CRAM: Compressed Random Access Memory

Jesper Jansson[1], Kunihiko Sadakane[2], and Wing-Kin Sung[3]

[1] Laboratory of Mathematical Bioinformatics, Institute for Chemical Research, Kyoto University, Gokasho, Uji, Kyoto 611-0011, Japan
jj@kuicr.kyoto-u.ac.jp
[2] National Institute of Informatics, 2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo 101-8430, Japan
sada@nii.ac.jp
[3] National University of Singapore, 13 Computing Drive, Singapore 117417
ksung@comp.nus.edu.sg

Abstract. We present a new data structure called the *Compressed Random Access Memory* (CRAM) that can store a dynamic string T of characters, e.g., representing the memory of a computer, in compressed form while achieving asymptotically almost-optimal bounds (in terms of empirical entropy) on the compression ratio. It allows short substrings of T to be decompressed and retrieved efficiently and, significantly, characters at arbitrary positions of T to be modified quickly during execution *without decompressing the entire string*. This can be regarded as a new type of data compression that can update a compressed file directly. Moreover, at the cost of slightly increasing the time spent per operation, the CRAM can be extended to also support insertions and deletions. Our key observation that the empirical entropy of a string does not change much after a small change to the string, as well as our simple yet efficient method for maintaining an array of variable-length blocks under length modifications, may be useful for many other applications as well.

1 Introduction

Certain modern-day information technology-based applications require random access to very large data structures. For example, to do genome assembly in bioinformatics, one needs to maintain a huge graph [18]. Other examples include dynamic programming-based problems, such as optimal sequence alignment or finding maximum bipartite matchings, which need to create large tables (often containing a lot of redundancy). Yet another example is in image processing, where one sometimes needs to edit a high-resolution image which is too big to load into the main memory of a computer all at once. Additionally, a current trend in the mass consumer electronics market is cheap mobile devices with limited processing power and relatively small memories; although these are not designed to process massive amounts of data, it could be economical to store non-permanent data and software on them more compactly, if possible.

The standard solution to the above problem is to employ secondary memory (disk storage, etc.) as an extension of the main memory of a computer. This

A. Czumaj et al. (Eds.): ICALP 2012, Part I, LNCS 7391, pp. 510–521, 2012.

technique is called *virtual memory*. The drawback of virtual memory is that the processing time will be slowed down since accessing the secondary memory is an order of magnitude slower than accessing the main memory. An alternative approach is to compress the data T and store it in the main memory. By using existing data compression methods, T can be stored in $nH_k + o(n \log \sigma)$-bits space [2,8] for every $0 \leq k < \log_\sigma n$, where n is the length of T, σ is the size of the alphabet, and $H_k(T)$ denotes the k-th order empirical entropy of T. Although greatly reducing the amount of storage needed, it does not work well because it becomes computationally expensive to access and update T.

Motivated by applications that would benefit from having a large virtual memory that supports fast access- and update-operations, we consider the following task: Given a memory/text $T[1..n]$ over an alphabet of size σ, maintain a data structure that stores T compactly while supporting the following operations. (We assume that $\ell = \Theta(\log_\sigma n)$ is the length of one machine word.)

- `access`(T, i): Return the substring $T[i..(i + \ell - 1)]$.
- `replace`(T, i, c): Replace $T[i]$ by a character $c \in [\sigma]$. [1]
- `delete`(T, i): Delete $T[i]$, i.e., make T one character shorter.
- `insert`(T, i, c): Insert a character c into T between positions $i-1$ and i, i.e., make T one character longer.

Compressed Read Only Memory: When only the `access` operation is supported, we call the data structure *Compressed Read Only Memory*. Sadakane and Grossi [17], González and Navarro [6], and Ferragina and Venturini [4] developed storage schemes for storing a text succinctly that allow constant-time access to any word in the text. More precisely, these schemes store $T[1..n]$ in $nH_k + \mathcal{O}\left(n \log \sigma \left(\frac{k \log \sigma + \log \log n}{\log n}\right)\right)$ bits[2] and `access`(T, i) takes $\mathcal{O}(1)$ time, and both the space and access time are optimal for this task. Note, however, that none of these schemes allow T to be modified.

Compressed Random Access Memory (CRAM): When the operations `access` and `replace` are supported, we call the data structure *Compressed Random Access Memory* (CRAM). As far as we know, it has not been considered previously in the literature, even though it appears to be a fundamental and important data structure.

Extended CRAM: When all four operations are supported, we call the data structure *extended CRAM*. It is equivalent to *the dynamic array* [16] and also solves *the list representation problem* [5]. Fredman and Saks [5] proved a cell probe lower bound of $\Omega(\log n / \log \log n)$ time for the latter, and also showed that $n^{\Omega(1)}$ update time is needed to support constant-time `access`. Raman *et al.* [16] presented an $n \log \sigma + o(n \log \sigma)$-bit data structure which supports `access`, `replace`, `delete`, and `insert` in $\mathcal{O}(\log n / \log \log n)$ time. Navarro and Sadakane [15] recently gave a data structure using $nH_0(T) + \mathcal{O}(n \log \sigma / \log^\epsilon n + \sigma \log^\epsilon n)$ bits that supports `access`, `delete`, and `insert` in $\mathcal{O}(\frac{\log n}{\log \log n}(1 + \frac{\log \sigma}{\log \log n}))$ time.

[1] The notation $[\sigma]$ stands for the set $\{1, 2, \ldots, \sigma\}$.

[2] Reference [17] has a slightly worse space complexity.

1.1 Our Contributions

This paper studies the complexity of maintaining the CRAM and extended CRAM data structures. We assume the uniform-cost word RAM model with word size $w = \Theta(\log n)$ bits, i.e., standard arithmetic and bitwise boolean operations on w-bit word-sized operands can be performed in constant time [9]. Also, we assume the memory consists of a sequence of bits, and each bit is identified with an address in $0, \ldots, 2^w - 1$. Furthermore, any consecutive w bits can be accessed in constant time. (Note that this memory model is equivalent under the word RAM model to a standard memory model consisting of a sequence of words of some fixed length.) At any time, if the highest address of the memory used by the algorithm is s, the space used by the algorithm is said to be $s + 1$ bits [10].

Our main results for the CRAM are summarized in:

Theorem 1. *Given a text $T[1..n]$ over an alphabet of size σ and any $\epsilon > 0$, after $\mathcal{O}(n \log \sigma / \log n)$ time preprocessing, the CRAM data structure for $T[1..n]$ can be stored in $nH_k(T) + \mathcal{O}\left(n \log \sigma \left((k+1)\epsilon + \frac{k \log \sigma + \log\log n}{\log n}\right)\right)$ bits for every $0 \leq k < \log_\sigma n$ simultaneously, where $H_k(T)$ denotes the k-th order empirical entropy of T, while supporting* $\texttt{access}(T, i)$ *in $\mathcal{O}(1)$ time and* $\texttt{replace}(T, i, c)$ *for any character c in $\mathcal{O}(1/\epsilon)$ time.*

Theorem 1 is proved in Section 5 below.

Next, by setting $\epsilon = \max\{\frac{\log \sigma}{\log n}, \frac{\log\log n}{(k+1)\log n}\}$, we obtain:

Corollary 1. *Given a text $T[1..n]$ over an alphabet of size σ and any $k = o(\log_\sigma n)$, after $\mathcal{O}(n \log \sigma / \log n)$ time preprocessing, the CRAM data structure for $T[1..n]$ can be stored in $nH_k(T) + \mathcal{O}\left(n \log \sigma \cdot \frac{k \log \sigma + \log\log n}{\log n}\right)$ bits while supporting* $\texttt{access}(T, i)$ *in $\mathcal{O}(1)$ time and* $\texttt{replace}(T, i, c)$ *for any character c in $\mathcal{O}(\min\{\log_\sigma n, (k+1)\log n / \log\log n\})$ time.*

For the extended CRAM, we have:

Theorem 2. *Given a text $T[1..n]$ over an alphabet of size σ, after spending $\mathcal{O}(n \log \sigma / \log n)$ time on preprocessing, the extended CRAM data structure for $T[1..n]$ can be stored in $nH_k(T) + \mathcal{O}\left(n \log \sigma \cdot \frac{k \log \sigma + (k+1)\log\log n}{\log n}\right)$ bits for every $0 \leq k < \log_\sigma n$ simultaneously, where $H_k(T)$ denotes the k-th order empirical entropy of T, while supporting all four operations in $\mathcal{O}(\log n / \log\log n)$ time.*

(Due to space limitations, the proof of Theorem 2 has been omitted from the conference version of our paper.)

Table 1 shows a comparison with existing data structures. Many existing dynamic data structures for storing compressed strings [7,11,13,15] use the fact $nH_0(S) = \log \binom{n}{n_1, \ldots, n_\sigma}$ where n_c is the number of occurrences of character c in the string S. However, this approach is helpful for small alphabets only because of the size of the auxiliary data. For large alphabets, generalized wavelet trees [3] can be used to decompose a large alphabet into smaller ones, but this slows down

Table 1. Comparison between previously existing data structures and the new ones in this paper. For simplicity, we assume $\sigma = o(n)$. The upper table lists results for the Compressed Read Only Memory (the first line) and the CRAM (the second and third lines), and the lower table lists results for the extended CRAM.

`access`	`replace`	Space (bits)	Ref.
$\mathcal{O}(1)$	—	$nH_k(T) + \mathcal{O}\left(n \log \sigma \cdot \frac{k \log \sigma + \log \log n}{\log n}\right)$	[4,6]
$\mathcal{O}(1)$	$\mathcal{O}(\min\{\log_\sigma n, \frac{(k+1)\log n}{\log \log n}\})$	$nH_k(T) + \mathcal{O}\left(n \log \sigma \cdot \frac{k \log \sigma + \log \log n}{\log n}\right)$	New
$\mathcal{O}(1)$	$\mathcal{O}(\frac{1}{\epsilon})$	$nH_k(T) +$ $\mathcal{O}\left(n \log \sigma \left(\frac{k \log \sigma + \log \log n}{\log n} + (k+1)\epsilon\right)\right)$	New

`access/replace/insert/delete`	Space (bits)	Ref.
$\mathcal{O}(\frac{\log^2 n}{\log \sigma})$	$nH_k(T) + o(n \log \sigma)$	[15]
$\mathcal{O}(\frac{\log \sigma \log n}{(\log \log n)^2})$	$nH_0(T) + \mathcal{O}\left(n \log \sigma \cdot \frac{1}{\log^\epsilon n}\right)$	[15]
$\mathcal{O}(\frac{\log n}{\log \log n})$	$nH_0(T) + \mathcal{O}\left(n \log \sigma \cdot \frac{\log \log n}{\log n}\right)$	New
$\mathcal{O}(\frac{\log n}{\log \log n})$	$nH_k(T) + \mathcal{O}\left(n \log \sigma \cdot \frac{k \log \sigma + (k+1) \log \log n}{\log n}\right)$	New

the access and update times. For example, if $\sigma = \sqrt{n}$, the time complexity of those data structures is $\mathcal{O}((\log n / \log \log n)^2)$, while ours is $\mathcal{O}(\log n / \log \log n)$, or even constant. Also, a technical issue when using large alphabets is how to update the code tables for encoding characters to achieve the entropy bound. Code tables that achieve the entropy bound will change when the string changes, and updating the entire data structure with the new code table is time-consuming.

Our results depend on a new analysis of the empirical entropies of *similar* strings in Section 3. We prove that *the empirical entropy of a string does not change a lot after a small change to the string* (Theorem 4). By using this fact, we can delay updating the entire code table. Thus, after each update operation to the string, we just change a part of the data structure according to the new code table. In Section 5, we show that the redundancy in space usage by this method is negligible, and we obtain Theorem 1.

Looking at Table 1, we observe that Theorem 1 can be interpreted as saying that for arbitrarily small, fixed $\epsilon > 0$, by spending $\mathcal{O}(n \log \sigma \cdot \epsilon(k+1))$ bits space more than the best existing data structures for Compressed Read Only Memory, we can also get $\mathcal{O}(1/\epsilon)$ (i.e., constant) time `replace` operations.

1.2 Organization of the Paper

Section 2 reviews the definition of the empirical entropy of a string and the data structure of Ferragina and Venturini [4]. In Section 3, we prove an important result on the empirical entropies of similar strings. In Section 4, we describe a technique for maintaining an array of variable-length blocks. Section 5 explains how to implement the CRAM to achieve the bounds stated in Theorems 1 above. Finally, Section 6 gives some concluding remarks.

2 Preliminaries

2.1 Empirical Entropy

The compression ratio of a data compression method is often expressed in terms of the *empirical entropy* of the input strings [12]. We first recall the definition of this concept. Let T be a string of length n over an alphabet $\mathcal{A} = [\sigma]$. Let n_c be the number of occurrences of $c \in \mathcal{A}$ in T. Let $\{P_c = n_c/n\}_{c=1}^{\sigma}$ be the empirical probability distribution for the string T. The 0-th order empirical entropy of T is defined as $H_0(T) = -\sum_{c=1}^{\sigma} P_c \log P_c$. We also use $H_0(p)$ to denote the 0-th order empirical entropy of a string whose empirical probability distribution is p.

Next, let k be any non-negative integer. If a string $s \in \mathcal{A}^k$ precedes a symbol c in T, s is called the *context of* c. We denote by $T^{(s)}$ the string that is the concatenation of all symbols, each of whose context in T is s. The k-th order empirical entropy of T is defined as $H_k(T) = \frac{1}{n}\sum_{s\in\mathcal{A}^k} |T^{(s)}|H_0(T^{(s)})$. It was shown in [14] that for any $k \geq 0$, $H_k(T) \geq H_{k+1}(T)$ holds, and $nH_k(T)$ is a lower bound for the output size of any compressor that encodes each symbol of T with a code that only depends on the symbol and its context of length k.

To prove our new results, we shall use the following theorem in Section 3:

Theorem 3 ([1, Theorem 16.3.2]). *Let p and q be two probability mass functions on $\mathcal{A}$ such that $||p-q||_1 \equiv \sum_{c\in\mathcal{A}} |p(c)-q(c)| \leq \frac{1}{2}$. Then $|H_0(p)-H_0(q)| \leq -||p-q||_1 \log \frac{||p-q||_1}{|\mathcal{A}|}$.*

The technique of *blocking*, i.e., to conceptually merge consecutive symbols to form new symbols over a larger alphabet, is used to reduce the redundancy of Huffman encoding for compressing a string. A string T of length n is partitioned into $\frac{n}{\ell}$ blocks of length ℓ each, then Huffman or other entropy codings are applied to compress a new string T_ℓ of those blocks. We call this operation *blocking of length* ℓ.

2.2 Review of Ferragina and Venturini's Data Structure

Here, we briefly review the data structure of Ferragina and Venturini from [4]. It uses the same basic idea as Huffman coding: replace every fixed-length block of symbols by a variable-length code in such a way that frequently occurring blocks get shorter codes than rarely occurring blocks.

To be more precise, consider a text $T[1..n]$ over an alphabet $\mathcal{A}$ where $|\mathcal{A}| = \sigma$ and $\sigma < n$. Let $\ell = \frac{1}{2}\log_\sigma n$ and $\tau = \log n$. Partition $T[1..n]$ into $\frac{n}{\tau\ell}$ super-blocks, each contains $\tau\ell$ characters. Each super-block is further partitioned into τ blocks, each contains ℓ characters. Denote the $\frac{n}{\ell}$ blocks by $T_i = T[(i-1)\ell + 1..i\ell]$ for $i = 1, 2, \ldots, n/\ell$.

Since each block is of length ℓ, there are at most $\sigma^\ell = \sqrt{n}$ distinct blocks. For each block $P \in \mathcal{A}^\ell$, let $f(P)$ be the frequency of P in $\{T_1, \ldots, T_{n/\ell}\}$. Let $r(P)$ be the rank of P according to the decreasing frequency, i.e., the number of distinct blocks P' such that $f(P') \geq f(P)$, and $r^{-1}(j)$ be its inverse function. Let $enc(j)$ be the rank j-th binary string in $[\epsilon, 0, 1, 00, 01, 10, 11, 000, \ldots]$.

The data structure of Ferragina and Venturini consists of four arrays:

- $V = enc(r(T_1)) \dots enc(r(T_{n/\ell}))$.
- $r^{-1}(j)$ for $j = 1, \dots, \sqrt{n}$.
- Table $T_{Sblk}[1..\frac{n}{\ell\tau}]$ stores the starting position in V of the encoding of every super-block.
- Table $T_{blk}[1..\frac{n}{\ell}]$ stores the starting position in V of the encoding of every block relative to the beginning of its enclosing super-block.

The algorithm for $\texttt{access}(T, i)$ is simple: Given i, compute the address where the block for $T[i]$ is encoded by using T_{Sblk} and T_{blk} and obtain the code which encodes the rank of the block. Then, from r^{-1}, obtain the substring. In total, this takes $\mathcal{O}(1)$ time. This yields:

Lemma 1 ([4]). *Any substring $T[i..j]$ can be retrieved in $\mathcal{O}(1+(j-i+1)/\log_\sigma n)$ time.*

Using the data structure of Ferragina and Venturini, $T[1..n]$ can be encoded using $nH_k + \mathcal{O}(\frac{n}{\log_\sigma n}(k \log \sigma + \log\log n))$ bits according to the next lemma.

Lemma 2 ([4]). *The space needed by V, r^{-1}, T_{Sblk}, and T_{blk} is as follows:*

- *V is of length $nH_k + 2 + \mathcal{O}(k \log n) + \mathcal{O}(nk \log \sigma/\ell)$ bits, simultaneously for all $0 \le k < \log_\sigma n$.*
- *$r^{-1}(j)$ for $j = 1, \dots, \sqrt{n}$ can be stored in $\sqrt{n} \log n$ bits.*
- *$T_{Sblk}[1..\frac{n}{\ell\tau}]$ can be stored in $\mathcal{O}(\frac{n}{\ell})$ bits.*
- *$T_{blk}[1..\frac{n}{\ell}]$ can be stored in $\mathcal{O}(\frac{n}{\ell} \log\log n)$ bits.*

3 Entropies of Similar Strings

In this section, we prove that the empirical entropy of a string does not change much after a small change to it. This result will be used to bound the space complexity of our main data structure in Section 5.4. Consider two strings T and T' of length n and n', respectively, such that the edit distance between T and T' is one. That is, T' can be obtained from T by replacement, insertion, or deletion of one character. We show that the empirical entropies of the two strings do not differ so much.

Theorem 4. *For two strings T and T' of length n and n', respectively, over an alphabet $\mathcal{A}$ such that the edit distance between T and T' is one, it holds for any integer $k \ge 0$ that $|nH_k(T) - n'H_k(T')| = \mathcal{O}((k+1)(\log n + \log|\mathcal{A}|))$.*

To prove Theorem 4, we first prove the following:

Lemma 3. *Let T be a string of length n over an alphabet $\mathcal{A}$, T^- be a string made by deleting a character from T at any position, T^+ be a string made by inserting a character into T at any position, and T' be a string by replacing a character of T into another one at any position. Then the following holds:*

$$|nH_0(T) - (n-1)H_0(T^-)| \le 4\log n + 3\log|\mathcal{A}| \quad (\text{if } n \ge 1) \tag{1}$$

$$|nH_0(T) - (n+1)H_0(T^+)| \le 4\log(n+1) + 4\log|\mathcal{A}| \quad (\text{if } n \ge 0) \tag{2}$$

$$|nH_0(T) - nH_0(T')| \le 4\log(n+1) + 3\log|\mathcal{A}| \quad (\text{if } n \ge 0) \tag{3}$$

Proof. Let $P(x)$, $P^-(x)$, $P^+(x)$, and $P'(x)$ denote the empirical probability of a character $x \in \mathcal{A}$ in T, T^-, T^+, and T', respectively, and let n_x denote the number of occurrences of $x \in \mathcal{A}$ in T. It holds that $P(x) = \frac{n_x}{n}$ for any $x \in \mathcal{A}$.

If a character c is removed from T, then $P^-(c) = \frac{n_c-1}{n-1}$, and $P^-(x) = \frac{n_x}{n-1}$ for any other $x \in \mathcal{A}$. Then $||P - P^-||_1 = \frac{n-n_c}{n(n-1)} + \sum_{x\in\mathcal{A},x\neq c} \frac{n_x}{n(n-1)} = \frac{2(n-n_c)}{n(n-1)}$. If $n = 1$, it holds $H_0(T) = 0$, and therefore $nH_0(T) - (n-1)H_0(T^-) = 0$ and the claim holds. If $n = n_c$, which means that all characters in T are c, it holds $H_0(T) = H_0(T^-) = 0$ and the claim holds. Otherwise, $\frac{2}{n(n-1)} \leq ||P - P^-||_1 \leq \frac{2}{n}$ holds. If $||P - P^-||_1 \leq \frac{1}{2}$, from Theorem 3, $|H_0(P) - H_0(P^-)| \leq -||P - P^-||_1 \log \frac{||P-P^-||_1}{|\mathcal{A}|} \leq \frac{2}{n} \log \frac{|\mathcal{A}|n(n-1)}{2}$. Then $|nH_0(T) - (n-1)H_0(T^-)| \leq n|H_0(P) - H_0(P^-)| + H_0(P^-) \leq 4 \log n + 3 \log |\mathcal{A}|$. If $||P - P^-||_1 > \frac{1}{2}$, which implies $n < 4$, $|nH_0(T) - (n-1)H_0(T^-)| \leq 3 \log |\mathcal{A}|$. This proves the claim for T^-.

If a character c is inserted into T, then $P^+(c) = \frac{n_c+1}{n+1}$, and $P^+(x) = \frac{n_x}{n+1}$ for any other $x \in \mathcal{A}$. Then $||P - P^+||_1 = \frac{2(n-n_c)}{n(n+1)}$. If $n = 0$, $H_0(T) = H_0(T^+) = 0$ and the claim holds. If $n = n_c$, which means that T^+ consists of only the character c, $H_0(T) = H_0(T^+) = 0$ and the claim holds. Otherwise, $\frac{2}{n(n-1)} \leq ||P-P^+||_1 \leq \frac{2}{n}$ holds. If $||P-P^+||_1 \leq \frac{1}{2}$, $|nH_0(T)-(n+1)H_0(T^+)| \leq n|H_0(P)-H_0(P^+)| + H_0(P^-) \leq 4 \log n + 3 \log |\mathcal{A}|$. If $||P - P^+||_1 > \frac{1}{2}$, which implies $n < 4$, $|nH_0(T) - (n+1)H_0(T^+)| \leq 4 \log |\mathcal{A}|$. This proves the claim for T^+.

If a character c of T is replaced with another character $c' \in \mathcal{A}$ ($c' \neq c$), then $||P - P'||_1 = \sum_{\alpha\in\mathcal{A}} |P(\alpha) - P'(\alpha)| = |\frac{n_c}{n} - \frac{n_c-1}{n}| + |\frac{n_{c'}}{n} - \frac{n_{c'}+1}{n}| = \frac{2}{n}$. If $||P - P'||_1 \leq \frac{1}{2}$, $|nH_0(T) - nH_0(T')| \leq n|H_0(P) - H_0(P')| \leq 4 \log n + 2 \log |\mathcal{A}|$. If $||P-P'||_1 > \frac{1}{2}$, which implies $n < 4$, $|nH_0(T)-nH_0(T')| \leq 3 \log |\mathcal{A}|$. If $c' = c$, $T' = T$ and $|nH_0(T) - nH_0(T')| = 0$. This completes the proof. □

Proof. (of Theorem 4) From the definition of the empirical entropy, $nH_k(T) = \sum_{s\in\mathcal{A}^k} |T^{(s)}|H_0(T^{(s)})$. Therefore, for each context $s \in \mathcal{A}^k$, we estimate the change of 0-th order entropy. Because the edit distance between T and T' is one, we can write $T = T_1cT_2$ and $T' = T_1c'T_2$ using two (possibly empty) strings T_1, T_2 and two (possibly empty) characters c, c'. For the context $T_1[n_1-k+1..n_1]$ ($n_1 = |T_1|$), denoted by s_0, the character c in the string $T^{(s_0)}$ will change to c'. The character $T_2[i]$ ($i = 1, 2, \ldots, k$) has the context $T_1[n_1 - k + 1 + i..n_1]cT_2[1..i-1]$, denoted by s_i, in T, but the context will change to $s'_i = T_1[n_1 - k + 1 + i..n_1]c'T_2[1..i-1]$ in T'. Thus, a character $T_2[i]$ is removed from the string $T^{(s_i)}$ and inserted into $T'^{(s_i)}$. Therefore, the entropies will change in at most $2k + 1$ strings $(T^{(s_0)}, T^{(s_1)}, \ldots, T^{(s_k)}, T'^{(s_1)}, \ldots, T'^{(s_k)})$. By Lemma 3, each one will change only $\mathcal{O}(\log n + \log |\mathcal{A}|)$. This proves the claim. □

4 Memory Management

This section presents a data structure for storing a set B of m variable-length strings over the alphabet $\{0, 1\}$, which is an extension of the one in [15]. The

data structure allows the contents of the strings and their lengths to change, but the value of m must remain constant. We assume a unit-cost word RAM model with word size w bits. The memory consists of consecutively ordered bits, and any consecutive w bits can be accessed in constant time, as stated above. A string over $\{0,1\}$ of length at most b is called a $(\leq b)$-*block*. Our data structure stores a set B of m such $(\leq b)$-blocks, while supporting the following operations:

- `address`(i): Return a pointer to where in the memory the i-th $(\leq b)$-block is stored $(1 \leq i \leq m)$.
- `realloc`(i, b'): Change the length of the i-th $(\leq b)$-block to b' bits $(0 \leq i \leq m)$. The physical address for storing the block (`address`(i)) may change.

Theorem 5. *Given that $b \leq m$ and $\log m \leq w$, consider the unit-cost word RAM model with word size w. Let $B = \{B[1], B[2], \ldots, B[m]\}$ be a set of $(\leq b)$-blocks and let s be the total number of bits of all $(\leq b)$-blocks in B. We can store B in $s + \mathcal{O}(m \log m + b^2)$ bits while supporting* `address` *in $\mathcal{O}(1)$ time and* `realloc` *in $\mathcal{O}(b/w)$ time.*

Theorem 6. *Given a parameter $b = \mathcal{O}(w)$, consider the unit-cost word RAM model with word size w. Let $B = \{B[1], B[2], \ldots, B[m]\}$ be a set of $(\leq b)$-blocks, and let s be the total number of bits of all $(\leq b)$-blocks in B. We can store B in $s + \mathcal{O}(w^4 + m \log w)$ bits while supporting* `address` *and* `realloc` *in $\mathcal{O}(1)$ time.*

(Due to lack of space, the proofs of Theorems 5 and 6 have been omitted from the conference version of our paper.)

From here on, we say that the data structure has parameters (b, m).

5 A Data Structure for Maintaining the CRAM

This section is devoted to proving Theorem 1. Our aim is to dynamize Ferragina and Venturini's data structure [4] by allowing `replace` operations. Ferragina and Venturini's data structure uses a code table for encoding the string, while our data structure uses two code tables, which will change during update operations.

Given a string $T[1..n]$ defined over an alphabet $\mathcal{A}$ $(|\mathcal{A}| = \sigma)$, we support two operations. (1) `access`(T, i): which returns $T[i..i + \frac{1}{2}\log_\sigma n - 1]$; and (2) `replace`$(T, i, c)$: which replaces $T[i]$ with a character $c \in \mathcal{A}$.

We use blocking of length $\ell = \frac{1}{2}\log_\sigma n$ of T. Let $T'[1..n']$ be a string of length $n' = \frac{n}{\ell}$ on an alphabet $\mathcal{A}^\ell$ made by blocking of T. The alphabet size is $\sigma^\ell = \sqrt{n}$. Each character $T'[i]$ corresponds to the string $T[((i-1)\ell+1)..i\ell]$. A super-block consists of $1/\epsilon$ consecutive blocks in T' (ℓ/ϵ consecutive characters in T), where ϵ is a predefined constant.

Our algorithm runs in phases. Let $n'' = \epsilon n'$. For every $j \geq 1$, we refer to the sequence of the $(n''(j-1)+1)$-th to $(n''j)$-th replacements as *phase* j. The preprocessing stage corresponds to phase 0. Let $T^{(j)}$ denote the string just before phase j. (Hence, $T^{(1)}$ is the input string T.) Let $F^{(j)}$ denote the frequency table of blocks $b \in \mathcal{A}^\ell$ in $T^{(j)}$, and $C^{(j)}$ and $D^{(j)}$ a code table and a decode table defined

below. The algorithm also uses a bit-vector $R^{(j-1)}[1..n'']$, where $R^{(j-1)}[i] = 1$ means that the i-th super-block in T is encoded by code table $C^{(j-1)}$; otherwise, it is encoded by code table $C^{(j-2)}$.

During the execution of the algorithm, we maintain the following invariant:

- At the beginning of phase j, the string $T^{(j)}$ is encoded with code table $C^{(j-2)}$ (we assume $C^{(-1)} = C^{(0)} = C^{(1)}$), and the table $F^{(j)}$ stores the frequencies of blocks in $T^{(j)}$.
- During phase j, the i-th super-block is encoded with code table $C^{(j-2)}$ if $R^{(j-1)}[i] = 0$, or $C^{(j-1)}$ if $R^{(j-1)}[i] = 1$. The code tables $C^{(j-2)}$ and $C^{(j-1)}$ do not change.
- During phase j, $F^{(j+1)}$ stores the correct frequency of blocks of the current T.

5.1 Phase 0: Preprocessing

First, for each block $b \in \mathcal{A}^\ell$, we count the numbers of its occurrences in T' and store it in an array $F^{(1)}[b]$. Then we sort the blocks $b \in \mathcal{A}^\ell$ in decreasing order of the frequencies $F^{(1)}[b]$, and assign a code $C^{(1)}[b]$ to encode them. The code for a block b is defined as follows. If the length of the code $enc(b)$, defined in Section 2.2, is at most $\frac{1}{2}\log n$ bits, then $C^{(1)}[b]$ consists of a bit '0', followed by $enc(b)$. Otherwise, it consists of a bit '1', followed by the binary encoding of b, that is, the block is stored without compression. The code length for any block b is upper bounded by $1+\frac{1}{2}\log n$ bits. Then we construct a table $D^{(1)}$ for decoding a block. The table has $2^{1+\frac{1}{2}\log n} = \mathcal{O}(\sqrt{n})$ entries and $D^{(1)}[x] = b$ for all binary patterns x of length $1 + \frac{1}{2}\log n$ such that a prefix of x is equal to $C^{(1)}[b]$. Note that this decode table is similar to r^{-1} defined in Section 2.2.

Next, for each block $T'[i]$ $(i = 1, \ldots, n')$, compute its length using $C^{(1)}[T'[i]]$, allocate space for storing it using the data structure of Theorem 6 with parameters $(1 + \ell\log\sigma, \frac{n}{\ell}) = (1 + \frac{1}{2}\log n, \frac{2n\log\sigma}{\log n})$, and $w = \log n$. From Lemma 2 and Theorem 6, if follows that the size of the initial data structure is $nH_k(T) + \mathcal{O}\left(\frac{n\log\sigma}{\log n}(k\log\sigma + \log\log n)\right)$ bits. Finally, for later use, copy the contents of $F^{(1)}$ to $F^{(2)}$, and initialize $R^{(0)}$ by 0. By sorting the blocks by a radix sort, the preprocessing time becomes $\mathcal{O}(n\log\sigma/\log n)$.

5.2 Algorithm for `Access`

The algorithm for $\texttt{access}(T, i)$ is: Given the index i, compute the block number $x = \lfloor (i-1)/\ell \rfloor + 1$ and the super-block number y containing $T[i]$. Obtain the pointer to the block and the length of the code by $\texttt{address}(x)$. Decode the block using the decode table $D^{(j-2)}$ if $R^{(j-1)}[x] = 0$, or $D^{(j-1)}$ if $R^{(j-1)}[x] = 1$. This takes constant time.

5.3 Algorithm for `Replace`

We first explain a naive, inefficient algorithm. If $b = T'[i]$ is replaced with b', we change the frequency table $F^{(1)}$ so that $F^{(1)}[b]$ is decremented by one and

$F^{(1)}[b']$ is incremented by one. Then new code table $C^{(1)}$ and decode table $D^{(1)}$ are computed from updated $F^{(1)}$, and all blocks $T'[j]$ $(j = 1, \ldots, n')$ are re-encoded by using the new code table. Obviously, this algorithm is too slow.

To get a faster algorithm, we can delay updating code tables for the blocks and re-writing the blocks using new code tables because of Theorem 4. Because the amount of change in entropy is small after a small change in the string, we can show that the redundancy of using code tables defined according to an old string can be negligible. For each single character change in T, we re-encode a super-block (ℓ/ϵ characters in T). After $\epsilon n'$ changes, the whole string will be re-encoded. To specify which super-block to be re-encoded, we use an integer array $G^{(j-1)}[1..n'']$. It stores a permutation of $(1, \ldots, n'')$ and indicates that at the x-th replace operation in phase j we rewrite the $G^{(j-1)}[x]$-th super-block. The bit $R^{(j-1)}[x]$ indicates if the super-block has been already rewritten or not. The array $G^{(j-1)}$ is defined by sorting super-blocks in increasing order of lengths of codes for encoding super-blocks.

We implement `replace`(T, i, S) as follows. In the x-th update in phase j,

1. If $R^{(j-1)}[G^{(j-1)}[x]] = 0$, i.e., if the $G^{(j-1)}[x]$-th super-block is encoded with $C^{(j-2)}$, decode it and re-encode it with $C^{(j-1)}$, and set $R^{(j-1)}[G^{(j-1)}[x]] = 1$.
2. Let y be the super-block number containing $T[i]$, that is, $y = \lfloor \epsilon(i-1)/\ell \rfloor$.
3. Decode the y-th super-block, which is encoded with $C^{(j-2)}$ or $C^{(j-1)}$ depending on $R^{(j-1)}[y]$. Let S' denote the block containing $T[i]$. Make a new block S from S' by applying the `replace` operation.
4. Decrement the frequency $F^{(j+1)}[S']$ and increment the frequency $F^{(j+1)}[S]$.
5. Compute the code for encoding S using $C^{(j-1)}$ if the y-th super-block is already re-encoded ($R^{(j-1)}[y] = 1$), or $C^{(j-2)}$ otherwise ($R^{(j-1)}[y] = 0$).
6. Compute the lengths of the blocks in y-th super-block and apply `realloc` for those blocks.
7. Rewrite the blocks in the y-th super-block.
8. Construct a part of tables $C^{(j)}$, $D^{(j)}$, $G^{(j)}$, and $R^{(j)}$ (see below).

To prove that the algorithm above maintains the invariant, we need only to prove that the tables $C^{(j-1)}$, $F^{(j)}$, and $G^{(j-1)}$ are ready at the beginning of phase j. In phase j, we create $C^{(j)}$ based on $F^{(j)}$. This is done by just radix-sorting the frequencies of blocks, and therefore the total time complexity is $\mathcal{O}(\sigma^l) = \mathcal{O}(\sqrt{n})$. Because phase j consists of n'' `replace` operations, the work for creating $C^{(j)}$ can be distributed in the phase. We represent the array $G^{(j-1)}$ implicitly by $(1/\epsilon)(1 + \frac{1}{2}\log n)$ doubly-linked lists L_d; L_d stores super-blocks of length d. By retrieving the lists in decreasing order of d we can enumerate the elements of $G^{(j)}$. If all the elements of a list have been retrieved, we move to the next non-empty list. This can be done in $\mathcal{O}(1/\epsilon)$ time if we use a bit-vector of $(1/\epsilon)(1 + \frac{1}{2}\log n)$ bits indicating which lists are non-empty. We copy $F^{(j)}$ to $F^{(j+1)}$ in constant time by changing pointers to $F^{(j)}$ and $F^{(j+1)}$. For each `replace` in phase j, we re-encode a super-block, which consists of $1/\epsilon$ blocks. This takes $\mathcal{O}(1/\epsilon)$ time. Therefore the time complexity for `replace` is $\mathcal{O}(1/\epsilon)$ time.

Note that during phase j, only the tables $F^{(j)}$, $F^{(j+1)}$, $C^{(j-2)}$, $C^{(j-1)}$, $C^{(j)}$, $D^{(j-2)}$, $D^{(j-1)}$, $D^{(j)}$, $G^{(j-1)}$, $G^{(j)}$, $R^{(j-1)}$, and $R^{(j)}$ are stored. The other tables are discarded.

5.4 Space Analysis

Let $s(T)$ denote the size of the encoding of T by our dynamic data structure. At the beginning of phase j, the string $T^{(j)}$ is encoded with code table $C^{(j-2)}$, which is based on the string $T^{(j-2)}$. Let $L^{(j)} = nH_k(T^{(j)})$ and $L^{(j-2)} = nH_k(T^{(j-2)})$.

After the preprocessing, $s(T^{(1)}) \leq L^{(1)} + \mathcal{O}\left(\frac{n \log \sigma}{\log n}(k \log \sigma + \log\log n)\right)$. If we do not re-encode the string, for each `replace` operation we write at most $1 + \frac{1}{2}\log n$ bits. Therefore $s(T^{(j)}) \leq s(T^{(j-2)}) + \mathcal{O}(n'' \log n)$ holds. Because $T^{(j)}$ is made by $2(n'' + \sqrt{n})$ character changes to $T^{(j-2)}$, from Theorem 4, we have $|L^{(j)} - L^{(j-2)}| = \mathcal{O}(n''(k+1)(\log n + \log \sigma))$. Therefore we obtain $s(T^{(j)}) \leq L^{(j)} + \mathcal{O}(\epsilon(k+1)n \log \sigma)$. The space for storing the tables $F^{(j)}$, $C^{(j)}$, $D^{(j)}$, $G^{(j)}$, $H^{(j)}$, and $R^{(j)}$ is $\mathcal{O}(\sqrt{n}\log n)$, $\mathcal{O}(\sqrt{n}\log n)$, $\mathcal{O}(\sqrt{n}\log n)$, $\mathcal{O}(n'' \log n) = \mathcal{O}(\epsilon n \log \sigma)$, $\mathcal{O}(n'' \log n)$, $\mathcal{O}(n'')$ bits, respectively.

Next we analyze the space redundancy caused by the re-encoding of super-blocks. We re-encode the super-blocks with the new code table in increasing order of their lengths, that is, the shortest one is re-encoded first. This guarantees that at any time, the space does not exceed $\max\{s(T^{(j)}), s(T^{(j-2)})\}$. This completes the proof of Theorem 1.

6 Concluding Remarks

We have presented a data structure called Compressed Random Access Memory (CRAM), which compresses a string T of length n into its k-th order empirical entropy in such a way that any consecutive $\log_\sigma n$ bits can be obtained in constant time (the `access` operation), and replacing a character (the `replace` operation) takes $\mathcal{O}(\min\{\log_\sigma n, (k+1)\log n/\log\log n\})$ time. The time for `replace` can be reduced to constant ($\mathcal{O}(1/\epsilon)$) time by allowing an additional $\mathcal{O}(\epsilon(k+1)n \log \sigma)$ bits redundancy. The extended CRAM data structure also supports the `insert` and `delete` operations, at the cost of increasing the time for `access` to $\mathcal{O}(\log n/\log\log n)$ time, which is optimal under this stronger requirement, and the time for each update operation also becomes $\mathcal{O}(\log n/\log\log n)$.

Preliminary experimental results indicate that our CRAM data structure supports faster reads/writes of short segments (from 16 to 256 bytes) than when using `gzip`. These experimental results will be reported in another paper.

An open problem is how to improve the running time of `replace` for the CRAM data structure to $\mathcal{O}(1)$ without using the $\mathcal{O}(\epsilon(k+1)n \log \sigma)$ extra bits.

Acknowledgments. JJ was funded by The Hakubi Project at Kyoto University. KS was supported in part by Funding Program for World-Leading Innovative R&D on Science and Technology (FIRST Program). WKS was supported in part by the MOE's AcRF Tier 2 funding R-252-000-444-112.

References

1. Cover, T.M., Thomas, J.A.: Elements of Information Theory. Wiley Interscience (1991)
2. Ferragina, P., Manzini, G.: Indexing compressed text. Journal of the ACM 52(4), 552–581 (2005)
3. Ferragina, P., Manzini, G., Mäkinen, V., Navarro, G.: Compressed representations of sequences and full-text indexes. ACM Transactions on Algorithms 3(2), article No. 20 (2007)
4. Ferragina, P., Venturini, R.: A simple storage scheme for strings achieving entropy bounds. Theoretical Computer Science 372(1), 115–121 (2007)
5. Fredman, M.L., Saks, M.E.: The cell probe complexity of dynamic data structures. In: Proceedings of ACM STOC, pp. 345–354 (1989)
6. González, R., Navarro, G.: Statistical Encoding of Succinct Data Structures. In: Lewenstein, M., Valiente, G. (eds.) CPM 2006. LNCS, vol. 4009, pp. 294–305. Springer, Heidelberg (2006)
7. González, R., Navarro, G.: Rank/select on dynamic compressed sequences and applications. Theoretical Computer Science 410(43), 4414–4422 (2009)
8. Grossi, R., Gupta, A., Vitter, J.S.: High-order entropy-compressed text indexes. In: Proceedings of ACM-SIAM SODA, pp. 841–850 (2003)
9. Hagerup, T.: Sorting and searching on the word RAM. In: Proceedings of Symposium on Theory Aspects of Computer Science (STACS 1998), pp. 366–398 (1998)
10. Hagerup, T., Raman, R.: An Efficient Quasidictionary. In: Penttonen, M., Schmidt, E.M. (eds.) SWAT 2002. LNCS, vol. 2368, pp. 1–18. Springer, Heidelberg (2002)
11. He, M., Munro, J.I.: Succinct Representations of Dynamic Strings. In: Chavez, E., Lonardi, S. (eds.) SPIRE 2010. LNCS, vol. 6393, pp. 334–346. Springer, Heidelberg (2010)
12. Kosaraju, S.R., Manzini, G.: Compression of low entropy strings with Lempel-Ziv algorithms. SIAM Journal on Computing 29(3), 893–911 (1999)
13. Mäkinen, V., Navarro, G.: Dynamic entropy-compressed sequences and full-text indexes. ACM Transactions on Algorithms 4(3), article No. 32 (2008)
14. Manzini, G.: An analysis of the Burrows-Wheeler transform. Journal of the ACM 48(3), 407–430 (2001)
15. Navarro, G., Sadakane, K.: Fully-functional static and dynamic succinct trees. Submitted for Journal Publication (2010), `http://arxiv.org/abs/0905.0768`; A preliminary version appeared in Proc. ACM-SIAM SODA, pp. 134–149 (2010)
16. Raman, R., Raman, V., Rao, S.S.: Succinct Dynamic Data Structures. In: Dehne, F., Sack, J.-R., Tamassia, R. (eds.) WADS 2001. LNCS, vol. 2125, pp. 426–437. Springer, Heidelberg (2001)
17. Sadakane, K., Grossi, R.: Squeezing succinct data structures into entropy bounds. In: Proceedings of ACM-SIAM SODA, pp. 1230–1239 (2006)
18. Simpson, J.T., Wong, K., Jackman, S.D., Schein, J.E., Jones, S.J.M., Birol, İ.: ABySS: A parallel assembler for short read sequence data. Genome Research 19(6), 1117–1123 (2009), `http://dx.doi.org/10.1101/gr.089532.108`