# Space Efficient Wavelet Tree Construction*

Francisco Claude[1], Patrick K. Nicholson[1], and Diego Seco[2]

[1] David R. Cheriton School of Computer Science, University of Waterloo, Canada
[2] University of A Coruña, A Coruña, Spain
{fclaude,p3nichol}@cs.uwaterloo.ca, dseco@udc.es

**Abstract.** Wavelet trees are one of the main building blocks in many space efficient data structures. In this paper, we present new algorithms for constructing wavelet trees, based on in-place sorting, that use virtually no extra space. Furthermore, we implement and confirm that these algorithms are practical by comparing them to a known construction algorithm. This represents a step forward for practical space-efficient data structures, by allowing their construction on more massive data sets.

## 1 Introduction

Succinct data structures supporting rank, select and access operations represent the core of most space efficient data structures [24, 21, 4]. For example, structures supporting rank and select over binary sequences allow for space efficient representation of trees [6, 7, 23, 12, 2].

One of the most elegant generalizations of such structures for binary rank, select and access is the *wavelet tree* [17]. Wavelet trees have not only proven to be a crisp theoretical solution, but also perform well in practice [13, 8]. As such, they are used by many practical compressed text indexes, such as the SSA [24], LZ77-Index [20] and SLP-Index [9, 10]. Another fact that makes wavelet trees interesting as a structure is that they support richer queries than initially expected. For example, they have been used for representing binary relations [4, 11], discrete set of points [5] and permutations [3], among others. In all of these domains, wavelet trees support a rich set of operations efficiently.

There has been a great deal of study on the performance of different variants of wavelet trees [8], considering the shape and internal representation of the bitmaps [18, 7, 25]. However, not much effort has been put into space efficient construction algorithms for wavelet trees.

We present several new algorithms for constructing wavelet trees, using virtually no extra space. This is a significant improvement over the naïve construction algorithm, as well as a previously known technique [22]. We discuss our results in detail at the end of this section, but first we elaborate on the problem.

Given a sequence $A$ of length $n$, denoted $A[0..n-1]$, drawn from an alphabet $\Sigma$ of size $\sigma$. A wavelet tree is a data structure that supports the following operations: (1) $\texttt{access}(A, i)$: retrieve the symbol at position $i$ in $A$. (2) $\texttt{rank}_a(A, i)$:

* This work was supported in part by the David R. Cheriton scholarships program (first author) and an NSERC of Canada PGS-D Scholarship (second author).

count how many times the symbol $a$ appears in $A[0..i]$. (3) $\texttt{select}_a(A, j)$: retrieve the position of the $j$-th occurrence of $a$ in $A$.

Supporting these operations in constant time, using $nH_0(A) + o(n)$ bits of space[3], was first solved for binary alphabets (i.e., $\Sigma = \{0, 1\}$) by Raman et al. [25]. Wavelet trees extend this result to arbitrary alphabets in the following way. Every node in a wavelet tree has two children, and we identify them as left and right. Each node represents a range $R \subseteq [1, \sigma]$ of the alphabet $\Sigma$, and its left child represents a subset $R_\ell \subset R$ and the right child a subset $R_r = R \setminus R_\ell$. Every node virtually represents a subsequence $A'$ of $A$ composed of elements whose value is in $R$. This subsequence is stored as a bitmap $B$ of length $|A'|$, and, for each position $i$ in the bitmap, a 0 bit means that position $i$ belongs to $R_\ell$ and a 1 bit means that it belongs to $R_r$.

One can access a position in $A$ by following the path from the root to a leaf guided by the bit representing the position at each level. When moving to the left child the position $i$ is mapped to $\texttt{rank}_0(B, i)$ and when moving to the right child the position is mapped to $\texttt{rank}_1(B, i)$. By similar observations, it is an easy exercise to show that a wavelet tree can support rank, select and access operations in $O(\lg \sigma)$ time[4], assuming that rank and select are supported in constant time on the bitmaps in each level. Furthermore, the wavelet tree uses $n \lg \sigma + o(n \lg \sigma)$ bits because the bitmaps representing the nodes use $n \lg \sigma$ bits in total and the little-oh term covers the cost of the rank and select structures.

Wavelet trees were later extended to *generalized wavelet trees*, where the fan out of each node is increased from 2 to $O(\text{polylog}(n))$, increasing the speed of the three operations to $O(\lg \sigma / \lg \lg n)$ time [14]. In favour of clarity, we focus on binary wavelet trees, though our results also extend to the generalized version.

A simple construction algorithm works in the following way. First, we build the root of the wavelet tree. To do this, we partition the alphabet $\Sigma$ into $\Sigma_\ell$ and $\Sigma_r$ according to some balancing rule, and then create a bitmap of length $n$ that stores a 0 in position $i$ if $A[i] \in \Sigma_\ell$ and a 1 in position $i$ otherwise. Then, we generate a copy of the subsequence of elements whose bit at position $i$ is a 0 and recurse on this subsequence to build the left subtree. Finally, we recurse on the 1 bits to construct the right subtree.

It is easy to see that this method is inefficient and requires $O(n \lg^2 \sigma)$ bits, since we copy the array $A$ at each of the $\lg \sigma$ levels. This can be improved by realizing that, at each level in the recursion, we can reuse the space from the previous level and avoid recopying the sequence. The only exception to this is the root, since our application might require us to leave $A$ unmodified after the construction process. Thus, we can construct the wavelet tree in $O(n \lg \sigma)$ extra bits, since we must copy $A$. An example of such an application is a library for succinct data structures, such as LIBCDS[5], where the user might want to further process the sequence used to build the wavelet tree.

[3] $H_0(A)$ denotes the 0th-order empirical entropy of the string $A$.
[4] Throughout this paper we denote $\lceil \log_2 n \rceil$ as $\lg n$.
[5] `http://libcds.recoded.cl`

*Our Results:* In this paper we present several new algorithms for constructing wavelet trees space efficiently. All of our results hold in the word-RAM model with word size $\Omega(\lg n)$. Throughout this section, we denote the input array as $A$ and the wavelet tree as $T$. All of our results are for uncompressed wavelet trees, that is, the case where the bitmaps of the wavelet tree occupy $n \lg \sigma$ bits. Thus $T$ occupies $n \lg \sigma + S(n \lg \sigma)$ bits, where $S(m)$ denotes the extra space required by auxiliary rank and select structures on a bitmap of length $m$. Extending our results to compressed wavelet trees, e.g., Huffman shaped wavelet trees, is left as an open problem. Before discussing the results, we need the following definitions.

We refer to a construction algorithm as *non-destructive* if $A$ is unmodified after $T$ has been constructed. If $A$ is modified, rendering it unusable, we say the construction algorithm is *destructive*.

Since there are many choices of rank and select data structures for bitmaps, we will attempt to state our results as generally as possible. We do, however, assume that the auxiliary structures used for bitmaps support rank and select in constant time. Let $C(m)$ denote the construction time for auxiliary rank and select structures on a bitmap of length $m$, and $E(m)$ denote the extra bits required to construct these structures. We note that in many existing algorithms, $E(n) = O(\lg n)$ extra bits. Given a bitmap $B[0..n-1]$, suppose we construct auxiliary rank and select structures in time $C(p)$ on a prefix of $B$, $B[0..p]$ where $0 \leq p < n$, such that we can answer rank and select queries on $B[0..p]$. If we can extend our rank and select structures to support queries on $B[0..q]$ for $p < q < n$ in $C(q-p)$ time, we say these rank and select structures can be constructed *incrementally*.

In Section 2 we present a non-destructive algorithm for constructing the wavelet tree $T$ in $O(n \lg \sigma + C(n \lg \sigma))$ time, using $O(\lg n \lg \sigma) + E(n \lg \sigma)$ bits beyond the space occupied by $A$ and $T$. This section serves as a warm-up, introducing many of the concepts used later in the paper.

In Section 3 we present a destructive algorithm for constructing the wavelet tree $T$ in $O(n \lg n \lg^2 \sigma + C(n \lg \sigma))$ time using $O(\lg n \lg \sigma) + E(n \lg \sigma)$ bits beyond the space required for $T$. In other words, this algorithm replaces $A$ with the bitmaps for each node in $T$. We also present a more practical algorithm that runs in $O(n \lg \sigma + C(n \lg \sigma))$ time and uses $n + O(\lg n \lg \sigma) + E(n \lg \sigma)$ bits beyond the space required for $T$. In all of the previous results, we show how to replace the $O(\lg n \lg \sigma)$ bit term in the space bound with $O(\lg n)$ bits, if the rank and select structures for $T$ can be constructed incrementally.

Finally, in Section 4 we provide experimental results for the algorithms discussed in Section 3. These results show that our algorithms are practical.

## 2 Encoding Scheme

In this section, we show how to reorder the elements of an array $A[0..n-1]$ according to the bitmap $B[0..n-1]$ representing the root of the wavelet tree. This allows us to construct $T$ without copying the subsequences of $A$ into separate arrays at each level. We refer to this process as *partitioning*, and it is the

bottleneck in any space efficient wavelet tree construction algorithm. After describing partitioning, we show how to reverse a partitioning step. That is, given the subsequence of elements from the left subtree, $A_\ell$, the elements from the right subtree, $A_r$, and the bitmap representing the root of the wavelet tree, $B$, we show how to rebuild $A$. We refer to this process as *merging*. In other words, denote string concatenation by $\cdot$ and let $A' = A_\ell \cdot A_r$. Partitioning is the process of constructing $A'$ from $A$ and $B$, and merging is the process of reconstructing $A$ from $A'$ and $B$.

### 2.1 Partitioning

We describe the method implemented in LIBCDS that shows a simple way of partitioning the array $A$ given that we can support constant time rank and select queries on the bitmap $B$ [22]. Let $n_\ell$ denote $\texttt{rank}_0(B, n-1)$. It is easy to see that the bitmap $B$ defines a permutation $\pi$ on the elements of array $A$ as follows:

$$\pi(i) = \begin{cases} \texttt{rank}_0(B,i) - 1 & \text{if } B[i] = 0, \\ \texttt{rank}_1(B,i) - 1 + n_l & \text{otherwise.} \end{cases} \tag{1}$$

One way of partitioning $A$ is to create an *auxiliary bitmap* $Aux$ of length $n$, where initially all of the bits are set to 0. We then do a scan of the array $A$ and, for each position $p = 0..n-1$, if $Aux[p] = 0$, we move the element at position $p$ to its corresponding place, position $q = \pi(p)$, and set $Aux[p] = 1$. We repeat this process with the element that was at position $q$ in $A$ until returning to a position $q'$ where $Aux[q'] = 1$ [22]. It is not hard to see that $q' = p$: following standard terminology [15], we call position $p$ a *cycle leader* of $\pi$, since it has the smallest index in the *cycle* of element swaps. Furthermore, we say that the elements in a cycle are *rotated* according to $\pi$. Thus, the auxiliary array is used to identify cycle leaders.

The procedure just described requires $n + O(\lg n)$ extra bits to identify cycle leaders and perform the partitioning in $O(n)$ time, assuming we can support rank operations on $B$ in constant time. The $O(\lg n)$ term comes from the constant number of pointers needed to scan the array and rotate the elements.

Let $\pi^k(i) = \pi(\pi^{k-1}(i))$, for $k > 1$ and $\pi^1(i) = \pi(i)$. If $\pi(i) = i$ then we say position $i$ is a *self-cycle*. Let $A'$ denote the array $A$ after it has been partitioned. In order to improve the space requirements, we prove the following property of the permutation $\pi$:

**Lemma 1.** *If $\pi(i) \neq i$ (i.e., position $i$ is not a self-cycle), then $\pi^k(i) \geq n_\ell$ for some $k \geq 1$. Similarly, if $j \geq n_\ell$ and $\pi(j) \neq j$, then $\pi^k(j) < n_\ell$ for some $k \geq 1$.*

*Proof.* The relative ordering of the elements that are symbols in $\Sigma_\ell$ in $A'[0..n_\ell - 1]$ is the same as the relative ordering of these elements in $A[0..n-1]$. Thus, if a rotation begins at an element in $\Sigma_\ell$ in position $i$, it will be moved to a position $0 \leq j < i$. Since the rotation will end at position $i$, at some point we must

encounter an element in $\Sigma_r$. By the definition of $\pi$, this element will be moved to a position $j' \geq n_\ell$. The second part of the lemma follows by symmetry. □

Based on the previous lemma, we make the following observation:

**Observation 1** *Rotating only the cycle leaders in positions where $B[i] = 0$ is sufficient to complete the partitioning of $A$ into $A'$, since every cycle that is not a self-cycle involves elements from both $\Sigma_\ell$ and $\Sigma_r$.*

We now show how to perform the partitioning *without* access to $Aux$. We continue assuming that $|\Sigma_\ell| \leq |\Sigma_r|$. If this condition does not hold, we can apply Observation 1 symmetrically, considering only positions where $B[i] = 1$.

The idea that allows us to discard $Aux$ is to encode it inside $A$ as we perform the partitioning. We do this by defining an invertible function $f : \Sigma_\ell \to \Sigma_r$. This function exists since $|\Sigma_\ell| \leq |\Sigma_r|$. We run exactly the same partitioning algorithm described in the beginning of this section, except that, during a rotation, every time we move a symbol $s \in \Sigma_\ell$ at position $i$ to position $\pi(i)$, we write $f(s)$ in position $\pi(i)$ instead. Since we do not have to rotate cycle leaders from $\Sigma_r$ by Observation 1, this encoding step is functionally equivalent to having access to $Aux$. Every time we encounter an element in $\Sigma_\ell$, that element would have had a 0 in its corresponding position in $Aux$, and we can ignore elements that either would have a 1 in $Aux$ or were originally in $\Sigma_r$.

After finishing this process, we need one extra pass to decode the values that are supposed to be in $\Sigma_\ell$. We do this by traversing $A'$ and replacing position $i$ by $f^{-1}(A[i])$ if $i < n_\ell$. Recall our assumption that $T$ is an uncompressed wavelet tree, i.e., each partitioning step moves the elements with a 0 as their most significant bit to the left child and each element with a 1 as their most significant bit to the right child. Since the ranges of $\Sigma$ spanned by $\Sigma_\ell$ and $\Sigma_r$ are contiguous and adjacent, computing $f()$ and $f^{-1}()$ boils down to a simple addition and subtraction, respectively.

**Lemma 2.** *Given an array $A$ over an alphabet $\Sigma$ and support for constant time rank and select operations on the bitmap $B$, we can partition $A$ in-place to generate $A'$ in $O(n)$ time using $O(\lg n)$ extra bits of space.*

### 2.2 Merging

The merging process is just the partitioning process in reverse. We describe this problem in a similar way, using the inverse permutation $\pi^{-1}$:

$$\pi^{-1}(i) = \begin{cases} \texttt{select}_0(B, i+1) & \text{if } i < n_l, \\ \texttt{select}_1(B, i+1-n_l) & \text{otherwise.} \end{cases} \tag{2}$$

It is easy to see that Lemma 1 and Observation 1 also hold in the merging case. The only difference is that now elements in $\Sigma_\ell$ are rotated to the right and elements from $\Sigma_r$ are rotated to the left. Thus, there is at least one element in $\Sigma_\ell$ and one in $\Sigma_r$ for each cycle of length greater than 1. These observations allow us to apply the same method as in Lemma 2, obtaining the following lemma.

**Lemma 3.** *Given the array $A'$ and support for constant time rank and select operations on the bitmap $B$, we can reconstruct $A$ in-place in $O(n)$ time using $O(\lg n)$ extra bits of space.*

Using a stack of size $O(\lg \sigma)$ pointers to keep track of the node in $T$ that we are currently processing, we can recursively apply *partitioning* to $A$, to construct $T$. After $T$ is constructed, we can reverse the process by *merging* to recover $A$. We can compute $B$ in linear time for each node.

**Theorem 2.** *There exists a non-destructive algorithm for constructing a wavelet tree $T$ that uses $O(n \lg \sigma + C(n \lg \sigma))$ time, and $O(\lg n \lg \sigma) + E(n \lg \sigma)$ bits beyond the space required for $T$ and the input array $A$.*

### 2.3 Extension to Generalized Wavelet Trees

The encoding scheme for generalized wavelet trees works in a similar way to that of the binary case. We begin by stating the generalized partitioning problem: given an array $A[0, n-1]$ with values in $\Sigma = [0, \sigma - 1]$ and a sequence $S$ with values in $D = [0, k-1]$ that partitions $\Sigma$ into $k$ disjoint sets $\Sigma_0, \ldots, \Sigma_{k-1}$, we want to generate an array $A'$ where elements of $A$ are stable-sorted by their corresponding value in $D$. We can describe the re-ordering in $A$ as a permutation: $\pi_k(i) = \texttt{rank}_j(S, i) + (\sum_{v<j} \texttt{rank}_v(S, n-1)) - 1,$ where $j = S[i]$.

**Lemma 4.** *The permutation $\pi_k$ is strictly increasing for positions containing the same value in $D$.*

*Proof.* We can write $\pi_k(i)$ as $\texttt{rank}_j(S, i) + g(j)$, where $j = S[i]$ and $g(j) = \sum_{v<j} \texttt{rank}_v(S, n-1)$, and $\texttt{rank}_j(S, i)$ is strictly increasing in $i$. □

**Lemma 5.** *Any cycle $\mathcal{C}$ in $\pi_k$ such that $|\mathcal{C}| > 1$, contains at least two positions $i, j$ such that $S[i] \neq S[j]$.*

*Proof.* By contradiction, assume $|\mathcal{C}| > 1$ and all positions $p_i \in \mathcal{C}$ satisfy $S[p_i] = s$ for a fixed $s$. Let $p = \min \mathcal{C}$. Since all positions in $\mathcal{C}$ point to elements in $S$ whose value is the same, then $\pi_k$ is strictly increasing in $\mathcal{C}$, thus, there is no $p_j$ such that $\pi_k(p_j) = p$, therefore, $\mathcal{C}$ is not a cycle. □

Now we can present the encoding method. Let $m \in [0, k-1]$ be the index such that $|\Sigma_m| \geq |\Sigma_j|$. We then generate $f_j : \Sigma_j \to \Sigma_m$ such that ${f_j}^{-1}$ exists. Then, the algorithm for partitioning works in the following way: for each cycle leader, we start rotating the elements if the position is not in $\Sigma_m$. Every time we rotate an element $e$ in $\Sigma_j$, we replace it with $f_j(e)$.

Once we finish the process, we do a final pass through $A'$ fixing the values at position $p$ using ${f_j}^{-1}$, where $j$ is determined by the position, i.e., $j = \min\{r | \sum_{v<r} \texttt{rank}_v(S, n-1) > p\}$.

There is one detail remaining, and this is how to compute $g(j) = \sum_{v<j} \texttt{rank}_v(S, n-1)$. We can do this by pre-computing all possible answers in linear time. This option requires $O(\sigma \lg n)$ bits of extra space. Another option

is to use compressed bitmaps to represent this in $\sigma \lg(n/\sigma) + o(n)$ bits, while supporting queries in constant time [25]. We now state the partitioning theorem for the generalized wavelet tree:

**Theorem 3.** *We can solve the partitioning problem for the generalized wavelet tree in $O(n\tau)$ time using $\min(\sigma \lg(n/\sigma) + o(n), O(\sigma \lg n))$ bits of extra space, where $\tau$ represents the maximum between the time to answer rank and access in a sequence over an alphabet of size $k$.*

The generalized merging process is similar, and the permutation $\pi_k^{-1}$ is defined as follows: $\pi_k^{-1}(i) = \texttt{select}_j(S, i + 1 - g(j))$, where $j = S[i]$ and $g(j) = \sum_{v<j} \texttt{rank}_v(S, n-1)$.

Lemmas 4 and 5 also apply to $\pi_k^{-1}$, since select is strictly increasing for positions that contain the same element. This allows us to apply the same encoding technique as before; the only difference is the transformation at the end. Instead of examining the range we are in, we examine the character in $S$ associated with our position.

**Theorem 4.** *We can solve the merging problem for the generalized wavelet tree in $O(n\tau)$ time using $\min(\sigma \lg(n/\sigma) + o(n), O(\sigma \lg n))$ bits of extra space, where $\tau$ represents the maximum between the time to answer access and select in a sequence over an alphabet of size $k$.*

Regarding the rank, select and access times in Theorems 3 and 4, there are many alternatives [17, 14, 16], in particular, it is possible to adapt the solution by [14] to compute the function $g()$ in constant time, achieving the following Corollary.

**Corollary 1.** *We can solve the partitioning and merging problems for the generalized wavelet tree in $O(n)$ time using $\min(n \lg(n/\sigma) + o(n), O(\sigma \lg n))$ extra bits of space, if the branching factor of the generalized wavelet tree is $k = O(\text{polylog}(n))$.*

## 3 Construction by Permuting Bits

In this section we show how to destructively permute the bits of an input array $A$, converting them into the bit vectors of a *binary* wavelet tree $T$.

Let $B_v$ represent the bitmap stored at the root $v$ of $T$, and $v_l$ and $v_r$ represent the left and right children of $v$. Define $\mathbb{B}(v) = B_v \cdot \mathbb{B}(v_l) \cdot \mathbb{B}(v_r)$. Thus, $\mathbb{B}(v)$ is the concatenation of the bitmaps stored at the nodes of $T$, in depth first order from the root, $v$. We describe an algorithm for computing $\mathbb{B}(v)$ that works by permuting the bits of $A$ in place. For our purposes in this section, we consider $A$ to be a bitmap of length $n \lg \sigma$. Let $\phi$ be the permutation that maps $A$ to $\mathbb{B}(v)$; we abuse notation and denote this as $\phi(A) = \mathbb{B}(v)$. We show that $\phi$ is the composition of $2\sigma$ permutations: two permutations, $\chi$ and $\psi$, corresponding to each node in $T$.

### 3.1 Overview of Permutations

In the next section we describe the two permutations $\chi$ and $\psi$ that correspond to the root $v$ of $T$. Before specifying the technical details of these permutations, we first briefly outline *what* they do to the array $A$.

The first permutation $\chi$ shifts the most significant bit of each character in $A$ to a prefix of the bitmap, preserving relative order. More precisely, $\chi(A)$ consists of an $n$ bit prefix $B_v[0..n-1]$, representing the most significant bits of each element in $A$ (which *is* the bitmap stored at the root of $T$), followed by $n$ *truncated characters* of length $\lg \sigma - 1$; the $i$-th truncated character is $A[i]$ without its most significant bit, for $0 \leq i < n$.

Let $A_t[0..n-1]$ represent the $n$ truncated characters. The second permutation, $\psi$, partitions $A_t[0..n-1]$ according to the bits $B_v[0..n-1]$. Thus, applying $\psi$ is equivalent to the partitioning step in a standard wavelet tree construction algorithm: if $B_v[i]$ is a 0, then $A_t[i]$ is partitioned to the left and, if $B_v[i]$ is a 1, then $A_t[i]$ is partitioned to the right.

After applying $\chi$ and $\psi$ to $A$, $\psi(\chi(A))$ consists of $B_v[0..n-1]$, which are the first $n$ bits of $\mathbb{B}(v)$, followed by the partitioned truncated characters, which can then be recursively converted into the remaining bits of $\mathbb{B}(v)$ in a depth first manner. In the sequel, we rely heavily on the following result of Fich et al.:

**Theorem 5 (Theorem 2.2 [15]).** *In the worst case, permuting an array of length $n$, given the permutation and its inverse, can be done in $O(n \lg n)$ time and $O(\lg n)$ additional bits of storage.*

The next two sections define the permutations $\chi$ and $\psi$, and their respective inverses.

### 3.2 Chopping the Most Significant Bits

Since $A$ is a bitmap of length $n \lg \sigma$, we refer to individual bits in $A$. Let $A = b_0, \ldots, b_{n \lg \sigma - 1}$, where $b_{j \lg \sigma}, ..., b_{(j+1) \lg \sigma - 1}$ are the bits in $A[j]$ for $0 \leq j < n$, in *decreasing order of significance*[6]. Using this notation, we can now describe $\chi(A, i)$, the $i$-th bit of $\chi(A)$ in terms of the bits in $A$, for $0 \leq i < n \lg \sigma$. If $\chi(A, i) = j$, then the $i$-th bit of $\chi(A)$ is the $j$-th bit of $A$, or $b_j$; in the equations we write $j$ instead of $b_j$ for readability.

$$\chi(A, i) = \begin{cases} \frac{i}{\lg \sigma} & \text{if } \lg \sigma \text{ divides } i, \\ i + n - \left\lfloor \frac{i}{\lg \sigma} \right\rfloor - 1 & \text{otherwise.} \end{cases}$$

Similarly, we can describe the permutation $\chi^{-1}(A, i)$ as follows:

$$\chi^{-1}(A, i) = \begin{cases} i \lg \sigma & \text{if } i < n, \\ i - n + \left\lfloor \frac{i-n}{\lg \sigma - 1} \right\rfloor + 1 & \text{otherwise.} \end{cases}$$

[6] If the characters are stored in increasing order of significance, then we can easily reverse their bits in $O(n \lg \sigma)$ time and $O(\lg n)$ extra bits.

*Running Time:* Using $\chi$ and $\chi^{-1}$ we can apply Theorem 5 to $A$. This allows us to compute $\chi(A)$ in place using $O(n \lg \sigma \lg(n \lg \sigma)) = O(n \lg \sigma \lg n)$ time.

### 3.3 Partitioning the Truncated Letters

We now describe how to compute $\psi(\chi(A))$ from $\chi(A)$. Note that $\chi(A) = B_v[0..n-1] \cdot A_t[0..n-1]$, and suppose we build a rank and select data structure over $B_v$. We discuss the space issues associated with this in the next section. The permutations $\psi$ and $\psi^{-1}$ make use of rank and select queries in order to determine how to partition $A_t[0..n-1]$. Not surprisingly, $\psi$ and $\psi^{-1}$ are *almost* identical to the permutations described in Equations 1 and 2, respectively. The only difference is that we need to account for the fact that the truncated characters $A_t$ begin at an offset of $n$ from the beginning of $A$ and consist of $\lg \sigma - 1$ bits. We omit the exact details since they are not difficult, but tedious.

*Running Time:* As before, using $\psi$ and $\psi^{-1}$ we can apply Theorem 5 to $\chi(A)$. Observe that we can easily swap $\lg \sigma - 1$ bit elements in constant time, assuming the word size is $\Omega(\lg n)$ and $n \geq \sigma$. This allows us to compute $\psi(\chi(A))$ in place using $O(n \lg n)$ time.

### 3.4 Overall Requirements

*Running Time:* We must apply $\chi$ and $\psi$ appropriately at each node in $T$ in order to compute $\phi(A)$. This means that our overall running time is $T(n, \lg \sigma) = T(n_l, \lg \sigma - 1) + T(n - n_l, \lg \sigma - 1) + O(n \lg n \lg \sigma)$, where $T(n, 1) = O(1)$. Since the height of the tree is bounded in terms of $\lg \sigma$ rather than $n$, $T(n, \lg \sigma) = O(n \lg n \lg^2 \sigma)$.

*Extra Space:* As discussed in Section 3.3, at each node $v$ in $T$ we construct auxiliary rank and select structures for the bitmap of length $m \leq n$, associated with $v$. Let $S(m)$ denote the number of bits required for the auxiliary structures. The auxiliary structures for $T$ require $S(n \lg \sigma)$ extra bits, since $\mathbb{B}(v)$ is a bitmap of length $n \lg \sigma$. Furthermore, we can release the memory used by the auxiliary structures for each $v \in T$ after we have applied the permutations to $v$. Thus, we can avoid using extra space for the auxiliary structures associated with $v$, with careful memory management.

In addition to the $O(1)$ extra pointers required by Theorem 5, we need a stack of size $O(\lg \sigma)$ pointers in order to remember our current location within $T$. However, we can get rid of the stack at the cost of some complexity. Suppose $w_1, ..., w_\sigma$ are the nodes of $T$ in depth-first order. Then by storing only $n$ and the bits representing the path to $w_i$, we can compute the offset and length of the bitmap representing $w_{i+1}$ in $O(\lg \sigma)$ time using the rank and select structures constructed for $w_1, ..., w_i$. Note that $w_1, ..., w_i$ represent a contiguous prefix of the bitmap $\mathbb{B}(v)$. Thus, if we can construct rank and select structures for $\mathbb{B}(v)$ incrementally, we can discard the stack.

*Trade Off:* If we have an extra $n$ bits available to us, then we can apply $\chi$ and $\psi$ to each node in $T$ in $O(n)$ time per level of $T$, using a strategy similar to Arroyuelo and Navarro [1]. To apply $\chi$, we copy the most significant bits in $A$ into the auxiliary bitmap, then shift the remaining bits to the end of $A$, and finally, overwrite the first $n$ bits of $A$ with those stored in the auxiliary bitmap. This requires $O(n)$ time if we make use of word-level parallelism during the shifting stage, or $O(n \lg \sigma)$ time if we shift the bits one at a time. To apply $\psi$ we use the auxiliary bitmap to store the cycle leaders, as described in Section 2. Thus, with the $n$ extra bits, we can compute $\phi(A)$ in $O(n \lg \sigma)$ time, or $O(n \lg^2 \sigma)$ time without word-level parallelism during the shifting stage.

**Theorem 6.** *Suppose we are given an array $A$ of $n$ symbols drawn from an alphabet of size $\sigma$. Let $C(m)$ denote the construction time for the auxiliary rank and select structures on a bitmap of length $m$, $E(m)$ denote the extra bits required during the construction of these structures, and $S(m)$ denote the total number of bits occupied by these structures. Furthermore, assume these structures support rank and select in constant time. We can permute the $n \lg \sigma$ bits of $A$, replacing $A$ with the bitmaps of a wavelet tree $T$ occupying $n \lg \sigma + S(n \lg \sigma)$ bits in:*

1. *$O(n \lg n \lg^2 \sigma + C(n \lg \sigma))$ time using $O(\lg \sigma \lg n) + E(n \lg \sigma)$ extra bits beyond the space occupied by $T$.*
2. *$O(n \lg \sigma + C(n \lg \sigma))$ time, and using $n + O(\lg n \lg \sigma) + E(n \lg \sigma)$ extra bits beyond the space occupied by $T$.*

*In both of the previous results, we can replace the $O(\lg n \lg \sigma)$ bit term in the space bound with $O(\lg n)$ bits, if we can construct the rank and select structures for $T$ incrementally.*

The same idea can be extended to the generalized setting, though we defer the details to a later version of this paper.

## 4 Experiments

As a proof of concept we implemented two of the algorithms described in Theorem 6. We refer to the algorithm that uses $O(\lg \sigma \lg n)$ extra bits and runs in $O(n \lg n \lg^2 \sigma)$ time as Permute, and the algorithm that uses $n + O(\lg \sigma \lg n)$ extra bits and runs in $O(n \lg^2 \sigma)$ time (not making use of word-level parallelism) as Permute2. For a base line comparison, we used the space efficient destructive construction algorithm found in Libcds, which is similar to the algorithm described in Section 2. The code for Permute is an adaptation of the code from [15, Figure 5], modified to use a simple heuristic speed-up called Fast-Break [19]. The code for Permute2 is even more straightforward since it uses an auxiliary bitmap.

The machine used for the experiments has an AMD Athlon$^{\text{TM}}$ 64 X2 Dual Core Processor 5600+, core speed 2900MHz, L1 Cache size 256KB and L2 Cache

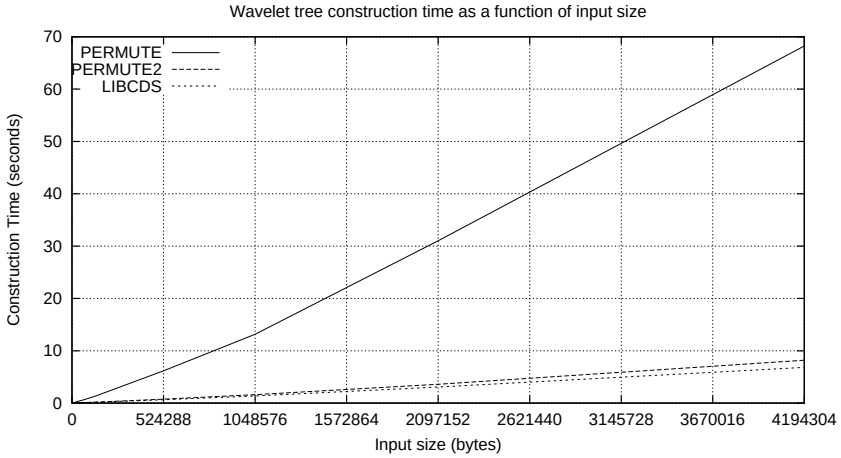


**Fig. 1.** A comparison of the space efficient construction algorithms PERMUTE and PERMUTE2 with the standard LIBCDS construction algorithm. In this experiment $\lg \sigma = 8$.

size 1024KB. It has 4GB of 800MHz main memory. The operating system installed is GNU/Linux (Ubuntu 10.04 LTS) and the code was compiled using GNU/g++ version 4.4.3, with optimization flags -O9.

We ran experiments for the case when $\lg \sigma = 8$, i.e., the array $A$ consists of 8-bit characters. For $n = 8, 16, 32, ..., 4194304$, we generated 10 strings of length $n$, where each string consists of $n$ 8-bit integers drawn *uniformly at random*. For each $n$ we kept track of the best, worst and average running time of the three construction algorithms. Since the best and worst times were very close to the average, we report only on the average time. A graph comparing the average construction times of the algorithms can be found in Figure 1. We obtained similar results when testing on prefixes of English and DNA text from `http://pizzachili.dcc.uchile.cl`.

As expected, PERMUTE is slower than LIBCDS for all tested values of $n$, by a factor that increases with $n$. For all values tested this factor was less than 11. Although this is a significant slow down, this experiment demonstrates that the space efficient algorithm we describe is implementable and that its performance is not impractical.

On the other hand, PERMUTE2 ran only slightly slower than LIBCDS, which has complexity $O(n \lg \sigma)$. This suggests that PERMUTE2 is highly competitive as a construction algorithm, due to low constant factors. Furthermore, since it only uses $n + O(\lg n \lg \sigma)$ extra bits on top of the space required for the wavelet tree, it provides a nice compromise between the slower PERMUTE algorithm and the construction algorithm currently used by LIBCDS.

## References


1. Arroyuelo, D., Navarro, G.: Space-efficient construction of lempel-ziv compressed text indexes. Information and Computation 209(7), 1070–1102 (2011)
2. Arroyuelo, D., Cánovas, R., Navarro, G., Sadakane, K.: Succinct trees in practice. In: Proc. ALENEX. pp. 84–97 (2010)
3. Barbay, J., Navarro, G.: Compressed representations of permutations, and applications. In: Proc. STACS. pp. 111–122 (2009)
4. Barbay, J., Claude, F., Navarro, G.: Compact rich-functional binary relation representations. In: Proc. LATIN. pp. 170–183. LNCS (2010)
5. Bose, P., He, M., Maheshwari, A., Morin, P.: Succinct orthogonal range search structures on a grid with applications to text indexing. In: Proc. WADS. vol. 5664, pp. 98–109 (2009)
6. Clark, D.: Compact Pat Trees. Ph.D. thesis, University of Waterloo (1996)
7. Clark, D.R., Munro, J.I.: Efficient suffix trees on secondary storage. In: Proc. SODA. pp. 383–391 (1996)
8. Claude, F., Navarro, G.: Practical rank/select queries over arbitrary sequences. In: Proc. SPIRE. pp. 176–187. LNCS 5280 (2008)
9. Claude, F., Navarro, G.: Self-indexed text compression using straight-line programs. In: Proc. MFCS. pp. 235–246. LNCS 5734 (2009)
10. Claude, F., Fariña, A., Martínez-Prieto, M., Navarro, G.: Compressed $q$-gram indexing for highly repetitive biological sequences. In: Proc. BIBE. pp. 86–91 (2010)
11. Farzan, A., Gagie, T., Navarro, G.: Entropy-bounded representation of point grids. In: Proc. ISAAC. pp. 327–338 (2010), part II
12. Farzan, A.: Succinct Representation of Trees and Graphs. Ph.D. thesis, University of Waterloo (2009)
13. Ferragina, P., González, R., Navarro, G., Venturini, R.: Compressed text indexes: From theory to practice. ACM JEA 13 (2009), 30 pages
14. Ferragina, P., Manzini, G., Mäkinen, V., Navarro, G.: Compressed representations of sequences and full-text indexes. ACM Trans. on Alg. 3(2), article 20 (2007)
15. Fich, F., Munro, J., Poblete, P.: Permuting in place. SIAM J. on Comp. 24, 266 (1995)
16. Golynski, A., Munro, J.I., Rao, S.S.: Rank/select operations on large alphabets: a tool for text indexing. In: Proc. SODA. pp. 368–373 (2006)
17. Grossi, R., Gupta, A., Vitter, J.S.: High-order entropy-compressed text indexes. In: Proc. SODA. pp. 841–850 (2003)
18. Jacobson, G.: Space-efficient static trees and graphs. In: Proc. FOCS. pp. 549–554 (1989)
19. Keller, J.: A heuristic to accelerate in-situ permutation algorithms. Inf. Proc. Lett. 81(3), 119–125 (2002)
20. Kreft, S., Navarro, G.: Self-indexing based on LZ77. In: Proc. CPM. pp. 41–54 (2011)
21. Mäkinen, V., Navarro, G.: Rank and select revisited and extended. Theo. Comp. Sci. 387, 332–347 (2007)
22. Mäkinen, V., Välimäki, N.: Personal communication
23. Munro, J.I.: Tables. In: Proc. FSTTCS. pp. 37–42 (1996)
24. Navarro, G., Mäkinen, V.: Compressed full-text indexes. ACM Comp. Surv. 39(1), article 2 (2007)
25. Raman, R., Raman, V., Rao, S.S.: Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In: Proc. SODA. pp. 233–242 (2002)