JCSE Journal of Computing Science and Engineering

Regular Paper

Journal of Computing Science and Engineering,
Vol. 9, No. 1, March 2015, pp. 1-8

Efficient Accessing and Searching in a Sequence of Numbers

Jungjoo Seo and Myoungji Han
Department of Computer Science and Engineering, Seoul National University, Seoul, Korea
jjseo@theory.snu.ac.kr, mjhan@theory.snu.ac.kr

Kunsoo Park*
Department of Computer Science and Engineering and Korea Institute of Computer Technology, Seoul National University, Seoul, Korea
kpark@theory.snu.ac.kr

Abstract
Accessing and searching in a sequence of numbers are fundamental operations in computing that are encountered in a wide range of applications. One of the applications of the problem is cryptanalytic time-memory tradeoff which is aimed at a one-way function. A rainbow table, which is a common method for the time-memory tradeoff, contains elements from an input domain of a hash function that are normally sorted integers. In this paper, we present a practical indexing method for a monotonically increasing static sequence of numbers where the *access* and *search* queries can be addressed efficiently in terms of both time and space complexity. For a sequence of n numbers from a universe $U = \{0, ..., m-1\}$, our data structure requires $n \lg(m/n) + O(n)$ bits with constant average running time for both *access* and *search* queries. We also give an analysis of the time and space complexities of the data structure, supported by experiments with rainbow tables.

Category: Smart and intelligent computing

Keywords: Data structure; Access/search; Rank/select; Time-memory tradeoff

I. INTRODUCTION

Given a monotonically increasing sequence A of n numbers from a finite universe $U = \{0, ..., m-1\}$ of cardinality m, let us consider the following two queries.

- *access*(i) : return the i-th number in A.
- *search*(x) : return an index i such that $A[i] = x$ and -1 otherwise.

We want to answer these two queries efficiently while consuming as little space as possible on the word RAM model with the word size $\Theta(\lg m)$ bits.

One of the applications of the problem is cryptanalytic time-memory tradeoff (TMTO) which is aimed at a one-way function. In TMTO, a number of huge tables of integers are generated and stored in non-decreasing order. Storing a sorted sequence of integers is exactly the problem we want to address in this paper. There are numerous areas beyond TMTO that encounter the integer indexing problems, such as database, text indexing, and social network graphs [1].

There are several data structures that represent a

Open Access http://dx.doi.org/10.5626/JCSE.2015.9.1.1 http://jcse.kiise.org

Received 24 December 2014; **Revised** 13 February 2015; **Accepted** 17 February 2015
*Corresponding Author

†A preliminary version of this paper was presented at the 17th Japan-Korea Joint Workshop on Algorithms and Computations (WAAC 2014).

 pISSN: 1976-4677 eISSN: 2093-8020

sequence of numbers [2–8]. The *wavelet tree* represents a sequence of numbers from the range [1..r] supporting *access*, *rank*, and *select* in $O(\lg r)$ time where $r = O(\text{poly-log}(n))$. Here, $rank_c(p, V)$ returns the number of c's up to position p in V and $select_c(j, V)$ returns the position of the j-th c in V. Ferragina et al. [8] improved the time complexity to constant time using $nH_0(A) + O(n)$ bits where $H_0(A)$ is the zero-order empirical entropy of A. Brodnik and Munro [4] presented a succinct data structure that supported *search* in constant time with the space requirement of $B + O(B)$ bits where $B = \lceil \lg \binom{m}{n} \rceil$ is the *information-theoretic lower bound* of space that is needed to store a set of n elements from a universe of size m. Pagh's data structure [5] achieved constant time with the improved space of $B + O(n)$ bits. Raman et al. [7] gave a succinct data structure that also supported *rank/select* operations.

In this paper, we show that there is a simple data structure that indexes a non-decreasing sequence of integers to support not only a membership query but also a random access operation. We also give an analysis for the time and space complexity of the data structure. The average running time of the two operations is constant, assuming that *select* is done in constant time, and the required space is $n \lg(m/n) + O(n)$. While theoretical succinct data structures in the literature are very complex to implement, the data structure explained in Section III is simple to implement. In Section IV, we give an improved data structure to support multisets, exploiting the idea of [7]. Because our data structures are based on *rank/select*, we adopted multiple implementations of *rank/select* data structures [7, 9, 10] and the experimental results are presented in Section V. To verify practicality, we tested our data structures on rainbow tables.

II. PRELIMINARIES

Here, we introduce a simple method to index a monotonically non-decreasing sequence A from U that is explained in [11, 12]. This method will be called *Sindex* throughout the paper. We denote m as the size of U and n as the size of A. Then we can represent any element of U with $\lceil \lg m \rceil$ bits. Consider the s most significant bits of each number in A where $s \le \lg n$. For each integer $0 \le i < 2^s$, if we can locate the boundaries of the maximal subarray $A[l..h]$ that contains numbers having i as a prefix of their binary representation, the numbers can be stored with $(\lceil \lg m \rceil - s)$ least significant bits without loss of information. To directly determine the boundaries, we build an index table I. An index table I contains 2^s elements of size $\lceil \lg n \rceil$ bits each, and $I[i]$ is the smallest number j such that the s most significant bits of $A[j]$ is greater than or equal to i. With the index table I and the reduced integer array R of n numbers of size $(\lceil \lg m \rceil - s)$ bits each, all elements in A are stored without loss of information.

A. Access

To retrieve the value of $A[i]$ for *access*(i), suppose $A[i]$ is the concatenation of two bit strings q and r of size s and $(\lceil \lg m \rceil - s)$ bits, respectively. To compute q, we search I for the position of the largest number that is smaller than or equal to i. r can be obtained by directly accessing the reduced array R. Because the number of elements in I is 2^s, *access*(i) requires $O(s)$ time with the index table.

B. Search

Let x be the given number to search for. Also, let x be the concatenation of two bit strings x_q and x_r where the sizes of x_q and x_r are s and $\lceil \lg m \rceil - s$ bits, respectively. First, we have to find the boundaries l and h of the maximal subarray so that all the elements in $A[l..h]$ have x_q as their prefixes. l can be obtained by accessing $I[x_q]$, and h is simply $I[x_q + 1] - 1$. Note that if there are no numbers of the prefix x_q in A, then $h = l - 1$, which indicates that x does not exist. After l and h where $l \le h$ are computed, we can determine the existence and the position of x by finding the x_r in R using binary search.

C. Space Requirement

The number of bits required for the index table method is the sum of bits for two components I and R. The spaces for I and R are $2^s \lceil \lg n \rceil$ and $n(\lceil \lg m \rceil - s)$, respectively. We set s to $\lfloor \lg n - \lg \lg n \rfloor$ to minimize the space requirement for the whole data structure. Thus, the total space is $O(n(\lg \frac{m}{n} + \lg \lg n))$.

D. Time Complexity

To analyze the time complexity of *access* and *search*, we set s to $\lfloor \lg n - \lg \lg n \rfloor$ to minimize the space requirement. The required time for *access* is $O(s) = O(\lg n)$ since binary search on the I table of size 2^s takes $O(\lg 2^s)$ and accessing R takes $O(1)$. For *search*, we analyze the time complexity in the average case assuming that each element of A is chosen uniformly at random from U.

THEOREM 1. *Assume Sindex and each element of A is randomly chosen from U. Given a number $x \in U$, the binary search on R in search can be done in $O(\lg \lg n)$ time in the average case.*

Proof. Consider a fixed element $x \in U$ to search for. Now imagine choosing n numbers from U to construct A. Let X_i, $1 \le i \le n$, be a random variable such that $X_i = 1$ if the i-th chosen number has the same prefix of size s with that of x, and $X_i = 0$ otherwise. Let $X = X_1 + \ldots + X_n$, i.e., X is the random variable that represents the number of elements in A that have the same prefix as that of x. X is the size of subarray to perform binary search in *search*. When a number is chosen randomly from U, the probability that

its prefix equals to that of x is $1/2^s = 1/2^{\lfloor \lg n - \lg \lg n \rfloor}$. Thus X_i is a Bernoulli random variable with $p = 1/2^{\lfloor \lg n - \lg \lg n \rfloor}$. Because X_i's are independent and identically distributed random variables, X is a binomial random variable with parameters n and $p = 1/2^{\lfloor \lg n - \lg \lg n \rfloor}$. Thus, $E[X] = np = n/2^{\lfloor \lg n - \lg \lg n \rfloor}$. By Jensen's inequality [13],

$$\begin{aligned} E[\lg X] &\le \lg E[X] \\ &= \lg \frac{n}{2^{\lfloor \lg n - \lg \lg n \rfloor}} \\ &\le \lg \frac{n}{2^{\lg n - \lg \lg n - 1}} \\ &= \lg \lg n + 1 \\ &= O(\lg \lg n) \end{aligned}$$

□

III. PRACTICAL INDEXING

In this section, a more efficient data structure with respect to time and space complexity is explained. The improved data structure will be called *Pindex* throughout this paper. To improve the space efficiency of the index table of *Sindex*, we adopt a unary index scheme from Elias [14] and Fano [15] that is used frequently in the literature [16, 17]. As in Section II, prefixes of a fixed length of each number in the given sequence are used to construct an index. To make the content complete, we first explain the representation from [14] and give the analyses of time and space complexity.

Given a monotonically increasing sequence A of n numbers from a finite universe U, let $z = \lfloor \lg n \rfloor$ and the *quotient* q_i be the z most significant bits of $A[i]$ and the *remainder* r_i be the $\lceil \lg m \rceil - z$ least significant bits. Note that the sequence of q_i is also monotonically non-decreasing, i.e., $0 \le q_i \le q_{i+1} < 2^z$ for $1 \le i < n$. The remainders r_1, ..., r_n are stored in table R by simply concatenating them using $n(\lceil \lg m \rceil - z)$ bits. To store the quotients q_1, ..., q_n, we use the unary representation for the differences of the consecutive quotients. More specifically, q_i is encoded to $0^{q_i - q_{i-1}}1$ where $q_0 = 0$ and 0^x is the bit string consisting of x zeros. The encoded quotients are concatenated to a single bit string Q. Q requires at most $2n$ bits, because the number of 1s is n and the number of 0s is at most $2^z \le n$. Note that the number of 1s is greater than or equal to that of 0.

Before we proceed with the analysis, let us briefly introduce *rank* and *select*, because they are performed on the bit string Q for *access* and *search*. *Rank* and *select* on a bit vector V are defined as follows.

- $rank_c(p, V)$: return the number of c's up to position p in V.
- $select_c(j, V)$: return the position of the j-th c in V.

c can be any of 0 or 1. There has been extensive research on the rank/select data structure in the literature that aims to achieve optimality of time and space theoretically [7] or to give practical implementations with plentiful experiments [9, 10, 18-20].

A. Access

We perform the same procedure that was introduced in [17]. Given a query $access(i)$, q_i and r_i need to be computed to obtain $A[i]$. To compute q_i, we first compute the position of the i-th 1 in Q, and then calculate the number of 0s up to the position of the i-th 1 in Q. Because the number of 0s before the i-th 1 is $\sum_{k=1}^{i} q_i - q_{i-1}$, q_i is the number of 0s up to the i-th 1 in Q. Thus, $q_i = select_1(Q, i) - i$. r_i can be obtained by accessing R directly. The required time for *access* is $O(se)$ where se is the cost of a *select*.

B. Search

Given a query $search(x)$ where $x \in U$, let q and r be the quotient and the remainder of x, respectively. As in Section II, we first determine the boundaries l and h of the maximal subarray so that all the numbers in $A[l..r]$ have q as their prefixes. If such a subarray exists, the first q occurrences of 0 should be followed by 1 and the size of the subarray is equal to the number of consecutive 1s following the q-th 0. Thus letting i and j be $select_0(Q, q)$ and $select_0(Q, q + 1)$, respectively, l and h are computed by $l = i - q + 1$ and $h = l + j - i - 1$. Note that $h = l - 1$ if there is no number that has q as its prefix A. Once we compute the boundary, the subarray $R[l..h]$ is searched for r by binary search.

THEOREM 2. *Assume P index and each element of A is randomly chosen from U. Given a number $x \in U$, the binary search on the remainder table R in search can be done in $O(1)$ time in the average case.*

Proof. Consider the random variable X in Section II-D. Because the $p = 1/2^z = 1/2^{\lfloor \lg n \rfloor}$, $E[X] = n/2^{\lfloor \lg n \rfloor}$. By Jensen's inequality, $E[\lg X] = O(1)$. □

COROLLARY 1. *Search for a given number $x \in U$ requires $O(se)$ time in the average case where se is the cost for select on Q.*

C. Space Requirement

The data structure explained in Section III consists of three components: table R, bit string Q and an auxiliary data structure to support *select* on Q. To store table R, we need $n(\lceil \lg m \rceil - \lfloor \lg n \rfloor)$ bits. Q requires at most $2n$ bits. Thus, the total space requirement depends on the data structure that is chosen to support *select* on Q. Let $L(u, l)$ be the required space to support *select* on a bit string of

length l that contains u ones. Then the total space requirement is $O(n)+n(\lceil \lg m \rceil - \lfloor \lg n \rfloor)+L(n, 2n)$. There are many data structures in the literature that support *select* [10]. Although the space requirements of their implementations differ, one can construct the auxiliary data structure using extra space less than the size of Q. Thus, we can say that $L(n, 2n)$ is $O(n)$. By omitting ceilings and floors, the space complexity becomes

$$n \lg \frac{m}{n} + O(n).$$

IV. MULTISET

While the *Pindex* data structure can accommodate multisets, there can be high redundancy in an R table. We show another data structure the *Mindex* that reduces the space requirement in the case of multisets by the technique used in [7]. Let S_A be the set that has elements of a monotonically increasing sequence A of size n, and P be a bit vector of length n where $P[i]$ is 0 if $A[i] = A[i-1]$ and 1 otherwise. Then we build a *Pindex* of S_A and a *rank*/*select* data structure of P for 1 bits. The i-th elements of A can be obtained by performing $access(rank_1(P, i))$ on S_A using *Pindex*. To address $search(x)$ on A, we first compute $i = search(x)$ using *Pindex* of S_A, and get $select_1(P, i)$.

COROLLARY 2. *Mindex requires* $\frac{n}{k} \lg \left(\frac{mk}{n}\right) + O(n)$ *bits where* $k = \frac{n}{|S_A|}$ *is the average repeated times of the elements in A.*

V. EXPERIMENTAL RESULTS

In the experiment, we measured the average size of range for binary search on the R array for *search* to verify the theorems that we proved, and tested the actual running time of *access* and *search*. The space requirement was also measured. The average binary search range on the R array does not depend on the implementation of rank/select data structure whereas the running time and space requirement do. In addition, we show the improvement of *Mindex* on the space requirement in case of multisets. To demonstrate the efficiency of *Pindex* and *Mindex* in the real world, we conducted an experiment with rainbow tables. Various implementations of rank/select data structures were used for the experiment [7, 9, 10]. The implementation of [9], [10] and [7] are referred to as *Kim*, *Vigna*, and *RRR*, respectively. For *RRR*, we adopted the SDSL-Lite library [21], which implements *RRR* using techniques described in [19] and [20].

Table 1 shows the average size of range for binary search on R table for *search* with various sizes of sequences. For each sequence size, random sequences are generated on three different integer distributions: uniform, normal and exponential. To generate a random number, we randomly chose a real number between 0 and 1, and then multiplied it by 2^{40}. The mean and standard deviation for the normal distribution are chosen to 0.5 and 0.05, respectively. The λ for the exponential distribution was set to 6. The average size of range for binary search on R table for *search* is measured by performing *search* a million times with numbers that are chosen from the universe. As we expected from the theorems, the average range size for *Pindex* is constant while it increases as n grows for *Sindex*. Note that the average size increases linearly with lg n. It increases discretely because we use rounding function for computing s in *Sindex*.

The space requirement for each implementation is shown in Fig. 1. All implementations of *Pindex* consume less space than *Sindex*. Among the *rank*/*select* data structures, the *RRR* implementation shows the best performance in terms of space requirement. For the sequence of size 2^{28}, the *RRR*-based *Pindex* shows about 31% performance improvement compared with *Sindex* in terms of space requirement.

The measured running time of *access* is presented in Fig. 2. The x-axis is the sizes of sequences and y-axis is the time taken to perform a million *accesses*. As can be seen, all *Pindex*'s except *RRR* took less time than *Sindex*.

Table 1. The average size of range for binary search on R table for *search* with various values of n where the size of universe is 2^{40}

n	Sindex			Pindex		
	Uniform	Normal	Exponential	Uniform	Normal	Exponential
2^2	1.999	2.000	2.001	1.499	1.624	1.500
2^4	3.999	3.998	4.002	1.626	1.499	1.435
2^6	7.998	8.006	8.012	1.690	1.594	1.376
2^{10}	15.997	16.013	15.984	1.642	1.567	1.433
2^{14}	15.998	16.003	15.998	1.633	1.573	1.428
2^{18}	31.999	31.976	31.973	1.634	1.570	1.434
2^{22}	32.012	32.041	32.030	1.632	1.569	1.433
2^{26}	31.993	32.028	31.939	1.631	1.569	1.433

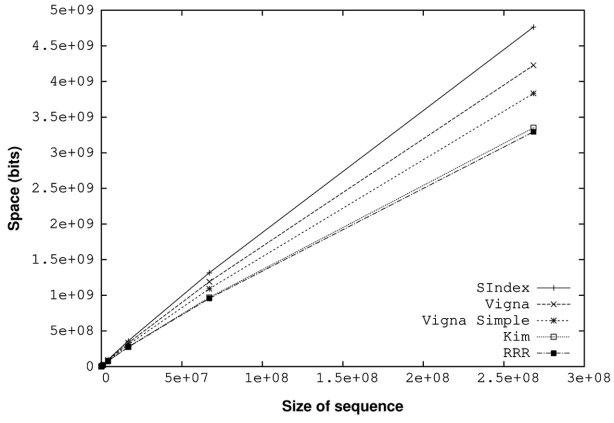

Fig. 1. Space requirement for *Sindex* and all implementations for *Pindex*.

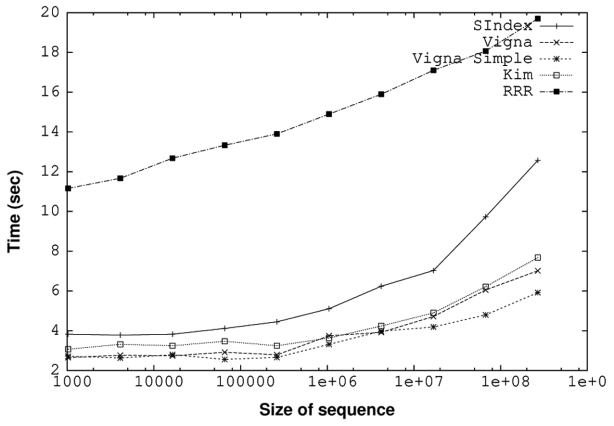

Fig. 2. Time for *access* for *Sindex* and all implementations for *Pindex*.

The difference becomes bigger when the size of sequence increases. There was no noticeable difference amongst three implementations of *Pindex* apart from *RRR*. *RRR* showed the worst performance, because *select* of *RRR* implementation [21] has $O(\lg n)$ complexity rather than constant time described in [7], where n is size of sequence.

Similarly, Fig. 3 presents the measured running time of *search*. Theoretically, *search* of *Sindex* has $O(\lg n)$ time complexity while *Vigna*, and *Kim* have constant complexity. Nevertheless, the results showed no remarkable difference amongst them. This is because the range of binary search in *search* of *Sindex* is negligibly small compared to the size of a sequence (see Table 1). *RRR* again showed the worst performance, because *select* takes $O(\lg n)$ in the implementation of *RRR* [21] which is invoked twice in *search*.

Table 2 shows the measured space requirements and running time of *access* and *search* of *Pindex* and *Mindex* on randomly generated multisets of size 2^{26} from a universe of size 2^{48}. For the *rank*/*select* data structure, we

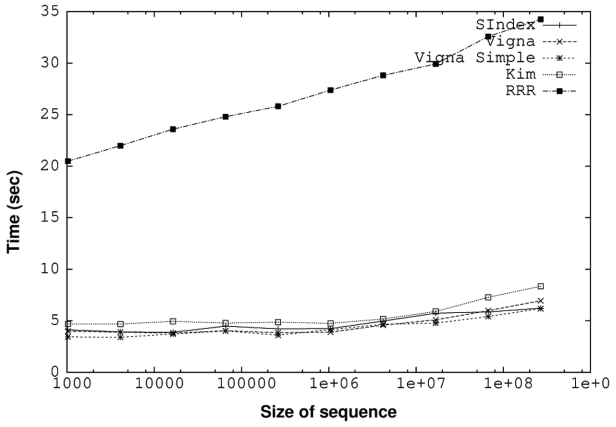

Fig. 3. Time for search for *Sindex* and all implementations for *Pindex*.

chose *RRR* and *Vigna* that showed efficiency in space and time, respectively. The running time was measured by performing *access* and *search* 10 million times with random queries. The value of improvement in Table 2 is the ratio of the space requirement of *Pindex* to that of *Mindex* with the same *rank*/*select* data structure. Because the size of the R table takes a dominant proportion of the space requirement, *Mindex* shows much better efficiency in space compared with *Pindex* in the case of multisets. Also, as the average redundancy grows, the space requirement decreases. The running times of *Mindex* for both *access* and *search* are slightly longer than those of *Pindex* because of there is one more *rank* and *select* invocation in *access* and *search*, respectively.

To demonstrate that the *Pindex* and *Mindex* are efficient in real-world applications, we tested the two data structures on rainbow tables. A rainbow table is one of the cryptanalytic time/memory tradeoff methods that aims to invert cryptographic hash functions. In a rainbow table, elements from an input domain of a hash function which are normally represented as integers are stored in sorted order. There are two types of rainbow tables: *perfect* and *non-perfect*. All elements in a perfect rainbow table are distinct while a non-perfect rainbow table may contain repeated elements. The rainbow tables that were used in the experiment were generated to invert SHA-1 hash function, and the input domain is a set of strings of length from 1 to 8 with lowercase, uppercase and digits. The size of the input domain is 221,919,451,578,090 $\approx 2^{47.657}$.

Table 3 shows the measured running time of *access* and *search*, and space requirements for *Sindex* and *Pindex* data structure for a perfect rainbow table of 80,517,490 distinct elements. Because a perfect rainbow table is a set, we do not consider *Mindex* here. Regardless of the choice of a *rank*/*select* data structure, *Pindex* consumes less space than *Sindex* as we expected. Although *RRR* has a disadvantage in run time, it outperforms in terms of the space requirement.

Table 2. The measured running time of *access* and *search* (seconds), and the space requirement (megabytes) for multisets of various average redundancies

Average redundancy			1	2	3	4
Pindex	RRR	Access	15.28	14.74	14.82	13.60
		Search	27.28	26.22	26.25	24.76
		Space (MB)	1425.37	1425.24	1425.14	1425.05
	Vigna	Access	5.96	5.11	4.47	4.77
		Search	6.73	5.98	6.13	5.64
		Space (MB)	1664.00	1664.00	1664.00	1664.00
Mindex	RRR	Access	21.02	21.31	20.36	19.88
		Search	36.86	36.88	36.2	35.23
		Space (MB)	1474.11	779.73	523.73	409.45
		Improvement	0.97	1.83	2.72	3.48
	Vigna	Access	7.21	5.21	4.87	4.43
		Search	9.17	7.54	6.73	6.32
		Space (MB)	1779.57	988.77	713.23	575.15
		Improvement	0.94	1.68	2.33	2.89

The sizes of multisets and the universe are 2^{26} and 2^{48}, respectively.

Table 3. The measured running time of *access* and *search* (seconds), and the space requirement (megabytes) for a perfect rainbow table that consists of 80,517,490 elements

	Sindex	Pindex	
		RRR	Vigna
Access	5.94	15.80	5.80
Search	7.84	28.10	6.85
Space (MB)	1666.54	1245.71	1486.35

Table 4. The measured running time of *access* and *search* (seconds), and the space requirement (megabytes) for two non-perfect rainbow tables

Size of multiset	Operation	Sindex	Pindex		Mindex	
			RRR	Vigna	RRR	Vigna
80,530,636 (1.83)	Access	7.79	16.20	5.68	23.05	6.55
	Search	6.42	28.30	6.62	39.74	8.62
	Space (MB)	1666	1245	1486	734	1004
	Improvement	-	1.34	1.12	2.27	1.66
202,331,368 (2.51)	Access	9.47	17.10	6.69	23.97	7.30
	Search	6.66	29.70	7.60	41.93	9.87
	Space (MB)	3971	2932	3488	1271	1872
	Improvement	-	1.35	1.62	2.31	1.86

To test the performance of *Mindex*, two non-perfect rainbow tables of size 80,530,636 and 202,331,368 were used for the experiment. A real number below the size of a multiset is the average redundancy of each table. The value of improvement is the ratio of the space of *Sindex* to each of *Pindex* and *Mindex* data structure. As shown in the table, all *Pindex* and *Mindex* achieved better performance in space compared with *Sindex*, and *Mindex* con-

sumes much less space than *Pindex* with both *rank*/*select* data structure.

VI. CONCLUSIONS

In this paper, we introduced two fundamental operations on a non-decreasing sequence of numbers and showed that there are efficient data structures with respect to both time and space complexity. The running times of both operations are proven to take constant time assuming that the numbers are chosen uniformly at random from their universe. We also showed that these data structures are practically efficient by performing experiments on real-world data, e.g., rainbow tables for cryptanalytic time-memory tradeoff. It would be interesting to find more applications of these data structures.

ACKNOWLEDGMENTS

This research was supported by Next-Generation Information Computing Development Program through the National Research Foundation of Korea funded by the Ministry of Science, ICT & Future Planning (2011-0029924).

REFERENCES

1. M. Curtiss, I. Becker, T. Bosman, S. Doroshenko, L. Grijincu, T. Jackson, et al., "Unicorn: a system for searching the social graph," *Proceedings of the VLDB Endowment*, vol. 6, no. 11, pp. 1150-1161, 2013.
2. A. C. C. Yao, "Should tables be sorted?" *Journal of the ACM*, vol. 28, no. 3, pp. 615-628, 1981.
3. M. L. Fredman, J. Komlós, and E. Szemerédi, "Storing a sparse table with 0(1) worst case access time," *Journal of the ACM*, vol. 31, no. 3, pp. 538-544, 1984.
4. A. Brodnik and J. I. Munro, "Membership in constant time and almost-minimum space," *SIAM Journal on Computing*, vol. 28, no. 5, pp. 1627-1640, 1999.
5. R. Pagh, "Low redundancy in static dictionaries with constant query time," *SIAM Journal on Computing*, vol. 31, no. 2, pp. 353-363, 2001.
6. L. Foschini, R. Grossi, A. Gupta, and J. S. Vitter, "When indexing equals compression: experiments with compressing suffix arrays and applications," *ACM Transactions on Algorithms*, vol. 2, no. 4, pp. 611-639, 2006.
7. R. Raman, V. Raman, and S. R. Satti, "Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets," *ACM Transactions on Algorithms*, vol. 3, no. 4, article no. 43, 2007.
8. P. Ferragina, G. Manzini, V. Makinen, and G. Navarro, "Compressed representations of sequences and full-text indexes," *ACM Transactions on Algorithms*, vol. 3, no. 2, article no. 20, 2007.
9. J. C. Na, J. E. Kim, K. Park, and D. K. Kim, "Fast computation of rank and select functions for succinct representation," *IEICE Transactions on Information and Systems*, vol. E92, no. 10, pp. 2025-2033, 2009.
10. S. Vigna, "Broadword implementation of rank/select queries," in *Proceedings of the 7th International Conference on Experimental Algorithms*, Provincetown, MA, 2008, pp. 154-168.
11. A. Biryukov, A. Shamir, and D. Wagner, "Real time cryptanalysis of A5/1 on a pc," in *Proceedings of the 7th International Workshop on Fast Software Encryption*, New York, NY, 2001, pp. 1-18.
12. J. Hong and S. Moon, "A comparison of cryptanalytic tradeoff algorithms," *Journal of Cryptology*, vol. 26, no. 4, pp. 559-637, 2012.
13. S. M. Ross, *A First Course in Probability*, 6th ed., Upper Saddle River, NJ: Prentice Hall, 2002.
14. P. Elias, "Efficient storage and retrieval by content and address of static files," *Journal of the ACM*, vol. 21, no. 2, pp. 246-260, 1974.
15. R. M. Fano, *On the Number of Bits Required to Implement an Associative Memory*, MIT Project MAC Computer Structures Group, 1971.
16. D. Belazzougui, P. Boldi, R. Pagh, and S. Vigna, "Theory and practice of monotone minimal perfect hashing," *Journal of Experimental Algorithmics*, vol. 16, article no. 3.2, 2011.
17. R. Grossi and J. S. Vitter, "Compressed suffix arrays and suffix trees with applications to text indexing and string matching," *SIAM Journal on Computing*, vol. 35, no. 2, pp. 378-407, 2005.
18. D. Okanohara and K. Sadakane, "Practical entropy-compressed rank/select dictionary," in *Proceedings of the 9th Workshop on Algorithm Engineering and Experiments (ALENEX)*, 2007, pp. 60-70.
19. F. Claude and G. Navarro, "Practical rank/select queries over arbitrary sequences," in *String Processing and Information Retrieval*, *Lecture Notes in Computer Science vol. 5280*, Heidelberg: Springer, pp. 176-187, 2009.
20. G. Navarro and E. Providel, "Fast, small, simple rank/select on bitmaps," in *Experimental Algorithms, Lecture Notes in Computer Science vol. 7276,* Heidelberg: Springer, pp. 295-306, 2012.
21. S. Gog, "SDSL-Lite," https://github.com/simongog/sdsl-lite.

Jungjoo Seo

Jungjoo Seo received his B.S. degree in Computer Science and Engineering from Sungkyunkwan University in 2009. He is currently a Ph.D. student in the Department of Computer Science and Engineering at Seoul National University. His research interests are in algorithms, computer theory, and cryptography.

Myoungji Han

Myoungji Han received his B.S. degree in Computer Science and Engineering from Seoul National University in 2010. He is currently a Ph.D. student in the Department of Computer Science and Engineering at Seoul National University. His research interests are in computer theory and string algorithms.

Kunsoo Park

Kunsoo Park received his B.S. and M.S. degrees in Computer Engineering from Seoul National University in 1983 and 1985, respectively, and Ph.D. degree in Computer Science from Columbia University in 1991. From 1991 to 1993, he was a Lecturer at King's College, University of London. He is currently a Professor in the Department of Computer Science and Engineering at Seoul National University. His research interests include design and analysis of algorithms, cryptography, and bioinformatics.