Comprehensive Performance Study of Hashing Functions

G. M. Sridevi, M. V. Ramakrishna, and D. V. Ashoka

Abstract

Most literature on hashing functions speaks in terms of hashing functions being either 'good' or 'bad'. In this paper, we demonstrate how a hashing function that gives good results for one key set, performs badly for another. We also demonstrate that, for a single key set, we can find hashing functions that hash the keys with varying performances ranging from perfect to worst distributions. We present a study on the effect of changing the prime number 'p' on the performance of a hashing function from H_1 Class of Universal Hashing Functions. This paper then explores a way to characterize hashing functions by studying their performance over all subsets of a chosen Universe. We compare the performance of some popular hashing functions based on the average search performance and the number of perfect and worst-case distributions over different key sets chosen from a Universe. The experimental results show that the division-remainder method provides the best distribution for most key sets of the Universe when compared to other hashing functions including functions from H_1 Class of Universal Hashing Functions.

Keywords: H_1 Universal Class of Hashing Functions, Radix transformation, Mid-Square method, Multiplicative Hashing, Division-remainder method.

MSC 2020: 68P05, 68P10, 68P20.

ACM CCS 2020: Information systems—Database management system engines, Information systems—Information Storage Systems

doi:10.56415/csjm.v31.10

1 Introduction

Hashing techniques have been used extensively in various applications involving storage and retrieval, cryptography, networking, cloud, etc. Hashing functions map keys to table indexes and provide fast retrieval of data on different types of storage. The terms 'good hashing function' and 'bad hashing function' are usually used while talking about how a hashing function distributes the data in the hash table. While some hashing functions might distribute keys perfectly without any collisions, a few others may result in all collisions. The term 'collision' refers to the event where more than one key gets mapped to the same location in memory. In general, a hashing function causing fewer collisions is said to be 'good' and is said to be 'bad' otherwise. Different textbooks on database and data structures talk about characteristics of a 'good' hashing function.

The performance of a hashing function is dependent on the input key set and cannot be considered to be 'good' or 'bad' independent of the input. It is reasonable to evaluate the performance of a hashing function over all possible subsets of a Universe and not just over one key set. This idea was proposed by Lum [1]. Lum recommends considering all the subsets of a Universe to study the performance of hashing functions. To our surprise, there is no reported literature which studies the performance based on this strategy so far. Carter and Wegman's H_1 class of Universal Hashing functions was proved to provide practical performance on real files and is expected to provide better results [2], [3]. A comparison of the hashing functions with H_1 class of Universal Hashing functions has not been done so far as per our knowledge.

In this paper, we demonstrate the effect of the prime number on the performance of functions from H_1 class of Universal Hashing functions. We then present an exhaustive study on the performance of some of the known hashing functions in comparison with functions from H_1 class of hashing functions based on Lum's model. While functions from Universal Class were expected to give the best results, experimental results show that the division-remainder method performs better than the functions from H_1 class of Universal Hashing functions for the Universe chosen.

The rest of the paper is organized as follows: Section II presents the related work; Section III demonstrates that the performance of hashing functions is dependent on the input, followed by a study on H_1 class of hashing functions and presents the model used for the study of hashing functions; Section IV presents the results and the paper is concluded in Section V.

2 Related Work

Most of the studies on hashing functions was presented in the 60s. Surprisingly, there has been no study on the performance of hashing functions since 70s. Peterson presented an early study on hashing functions based on the number of probes required to find a record [4]. Buchholz provided an analysis of hashing functions covering different aspects of hashing functions including an analysis of overflow handling methods but with minimal experimental results [5]. According to the author, a hashing function that involves division of the key by a prime value provides efficient distribution of keys. Few other studies on hashing techniques were presented by Mc Ilroy [6], Roberts [7] and Schay et al. [8], [9]. The performance of different hashing functions over different sets of keys of varying datatypes was presented by Lum et al. with a conclusion that the division method provides the best search results with fewer collisions [10].

Later, Lum proposed a way to characterize the hashing functions by selecting key sets from a key space and hashing the keys using different hashing techniques [11]. Sorenson et al.'s survey on different hashing techniques along with collision-resolution techniques presents an analysis from a practitioner's point of view [12]. Deutscher et al. provide an excellent characterization and comparison of distribution-dependent hashing functions with that of division method [3].

Ramakrishna's study on H_1 class of Universal hashing functions defined by Carter and Wegman concludes that functions from this class can provide practical performance on real files [10]. In this paper, we present a comparison of the performance of some of the hashing techniques with functions from H_1 class. In the next section, we demonstrate that any hashing function cannot be declared as 'good' or 'bad' without considering the input given to the hashing function.

3 Hashing Functions: 'good' or 'bad'

In this section, we consider a few examples to illustrate that the same hash function can perform differently for different key sets. We chose functions from H_1 class of Universal hashing functions defined by Carter and Wegman which have been claimed to be good for providing practical performance on real files [2]. A separate chaining method is applied to link the keys that result in collisions for our examples. For the demonstration, we consider 2 different scenarios:

Case 1: Same hashing function, different key sets.
We hashed 4 different key sets with the hashing function $h(x) = ((519x + 703)\, mod\, 36373)\, mod\, 13$ to a hash table of size 13. The key sets considered for the demonstration of the first case are as follows:

$$K_1 = \{936, 748, 996, 815, 864, 867, 730, 971\}$$
$$K_2 = \{772, 841, 738, 907, 876, 932, 713, 816\}$$
$$K_3 = \{757, 861, 929, 744, 755, 797, 808, 836\}$$
$$K_4 = \{715, 952, 899, 754, 860, 913, 992, 781\}$$

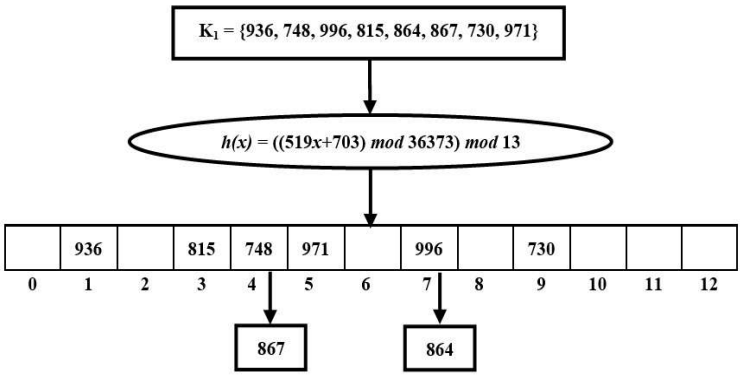

Figure 1. Good/Normal Distribution of K_1 by $h(x)$

Fig. 1 shows the distribution of the keys in key set K_1 to the hash table using the hash function $h(x)$. The hash function $h(x)$ causes only 2 collisions for key set K_1. The function $h(x)$ distributes the keys from K_2 without any collisions making it a perfect hashing function for key

set K_2. On the other hand, it results in a larger number of collisions for K_3, with all keys hashing to either table index 8 or to index 10, and in all collisions for K_4. The average search performance of $h(x)$ for different key sets is shown in Table 1. The hash function's performance over the key sets was found to be good for K_1; perfect for K_2; bad for K_3 and worst for K_4 with all keys hashing to table index 11. As we can see, a hashing function's performance depends on the key set chosen.

Table 1. Average Search Performance of $h(x)$ over different key sets

Key Set	Average Successful Search Length	Performance
K_1	1.25	Good
K_2	1	Perfect
K_3	2.625	Average/Normal
K_4	4.5	Worst

Case 2: Same key set, different functions.
In the previous scenario, we showed the performance of a single hashing function on different key sets. In another scenario, we show that, for a single key set, we can find functions h_1, h_2... such that h_1 is perfect, h_2 is average and so on.

For our demonstration of the second case, we consider a key set K_5 = {763, 789, 841, 867, 893, 919, 945, 997} of size 8 chosen randomly from a Universe U ranging from 700 to 1000. For the demonstration, the key set K_5 is hashed using 4 different hashing functions listed below:

$$h_1(x) = ((412x + 371)\, mod\, 5443)\, mod\, 13$$
$$h_2(x) = ((321x + 576)\, mod\, 27283)\, mod\, 13$$
$$h_3(x) = ((513x + 413)\, mod\, 106759)\, mod\, 13$$
$$h_4(x) = ((417x + 294)\, mod\, 158647)\, mod\, 13$$

Fig. 2 shows the distribution of the key set K_5 by the hashing function $h_1(x)$ which is found to be perfect with no collisions. The same key set K_5 results in a few collisions when hashed with the function $h_2(x)$. A larger number of collisions is seen when K_5 is hashed with

$h_3(x)$ with all keys getting hashed to either index 0 or index 3 of the hash table. The hashing function $h_4(x)$ hashes the key set with all collisions by mapping all the keys to the address 0. The average successful search performance of the hash functions over the key set K_5 is shown in Table 2. This shows that, for the same key set, different hashing functions may perform differently. Choosing a hashing function that provides the best results is vital for any application.

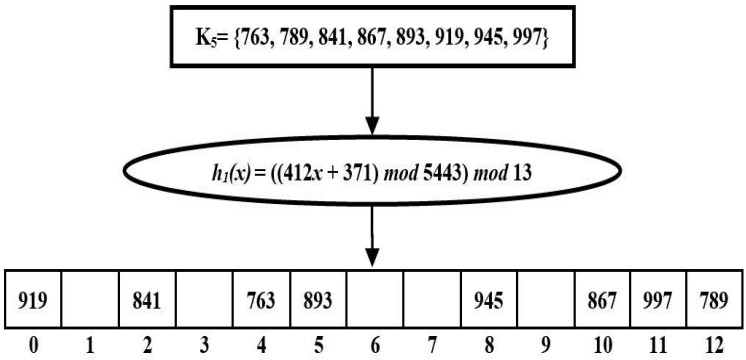

Figure 2. Perfect distribution for K_5 by $h_1(x)$

Table 2. Average Search Performance of different hashing functions over K_5

Hashing Function	Average Successful Search Length	Performance
$h_1(x)$	1	Perfect
$h_2(x)$	1.625	Good
$h_3(x)$	3	Bad
$h_4(x)$	4.5	Worst

We have illustrated with examples that the performance of any hashing function is dependent on the key set and, for any given key set, we can find functions that vary in performance. Also, given any hashing function, we can find key sets for which the performance of the hashing function varies from perfect to good, bad, or even worse.

This can be done for any other class of hashing functions such as radix transformation, division-remainder method, and so on.

3.1 The effect of the prime 'p' on the performance of H_1 Class of Hashing Functions

We demonstrated that a hashing function cannot be considered 'good' or 'bad' without taking the input into consideration. For the demonstration, we used hashing functions from H_1 class of Universal hashing functions defined by Carter and Wegman of the form

$$h(x) = ((c * x + d) \, mod \, p) \, mod \, m, \tag{1}$$

where c and d are chosen at random, p is a large prime, and m is the table size. Functions from H_1 class are said to be Universal indicating that the maximum number of collisions expected is nearly n/m, where n is the size of the key set and m is the hash table size. We can observe that the performance of the hashing functions varied depending on the values chosen for 'c', 'd', and 'p'. This motivated us to study the effect of varying the values of 'p' on the performance of H_1 class of Universal hashing functions.

In particular, we chose a key set with keys that generate the same remainder on dividing by the hash table size m taken as 13. We studied the performance of functions from H_1 class over a key set K of size 10 ranging from 700 to 1000. The key set considered for the study is

$$K = \{737, 945, 763, 815, 867, 841, 893, 789, 919, 997\}.$$

We can see that the keys chosen would result in all collisions if hashed using the modulo method. The keys from the key set K were hashed to the hash table using hashing functions of the form in Eq. (1). The values for 'c' and 'd' were kept constant at $c = 453$ and $d = 657$. The performance of the hashing function was evaluated for different values for 'p' to study the effect of the prime number on the performance of the hashing function and is shown in Table 3. The results indicate that as the ratio of '$(cx+d)$' to 'p' decreases, the number of collisions increases. Larger values of 'p' result in a higher number of collisions. If the value

of 'p' is too large, it results in all collisions. Whereas, smaller values result in fewer collisions and can also provide perfect distribution. In the next subsection, we present a comprehensive study of some of the popular hashing functions.

Table 3. Performance variation of a hashing function $h(x)$ on varying 'p'

$p$value	No. of overflow records	Average Search length	$\frac{(c*x+d)}{p}$	Search Performance
226637	9	5.5	1.47	Worse
170689	8	4.6	1.95	Bad
156967	9	5.5	1.1	Worse
128761	8	3.0	2.59	Average
70571	7	2.7	4.74	Average
36373	7	2.3	9.19	Average
32203	5	1.7	10.38	Good
30241	6	1.8	11.06	Good
19739	3	1.3	16.94	Good
13219	1	1.1	25.30	Good
10687	0	1	31.30	Perfect
7477	2	1.2	44.73	Good
5879	0	1	56.9	Perfect
2357	0	1	141.9	Perfect

3.2 Exhaustive Performance Study of Hashing Functions

We demonstrated that the performance of a hashing function is highly dependent on the input. Based on this fact, the best strategy to study the performance of a hashing function over any given Universe of keys would be to consider the performance of the hashing function over all the possible subsets of the Universe rather than on just a few key sets. As already demonstrated, different hashing functions may provide varied performance for same key set while they behave differently for other key sets. Hence, we consider exhaustive subsets of the Universe, according to which all possible subsets of a Universe must be considered for

hashing to properly understand how a hashing function distributes the data. This was suggested by Lum in his earlier studies. A study on Universal hashing functions based on this idea is presented here in comparison with other existing methods.

Consider a Universe of keys $U = \{k_1, k_2, k_3, ..., k_n\}$ of size n. There are ${}^nC_r$ possible subsets KS (Key Set) of size r. Each of these subsets is hashed to the Hash Table individually, and the average search performance of the hashing function over all the subsets is taken into consideration to study the performance. Some key sets may get distributed perfectly without any collisions. Some may be worst cases (all keys colliding) and others will be in between. We evaluate the performance of the hashing functions by taking an average of all the extreme cases. The algorithm for the comprehensive study is presented in Algorithm 1.

Algorithm 1.

Input:
Universe U= {10, 11, ..., 29}
Universe Size n = 20
Table Size m=10
Subset Sizes r = {3, 4, ..., 10}

Begin:
1: For each r
Generate all subsets/key sets of U (KS) of size r
//$KS = \{KS_1, KS_2, ..KS_{{}^nC_r}\}$
2: For each subset KS_x //$KS_x = \{k_1, k_2, ..., k_r\}$
Initialize Total Search Length for KS_x to 0
3: For each key k_x in a subset KS_x
hash(k_x)
Compute Search length for k_x //($SL(k_x)$)
Total Search Length for KS_x= Total Search Length for KS_x + $SL(k_x)$
End For3
Calculate Average Search Length over KS_x ($ASL(KS)$)
End For2
$ASL(KS)$ = Average search Length over ${}^nC_r$ subsets
End For1

4 Results and Discussion

For our experiment, we considered mid-square method, radix transformation, division-remainder method, multiplicative method [2], [13], and H_1 class of hashing functions. A brief explanation of the hashing functions considered for the study is given below.

Mid-square method extracts the middle portion of square value of the key as the hash address for the key.

For example, for the key $x = 2518$, squaring the key gives 63,40,324. The middle of the squared value, i.e., '0' is taken as the hash address to store the key x.

Radix transformation transforms the key from decimal to a different base and the resulting value is used to generate the hash address. The key can be converted to any base of our choice. For demonstration, we convert the key to base 9.

Consider a key $x = 7698$. Converting the decimal 7698 to base 9 gives the value 11503. The converted value is then used to generate the hash address. For a table size of $m = 10$, we use a modulo operation on the transformed value with the table size to generate a hash address within the address range 0 to 9. For $m = 10$, we get 3 ($11503 mod 10$) as the hash address.

Division-remainder method or modulo method is of the form $h(x) = x \, mod \, m$, where x is the key and m is the hash table size. The remainder obtained on dividing the key by table size is taken as the hash address.

For $x = 1893$ and $m = 10$, we get 3 (1893 mod 10) as the hash address.

Multiplication method is of the form $h(x) = floor(m * ((x * c) \, mod \, 1))$, where c is a floating-point value ranging between 0 to 1 and m is the table size. The key x is multiplied by c, and the fractional part is extracted by applying a modulo operation with 1 on the product obtained. The remainder is then multiplied by the table size and the real part is taken as the hash address.

For example, for a key $x = 1788$, if $c = 0.572593$, $x * c$ is 1023.796284. Modulo operation on the product with 1 gives 0.796284 as the remainder which is multiplied with the table size m. If $m = 10$, we get 7.96284 as a result, from which the floor value is taken as the hash address, i.e., 7.

H_1 **class of hashing functions** defined by Carter and Wegman are of the form $h(x) = ((cx + d)\, mod\, p)\, mod\, m$, where c and d are integers chosen at random, p is a large prime, and m is the table size. For $x = 18$, if $c = 58, d = 67$, $p = 137$, and $m = 10$, we get 5 as the hash address.

For any given set of size n, the number of possible subsets of size r is exponential and is equal to ${}^nC_r$. Thus, we have to keep the size of the Universe n to be small so that the total time taken to conduct the experiments is within manageable limits. For $n = 100$, the number of possible subsets of size $r = 7$ is ${}^{100}C_7$ =16007560800 and the time taken to generate all the subsets is nearly 42 hours. For r = 8, we get 186087894300 possible subsets which is estimated to take nearly 21 days to generate. For larger values of $r = 9\, and\, 10$, we get 1902231808400 and 17310309456440 possible subsets. It would take longer to generate and hash all the subsets to the hash table; hence, we chose a smaller Universe U of size $n = 20$, for which the number of subsets is 184756 for $r = 10$, which takes lesser time to generate. The Universe U chosen ranges from 10 to 29. Subsets are constructed for different sizes 'r' ranging from 3 to 10. Each of these subsets is hashed to a Hash Table of size $m = 10$, and the average search performance is calculated over all the subsets.

Each of the hashing functions was used to hash all the possible subsets of the Universe for subset sizes $r = 3\, to\, 10$. For Universal hashing functions, we generated 100 different functions by varying the values of 'c', 'd' at random and keeping the prime 'p' as constant. The average of Average search length for the 100 hashing functions was used to compute the search performance of Universal hashing functions. For example, if $r = 3$, for each hashing function, we get 1140 subsets; we take the average search length for all the subsets for each hashing function. This is repeated for 100 hashing functions. Later we take the

average of the 100 average search lengths to compute the performance of Universal hashing functions.

Table 4. Comparison based on Average Successful Search Length

Subset Size (r)	3	4	5	6	7	8	9	10
No. of Subsets (${}^nC_r$)	1140	4845	15504	38760	77520	125970	167960	184756
Mid-Square method	1.12	1.19	1.25	1.31	1.38	1.44	1.5	1.57
Modulo method	1.05	1.08	1.1	1.13	1.16	1.18	1.21	1.23
Multiplication method	1.13	1.19	1.25	1.31	1.38	1.44	1.5	1.57
Decimal to base 11	1.06	1.09	1.12	1.16	1.19	1.22	1.25	1.28
Decimal to base 15	1.16	1.23	1.31	1.39	1.47	1.55	1.63	1.71
Universal Hashing	1.07	1.11	1.15	1.18	1.22	1.26	1.29	1.33

Table 4 shows the results of experiments based on successful search length. The subset size r (shown in the first row) varies from 3 to 10. For each size, we generated all possible subsets of that size taking the elements from our small Universe. The total number of subsets generated in each case is shown in the second row (${}^nC_r$). After hashing all the keys in a subset, the successful search length is computed. The average of this search length over all the subsets is computed and shown in the next six rows corresponding to each method of the hashing

function used.

Table 5. Comparison based on Percentage of Perfect distributions

Subset Size (r)	3	4	5	6	7	8	9	10
No. of Subsets (^nC_r)	1140	4845	15504	38760	77520	125970	167960	184756
Mid-Square method	65.96	42.13	21.96	8.79	2.41	0.34	0	0
Modulo method	84.21	69.35	52.01	34.67	19.81	9.15	3.05	0.55
Multiplication method	67.63	50.09	33.69	20.52	10.79	4.65	1.46	0.25
Decimal to base 11	81.40	64.75	46.31	29.05	15.40	6.49	1.94	0.31
Decimal to base 15	59.65	36.12	18.78	8.31	3.04	0.88	0.18	0.02
Universal Hashing	78.77	60.56	41.49	24.75	12.43	4.96	1.41	0.22

Tables 5 and 6 show the percentage of distributions that are perfect(P) and worst(W) respectively for different hashing functions. For example, with $r = 3$, we have 1140 possible subsets. A distribution is perfect if the resulting successful search length is precisely 1.0 or if there were no collisions. With the mid-square method, out of 1140 subsets, 752 distributions gave a search length of 1.0. Thus, we get (752/1140) * 100 = 65.96 % of perfect distributions for mid-square method depicted in Table 5. Conversely, out of 1140 subsets, 22 subsets result in all collisions which gives us (22/1140) * 100 = 1.93% of the worst

Table 6. Comparison based on Percentage of worst-case distributions

Subset Size (r)	3	4	5	6	7	8	9	10
No. of Subsets (^nC_r)	1140	4845	15504	38760	77520	125970	167960	184756
Mid-Square method	1.93	0.31	0.04	0	0	0	0	0
Modulo method	0	0	0	0	0	0	0	0
Multiplication method	2.68	0.91	0.33	0.11	0.03	0.01	0.001	0.0001
Decimal to base 11	0.18	0	0	0	0	0	0	0
Decimal to base 15	3.51	0.62	0.08	0.01	0	0	0	0
Universal Hashing	0.37	0.02	0	0	0	0	0	0

distribution for the mid-square method depicted in Table 6. Similarly, the division-remainder method provides 960 perfect distributions and 0 worst-case distributions for $r = 3$. Our experiments show that the division-remainder method gives the highest percentage of perfect distributions and the least percentage of the worst distributions for all subset sizes when compared to other methods.

While we were inclined to believe that functions from H_1 class would provide the best results when compared to other hashing techniques, it was surprising to see that, contrary to our belief, the average search performance of functions from H_1 class was not as good as division-remainder method. The performance of functions from H_1 was closely behind that of the remainder method and radix transformation to base 11. Even the number of subsets with perfect distribution was highest with the remainder method.

5 Conclusions

In this paper, we have presented the performance of some popular hashing techniques over all the possible subsets of a Universe. For the Universe chosen, the remainder method distributed the subsets of different sizes with fewer collisions resulting in a smaller average successful search. It was closely followed by the radix transformation method to base 11. The division-remainder method also gave the highest percentage of perfect distributions followed by radix transformation to base 11 for all subset sizes r. A smaller value for the prime 'p' improved the performance of the Universal Hashing function making it better than the radix transformation but it did not do better than the division-remainder method which was contrary to our belief.

Acknowledgments

This research was supported by Visvesvaraya Technological University, Jnana Sangama, Belagavi.

Conflict of interest

The authors declare no potential conflict of interests.

References

[1] V. Y. Lum, P. S. Yuen, and M. Dodd, "Key-to-address transform techniques: A fundamental performance study on large existing formatted files," *Communications of the ACM*, vol. 14, no. 4, pp. 228–239, 1971.

[2] J. L. Carter and M. N. Wegman, "Universal classes of hash functions," in *Proceedings of the ninth annual ACM symposium on Theory of computing*, 1977, pp. 106–112.

[3] R. Deutscher, P. G. Sorenson, and J. P. Tremblay, "Distribution-dependent hashing functions and their characteristics," in *Proceedings of the 1975 ACM SIGMOD international conference on Management of data*, 1975, pp. 224–236.

[4] W. W. Peterson, "Addressing for random-access storage," *IBM journal of Research and Development*, vol. 1, no. 2, pp. 130–146, 1957.

[5] W. Buchholz, "File organization and addressing," *IBM Systems Journal*, vol. 2, no. 2, pp. 86–111, 1963.

[6] M. D. Mc Ilroy, "A variant method of file searching," *Communications of the ACM*, vol. 6, no. 3, p. 101, 1963.

[7] D. C. Roberts, "File organization techniques," in *Advances in Computers*. Elsevier, 1972, vol. 12, pp. 115–174.

[8] G. Schay and N. Raver, "A method for key-to-address transformation," *IBM Journal of research and development*, vol. 7, no. 2, pp. 121–126, 1963.

[9] G. Schay Jr and W. G. Spruth, "Analysis of a file addressing method," *Communications of the ACM*, vol. 5, no. 8, pp. 459–462, 1962.

[10] M. V. Ramakrishna, "Hashing practice: analysis of hashing and universal hashing," *ACM SIGMOD Record*, vol. 17, no. 3, pp. 191–199, 1988.

[11] V. Y. Lum, "General performance analysis of key-to-address trans-

formation methods using an abstract file concept," *Communications of the ACM*, vol. 16, no. 10, pp. 603–612, 1973.

[12] P. Sorenson, J. Tremblay, and R. Deutscher, "Key-to-address transformation techniques," *INFOR: Information Systems and Operational Research*, vol. 16, no. 1, pp. 1–34, 1978.

[13] J. Von Neumann, "13. various techniques used in connection with random digits," *Appl. Math Ser*, vol. 12, no. 36-38, p. 3, 1951.

G. M. Sridevi, M. V. Ramakrishna,
D. V. Ashoka

Received October 26, 2022
Accepted January 25, 2023

G. M. Sridevi
ORCID: https://orcid.org/ 0000-0003-3864-9983
Research Scholar - Visvesvaraya Technological University (VTU),
Dayananda Sagar Academy of Technology and Management,
Department of Information Science and Engineering,
Udayapura, Kanakapura Road, Bengaluru-560082, India.
E–mail: sridevi.gereen87@gmail.com

M. V. Ramakrishna
ORCID: https://orcid.org/0000-0001-7058-7562
SJB Institute of Technology, Department of Information Science and Engineering,
BGS Health and Education City, Dr. Vishnuvardhan Road,
Bengaluru-560060, India.
E–mail: mvrama@yahoo.com

D. V. Ashoka
ORCID: https://orcid.org/ 0000-0003-1326-2387
JSS Academy of Technical Education
Department of Information Science and Engineering,
Dr. Vishnuvardhan Road,
Bengaluru-560060, India.
E–mail: dr.dvashoka@gmail.com