Olympiads in Informatics, 2016, Vol. 10, 19–37

DOI: 10.15388/ioi.2016.02

Wavelet Trees for Competitive Programming

Robinson CASTRO[1], Nico LEHMANN[1], Jorge PÉREZ[1,2], Bernardo SUBERCASEAUX[1]

[1]*Department of Computer Science, Universidad de Chile*
Beauchef 851, Santiago, Chile
[2]*Chilean Center for Semantic Web Research*
email: {rocastro, nlehmann, jperez, bsuberca}@dcc.uchile.cl

Abstract. The wavelet tree is a data structure to succinctly represent sequences of elements over a fixed but potentially large alphabet. It is a very versatile data structure which exhibits interesting properties even when its compression capabilities are not considered, efficiently supporting several queries. Although the wavelet tree was proposed more than a decade ago, it has not yet been widely used by the competitive programming community. This paper tries to fill the gap by showing how this data structure can be used in classical competitive programming problems, discussing some implementation details, and presenting a performance analysis focused in a competitive programming setting.

Key words: wavelet tree, data structures, competitive programming, quantile query, range query.

1. Introduction

Let $S = (s_1, \ldots, s_N)$ be a sequence of integers and consider the following query over S.

Query 1. *Given a pair of indices (i, j) and a positive integer k, compute the value of the k-th smallest element in the sequence $(s_i, s_{i+1}, \ldots, s_j)$.*

Notice that Query 1 essentially asks for the value of the element that would occupy the k-th position when we sort the sequence $(s_i, s_{i+1}, \ldots, s_j)$. For example, for the sequence $S = (3, 7, 5, 2, 3, 2, 9, 3, 5)$ and the query having $(i, j) = (3, 7)$ and $k = 4$, the answer would be 5, as if we order sequence $(s_3, s_4, s_5, s_6, s_7) = (5, 2, 3, 2, 9)$ we would obtain $(2, 2, 3, 5, 9)$ and the fourth element in this sequence is 5. Consider now the following *update* query.

Query 2. *Given an index i, swap the elements at positions i and $i + 1$.*

That is, if $S = (3, 7, 5, 2, 3, 2, 9, 3, 5)$ and we apply Query 2 with index 5, we would obtain the sequence $S' = (3, 7, 5, 2, 2, 3, 9, 3, 5)$.

Consider now a competitive programming setting in which an initial sequence of 10^6 elements with integer values in the range $[-10^9, 10^9]$ is given as input. Assume that

a sequence of 10^5 queries, each query of either type 1 or type 2, is also given as input. The task is to report the answer of all the queries of type 1 considering the applications of all the update queries, every query in the same order in which they appear in the input. The wavelet tree (Grossi, 2015) is a data structure that can be used to trivially solve this task within typical time and memory limits encountered in programming competitions.

The wavelet tree was initially proposed to succinctly represent sequences while still being able to answer several different queries over this succinct representation (Grossi *et al.*, 2003; Navarro, 2014; Grossi, 2015). Even when its compression capabilities are not considered, the wavelet tree is a very versatile data structure. One of the main features is that it can handle sequences of elements over a fixed but potentially large alphabet; after an initial preprocessing, the most typical queries (as Query 1 above) can be answered in time $O(\log \sigma)$, where σ is the size of the underlying alphabet. The preprocessing phase usually constructs a structure of size $O(K \times n \log \sigma)$ for an input sequence of n elements, where K is a factor that will depend on what additional data structures we use over the classical wavelet tree construction when solving a specific task.

Although it was proposed more than a decade ago (Grossi *et al.*, 2003), the wavelet tree has not yet been widely used by the competitive programming community. We conducted a *social experiment* publishing a slightly modified version of Query 1 in a well known Online-Judge system. We received several submissions from experienced competitive programmers but none of them used a wavelet tree implementation to solve the task. This paper tries to fill the gap by showing how this structure can be used in classical (and no so classical) competitive programming tasks. As we will show, its good performance to handle big alphabets, the simplicity of its implementation, plus the fact that it can be smoothly *composed* with other typical data structures used in competitive programming, give the wavelet tree a considerable advantage over other structures.

Navarro (2014) presents an excellent survey of this data structure showing the most important practical and theoretical results in the literature plus applications in a myriad of cases, well beyond the one discussed in this paper. In contrast to Navarro's survey, our focus is less on the properties of the structure in general, and more on its practical applications, some adaptations, and also implementation targeting specifically the issues encountered in programming competitions. Nevertheless, we urge the reader wanting to master wavelet trees to carefully read the work by Navarro (2014).

2. The Wavelet Tree

The wavelet tree (Grossi, 2015) is a data structure that recursively partitions a sequence S into a tree-shaped structure according to the values that S contains. In this tree, every node is associated to a subsequence of S. To construct the tree we begin from the root, which is associated to the complete sequence S. Then, in every node, if there are two or more distinct values in its corresponding sequence, the set of values is split into two non-empty sets, L and R; all the elements of the sequence whose values belong to L

form the left-child subsequence; all the elements whose values belong to R form the right-child subsequence. The process continues recursively until a leaf is reached; a leaf corresponds to a subsequence in which all elements have the same value, and thus no partition can be performed.

Fig. 1 shows a wavelet tree constructed from the sequence

$$S = (3, 3, 9, 1, 2, 1, 7, 6, 4, 8, 9, 4, 3, 7, 5, 9, 2, 7, 3, 5, 1, 3).$$

We split values in the first level into sets $L = \{1, \ldots, 4\}$ and $R = \{5, \ldots, 9\}$. Thus the left-child of S is associated to $S' = (3, 3, 1, 2, 1, 4, 4, 3, 2, 3, 1, 3)$. If we continue with the process from this node, we can split the values into $L' = \{1, 2\}$ and $R' = \{3, 4\}$. In this case we obtain as right child a node associated with the sequence $S'' = (3, 3, 4, 4, 3, 3, 3)$. Continuing from S'', if we split the values again (into sets $\{3\}$ and $\{4\}$), we obtain the subsequence $(3, 3, 3, 3, 3)$ as left child and $(4, 4)$ as right child, and the process stops.

For simplicity in the exposition, given a wavelet tree T we will usually talk about *nodes* in T to denote, interchangeably, the actual nodes that form the tree and the subsequences associated to those nodes. Given a node S in T, we denote by $\texttt{Left}_T(S)$ its left-child and by $\texttt{Right}_T(S)$ its right-child in T. The *alphabet* of the tree is the set of different values that its root contains. We usually assume that the alphabet of a tree is a set $\Sigma = \{1, 2, \ldots, \sigma\}$. Without loss of generality, and in order to simplify the partition process, we will assume that every node S in T has an associated value $\texttt{m}_T(S)$ such that $\texttt{Left}_T(S)$ contains the subsequence of S composed of all elements of S with values $c \leq \texttt{m}_T(S)$, and $\texttt{Right}_T(S)$ the subsequence of S composed of all elements with values $c > \texttt{m}_T(S)$. (In Fig. 1 the value $\texttt{m}_T(S)$ is depicted under every node.) We can also associate to every node S in T, two values $\texttt{l}_T(S)$ and $\texttt{r}_T(S)$, such that S corresponds to the subsequence of the root of T containing all the elements whose values are in the range $[\texttt{l}_T(S), \texttt{r}_T(S)]$. Notice that a wavelet tree with alphabet $\{1, \ldots, \sigma\}$ has exactly σ leaves. Moreover, if the construction is done splitting the alphabet into halves in every node, the depth of the wavelet tree is $O(\log \sigma)$.

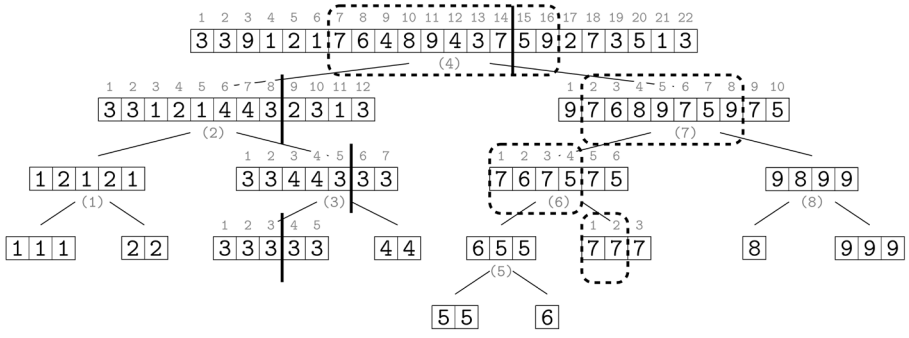


Fig. 1. Wavelet tree for the sequence $S = (3, 3, 9, 1, 2, 1, 7, 6, 4, 8, 9, 4, 3, 7, 5, 9, 2, 7, 3, 5, 1, 3)$. Solid lines illustrate the execution of $\texttt{rank}_3(S, 14)$. Dashed lines show the execution of $\texttt{quantile}_6(S, 7, 16)$.

As we will see in Section 4, when implementing a wavelet tree the complete information of the elements stored in each subsequence of the tree is not actually necessary. But before giving any details on how to efficiently implement the wavelet tree, we use the abstract description above to show the most important operations over this data structure.

Traversing the Wavelet Tree

The most important abstract operation to traverse the wavelet tree is to map an index in a node into the corresponding indexes in its left and right children. As an example, let S be the root node of wavelet tree T in Fig. 1, and $S' = \texttt{Left}_T(S)$. Index 14 in S (marked in the figure with a solid line) is mapped to index 8 in S' (also marked in the figure with a solid line). That is, the portion of sequence S from index 1 to index 14 that is mapped to its left child, corresponds to the portion of sequence S' from index 1 to 8. On the other hand, index 14 in root sequence S is mapped to index 6 in $\texttt{Right}_T(S)$.

We encapsulate the operations described above into two abstract functions, $\texttt{mapLeft}_T(S, i)$ and $\texttt{mapRight}_T(S, i)$, for an arbitrary non-leaf node S of T. In Fig. 1, if S is the root, $S' = \texttt{Left}_T(S)$ and $S'' = \texttt{Right}_T(S')$, then we have $\texttt{mapLeft}_T(S, 14) = 8$, $\texttt{mapRight}_T(S', 8) = 5$ and $\texttt{mapLeft}_T(S'', 5) = 3$ (all indexes marked with solid lines in the figure). Function $\texttt{mapLeft}_T(S, i)$ is essentially counting how many elements of S until index i are mapped to the left-child partition of S. Similarly $\texttt{mapRight}_T(S, i)$ counts how many elements of S until index i are mapped to the right-child partition of S.

As we will describe in Section 4, these two operations can be efficiently implemented (actually can be done in constant time). But before going into implementation details, we show how $\texttt{mapLeft}_T$ and $\texttt{mapRight}_T$ can be used to answer three different queries by traversing the wavelet tree, namely, *rank*, *range quantile*, and *range counting*.

2.1. *Rank*

The *rank* is an operation performed over a sequence S that counts the occurrences of value q until an index i of S. It is usually denoted by $\texttt{rank}_q(S, i)$. That is, if $S = (s_1, \ldots, s_n)$ then

$$\texttt{rank}_q(S, i) = |\{k \in \{1, \ldots, i\} \mid s_k = q\}|.$$

So for example, in sequence S in Fig. 1 we have that $\texttt{rank}_3(S, 14) = 3$.

Assume that T is a wavelet tree for S, then $\texttt{rank}_q(S, i)$ can be easily computed with the following strategy. If $q \leq \texttt{m}_T(S)$ then we know that all occurrences of q in S appear in the sequence $\texttt{Left}_T(S)$, and thus $\texttt{rank}_q(S, i) = \texttt{rank}_q(\texttt{Left}_T(S), \texttt{mapLeft}_T(S, i))$. Similarly, if $q > \texttt{m}_T(S)$ then $\texttt{rank}_q(S, i) = \texttt{rank}_q(\texttt{Right}_T(S), \texttt{mapRight}_T(S, i))$. We

repeat this process until we reach a leaf node; if we reach a leaf S with this process, we know that $\texttt{rank}_q(S, i) = i$.

In Fig. 1 the execution of $\texttt{rank}_3(S, 14)$ is depicted with solid lines. We map index 14 down the tree using either $\texttt{mapLeft}_T$ or $\texttt{mapRight}_T$ depending on the $\texttt{m}_T$ value of every node in the path. We first map 14 to 8 (to the left), then 8 to 5 (to the right) and finally 5 to 3 (to the left), reaching a leaf node. Thus, the answer to $\texttt{rank}_3(S, 14)$ is 3.

Rank is computed by performing $O(\log \sigma)$ calls to (either) $\texttt{mapLeft}_T$ or $\texttt{mapRight}_T$, thus the time complexity is $O(M \times \log \sigma)$ where M is the time needed to compute the map functions. Also notice that a rank operation that counts the occurrences of q between indexes i and j can be computed by $\texttt{rank}_q(S, j) - \texttt{rank}_q(S, i-1)$, and thus the time complexity is also $O(M \times \log \sigma)$.

2.2. *Range Quantile*

The range quantile operation is essentially Query 1 described in the introduction: given a sequence $S = (s_1, \ldots, s_n)$, $\texttt{quantile}_k(S, i, j)$ is the value of the k-th smallest element in the sequence $(s_i, s_{i+1}, \ldots, s_j)$. For instance in Fig. 1 for the root sequence S we have that $\texttt{quantile}_6(S, 7, 16) = 7$. It was shown by Gagie *et al.* (2009) that wavelet trees can efficiently solve this query.

To describe how the wavelet tree can solve quantile queries, lets begin with a simpler version. Assume that $i = 1$ and thus, we want to find the k-th smallest element among the first j elements in S. Then having a wavelet tree T for S, $\texttt{quantile}_k(S, 1, j)$ can be easily computed as follows. Let $c = \texttt{mapLeft}_T(S, j)$. Recall that $\texttt{mapLeft}_T(S, j)$ counts how many elements of S until index j are mapped to the left-child of S. Thus if $k \leq c$ then we know for sure that the element that we are searching for is in the left subtree, and can be computed as

$$\texttt{quantile}_k(\texttt{Left}_T(S), 1, \texttt{mapLeft}_T(S, j)).$$

On the other hand, if $k > c$ then the element that we are searching for is in the right subtree, but it will no longer be the k-th smallest in $\texttt{Right}_T(S)$ but the $(k-c)$-th smallest and thus can be computed as

$$\texttt{quantile}_{(k-c)}(\texttt{Right}_T(S), 1, \texttt{mapRight}(S, j)).$$

This process can be repeated until a leaf node is reached, in which case the answer is the (single) value stored in that leaf.

When answering $\texttt{quantile}_k(S, i, j)$ the strategy above generalizes as follows. We first compute $c = \texttt{mapLeft}_T(S, j) - \texttt{mapLeft}_T(S, i-1)$. Notice that c is the number of elements of S from index i to index j (both inclusive) that are mapped to the left. Thus, if $k \leq c$ then the element we are searching for is in the leftchild of S between the indexes $\texttt{mapLeft}_T(S, i-1) + 1$ and $\texttt{mapLeft}_T(S, j)$, and thus the answer is

$$\texttt{quantile}_k(\texttt{Left}_T(S), \texttt{mapLeft}_T(S, i-1)+1, \texttt{mapLeft}_T(S, j)).$$

If $k > c$ then the desired element is in the right and can be computed as

$$\texttt{quantile}_{(k-c)}(\texttt{Right}_T(S), \texttt{mapLeft}_R(S, i-1)+1, \texttt{mapLeft}_R(S, j)).$$

As before, the process is repeated until a leaf node is reached, in which case the answer is the value stored in that leaf. In Fig. 1 the complete execution of $\texttt{quantile}_6(S, 7, 16)$ is depicted with dashed boxes in every visited node.

As for the case of the rank operation, quantile can be computed in time $O(M \times \log\sigma)$ where M is the time needed to compute the map functions.

2.3. *Range Counting*

The range counting query $\texttt{range}_{[x,y]}(S, i, j)$ counts the number of elements with values between x and y in positions from index i to index j. That is, if $S = (s_1, \ldots, s_n)$ then

$$\texttt{range}_{[x,y]}(S, i, j) = |\{k \in \{i, \ldots, j\} \mid x \leq s_k \leq y\}|.$$

A sequence of size n can be understood as the representation of a grid with n points such that no two points share the same row. A general set of n points can be mapped to such a grid by storing real coordinates somewhere and breaking ties somehow. For this representation the range counting query corresponds to count the number of points in a given subrectangle (Navarro, 2014).

To answer a range counting query over a wavelet tree T we can use the following recursive strategy. Consider the interval $[\texttt{l}_T(S), \texttt{r}_T(S)]$ of possible elements of a sequence S. If $[\texttt{l}_T(S), \texttt{r}_T(S)]$ does not intersect $[x, y]$, then no element of the sequence is in $[x, y]$ and the answer is 0. Another case occurs when $[\texttt{l}_T(S), \texttt{r}_T(S)]$ is totally contained in $[x, y]$; in this case all the elements of the sequence between i and j are counted, so the answer is $|[i, j]| = j - i + 1$.

The last case (the recursive one) is when $[\texttt{l}_T(S), \texttt{r}_T(S)]$ intersects $[x, y]$ (but is not completely contained in $[x, y]$); in that case the answer is the sum of the range counting query evaluated in both children. The queries for children are called with the same $[x, y]$ as in the parent's call, but the indexes i and j are replaced by the mappings of these indexes. That is, the answer is

$$\texttt{range}_{[x,y]}(\texttt{Left}_T(S), \texttt{mapLeft}_T(S, i), \texttt{mapLeft}_T(S, j)) + \texttt{range}_{[x,y]}(\texttt{Right}_T(S), \texttt{mapRight}_T(S, i), \texttt{mapRight}_T(S, j)).$$

Note that if `range` is called on a leaf node S, then $\texttt{l}_T(S) = \texttt{r}_T(S) = z$, so the interval is either completely contained (if $z \in [x, y]$) or completely outside (if $z \notin [x, y]$). Both cases are already considered.

It is not difficult to show that for a range counting query, we have to make at most $O(\log \sigma)$ recursive calls (Gagie *et al.* (2012) show detailed proof), and thus the time complexity is, as for rank and quantile, $O(M \times \log \sigma)$ where M is the time needed to compute the map functions.

3. Simple Update Queries

We now discuss some simple update queries over wavelet trees. The idea is to shed light on the versatility of the structure to support less classical operations. We looked for inspiration in typical operations found in competitive programming problems to design update queries that preserve the global structure of the wavelet tree. We only describe the high level idea on how these queries can be adopted by the wavelet tree, and we later (in Section 4) discuss on how to efficiently implement them.

3.1. *Swapping Contiguous Positions*

Consider Query 2 in the introduction denoted by $\texttt{swap}(S,\, i)$. That is, a call to $\texttt{swap}(S,\, i)$ changes $S = (s_1,\, \ldots,\, s_n)$ into a sequence $(s_1,\, \ldots,\, s_{i+1},\, s_i,\, \ldots,\, s_n)$.

The operation $\texttt{swap}(S,\, i)$ can be easily supported by the wavelet tree as follows. Assume first that $s_i \leq \texttt{m}_T(S)$. Then we have two cases depending on the value of s_{i+1}. If $s_{i+1} > \texttt{m}_T(S)$, we know that s_i is mapped to the left subtree while s_{i+1} is mapped to the right subtree. This means that swapping these two elements does not modify any of the nodes of the tree that are descendants of S. In order to modify S, besides actually swapping the elements, we should update $\texttt{mapLeft}_T(S,\, i)$ and $\texttt{mapRight}_T(S,\, i)$; $\texttt{mapLeft}_T(S,\, i)$ should be decremented by 1 and $\texttt{mapRight}_T(S,\, i)$ should be incremented by 1 as the new element in position i is now mapped to the right subtree. Notice that these are the only two updates that need to be done to the map functions.

The other case is if $s_{i+1} \leq \texttt{m}_T(S)$. Notice that both s_i and s_{i+1} are mapped to $\texttt{Left}_T(S)$, and moreover, they are mapped to contiguous positions in that sequence. In this case, no update should be done to $\texttt{mapLeft}_T(S,\, i)$ or $\texttt{mapRight}_T(S,\, i)$. Thus, besides actually swapping the elements in S, we should only recursively perform the operation $\texttt{swap}(\texttt{Left}_T(S),\, \texttt{mapLeft}_T(S,\, i))$. The case in which $s_i > \texttt{m}_T(S)$ is symmetrical. The complete process is repeated until a leaf node is reached, in which case nothing should be done.

To perform the swap in the worst case we would need to traverse from top to bottom of the wavelet tree. Moreover, notice that the map functions $\texttt{mapLeft}_T$ and $\texttt{mapRight}_T$ are updated in at most one node. Thus the complexity of the process is $O(M \times \log \sigma + K)$ where K is the time needed to update $\texttt{mapLeft}_T$ and $\texttt{mapRight}_T$, and M is the time needed to compute the map functions.

3.2. *Toggling Elements*

Assume that every element in a sequence S has two possible states, *active* or *inactive*, and that an operation $\texttt{toggle}(S, i)$ is used to change the state of element i from *active* to *inactive*, or from *inactive* to *active* depending on its current state. Given this setting, we want to support all the queries mentioned in Section 2, but only considering active elements. For example, assume that $S = (1, 2, 1, 3, 1, 4)$ and only the non 1 elements are active. Then a query $\texttt{quantile}_2(S, 1, 6)$ would be 3.

A simple augmentation of the wavelet tree can be used to support this update. Besides $\texttt{mapLeft}_T$ and $\texttt{mapRight}_T$, we use two new mapping/counting functions $\texttt{activeLeft}_T$ and $\texttt{activeRight}_T$. For a node S and an index i, $\texttt{activeLeft}_T(S, i)$ is the number of active elements until index i that are mapped to the left child of S, and similarly $\texttt{activeRight}_T(S, i)$ is the number of active elements mapped to the right child. Besides this we can also have a count function for the leaves of the tree, $\texttt{activeLeaf}_T(S, i)$, that counts the number of active elements in a leaf S until position i. We next show how these new mapping functions should be updated when a toggle operation is performed. Then we describe how the queries in Section 2 should be adapted.

Upon an update operation $\texttt{toggle}(S, i)$ we proceed as follows. If $s_i \leq \texttt{m}_T(S)$ then we should update the values of $\texttt{activeLeft}_T(S, j)$ for all $j \geq i$ adding 1 to $\texttt{activeLeft}_T(S, j)$ if s_i was previously inactive, or substracting 1 in case s_i was previously active. Now, given that s_i is mapped to the left child of S, we proceed recursively with $\texttt{toggle}(\texttt{Left}_T(S), \texttt{mapLeft}_T(S, i))$. If $s_i > \texttt{m}_T(S)$, we proceed symmetrically updating $\texttt{activeRight}_T(S, j)$ for $j \geq i$, and recursively calling $\texttt{toggle}(\texttt{Right}_T(S), \texttt{mapRight}_T(S, i))$. We repeat the process until a leaf is reached, in which case $\texttt{activeLeaf}_T$ should also be updated (similarly as for $\texttt{activeLeft}_T$). The complexity of the toggle operation is then $O((A + M) \times \log \sigma)$, where A is the time needed to update $\texttt{activeLeft}_T$ and $\texttt{activeRight}_T$ in every level (plus $\texttt{activeLeaf}_T$ in the last level), and M is the time needed to compute the map functions $\texttt{mapLeft}_T$ and $\texttt{mapRight}_T$.

Consider now the $\texttt{quantile}_k(S, i, j)$ query. Recall that for this query we first computed a value c representing the number of elements of S from index i to index j that are mapped to the left. If $k \leq c$ we proceeded searching for $\texttt{quantile}_k$ in the left subtree, and if $k \geq c$ we proceeded searching for $\texttt{quantile}_{(k-c)}$ in the right subtree (mapping indexes i and j accordingly in both cases). In order to consider the active/inactive state of each element, we only need to change how c is computed; we need to consider now how many active elements from index i to index j are mapped to the left, and thus c is computed as

$$c = \texttt{activeLeft}_T(S, i-1) - \texttt{activeLeft}_T(S, j).$$

Then, we proceed exactly as before: if $k \leq c$ we search for $\texttt{quantile}_k$ in the left subtree, otherwise, we search for $\texttt{quantile}_{(k-c)}$ in the right subtree. Notice that we always assume that when executing $\texttt{quantile}_k(S, i, j)$ the number of active elements

between i and j in S is not less than k (which can be easily checked using $\texttt{activeLeft}_T$ and $\texttt{activeRight}_T$).

Queries $\texttt{rank}_q$ and $\texttt{range}_{[x,y]}$ are even simpler. In the case of $\texttt{rank}_q$ we only need to consider the active elements when we reach a leaf; in the last query $\texttt{rank}_q(S,\, i)$ in a leaf S, we just answer $\texttt{activeLeaf}_T(S,\, i)$. In the case of $\texttt{range}_{[x,y]}(S,\, i,\, j)$, we almost keep the recursive strategy as before but now when $[\texttt{l}_T(S),\, \texttt{r}_T(S)]$ is totally contained in $[x,\, y]$ we only have to consider the number of active elements between index i and index j, which is computed as

$$(\texttt{activeLeft}_T(S,\, j) + \texttt{activeRight}_T(S,\, j)) - (\texttt{activeLeft}_T(S,\, i-1) + \texttt{activeRight}_T(S,\, i-1)).$$

In the case in which S is a leaf, this value is computed as $\texttt{activeLeaf}_T(S,\, j) - \texttt{activeLeaf}_T(S,\, i-1)$.

The complexity of these new queries is $O((L + M) \times \log\sigma)$ where L is the time needed to compute the $\texttt{activeLeft}_T$ and $\texttt{activeRight}_T$ functions and M is the time needed to compute the map functions.

3.3. *Adding and Deleting Elements from the Beginning or End of the Sequence*

Consider the operations $\texttt{pushBack}(S,\, a)$, $\texttt{popBack}(S)$, $\texttt{pushFront}(S,\, a)$ and $\texttt{popFront}(S)$, with their typical meaning of adding/deleting elements to/from the beginning or ending of sequence S.

First notice that when adding or deleting elements we might be changing the alphabet of the tree. To cop with this, we assume that the underlying alphabet Σ is fixed and that the tree is constructed initially from a sequence mentioning all values in Σ. Thus, initially there is a leaf in the tree for every possible value. We also assume that in every moment there is an *active alphabet*, which is a subset of Σ, containing the values actually mentioned in the tree. To support this we just allow some sequences in the tree to be empty; if there is some value k of Σ not present in the tree at some point, then the sequence corresponding to the leaf node associated with k is the empty sequence. It is straightforward to adapt all the previous queries to this new setting.

Consider now $\texttt{pushBack}(S,\, a)$ and assume that before the update we have $|S| = n$. Then, besides adding a to the end of sequence S, we should update (or more precisely, create) $\texttt{mapLeft}_T(S,\, n{+}1)$ and $\texttt{mapRight}_T(S,\, n{+}1)$. If $a \leq \texttt{m}_T(S)$ then we let $\texttt{mapLeft}_T(S,\, n+1) = \texttt{mapLeft}_T(S,\, n) + 1$ and $\texttt{mapRight}_T(S,\, n+1) = \texttt{mapRight}_T(S,\, n)$, and then perform $\texttt{pushBack}(\texttt{Left}_T(S),\, a)$. If $a > \texttt{m}_T(S)$ then we let $\texttt{mapLeft}_T(S,\, n+1) = \texttt{mapLeft}_T(S,\, n)$ and $\texttt{mapRight}_T(S,\, n+1) = \texttt{mapRight}_T(S,\, n){+}1$, and then perform $\texttt{pushBack}(\texttt{Right}_T(S),\, a)$. Finally when we reach a leaf node, we just add a to the corresponding sequence.

The $\texttt{popBack}(S)$ operation is similar. Assume that $|S| = n$, then besides deleting the last element in S, we should only delete that element from the corresponding sub-

tree. Thus, if $s_n \leq \mathtt{m}_T(S)$ then we do $\mathtt{popBack}(\mathtt{Left}_T(S))$, and if $s_n > \mathtt{m}_T(S)$ we do $\mathtt{popBack}(\mathtt{Right}_T(S))$. When we reach a leaf node we just delete any element from it. Notice that in this case no $\mathtt{mapLeft}_T$ or $\mathtt{mapRight}_T$ needed to be updated.

The `pushFront` and `popFront` are a bit more complicated. When we do $\mathtt{pushFront}(S,\, a)$ we should do a complete remapping: if $a \leq \mathtt{m}_T(S)$ then for every $i \in \{1,\, \ldots,\, n\}$ we should do

$$\mathtt{mapLeft}_T(S,\, i+1) = \mathtt{mapLeft}_T(S,\, i) + 1$$
$$\mathtt{mapRight}_T(S,\, i+1) = \mathtt{mapRight}_T(S,\, i)$$

and finally set $\mathtt{mapLeft}_T(S,\, 1) = 1$ and $\mathtt{mapRight}_T(S,\, 1) = 0$ and perform the call $\mathtt{pushFront}(\mathtt{Left}_T(S),\, a)$. If $a > \mathtt{m}_T(S)$ then we should do

$$\mathtt{mapLeft}_T(S,\, i+1) = \mathtt{mapLeft}_T(S,\, i)$$
$$\mathtt{mapRight}_T(S,\, i+1) = \mathtt{mapRight}_T(S,\, i) + 1$$

and finally set $\mathtt{mapLeft}_T(S,\, 1) = 0$ and $\mathtt{mapRight}_T(S,\, 1) = 1$ and perform the call $\mathtt{pushFront}(\mathtt{Right}_T(S),\, a)$. When a leaf node is reached we just add a at the beginning of the corresponding sequence. The $\mathtt{popFront}(S)$ operation is similar. Let $|S| = n$. If $s_1 \leq \mathtt{m}_T(S)$ then we should update $\mathtt{mapLeft}_T\ (S,\, i)$ to $\mathtt{mapLeft}_T\ (S,\, i+1) - 1$, and $\mathtt{mapRight}_T(S,\, i)$ to $\mathtt{mapRight}_T(S,\, i+1)$ for all i from 1 to $n-1$, and then do $\mathtt{popFront}(\mathtt{Left}_T(S))$. Symmetrically if $s_1 > \mathtt{m}_T(S)$ then we should update $\mathtt{mapLeft}_T(S,\, i)$ to $\mathtt{mapLeft}_T(S,\, i+1)$, and $\mathtt{mapRight}_T(S,\, i)$ to $\mathtt{mapRight}_T(S,\, i+1) - 1$ for all i from 1 to $n-1$, and then do $\mathtt{popFront}(\mathtt{Right}_T(S))$. Upon reaching a leaf node, we just delete the value from the front.

The complexity of all the operations above is $O((A+M) \times \log\sigma)$ where A is the time needed to update `mapLeft` or `mapRight` in every level, and M is the time needed to compute the map functions. Just notice that for the cases of the `pushFront` and `popFront` we have to update several values of `mapLeft` and `mapRight` per level.

4. Implementation

In this section we explain how to build a wavelet tree and how to construct the auxiliary structures to support the mapping operations efficiently. Based on this construction we also discuss how to implement queries explained in the previous section. Additionally, we present an implementation strategy alternative to the direct pointer based one. We implemented both approaches in `C++` and the code is available in github[1].

[1] `https://github.com/nilehmann/wavelet-tree`

4.1. *Construction*

A wavelet tree implementation represents an array $A[0,\ n-1]$ of integers in the interval $[0,\ \sigma-1]$. Our construction is based on a node structure and pointers between those nodes. Every node v will be identified by two elements ℓ_v and r_v (which essentially correspond to $\mathtt{l}_T(v)$ and $\mathtt{r}_T(v)$ in Section 2), and an associated sequence A_v which is the subsequence of A formed by selecting elements in the range $[\ell_v,\ r_v]$. As we will see, values ℓ_v and r_v and the sequence A_v do not need to be explicitly stored and can be computed when traversing the tree if needed.

The construction of a wavelet tree starts creating the root node associated to the original array A and the interval $[0,\ \sigma-1]$. We then proceed recursively as follows. In each node v we found the middle of the interval $m_v = (r_v+\ell_v)/2$ (which corresponds to the value $\mathtt{m}_T(v)$ described in Section 2). We create two new nodes u and w as left-child and right-child of v, respectively. Then, we perform a stable partition of the array A_v into two arrays A_u and A_w, such that A_u contains all values less than or equal to m_v and A_w those greater than m_v. The construction continues recursively for the left node with the array A_u and the interval $[\ell_v,m_v]$, and for the right node with A_w and the interval $[m_v+1,\ r_v]$. The base case is reached when the interval represented by the node contains only one element, i.e., $r_v = \ell_v$. It is not necessary to store arrays A_v corresponding to each node v. They are only materialized at building time to construct the auxiliary structures to support the mapping operations required to traverse the tree as described below.

As previously discussed, the fundamental operations `mapLeft` and `mapRight` correspond to count how many symbols until position i belong to the left and right node respectively. To support these operations, when building a node v we precompute for every position i how many elements in the array A_v belong to the right node – they are greater than m_v – and store the results in an array C_v. We could store a similar array C'_v to store how many elements belong to the left node, but it is easy to note that values of both arrays are related as follows: $C'_v[i] = i - C_v[i] + 1$.

To understand how C_v is computed, it turns out useful to associate a bitvector B_v that marks with 0's elements less than or equal to m_v and with 1's those greater than m_v. This bitvector must support the operation of counting how many bits are set to 1 until a position i, which is commonly referred as a rank operation. Our array C_v is computed on build time as the partial sum of B_v by the recurrence $C_v[0] = B_v[0]$, $C_v[i] = C_v[i-1] + B_v[i]$, thus supporting the rank operation in constant time. The compression characteristics of the wavelet tree arise mainly because it is possible to represent these bitvectors succinctly while maintaining constant-time rank queries (Clark, 1998; Okanohara and Sadakane, 2007; Raman *et al.*, 2002). However, in a competitive programming setting memory constraints are less restrictive and our representation shows off to be sufficient. In case the memory is an issue, a practical and succinct implementation is presented by González *et al.* (2005).

4.2. *Implementing Queries and Updates*

We now briey discuss how every operation in Section 2 can be efficiently implemented.

`mapLeft` and `mapRight`. These two operations can be easily implemented with the array C_v; in a node v the number of elements until position i that go to the left is $i - C_v[i] + 1$. Since we are indexing from 0, position i is mapped to the left to position $i - C_v[i]$. Analogously, a position i is mapped to the right to position $C_v[i] - 1$. Notice that both mapping functions can thus be computed in constant time, which implies that `rank`, `quantile` and `range` operations can be implemented in $O(\log \sigma)$ time.

`swap`. The swap operation first map the position i down the tree until we reach a node v where the update needs to be performed. At this point the (virtual) bitvector B_v is such that $B_v[i] \neq B_v[i+1]$. Swapping both bits can only change the count of 1's until position i, and thus, only $C_v[i]$ should be updated. If $B_v[i] = 0$ we do $C_v[i] = C_v[i] + 1$, and if $B_v[i] = 1$ we do $C_v[i] = C_v[i] - 1$. This shows that the map functions can be updated in constant time after a swap operation, which implies that the complexity of swap is also $O(\log \sigma)$.

`toggle`. In this case we only need to implement `activeLeft`, `activeRight` and `activeLeaf`. To mark which positions are active we can use any data structure representing sequences of 0's and 1's that efficiently supports partial sums and point updates. For example we can use a *binary indexed tree* (BIT) (Fenwick, 1994) which is a standard data structure used in competitive programming that supports both operations in $O(\log n)$ time. Thus with a BIT we are adding a logarithmic factor for each query and now `rank`, `quantile` and `range` operations as well as `toggle` can be implemented in $O(\log n \times \log \sigma)$. In terms of construction, when using a BIT in every level we are only paying a constant factor in the size of the wavelet tree.

`pushBack` and `popBack`. These operations only modify the array C_v in some nodes. Pushing an element at the end updates the (virtual) bitvector B_v appending a new 0 or 1 (depending on the comparison between the new element and m_v), so C_v being a partial sum of B_v of size n_v only needs a $C_v[n_v] = C_v[n_v - 1] + B_v[n_v - 1]$ update. Popping an element from the end updates B_v and C_v doing the inverse operation, so if C_v is of size n_v we only need to delete $C_v[n_v - 1]$ from memory. Both operations can be done in amortized constant time using a dynamic array, thus the complexity of all queries plus `pushBack` and `popBack` is $O(\log \sigma)$ time.

`pushFront` and `popFront`. These are similar to `pushBack` and `popBack`, but act at the beginning of the bitvector B_v. To prepend a bit b to a bitvector B_v we must prepend its value to C_v. If the value of b is equal to 1 we must also increment by 1 every value in C_v. Because it is too slow to update every position of C_v, we define a counter δ_v that starts at 0 and is incremented by 1 every time a bit equal to 1 is prepended. We then just prepend $b - \delta_v$ to C_v, in which case the real count of ones until position i is obtained by $C_v[i] + \delta_v$. Popping an element is as easy as deleting the first element of C_v from

memory and decrementing δ_v by 1 if the value of $C_v[0] + \delta_v$ was equal to 1. If we want to mix front and back operations, we could use a structure such as a dequeue (Knuth, 1997), which allows amortized constant time insertions at the beginning and end of an array while maintaining constant random access time. Thus the complexity of all queries is still $O(\log \sigma)$ time.

4.3. *Big Alphabets and the Wavelet Matrix*

In a competitive programming setting the size of the array A will depend on time restrictions, but typically it will not exceed 10^6. However the number of possible values that A can store could be without any problems around 10^9. Thus the number of values actually appearing in A is much smaller than the range of possible values. For this reason one usually have to map the values that appear in the sequence to a range $[0,\ \sigma - 1]$. Commonly, this will require a fairly fast operation to translate from one alphabet to the other with a typical implementation using, for example, a binary search tree or a sorted array combined with binary search.

To avoid having this map operation, the wavelet tree could be constructed directly over the range of all possible values allowing the subsequences of some nodes to be empty. A naive pointer-based construction will require $O(\sigma)$ words which might be excessive for $\sigma = 10^9$. Because many nodes will represent empty subsequences, one can save some space explicitly tracking when some subsequences become empty in the tree.

There is an alternative implementation of the wavelet tree called *wavelet matrix* (Claude *et al.*, 2015) that was specifically proposed in the literature to account for big alphabets. Given an alphabet its size can be extended to match the next power of two, yielding a complete binary tree for the wavelet tree representation. For each level, we could then concatenate the bitvectors of each node in that level and represent the structure with a single bitvector by level. The border between each node is lost, but it can be computed on the fly when traversing. This means extra queries yielding worse performance. Instead, the wavelet matrix breaks the restriction that in each level siblings must be represented in contiguous positions in the bitvector. When partitioning a node at some level ℓ the wavelet matrix sends all zeroes to the left section of level $\ell + 1$ and all ones to the right. The left and the right child of some node at level ℓ do not occupy contiguous positions in the bitvector at level $\ell + 1$, but the left (resp. right) child is represented in contiguous positions in the left (resp. right) section of the level $\ell + 1$. Additionally, a value z_ℓ is maintained at each level to mark how many elements were mapped to the left.

With this structure the traversing operations can be directly implemented by performing rank operations on bitvectors at each level. Specifically, instead of maintaining an array C_v for every node, we maintain an array C_ℓ for each level. Array C_ℓ store the cumulative number of 1's in level ℓ. Then, a position i at level ℓ is mapped to the left to position $i - C_\ell[i]$ at level $\ell + 1$. The same position i is mapped to the right to position $z_\ell + C_\ell[i] - 1$.

The wavelet matrix has the advantage of being implementable using only $O(\log \sigma)$ extra words of memory instead of the $O(\sigma)$ used to store the tree structure in the pointer based alternative while maintaining fast operations. This $O(\log \sigma)$ words are insignificant even for $\sigma = 10^9$, which means that the structure could be constructed directly over the original alphabet. On the other hand the wavelet matrix is somehow less adaptable, because it does not support directly the `pop` and `push` updates. However, it can support `swap` and `toggle` in a similar way as the one described for the wavelet tree.

5. Wavelet Trees in Current Competitive Programming

We conducted a *social experiment* uploading 3 different problems to the *Sphere Online Judge* (SPOJ)[2]. All the problems can be solved with the techniques shown in the previous sections. We analyze the solutions to these problems submitted by SPOJ users. Our analysis reveal two main conclusions: (1) experienced programmers do not consider the use of wavelet trees, even in the case that its application is straightforward, and (2) for the most complex cases when they succeed, they use fairly involved techniques producing solutions that are dangerously close to time and memory limits. We have found, however, some incipient references of wavelet trees in the competitive programming community[3], as well as more detailed explanations in Japanese[4], which obviously establish an idiomatic barrier for many programmers.

We identify the three mentioned problems as ILKQ1, ILKQ2 and ILKQ3 and they are described as follows.

ILKQ1 considers a slightly modified version of the quantile query. The size of the initial sequences is 10^5, the range of possible integer values in the sequence is $[-10^9, 10^9]$, and the number of queries is 10^5. The time limit is 1s.

Link: `http://www.spoj.com/problems/ILKQUERY`

ILKQ2 considers rank queries plus toggling the state of arbitrary elements. The size of the initial sequence is 10^5, the range of possible values is $[-10^9, 10^9]$, and the number of rank plus toggle queries is 10^5. The time limit is 0.4s.

Link: `http://www.spoj.com/problems/ILKQUERY2`

ILKQ3 considers the quantile query of ILKQ1 plus swaps of arbitrary contiguous positions. The size of the initial sequences is 10^6, the range of possible values is $[-10^9, 10^9]$, and the number of quantile plus swap queries is 10^5. The time limit is 1s.

Link: `http://www.spoj.com/problems/ILKQUERYIII/`

Notice that ILKQ3 although involves the same query as ILKQ1, it is considerable harder as it can mix updates (in the form of swaps) and the input sequence can be 10 times bigger than for ILKQ1. Table 1 shows an analysis of the submissions received[5].

[2] `http://www.spoj.com/`

[3] `http://codeforces.com/blog/entry/17787`

[4] `http://d.hatena.ne.jp/sune2/20131216/1387197255`

[5] This data considers only until late March 2016.

Table 1
General submission statistics

	Submitted	Accepted	Non-accepted		
			WA	TLE	RTE
ILKQ1	49	9	19	18	3
ILKQ2	32	6	15	8	3
ILKQ3	35	2	12	15	6

5.1. *Analysis of Users Submitting Solutions*

We received submissions from several type of users, several of them can be considered as experienced programmers. From them, even expert coders (rank 100 or better on SPOJ) got lot of Wrong Answers (WA) or Time Limit Exceeded (TLE) verdicts which shows the intrinsic difficulty of the problems. Considering the three problems, 5 out of the 10 distinct users who got an Accepted (AC) veredict have rank of 60 or better on SPOJ, and 8 are well-known ACM-ICPC World finalists. For problem ILKQ3 we received only two AC. Both users solved the problem after several WA or TLE verdicts. For ILKQ1 and ILKQ2 the best ranked submitter was the top 1 user in SPOJ who obtained AC in both problems. For ILKQ3, the best ranked submitter was among the top 5 in SPOJ and obtained only TLE veredicts.

5.2. *Analysis of the Submitted Solutions*

As we have told before, we received 0 submissions implementing a wavelet tree solution. We now briefly analyze the strategies of the submissions received. For the sake of the space, we cannot deeply analyze every strategy but we provide some pointers for the interested reader.

The most common approach for ILKQ1 was sorting queries (as the problem is offline) plus the use of a tree data structure. One of the mainly used in this case was *mergesort tree*. In a mergesort tree, nodes represent sequences of contiguous elements of the original array and contains a sorted version of those sequences. Leaves represent one element of the array, and the tree is built recursively by merging pairs of nodes starting from the leaves. The construction can be done in $O(n \log n)$ time and space. Quantile queries can be answered by identifying the, at most $O(\log n)$, nodes that define a query range, and then doing two (nested) binary searches, one for counting elements less than or equal than a value X, and the second over X to find the k-th minimum element. The total strategy gives $O(\log^3 n)$ time which can be optimized up to $O(\log^2 n)$ using *fractional cascading*. This was enough given the time constraints.

For ILKQ2 and ILKQ3 sorting of queries or any offline approach is not directly useful as queries are mixed with updates. For ILKQ2 we received some submissions implementing a *square root decomposition* strategy, and run extremely close to the time limit. The most successful strategy in both problems was the use of ideas coming from *persistent data structures*, in particular *persistent segment trees*[6]. As in any persistent structure, the main idea is to efficiently store different states of it. Exploiting the fact that consecutive states do not differ in more than $O(\log n)$ nodes, it is possible to keep n different segment trees in $O(n \log n)$ space. Persistent segment trees can be used to answer quantile queries but need some more work to adapt them for updates like swaps as in ILKQ3. The two correct solutions that we received for ILKQ3 make use of this structure. It's relevant to notice that given the input size and the updates, implementing a persistent segment tree for this problem can use a considerable amount of memory. In particular, one of the AC submissions used 500MB and the other 980MB. Our wavelet tree solution uses only 4MB of memory.

6. Performance Tests

Existing experimental analyses about wavelet trees focus mostly on compression characteristics (Claude *et al.*, 2015). Moreover, they do not consider the time required to build the structure because from the compression point of view the preprocessing time is not the most relevant parameter. Thus, we conducted a series of experiments focusing on a competitive-programming setting where the building time is important and restrictions on the input are driven by typical tight time constraints. The idea is to shed some light on how far the input size can be pushed. We expect these results to be useful for competitors as well as for problem setters.

We performed experimental tests for our wavelet tree and wavelet matrix implementations comparing construction time and the performance of rank, quantile and range counting queries. We consider only alphabets of size less than the size of the sequence. To analyze the impact of the alphabet size, we performed tests over sequences of different *profiles*. A profile is characterized by the ratio between the size of the alphabet and the size of the sequence. For example, a sequence of size 10^3 and profile 0.5 has an alphabet of size 500.

Measurements. To measure construction time we generated random sequences of increasing size for different profiles. For each size and profile we generated 1,000 sequences and we report the average time. For queries rank, quantile and range counting, we generated 100,000 queries uniformly distributed and averaged their execution time. The machine used is an Intel® Core™ i7-2600K running at 3.40GHz with 8GB of RAM memory. The operating system is Arch-Linux running kernel 4.4.4. All our code are single-threaded and implemented in `C++`. The compiler used is `gcc` version 5.3.0, with optimization flag `-O2` as customary in many programming contests.

[6] `bit.ly/persistent-segment-tree`

Results. No much variance was found in the performance between different profiles, but as may be expected sequences of profile 1 – i.e., permutations – reported higher time in construction and queries. Thus, we focus on the analysis of permutations to test performance on the most stressing setting. For the range of input tested we did not observe big differences between the wavelet tree and the wavelet matrix, both for construction and query time. Though there are little differences, they can be attributed to tiny implementation decision and not to the implementation strategy itself.

Regarding the size of the input (Fig. 2), construction time stays within the order of 250 milliseconds for sequences of size less than or equal to 10^6, but scales up to 2 seconds for sequences of size 10^7, which can be prohibitive for typical time constraints. For the case of queries rank, quantile and range counting we report in Fig. 3 the number of queries that can be performed in 1 second for different sizes of the input sequence. For rank and quantile, around 10^6 queries can be performed in 1 second for an input of size 10^6. In contrast for range counting, only 10^5 queries can be performed in the same setting (Fig. 3).

It would be interesting as future work to perform a deep comparison between the wavelet tree and competing structures for similar purposes such as mergesort trees and persistent segment trees, testing time and memory usage. From our simple analysis in the previous section one can infer that wavelet trees at least scales better in terms of memory usage, but more experimentation should be done to draw stronger conclusions.

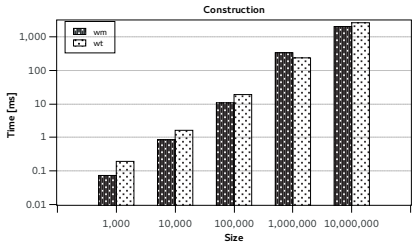


Fig. 2. Construction time in milliseconds.

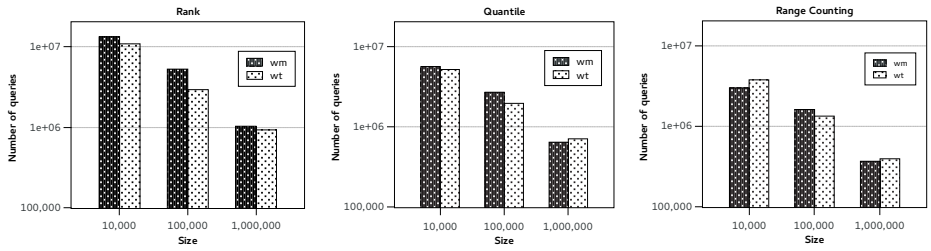


Fig. 3. Number of queries that can be performed in one second.

7. Concluding Remarks

Problems involving advanced data structures are appearing increasingly often in world-wide programming competitions. In this scenario, competitive programmers often prefer versatile structures that can be used for a wide range of problems without making a lot of changes. Structures such as binary indexed trees or (persistent) segment trees, to name a few, conform part of the lower bound for competitors, and must be in the toolbox of any programmer. The wavelet tree has proven to be a really versatile structure but, as we have evidenced, not widely used at the moment. However, we have noted that some programmers have already perceived the virtues of the wavelet tree. We believe that the wavelet tree, being quite easy to implement, and having such amount of applications, is probably becoming a structure that every competitive programmer should learn. With this paper we try to fill the gap and make wavelet trees widely available for the competitive programming community.

Acknowledgments

J. Pérez is supported by the Millennium Nucleus Center for Semantic Web Research, Grant NC120004, and Fondecyt grant 1140790.

References

Clark, D. (1998). *Compact Pat Trees*. PhD thesis, University of Waterloo.

Claude, F., Navarro, G., Ordóñez, A. (2015). The wavelet matrix: an efficient wavelet tree for large alphabets. *Inf. Syst.*, 47, 15–32.

Fenwick, P. M. (1994). A new data structure for cumulative frequency tables. *Software: Practice and Experience*, 24(3), 327–336.

Gagie, T., Navarro, G., and Puglisi, S.J. (2012). New algorithms on wavelet trees and applications to information retrieval. *Theor. Comput. Sci.*, 426, 25–41.

Gagie, T., Puglisi, S.J., Turpin, A. (2009). Range quantile queries: another virtue of wavelet trees. In: *SPIRE*. 1–6.

González, R., Grabowski, S., Mäkinen, V., Navarro, G. (2005). Practical implementation of rank and select queries. In: *WEA*. 27–38.

Grossi, R. (2015). Wavelet trees. In: *Encyclopedia of Algorithms*. Springer.

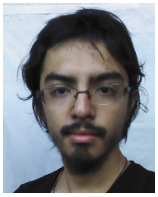
Grossi, R., Gupta, A., Vitter, J.S. (2003). High-order entropy-compressed text indexes. In: *SODA'03*. 841–850.

Knuth, D. (1997). *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. Third Edition. Addison-Wesley. 238–243.

Navarro, G. (2014). Wavelet trees for all. *J. Discrete Algorithms*, 25, 2–20.

Okanohara, D., Sadakane, K. (2007). Practical entropy-compressed rank/select dictionary. In: *ALENEX*. 60–70.

Raman, R., Raman, V., Rao, S. (2002). Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In: *SODA*. 233–242.



R. Castro is a computer engineering student at Department of Computer Science, Universidad de Chile, bronze medalist of IOI'2013 in Brisbane, and problem setter for OCI (Chilean Olympiad in Informatics).

N. Lehmann is master's student in computer science at Department of Computer Science, Universidad de Chile. His research interests are semantics, design and implementation of programming languages and type systems. He is the Scientific Committee Director of OCI (Chilean Olympiad in Informatics). Deputy Leader at IOI 2014 and 2015. Chilean Judge of ACM ICPC 2014 and 2015.

J. Pérez is Associate Professor at the Department of Computer Science, Universidad de Chile, and an Associate Researcher of the Chilean Center for Semantic Web Research. His research interests are database theory, data exchange and integration, graph databases, and the application of database technologies to the Semantic Web and the Web of Data. He is one of the directors of OCI (Chilean Olympiad in Informatics), and the Chilean team leader at IOI since 2013.

B. Subercaseaux is an undergraduate student of computer science at Department of Computer Science, Universidad de Chile, and problem setter for OCI (Chilean Olympiad in Informatics).