Contents lists available at ScienceDirect

Journal of Discrete Algorithms

www.elsevier.com/locate/jda

Improved Range Minimum Queries ☆

Héctor Ferrada, Gonzalo Navarro *

Center of Biotechnology and Bioengineering, Department of Computer Science, University of Chile, Santiago, Chile

ARTICLE INFO

Article history:
Available online 17 October 2016

Keywords:
Range Minimum Queries
Lowest common ancestors
Cartesian trees
Balanced Parentheses
Compact data structures

ABSTRACT

Fischer and Heun [SICOMP 2011] proposed the first Range Minimum Query (RMQ) data structure on an array $A[1,n]$ that uses $2n+o(n)$ bits and answers queries in $O(1)$ time without accessing A. Their scheme converts the Cartesian tree of A into a general tree, which is represented using DFUDS. We show that, by using instead the BP representation, the formula becomes simpler since border conditions are eliminated. We also improve the current implementation of the BP representation for this purpose. This leads to the fastest and most compact practical implementation to date.

1. Introduction

The Range Minimum Query (RMQ) problem is, given an array $A[1,n]$ with elements from a totally ordered set, build a data structure that receives any pair of positions $1 \le i \le j \le n$ and returns

$$\mathsf{rmq}_A(i,j) = \mathrm{argmin}_{i \le k \le j} A[k],$$

that is, the position of a minimum value in $A[i,j]$. In many cases one prefers the leftmost position when there are ties.

The RMQ problem is a fundamental one and has a long history, intimately related to another key problem: the LCA (lowest common ancestor) problem on general ordinal trees is, given nodes u and v, return $\mathsf{lca}(u,v)$, the lowest node that is an ancestor of both u and v. Gabow et al. [10] showed that RMQs can be reduced to computing LCAs on a particular tree, called the *Cartesian tree* [22] of $A[1,n]$. Later, Berkman and Vishkin [4] showed that the LCA problem on any tree can be reduced to an RMQ problem, on an array derived from the tree. In this array, consecutive entries differ by ± 1. Bender and Farach [2] then gave a solution for this so-called ± 1-RMQ problem in constant time and linear space (i.e., $O(n)$ words). Sadakane [20] improved the space of that solution, showing that LCAs on a tree of n nodes can be handled in constant time using $2n+o(n)$ bits (including the tree representation [17]). Finally, Fischer and Heun [8] showed that the Cartesian tree can be represented using $2n+o(n)$ bits so that RMQs on A can be transformed into LCA queries on the succinct tree, and this leads to an RMQ solution that also uses $2n+o(n)$ bits and *does not need to access A at query time*.

Fischer and Heun's solution has become a fundamental building block for many succinct data structures, for example for ordinal trees [20,15,19], suffix trees [20,9], document retrieval [21,16], two-dimensional grids [18], Lempel–Ziv parsing [5], etc.

Their RMQ computation [8] uses three kinds of operations: several rank/selects on bitvectors [14,6], one ± 1-RMQ [2], and one open on parentheses [17]. Although all can be implemented in constant time, in practice the last two operations are significantly slower than rank/select [1]. In particular, open is needed just to cover a border case where one node is an

☆ Funded with basal funds FB0001, CONICYT, Chile. An early version of this article appeared in *Proc. Data Compression Conference 2016*.
* Corresponding author.
E-mail addresses: hferrada@dcc.uchile.cl (H. Ferrada), gnavarro@dcc.uchile.cl (G. Navarro).

http://dx.doi.org/10.1016/j.jda.2016.09.002
1570-8667/© 2016 Elsevier B.V. All rights reserved.

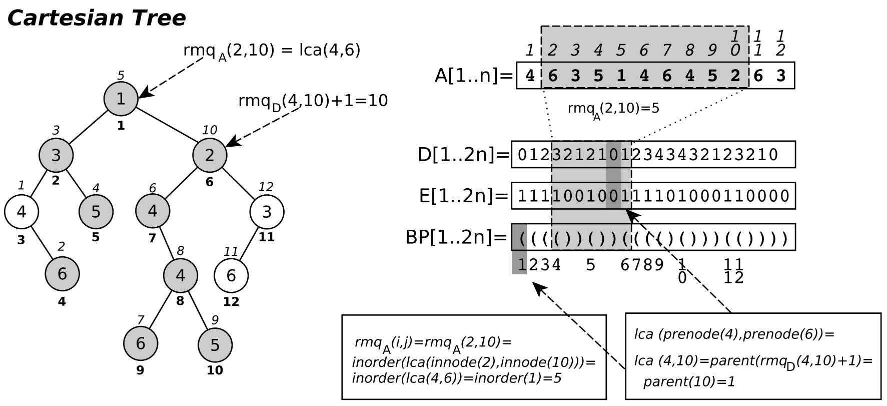
Cartesian Tree
rmq_A(2,10) = lca(4,6)
rmq_D(4,10)+1=10
A[1..n]= 4 6 3 5 1 4 6 4 5 2 6 3
rmq_A(2,10)=5
D[1..2n]= 012321210123434321232 10
E[1..2n]= 111100100111101000110000
BP[1..2n]=
rmq_A(i,j)=rmq_A(2,10)= inorder(lca(innode(2),innode(10)))= inorder(lca(4,6))=inorder(1)=5
lca (prenode(4),prenode(6))= lca (4,10)=parent(rmq_D(4,10)+1)= parent(10)=1

Fig. 1. An example array $A[1,12]$ (top right) and its Cartesian tree (left). We choose preorder numbers as node identifiers (in bold under the nodes), and also write inorder values on top of the nodes, in slanted font. The left rectangle on the bottom shows how query $\mathrm{rmq}_A(2,10)$ translates into query $\mathrm{lca}(4,6)$ on the Cartesian tree. We also show how this query, in turn, maps into $\mathrm{rmq}_D(4,10)$, on the array D of depths of the tree. Array E tells if consecutive entries of D increase or decrease, and is the same as a BP representation of the tree. The right rectangle on the bottom shows how query $\mathrm{lca}(4,10)$ is solved using $\mathrm{rmq}_D(4,10)$ and parent on the parentheses. This rmq_D query is a simpler ± 1-RMQ problem. Now the nodes 4, 10, and 1 do not refer to preorders but to positions in BP, obtained from preorders with prenode. The corresponding preorder values are written below the BP array.

ancestor of the other in the Cartesian tree. Grossi and Ottaviano [13] replaced open by further rank/selects in this case, thus improving the time significantly.

Their formula [8,13] represents the Cartesian tree using DFUDS [3]. In this paper we show that, if we use instead the BP representation for the tree [17], the RMQ formula can be considerably simplified because the border case does not need special treatment. In addition, we improve the current implementations of the BP representation, tailoring them to solve RMQs. The result is the fastest and most compact RMQ implementation so far: our structure uses $2.1n$ bits of space and answers RMQs in 1–3 microseconds. Current implementations in Simon Gog's *SDSL* [12] (https://github.com/simongog/sdsl-lite) and Giuseppe Ottaviano's *Succinct* [13] (https://github.com/ot/succinct) use from $2.6n$ to $2.8n$ bits. Our implementation is also 3–6 times faster than that in *SDSL* and twice as fast as the implementation in *Succinct*. It is also 2–4 times faster than our own implementation of Fischer and Heun's RMQ, while using less space.

2. State of the art

Gabow et al. [10] showed that RMQs can be reduced to computing LCAs on a particular tree, called the *Cartesian tree* [22] of $A[1,n]$. This is a binary tree whose root is the position p of a minimum in $A[1,n]$ (the leftmost/rightmost one if we want that RMQs return the leftmost/rightmost minimum). Then its left and right children are the Cartesian trees of $A[1,p-1]$ and $A[p+1,n]$, respectively. Any cell $A[p]$ is thus represented by the Cartesian tree node with inorder position p, and it holds

$$\mathrm{rmq}_A(i,j) = \mathrm{inorder}(\mathrm{lca}(\mathrm{innode}(i),\mathrm{innode}(j))), \quad (1)$$

where inorder and innode map from nodes to their inorder values and vice versa. Fig. 1 shows an example array A and its Cartesian tree, and the translation of a query (ignore the other elements for now).

Later, Berkman and Vishkin [4] showed that the LCA problem on any tree can be reduced to an RMQ problem, on an array $D[1,2n]$ containing the depths of the nodes traversed along an Eulerian tour on the tree: the LCA corresponds to the minimum in D between a cell of u and a cell of v in the array. Note that consecutive cells in D differ by ± 1. Bender and Farach [2] represented those entries as a bitvector $E[1,2n]$: $E[i]=1$ if $D[i]-D[i-1]=+1$ and $E[i]=0$ if $D[i]-D[i-1]=-1$, with $E[1]=1$. On top of E, they gave a simple $O(1)$-time solution to this restricted ± 1-RMQ problem using $O(n)$ words of space. Fig. 1 also shows this arrangement.

Therefore, one can convert an RMQ problem on A into an LCA problem on the Cartesian tree of A, then convert this problem into a ± 1-RMQ problem on the depths of the Eulerian tour of the Cartesian tree, and finally solve this restricted ± 1-RMQ problem in constant time. This solution requires $O(n)$ words of space.

Interestingly, the bitvector $E[1,2n]$ used to answer LCA queries on a tree of n nodes defines the topology of the tree. If we traverse the tree in DFS order and write an opening parenthesis when we first arrive at a node and a closing one when we leave it, the resulting sequence of parentheses, $P[1,2n]$, is exactly $E[1,2n]$ if we interpret the opening parenthesis as a 1 and the closing one as a 0. In particular, consider the following two operations on bitvectors: $\mathrm{rank}_b(E,i)$ is the number of bits equal to b in $E[1,i]$, and $\mathrm{select}_b(E,j)$ is the position of the jth bit b in E. Both operations can be implemented in $O(1)$ time using just $o(n)$ additional bits on top of E [14,6]. Then, if we identify a node x with the position of its opening parenthesis in P (which is a 1 in E), then the preorder position of x is $\mathrm{preorder}(x)=\mathrm{rank}_1(E,x)$, the node with preorder i is $\mathrm{prenode}(i)=\mathrm{select}_1(E,i)$, x is a leaf iff $E[x+1]=0$, and the depth of x is $D[x]=\mathrm{rank}_1(E,x)-\mathrm{rank}_0(E,x)=2\cdot\mathrm{rank}_1(E,x)-x$.

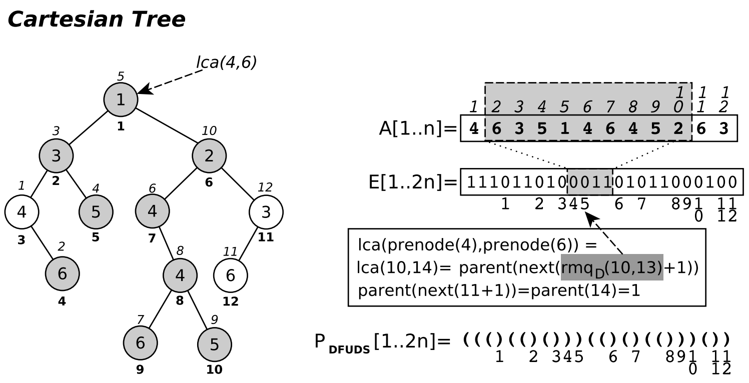
Cartesian Tree
lca(4,6)
A[1..n]= 4 6 3 5 1 4 6 4 5 2 6 3
E[1..2n]= 111011010001101011000100
lca(prenode(4),prenode(6)) =
lca(10,14)= parent(next(rmq_D(10,13)+1))
parent(next(11+1))=parent(14)=1
P_DFUDS[1..2n]= ((()(()()))(()()(()))())

Fig. 2. The same arrangement of Fig. 1, now on the DFUDS representation of the Cartesian tree. The query $\mathsf{rmq}_A(2, 10)$ becomes $\mathsf{lca}(4, 6)$, which we translate into $\mathsf{lca}(10, 14)$ when the node identifiers become positions in DFUDS instead of preorders (the translation is shown on the bottom of the sequence P_{DFUDS}).

This parentheses representation (called BP, for Balanced Parentheses) was indeed known, and it was even possible to navigate it in constant time by using just $2n + o(n)$ bits [17,11]. This navigation was built on top of three primitives on parentheses: $\mathsf{open}(x)/\mathsf{close}(x)$ gave the position of the opening/closing parenthesis matching the closing/opening one at $P[x]$, and $\mathsf{enclose}(x)$ gave the opening parenthesis position y so that $[y, \mathsf{close}(y)]$ contained $P[x]$ most tightly. Many tree traversal operations are built on top of those primitives, for example the parent of x is $\mathsf{parent}(x) = \mathsf{enclose}(x)$, its next sibling is $\mathsf{close}(x) + 1$ (if it exists), its first child is $x + 1$ (if it exists), its subtree size is $(\mathsf{close}(x) - x + 1)/2$, x is an ancestor of y iff $x \le y \le \mathsf{close}(x)$, etc.

Now, since E coincides with P, one could add the powerful lca operation to the BP representation! Bender and Farach's solution [2] applied on the bitvector $E[1, 2n]$ actually implements RMQs on the virtual array D. However, their ± 1-RMQ solution used $O(n)$ words. Sadakane [20] improved their solution to use $O(n(\log\log n)^2/\log n) = o(n)$ bits, and thus obtained a constant-time algorithm for $\mathsf{lca}(x, y)$ on the BP representation (let $x < y$):

$$\text{if } y \le \mathsf{close}(x) \text{ then return } x$$

$$\text{else return } \mathsf{parent}(\mathsf{rmq}_D(x, y) + 1)$$

where the first line addresses the special case where x is an ancestor of y, and rmq_D refers to the ± 1-RMQ solution using $E[1, 2n]$. The rationale of the second line is that, since x and y descend from two distinct children of $z = \mathsf{lca}(x, y)$, then $D[x, y]$ is minimized at the closing parenthesis that terminates each child of z, from the one that contains x to the one preceding that containing y. Adding 1 we get to the next sibling of that child, then we return its parent z. See Fig. 1 once again.

Benoit et al. [3] presented an alternative format to represent a general tree using $2n$ parentheses, called DFUDS. We traverse the tree in DFS order, but this time, upon arriving for the first time to a node with d children, we write d opening parentheses and a closing one (in particular, a leaf is represented with a closing parenthesis). Nodes are identified with that closing parenthesis.[1] It can be shown that the resulting sequence is also balanced if we append an artificial opening parenthesis at the beginning, and many traversal operations can be carried out with the primitives open, close, and enclose. In particular, we can directly arrive at the ith child of x with $\mathsf{next}_0((\mathsf{close}(x - i) + 1)$, where $\mathsf{next}_0(t) = \mathsf{select}_0(\mathsf{rank}_0(t - 1) + 1)$ finds the first 0 from t. The number of children of x can be computed as $d = x - \mathsf{prev}_0(x) + 1$, where $\mathsf{prev}_0(t) = \mathsf{select}_0(\mathsf{rank}_0(t - 1))$ finds the last 0 before t. In DFUDS, nodes are also listed in preorder, and there is a closing parenthesis terminating each, thus $\mathsf{preorder}(x) = \mathsf{rank}_0(E, x)$.

Jansson et al. [15] showed that $\mathsf{lca}(x, y)$ can also be computed on the DFUDS representation, as follows (let $x < y$):

$$\text{return } \mathsf{parent}(\mathsf{next}_0(\mathsf{rmq}_D(x, y - 1) + 1)),$$

where no check for ancestorship is needed.[2] The rationale is similar as before: since in DFUDS D decreases by 1 along each subtree area, $\mathsf{rmq}_D(x, y - 1)$ finds the final closing parenthesis of the child of $z = \mathsf{lca}(x, y)$ that precedes the one containing y. Adding 1 and finding the parent gives z. The formula for $\mathsf{parent}(w)$ in DFUDS is $\mathsf{next}_0(\mathsf{open}(\mathsf{prev}_0(w)))$. Fig. 2 shows our example, now on DFUDS. The formula with DFUDS turns out to be simpler than with BP.

Now we could represent a tree of n nodes in $2n + o(n)$ bits and compute lca on it in constant time, and Eq. (1) allowed us to convert rmq_A into an lca operation on its Cartesian tree. It seems that the road to constant-time rmq_A using just the $2n + o(n)$ bits of its Cartesian tree, and without accessing A, was paved! However, there was still a problem: how to support the operations inorder and innode on the Cartesian tree. Sadakane [20] had solved the problem on suffix trees, but

[1] In some cases, the first opening parenthesis is used, but the closing one is more convenient here.
[2] The check is present in their paper, but it is unnecessary (K. Sadakane, personal communication).

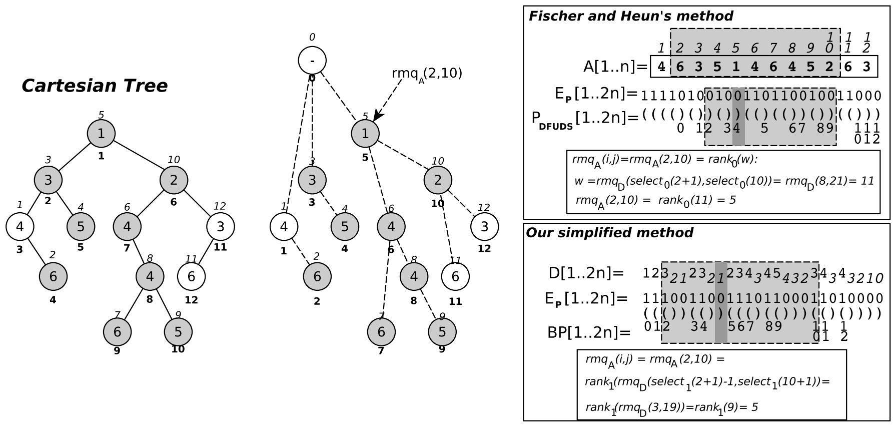

Fig. 3. The general tree derived from the example Cartesian tree. Note how inorder numbers of the binary Cartesian tree became preorder numbers in the general tree (we start preorders from 0 to help see the mapping). On the right, the formulas used by Fischer and Heun based on DFUDS (on the top) and the one proposed in this paper, based on BP (on the bottom). To reuse the same isomorphism of Fischer and Heun, we illustrate the variant of our formula that sets the nodes in the leftmost path of the tree as the children of the root.

in his case the tree had exactly one leaf per entry in A, so he only needed to find the ith leaf, and this could be done by extending rank/select operations to find 10s (BP) or 00s (DFUDS) in E. In the general case, one could add artificial leaves to every node, but this would increase the space to $4n + o(n)$ bits.

Fischer and Heun [8] found a solution that used just $2n + o(n)$ bits, which also turned out to be asymptotically optimal. The idea is to use a known isomorphism (see, e.g., [17]) between binary trees of n nodes and general ordinal trees of $n + 1$ nodes: We create an extra root for the general tree, and its children are the nodes in the leftmost path of the binary tree. Recursively, the right subtree of each node x in the leftmost path is converted into a general tree, using x as its extra root. A key property of this transformation is that inorders in the binary tree become preorders (plus 1) in the general tree. As seen, we can easily map between nodes and their preorders in general trees. Fig. 3 continues our example.

However, the lca in the Cartesian tree (which is what we want) is not the same lca in the resulting general tree; some adjustments are necessary. Fischer and Heun chose to use DFUDS for their $\mathrm{rmq}_A(i, j)$ solution, where it turns out that the adjustments to use a general tree actually remove the need to compute parent, but add back the need to check for ancestorship:

$$\begin{aligned}
&w \leftarrow \mathrm{rmq}_D(\mathrm{select}_0(i+1), \mathrm{select}_0(j))\\
&\text{if } \mathrm{rank}_0(\mathrm{open}(w)) = i \text{ then return } i\\
&\text{else return } \mathrm{rank}_0(w)
\end{aligned} \tag{2}$$

The select_0 operations find the nodes with preorder i and $j - 1$ (recall there is an extra root with preorder 1), then w is the position of the closing parenthesis of the result. The next line verifies that x is not an ancestor of y, and the last line returns the corresponding preorder value. For this formula to be correct, it is necessary that rmq_D returns the position of the leftmost minimum. Fig. 3 (top left) shows a query.

Grossi and Ottaviano [13] replaced the ancestorship test by one that does not use the costly open operation:

$$\begin{aligned}
&w \leftarrow \mathrm{rmq}_D(\mathrm{select}_0(i+1), \mathrm{select}_0(j))\\
&\text{if } D[\mathrm{select}_0(i)+1] \le D[w-1] \text{ then return } i\\
&\text{else return } \mathrm{rank}_0(w)
\end{aligned} \tag{3}$$

where as explained we can compute $D[k] = 2 \cdot \mathrm{rank}_1(E, k) - k$.

3. A simplified implementation

The current implementations of rmq_A build on the DFUDS representation of the general tree derived from the Cartesian tree, and follow either the formula of Fischer and Heun [8] (Eq. (2), in *SDSL*), or that of Grossi and Ottaviano [13] (Eq. (3), in *Succinct*). We show that, if we use the BP representation instead of DFUDS, we obtain a simpler formula. Let us assume, as before, that rmq_D returns the leftmost minimum. Then, our conversion from the binary Cartesian tree into a general tree must go in the opposite direction: the children of the extra root are the nodes in the *rightmost* path of the binary tree, and so on recursively. With this representation, it turns out that a correct formula is

$$\mathsf{rmq}_A(i,j) \;=\; \mathsf{rank}_0(\mathsf{rmq}_D(\mathsf{select}_0(i), \mathsf{select}_0(j))) \tag{4}$$

where no checks for ancestorship are necessary. Now we prove this formula is correct.

Lemma 1. *On a rightmost-path general tree built from the Cartesian tree of A, Eq. (4) holds.*

Proof. On the rightmost-path representation, the binary tree node with inorder i becomes the general tree node with *postorder* i, which is easily seen by induction. The closing parentheses of nodes x and y, which have postorders i and j, are thus found with $p = \mathsf{select}_0(i)$ and $q = \mathsf{select}_0(j)$. Now let $z = \mathsf{lca}(x, y)$. Then, in the Cartesian tree, x descends from the left child of z, z_l, and y descends from the right child, z_r. In the general tree, z_l is the first child of z, whereas z_r is its next sibling. Therefore the closing parenthesis of z, at position r, is between p and q. Further, y descends from some sibling z' to the right of z. Between p and q, the minima in D occur at the closing parentheses of z and of its siblings to the right, up to (but not including) z'. Thus the leftmost of those positions is precisely r, where z closes. Finally, $\mathsf{rank}_0(r)$ is the postorder position of z, and the inorder position of the cell in A.

The formula also works if y descends from x in the Cartesian tree. Since $i < j$, the inorder of x is smaller than the inorder of y, and thus y can only descend from the right child of x. Then the first minima in $[p, q]$ is precisely p, the closing parenthesis of x, and thus $z = x$. $\square$

If we want to use the leftmost-path mapping, we need that rmq_D returns the rightmost minimum position in the range. In this case, it holds

$$\mathsf{rmq}_A(i,j) \;=\; \mathsf{rank}_1(\mathsf{rmq}_D(\mathsf{select}_1(i+1)-1, \mathsf{select}_1(j+1))).$$

In this case, we must subtract 1 from p (which is now the position where node x opens) to ensure that the rightmost minimum in $D[p-1, q]$ is actually $p-1$ when y descends from x. Fig. 3 (bottom right) shows a query.

3.1. Construction

This representation is easily built in a way similar to the DFUDS-based one [8]. Consider the version using the rightmost-path mapping (the other is similar). We will write the parentheses of $E[1, 2n]$ right-to-left, starting with a 0 (i.e., a closing parenthesis) at its end. We start with an empty stack S, and traverse $A[n]$ to $A[1]$. At the point where we are to process $A[i]$, the stack S maintains left-to-right minima in $A[i+1, n]$. To process $A[i]$, we pop from S all the elements $\geq A[i]$, prepending a 1 (i.e., an opening parenthesis) to E each time an element is popped, until S becomes empty or its top is $< A[i]$. Now we push $A[i]$ in S and prepend a 0 to E. This is continued until $A[1]$ is processed. Finally, we prepend as many 1s to E as necessary to complete $2n$ bits.

This process requires $O(n)$ time and its extra space for S is proportional to the height of the Cartesian tree of A. While this is usually negligible, the space can become $O(n)$ words in the worst case. Fischer and Heun [8, Sec. 5.2.2] reduce it to n bits in a way that we can use verbatim in our case.

4. Implementing Balanced Parentheses

The most successful implementation of Balanced Parentheses uses Range Min–Max Trees (rmM-trees) [19,1]. The BP sequence $E[1, 2n]$ is cut into blocks of length b. Each block then becomes a leaf of the rmM-tree, which stores several variables. To describe them, let us introduce the notion of *excess*, which is the number of 1s minus the number of 0s in a bit string up to certain position:

$$\mathsf{excess}(S,i) \;=\; \mathsf{rank}_1(S,i) - \mathsf{rank}_0(S,i) \;=\; 2\cdot \mathsf{rank}_1(S,i) - i,$$

where we note that, if $D[1, 2n]$ is the sequence of depths we have been using and $E[1, 2n]$ is the associated bit sequence, then $D[i] = \mathsf{excess}(E, i)$.

Then the relevant variables associated with each rmM-tree leaf representing bits $L[1, b]$ are $e = \mathsf{excess}(L, b)$ (the local excess produced by the leaf), and $m = \min_{1 \leq i \leq b} \mathsf{excess}(L, i)$ (the minimum left-to-right excess along the leaf). The rmM-tree is a perfect binary tree on those leaves, where the internal nodes store the same fields e and m with respect to the area they cover. That is, let v have left and right children v_l and v_r, respectively, then $v.e = v_l.e + v_r.e$ and $v.m = \min(v_l.m, v_l.e + v_r.m)$.

We can then compute any operation $\mathsf{rmq}_D(p, q)$ as follows. First, we determine the maximal block-aligned range $[p', q']$ inside $[p, q]$. Then we scan the range $[p, p'-1]$ sequentially, obtaining the minimum excess *min* and its excess $exc = \mathsf{excess}(p, p'-1)$. Then, if $[p', q']$ is not empty, we start at the rmM-tree leaf v started by position p'. We set $min \leftarrow (min, exc + v.m)$ and update $exc \leftarrow exc + v.e$. Now we start climbing up the path from v. If v is a right child of its parent, we just move to its parent. Otherwise, we see if its right sibling v' is contained in $[p', q']$. If it is, we process it (setting $min \leftarrow \min(min, exc + v'.m)$ and $exc \leftarrow exc + v'.e$) and then go to the parent of v. If, instead, v' is not contained in $[p', q']$,

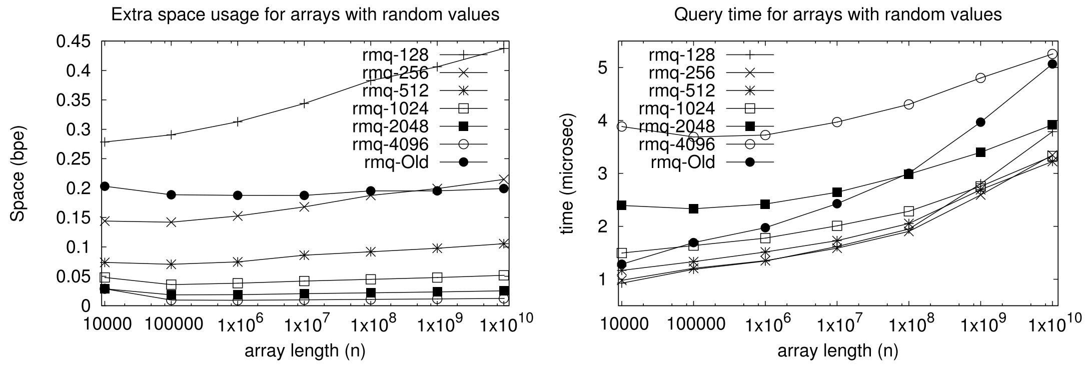

Fig. 4. Query space and time on random arrays, for ranges of size 10,000, comparing the standard with our new implementations. Space is measured in bits per element (bpe).

we switch to $v \leftarrow v'$ and start the descent: Let v_l and v_r be the left and right children of v, respectively. Then, if v_l is contained in $[p', q']$ we process v_l as before and descend to v_r, otherwise we descend to v_l. At the end, we reach the leaf of position $q' + 1$, which is traversed sequentially up to position q to complete the process.

Once the minimum value is clear, we must find its leftmost occurrence in $D[p, q]$. If it occurred in $[p, p' - 1]$, or occurred only in $[q' + 1, q]$, then we already know its position. Otherwise, its leftmost occurrence is in some rmM-tree node v we know. We then move down from v to find its position: if $v_l.m \le v_l.e + v_r.m$, we descend to v_l, otherwise to v_r. We finally reach a leaf and scan it to find the position of the leftmost minimum.

By setting $b = \Theta(\log^2 n)$ and using precomputed tables to process the leaves by chunks of $(\log n)/2$ bits, the total time is $O(\log n)$ and the extra space of the rmM-tree and precomputed tables is $O(n/\log n) = o(n)$.

Operations rank and select can be solved similarly, the former by computing $exc = \mathsf{excess}(E, i)$ and then using $\mathsf{rank}_1(i) = (exc + i)/2$ or $\mathsf{rank}_0(i) = (i - exc)/2$. For $\mathsf{select}_1(j)$ we move down from the rmM-tree root looking for the position i where $\mathsf{excess}(E, i) = 2 \cdot j - i$, and for $\mathsf{select}_0(j)$ we aim to $\mathsf{excess}(E, i) = i - 2 \cdot j$.

Our implementation carries out the following optimizations:

1. Instead of the fields e in all the nodes, we store an array $exc[1, 2n/b]$ with $exc[i] = \mathsf{excess}(E, b \cdot i)$, using as many bits as necessary (in many cases, the maximum excess is not large). Further, when b is even, those sampled excesses are also even, so we save one further bit. To solve rank, we use the table exc to find the rank up to the previous sampled position, and process the last block sequentially.
2. To solve $\mathsf{select}(j)$, we store a table giving the blocks where the answer to every bth value of j falls, using as many bits as necessary. We then compute rank up to that block and sequentially scan from its beginning until reaching the desired rank j. In the conference version [7] we just used binary search on table exc, which saved little space but was considerably slower.
3. To solve rmq_D, we avoid scanning the last area $[q' + 1, q]$ if its block minimum is not smaller than our current minimum value min. Note that its block minimum may be smaller than the minimum in $[q' + 1, q]$, but not larger. In the conference version we stored the position of the minimum to avoid the descent, but this turns out to take too much extra space for a small saving in time.
4. The precomputed tables process bytes of the leaf, so they are very small and usually fit in cache, and we read aligned data.

5. Experimental results

5.1. Tuning and older versions

Our first experiment compares our improved implementation with the standard one, which was used in our conference version [7] with block size $b = 256$. We show various block sizes for our new version, so as to choose a good representative. The data are arrays A of sizes from $n = 10^4$ to $n = 10^{10}$, with randomly chosen ranges $[i, j]$ of fixed length 10,000. Fig. 4 shows the results, where “rmq-Old” stands for the standard implementation and “rmq-b” for the new ones. The space shown is in addition to the $2n$ bits used by the parentheses.

It can be seen that our new implementation is far more efficient, in space and especially in time. For the rest of the experiments, we will choose $b = 512$ as a compromise value between space and time.

We compare our implementation with those in *SDSL* and *Succinct*, which are based on DFUDS (Eqs. (2) and (3), respectively). As a control, we also implement ourselves the DFUDS-based solution of Eq. (2) using rmM-trees and our rank/select components; this is called *DFUDS* in our charts.

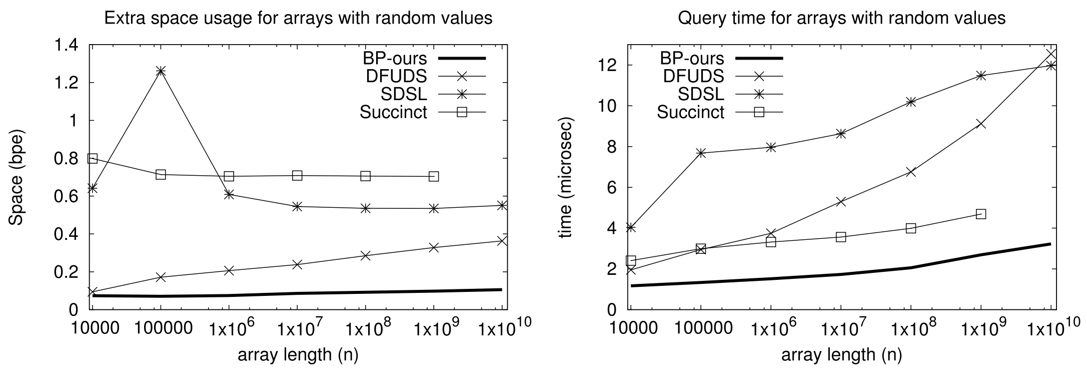

Fig. 5. Query space and time on random arrays, for ranges of size 10,000.

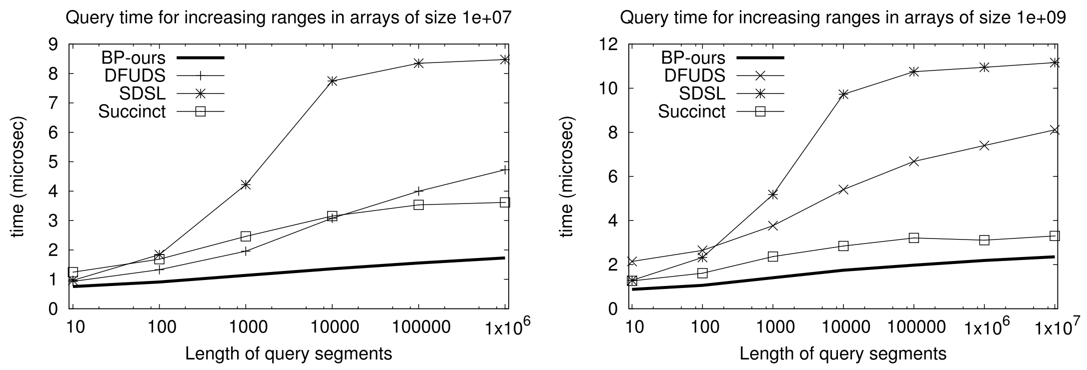

Fig. 6. Query time on random arrays, for ranges of increasing size and two values of n.

5.2. Arrays with random values

We first compare the four implementations on the same randomly generated arrays A of the previous experiment. Fig. 5 shows the results (*Succinct* did not build on the largest arrays). Our implementation uses always below 2.1 bits per element (bpe), that is, 0.1 on top of the 2 bpe needed by the BP (or DFUDS) representation. Our DFUDS implementation, instead, increases the space because the average excess grows with n in this format, and thus the rmM-tree counters need more bits. The implementations in *SDSL* and *Succinct* use at least 2.6–2.8 bpe.

Our solution is also the fastest, taking 1–3 microseconds (μs) per query as n grows. It is followed by *Succinct* and, far away, by *SDSL*. Our *DFUDS* implementation is fast for short arrays, but it becomes slower when n grows. This is probably because operation open matches a farther parenthesis as n grows; the same effect can be seen in *SDSL*. In *Succinct*, instead, operation open is avoided, and thus the growth is much milder. In our BP-based implementation, the growth with n is also mild, owing only to traversing a higher rmM-tree.

Fig. 6 shows how the times are affected by the size of the query range. As it can be seen, our implementation and *Succinct* show a very slow increase, whereas times grow much faster in *SDSL* and *DFUDS*. This may be due to the open operation, whose time grows in practice with the distance to its parent. Larger intervals return nodes closer to the root, whose former siblings are larger, and so is the distance to the parent in DFUDS.

5.3. Arrays with broadly increasing or decreasing values

Our final experiment measures the effect of the order in A on the space and time of the structures. Given a parameter Δ, our entry $A[i]$ is chosen at random in $[i-\Delta, i+\Delta]$, or in $[n-i-\Delta, n-i+\Delta]$, thus the smaller Δ, the more sorted is A in increasing/decreasing order. Fig. 7 shows the results.

Our implementation maps the leftmost path of the Cartesian tree to the children of the general tree. As a result, the structure takes slightly more space and time when the array is more sharply increasing, because the general tree is deeper and the rmM-tree stores larger values. Instead, it does not change much when A is decreasing (one could use one mapping or the other as desired, since we know A at construction time, thus never using more than 2.1 bpe). *DFUDS* shows the opposite effect, because the DFUDS excesses are smaller when the tree is deeper. The effect is more pronounced than in our structure, and it also affects the time performance. It is not clear how can one use the rightmost-path mapping in the case of DFUDS, however, as it is not symmetric (we can reverse the array if we do not mind returning the rightmost position

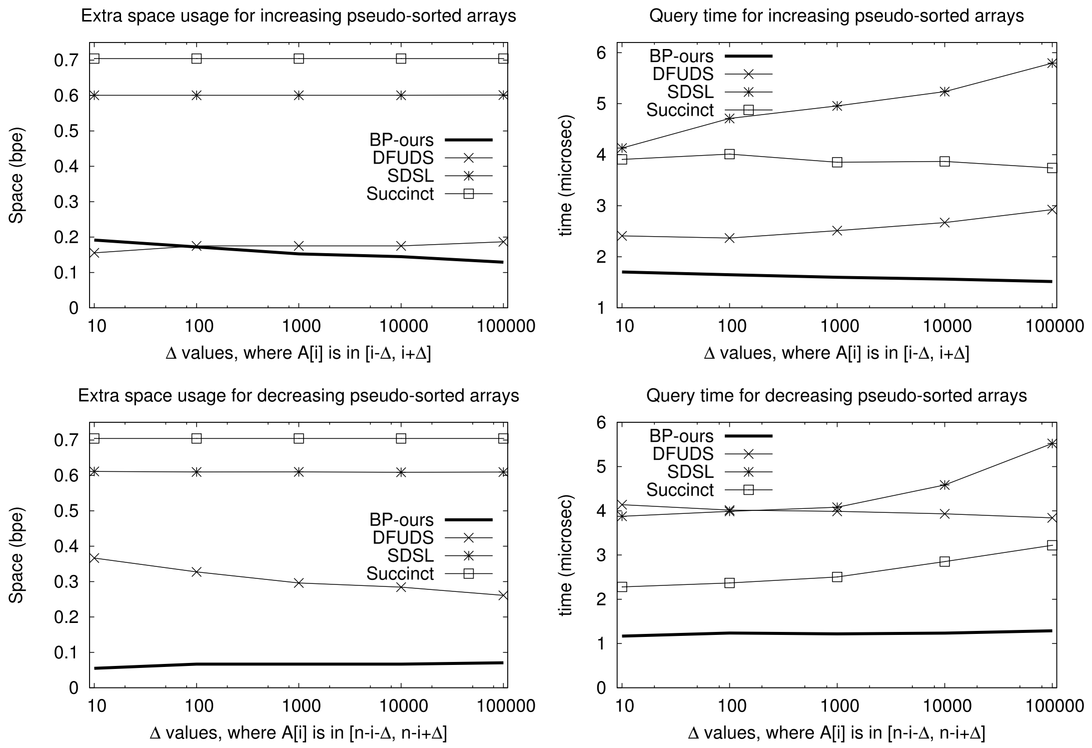

Fig. 7. Query time on pseudo-sorted arrays, $n = 10^6$ and ranges of size 10,000.

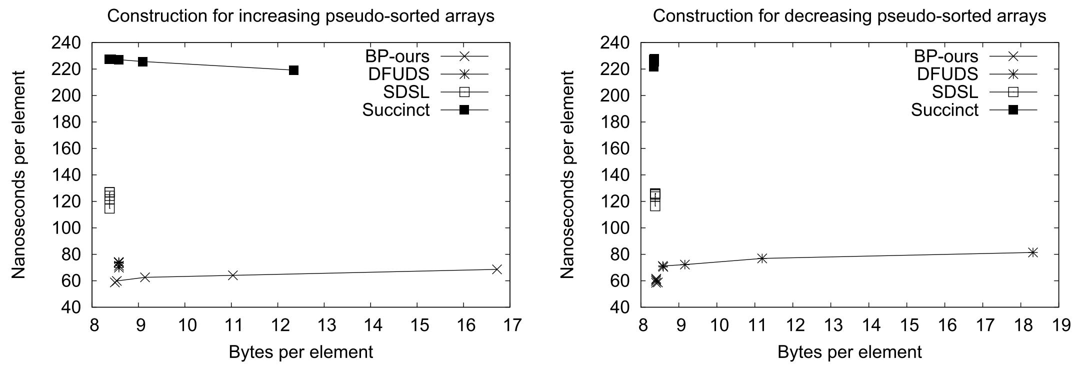

Fig. 8. Construction time and space on random and pseudo-sorted arrays, $n = 10^9$.

of the minimum). The space of *SDSL* and *Succinct* is not affected at all by the lack of randomness, but *SDSL* turns out to be faster on less random arrays, regardless of whether they are increasing or decreasing. *Succinct* performs better when the values tend to be decreasing and worse when they are increasing. Our times are, just like the space, negatively affected by increasing values, but still they are much better than the others and, as said, we can choose to map the rightmost path in this case.

5.4. Construction times

Fig. 8 shows the space and time for construction with the different implementations, for the same range of values of Δ as above. The array A uses 64-bit integers, so the minimum space usage is 8 bytes per element. The constructions use a simple stack, except *SDSL*, which uses $O(n)$ bits. When the elements are random, the size of the stack is insignificant (below 4 bits per element), and thus the higher time (110–130 versus 60–70 nanoseconds per element) required to build using $O(n)$ bits does not pay off. Instead, when the values are mostly increasing (or decreasing, depending on whether one uses the leftmost or the rightmost path), the construction space grows significantly. Our *BP* and *DFUDS* construction use similar time and space, except for the leftmost/rightmost difference. *Succinct* is the slowest at construction (around 230 nanoseconds), but it uses less space than our stack-based constructions when the array values are skewed.

Note, again, that our implementation performs better on random and decreasing arrays than on increasing ones in all aspects: construction space, final space, and query times. *DFUDS* is the opposite. The difference is that, while our formula can be reversed and thus we can switch to using rightmost instead of leftmost paths to handle increasing arrays, the formula of *DFUDS* appears to be fixed and thus it must perform worse on decreasing arrays.

6. Conclusions

We have presented an alternative design to Fischer and Heun's RMQ solution that uses $2n + o(n)$ bits and constant time [8]. Our implementation uses $2.1n$ bits and takes 1–3 microseconds per query. This is noticeably smaller and faster than the current implementations in libraries *SDSL* and *Succinct*, which follow Fischer and Heun's design. By using BP instead of DFUDS succinct tree representation, our RMQ formula simplifies considerably, and besides we performed some optimizations to the BP implementation. We have left our implementation publicly available at `https://github.com/hferrada/rmq.git`, and our DFUDS-based one at `https://github.com/hferrada/rmqFischerDFUDS.git`.

Any ± 1-RMQ implementation can be used together with our new formula. Our current implementation of ± 1-RMQs is not formally constant time, as it builds on rmM-trees [19,1]. Although truly constant-time solutions are not promising in practice [20,19], and we have shown that the time of rmM-trees grows very slowly with n, it would be interesting to devise a practical and constant-time solution.

Acknowledgements

We thank the reviewers for their helpful comments.

References

[1] D. Arroyuelo, R. Cánovas, G. Navarro, K. Sadakane, Succinct trees in practice, in: Proc. 12th ALENEX, 2010, pp. 84–97.
[2] M. Bender, M. Farach-Colton, The LCA problem revisited, in: Proc. 4th LATIN, in: Lect. Notes Comput. Sci., vol. 1776, 2000, pp. 88–94.
[3] D. Benoit, E.D. Demaine, J.I. Munro, R. Raman, V. Raman, S.S. Rao, Representing trees of higher degree, Algorithmica 43 (4) (2005) 275–292.
[4] O. Berkman, U. Vishkin, Recursive star-tree parallel data structure, SIAM J. Comput. 22 (2) (1993) 221–242.
[5] G. Chen, S.J. Puglisi, W.F. Smyth, Lempel–Ziv factorization using less time & space, Math. Comput. Sci. 1 (2008) 605–623.
[6] D. Clark, Compact PAT trees, PhD thesis, Univ. of Waterloo, Canada, 1996.
[7] H. Ferrada, G. Navarro, Improved range minimum queries, in: Proc. 26th DCC, 2016, pp. 516–525.
[8] J. Fischer, V. Heun, Space-efficient preprocessing schemes for range minimum queries on static arrays, SIAM J. Comput. 40 (2) (2011) 465–492.
[9] J. Fischer, V. Mäkinen, G. Navarro, Faster entropy-bounded compressed suffix trees, Theor. Comput. Sci. 410 (51) (2009) 5354–5364.
[10] H.N. Gabow, J.L. Bentley, R.E. Tarjan, Scaling and related techniques for geometry problems, in: Proc. 16th STOC, 1984, pp. 135–143.
[11] R.F. Geary, N. Rahman, R. Raman, V. Raman, A simple optimal representation for balanced parentheses, Theor. Comput. Sci. 368 (3) (2006) 231–246.
[12] S. Gog, T. Beller, A. Moffat, M. Petri, From theory to practice: plug and play with succinct data structures, in: Proc. 13th International Symposium on Experimental Algorithms (SEA), in: Lect. Notes Comput. Sci., vol. 8504, 2014, pp. 326–337.
[13] R. Grossi, G. Ottaviano, Design of practical succinct data structures for large data collections, in: Proc. 12th SEA, in: Lect. Notes Comput. Sci., vol. 7933, 2013, pp. 5–17.
[14] G. Jacobson, Space-efficient static trees and graphs, in: Proc. 30th FOCS, 1989, pp. 549–554.
[15] J. Jansson, K. Sadakane, W.-K. Sung, Ultra-succinct representation of ordered trees with applications, J. Comput. Syst. Sci. 78 (2) (2012) 619–631.
[16] R. Konow, G. Navarro, Faster compact top-k document retrieval, in: Proc. 23rd DCC, 2013, pp. 351–360.
[17] J.I. Munro, V. Raman, Succinct representation of balanced parentheses and static trees, SIAM J. Comput. 31 (3) (2001) 762–776.
[18] G. Navarro, Y. Nekrich, L.M.S. Russo, Space-efficient data-analysis queries on grids, Theor. Comput. Sci. 482 (2013) 60–72.
[19] G. Navarro, K. Sadakane, Fully-functional static and dynamic succinct trees, ACM Trans. Algorithms 10 (3) (2014) 16.
[20] K. Sadakane, Compressed suffix trees with full functionality, Theory Comput. Syst. 41 (4) (2007) 589–607.
[21] K. Sadakane, Succinct data structures for flexible text retrieval systems, J. Discret. Algorithms 5 (2007) 12–22.
[22] J. Vuillemin, A unifying look at data structures, Commun. ACM 23 (4) (1980) 229–239.