2021 16th International Conference on Electronics Computer and Computation (ICECCO) | 978-1-6654-0945-2/21/$31.00 ©2021 IEEE | DOI: 10.1109/ICECCO53203.2021.9663778

Evaluation of Collision Resolution Methods Using Asymptotic Analysis

Ahmed Dalhatu Yusuf[1], Saleh Abdullahi[2], Moussa Mahamat Boukar[3] and Salisu Ibrahim Yusuf[4]

Nigerian Communications Commission[1]
Department of Computer Science, Nile University of Nigeria[2,3,4]

dalhatuahmed@gmail.com[1], saleh.abdullahi@nileuniversity.edu.ng[2],
musa.muhammed@nileuniversity.edu.ng[3], salisu.ibrahim.yusuf@gmail.com[4]

***Abstract*—The basic idea of hashing is to use a hash function $h(k)$ to reduce a giant universe to a reasonably small table such that $U = \{0, 1, .., |U| - 1\} \rightarrow T\{0, 1, ..., |T| - 1\}$ with $Pr_h\{h(key) \rightarrow T\} = 1/|K|$ which is independent of $h(y)$ for all $x \quad y \in U$. Hashing has been an efficient implicit and explicit search technique for retrieving an element from a collection for many years. Collision is inevitable in a high load factor environment, thus several algorithms for resolving collision to find an alternative slot for a key x in a hash table have been presented. One of the important property of these algorithms is running time complexity. Here, we analysed the performance of these algorithms based on runtime analysis for retrieving and inserting a key using asymptotic analysis. We discovered that many of these algorithms have a non constant insertion and retrieval time. Furthermore, only one algorithm was discovered to have an exact constant time for both lookup and insertion of any arbitrary key into a hash table.**

Keywords—Hashing; algorithm; lookup; insertion; collision resolution; hash table; hash function; slot

I. INTRODUCTION

Data structure refers to the process of organising a collection of data. The primary goals of the data structure are space and time complexity [10], [11], [1]. Several data structures exist for insertion, searching, and sorting data. Searching is a process of checking an item with a precise attribute from a collection of items [2]. Searching strategies include *linear* search, which goes sequentially through the list until an element is found or the entire list has been checked [26], it is time complexity is $O(n)$, *binary* search, at every step it cuts the search space into two [9], it is time complexity is $O(log\ n)$, *hashing*, maps an item into a bucket in a hash table [9], [12], [10] to achieve $O(1)$ time complexity to find and insert an item. Hash table uses an array to store a collection of items. An *i* item is stored in a slot *h(k)*, where *h(k)* is a hash function that computes the slot index of the item and maps an item into a slot with a constraint that each slot has (1) item. Hash function demonstrates efficient data operations such as insertion, retrieval, or matching in many areas in computing [8], [13], [4] and also a research has shown that a widely use python dictionary was implemented using an open address hashing technique [19]

To achieve a constant access time for insertion and lookup researchers have discovered techniques for handling collision condition in a hash table these include, multi-indexes hashing, cuckoo hashing, separate chaining hashing, etc. This condition exist when dissimilar keys have the same hash value.

The work of Ahmed *et al* [7] was conducted based on the time complexity informed by the algorithm designers, other studies focuses on presenting applications of hashing in areas of computing [8], [23], [13], and a number of other works on security hash functions [24]. Here, we analysed the runtime complexity of the existing techniques using asymptotic analysis. We also provide a suggestion of what researchers should do to improve the efficiency of hashing.

II. RELATED WORK

In this section, we presented several research works that are conducted on hashing techniques to ensure effective data operation, like insert, retrieve, and delete an element from a collection of an element in a hash table. The review also includes other works that are relevant to the concept of this research.

A. Data Structure

To deal with the multifaceted nature of issues and the critical thinking process, computer researchers ”use abstractions” to permit them to concentrate on the main problem without losing all sense of direction in the subtleties [6]. By making models of the difficult area, we can use a superior and increasingly productive critical thinking process. These models permit us to portray the information that our calculations will control in a significantly more reliable manner as for the difficulty itself.

Abstraction refers to hiding detail of a method and permits users to see and use it at high level [28]. An abstract data type is also bellowed an ADT, is a coherent depiction of how we see the information and the tasks that are permitted regardless of how they will be actualized. This implies we are concerned uniquely with what the information appears to us and not with how in the long run be built. By giving this degree of reflection, we are making an epitome around the information. The thought is that by typifying the subtleties of the execution, we are concealing them from the client's view. This is called data covering up. The process of implementing abstract data

978-1-1-6654-0945-2/21/$31.00 ©2021 IEEE

Authorized licensed use limited to: Wikipedia. Downloaded on January 20,2025 at 09:29:27 UTC from IEEE Xplore. Restrictions apply.

type is called data structure, which necessitates that we give a description/attributes and method of the information using built-in data types of programming [9]. Therefore, with this technique problem could be solved efficiently.

B. Analysis of Algorithms

Algorithms are commonly composed for taking care of certain issues or instrument through machines, the calculations might be a few in numbers, further to these the productivity of the created calculations for the said issue should be evaluated: the components which are to be evaluated are time unpredictability, space multifaceted nature, managerial expense and quicker usage, etc. One of the successful strategies for contemplating the proficiency of an algorithm is ”$Big - O$ notations” [2].

However the ”Big_O notation” is containing numerical capacities and their examination bit by bit, it represents the procedure of estimating the intricacy factor. The yield is constantly expected as a smooth line or bends with a littler and static incline. This notation example gives the tight upper bound of the given method. Demonstrated, as $f\ (n) = O(g(n))$. That implies, at bigger estimations of n, the upper bound of $f(n)$ is $g(n)$ is shown in fig. 1.

Similarly, for lower bounding, omega Ω notation is used to

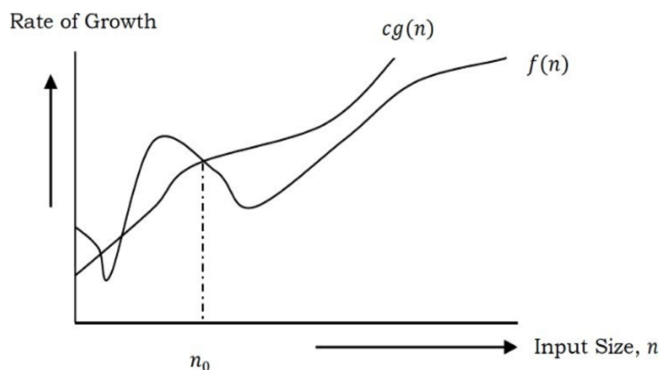

Fig. 1. Big O Notation (Upper Bounding Function)

demonstrate that. for example, gives the tight lower bound of the given method. Demonstrated, as $f\ (n) = \Omega(g(n))$. That implies, at bigger estimations of n, the lower bound of $f(n)$ is $g(n)$ [2].

C. Survey on Hashing Technique

Ahmed *et al* [7] carried out a review on different collision resolution techniques in a hash table. Highlighting the hash function employed in each method, how key is hashed into a hash table, key retrieval strategies and costs based on worse-case runtime complexity, alongside problems associated with each existing technique and we also provided a research gap in hashing technique for researchers to improve on. However, the work was done based on complexity identified by the algorithm designers.

The research effort of Lianhua *et al* [8] proffered the main ideas on the prevailing hashing techniques for various ”data and applications”. The work was conducted due to an increase in the volume of data generated every day from diverse areas such as social network activities, daily transactions in the business domain, data from IoT applications and other numerous domains. This increment of data has led to significant issues in analysing and processing data and hashing strategy has been an efficient approach for fast data access for decennaries. The techniques and applications reviewed in their work have shown the positive impact hashing has on the performance of various applications such as networking, image classification, text classification and thus makes it an interesting area of study in order to make real-world applications very efficient.

The work of Tom [20] Indicated that several algorithms are implemented using dictionary complex type that most high-level programming comes with. This dictionary type could be implemented in a number of ways with a different data structure. However, a study has shown that for better lookup performance it should be implemented with a hash table. Python dictionary takes advantage of that, it uses a hash table with open addressing. But the problem with that is whenever a collision occurs probing method has to happen. Therefore, In an environment where the collision is high then the lookup performance reduces. The trivial method is to use chained a hash for dictionary implementation.

D. Data-Situated Hashing

Data-situated hashing refers to a technique used in speeding-up the time for retrieving or comparing data, where a hash table content data used for query feedback.

Yuanyuan *et al* [5] improved upon open address cuckoo hashing by overcoming the problem of an infinite loop at a time of insertion, which reduces the efficiency of query processing. They proposed a better technique called SmartCuckoo, which presents the relationship of hashing using directed pseudoforest and uses it subsequently to indict element placement for the correct determination of the existence of an infinite loop. SmartCuckoo can also predict insertion failure without going through a number of probing steps. The work has been implemented on a ”cloud storage system” the source code has been released for public users.

Peter *et al* [15] presented an approach for resolving collision in one-dimensional array. The technique concatenated (dot (.)) the key and the *h(k)* and insert in the first empty bucket in the hash table. For example, in a hash table of *size* 7, *h(23) = 23* mod *7*, will be placed 2.23 at the first available cell in the hash table. Searching an element in this technique is linear *O*(n), hence the *h(k)* will not help in locating the key from the hash table.

Abhay [23] presented a technique that minimised the memory space used in implementing the hash table by compressing the key for data-item in the hash table. The hash value $h(key)$ can be generated by any hash function and subsequently, the compressed value is generated to mapped

Authorized licensed use limited to: Wikipedia. Downloaded on January 20,2025 at 09:29:27 UTC from IEEE Xplore. Restrictions apply.

the input key.

Sailesh *et al* [13] proposed a technique that guarantees constant time for retrieving keys at high load factor settings and memory minify bandwidth in high-performance networking subsystem. The method uses a hash table with a number of multiple logical chunks given a $key\ n$ likely slot in memory. An item will be mapped with $h(k)$, which inserted the key in the search space U in the scope of the chunked subscripts of the hash tables. i.e $h: U = \{0, 1, ..., |U-1\}$, where $|U|$ is the length of the chunked hash table. An item can be inserted in a bucket $h(k)$ in any of the chunked hash tables. In the event where all the chunked buckets for *h(k)* are not empty then a collision is inevitable.

E. Security-Situated Hashing

Security-situated hashing is a verification and validation technique that uses hashing technique to verify and validate whether a data has been altered by a third-party.

Randomized hashing was proposed by Shai *et al* [14] as a process that takes a message and return a hash value of the message that can be used in digital signature without any modification in traditional hash function such as SHA. The objective of their work is to free "digital signature schemes" from their dependence on collision contention.

F. Collision Resolution Strategies

The circumstance where a hash value calculated using a hash function matches with another hash value that is already involved within the hash table is named as collision [25]. To place all the colliding keys in the hash table that could be achieved using a method called collision resolution. Collision resolution refers to a situation when two items hash to the same slot, and a systematic method must be used to insert the second item in another slot in the hash table [17].

III. Analysis of Some of the Important Reviewed Algorithms

Here, we provided asymptotic analysis of the existing hashing technique based on runtime complexity for insertion and retrieving key.

Algorithm 1 Insertion algorithm for Linear Probing [26]

$x \leftarrow h(key\ mod\ n)$ // constant time
$i \leftarrow 0$ // constant time
while $(x \quad NULL)\{$ // n times
  $i \leftarrow i + 1$ // constant time
  $x \leftarrow x + i\ mod\ n$ // constant time
}
// constant time
if $T[h(key)] = NULL$ **then**
  map key into $h(key)$ slot // constant time
end if

Total $= c_0 + c_1 * n + c_2 + c_3 + c_4 + c_5 = O(n).$

Algorithm 2 Lookup algorithm for Linear Probing [26]

$x \leftarrow h(key\ mod\ n)$ // constant time
$i \leftarrow 0$ // constant time
while $(x < n)\{$ // n times
// constant time
if $T[x] = key$ **then**
  print("key found") // constant time
  break // constant time
end if
  $i \leftarrow i + 1$ // constant time
  $x \leftarrow x + i\ mod\ n$ // constant time
}
// constant time
if $T[x] = NULL$ **then**
  print("key does not found") // constant time
end if
break // constant time
}

Total $= c_0+c_1*n+c_2+c_3+c_4+c_5+c_6+c_7+c_8+c_9 = O(n).$

Algorithm 3 Quadratic Probing [22] Insertion algorithm

$x \leftarrow h(key\ mod\ n)$ // constant time
$i \leftarrow 0$ // constant time
while $(x \quad NULL)\{$ // n times
  $i \leftarrow i + 1$ // constant time
  $x \leftarrow x + i^2 mod\ n$ // constant time
}
// constant time
if $T[h(key)] = NULL$ **then**
  map key into $h(key)$ slot // constant time
end if

Total $= c_0 + c_1 * n + c_2 + c_3 + c_4 + c_5 = O(n).$

Algorithm 4 Quadratic Probing [22] Lookup algorithm

$x \leftarrow h(key\ mod\ n)$ // constant time
$i \leftarrow 0$ // constant time
while $(x < n)\{$ // n times
// constant time
if $T[x] = key$ **then**
  print("key found") // constant time
  break // constant time
end if
  $i \leftarrow i + 1$ // constant time
  $x \leftarrow x + i^2\ mod\ n$ // constant time
// constant time
if $T[x] = NULL$ **then**
  print("key does not found") // constant time
end if
break // constant time
}

Total $= c_0+c_1*n+c_2+c_3+c_4+c_5+c_6+c_7+c_8+c_9 = O(n).$

Authorized licensed use limited to: Wikipedia. Downloaded on January 20,2025 at 09:29:27 UTC from IEEE Xplore. Restrictions apply.

Algorithm 5 Double hashing [18] Insertion algorithm

```
x ← h_x(key) + h_y(key) mod n)  // constant time
i ← 0  // constant time
for (i = 1, ..., n − 1){   // n times
 // constant time
if T[x] = NULL then
   map key into h(key) slot // constant time
end if
   break // constant time
}
where h_x and h_y ∈ ∪ = {h_a, h_b, ..., h_z}
```

Total $= c_0 + c_1 * n + c_2 + c_3 + c_4 = O(n).$

Algorithm 6 Double hashing [18] lookup algorithm

```
i ← 0 // constant time
x ← h_x(key)  // constant time
y ← h_y(key)  // constant time
while (T[(x + i * y) mod n] ≠ key){   // n times
 // constant time
if T[(x + i * y) mod n] = −1 then
   print "key does not exist" // constant time
      break // constant time
end if
i ← i + 1 }  // constant time
print "Key found" // constant time
```

Total $= c_0 + c_1 + c_2 * n + c_3 + c_4 + c_5 + c_6 + c_7 = O(n).$

Algorithm 7 Coalesced Hashing [3] Insertion algorithm

```
x ← h(key mod n) // constant time
i ← n // constant time
while (x /= NULL and i > −1){  // n times
    i ← i − 1  // constant time
    x ← x + i mod n // constant time
}
 // constant time
if T[h(key)] = NULL then
   map key at position x slot // constant time
   set pointer from the colliding position // constant time
end if
```

Total $= c_0 + c_1 * n + c_2 + c_3 + c_4 + c_5 = O(n).$

Algorithm 8 Coalesced Hashing [3] Lookup algorithm

```
x ← h(key mod n)  // constant time
i ← n  // constant time
while (x < n and i > −1){   // n times
 // constant time
if T[x] = key then
   print("key found") // constant time
   break  // constant time
end if
    i ← i − 1  // constant time
    x ← x + i mod n  // constant time
}
 // constant time
if T[x] = NULL then
   print("key does not found") // constant time
end if
break  // constant time
}
```

Total $= c_0 + c_1 * n + c_2 + c_3 + c_4 + c_5 + c_6 + c_7 + c_8 + c_9 = O(n).$

Algorithm 9 Separate Chaining with Binary Tree [21] Insertion algorithm

```
x ← h(key mod n) // constant time
i ← n  // constant time
while (x /= NULL and i > −1){   // n times
    i ← i − 1  // constant time
    x ← x + i mod n   // constant time
}
 // constant time
if T[h(key)] = NULL then
   map key at position x slot // constant time
   set pointer from the colliding position // constant time
end if
```

Total $= c_0 + c_1 * n + c_2 + c_3 + c_4 + c_5 = O(n).$

Authorized licensed use limited to: Wikipedia. Downloaded on January 20,2025 at 09:29:27 UTC from IEEE Xplore. Restrictions apply.

Algorithm 10 Cuckoo Hashing [16] Insertion algorithm

$h_1(key) \leftarrow key \ mod \ n$ // constant time
$h_2(key) = (key/n) \ mod \ n$ // constant time
// constant time
if $T_1[h_1(key) = key] \ OR \ T_2[h_2(key) = key]$ **then**
print "Key already exist" // constant time
end if
while (true){ // n times
key swap $T_1[h_1(key)]$ // constant time
// constant time
if $key = NULL$ **then**
key swap $T_2[h_2(key)]$ // constant time
end if
// constant time
if $key = NULL$ **then**
key swap $T_2[h_2(key)]$ // constant time
end if
}
// constant time
if $T[h(key)] = NULL$ **then**
rehash all keys then try inserting the key // constant time
end if

Total $= c_0 + c_1 + c_2 + c_3 * n + c_4 + c_5 + c_6 + c_7 + c_8 + c_9 + c_{10} = O(n)$.

Algorithm 11 Cuckoo Hashing [16] Lookup algorithm

$h_1(key) \leftarrow key \ mod \ n$ // constant time
$h_2(key) = (key/n) \ mod \ n$ // constant time
// constant time
if $T_1[h_1(key) = key] \ OR \ T_2[h_2(key) = key]$ **then**
print "Key found" // constant time
end if
print "Key does not exist" // constant time

Total $= c_0 + c_1 + c_2 + c_3 + c_4 = O(1)$.

Algorithm 12 Multi-indexes hashing [27] Insertion algorithm

$input \leftarrow key$ // constant time
$key_{layer} \leftarrow h_1(input)$ // constant time
// constant time
if $T[key_{layer}] = exist$ **then**
// constant time
if $T[key_{layer}].value = input$ **then**
print("Key already exist in the hash table") // constant time
else
$new_inner_{layer} \leftarrow h_2(key_{new})$ // constant time
// constant time
if $T[key_{layer}].value = inner_layer_{pointer}$ **then**
print("Collision occur") // constant time
$old_inner_{layer} \leftarrow h_2(key_{old})$ // constant time
$T[new_inner_{layer}] \leftarrow input$ // constant time
$T[old_inner_{layer}] \leftarrow T[key_{layer}].value$ // constant time
$T[key_{layer}].value \leftarrow inner_layer_{pointer}$ // constant time
else
// constant time
if $T[new_inner_{layer}] = exist$ **then**
print("Collision occur") // constant time
else
$T[new_inner_{layer}] \leftarrow input$ // constant time
end if
end if
end if
else
$T[key_{layer}] \leftarrow input$ // constant time
end if

Total time $= c_0 + c_1 + c_2 + c_3 + c_4 + c_5 + c_6 + c_7 + c_8 + c_9 + c_{10} + c_{11} + c_{12} + c_{13} + c_{14} + c_{15} = O(1)$.

Algorithm 13 Multi-indexes hashing [27] Lookup algorithm

$input \leftarrow key$ // constant time
$key_{layer} \leftarrow h_1(input)$ // constant time
// constant time
if $T[key_{layer}] = exist \ \& \ T[key_{layer}].value = input$ **then**
print("Key exist in the hash table") // constant time
else
$inner_{layer} \leftarrow h_2(key)$ // constant time
$key_{layer} \leftarrow T[key_{layer}].value + inner_{layer}$ // constant time
// constant time
if $T[key_{layer}] = exist \ \& \ T[key_{layer}].value = input$ **then**
print("Key exist in the hash table") // constant time
else
print("Key does NOT exist in the hash table") // constant time
end if
end if

Total $= c_0 + c_1 + c_2 + c_3 + c_4 + c_5 + c_6 + c_7 + c_8 = O(1)$.

Authorized licensed use limited to: Wikipedia. Downloaded on January 20,2025 at 09:29:27 UTC from IEEE Xplore. Restrictions apply.

IV. Conclusion

In this work, we reviewed the performance of relevant hashing approaches based on their runtime complexity for mapping and retrieving a key from a hash table using asymptotic analysis. We found that the most efficient technique is multi-indexes hashing which has a constant time complexity $O(1)$ for both the insertion and lookup. This work also highlight hashing method applied in many domain such as security hashing, applications of hashing and survey work conducted in the area of hashing. Here, we focuses on runtime complexity. However other future work should examine performance of hash function applied in each technique along with the memory space analysis.

References

[1] Black, Paul E. "DADS: The On-Line Dictionary of Algorithms and Data Structures," NIST: Gaithersburg, MD, USA, 2020.

[2] Narasimha Karumanchi. "Data Structures And Algorithms Made Easy," 2017.

[3] Sriram, Ranjena, et al. "Efficient Data Cleaning Algorithm and Swift Unique User Identification Algorithm Using Coalesced Hashing and Binary Search Techniques for Web Usage Mining," International Journal of Pure and Applied Mathematics 118.18, 2018.

[4] Brenton Lessley and Hank Childs. "Data-Parallel Hashing Techniques for GPU Architectures," IEEE Transactions on Parallel and Distributed Systems Volume: 31, Issue: 1, 2020.

[5] Sun, Yuanyuan, et al. "SmartCuckoo: a fast and cost-efficient hashing index scheme for cloud storage systems." 2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17). 2017.

[6] Wing, J. M. "Computational thinking," 2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC). doi:10.1109/vlhcc.2011.6070404, 2011.

[7] Ahmed Dalhatu Yusuf, Saleh Abdullahi, Moussa Mahamat Boukar and Salisu Ibrahim Yusuf. "Collision Resolution Techniques in Hash Table: A Review," International Journal of Advanced Computer Science and Applications, Vol. 12, No. 9, 2021.

[8] Lianhua Chi, Xingquan Zhu. "Hashing techniques: A survey and taxonomy," ACM Computing Surveys, Vol. 50, No. 1, Article 11, 2017.

[9] Brad Miller, David Ranum. "Problem Solving with Algorithms and Data Structures," 2013.

[10] Necaise, Rance D. "Data structures and algorithms using Python," Wiley Publishing, 2010.

[11] Cormen, Thomas H., et al. Introduction to algorithms. MIT press, 2009.

[12] Michael T. Goodrich, Roberto Tamassia and David M. Mount. "Data Structures and Algorithms in C++," (Second Edition), 2009.

[13] Sailesh Kumar, Patrick Crowley. "Segmented Hash: An Efficient Hash Table Implementation for High Performance Networking Subsystems," 2005 Symposium on Architectures for Networking and Communications Systems (ANCS), 2005.

[14] Halevi, Shai, and Hugo Krawczyk. "Strengthening digital signatures via randomized hashing," Annual International Cryptology Conference. Springer, Berlin, Heidelberg, 2006.

[15] Nimbe, Peter, Samuel Ofori Frimpong, and Michael Opoku. "An efficient strategy for collision resolution in hash tables," International Journal of Computer Applications 99.10, 2014.

[16] Jane Rubel A. Jeyaraj, Sundarakantham Kambaraj and Velmurugan Dharmarajan. "High-speed data deduplication using Parallelized Cuckoo Hashing," Turkish Journal of Electrical Engineering & Computer Sciences 2018.

[17] Agrawal, Anand, Sriram Bhyravarapu, and Nuthalapati Venkata Krishna Chaitanya. "Matrix hashing with two level of collision resolution." 2018 8th International Conference on Cloud Computing, Data Science & Engineering (Confluence). IEEE, 2018.

[18] Phillip G.Bradford and Michael N. Katehakis, "A Probabilistic Study on Combinatorial Expanders and Hashing", SIAM Journal on Computing, 37 (1): 83-111, doi:10.1137/S009753970444630X, 2017.

[19] umar, Arun, and Supriya P. Panda. "A survey: how python pitches in IT-world." 2019 International Conference on Machine Learning, Big Data, Cloud and Parallel Computing (COMITCon). IEEE, 2019.

[20] Van Dijk, Tom. "Analysing and improving hash table performance," 10th Twente Student Conference on IT. University of Twente, Faculty of Electrical Engineering and Computer Science, 2009.

[21] Dhar, Siddharth, et al. "A tree based approach to improve traditional collision avoidance mechanisms of hashing." 2017 International Conference on Inventive Computing and Informatics (ICICI). IEEE, 2017.

[22] Konheim, Alan G. "Hashing in computer science: Fifty years of slicing and dicing," John Wiley & Sons, 2010.

[23] Abhay Kulkarni. "Efficient Hash Table Key Storage," Avago Technolo- gies International Sales Pte . Limited, Singapore (SG), 2019.

[24] Arvind K. Sharma ; S.K. Mittal. "Cryptography & Network Security Hash Function Applications, Attacks and Advances: A Review,". 2019 Third International Conference on Inventive Systems and Control (ICISC), 2019.

[25] Joux, Antoine. "Multicollisions in iterated hash functions. Application to cascaded constructions." Annual International Cryptology Conference. Springer, Berlin, Heidelberg, 2004.

[26] Knuth, Donald E. "Sorting and searching (6. printing, newly updated and rev. ed.). Boston [ua]." 2000.

[27] Ahmed Dalhatu Yusuf, Saleh Abdullahi, Moussa Mahamat Boukar and Salisu Ibrahim Yusuf. "A Multi-Indexes Based Technique for Resolving Collision in a Hash Table," IJCSNS International Journal of Computer Science and Network Security, VOL. 21 No. 9, September 2021.

[28] Byron Park and Dewan Tanvir Ahmed. "Abstracting Learning Methods for Stack and Queue Data Structures in Video Games," 2017 International Conference on Computational Science and Computational Intelligence, 2017.

Authorized licensed use limited to: Wikipedia. Downloaded on January 20,2025 at 09:29:27 UTC from IEEE Xplore. Restrictions apply.