Research Reports on Computer Science
https://ojs.wiserpub.com/index.php/RRCS/

UNIVERSAL WISER PUBLISHER

Research Article

Linked List Elimination from Hashing Methods

Mahmoud Naghibzadeh[1*], Bahram Naghibzadeh[2]

[1]Department of Computer Engineering, Ferdowsi University of Mashhad, Mashhad, Iran
[2]Department of General Practice, Monash University, Notting Hill, Victoria, Australia
E-mail: naghibzadeh@um.ac.ir

Received: 8 September 2021; **Revised:** 24 October 2021; **Accepted:** 10 November 2021

Abstract: Hashing has been used for decades in many fields such as encryption, password verification, and pattern search. Hash systems consist mainly of three components: the hash function, the hash table, and the linked lists that are attached to the hash table. One of the major benefits of using a hash function is reduction in the runtime of the hash-based software systems. However, their linked lists are a major source of time consumption. In this paper, an innovative method is proposed to remove all the linked lists attached to the hash table and collect the necessary information in a one-dimensional array. The method can be used to create an index for the human genome. The human genome is the size of a million-page book with no index, and it is difficult to find the needed information. The proposed method transforms list search operations with linear time complexity into array searches with logarithmic time complexity. In a sample problem, finding inversions in genomic sequences, the proposed indexing system is compared with traditional hashing systems with linked lists. It is demonstrated that, in addition to time complexity reduction, the proposed method reduces the space required for the hash system to one half of what is used by linked list based methods.

***Keywords*:** hashing systems, linked list elimination, genome indexing, locating inversions, cryptography

1. Introduction

The hash function is widely used in a variety of computer programs. Depending on the application, a hash system can have several components. However, hash function is common in all hash systems. What a hash function does is to convert a data string to another string called value. To get acquainted with the components of the hashing system, two very simple hashing methods are briefly discussed.

Suppose a 32-bit data is given and an extra bit value is to be calculated and attached to these 32 bits to make it a 33-bit odd parity data. The hashing function (or algorithm) is simple: if the number of digits that are one in the input data is odd then the parity bit is 0, otherwise it is 1. In this example, the input is the 32-bit data. The parity bit is the output value. This is a preliminary example of a hash function in cryptography and password hashing [1, 2]. In cryptography, a hashing system is composed of three components: the variable length input data, the hash function, and the scrambled output. From this example, it is valid to say that from the output it is not possible to find out what the input data is. In other words, the hash function is a one way function. It is also valid to say that the output value is of fixed length. In addition, it is safe to say the length of input data can be variable. Here, there was no need for hash table and linked list data structures. The message-digest 5 (MD5) algorithm is a practical example of hashing method in cryptography. It accepts the input data and generates a fixed-size hash value of 128 bits. The weakness of MD5 is that it

DOI: https://doi.org/10.37256/rrcs.1120221145

is possible for two different input messages to have the same hash values. This is called *collision* which is not allowed in cryptography, but frequently occurs in other applications of hashing. Three extensions of MD5 are already designed and weaknesses of MD5 are fixed. Secure Hash Algorithms (SHA-1, SHA-2, and SHA-3) are these extensions [3]. A lightweight cryptographic hash function is used by wireless sensor networks [4, 5].

The next example is from the field of bioinformatics. The concern of bioinformatics is using the power of computer hardware, software, and storage capacities for analysis of genomic data, help diagnose disease [6, 7], and design drugs.

The human genome sequence was reported in 2001. Its one-dimensional representation is a sequence of about 3.2 billion nucleotides. The declared genome is not the genome of only one person, but it is the consensus genome of thousands of healthy people from many countries who participated in the Human Genome Project [8]. The construction alphabet of the genome is composed of four nucleotides: A(adenine), T(thymine), G(guanine), and C(cytosine). In the reported genome, the same nucleotide is selected for locations where the nucleotides of all participants are the same. However, for places where not all nucleotides are the same, a consensus method is applied and a nucleotide is selected. There are many sites which store and make available all kinds of genomic data, e.g., the National Center for Biotechnology Information (NCBI) website [9].

Imagine a book with one-million pages is published, and it has no index, and the book is not a self-contained index one such as a dictionary. However, since the locations of chromosomes and genes are known we can say it has a table of contents. The human genome is one such book. The efficient hash-based index proposed here is suitable for such long sequences and many other types of sequences. The most valuable novelty of this work is the efficiency in searching for a given pattern in a given long sequence. After the index of the hash table for the given pattern is found, verification of whether the pattern exists in the given approximate location using linked lists takes proportional to $O(k)$ operations whereas it takes $O(\log_2 k)$ with the proposed method. This is a remarkable improvement especially when k, i.e., number of nodes in the list, is large. Another achievement is in space requirement. The space needed by hash system of the proposed method, as it is analyzed in the evaluation section is approximately 50% of what is needed by the traditional linked list method. This achievement is also important because for large texts (or genomic sequences) the space required by the hash system is very large.

In the next section, a formal definition of the problem which is studied in this paper is presented.

2. Problem definition

The assembled genome from many healthy humans from many countries is called the reference genome. The human genome may face a variety of mutations through the years. From what we have seen in the past two years of COVID-19 pandemic, we know very well what a mutation is. Any changes, such as deletion, insertion, or exchange in the nucleotide sequence of a genome is called mutation. The most reliable technique to find the differences between a subject's genome with that of the reference genome is optimal alignment of the two genomes. However, this is a simple task and at the same time an impossible one. To align two sequences of lengths m and n, the components of an m by n array must be calculated [10]. Suppose calculation of each element needs 10 machine instructions. For two human genomes with length 3.2 Giga nucleotides each, more than 10^{19} operations must be performed. With a *desktop* computer capable of performing say 100 million instructions per second (MIPS) it will take 1,157,407 days to do the alignment! However, another disaster occurs elsewhere. The main memory needed for the two-dimensional array, using a 4-byte word, is $4 * 10^{10}$ gigabytes! Although there are solutions for these problems, we must remember that the developed tool should be usable by genetic laboratories with common computing facilities. In any case, as it was mentioned before, a book must have an index. The human genome is as big as a one million page book with 3,200 characters per page. Therefore, an index can definitely help finding what we are looking for in the genome. In the next section, a hash-based index is proposed for all kinds of genomic sequences. Later, the linked lists attached to the hash table will be replaced by an array and, by doing so, the worst case time complexity of searching for an item is reduced from $O(n)$ to $O(\log_2 n)$. At the same time, the space requirement (not space complexity), is also reduced.

2.1 *Hash index for genomic sequences*

Unlike textual words in a book that are spaced apart, there is no such separator in the genomic sequences.

Therefore, the hashing unit must be defined. Here, 6 consecutive nucleotides are taken to be the hashing unit, based on which the index is built. This unit length can be changed depending on the application for which the index is being made. A simple but very efficient coding system is used for the hash function. A two-bit binary code for each nucleotides A, C, G, and T is considered to be 00, 01, 10, and 11, respectively. For example, the code for AGATTC is $(001000111101)_2$ which is equivalent to 573 in base 10. When AGATTC is given as input, after it is coded and the code is evaluated, it will point to the row number 573 of the hash table. This hashing system has been used before [11] and the novelty of this research is in removing the linked lists that are connected to the hash table.

For the design of an index for genomic sequences, this coding system is collision free. It is true that the code for A, AA, AAA, … are all zero and one may think that they all point to Row 0 of the hash table. However, the unit of hashing is fixed and it is 6 nucleotides, therefore, the only string which will point to row number 0 is AAAAAA.

In a long genome, such as the human genome, there may be many occurrences of each hashing unit. For example, AGATTC may be repeated say 2,000 times in different places of the genome. Therefore, a linked list of length 2000 will be attached to this row. This linked list represents the locations of occurrences of the AGATTC on the genome. The overall structure of the hashing system is shown in Figure 1. In this figure, AGATTC is hashed and the hash value is 573. It points to row number 573 of the hash table and this row points to a linked list with its nodes having two fields each. The first field of the first node stores the first location of the input genome where AGATTC appears. Since the reference genome's locations start from zero, therefore AGATTC starts from Location 3, i.e., fourth location, of the sequence. If the next occurrence of AGATTC happens to be in Locations 28 to 33 of the input sequence a new node will be added at the end of the list which is connected to Row 573 of the hash table, and in its data field the Value 28 will be stored and links will be updated.

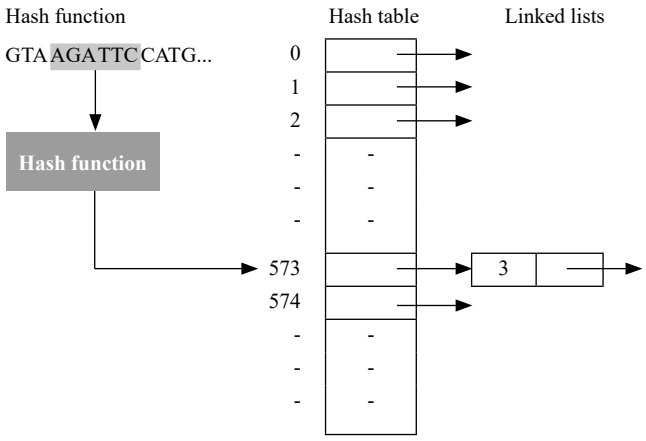

Figure 1. Hashing components of genome indexing

An important property of the designed hashing system for genomic sequences in this paper is that the first substring to be hashed is from Location 0 to Location 5 of the input sequence, the second one is from Location 1 to Location 6 (not from Locations 6 to 11), and so on. Therefore, for an input sequence of length n there will by n-5 subsequences to be hashed. Talking about the human genome consisting of 3.2 Giga characters, 3.2 Giga-5 substrings are hashed. From here on, for the sake of simplicity, we say 3.2 Giga substrings will be hashed.

3. Hash-based index without linked lists

With the current COVID-19 pandemic, awareness of people about mutations in the genome sequences has increased. People working in this area know that the virus responsible for SARS (Severe Acute Respiratory Syndrome), MERS (Middle East Severe Respiratory Syndrome), COVID-19, and its new variants delta and lambda, are all products

of mutations in the same original genome. This explains that mutations can change the behavior of a virus and increase the degree of its burden. Mutations in the human genome are not as rapid as what we have seen in the coronavirus. However, mutations in human genome do exist and can be the cause of many diseases such as cancer and Parkinson's disease. The existence of insert/delete, abnormal tandem repeats of patterns, replacement of nucleotides, changes in the number of repeats in a repeating pattern, and inversion are a few types of mutations in genomic sequences. To clarify the use of having an efficient index for a genome in finding any of these abnormalities in a patient's genome, we will concentrate on *inversion* which is less known compared to other types of mutations.

Suppose the hash index for the whole reference genome is to be made which is structurally similar to Figure 1. This is an offline task which is made only once, similar to making the index of a textbook which is made once and later millions of users can use it. The task is simple, however, Algorithm 1 shows the general method. In this algorithm, S represents the input genome and n represents the length of S. Variable k represents the length of the hashing unit which was set to 6 in the previous section.

Algorithm 1. Production of hash-based genome index

```
void HashIndex (char S[], int n, int k, int Htable[])
{
    int i; int b; boolean a;
    for ( i =0; i <= n-k; n++)
    {
      a= code (S[i] to S[i+k-1]);
      b= convert-to-integer(a);
      add a new node y to the list of Htable[b];
    }
}
```

Inversion is a kind of mutation in which the orientation of a segment of the genome is inverted or reversed end-to-end. This could happen for a short segment or a long one. Inversion is very frequent and could be associated with many kinds of diseases. For example, certain cancers, growth retardation, infertility, pregnancy loss, and congenital anomalies are associated with inversions in Chromosome 9 of humans. Chromosome 9 is about 138 million nucleotides long. Finding inverted segments and their positions is a difficult task [12].

Having the hash index of the human genome, suppose the task is to find all inversions of Chromosome 9. Using the same hashing unit size, 6, Chromosome 9 is scanned from end to front. Suppose Chromosome 9 ends with GTCTGATG. So, GTAGTC, TAGTCT, and AGTCTG will be processed in that order. The processing of each one includes looking for it in the genome's index and if found analyzing its location. If it passes the position test then go to actual sequences, i.e., reference genome and patient's Chromosome 9. Use a seed and extend approach [13] to find the actual inverted string which might be longer than 6 nucleotides. The limitation of this method is that inversions of shorter lengths than 6 nucleotides cannot be found. If the intention is to find such inversions the indexing must be done for shorter hash units.

To find all inversions [14] in Chromosome 9, the number of times we must go to hash table and scan the related linked list is 138 million. In each case, a sequential traverse of the related linked list is performed. It is this sequential scan that can be improved and bring it down to a binary search scan by using the proposed method. The analysis section of the paper talks about the two methods' time complexities. In the following, the novel idea of eliminating linked lists from the hashing system represented by Figure 1 is discussed.

3.1 *Linked lists elimination of hashing system*

The most important property of the hashing system which is represented by Figure 1 and helps us to design an efficient index for genomic sequences is that in each linked list which is attached to the hash table, the nodes are ascendingly sorted. The sort is based on the numeric values saved in the nodes. The guideline for modifying the hashing system and producing an efficient one follows.

Create a new array called *loc* in order to save the locations of all hashed substring. Then, start with the first linked

list attached to the hash table and process all the lists in order. Traverse each list from its start towards its end, and add the observed values in nodes of the list to consecutive locations of Array *loc*. After each list is completely traversed, in the hash table itself, save the start and the end indices of Array *loc* in which the values of nodes of this list are stored. This means we should have two fields in the new hash table.

After this phase is completed, Figure 1 is completely replaced with Figure 2. In this figure, it is assumed that the subsequence AGATTC is seen in Locations 3, 837, 65673, 497383.

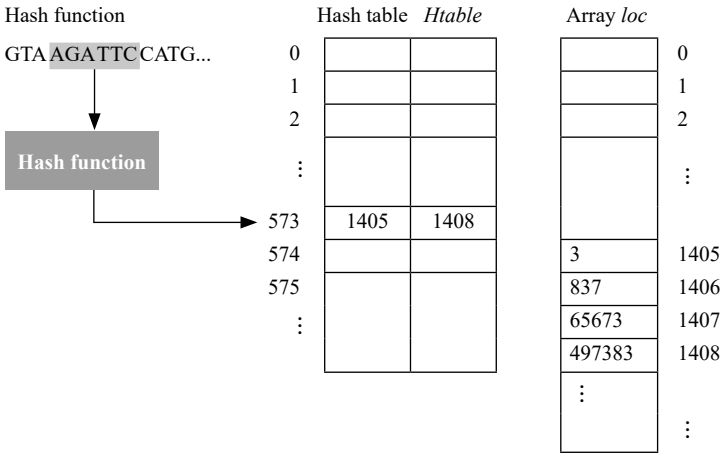

Figure 2. Hashing system with no linked list

The hashing system of Figure 2 is the alternative to the hashing system of Figure 1. With the completion of processing the hash table and linked lists of Figure 1 and producing the hash table and Array *loc* of Figure 2, for the human genome, either one of the two figures can be used as an index for the human genome. Later we will talk about the superiority of the method not having linked lists, i.e., Figure 2, over the one that has linked lists.

In the following, a short sequence of 30 nucleotides is selected and the 3 phases of the algorithm: generating the linked list of hash system, converting it to a linked list free hashing system, and searching for a subsequence is discussed. The sequence is shown in Table 1. The first row of the table is the actual sequence and its second row represents the location of each nucleotide. For the sake of simplicity, in this example, the hashing unit is assumed to be two nucleotides.

Table 1. The beginning 30 nucleotides of a genomic sequence

G	C	T	C	T	G	C	T	C	C	G	G	C	A	A	A	C	A	C	G	C	G	C	T	A	G	A	T	A	T
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29

Phase 1: In this phase, an ordinary linked list hashing system is generated. Since the length of the hashing unit is 2 and each nucleotide is represented by a two-digit binary number, each sequence is represented by a four-digit binary number. Therefore, the hash table will have $2^4 = 16$ rows which are numbered from 0 to 15. The first two characters of the sequence is GC which is encoded as 1001 in base 2, and it is equal to 9 in base 10. To Row 9 of the hash table a linked list is connected with its value being equal to 0, i.e., the index of the first character of the GC string on the original sequence. It is important to know that the second hashing subsequence is CT which is in Locations 1 and 2 of the sequence. Figure 3(a) represents the linked list hashing system which is produced in this phase.

Phase 2: In this phase, we first create a new hash table, *Htable*, which has two columns and 16 rows. Then, we create a one-dimensional array named *loc* with *n-1* rows, where n is the length of the reference sequence. Each row of

the *Htable* will finally show the beginning and the end indices of a range in the Array *loc* that belongs to this row of the *Htable*. To complete the information of table *loc*, we start with the first link list of the first phase and process all the lists one by one and move their data to Array *loc* in the order they are visited. For example, the first list of the previous phase has two components, 13 and 14, which are saved at Rows 0 and 1 of Array *loc*, respectively. Afterwards, Values 0 and 1 which represent the beginning and the end indices of the Array *loc* locations which are used for this list are saved at Row 0 of *Htable*, see Figure 3(b). After this phase is performed, there would be no need for first phase's information, hence the occupied memory space is freed.

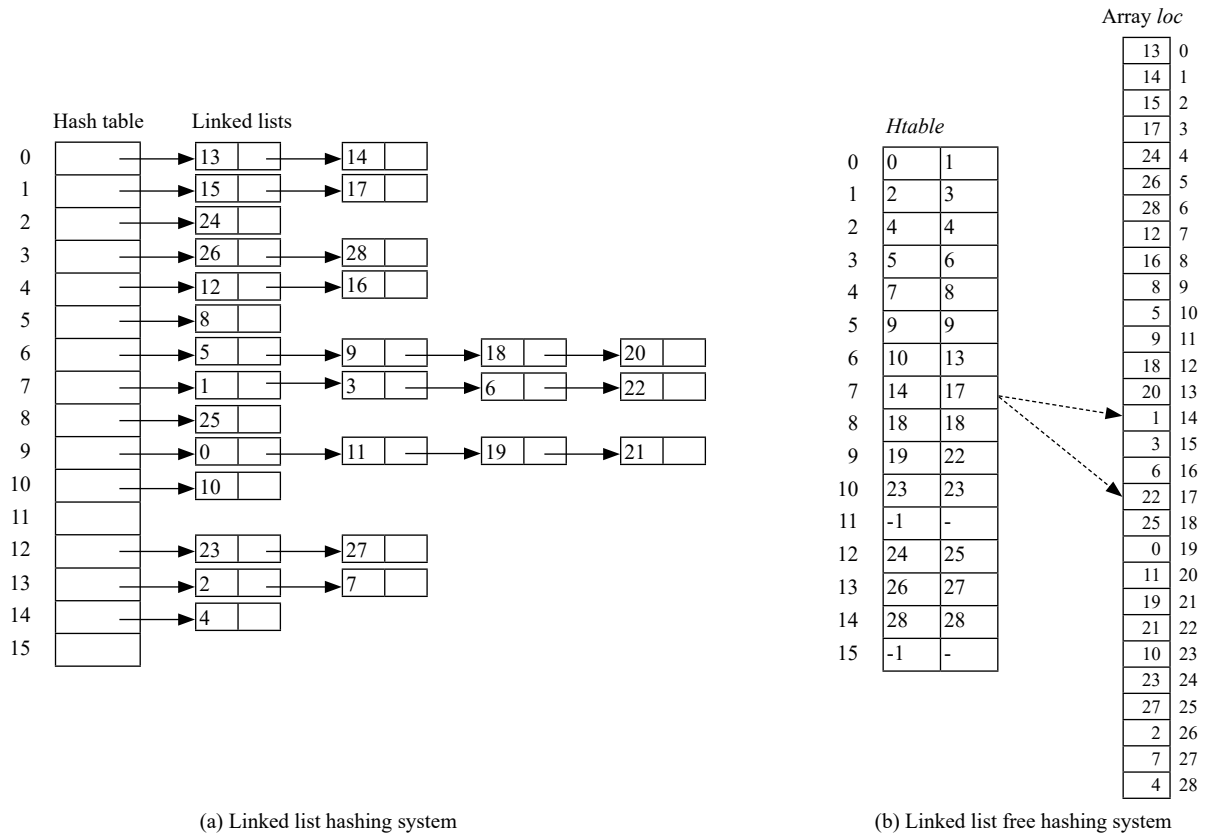

Figure 3. Conversion of linked list hash system to linked list free tables

Phase 3: To search for a specific subsequence of a sequence the tailored binary search algorithm which is presented later is used.

Now suppose we like to investigate whether ATGCTG is part of an inverted string or not. Following our previous example, let's suppose ATGCTG starts from Location *x* of Chromosome 9 of a patient. The following steps should be taken.

- Find out how far is Location *x* of Chromosome 9 of the reference genome from the beginning of the complete reference genome. If this value is represented by *y* it means, in the reference genome, the Location *x* of its Chromosome 9 is as far away as *y* from the beginning of the genome. Value *y* is an estimate because *x* is taken from patients' Chromosome 9, not from the reference genome. Since this value is non-exact, we have to develop a tailored binary search.
- Take GTCGTA (the inverse of ATGCTG) as input to hashing system of Figure 2 and find its corresponding row in hash table. If it is found in hash table, suppose the two values stored in this row are represented by variables *low* and *high*. If *low=-1* it means GTCGTA does not exist in the reference genome and further processing of this string is terminated. Otherwise, a binary search is applied to locations *low* to *high* of the Array *loc* to find *y*. Using

binary search is the main superiority of the proposed hash index in comparison to the traditional hash index. We do not expect to find the exact value *y* in these locations all the time, because *y* is just an estimate. Another clue is needed, binary search has to be modified to find the closest value to *y*, in this range of the array. The *BinSearch* algorithm follows. In *BinSearch*, if the exact *y* is not found before the last comparison, the last comparison before the algorithm terminates is the one indicating the closest value to *y*.

- If *y* is found or the value closest to *y* is acceptable, name this location as *z*. The next step is to physically go to Location *x* of Chromosome 9 of the patient and Location *z* of the reference genome and use a seed and extend method to recognize the full length inverted string. The *FindInversions* algorithm follows.

Algorithm 2. Tailored binary search to find the closest value to a given value

```
int BinSearch(int low, int high, int y)
{
    while (low < high)
    {
      m = (low+high)/2;
      if (loc[m]==y) return loc[m];
      if ( y > G[m])  low = m + 1;
      else high=m-1;
    }
    return loc[low];
}
```

In the *BinSearch*, *G* represents the human genome sequence which is a global variable. It is assumed that upon entry *low* is less than or equal to *high*. Algorithm *FindInversions* finds inversions in the Chromosome 9 of a patient using the proposed hash index of the human genome.

Algorithm 3. Finding all inversions in Chromosome 9

```
void FindInversions(int n, int m)
{
    char G[]=input(human genome); //global
    char C9[]=input(Chromosome); //global
    int b; boolean a; int low; int high; int y; int z;
    m--; //last position of C9[]
    while (m >= 5)
    {
      for (i=0; i<=5; i++)  S[i]=C9[m-i];
      m-- ;    a= code (S[0] to S[5]);
      b= convert-to-integer(a);
      low=Htable[b][0]; high=Htable[b][1];
      if (low != -1)
      {
        y=compute(m-5);  //estimate location on genome
        z= BinSearch(low, high, y);
        if (Inversion(z, m-5)) report;
      }
    }
}
```

4. Evaluation

The efficiency of the proposed linked list free indexing system is in its use of binary search which replaced the traditional hashing systems. Suppose for the developed application the number of times the hashing systems is referred to is m and the length of the longest list (in the linked list method) is k, the time complexity of the method with linked lists is $O(mk)$ while for the proposed method it is $O(m\log_2 k)$. Variable m is common in both time complexities.

Concentrating on their differences, in the worst case k is equal to the human genome's length which is $3.2 * 2^{30}$ and the method with linked list will take 2^{25} as much time as the one proposed in this paper.

On the other hand, let's take the best case of the traditional hash table with linked lists. Hash table has 2^{12} rows. The best case for linked list method is when all lists have the same size, i.e., $3.2 * \frac{2^{30}}{2^{12}} = 3.2 * 2^{18}$. In this case, the method with linked list will take 2^{15} as much time as the method presented here.

Run-time memory utilization (not space complexity) is another factor. Assume that each data value and each link takes 4 bytes of main memory. The method with linked list needs 25.6 Giga + 16k bytes of main memory. The proposed method needs 12.8 Giga + 32k of main memory. In other words, the proposed method needs about half as much main memory as the method with linked lists.

5. Conclusion

In some applications such as cryptography, hash function is the only used component of hashing systems. In most applications such as pattern search, all three components, hash function, hash table, and linked lists which are connected to hash table are essential. Traversing linked lists is a very frequent operation which causes the time complexity of hash-based algorithms to become unacceptable. In this research, a novel hashing system is proposed in which linked lists are completely removed and a one-dimensional array is built, instead. The search in linked lists with time complexity of $O(k)$ is replaced by a search with time complexity $O(\log_2 n)$ in the proposed method. In addition, the main memory space needed by the proposed method is half as much as traditional hashing methods. Finding inversions in patients' genomic sequences is used as an example to show the significance of the proposed method with respect to both time and space. The human genome is available online and it is the size of a million-page book with no index, hence it is difficult to find any information in it. The proposed method makes possible the design of an efficient index for any genomic sequence towards information retrieval or finding mutations.

Conflict of interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] Qadir AM, Varol N. A review paper on cryptography. In: *2019 7th international symposium on digital forensics and security (ISDFS)*. IEEE; 2019. p.1-6. Available from: doi: 10.1109/ISDFS.2019.8757514.

[2] Luo Y, Su Z, Zheng W, Chen Z, Wang F, Zhang Z, et al. A Novel Memory-hard Password Hashing Scheme for Blockchain-based Cyber-physical Systems. *ACM Transactions on Internet Technology*. 2021; 21(2): 1-21. Available from: doi: 10.1145/3408310.

[3] Arribas V. Beyond the limits: SHA-3 in just 49 slices. In: *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE; 2019. p.239-245. Available from: doi: 10.1109/FPL.2019.00044.

[4] Zhang DG, Gao JX, Liu XH, Zhang T, Zhao DX. Novel approach of distributed & adaptive trust metrics for MANET. *Wireless Networks*. 2019; 25: 3587-3603. Available from: doi: 10.1007/s11276-019-01955-2.

[5] Zhang D, Li G, Zheng K, Ming X, Pan ZH. An Energy-Balanced Routing Method Based on Forward-Aware Factor

for Wireless Sensor Networks. *IEEE Transactions on Industrial Informatics*. 2013; 10(1): 766-773. Available from: doi: 10.1109/TII.2013.2250910.

[6] Contreras-López O, Moyano TC, Soto DC, Gutiérrez RA. Step-by-Step Construction of Gene Co-expression Networks from High-Throughput Arabidopsis RNA Sequencing Data. In: Ristova D, Barbez E. (eds.) *Root Development. Methods in Molecular Biology, vol 1761*. New York, NY: Humana Press; 2018. p.275-301. Available from: doi:10.1007/978-1-4939-7747-5_21.

[7] Nasiri JA, Sabzekar M, Yazdi HS, Naghibzadeh M, Naghibzadeh B. Intelligent arrhythmia detection using genetic algorithm and emphatic SVM (ESVM). In: *2009 Third UKSim European Symposium on Computer Modeling and Simulation*. IEEE; 2009. p.112-117. Available from: doi: 10.1109/EMS.2009.116.

[8] National Human Genome Research Institute. *The Human Genome Project.* Available from: https://www.genome.gov/human-genome-project [Accessed 29th August 2021].

[9] National Center for Biotechnology Information (NCBI). *NCBI Home*. Available from: https://www.ncbi.nlm.nih.gov/ [Accessed 15th August 2021].

[10] Al-Hussien RA, Baker QB, Al-Ayyoub M. Fast exact sequence alignment using parallel computing. In: *2018 9th International Conference on Information and Communication Systems (ICICS)*. IEEE; 2018. p.187-191. Available from: doi: 10.1109/IACS.2018.8355464.

[11] Naghibzadeh M. *New Generation Computer Algorithms.* Independently published; 2021.

[12] Shao H, Ganesamoorthy D, Duarte T, Cao MD, Hoggart CJ, Coin LJ. npInv: accurate detection and genotyping of inversions using long read sub-alignment. *BMC Bioinformatics.* 2018; 19: 261. Available from: doi: 10.1186/s12859-018-2252-9.

[13] Mir A, Naghibzadeh M, Saadati N. INDEX: Incremental depth extension approach for protein–protein interaction networks alignment. *Biosystems.* 2017; 162: 24-34. Available from: doi: 10.1016/j.biosystems.2017.08.005.

[14] Giner-Delgado C, Villatoro S, Lerga-Jaso J, Gayà-Vidal M, Oliva M, Castellano D, et al. Evolutionary and functional impact of common polymorphic inversions in the human genome. *Nature communications*. 2019; 10: 4222. Available from: doi: 10.1038/s41467-019-12173-x.