On Wavelet Tree Construction*

German Tischler

Institut für Informatik, Universität Würzburg, Am Hubland,
97074 Würzburg, Germany
tischler@informatik.uni-wuerzburg.de

Abstract. The wavelet tree is a compact data structure allowing fast rank, select, access and other queries on non binary sequences. It has many applications in indexed pattern matching and data compression. In contrast to applications of wavelet trees their construction has so far been paid little attention. In this paper we discuss time and space efficient algorithms for constructing wavelet trees.

1 Introduction

The wavelet tree was introduced by Grossi, Gupta and Vitter in 2003 (cf. [5]). Augmented with binary rank and select dictionaries it allows rank, select and access queries on strings over an alphabet of size σ in time $O(\log \sigma)$. Such queries are used in many compressed index structures such as the compressed suffix array (cf. [5]) and FM type indexes (cf. [2]). The space required by the uncompressed wavelet tree is the same as the space required for the input string. Note that this statement refers only to the tree itself. Additional space is necessary for the rank and select dictionaries. We neglect this space in this paper, as these dictionaries are not required for the construction of the tree. Apart from rank, select and access queries, wavelet trees also efficiently support orthogonal range queries (counting and enumerating points in a given rectangle on the plane, cf. [9]) and range quantile queries (finding the k'th smallest element in a subinterval of a sequence, cf. [4]). Furthermore, wavelet trees facilitate data compression (cf. e.g. [1]). While many papers discuss functionality and applications of the wavelet tree, the construction of wavelet trees has so far gotten little attention. It is simple to set up an algorithm for constructing the wavelet tree in time $O(n \log \sigma)$ which is linear in the size of the number of bits in the output, but such a straight-forward algorithm will require at least an amount of memory which is three times as large as the size of the input. For a large input this may be prohibitive. In this paper we discuss algorithms for producing the wavelet tree of a string with various time and space bounds. We mainly focus on balanced wavelet trees. Most of our approaches can be translated to Huffman shaped wavelet trees. We omit most details about Huffman shaped wavelet trees though due to lack of space. Throughout this paper we assume a word RAM model with a word size of w bits and log denotes the base 2 logarithm.

* Part of this work was done while the author was a Newton Fellow at King's College London.

R. Giancarlo and G. Manzini (Eds.): CPM 2011, LNCS 6661, pp. 208–218, 2011.

The paper is structured as follows. In Section 2 we present efficient algorithms for sorting integers with respect to one of their bits. In Section 3 we present several algorithms for the generation of wavelet trees from strings. We conclude the paper in Section 4, where we give some open problems.

2 Stable Sorting of Bit Key Sequences

It is well known that a finite sequence of length m can be sorted stably and in place in time $O(m \log m)$ in the comparison model (cf. [10]). In the following we provide simple and practical algorithms for the case of binary keys in the word RAM model. By binary keys we mean that we are sorting integers of $k \in O(w)$ bits (not necessarily binaries), but for the comparison of two numbers only a certain single bit of the binary representation of the integers is relevant. In particular the keys are part of the integers. The length m of the sequence is assumed to be such that $\log m \in O(w)$

The following observation is derived from the permutation operation by Kronrod (cf. [8]). In its formulation we consider sequences of binaries, the generalisation to bit key sequences is straight forward.

Observation 1. *Let X and Y be sorted bit sequences of length $|X|$ and $|Y|$ concatenated in a sequence Z of length $|Z| = |X| + |Y|$. More precisely let $X = 0^{x_0}1^{x_1}$, $Y = 0^{y_0}1^{y_1}$ and $Z = XY$. Then X and Y can be merged stably and in place in time $O(|X| + |Y|) = O(|Z|)$.*

Proof. The stably merged sequence is obtained by computing

$$X[0\,..\,x_0 - 1]((X[x_0\,..\,x_0 + x_1 - 1])^R(Y[0\,..\,y_0 - 1])^R)^R Y[y_0\,..\,y_0 + y_1 - 1]$$

where R denotes the reversal operator. As reversal can be implemented in place in linear time, we obtain the given runtime bound.

An $O(m \log m)$ time stable in-place sorting algorithm for binary keys can be obtained by implementing a common merge sort approach while using the method given in the proof of Observation 1 for merging. Note that this approach requires only a constant number of words of additional memory. A pseudo-code version of the algorithm is shown in Algorithm 1. The parameter l denotes the length of pre sorted sub arrays, i.e. we would set it to 1 for an array we have no further information about. For $l > 1$ the algorithm works on the assumption that the portions $A[il\,..\,\min((i+1)l, m) - 1]$ are already sorted for $i = 0, 1, \ldots$ (if the left bound is larger than the right bound this notation denotes an empty sub array).

An interesting and important feature of this sorting algorithm consists in the fact that it still works if we do not move the bits we sort by in the reversal operations. This means we can sort the information attached to the sorted bits by the bits while keeping the keys in their original place. We name this algorithm working without moving the keys BITMERGESORTKEYS. To see why this works note that the values b_l and o_r computed in line 4 and 5 of the algorithm remain

Algorithm 1. Bit comparison based merge sort algorithm

BITMERGESORT(A, m, b, l)

1 **while** $l < m$ **do**
2 **for** $i \leftarrow 0$ **to** $\lceil \frac{m}{2l} \rceil - 1$ **do**
3 $(l_l, r_l, l_r, r_r) \leftarrow (2il, (2i+1)l), (2i+1)l, \min(2(i+1)l, m))$
4 $b_l \leftarrow |\{j \mid l_l \leq j < r_l \text{ and } A[j]\&b = b\}|$
5 $o_r \leftarrow |\{j \mid l_r \leq j < r_r \text{ and } A[j]\&b \neq b\}|$
6 REVERSE$(A[r_l - b_l \,..\, r_l - 1])$
7 REVERSE$(A[l_r \,..\, l_r + o_r - 1])$
8 REVERSE$(A[r_l - b_l \,..\, l_r + o_r - 1])$
9 $l \leftarrow 2l$

the same even if we do not move the keys. This is due to the fact that for the counting of the zeroes and ones it is unimportant where they are in the considered interval. The considered intervals though remain the same, as they do not depend on the sorted data.

If we are given another stable bit key sorting algorithm $\mathcal{A}$ sorting portions of length $\frac{m}{c}$ in time $t(\frac{m}{c})$ (with or without moving the keys), then we can obtain a bit key sorting algorithm (with or without moving the keys respectively) running in time $O(ct(\frac{m}{c}) + m \log c)$ and requiring the same space as $\mathcal{A}$ by first sorting portions of length $\frac{m}{c}$ and finally merging the portions using $\log c$ merging stages based on Observation 1.

Another interesting property of the algorithm BITMERGESORTKEYS consists in the fact that it can easily be modified to perform the reverse operation, i.e. restoring the unsorted from the sorted sequence while keeping the keys in place. For this purpose we only need to reverse the order of lines 6–8 and let l run from its maximal value down to 1 in the while loop. In fact this reversibility holds for the whole wavelet tree construction process such that we can obtain algorithms transforming a wavelet tree to the string it represents satisfying the same time and space constraints as in the string to wavelet tree direction. We omit the details due to lack of space.

The translation of the algorithm BITMERGESORTKEYS to the case of a Huffman coded sequence is not quite as simple as the algorithm for the case of block code. We need to know each bit of the Huffman code of a symbol to be able to determine the length of the code. We thus switch from an iterative bottom up formulation of the recursive structure of the algorithm to a traversal of the recursion tree in depth first order. We use a stack for the control of the recursion tree traversal. The algorithm thus requires $O(\log n)$ additional words of memory instead of a constant number of words. During the depth first traversal of the recursion tree we visit each node twice, first while descending into the tree and second while returning from the bottom of the tree. During the first visit we need to determine the code length assigned to the left and right child, as we

need to pass on the relevant information for the handling of the respective child nodes. During the second visit we merge the partial results produced for the child nodes. Whenever we have finished handling a node we need to return the number of relevant zero and one bits used for sorting.

The algorithms BITMERGESORT and BITMERGESORTKEYS can be used in combination with other sorting approaches. More precisely they can be used to combine partial results produced by other stable bit key sorting algorithms. The algorithm BITMERGESORT can be combined with a bucket sorting approach in a recursive scheme which produces free space within the array by compressing sorted portions as in [3]. This yields an $O(m \log k)$ time stable in place bit key sorting algorithm moving the keys. We omit the details here because this algorithm is in this form not suitable for our application of wavelet tree construction as we want the keys to stay in place while we sort. The algorithm may be interesting in it's own right though. Here we consider two approaches, where only the first one seems easily translatable to the case of sorting Huffman encoded sequences.

The first approach we consider is a combination of the algorithm BITMERGESORTKEYS and bucket sorting. It requires $\frac{(k-1)m}{c}$ bits of additional space where $c \geq 1$. We can choose c as a function of m. For values of about $c > \frac{m}{2}$ the algorithm turns into a pure run of BITMERGESORTKEYS, as we do not have sufficient space to sort more than one element using bucket sort. Assume we have $d(k-1)$ bits of additional space, where $d = \lfloor \frac{m}{c} \rfloor > 1$. Then we sort each sub array $A[id \,..\, \min((i+1)d, m)-1]$ using bucket sort. Note that there is no need to copy the key bits to the buckets, we just leave them where they are while we first copy the attached information to the correct of the two buckets and then copy it back in sorted order (i.e. first the bucket for zero, then the bucket for one). After all the sub arrays have been handled, we merge the partial results via the algorithm BITMERGESORTKEYS using $l = d$. We call this algorithm BITBUCKETSORTKEYS. It takes time $O(m \log c)$, i.e. by choosing the amount of additional memory we provide we can get any runtime between $O(m)$ and $O(m \log m)$. The linear time version requires additional space within a constant of $m(k-1)$ bits, the $O(m \log m)$ time version requires only a constant number of additional memory words, i.e. $O(\log m)$ additional bits.

The second approach uses the sorting method proposed in appendix B of [7]. It uses additional space in the order of $O(\sqrt{m}(\log m + k))$ additional bits to stably sort a sequence of k bit numbers of length m according to binary keys in time $O(m)$. As in the simpler approach BITBUCKETSORTKEYS, there is no need to move the keys during the procedure. The algorithm works in two phases. In the first phase, the integers are sorted into buckets, where each bucket is represented as a list of blocks. Each block has space for $O(\sqrt{m})$ elements, i.e. it requires $O(\sqrt{m}k)$ bits. In the case of binary keys we need $O(1)$ blocks in addition to the array to be sorted. These blocks are called the external buffer. The array is scanned from left to right and elements are distributed to the buckets. This effectively means they are copied to the external buffer. As soon as a block in the external buffer is full, it is copied to some free block in the original array.

Due to the allocation strategy of the algorithm it is guaranteed that such a free block exists. The block is not necessarily copied into the right position of the array, thus we need to note where it should be. This is stored in an additional array of size $O(\sqrt{m}\log\sqrt{m}) = O(\sqrt{m}\log m)$ bits. In the second phase the blocks are permuted to obtain the sorted sequence. This is done by following cycles in the permutation stored in the additional array. The interested reader can find more details in [7]. We call the variant of this algorithm leaving the keys in place KSBBITBUCKETSORTKEYS.

The algorithm BITBUCKETSORTKEYS can be translated to the case of a Huffman coded array. Like for the translation of the algorithm BITMERGESORTKEYS we use a stack facilitating a depth first traversal of the recursion tree. When we have reached a sufficient depth at which the code length of the considered sub array is reduced enough we switch to bucket sorting and truncate the recursion tree. From the bucket sorting we need to return the number of 0 and 1 key bits found as if we would return from a recursive call to BITMERGESORTKEYS. It is unclear, whether the approach used in KSBBITBUCKETSORTKEYS can easily be applied to Huffman coded sequences.

A simple modification of BITMERGESORT allows us to transform a sequence $A = a_0, b_0, a_1, b_1, \ldots, b_{m-1}$ such that the a_i are bits and the b_i numbers of $k-1$ bits in place in time $O(m \log m)$ to the sequence $B = a_0, a_1, a_2, \ldots, a_{m-1}, b_0, b_1, \ldots, b_{m-1}$. For this purpose we may imagine the a_i are assigned key 0, the b_i key 1 and we perform the sorting procedure according to these keys. The reversal operations are easily modified for taking into account that the elements of the sequence do not all have the same length in this case. We name this algorithm TRANSPOSE$_k$.

If we are given another method which is able to transpose sequences of length $\frac{m}{c}$ for some $c \geq 1$ in time $O(t(\frac{m}{c}))$, then we can use an adapted version of TRANSPOSE$_k$ and this other method to transpose an array in time $O(ct(\frac{m}{c}) + m \log c)$ by first transposing portions of length $\frac{m}{c}$ and finally merging the portions using the adapted version of TRANSPOSE$_k$.

As for sorting we can consider several methods for transposition using additional space. A first simple method for transposing a sequence of length $\frac{m}{c}$ in time $O(\frac{m}{c})$ can be implemented easily if we have $\frac{m}{c}$ additional bits of space. We first copy the bits moved to the front to the additional space, move the other numbers into place by copying them to the back and copy the bits from the additional space back to the front. A second method can be based on the sorting scheme used in KSBBITBUCKETSORTKEYS. The only adaption necessary here is to make sure we fill blocks of a fixed size, i.e. $O(\sqrt{m}(\delta - 1))$ single bits make one block and $O(\sqrt{m})$ integers (the rest of the integers without the keys) make one block, so the second stage can handle blocks of equal size in bits. As above the justification for describing the first simpler method is that it can easily be applied to Huffman coded sequences, while this is not clear for the second method.

In the presence of available additional memory we will mean these hybrid approaches instead of the pure merge sort based variant below if we use the name

TRANSPOSE$_k$ (i.e. using the second method similiar to KSBBITBUCKETSORTKEYS for block code and the first method for Huffman code).

As the transposition method given is an adaption of the sorting method, the translation to the Huffman code case can formulated analogously.

3 Generating Wavelet Trees

3.1 Definitions

Let Σ be a finite alphabet equipped with an injective function $r : \Sigma \mapsto \mathbb{N}$ assigning a rank to each symbol. We assume that $|\Sigma| \geq 2$, otherwise all problems handled in the following become trivial. Let $\sigma = \max\{r(a)|a \in \Sigma\}$ be the maximal rank of a symbol in the alphabet. Each rank and thus each alphabet symbol can be represented in $\delta = \lceil \log_2(\sigma + 1) \rceil$ bits. We assume that $\delta \in O(w)$, i.e. our word RAM can access the representation of each symbol in constant time.

Let $b : \Sigma \times \mathbb{N} \mapsto \{0, 1\}$ be defined by $b(a, i) = 1$ iff the i'th least significant bit in $r(a)$ is 1, i.e. $r(a) = \sum_{i=0}^{\delta-1} b(a, i)2^i$.

Throughout this section we consider an input sequence $S \in \Sigma^n$ of length $n \in \mathbb{N}$, which we assume is given as a bit sequence

$$B = b(S[0], \delta - 1)b(S[0], \delta - 2) \dots b(S[0], 0)b(S[1], \delta - 1) \dots b(S[n - 1], 0)$$

of length $|B| = n\delta$, where we assume $\log n \in O(w)$. We identify the symbol $S[i]$ with the block of bits $B[\delta i] \dots B[\delta(i + 1) - 1]$ in B.

Let B_k for $0 \leq k < \delta$ denote the subsequence given by

$$\begin{aligned} B_k = {} & b(S[0], \delta - k - 1)b(S[0], \delta - k - 2) \dots b(S[0], 0) \\ & b(S[1], \delta - k - 1) \dots b(S[n - 1], 0) \ , \end{aligned}$$

i.e. the k most significant bits are masked from each symbol.

In analogy to S and B we define for $0 \leq k < \delta$ the sequence S_k of length n by setting $S_k[i] = B_k[i(\delta - k)] \dots B_k[(i + 1)(\delta - k) - 1]$. Let $S^{(i,k)}$ denote the subsequence of S containing exactly those symbols a such that $\sum_{j=0}^{k-1} b(a, \delta - j - 1)2^{k-j-1} = i$, i.e. the binary interpretation of the top k bits is i. We use S_k^i as a more convenient notation for $(S^{(i,k)})_k$ below and use S^i as a short form of $S^{(i,1)}$.

The *balanced wavelet tree* of the string S is a binary tree such that the root of the tree contains the bit sequence $B[0], B[\delta], B[2\delta], \dots, B[(n - 1)\delta]$, i.e. the most significant bit of each character's rank. If $\delta = 1$, then the root of the balanced wavelet tree is a leaf. Otherwise, if $\delta > 1$, then the left child of the root is the balanced wavelet tree of the subsequence S_1^0 (i.e. the subsequence S^0 with the most significant bit removed. Accordingly we substitute δ by $\delta - 1$ in the construction.) and the right child is the balanced wavelet tree of S_1^1. The tree is represented by the concatenation of the bit-vectors found in the tree nodes traversed in breadth first order. The motivation for storing the nodes in breadth

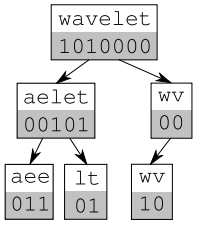

Fig. 1. Wavelet tree for the string *wavelet* represented by 101000000101000110110

first order is that this representation augmented with a rank dictionary allows us to navigate in the tree without using pointers, as the start of the left child of a node always equals the start of a node plus n bits and the start of the right child can be determined by the start of the left child plus the number of 0 bits assigned to the node. The width of a left (right) child is determined by the number of zero (one) bits in its parent node.

For the computation of the rank, select, access, etc. functions using the balanced wavelet tree the reader is referred to the respective literature (cf. [4,5,9]).

As an example consider the string $T = \texttt{wavelet}$ over the alphabet $\Gamma = \{\texttt{a}, \texttt{e}, \texttt{l}, \texttt{t}, \texttt{v}, \texttt{w}\}$ equipped with the rank function given by the pairs $(\texttt{a}, 0 = 000)$, $(\texttt{e}, 1 = 001)$, $(\texttt{l}, 2 = 010)$, $(\texttt{t}, 3 = 011)$, $(\texttt{v}, 4 = 100)$ and $(\texttt{w}, 5 = 101)$. The balanced wavelet tree of T is shown in Figure 1. The tree only consists of the bit sequences shown with a grey background, the letters are provided for demonstration purposes only.

For a precise definition of Huffman shaped wavelet trees the reader is referred to the literature (cf. [2]). In short the balanced structure of the block code based wavelet tree is replaced by a tree structure based on the tree of a Huffman code (cf. [6]) derived from the input sequence. This reduces the space used from δn bits to Mn bits, where M denotes the average code length used by the code per symbol for the complete input sequence. In contrast to the balanced wavelet tree navigation in the Huffman shaped wavelet tree is not easily done without pointers between the nodes. We consider the pre order depth first concatenation of the nodes' binary sequences as the representation of the tree. For the construction of Huffman shaped wavelet trees we assume that the input sequence is Huffman coded and we have supporting data structures allowing fast encoding and decoding of the underlying code.

3.2 Wavelet Tree Construction Algorithms

The algorithms we will present generate the tree top down. They are based on the sorting schemes KSBBITBUCKETSORTKEYS in the balanced case (where we assume we have $0 \leq O(\frac{\sqrt{n}(\delta+\log n)}{c}) \leq O(\sqrt{n}(\delta + \log n))$ free bits) and BITBUCKETSORTKEYS in the Huffman case (where we assume we have $0 \leq \frac{n(M-1)}{c} \leq n(M-1)$ free bits).

We first discuss the case of balanced wavelet trees. We consider one breadth and one depth first approach. The first algorithm which we call WTBFS is based on a breadth first traversal, i.e. it generates the tree level per level, while it uses only a constant number of words of additional space for the control of the tree traversal. It consists of $\delta - 1$ repetitions of two phases. Consider iteration k, where $0 \leq k < \delta - 1$. The first phase sorts 2^k arrays of total length n using the algorithm KSBBITBUCKETSORTKEYS. These arrays are $S_k^0, \ldots, S_k^{2^k-1}$, i.e. the data relevant for the levels k to $\delta - 1$ of the tree. The second phase uses the algorithm TRANSPOSE$_k$ to move the bits corresponding to the k'th level into place. Both phases take $O(n \log c)$ time while we have $\delta - 1$ such phases, thus the overall time for the sort and transposition operations is $O(\delta n \log c)$. As an example consider the string `wavelet` represented by 5041213 which is (101)(000)(100)(001)(010)(001)(011) in binary where $\delta = 3$ (the parentheses are inserted to improve readability). We first sort by the most significant bit while keeping the keys in place. This yields (100)(001)(110)(001)(011)(001)(000). The first bit in each triple is still the same. The second and third bit of each triple is obtained by taking the second and third bit of the original sequence sorted by the first bit (which would be (000)(001)(010)(001)(011)(101)(100), this sequence is however not computed explicitly). Then we apply TRANSPOSE$_3$ to (100)(001)(110)(001)(011)(001)(000), which yields the intermediate result (1010000)((00)(01)(10)(01)(11)(01)(00)). The first $n = 7$ bits are the first n bits of the output. The rest is the concatenation of the sequences S_1^0 and S_1^1. Subsequently for level $k > 0$ below the root we are confronted with the problem of determining where the sequences S_k^i for $0 \leq i < 2^k$ start and end in the bit sequence starting from index $k\delta$. To find the number of symbols in the input which equal i in the k most significant bits, we use a top down scan of the existing output. The starting point of the sequence S_k^i is obtained by accumulating the number of symbols assigned to sequences S_k^j for $j < i$. These operations can be performed in time $O(\sigma\delta n)$ using a constant number of additional words of memory for one level. The overall runtime of the algorithm is thus $O(\delta n \log c + \delta\sigma\delta n) = O(\delta n(\log c + \sigma\delta))$. Consider for instance our example above. We have produced the output for the first level of the tree. The next level has two inner nodes. We consider them from low to high value. For $i = 0$ there is no smaller value, thus the left bound is 0. The number of values which equal $i = 0$ is 5 (the number of zero bits on the first level). Thus the range of S_1^0 extends from bit $1 \cdot n + 0 \cdot (3 - 1)$ to $1 \cdot n + (0 + 5) \cdot (3 - 1)$ (where $1 \cdot n$ denotes the start of the input to be processed (we have processed one level, thus the 1) and $3 = \delta$). The range for S_1^1 extends from the right bound of S_1^{1-1} to $1 \cdot n + (0 + 5 + 2) \cdot (3 - 1)$ (where the 2 denotes the number of 1 bits on the first level).

The second algorithm which we call WTDFS performs the sorting along a pre order depth first traversal of the tree using a stack of depth $O(\delta)$, where each stack element consists of an index interval and a depth and thus requires $O(\log n + \log \delta)$ bits. In the beginning the stack contains the pair $([0, n), 0)$ representing the root of the tree. All the transposition operations are postponed until

the end of the algorithm. A slight adaption of the sorting algorithm is required due to the fact that the sorted data is not contiguous (i.e. when we perform the sorting for the lower levels, the bits corresponding to the levels above will still be between the bits relevant for sorting), this however does not change the runtime of the algorithm. The algorithm uses $O(\delta)$ additional words of memory and has a runtime of $O(\delta n \log c)$. During the stack based traversal we need to take care not to push empty intervals onto the stack, otherwise the term $n \log c$ in the runtime of the algorithm is replaced by $\max\{n \log c, \sigma\}$. As an example again consider the string `wavelet` represented by (101)(000)(100)(001)(010)(001)(011). On the stack we find $([0, 7), 0)$. We first sort the interval $[0, 7)$ at level 0 by the most significant bit while keeping the keys in place. As above this yields (1(00))(0(01))(1(10))(0(01))(0(11))(0(01))(0(00)). Now we count the number of zeroes found in the most significant bits. This number is 5. Thus we push the tuples $([5, 7), 1)$ and $([0, 5), 1)$ onto the stack as to be handled after we have popped the tuple $([0, 7), 0)$. The interval $[0, 5)$ denotes the left child node, $[5, 7)$ the right child node and the final 1 denotes the depth of the respective child nodes. The next node handled is the left child of the root denoted by $([0, 5), 1)$. We sort the subsequence (00)(01)(10)(01)(11) (the interval $[0, 5)$ without the output for level zero of the tree) by the most significant bit while keeping the keys in place. This yields (00)(01)(11)(00)(11) and the global bit string becomes (1(00))(0(01))(1(11))(0(00))(0(11))(0(01))(0(00)). The node has no children, so we push no elements onto the stack. The next node handled is $([5, 7), 1)$. We sort (01)(00) by the most significant bit while leaving the keys in place and obtain (01)(00). Globally we obtain (1(00))(0(01))(1(11))(0(00))(0(11))(0(01))(0(00)). The node has no children, so we push no elements onto the stack. The stack is now empty. It remains to perform the postponed transpositions. We first perform TRANSPOSE$_3$ and then TRANSPOSE$_2$, where TRANSPOSE$_3$ handles all the bits and TRANSPOSE$_2$ does not touch the front n bits. This gives us our final result (1010000)(0010100)(0110110).

The following table shows a comparison of the two algorithms.

algorithm	runtime	additional space (bits)
WTBFS	$O(\sigma\delta^2 n + \delta n \log c)$	$O(\frac{\sqrt{n}(\delta+\log n)}{c}) + O(\log n)$
WTDFS	$O(\delta n \log c)$	$O(\frac{\sqrt{n}(\delta+\log n)}{c}) + O(\delta(\log n + \log \delta))$

For the generation of Huffman shaped wavelet trees we use a pre order depth first traversal of the tree and perform the transposition operations directly after a node has been handled. For the sorting we use an additional space of $\frac{(M-1)n}{c} + O(\log^2 n)$ bits for some $c \geq 1$ which may be a function of n. As above M denotes the average code length used by the Huffman code per symbol for the complete input sequence. The term $\frac{(M-1)n}{c}$ may be reduced to zero by using a pure merge sort approach. The $O(\log^2 n)$ term stems from the stack of depth $\log n$ used for sorting. Let m denote the maximum code length of the Huffman code used. Then we need a stack of size $O(m)$ to facilitate the depth first traversal through the

wavelet tree during construction. The elements on the stack consist of the left and right bound of the relevant sub array (each stored using $O(\log(Mn))$ bits) and the code prefix assigned to the represented node (which has a length of up to m bits. Each stack element stores only a single bit of the prefix though). Thus we need $O(m \log(Mn))$ bits for the stack. Overall the space required for the algorithm thus is $\frac{(M-1)n}{c} + O(\log^2 n) + m \log(Mn)$ bits. Assuming the encoding and decoding routines for the Huffman code are fast enough, we can expect the algorithm to run in time $O(Mn \log c)$.

4 Conclusion

In this paper we have discussed time and space efficient algorithms for constructing wavelet trees from strings. We have given algorithms which can be parametrised by the amount of additional space they are allowed to use for both balanced and Huffman shaped wavelet trees. For the case of balanced wavelet trees this additional space can be between constant additional space (a fixed number of words or $O(\log n)$ additional bits) and $O(\sqrt{n}(\delta + \log n))$ bits. On the assumption of a constant sized alphabet we can get any runtime between $O(n)$ and $O(n \log n)$. Interesting problems include whether the $O(n \log n)$ time bound can be broken, if we do not allow an algorithm to use additional memory and if we can obtain a linear runtime in $O(\delta n)$ using an additional space of $o(\sqrt{n}(\delta + \log n))$ bits.

The author would like to thank the anonymous reviewers for helpful comments and Juha Kärkkäinen and Roberto Grossi for interesting discussions on the topic of this paper, in particular concerning the hint to the linear time sorting algorithm given by Kärkkäinen et al. in [7].

References

1. Ferragina, P., Giancarlo, R., Manzini, G.: The myriad virtues of wavelet trees. Inf. Comput. 207(8), 849–866 (2009)
2. Ferragina, P., Manzini, G., Mäkinen, V., Navarro, G.: An alphabet-friendly FM-index. In: Apostolico, A., Melucci, M. (eds.) SPIRE 2004. LNCS, vol. 3246, pp. 150–160. Springer, Heidelberg (2004)
3. Franceschini, G., Muthukrishnan, S.M., Pătraşcu, M.: Radix sorting with no extra space. In: Arge, L., Hoffmann, M., Welzl, E. (eds.) ESA 2007. LNCS, vol. 4698, pp. 194–205. Springer, Heidelberg (2007)
4. Gagie, T., Puglisi, S., Turpin, A.: Range quantile queries: Another virtue of wavelet trees. In: Karlgren, J., Tarhio, J., Hyyrö, H. (eds.) SPIRE 2009. LNCS, vol. 5721, pp. 1–6. Springer, Heidelberg (2009)
5. Grossi, R., Gupta, A., Vitter, J.S.: High-order entropy-compressed text indexes. In: SODA, pp. 841–850 (2003)
6. Huffman, D.A.: A method for the construction of minimum-redundancy codes. Proceedings of the Institute of Radio Engineers 40(9), 1098–1101 (1952)

7. Kärkkäinen, J., Sanders, P., Burkhardt, S.: Linear work suffix array construction. J. ACM 53(6), 918–936 (2006)
8. Kronrod, M.A.: Optimal ordering algorithm without operational field. Soviet Math. Dokl. 10, 744–746 (1969)
9. Mäkinen, V., Navarro, G.: Rank and select revisited and extended. Theoretical Computer Science 387(3), 332–347 (2007); The Burrows-Wheeler Transform
10. Salowe, J., Steiger, W.: Simplified stable merging tasks. J. Algorithms 8(4), 557–571 (1987)