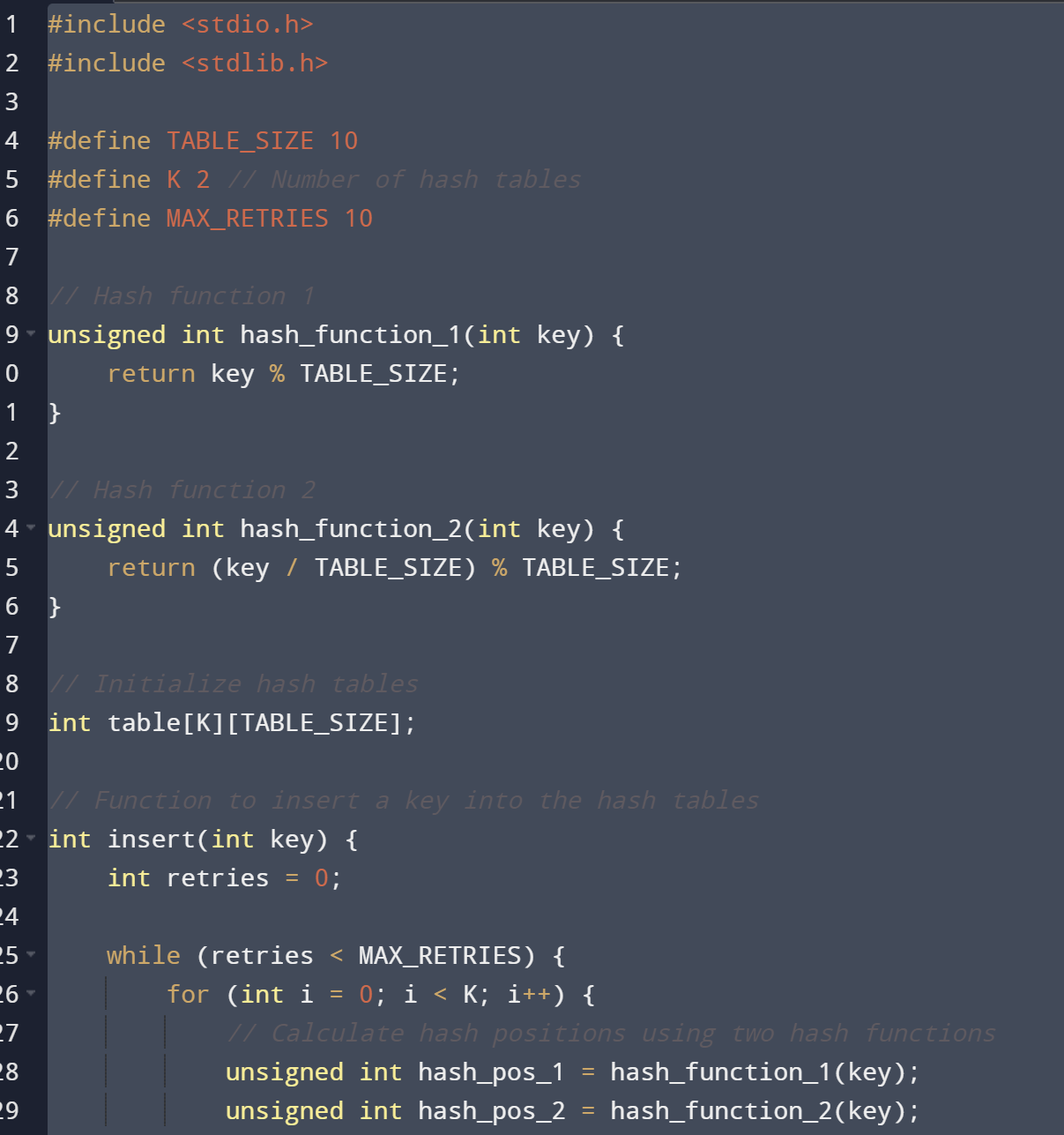
Key 20 found in table 1

Key 25 found in table 1

Key 40 not found

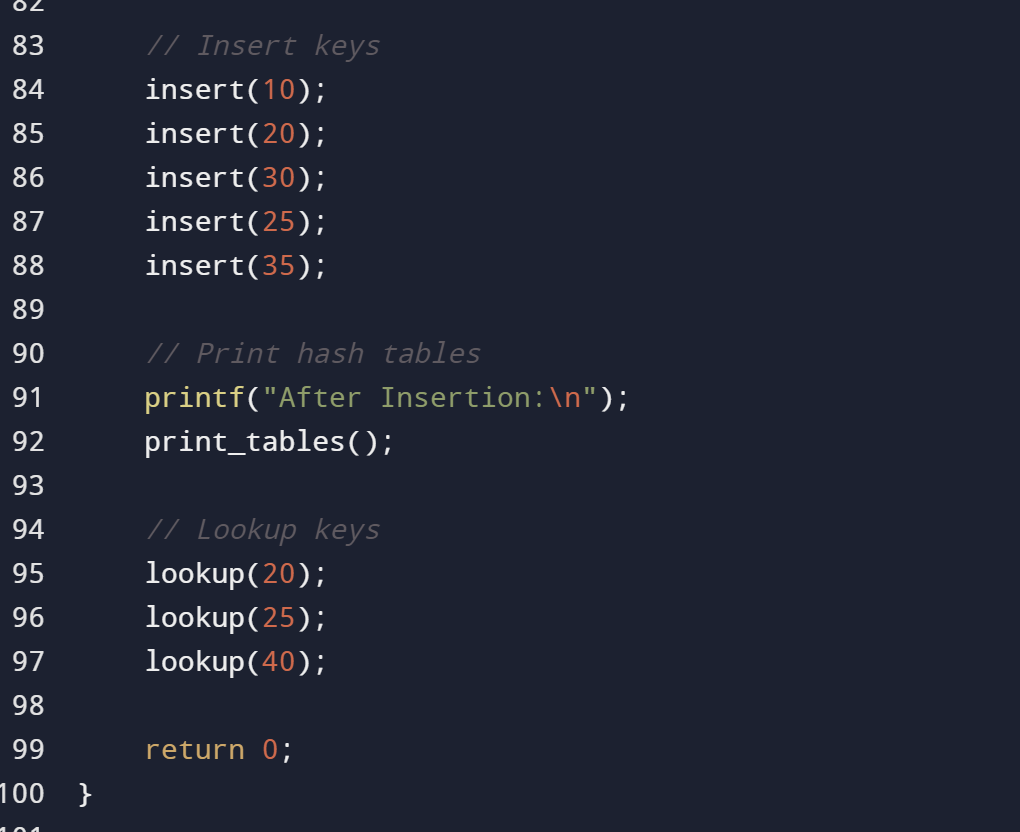


**8.SCREENSHOTS:**

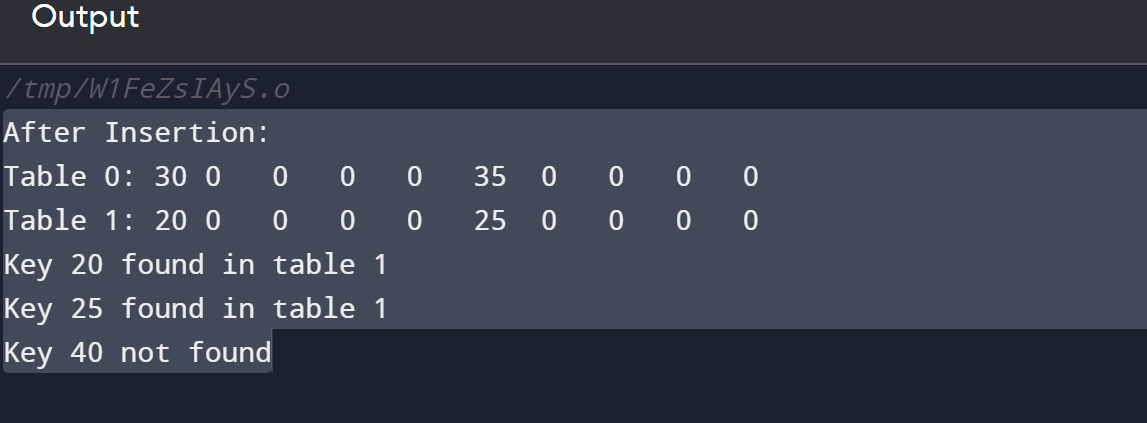








**Output:**





**9.CONCLUSION**

In conclusion, Cuckoo hashing is a hash table-based data structure that addresses collisions by using multiple hash functions and multiple hash tables. The primary goals of Cuckoo hashing include achieving constant-time worst-case lookup and insertion operations. The algorithm relies on a displacement strategy, where existing elements are moved to alternative positions in the hash tables when collisions occur. The fault tolerance mechanism allows a limited number of displacements during insertion, and dynamic resizing is performed when necessary to maintain a balanced load factor. Cuckoo hashing offers a theoretical constant-time worst-case performance, but its practical efficiency may depend on factors such as hash function design, memory access patterns, and the handling of edge cases. Overall, Cuckoo hashing provides an interesting approach to collision resolution in hash tables, combining simplicity with the potential for efficient worst-case performance under certain conditions.



**REFERENCE:**

1. [*Pagh, Rasmus*](https://en.wikipedia.org/wiki/Rasmus_Pagh); Rodler, Flemming Friche (2001). "Cuckoo Hashing". Algorithms — ESA 2001. Lecture Notes in Computer Science. Vol. 2161. [*CiteSeerX*](https://en.wikipedia.org/wiki/CiteSeerX_(identifier)) [*10.1.1.25.4189*](https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.25.4189). [*doi*](https://en.wikipedia.org/wiki/Doi_(identifier)):[*10.1007/3-540-44676-1\_10*](https://doi.org/10.1007%2F3-540-44676-1_10). [*ISBN*](https://en.wikipedia.org/wiki/ISBN_(identifier)) [*978-3-540-42493-2*](https://en.wikipedia.org/wiki/Special:BookSources/978-3-540-42493-2).
2. [**^**](https://en.wikipedia.org/wiki/Cuckoo_hashing#cite_ref-2) [*"ESA - European Symposium on Algorithms: ESA Test-of-Time Award 2020"*](https://web.archive.org/web/20210522135306/http:/esa-symposium.org/test-of-time-award.php). esa-symposium.org. Award committee: [*Uri Zwick*](https://en.wikipedia.org/wiki/Uri_Zwick), [*Samir Khuller*](https://en.wikipedia.org/wiki/Samir_Khuller), [*Edith Cohen*](https://en.wikipedia.org/wiki/Edith_Cohen). Archived from [*the original*](http://esa-symposium.org/test-of-time-award.php) on 2021-05-22*. Retrieved 2021-05-22*.
3. [**^**](https://en.wikipedia.org/wiki/Cuckoo_hashing#cite_ref-3) Kutzelnigg, Reinhard (2006). [*Bipartite random graphs and cuckoo hashing*](https://www.dmtcs.org/dmtcs-ojs/index.php/proceedings/article/viewFile/dmAG0133/1710.pdf) *(PDF)*. Fourth Colloquium on Mathematics and Computer Science. Discrete Mathematics and Theoretical Computer Science. Vol. AG. pp. 403–406.
4. [**^**](https://en.wikipedia.org/wiki/Cuckoo_hashing#cite_ref-4) Cohen, Jeffrey S., and Daniel M. Kane. "Bounds on the independence required for cuckoo hashing." ACM Transactions on Algorithms (2009).
5. [**^**](https://en.wikipedia.org/wiki/Cuckoo_hashing#cite_ref-5) Pǎtraşcu, Mihai, and Mikkel Thorup. "The power of simple tabulation hashing." Journal of the ACM (JACM) 59.3 (2012): 1-50.
6. [**^**](https://en.wikipedia.org/wiki/Cuckoo_hashing#cite_ref-6) Aumüller, Martin, Martin Dietzfelbinger, and Philipp Woelfel. "Explicit and efficient hash families suffice for cuckoo hashing with a stash." Algorithmica 70.3 (2014): 428-456.
7. ^ [Jump up to:***a***](https://en.wikipedia.org/wiki/Cuckoo_hashing#cite_ref-mitzenmacher2009survey_7-0) [***b***](https://en.wikipedia.org/wiki/Cuckoo_hashing#cite_ref-mitzenmacher2009survey_7-1) *[Mitzenmacher, Michael](https://en.wikipedia.org/wiki/Michael_Mitzenmacher" \o "Michael Mitzenmacher)* (2009-09-09). [*"Some Open Questions Related to Cuckoo Hashing"*](http://www.eecs.harvard.edu/~michaelm/postscripts/esa2009.pdf) *(PDF)*. Proceedings of ESA 2009*. Retrieved 2010-11-10*.
8. [**^**](https://en.wikipedia.org/wiki/Cuckoo_hashing#cite_ref-8) Dietzfelbinger, Martin; Weidling, Christoph (2007). [*"Balanced allocation and dictionaries with tightly packed constant size bins"*](https://doi.org/10.1016%2Fj.tcs.2007.02.054). Theoret. Comput. Sci. **380** (1–2): 47–68. [*doi*](https://en.wikipedia.org/wiki/Doi_(identifier)):[*10.1016/j.tcs.2007.02.054*](https://doi.org/10.1016%2Fj.tcs.2007.02.054). [*MR*](https://en.wikipedia.org/wiki/MR_(identifier)) [*2330641*](https://mathscinet.ams.org/mathscinet-getitem?mr=2330641).