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 \( U = \{0, \ldots, m-1\} \), 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 \( U = \{0, \ldots, m-1\} \) of cardinality \( m \), let us consider the following two queries.

- \( \text{access}(i) \) : return the \( i \)-th number in \( A \).
- \( \text{search}(x) \) : 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
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(\log r) time where r = O(polylog(n)). Here, rank(p, V) returns the number of c’s up to position p in V and select(j, V) returns the position of the j-th c in V. Ferragina et al. [8] improved the time complexity to constant time using nH0(A) + O(n) bits where H0(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 \(B = \lceil \log_2 n \rceil\) 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 \(n \log (m/n) + O(n)\). 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 \([\log m]\) bits. Consider the s most significant bits of each number in \(A\) where \(s \leq \log n\). For each integer \(0 \leq i < 2^s\), if we can locate the boundaries of the maximal subarray \(A[i..h]\) that contains numbers having \(i\) as a prefix of their binary representation, the numbers can be stored with \([\log m]\) 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 \([\log n]\) bits each, and \(A[i]\) is the smallest number \(j\) such that the \(s\) most significant bits of \(A[j]\) are greater than or equal to \(i\). With the index table \(I\) and the reduced integer array \(R\) of \(n\) numbers of size \([\log m]\) 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 \([\log m]\) 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 \([\log m]\) 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 \([x_q + 1] - 1\). 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 \(l \leq h\) 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 \(2[\log n] + [\log m] - s\) respectively. We set \(s\) to \([\log n\log n]\) to minimize the space requirement for the whole data structure. Thus, the total space is \(O(n[\log m] + \log n)\).

D. Time Complexity

To analyze the time complexity of access and search, we set \(s\) to \([\log n\log n]\) to minimize the space requirement. The required time for access is \(O(s) = O(\log n)\) since binary search on the table of size \(2^s\) takes \(O(\log 2^s)\) 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 \(O(\log \log n)\) 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 \(X_i\), \(1 \leq i \leq n\), 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

http://dx.doi.org/10.5626/JCSE.2015.9.1.1
its prefix equals to that of \( x \) is \( 1/2^i = 1/2^{\lfloor \log_2 n \rfloor} \). Thus \( X_i \) is a Bernoulli random variable with \( p = 1/2^{\lfloor \log_2 n \rfloor} \).

Because \( X_i \)'s are independent and identically distributed random variables, \( X \) is a binomial random variable with parameters \( n \) and \( p = 1/2^{\lfloor \log_2 n \rfloor} \). Thus, \( E[X] = np = n/2^{\lfloor \log_2 n \rfloor} \).

By Jensen’s inequality [13],

\[
E[\log X] \leq \log E[X] = \log \frac{n}{2^{\lfloor \log_2 n \rfloor + 1}} \\
\leq \log \frac{n}{2^{\lfloor \log_2 n \rfloor}} = \log \log n + 1 \\
= O(\log \log n)
\]

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 \( \text{Pindex} \) throughout this paper. To improve the space efficiency of the index table of \( \text{Sindex} \), 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 \( z = \lceil \log n \rceil \) and the quotient \( q_i \) be the \( z \) most significant bits of \( A[i] \) and the remainder \( r \), be the \( \lceil \log m \rceil - z \) least significant bits. Note that the sequence of \( q_i \) is also monotonically non-decreasing, i.e., \( 0 \leq q_i \leq q_{i+1} < 2^z \) for \( 1 \leq i < n \). The remainders \( r_1, \ldots, r_n \) are stored in table \( R \) by simply concatenating them using \( n(\lceil \log m \rceil - z) \) bits. To store the quotients \( q_1, \ldots, q_n \), we use the unary representation for the differences of the consecutive quotients. More specifically, \( q_i \) is encoded to \( 0^{q_i-1}1 \) where \( q_0 = 0 \) and \( 0^i \) 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 \( 2^z \leq n \). 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 \( \text{rank} \) and \( \text{select} \), because they are performed on the bit string \( Q \) for \( \text{access} \) and \( \text{search} \).

- \( \text{rank}(p, V) \) : return the number of \( c \)'s up to position \( p \) in \( V \).
- \( \text{select}(j, V) \) : 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 \( \text{access}(i) \), \( q_i \) 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 \( \sum_{j=i}^{i} q_j - q_{j-1} \), \( q_i \) is the number of 0s up to the \( i \)-th 1 in \( Q \). Thus, \( q_i = \text{select}(i, Q) - i \). \( r_i \) can be obtained by accessing \( R \) directly. The required time for \( \text{access} \) is \( O(\log n) \) where \( se \) is the cost of a \( \text{select} \).

B. Search

Given a query \( \text{search}(x) \) 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 \( \text{select}(Q, q) \) and \( \text{select}(Q, q + 1) \), respectively, \( l \) and \( h \) are computed by \( l = i - q + 1 \) and \( h = j + i - 1 \). Note that \( h = l + 1 \) if there is no number that has \( q \) as its prefix \( A \). Once we compute the boundary, the subarray \( A[l..h] \) 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 \( p = 1/2^x = 1/2^{\lfloor \log n \rfloor} \), \( E[X] = n/2^{\lfloor \log n \rfloor} \).

By Jensen’s inequality, \( E[\log X] = O(1) \).

COROLLARY 1. Search for a given number \( x \in U \) requires \( O(\log \log n) \) time in the average case where \( se \) is the cost for \( \text{select} \) 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 \( \text{select} \) on \( Q \). To store table \( R \), we need \( n(\lceil \log m \rceil - \lceil \log n \rceil) \) bits. \( Q \) requires at most \( 2n \) bits. Thus, the total space requirement depends on the data structure that is chosen to support \( \text{select} \) on \( Q \). Let \( L(u, l) \) be the required space to support \( \text{select} \) on a bit string of
length L that contains n ones. Then the total space requirement is $O(n) + n \left(\lceil \log m \rceil - \lceil \log n \rceil \right) + L(n, 2n)$. There are many data structures in the literature that support select [10]. Although the space requirements of their implementations differ, one can construct the auxilary 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

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

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 $S_j$ 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 $S_j$ and a rank/select data structure of $P$ for 1 bits. The i-th elements of $A$ can be obtained by performing access(rank($P, i$)) on $S_j$ using Pindex. To address search($x$) on $A$, we first compute $i = \text{search}(x)$ using Pindex of $S_j$, and get select($P, i$).

Corollary 2. Mindex requires $\frac{n}{k} \log \left(\frac{m}{n}\right) + O(n)$ bits where $k = \frac{n}{\lceil \frac{m}{n} \rceil}$ 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 Pinex 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 lambda 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 $\log n$. 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>
<thead>
<tr>
<th>$n$</th>
<th>Uniform</th>
<th>Normal</th>
<th>Exponential</th>
<th>Uniform</th>
<th>Normal</th>
<th>Exponential</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^2$</td>
<td>1.999</td>
<td>2.000</td>
<td>2.001</td>
<td>1.499</td>
<td>1.624</td>
<td>1.500</td>
</tr>
<tr>
<td>$2^4$</td>
<td>3.999</td>
<td>3.998</td>
<td>4.002</td>
<td>1.626</td>
<td>1.499</td>
<td>1.435</td>
</tr>
<tr>
<td>$2^6$</td>
<td>7.998</td>
<td>8.006</td>
<td>8.012</td>
<td>1.690</td>
<td>1.594</td>
<td>1.376</td>
</tr>
<tr>
<td>$2^{10}$</td>
<td>15.997</td>
<td>16.013</td>
<td>15.984</td>
<td>1.642</td>
<td>1.567</td>
<td>1.433</td>
</tr>
<tr>
<td>$2^{14}$</td>
<td>15.998</td>
<td>16.003</td>
<td>15.998</td>
<td>1.633</td>
<td>1.573</td>
<td>1.428</td>
</tr>
<tr>
<td>$2^{18}$</td>
<td>31.999</td>
<td>31.976</td>
<td>31.973</td>
<td>1.634</td>
<td>1.570</td>
<td>1.434</td>
</tr>
<tr>
<td>$2^{22}$</td>
<td>32.012</td>
<td>32.041</td>
<td>32.030</td>
<td>1.632</td>
<td>1.569</td>
<td>1.433</td>
</tr>
<tr>
<td>$2^{26}$</td>
<td>31.993</td>
<td>32.028</td>
<td>31.939</td>
<td>1.631</td>
<td>1.569</td>
<td>1.433</td>
</tr>
</tbody>
</table>

http://dx.doi.org/10.5626/JCSE.2015.9.1.1

Jungjoo Seo et al.
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 \( 2^{36} \) from a universe of size \( 2^{48} \). For the rank/select data structure, we 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 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 \( 2^{21,919,451,578,090} \approx 2^{21,657} \).

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.
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-

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

<table>
<thead>
<tr>
<th>Average redundancy</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RRR</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pindex</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Access</td>
<td>15.28</td>
<td>14.74</td>
<td>14.82</td>
<td>13.60</td>
</tr>
<tr>
<td>Search</td>
<td>27.28</td>
<td>26.22</td>
<td>26.25</td>
<td>24.76</td>
</tr>
<tr>
<td>Space (MB)</td>
<td>1425.37</td>
<td>1425.24</td>
<td>1425.14</td>
<td>1425.05</td>
</tr>
<tr>
<td>Vigna</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Access</td>
<td>5.96</td>
<td>5.11</td>
<td>4.47</td>
<td>4.77</td>
</tr>
<tr>
<td>Search</td>
<td>6.73</td>
<td>5.98</td>
<td>6.13</td>
<td>5.64</td>
</tr>
<tr>
<td>Space (MB)</td>
<td>1664.00</td>
<td>1664.00</td>
<td>1664.00</td>
<td>1664.00</td>
</tr>
<tr>
<td>RRR</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mindex</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Access</td>
<td>21.02</td>
<td>21.31</td>
<td>20.36</td>
<td>19.88</td>
</tr>
<tr>
<td>Search</td>
<td>36.86</td>
<td>36.88</td>
<td>36.2</td>
<td>35.23</td>
</tr>
<tr>
<td>Space (MB)</td>
<td>1474.11</td>
<td>779.73</td>
<td>523.73</td>
<td>409.45</td>
</tr>
<tr>
<td>Vigna</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Access</td>
<td>7.21</td>
<td>5.21</td>
<td>4.87</td>
<td>4.43</td>
</tr>
<tr>
<td>Search</td>
<td>9.17</td>
<td>7.54</td>
<td>6.73</td>
<td>6.32</td>
</tr>
<tr>
<td>Space (MB)</td>
<td>1779.57</td>
<td>988.77</td>
<td>713.23</td>
<td>575.15</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Operation</th>
<th>Sindex</th>
<th>Pindex</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access</td>
<td>5.94</td>
<td>15.80</td>
</tr>
<tr>
<td>Search</td>
<td>7.84</td>
<td>28.10</td>
</tr>
<tr>
<td>Space (MB)</td>
<td>1666.54</td>
<td>1245.71</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Size of multiset</th>
<th>Operation</th>
<th>Sindex</th>
<th>Pindex</th>
<th>Mindex</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access</td>
<td>7.79</td>
<td>16.20</td>
<td>5.68</td>
<td>23.05</td>
</tr>
<tr>
<td>Search</td>
<td>6.42</td>
<td>28.30</td>
<td>6.62</td>
<td>39.74</td>
</tr>
<tr>
<td>Space (MB)</td>
<td>1666</td>
<td>1245</td>
<td>1486</td>
<td>734</td>
</tr>
<tr>
<td>Improvement</td>
<td>-</td>
<td>1.34</td>
<td>1.12</td>
<td>2.27</td>
</tr>
<tr>
<td>Access</td>
<td>9.47</td>
<td>17.10</td>
<td>6.69</td>
<td>23.97</td>
</tr>
<tr>
<td>Search</td>
<td>6.66</td>
<td>29.70</td>
<td>7.60</td>
<td>41.93</td>
</tr>
<tr>
<td>Space (MB)</td>
<td>3971</td>
<td>2932</td>
<td>3488</td>
<td>1271</td>
</tr>
<tr>
<td>Improvement</td>
<td>-</td>
<td>1.35</td>
<td>1.62</td>
<td>2.31</td>
</tr>
</tbody>
</table>

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

**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.